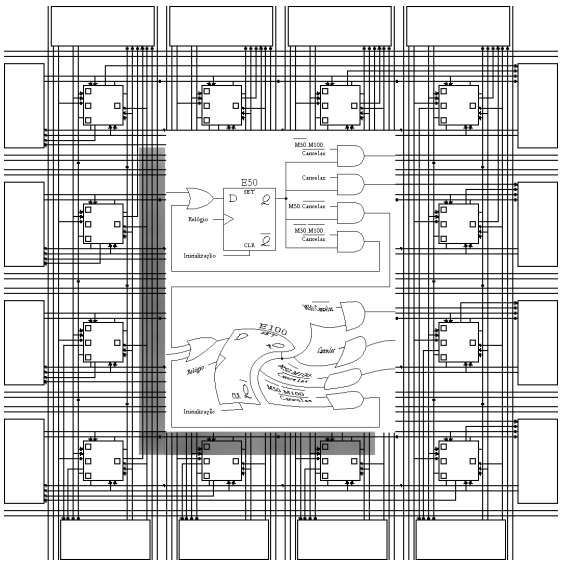
Universidade de Aveiro
Departamento de Electrónica e Telecomunicações

2000

Arnaldo Silva Rodrigues de Oliveira

Modelos, Métodos e Ferramentas para Implementação de Unidades de Controlo Virtuais



Universidade de Aveiro
Departamento de Electrónica e Telecomunicações

2000

Arnaldo Silva Rodrigues de Oliveira

Modelos, Métodos e Ferramentas para Implementação de Unidades de Controlo Virtuais

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Electrónica e de Telecomunicações, realizada sob orientação científica do Dr. Valery Sklyarov, Professor Catedrático Visitante do Departamento de Electrónica e de Telecomunicações da Universidade de Aveiro

O Júri

Presidente Prof. Dr. António Manuel de Brito Ferrari de Almeida
Professor Catedrático da Universidade de Aveiro

Prof. Dr. João Paulo Cacho Teixeira
Professor Associado com Agregação do Instituto Superior Técnico da Universidade Técnica de Lisboa

Prof. Dr. Valery Anatolevitch Sklyarov
Professor Catedrático Visitante da Universidade de Aveiro

Agradecimentos

Em primeiro lugar, gostaria de agradecer ao meu orientador, o Prof. Dr. Valery Sklyarov, pelo facto de me ter ingressado na área dos sistemas reconfiguráveis, por ter contribuído para o aprofundamento dos meus conhecimentos sobre o projecto de circuitos sequenciais e por ter proposto e orientado este trabalho, demonstrando sempre a sua disponibilidade e acompanhamento. Agradeço também ao Prof. Dr. António Ferrari por todos os comentários e sugestões que contribuíram para o aperfeiçoamento desta dissertação.

Reconheço também o apoio de todos os membros do Laboratório de Sistemas Computacionais do Departamento de Electrónica e Telecomunicações. Em particular, gostaria de agradecer à Engª. Andreia Melo, ao Eng. Artur Pereira e ao Doutor António Adrego da Rocha por todas as discussões e sugestões durante a realização deste trabalho, as quais se revelaram extremamente úteis.

Do ponto de vista financeiro, gostaria de agradecer à Fundação para a Ciência e a Tecnologia, a qual suportou este trabalho através da Bolsa de Mestrado – Apoio à Dissertação com a referência PRAXIS XXI/BM/17678/98. A apresentação deste trabalho em algumas conferências e *workshops* internacionais foi possível graças ao apoio da Fundação Calouste Gulbenkian, do IEETA – Instituto de Engenharia Electrónica e Telemática de Aveiro e da rede BELSIGN – *Behavioral Design Methodologies for Digital Systems* do programa da Comunidade Europeia HCM – *Human Capital Mobility* sob o contrato CHRX-CT94-0459.

Por último, mas de uma forma muito especial, agradeço à minha família todo o apoio que me deram, em particular, aos meus pais Maria José e Arnaldo, às minhas irmãs Maria Helena e Cristiana e finalmente à Andreia. A eles dedico este trabalho.

Resumo

As unidades de controlo são um dos tipos de circuitos digitais mais importantes, estando presentes numa grande diversidade de aplicações, desde simples controladores de semáforos até sistemas complexos de processamento de dados. A sua função é estabelecer a sequência de operações realizadas pelo sistema a que pertencem. Dependendo da aplicação, podem ser utilizadas isoladamente ou em conjunto com outros componentes, tais como unidades de execução, sensores e actuadores. Como um caso particular de circuitos sequenciais, são normalmente descritas por modelos orientados ao estado, dos quais a máquina de estados finitos é o exemplo mais conhecido. No entanto, com a crescente complexidade dos sistemas e consequentemente das suas unidades de controlo, este modelo deixou de ser adequado para realizar a sua especificação, uma vez que não suporta a descrição explícita de hierarquia e concorrência. Nesta dissertação são abordados alguns dos modelos e linguagens mais apropriadas para este fim, em particular a máquina de estados finitos hierárquica e/ou paralela, os esquemas de grafos hierárquicos e/ou paralelos e os *Statecharts*. O aparecimento na última década de dispositivos lógicos de elevada capacidade e programáveis pelo utilizador foi responsável por importantes alterações no projecto de sistemas digitais, principalmente ao nível do tempo, custo e flexibilidade de projecto. Além disso, os dispositivos programáveis dinamicamente, deram origem a uma nova classe de circuitos: os sistemas reconfiguráveis. Estes podem ser usados para construir sistemas computacionais modificáveis, pelo que permitem combinar as vantagens de uma solução programável com o elevado desempenho de uma implementação em hardware. Os sistemas reconfiguráveis podem também ser utilizados em aplicações onde a quantidade de recursos de hardware necessários para uma implementação integral seja elevada e onde nem todos os sub-sistemas sejam necessários em simultâneo, sendo portanto possível e até desejável uma implementação parcial em conjunto com a sua reconfiguração dinâmica. Neste caso, devido à analogia com os sistemas de memória virtual, os sistemas reconfiguráveis são também designados por sistemas de hardware virtual e os respectivos circuitos de controlo por unidades de controlo virtuais. Como as unidades de controlo são específicas de cada projecto e normalmente bastante irregulares, torna-se necessário o estabelecimento de algumas restrições de forma a simplificar o seu projecto e reconfiguração. Nesta dissertação é proposta uma arquitectura de unidades de controlo virtuais baseada numa estrutura predefinida, parametrizável e optimizada para o dispositivo de implementação utilizado, a FPGA XC6200 da Xilinx. Esta arquitectura em conjunto com os modelos e os dispositivos lógicos programáveis utilizados permite construir unidades de controlo complexas, flexíveis, extensíveis e reutilizáveis. O processo de síntese de unidades de controlo é também abordado, com uma atenção especial para as técnicas mais apropriadas para as FPGAs. Finalmente, para suportar a reconfiguração dinâmica dos circuitos desenvolvidos, foi construída uma biblioteca de classes em C++ e um controlador (*device driver*) para a placa de desenvolvimento utilizada neste trabalho.

Abstract

Control units are one of the most important types of digital circuits. They are used in a great variety of applications, from simple traffic light controllers to complex data processing systems. Their function is to establish the sequence of operations accomplished by the system they belong to. Depending on the application, they can be used separately or together with other components, such as execution units, sensors or actuators. Because control units are a particular kind of sequential circuits, they are usually described by state-oriented models, the finite state machine being the best known example. However, with the increasing complexity of the systems and consequently of its control units, this model is becoming less adapted to perform its specification, because it does not support the explicit description of hierarchy and concurrency. This dissertation presents some of the models and languages better suited to this purpose, in particular the hierarchical and/or parallel finite state machine, the hierarchical and/or parallel graph schemes and the Statecharts formalisms.

The appearance in the last decade of high capacity user programmable logic devices was responsible for important modifications in the way digital systems are designed, mainly in terms of time, cost and design flexibility. Besides, the availability of dynamically reconfigurable devices made possible the emergence of a new class of circuits: the reconfigurable systems. They can be used to build modifiable computational systems combining the advantages of a programmable solution with the high performance of a hardware implementation. The reconfigurable systems can also be used in applications where the amount of required hardware resources for an integral implementation is too high or at least bigger than desirable and where all the subsystems are not necessary simultaneously, making possible a partial implementation in conjunction with its dynamic reconfiguration. Due to the analogy with virtual memory systems, these systems can also be called virtual hardware systems and their respective control circuits virtual control units. Because the control units are specific for each project and usually very irregular, it is necessary to impose some constraints in order to simplify their design and reconfiguration. In this dissertation an architecture for virtual control units based on a predefined, parameterizable and optimized template for a particular target device, the FPGA XC6200 from Xilinx, is proposed. However, the main ideas of this architecture can also be easily applied to other FPGA families. The use of this architecture together with the hierarchical and/or parallel models and the dynamically reconfigurable logic devices allows building complex, flexible, extensible and reusable control units. This contributes to a decrease in development time and permits system updates after the completion of the design and manufacturing cycles. The control unit synthesis process is also presented with special emphasis on the techniques better suited to FPGA implementation. Finally, to support the dynamic reconfiguration of the developed circuits, a C++ class library and a device driver for the development system used in this work were built.

Índice

Lista de Figuras

Lista de Tabelas

1 Introdução

Sumário

Este capítulo introduz alguns dos conceitos fundamentais que servem de base aos restantes capítulos desta dissertação. Mais concretamente, são abordados o processo de projecto de um sistema digital e os conceitos de sistema computacional e de unidade de controlo.

No contexto do projecto de um sistema digital, são introduzidas as noções de modelo e de síntese de um modelo como um processo de refinamento do mesmo. Com o objectivo de uniformizar as várias formas, que normalmente aparecem na literatura, de classificação dos modelos e das transformações que sobre eles podem ser realizadas, é também apresentada uma proposta de taxonomia de modelos de circuitos electrónicos em geral. Esta taxonomia é baseada numa relação ortogonal entre os tipos de representação de um modelo, normalmente designados por vistas e os níveis de granulosidade dos componentes utilizados nesse modelo, vulgarmente chamados níveis de abstracção.

Neste capítulo são também descritas as fases de projecto de sistemas digitais e as ferramentas de desenvolvimento assistido por computador que as realizam. Seguidamente, são apresentadas as duas metodologias de projecto mais usadas no desenvolvimento de sistemas digitais, a de captura e simulação e a de descrição e síntese.

Os sistemas computacionais, como sistemas que realizam processamento de dados, são normalmente constituídos por dois tipos de módulos principais: unidade de execução e unidade de controlo. A unidade de execução realiza as operações de processamento propriamente dito, enquanto a unidade de controlo assegura a sequência correcta de operações da unidade execução.

Nas duas últimas secções deste capítulo são apresentados respectivamente os objectivos deste trabalho e a organização da dissertação.

1.1 Projecto de Sistemas Digitais

De uma forma geral, o projecto de um produto ou sistema pode ser definido como uma sequência de passos desde a sua idealização até à elaboração de planos que descrevem de forma detalhada a sua construção. Neste processo podem estar envolvidas diversas pessoas, cada uma com uma função específica, desde a concepção até à fabricação, passando pelo desenvolvimento e teste.

O processo de projecto é altamente influenciado por diversos factores, entre eles, o tipo de produto, o tempo de desenvolvimento pretendido, as ferramentas de projecto utilizadas e as tecnologias empregues na sua fabricação [Gajski97]. Independentemente destes factores, existe um conjunto de etapas que de uma forma ou de outra estão geralmente presentes no projecto de um produto, como por exemplo, a especificação, a simulação, a síntese, a verificação, a implementação e o teste. Estas etapas serão apresentadas de forma resumida na secção 1.1.3. Antes porém, vamos rever outras noções fundamentais no projecto de sistemas digitais, nomeadamente a de modelo, a de tipo de representação, a de nível de granulosidade, a de nível de abstracção e por último, as de abordagem estruturada e de metodologia de projecto.

Por conveniência, no processo de projecto são normalmente utilizados modelos. Um modelo de um sistema é uma abstracção, ou seja uma representação que mostra as características relevantes sem os detalhes associados [Micheli94]. A síntese é a geração de um modelo a partir de outro menos detalhado, sendo portanto um processo de refinamento. Por outro lado, a análise é a operação inversa da síntese, ou seja, a obtenção de um modelo menos detalhado a partir de outro mais detalhado. Mais à frente serão dadas definições mais precisas de síntese e de análise.

A elaboração de um modelo, ou modelação de um circuito ou sistema tem duas finalidades. Primeiro, o desenvolvimento de um modelo ajuda o projectista a formalizar uma solução. Segundo, um modelo de um circuito pode ser processado por computador para procurar erros de projecto, realizar a sua simulação e prever as suas características temporais. Adicionalmente, existem várias ferramentas de projecto assistido por computador (*Computer Aided Design – CAD*) que realizam automaticamente todos ou alguns dos passos da síntese, incluindo optimização do projecto e transformação do projecto de uma forma abstracta numa realização física.

Na definição, projecto e fabricação de um produto, cada pessoa envolvida debruça-se sobre um aspecto diferente do mesmo, necessitando de informação específica para realizar o seu trabalho. Por este motivo, os vários modelos do sistema utilizados ao longo do seu projecto diferem no tipo de informação que se pretende realçar. Os modelos podem ser classificados em termos de tipos de representação e níveis de granulosidade. Em conjunto, estas duas classificações definem o nível de abstracção de um modelo.

Aqui vão ser considerados três tipos de representação: comportamental, estrutural e física. Quanto aos níveis de granulosidade, a sua classificação pode ser feita em cinco níveis distintos: geométrico, circuito, lógico, arquitectural e sistema. Esta taxonomia dos modelos é uma tentativa para uniformizar as diversas classificações publicadas por diversos autores [Gasjki97, Micheli94, NelNagCarIrw95], que apesar de

não possuírem designações ou definições contraditórias, não são capazes de caracterizar um modelo e as respectivas transformações de forma completa.

Vamos agora detalhar um pouco mais cada um destes assuntos, começando pelos tipos de representação.

1.1.1 Tipos de Representação

Os três tipos mais comuns de representação de um modelo são a comportamental, a estrutural e a física [Gajski97].

Numa representação comportamental ou funcional, o sistema é visto como uma caixa preta sendo o aspecto mais importante a especificação do seu comportamento em função das entradas e do tempo. Por outras palavras, uma representação comportamental descreve a funcionalidade do sistema sem quaisquer detalhes sobre a sua implementação, definindo a sua resposta a qualquer combinação dos valores de entrada mas sem descrever a forma como deve ser projectado ou construído utilizando um dado conjunto de componentes. Esta é em geral a representação do primeiro modelo elaborado para o sistema a projectar, permitindo manipular as características essenciais do projecto, sem entrar em pormenores de implementação, irrelevantes numa fase inicial. Ao mesmo tempo, proporciona uma descrição de fácil leitura para o desenvolvimento, documentação e manutenção do projecto.

Por outro lado, numa representação estrutural, o sistema é definido como um conjunto de componentes e suas interligações. Ao contrário da representação comportamental, descreve a constituição do sistema sem uma referência explicita à sua funcionalidade. Contudo, é possível nalguns casos derivar a funcionalidade a partir da estrutura. No entanto, este pode ser um processo complexo e sujeito a erros especialmente quando o número de componentes é elevado.

Finalmente, uma representação física é aquela que descreve as características físicas do sistema, definindo as dimensões e posição de cada componente da representação estrutural. De notar que enquanto a representação estrutural descreve as conexões entre os diversos componentes do sistema, somente a representação física descreve com precisão as suas relações espaciais. Por outras palavras, a representação física é utilizada para descrever a forma final do sistema após a sua fabricação, especificando características como o seu tamanho, peso, consumo e dissipação de potência, localização das entradas e saídas, etc.

1.1.2 Níveis de Granulosidade

No projecto de sistemas electrónicos, cada um dos tipos de representação atrás descritos (comportamental, estrutural e físico) pode ser usado a diferentes níveis de granulosidade consoante o tipo de objectos utilizados no modelo. A generalidade dos autores [Micheli94, Gasjki97, NelNagCarIrw95] utiliza a designação de nível de abstracção. No entanto, a primeira parece ser mais correcta, uma vez que na passagem entre as representações física, estrutural e comportamental, existe também um procedimento de abstracção dos detalhes de implementação.

Em geral, podem ser identificados cinco níveis de granulosidade: geométrico, circuito, lógico, arquitectural e sistema. Os objectos que melhor caracterizam cada um destes níveis de granulosidade são respectivamente os materiais semicondutores, os

transístores, as portas lógicas, os registos e os processadores. O nível de granulosidade de um modelo é estabelecido pelo objecto de maior complexidade nele existente. A relação entre estes níveis de granulosidade e os tipos de representação está resumida na Tabela 1.1. Para o nível lógico, a sua leitura pode ser feita da seguinte maneira: as unidades funcionais e de armazenamento são exemplos de componentes do nível lógico, compostos por portas lógicas e flip-flops e cuja funcionalidade é descrita por expressões Booleanas.

Os componentes relativos às representações físicas de cada nível de granulosidade são utilizados como blocos predefinidos no nível seguinte.

Ao nível geométrico, os componentes electrónicos elementares são constituídos por entidades geométricas formadas por materiais semicondutores, condutores e isoladores, que quando agrupadas de acordo com uma estrutura bem definida apresentam as propriedades eléctricas dos componentes electrónicos elementares que são utilizados no nível de abstracção seguinte. Do ponto de vista comportamental, cada um destes componentes é representado por um conjunto de equações diferenciais ou relações tensão-corrente que descrevem a sua operação nas respectivas zonas de funcionamento.

Tal como indicado na Tabela 1.1, os componentes principais do nível do circuito são os transístores, as resistências e os condensadores. Estes podem ser combinados para construir circuitos analógicos e digitais com uma dada funcionalidade. À semelhança do caso anterior, esta funcionalidade é normalmente descrita por um conjunto de equações diferenciais ou por algum tipo de relações tensão-corrente, o que é natural, uma vez que ainda nos encontramos no domínio analógico. A representação física destes circuitos, designada por célula, consiste em componentes elementares e nos condutores que os interligam.

Na Tabela 1.1 pode-se também observar que os componentes principais do nível lógico são as portas lógicas e flip-flops. Ambos são exemplos de células digitais, com entradas e saídas bem definidas ao longo das suas extremidades. Estas células podem ser agrupadas para formar módulos ou unidades de armazenamento ou funcionais que são usadas como componentes básicos no nível arquitectural. Estas unidades são descritas do ponto de vista comportamental por equações Booleanas e diagramas de máquinas de estados finitos.

Tal como indicado na Tabela 1.1, ao nível arquitectural, um circuito digital é constituído por unidades funcionais e de armazenamento, tais como, registos, somadores, comparadores, multiplicadores, contadores, filas, pilhas e unidades de execução e respectivas interligações. Cada um destes componentes é um módulo com dimensões, tempo de propagação e posição das entradas e saídas bem definidas. Estes componentes podem ser agrupados e interligados em circuitos integrados ou macro blocos, os quais são utilizados como componentes básicos no nível de abstracção seguinte. Em geral, estes circuitos são descritos por algoritmos, fluxogramas, conjuntos de instruções e máquinas de estado algorítmicas. Este nível é também vulgarmente designado por nível de transferência entre registos (*Register Transfer Level - RTL* [ErcLanMor99, Gajski97]), uma vez que durante o processamento, a informação é transferida entre registos, passando eventualmente pelas unidades funcionais.

	Tipo de representação		
Nível de granulosidade	**Comportamental**	**Estrutural**	**Física**
Geométrico (Semicondutor)	Equações diferenciais, relações tensão-corrente	Materiais semicondutores, condutores e isoladores	Componentes elementares
Circuito (Transístor)	Equações diferenciais, relações tensão-corrente	Transístores, resistências, condensadores	Células analógicas e digitais
Lógico (Porta Lógica)	Equações Booleanas, máquinas de estados finitos	Portas lógicas, flip-flops	Módulos, unidades funcionais e de armazenamento, circuitos integrados SSI-MSI
Arquitectural (Registo)	Algoritmos, fluxogramas, conjuntos de instruções, máquinas de estados finitos generalizadas	Registos, somadores, comparadores, contadores, filas, pilhas, unidades de execução	Circuitos integrados LSI, macro blocos para circuitos VLSI
Sistema (Processador)	Especificações executáveis, programas	Processadores, controladores, memórias, ASICs	Placas de circuito impresso, circuitos integrados VLSI

Tabela 1.1 – Representações de um projecto e níveis de abstracção.[1]

O nível de granulosidade mais elevado apresentado na Tabela 1.1 é o de sistema. Um sistema é constituído por componentes relativamente complexos tais como, processadores, memória, controladores de periféricos e interfaces, para além de outros circuitos específicos da aplicação. Geralmente, são colocados numa placa de circuito impresso um ou mais componentes deste tipo e interligados por condutores gravados na mesma. Actualmente, devido às elevadas capacidade de integração disponíveis é possível implementar no mesmo circuito integrado um elevado número de componentes ou circuitos, podendo estes estar implementados na mesma pastilha ou em pastilhas diferentes com um substracto comum. A integração possui vantagens importantes, nomeadamente a diminuição do tamanho do circuito e o aumento do desempenho e fiabilidade do sistema. Recentemente, os circuitos constituídos por componentes deste nível de granulosidade e implementados num único encapsulamento começaram a ser designados por sistemas integrados (*Systems on a Chip*

[1] As definições de ASIC e das escalas de integração (VLSI, LSI, MSI e SSI) serão apresentadas no capítulo 2.

– *SoC*) [VLSI99]. Os sistemas a este nível são normalmente descritos de forma comportamental em linguagem natural, por uma especificação executável, ou por um programa escrito numa linguagem de programação. A designação de sistema para este nível de granulosidade é utilizada por diversos autores [Micheli94, NelNagCarIrw95], além disso, deve ser coerente com a noção de sistema integrado. No entanto, reconhecemos que, de todos os níveis de granulosidade, este é possivelmente o que possui uma designação menos consensual devido à dificuldade em definir o que é um sistema. Para este efeito, vamos estabelecer que um sistema é uma entidade física ou lógica, com apresentação e complexidade adequadas para uma dada aplicação e que por isso pode ser considerado um produto relativamente autónomo. Em geral, um sistema pode ser constituído por componentes eléctricos, electrónicos e mecânicos, podendo também possuir interfaces para interacção com o exterior e para fornecimento de energia. Tendo por base esta definição, tanto as placas de circuito impresso como os circuitos integrados VLSI podem ser considerados sistemas.

Dentro do mesmo nível de granulosidade, um modelo pode possuir várias representações. Da análise da Tabela 1.1 podemos concluir que as duas formas de classificar um modelo possuem uma relação de ortogonalidade. Esta relação está ilustrada de forma mais explícita no plano de abstracção da Figura 1.1(a). Para compreender a ortogonalidade entre os níveis de granulosidade e os tipos de representação, consideremos os seguintes dois exemplos:

- ao nível arquitectural, uma representação comportamental de um modelo de um circuito é um conjunto de operações e suas dependências enquanto numa representação estrutural, o modelo do mesmo circuito é um conjunto de componentes tais como registos e unidades funcionais e respectivas interligações que implementam essas operações.
- um modelo comportamental ao nível lógico de um circuito sequencial pode ser dado por um diagrama de transição de estados, sendo a sua representação estrutural uma interligação de portas lógicas e flip-flops.

Para os restantes casos podem ser feitos raciocínios análogos.

O nível de abstracção de um modelo depende do tipo de representação e do nível de granulosidade dos componentes nele utilizados, uma vez que um modelo é tão mais abstracto quanto menos detalhes de implementação possuir, ou seja, quanto mais longe estiver do nível físico geométrico Figura 1.1(b).

Na Figura 1.1(a) estão também representadas as transformações mais importantes que se podem realizar sobre os modelos, nomeadamente a síntese, a análise, a decomposição descendente e a montagem ascendente. Tanto a síntese como a decomposição descendente são processos de refinamento, mas enquanto a síntese opera sobre o tipo de representação, a decomposição descendente opera sobre o nível de granulosidade. Por outras palavras, enquanto a síntese nos permite alcançar a implementação física do circuito, a decomposição descendente permite-nos realizar a sua divisão em entidades mais simples dentro do mesmo tipo de representação.

No caso de circuitos digitais distinguem-se em particular as seguintes tarefas de síntese: síntese lógica, síntese arquitectural e síntese de sistema, as quais serão descritas

mais à frente. Na transformação de uma representação estrutural de um modelo para uma representação física, o processo de síntese é mais vulgarmente designado por implementação ou projecto físico [Micheli94, Gajski97].

A análise e a montagem ascendente correspondem às transformações inversas da síntese e da decomposição descendente, respectivamente. Tal como representado na figura, a análise pode ser dividida nas seguintes duas componentes:

- Análise física – permite extrair a representação estrutural, através da informação da implementação.
- Análise estrutural – extrai o comportamento a partir da representação estrutural.

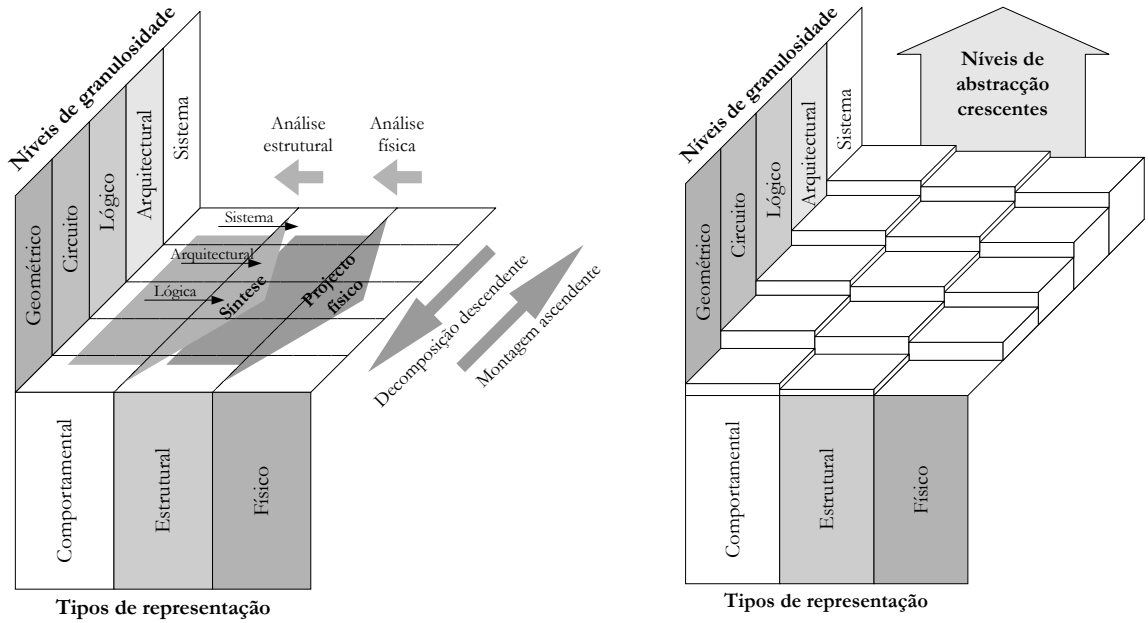


Figura 1.1 – Representação gráfica do plano de abstracção (a) relação entre os tipos de representação, os níveis de granulosidade e as transformações de síntese, análise, decomposição descendente e montagem ascendente; (b) variação do nível de abstracção em função do tipo de representação e do nível de granulosidade.

A montagem ascendente (*bottom-up assembly*) e a decomposição descendente (*top-down decomposition*) [Katz94] são transformações multi-nível que operam sobre o mesmo tipo de representação e são particularmente úteis no projecto estruturado de sistemas não triviais. Na Figura 1.2 está ilustrado um exemplo de montagem e decomposição estrutural de um sistema.

Um sistema pode ser implementado utilizando um procedimento de montagem ascendente, através da construção sucessiva de blocos mais complexos a partir do agrupamento de blocos mais simples, até se atingir a funcionalidade pretendida. No entanto, como é mais fácil compreender o funcionamento de todo o sistema observando os seus componentes e as respectivas interacções, a decomposição

descendente é uma abordagem mais atractiva, sendo uma boa estratégia para construir qualquer tipo de sistema complexo.

A decomposição descendente é a aplicação do princípio "dividir para conquistar", o qual constitui um método eficaz para vencer a complexidade. A decomposição descendente é um processo iterativo que permite a divisão de um problema complexo em partes cada vez mais simples, até a funcionalidade de cada uma destas partes poder ser implementada de forma directa com componentes disponíveis numa dada biblioteca. A principal vantagem desta abordagem é a flexibilidade na exploração de possíveis alternativas de projecto [McFKow90]. O processo de projecto começa com uma solução inicial onde são tomadas as decisões mais importantes, sendo os refinamentos adicionados em cada passo, permitindo assim a exploração de soluções alternativas. Numa decomposição descendente, a descrição de um projecto também é simplificada uma vez que não é necessário trabalhar com vários níveis de abstracção ou tipos de representação em simultâneo [McFKow90].

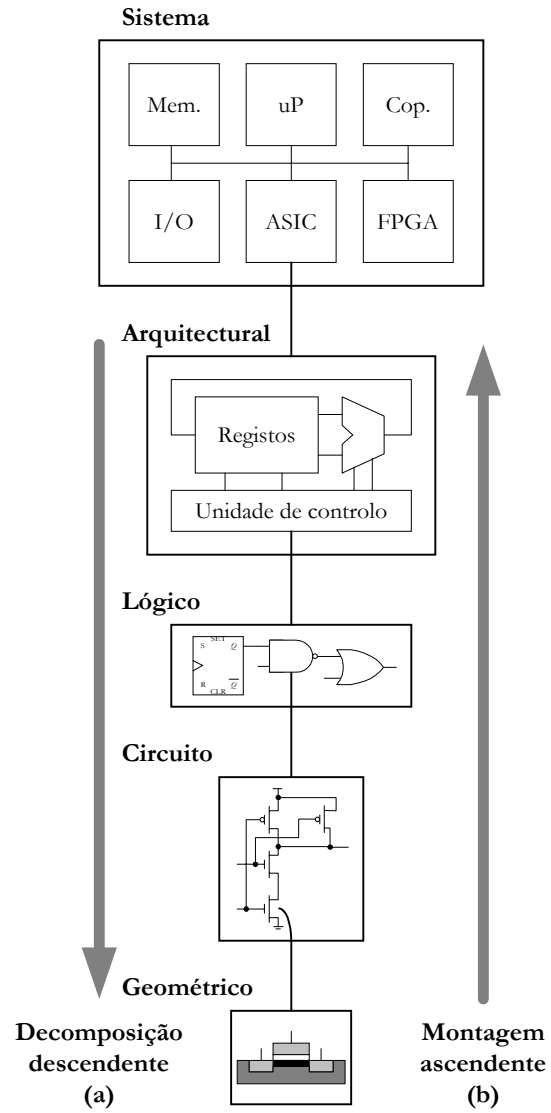


Figura 1.2 – Decomposição descendente (a) e montagem ascendente (b) estrutural de um sistema.

A especificação de um sistema complexo usando decomposição descendente requer a utilização de hierarquia. As especificações hierárquicas são portanto essenciais na gestão da complexidade dos sistemas, quer em termos do tamanho da especificação quer ao nível da facilidade de leitura, possuindo as seguintes vantagens:

- Facilitam o domínio da complexidade através da criação de macro blocos de hardware ou software usando encapsulamento;
- Possibilitam a reutilização desses macro blocos;
- Permitem a migração para hardware de algoritmos complexos normalmente implementados em software.

Na prática, é muitas vezes utilizada uma abordagem mista, na qual o sistema é decomposto sucessivamente em sub-sistemas mais simples, sendo depois cada um implementado com componentes resultantes de uma composição de outros elementos de menor complexidade.

1.1.3 Etapas de Projecto

O ciclo de projecto de um sistema digital é composto por diversas etapas, desde a sua concepção até à construção física, incluindo a especificação, desenvolvimento de bibliotecas, síntese, optimização, implementação, documentação e teste, bem como vários procedimentos de simulação, verificação e avaliação ao longo de todo o processo para validar os resultados das várias etapas. A Figura 1.3 ilustra estas etapas, localizando o resultado de cada uma no plano de abstracção.

A partir de uma descrição resultante da concepção, é desenvolvido um modelo que ao longo do projecto é sistematicamente transformado num circuito digital. Um projecto é examinado através da verificação ou simulação dos seus modelos para analisar a sua operação. A simulação e a verificação são duas abordagens diferentes para validar o funcionamento do circuito. Enquanto a verificação garante uma operação correcta do circuito, quaisquer que sejam as condições de entrada, a simulação só garante para as condições testadas. O modelo é revisto e examinado de novo até serem obtidas as respostas correctas. Para além de verificar o funcionamento correcto, o efeito de diferentes opções de projecto no desempenho do circuito podem ser avaliadas de forma a serem tomadas as decisões eficazes do ponto de vista de custos.

Quando o comportamento modelado do sistema for o correcto, é realizado e implementado o projecto físico. Finalmente, para detectar dispositivos com falhas, o circuito é testado através da comparação dos resultados com os obtidos do comportamento modelado.

Actualmente, muitos dos sistemas digitais contêm o equivalente a milhares ou milhões de portas lógicas, sendo portanto bastante complexos. A maior parte destes sistemas são fabricados num único circuito integrado VLSI. O seu projecto e fabricação é um processo complexo e dispendioso. Para assegurar uma probabilidade elevada de que o circuito vai funcionar correctamente quando for construído é necessário verificar o projecto do circuito lógico antes de iniciar a sua fabricação. O mesmo é verdade quando se projecta sistemas digitais com vários circuitos integrados e placas de circuito impresso. Os circuitos e sistemas desta complexidade são

virtualmente impossíveis de desenvolver e verificar sem o suporte de ferramentas de CAD que constituem uma ajuda fundamental no projecto de produtos de qualidade e economicamente viáveis.

Nas próximas secções descrevem-se de forma resumida as várias etapas e respectivas ferramentas de CAD envolvidas no projecto de um sistema digital:

- Especificação e modelação
- Desenvolvimento de bibliotecas
- Síntese e optimização
- Projecto físico
- Simulação, verificação e avaliação
- Construção e depuração do protótipo
- Documentação

Estas etapas estão representadas na Figura 1.3.

Especificação e Modelação

Especificação

A primeira tarefa no projecto de um sistema digital é a sua especificação, através da qual se descreve a funcionalidade pretendida, os interfaces com o exterior e outras características relevantes para o seu projecto e utilização, tais como velocidade de operação, tecnologia de implementação, consumo de potência, etc. [Gajski97, ErcLanMor99]. No entanto, nesta fase ainda não está definida, pelo menos com precisão, a forma como esta funcionalidade vai ser implementada. Esta descrição deve ser adequada para:

- Servir de ponto de partida para o projecto e implementação do sistema a partir de componentes mais simples;
- Utilizar o sistema como um componente de outro sistema mais complexo.

Tradicionalmente, este processo inclui a elaboração de um esboço da arquitectura de alto nível do sistema, normalmente na forma de um diagrama de blocos. Neste diagrama cada bloco possui uma funcionalidade bem definida que pode ser especificada por relações matemáticas, por um algoritmo, ou ainda descrita em linguagem natural. Os tipos e formatos de dados passados entre blocos e através dos portos de entrada e saída do sistema podem também ser especificados neste diagrama. Geralmente, este tipo de especificação tende a ser vaga e incompleta, devido a limitações do método utilizado. Por isso, esta abordagem de especificação está limitada a sistemas pouco complexos, principalmente, no caso de serem utilizadas descrições em linguagem natural. Estes factores poderão não constituir um problema, uma vez que a especificação serve como ponto de partida para os projectistas, podendo ser modificada e refinada à medida que o projecto evolui.

Mais recentemente, com a complexidade crescente dos projectos, surgiram novos requisitos para os métodos de especificação, de forma a serem capazes de gerir essa complexidade. Assim, uma especificação deve ser tão completa e tão simples

quanto possível, devendo conter todos os pormenores importantes mas deixando de fora todos os outros. Deve também ser formal, no sentido de que a sua interpretação não deve ser ambígua. As especificações em linguagem natural, deram então lugar às especificações executáveis. A crescente popularidade das segundas deve-se ao facto de poderem ser verificadas, analisadas e sintetizadas mais fácil e correctamente por ferramentas específicas e de acordo com um conjunto de regras predefinidas.

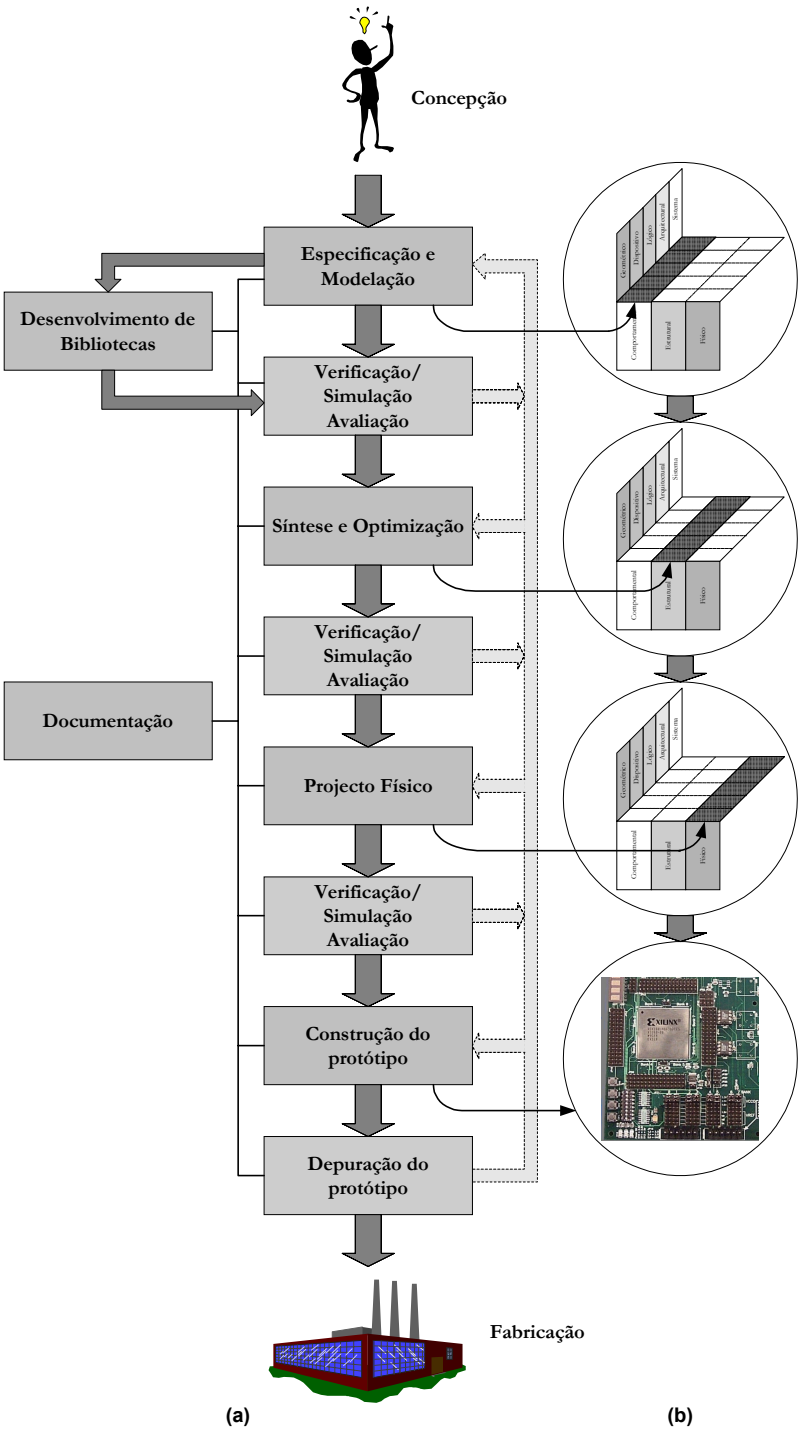


Figura 1.3 – Ciclo de projecto de um sistema digital (a) e correspondência com o plano de abstracção (b).

Existem vários métodos para a especificação de sistemas digitais, tais como linguagens gráficas (Esquemas de Grafos Hierárquicos [Sklyarov84] e *Statecharts* [Harel87]), linguagens de descrição de hardware (*Hardware Description Languages – HDLs*), (VHDL [IEEE94, Ashenden96] e Verilog [ThoMoo96]) e mais recentemente linguagens de programação orientadas por objectos com bibliotecas apropriadas (C++ [GupLia97] e Java [BelHut98]). O método mais apropriado para um caso em particular, depende da complexidade do sistema e da finalidade da especificação.

A descrição do sistema resultante da especificação encontra-se na região comportamental do plano de abstracção (ver Figura 1.3).

Modelação

Quando uma especificação do sistema não é elaborada com base em métodos formais, em particular, no caso de ser escrita em linguagem natural, torna-se necessário desenvolver modelos mais precisos, de forma a que a generalidade das etapas seguintes do projecto possam ser realizadas por ferramentas de CAD apropriadas. Isto possui vantagens quer em termos de tempo de desenvolvimento quer na qualidade da implementação final do circuito.

Se por outro lado a especificação tiver sido feita usando métodos formais e descrições executáveis a fase de modelação confunde-se muitas vezes com a especificação, uma vez que esta pode ser utilizada, em princípio, directamente nas restantes fases do projecto.

O desenvolvimento de modelos apropriados para o tratamento pelas ferramentas de CAD para projecto de sistemas digitais pode ser feito de duas maneiras distintas: por captura gráfica ou por descrição textual. Com o aparecimento das especificações executáveis, a segunda maneira é cada vez mais utilizada.

Em qualquer projecto, a dada altura é necessário descrever uma ou mais secções de forma estrutural tendo por base uma dada biblioteca de componentes. Para este efeito, os componentes e suas interconexões podem ser especificados textualmente, apesar deste ser um procedimento maçador e sujeito a erros. Por outro lado, uma representação estrutural pode ser realizada de forma mais fácil e precisa com uma ferramenta de captura. Estas permitem ao projectista a selecção, posicionamento e interligação de componentes de forma gráfica. Este tipo de representação estrutural é também normalmente designada por esquemático e as ferramentas utilizadas para este efeito denominadas por ferramentas de captura de esquemático.

Alternativamente, a captura de representações estruturais de um projecto pode também ser realizada eficientemente com linguagens de descrição de hardware tais como o VHDL ou o Verilog. Para além de capturarem representações estruturais, estas linguagens também permitem capturar representações comportamentais que descrevem formalmente os algoritmos que definem o comportamento do circuito, podendo o projecto ser descrito como uma combinação do seu comportamento e da sua estrutura. A descrição do projecto começa normalmente com a definição do comportamento de cada bloco existente num diagrama de blocos estrutural de alto nível. À medida que o projecto progride, cada um destes comportamentos é decomposto recursivamente numa estrutura de blocos de mais baixo nível, até ao projecto consistir somente numa hierarquia de blocos, onde os de mais baixo nível são

componentes específicos de uma dada biblioteca. Assim, cada projecto é descrito a diferentes níveis de abstracção, cada um contendo diferentes tipos de informação e de detalhe. Este tipo de descrições hierárquicas são úteis para verificar diversas propriedades do projecto e a conformidade com as restrições impostas. Para além disso, simplifica a gestão de um projecto, facilita a comunicação entre projectistas, diminui os erros de projecto, permite reutilizar componentes desenvolvidos em projectos anteriores e suporta a sua evolução e manutenção. Cada uma destas descrições é chamada um modelo do sistema, uma vez que possui somente a informação relevante para o fim em causa. A informação contida em cada modelo pode ser utilizada por outros projectistas ou por outras ferramentas de CAD para análise, síntese e avaliação da qualidade do projecto.

Desenvolvimento de bibliotecas

No caso do projecto ter sido total ou parcialmente especificado de forma estrutural, é necessário proceder ao refinamento ou à decomposição dos respectivos blocos em componentes mais simples. O objectivo deste processo é assegurar que o circuito é composto unicamente por instâncias de componentes da biblioteca que contém as primitivas disponíveis na tecnologia de fabricação utilizada. Esta biblioteca possui normalmente componentes do nível lógico, apesar de ser frequente encontrar bibliotecas com componentes de mais do que um nível de abstracção tais como registos (nível arquitectural) e portas lógicas (nível lógico).

Qualquer projecto necessita normalmente de componentes que não se encontram em nenhuma das bibliotecas fornecidas, pelo que é necessário desenvolvê-los. Nestes casos, é frequente a construção de bibliotecas de componentes proprietários, de forma a poderem ser reutilizados noutros projectos.

Os componentes de uma biblioteca devem ser projectados, testados e bem documentados de forma a que qualquer pessoa os possa utilizar sem ter de analisar a sua estrutura. Um componente é caracterizado por diversos atributos, entre eles a funcionalidade, os portos de entrada e saída, as dimensões físicas, os parâmetros eléctricos e temporais e os modelos destinados às ferramentas de CAD para síntese, simulação, verificação, projecto físico e teste.

Síntese e Optimização

No processo de projecto, a síntese é procedimento através do qual se converte uma especificação ou uma descrição comportamental de um componente numa descrição estrutural usando componentes de níveis de abstracção inferiores pertencentes a uma dada biblioteca [Gajski97, Micheli94]. A síntese pode ser vista como um processo de refinamento da descrição comportamental à qual é adicionada informação estrutural em cada iteração. Na prática, isto traduz-se normalmente na partição da descrição comportamental em vários blocos. A descrição estrutural resultante contém cada um dos blocos obtidos e suas interligações, bem como as suas descrições comportamentais. Este procedimento é repetido até cada bloco representar um dos componentes da biblioteca alvo. Nos procedimentos de síntese estão geralmente integrados mecanismos de optimização, os quais permitem melhorar o desempenho e/ou reduzir o custo do sistema projectado.

Tendo por base os níveis de abstracção da Tabela 1.1, podem ser identificadas as seguintes tarefas distintas de síntese:

- **Síntese de sistema** – decompõe uma especificação abstracta da funcionalidade do sistema em várias tarefas e efectua a sua partição entre uma implementação em hardware ou em software. No caso do hardware, as tarefas são implementadas numa estrutura de componentes ao nível do sistema, tais como processadores, memórias e ASICs. No caso do software as várias tarefas poderão corresponder a diferentes módulos ou objectos do programa. Um dos possíveis objectivos da síntese de sistema é reutilizar componentes predefinidos, dos quais os blocos de propriedade intelectual (*Intellectual Property – IP*) são um bom exemplo. Na área dos sistemas embutidos (*embedded systems*), a síntese de sistema é também vulgarmente designada por co-síntese ou co-projecto de hardware-software [GupMic93, ThoAdaSch93, Wolf94, IsmJer95, MicGup97, StaWol97].
- **Síntese arquitectural** – um modelo comportamental ao nível arquitectural pode ser abstraído como um conjunto de operações e dependências, normalmente descritas por algoritmos, fluxogramas ou conjuntos de instruções. A partir desse modelo, a síntese arquitectural gera uma representação estrutural ao mesmo nível que descreve a forma como essas operações devem ser implementadas. Isto consiste na identificação do tipo e determinação da quantidade dos recursos de hardware que implementam as respectivas operações, na sua interligação, no escalonamento temporal das operações e na sua associação com os recursos alocados. Por outras palavras, a síntese arquitectural define um modelo estrutural de uma unidade de execução, como uma interligação de recursos e um modelo lógico de uma unidade de controlo, que emite os sinais que vão controlar a unidade de execução de acordo com o escalonamento efectuado. Exemplos de componentes utilizados neste nível de abstracção são contadores, registos, pilhas, filas, somadores e multiplicadores. Outras designações para a síntese arquitectural são síntese de alto nível ou síntese estrutural, uma vez que determina a estrutura macroscópica do circuito. No entanto, por uma questão de uniformidade e clareza, é preferível a utilização da primeira designação. Os parâmetros macroscópicos da implementação, tais como área do circuito e desempenho, dependem fortemente deste processo, o qual, determina também o nível de paralelismo das operações. A síntese arquitectural é muito útil no projecto de circuitos de interface, coprocessadores específicos, algoritmos para processamento digital de sinal, entre outros. A síntese arquitectural está bem documentada em [McFParCam90, CamWol91, GajDutWuLin92, MicLauDuz92, GajRam94, Micheli94, WalCha95].
- **Síntese lógica** – é o processo através do qual é gerada uma representação estrutural de um modelo comportamental ao nível lógico. Isto consiste na manipulação das especificações lógicas, normalmente expressões Booleanas, para criar modelos baseados em primitivas lógicas e interconexões, tentando ao mesmo tempo reduzir a quantidade de lógica necessária, o atraso de

propagação, ou consumo de potência. Assim, a síntese lógica determina a estrutura microscópica (i. e. ao nível das portas lógicas) do circuito. A transformação de um modelo lógico numa interconexão de instâncias de células de uma dada biblioteca é o último passo da síntese lógica sendo normalmente designada por mapeamento na tecnologia. Um modelo ao nível lógico de um circuito pode ser representado por um diagrama de transição de estados, por uma tabela de verdade, por um diagrama esquemático ou por uma linguagem de descrição de hardware. Em qualquer dos casos o modelo pode ser elaborado pelo projectista ou sintetizado a partir de outro de nível superior. A síntese lógica está documentada em [MicLauDuz92, Micheli94].

- **Síntese sequencial** – corresponde a um caso particular da síntese lógica, sendo utilizada para gerar circuitos que contenham elementos de memória, tais como unidades de controlo descritas por máquinas de estados finitos. O resultado da síntese sequencial é uma interligação de portas lógicas e flip-flops. Os objectivos da síntese sequencial são a minimização do número de elementos de memória usados no circuito, a geração de uma codificação de estados e entradas que reduza o seu custo, a diminuição do atraso entre as entradas e saídas e por último a simplificação das expressões Booleanas resultantes dos passos anteriores e necessárias para a implementação do circuito. A síntese sequencial está documentada em [AshDevNew92, Micheli94].

Abaixo do nível lógico, as técnicas de síntese estão divididas de acordo com as áreas de aplicação especificas e são altamente dependentes das tecnologias de implementação utilizadas [Gajski97]. O modelo do sistema resultante do processo de síntese encontra-se na região estrutural do plano de abstracção (ver Figura 1.3).

Na Figura 1.4 está ilustrada a sequência de tarefas de síntese de um sistema a partir de uma descrição comportamental abstracta. Por uma questão de generalidade, considera-se também que o sistema pode ser constituído por componentes de software.

A síntese pode ser realizada manualmente ou de forma automática por ferramentas de CAD especificas. Devido à sua complexidade o processo manual é bastante moroso, sujeito a erros e actualmente virtualmente impossível, principalmente no caso de projectos com milhões de portas lógicas, como os circuitos integrados VLSI. Além disso, as ferramentas de CAD disponíveis actualmente são capazes de sintetizar circuitos complexos, em espaços de tempo relativamente reduzidos e na maioria dos casos a qualidade do resultado final é superior àquela que se conseguiria manualmente.

As ferramentas de síntese lógica são muito úteis no projecto de circuitos combinatórios tais como unidades aritméticas, comparadores, codificadores, descodificadores, etc. Por outro lado, as ferramentas de síntese sequencial são usadas essencialmente no projecto de unidades de controlo.

Quanto às ferramentas de síntese arquitectural, são utilizadas para converter expressões aritméticas, conjuntos de instruções ou descrições algorítmicas em

estruturas ao nível dos registos, minimizando a área do circuito e/ou o tempo de execução.

Por último, as ferramentas de síntese de sistema são as que se encontram menos desenvolvidas actualmente. No entanto, esta é uma área de investigação bastante activa em grande parte devido às elevadas capacidades de integração disponíveis, sendo possível implementar num único circuito integrado um sistema bastante complexo.

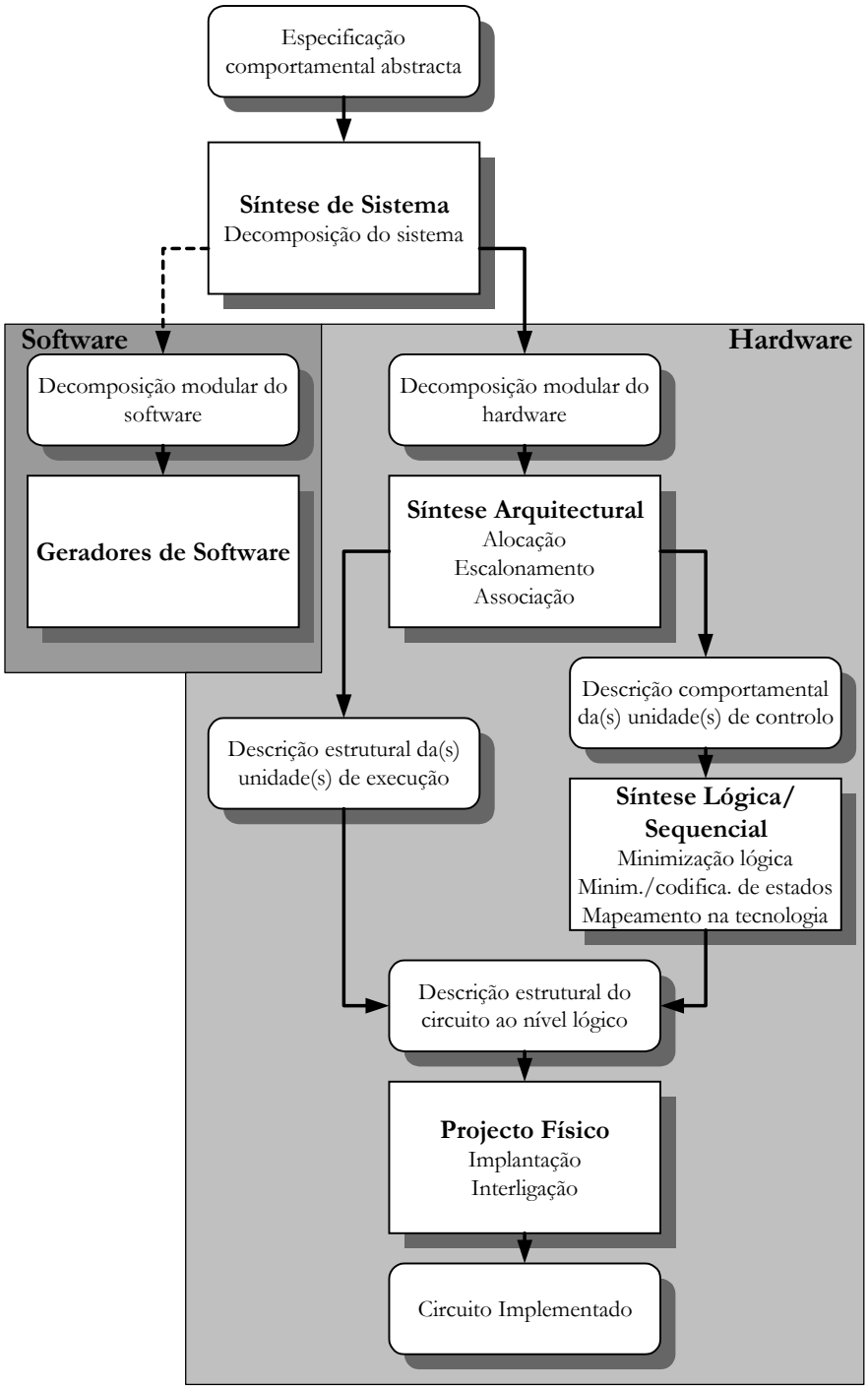


Figura 1.4 – Síntese de um sistema a partir de uma descrição comportamental abstracta.

Projecto Físico

A etapa seguinte do desenvolvimento do sistema é o seu projecto físico, também designado por implementação, o qual consiste na implantação e interligação dos componentes resultantes da fase de síntese e optimização do circuito. O projecto físico é normalmente executado em placas de circuito impresso e/ou circuitos integrados.

O primeiro passo do projecto físico é a implantação de cada componente do circuito numa posição específica da placa de circuito impresso ou nas células ou módulos do circuito integrado, de forma a minimizar a área total por eles ocupada. Esta minimização é baseada no tamanho e forma de cada componente da representação estrutural e nas posições dos seus pinos ou portos de entrada e saída.

Uma vez realizada a implantação de todos os componentes, o passo seguinte é a determinação da localização das suas interligações, de forma a cumprir os seguintes objectivos:

- Minimizar o comprimento máximo de cada ligação ou a soma dos comprimentos de todas as ligações, diminuindo a área por elas ocupada, bem como os respectivos tempos de propagação;
- Assegurar um traçado correcto das linhas de relógio de forma a evitar desfasamentos, que serão responsáveis por um funcionamento incorrecto do circuito;
- Reduzir o acoplamento capacitivo e eliminar possíveis interferências entre as ligações;
- Optimizar a utilização dos níveis de metalização da placa de circuito impresso ou do circuito integrado, que contribuem significativamente para o custo final do circuito;
- Certificar que todos os componentes recebem a tensão e a corrente necessárias para garantir o seu funcionamento correcto, através da dimensão adequada das interligações.

O não cumprimento de alguns destes objectivos pode ser uma fonte de falhas intermitentes ou permanentes do circuito.

As ferramentas de CAD disponíveis para o projecto físico, simplificam e aceleram a sua realização através da execução automática das tarefas de implantação e interligação dos componentes do circuito. Normalmente, para melhorar a qualidade dos resultados obtidos, é também possível a utilização de directivas de orientação das ferramentas de implementação. Por outro lado, a realização manual destas tarefas, é útil nos casos em que se pretende obter implementações bastante optimizadas, quer em termos de área quer em temos de desempenho do circuito. No entanto, devido à sua morosidade, o número de operações manuais de implantação e interligação deve ser reduzido. Estas ferramentas incorporam também mecanismos de verificação automática dos objectivos acima descritos.

O formato de dados de entrada é geralmente uma lista de ligações (*netlist*), a qual contém todos os componentes do circuito e as suas conexões e propriedades. Após a implementação, algumas destas ferramentas são capazes de adicionar à lista de ligações inicial, informação sobre os atrasos dos componentes e interligações do circuito, para

permitir uma verificação ou simulação mais precisa antes da sua construção. Este procedimento é normalmente conhecido por *back annotation*.

A implementação do sistema resultante do projecto físico encontra-se na região física do plano de abstracção (ver Figura 1.3).

Verificação, Simulação e Avaliação

A validação dos modelos e da implementação de um sistema digital pode ser baseada pelo menos em três métodos diferentes: verificação formal, verificação por simulação e avaliação. Por questões de simplicidade de linguagem, as duas primeiras passaram a ser designadas a partir de agora por verificação e simulação, respectivamente. Estes procedimentos estão distribuídos ao longo de todo o processo de projecto de forma a validar os resultados obtidos nas etapas que os precedem. Isto é importante porque quanto mais cedo for detectado um problema de especificação, ou um erro ou abordagem incorrecta de projecto, mais fácil e mais barata é a sua reformulação ou correcção. De notar que a utilização destes procedimentos só é técnica e economicamente viável devido à existência de ferramentas de CAD que os realizam. Se após uma dada etapa do projecto os resultados obtidos da validação forem positivos, o projecto pode prosseguir para a etapa seguinte, caso contrário deve retornar ao ponto em que seja possível adoptar outras opções de projecto ou corrigir os erros existentes. Em geral, os modelos de um sistema são validados após as seguintes fases de projecto:

- Especificação e modelação – para assegurar que o comportamento modelado do sistema é aquele que efectivamente se pretende quando este for construído.
- Síntese e optimização – para verificar que as descrições comportamental e estrutural do sistema possuem a mesma funcionalidade, ou seja, que o projecto foi sintetizado correctamente.
- Projecto físico – para certificar que o desempenho do sistema implementado cumpre os requisitos da sua especificação.

Simulação

A simulação é o tipo mais popular de verificação de um sistema digital. Os seus objectivos principais são: a verificação lógica, a análise do desempenho e o desenvolvimento de testes. No diagrama de blocos da Figura 1.5 está ilustrada uma estrutura típica de um ambiente de simulação. O modelo do circuito é fornecido ao simulador na forma de uma descrição funcional ou de uma lista de ligações, consoante o estado de desenvolvimento do circuito. Em qualquer dos casos o simulador deve aceder às bibliotecas apropriadas para obter a implementação das funções e os modelos de simulação dos componentes utilizados.

Um modelo de um circuito lógico é validado por simulação através da aplicação de vectores de teste às suas entradas. Um vector de teste é uma lista ordenada de uns e zeros, cada um correspondendo ao valor do estímulo a aplicar a uma dada entrada do circuito. Se o objectivo da simulação for a validação lógica de um circuito combinatório, a ordem pela qual os vectores de teste são aplicados é irrelevante.

Numa simulação exaustiva, são utilizadas todas as combinações de entrada, o que permite verificar a totalidade da tabela de verdade do circuito. Isto corresponde a 2^n vectores para testar um circuito combinatório com n entradas, o que pode ser impraticável para circuitos com muitas entradas. Nesses casos, os conjuntos de teste são concebidos de forma a permitirem detectar o maior número de falhas possível e a verificar as operações mais comuns e críticas do circuito. No caso de circuitos sequenciais a situação é ainda mais complexa porque a ordem pela qual são aplicados os vectores de teste é importante.

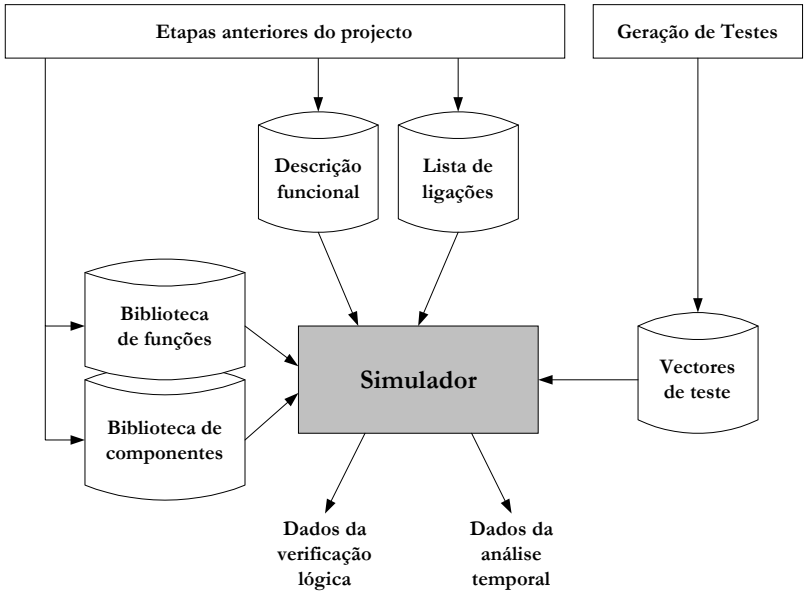


Figura 1.5 – Ambiente típico de simulação de sistemas digitais.

O número e tipo dos vectores de teste utilizados estabelecem o nível de confiança do projectista na simulação, ou seja de que o circuito projectado não possui erros. Assim, um modelo só é normalmente simulado para um subconjunto de todos os valores de entrada possíveis. Por esta razão, quando se usa um simulador deve-se procurar utilizar os conjuntos de valores de entrada e de saída que permitam testar todos os caminhos e componentes do projecto. Esta pode ser uma tarefa complexa no caso de projectos complexos com milhões de portas lógicas ou transístores. As respostas obtidas à saída das portas lógicas e flip-flops pretendidos são depois capturadas e comparadas com as tabelas de verdade, equações lógicas, linguagens de descrição de hardware (*Hardware Description Languages – HDL*) ou outra forma de especificação, a partir da qual o modelo foi desenvolvido. Nas etapas iniciais do ciclo de projecto, o objectivo principal da simulação é validar o funcionamento do circuito do ponto de vista lógico. Por este motivo são normalmente utilizados modelos ideais ou simplificados dos componentes, que não apresentam tempos de atraso na sua resposta aos estímulos de entrada. Isto permite separar a informação temporal da função lógica, para simplificar as análises preliminares.

Para analisar o desempenho de um circuito, o modelo de cada componente deve descrever da forma mais fiel possível as características físicas do dispositivo correspondente. Em particular, deve ser modelado com precisão, o tempo necessário

para que o dispositivo responda a um estímulo de entrada, vulgarmente referido por atraso de propagação. A utilização de modelos precisos permite, a partir da análise dos resultados de simulação, prever o atraso de propagação total entre quaisquer entradas e saídas do circuito e a detecção de outros problemas temporais, tais como picos e transições acidentais (*spikes e hazards*). Quando o projectista é confrontado com várias opções de projecto, a simulação constitui um método para avaliação dos efeitos de cada opção no desempenho do circuito, permitindo realizar as melhores escolhas.

Um circuito deve ser testado após a sua fabricação para determinar se possui componentes ou interligações defeituosas. Através de equipamento apropriado e para cada falha que se pretende detectar, são aplicados vectores de teste às entradas do circuito e analisadas as saídas obtidas. Estas diferem nos casos de um circuito que funcione correctamente e de um circuito que apresente uma ou várias falhas, permitindo a sua detecção. A simulação de falhas é o processo através do qual se força a ocorrência de várias falhas num circuito e se determina se são ou não detectáveis por um dado conjunto de vectores de teste. Os resultados deste processo indicam a percentagem de falhas que pode ser detectada por um conjunto de vectores de teste em particular. Assim, a simulação de falhas auxilia no desenvolvimento de um conjunto de vectores de teste para um circuito lógico, que permita detectar uma percentagem aceitável de falhas no menor intervalo de tempo possível.

A maioria dos simuladores lógicos são baseados em eventos. Um evento é definido como uma alteração do valor de um sinal num dado instante. Um simulador baseado em eventos está organizado em torno de uma fila de eventos. Esta serve para os armazenar pela ordem em que a sua ocorrência está escalonada. Em cada passo da simulação é removido o primeiro evento da fila e alterado o valor do respectivo sinal. Se este sinal for uma entrada de um ou mais componentes, então o valor de saída de cada componente afectado é recalculado. Para cada valor de saída calculado que seja diferente do anterior, é gerado e colocado na fila um novo evento para o respectivo sinal. O instante em que este deve ocorrer corresponde ao instante actual mais o atraso de propagação do respectivo componente. Estas operações são repetidas durante a simulação, até a fila de eventos ficar vazia ou até ter decorrido o intervalo de tempo especificado. A simulação é iniciada com a conversão do conjunto de entradas de teste num conjunto de eventos e na sua inserção na fila de acordo com o seu escalonamento. Durante a simulação é mantido um registo de todos os eventos na forma de uma tabela ou diagrama temporal, a partir do qual são gerados e examinados os resultados da simulação.

Verificação

A simulação juntamente com os respectivos vectores, tornou-se demasiado morosa e consequentemente inadequada para a validação de projectos complexos com milhões de portas lógicas, tais como os sistemas integrados. Por outro lado, as técnicas e ferramentas de verificação formal não necessitam de vectores de simulação e permitem validar de forma simbólica os modelos do circuito integrado ao longo do seu fluxo de projecto [Milne94]. Usando um conjunto de técnicas de manipulação pode-se provar que dois modelos diferentes do mesmo projecto são equivalentes quaisquer que sejam as condições lógicas do circuito. De uma forma simplista, a verificação formal

baseia-se na aplicação de métodos matemáticos rigorosos para validar exaustivamente a operação do sistema. Para o projecto de circuitos integrados, as ferramentas actuais de verificação formal permitem comprovar a sua funcionalidade e pertencem a uma das seguintes três categorias: teste de equivalência, teste de modelos e demonstração de teoremas [Lipman98].

Uma vantagem da verificação formal em relação às ferramentas de simulação é o facto de não necessitarem de vectores de simulação. Este factor é importante, porque os sistemas com milhões de portas lógicas necessitam de uma quantidade enorme de vectores para verificar minimamente a sua funcionalidade. Outra vantagem da verificação formal é ser exaustiva, uma vez que as ferramentas que a realizam, fazem-no de forma independente das condições de entrada ou do estado do circuito. A simulação não é um mecanismo de validação exaustivo porque a cobertura e detecção de eventuais falhas do circuito, depende da escolha correcta dos vectores utilizados. A desvantagem da verificação formal é a necessidade de aprendizagem de novas técnicas de verificação para que se possa aplicar eficazmente as ferramentas que as implementam no projecto de sistemas complexos. Além disso, o número de ferramentas de verificação formal existentes actualmente é reduzido e o seu custo bastante elevado [Lipman98].

A verificação formal não é um substituto mas sim um complemento dos métodos de simulação baseados em vectores. A simulação continuará a ser utilizada, por exemplo, nas últimas fases do projecto de um sistema, onde é necessária informação temporal detalhada para determinar o desempenho do circuito [Lipman98].

Avaliação

A avaliação do projecto permite por um lado apurar se os requisitos da especificação foram satisfeitos, tais como área ou consumo de potência e por outro, averiguar se o projecto desenvolvido é realmente a melhor das várias alternativas possíveis quer em termos de tempo de desenvolvimento, quer em termos da qualidade final do circuito. Esta avaliação baseia-se normalmente em medidas de qualidade, tais como, custo, desempenho e testabilidade. Por exemplo, uma das métricas mais importantes é o custo de produção de um produto. Esta métrica é normalmente aproximada pelo tamanho ou área, uma vez que a área de um circuito integrado ou placa de circuito impresso é proporcional ao custo da sua fabricação. O número de pinos de entrada e saída é outra métrica importante, já que o custo do encapsulamento é proporcional ao número de pinos. Recentemente, o consumo de potência tornou-se igualmente importante devido ao aparecimento em massa de equipamentos portáteis, tais como computadores e telefones sem fios. Uma vez que o consumo de potência determina o tamanho das baterias, possui também bastante influência no peso final do produto.

Outro aspecto importante no projecto de um produto é o seu desempenho, podendo ser medido de várias formas. As três métricas de desempenho mais populares são: o atraso entre as entradas e as saídas, o período de relógio e o tempo necessário para um programa, algoritmo, ou instrução completar a sua execução. Em geral, os componentes com menores atrasos, os circuitos com ciclos de relógio mais pequenos e

as aplicações com menores tempos de execução são considerados como tendo melhores desempenhos.

Finalmente, as métricas de testabilidade são definidas em termos do número de falhas de fabricação detectáveis e do número de padrões de teste necessários para detectar todas essas falhas. Cada padrão ou vector de teste contém um conjunto de entradas e os correspondentes valores de saída esperados no caso de uma operação sem falhas. Em geral, o número de potenciais falhas é proporcional ao número de padrões de teste necessários, que por sua vez é proporcional ao tempo necessário para testar o produto fabricado.

Quanto mais avançado for o estado de desenvolvimento do projecto, maior é a precisão das métricas usadas na sua avaliação. No entanto, são também maiores os impactos negativos no custo do projecto resultantes da correcção de eventuais problemas.

Documentação

A documentação é uma tarefa que deve ser efectuada ao longo de todo o projecto, devendo conter toda a informação relativa ao sistema desenvolvido. No entanto, é após a validação do projecto físico e antes da construção do protótipo do sistema que são preparadas as versões candidatas a definitivas dos documentos.

Em geral, existe a necessidade de fazer a sua actualização após a construção e depuração do protótipo e em muitos casos também durante a própria fabricação do produto. A documentação do domínio público inclui geralmente as representações comportamental e física do produto, mas não possui informação detalhada sobre a representação estrutural, a qual é considerada informação proprietária que é revelada somente aos departamentos da empresa responsáveis pela fabricação.

A informação comportamental é dada normalmente na forma de um diagrama de blocos acompanhado por fluxogramas que descrevem o comportamento de todo o sistema ou de algumas das suas partes. Adicionalmente, a documentação comportamental pode apresentar os protocolos de comunicação, os quais especificam a forma como o sistema interage com o ambiente que o rodeia, sendo normalmente fornecidos na forma de diagramas temporais.

Por outro lado, a representação física contém o tamanho, o encapsulamento e os nomes e posições de todos os conectores. Finalmente, esta documentação especifica os valores mínimos, nominais e máximos da corrente, tensão, consumo de potência, temperaturas, atrasos temporais e outros parâmetros do sistema.

Construção do Protótipo

Após o projecto físico e documentação, são normalmente construídas algumas unidades do sistema, as quais servirão para fazer uma avaliação final e depuração do produto desenvolvido. Nesta fase, os planos finais do projecto são convertidos num circuito integrado ou placa de circuito impresso através da montagem e interligação física dos diversos componentes que constituem o sistema.

Após a certificação de que o produto cumpre todos os objectivos preestabelecidos, o sistema passa então à fase de fabricação.

Depuração do Protótipo

Após a construção das primeiras unidades, é efectuada a sua depuração para assegurar que o funcionamento final do sistema é o esperado. À semelhança das etapas de verificação, simulação e avaliação, se os resultados obtidos não forem satisfatórios deve-se recuar no processo de projecto para corrigir eventuais problemas e só depois o produto deverá passar à fase de fabricação. De notar, que é nesta fase que a detecção de erros de projecto possui consequências mais graves devido aos elevados custos que acarreta.

Fabricação

A fabricação não faz na realidade parte do processo de projecto, sendo antes o seu objectivo final. Depois da fabricação são normalmente realizados testes para identificar, separar e se possível corrigir os produtos que não estejam a funcionar correctamente. Em geral, este procedimento consiste na aplicação de padrões de teste à unidade a testar e na comparação das saídas obtidas com as esperadas no caso de estar a funcionar correctamente. Este testes podem ser realizados à velocidade nominal de operação ou a velocidades inferiores àquelas que o produto normalmente opera. No caso de produtos com elevado desempenho o teste à velocidade normal é bastante difícil de realizar, uma vez que o equipamento de teste deve ser bastante mais rápido que o dispositivo a testar.

1.1.4 Metodologias de Projecto

Durante o projecto de um sistema electrónico digital, qualquer um dos níveis de granulosidade, tipos de representação e etapas de projecto podem ser utilizadas diversas vezes, dependendo dos objectivos, tecnologias, componentes, bibliotecas e alternativas de projecto que se pretende explorar. Uma vez que esta exploração pode ser realizada de diversas formas, deve ser escolhida cuidadosamente uma metodologia de projecto, a qual determina o subconjunto mais indicado de níveis de granulosidade, tipos de representação, tarefas de síntese e ferramentas de CAD utilizadas em cada fase do processo de projecto [Gajski97]. De todas as metodologias disponíveis, importa realçar duas, uma por questões históricas e a outra por ser a mais actual.

A primeira metodologia de projecto foi baseada nos procedimentos de captura de esquemático e simulação [GajVahNarGon94]. Nesta metodologia, o ponto de partida é a elaboração de um diagrama de blocos, que descreve a arquitectura do sistema a projectar. O passo seguinte é a conversão da funcionalidade de cada bloco no diagrama esquemático do circuito que o implementa, para a seguir ser realizada a sua captura por ferramentas de CAD apropriadas. A verificação da funcionalidade é efectuada através de simulação.

Mais recentemente, quando a síntese lógica foi adoptada pela comunidade de projectistas e passou a fazer parte do processo de projecto, começaram-se a utilizar expressões Booleanas e diagramas de máquinas de estados finitos para descrever lógica, em vez da captura de portas lógicas e flip-flops em diagramas esquemáticos. Por outro lado, esta nova metodologia encorajou a captura do projecto usando descrições comportamentais, baseadas em linguagens de descrição de hardware, tendo a metodologia baseada em captura e simulação dado lugar a uma baseada em descrição e

síntese [GajVahNarGon94]. Nesta nova metodologia, a estrutura do projecto é sintetizada automaticamente por ferramentas de CAD, em vez de ser gerada manualmente. O processo manual por ser maçador e sujeito a erros, só é viável para circuitos triviais. Uma vez que esta metodologia pode ser aplicada a vários níveis de abstracção, a sua aplicação evoluiu no sentido dos níveis de abstracção mais elevados, o que resultou em ganhos significativos de produtividade [GajVahNarGon94]. De notar, no entanto, que esta metodologia está mais definida, é melhor suportada por ferramentas de CAD e produz melhores resultados nos níveis de abstracção mais baixos, existindo ainda muito a fazer principalmente ao nível comportamental do sistema.

1.2 Unidades de Controlo

Os sistemas computacionais, bem como outros tipos de dispositivos digitais que realizam processamento de dados, podem ser decompostos numa unidade de execução e numa unidade de controlo [GajVahNarGon94, Micheli94] (ver Figura 1.6). O exemplo mais comum deste tipo de dispositivos é o microprocessador. Ambas as unidades podem possuir entradas e saídas que ligam a outros componentes do sistema (ex. memórias e periféricos), ou a outros dispositivos existentes no ambiente onde o sistema está inserido, como por exemplo, sensores e actuadores. Enquanto as linhas de entrada e saída da unidade de controlo são tipicamente sinais de um só bit (ex. linhas de interrupção e de controlo de memória), no caso da unidade de execução são sinais multi-bit, (ex. barramentos de dados e endereços). As duas unidades estão interligadas por intermédio de linhas de controlo e linhas de estado, cuja função será explicada mais à frente.

A unidade de execução realiza o processamento propriamente dito e é constituída por registos, multiplexadores e unidades funcionais, tais como unidades aritméticas e lógicas, multiplicadores e deslocadores. Numa operação típica da unidade de execução, os operandos são lidos dos respectivos registos, o resultado é calculado nas unidades funcionais e por último escrito no registo de destino. Consoante a arquitectura do dispositivo, os registos que armazenam os operandos e o resultados das operações podem ser internos à unidade de execução ou pertencentes a uma memória externa. Normalmente esta memória está ligada à unidade de execução por intermédio de barramentos. O estado interno da unidade de execução, tais como resultados de comparações ou condições de erro, é comunicado à unidade de controlo por intermédio das linhas de estado.

A unidade de controlo, estabelece a sequência de operações realizadas pela unidade de execução e é normalmente modelada por uma máquina de estados finitos. A sua estrutura baseia-se num registo de estado que armazena o estado actual, na lógica de estado seguinte que calcula o novo estado que será armazenado no registo de estado e por último, na lógica de saída que determina o valor das saídas da unidade de controlo em função do estado actual e das entradas.

Numa máquina de estados finitos, a transição entre estados é realizada por um sinal de relógio que assegura a sincronização correcta do circuito. A unidade de controlo realiza um conjunto de instruções que depende dos valores dos seus sinais de entrada, ou seja, das entradas de controlo do sistema e dos sinais de estado da unidade

de execução. A geração dos sinais de controlo da unidade de execução bem como das saídas de controlo do sistema depende das instruções executadas. A função dos primeiros é definir as operações realizadas na unidade de execução bem como os respectivos operandos.

Alguns dos aspectos mais importantes do projecto de uma unidade de controlo, nomeadamente as arquitecturas de implementação e os processos de especificação e de síntese serão abordados nos capítulos 3, 4 e 5 desta dissertação.

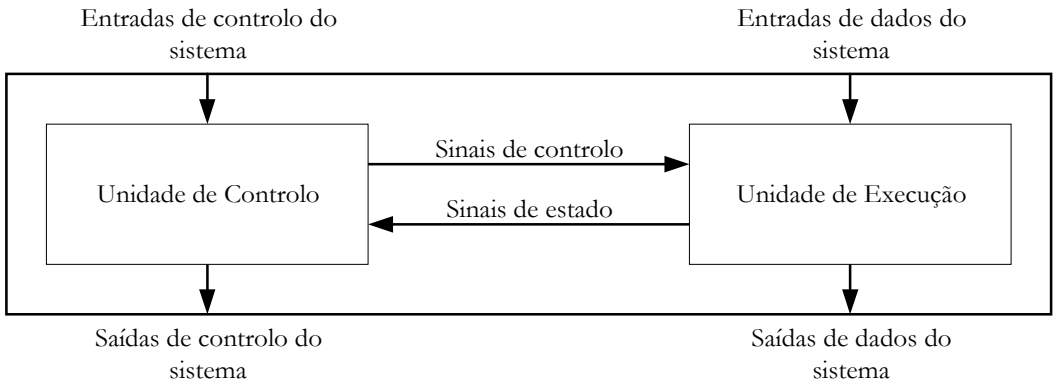


Figura 1.6 – Estrutura de um sistema computacional.

1.3 Objectivos do Trabalho

Este trabalho tem como objectivos principais a investigação de modelos, a concepção de arquitecturas e o desenvolvimento de ferramentas para o projecto e implementação de unidades de controlo complexas, flexíveis, extensíveis e reutilizáveis. Actualmente, com as elevadas capacidades de integração disponíveis, a complexidade crescente dos projectos e finalmente a necessidade de tempos de desenvolvimento cada vez menores, os objectivos referidos acima descritos são bastante importantes.

No caso particular das unidades de controlo, a flexibilidade significa a possibilidade de modificar o comportamento definido, num reduzido período de tempo e com o mínimo de esforço. Quanto à extensibilidade, é uma medida da facilidade com que se pode ampliar o comportamento da unidade de controlo após o seu projecto inicial, de forma a melhorá-lo ou acrescentar funcionalidades. Por último, a reutilização torna possível a incorporação em novos projectos de componentes desenvolvidos e testados em projectos anteriores, aumentando a qualidade do produto final e reduzindo o tempo de desenvolvimento [MicGup97]. Nenhum destes objectivos é alcançável com a utilização do modelo tradicional de descrição de unidades de controlo, a máquina de estados finitos [Kohavi70]. Isto deve-se ao facto de qualquer alteração que se pretenda realizar no comportamento da unidade de controlo, implicar a repetição de todos os passos de projecto. Por outro lado, o modelo de máquinas de estados finitos também não é indicado para a descrição de unidades complexas, uma vez que quando o número de estados, transições, entradas ou saídas é elevado, as descrições baseadas neste modelo são de difícil realização e análise [GajVahNarGon94], sendo portanto imprescindível a adopção de outros modelos mais

apropriados. Uma das abordagens mais conhecidas para vencer a complexidade é baseada no principio "dividir para conquistar", o que neste caso se traduz na utilização de um método de especificação que suporte uma decomposição hierárquica do algoritmo que descreve o comportamento da unidade de controlo, em sub-algoritmos de menor complexidade, logo mais tratáveis e em certos casos também reutilizáveis.

Ainda quanto à flexibilidade e extensibilidade, pretende-se também que estas propriedades sejam utilizadas dinamicamente, isto é, que possamos alterar o comportamento da unidade de controlo em pleno funcionamento, através da modificação de sub-algoritmos existentes, ou da adição de novos sub-algoritmos. De notar que estes procedimentos não devem conduzir a um comportamento imprevisível da unidade de controlo, pelo que uma das possibilidades é permitir somente a modificação de sub-algoritmos que não se encontrem activos. Este objectivo tem implicações na metodologia de projecto e na arquitectura de implementação, as quais devem promover a independência e modularidade de cada sub-algoritmo quer durante o projecto, quer na sua realização física. As tecnologias utilizadas na implementação do circuito possuem também um papel fundamental, devendo os respectivos dispositivos, ser reconfiguráveis dinâmica e parcialmente, isto é, permitirem a modificação da funcionalidade implementada em determinadas partes do dispositivo sem que seja necessário suspender completamente a sua operação para realizar uma reconfiguração completa. Os dispositivos que preenchem estes requisitos são algumas das FPGAs com memória de configuração do tipo RAM estática [Xilinx97a]. A utilização deste tipo de dispositivos tem também outra potencialidade que interessa explorar: a implementação de unidades de controlo complexas com recursos de hardware limitados, mantendo na FPGA somente os circuitos que implementam os sub-algoritmos necessários em cada momento. A informação de configuração dos restantes sub-algoritmos está armazenada numa memória de configuração externa, mais económica que uma FPGA, sendo transferida para a memória de configuração da FPGA somente quando necessária. Devido à sua semelhança com os sistemas de memória virtual, os circuitos que utilizam estas facilidades, são normalmente designados por sistemas de hardware virtual [KurBagAthMuñ00]. No caso particular dos circuitos de controlo, são denominados unidades de controlo virtuais.

1.4 Organização da Dissertação

Além deste capítulo de introdução, esta dissertação possui mais sete capítulos, cinco anexos, uma lista de referências e uma lista de acrónimos. Os assuntos abordados em cada capítulo podem ser resumidos da seguinte forma:

- Capítulo 2 – "Dispositivos Lógicos Programáveis" – descreve resumidamente os vários tipos de dispositivos lógicos programáveis que podem ser utilizados para implementar sistemas digitais. A descrição abrange as PROMs, as PLAs, as PALs, os CPLDs e as FPGAs, bem como as respectivas tecnologias de programação. Seguidamente é feita uma comparação, em termos de vários parâmetros, entre os dispositivos programáveis pelo utilizador e os programados por máscara. Na última parte deste capítulo é apresentado um resumo das características mais importantes da família de FPGAs reconfiguráveis dinâmica e parcialmente utilizada neste

trabalho, a XC6200 da Xilinx. Esta discussão abrange, entre outros aspectos, a arquitectura e as ferramentas de desenvolvimento.

- Capítulo 3 – "Modelos e Arquitecturas de Unidades de Controlo" – apresenta vários modelos e arquitecturas que podem ser utilizados no projecto de unidades de controlo. Quanto aos primeiros, são descritos alguns exemplos de modelos orientados ao estado, uma vez que são os mais apropriados para descrever o comportamento de unidades ou circuitos de controlo. Nomeadamente, são apresentadas a máquina de estados finitos, a máquina de estados finitos com unidade de execução, as redes de Petri, a máquina de estados finitos hierárquica e/ou paralela e a máquina de estados finitos virtual. Quanto às arquitecturas são apresentadas as seguintes: controlador, controlador com unidade de execução, controlador hierárquico e controlador virtual. Finalmente, são também abordados alguns aspectos relacionados com a sincronização de controladores hierárquicos.
- Capítulo 4 – "Especificação de Unidades de Controlo" – começa por fazer o levantamento das características mais importantes que as linguagens de especificação devem possuir para suportarem eficientemente a descrição comportamental de unidades de controlo de qualquer complexidade. Seguidamente são apresentadas algumas das linguagens mais utilizadas, nomeadamente, os métodos tradicionais baseados em diagramas e tabelas de transição de estados, as máquinas de estado algorítmicas, os esquemas de grafos e as suas variantes hierárquicas e paralelas, os *Statecharts* e, finalmente, a linguagem de descrição de hardware VHDL.
- Capítulo 5 – "Síntese de Unidades de Controlo" – descreve resumidamente o processo de síntese de unidades de controlo configuradas fisicamente (*hardwired*). Sendo as FPGAs dispositivos lógicos multi-nível, é dada especial atenção às técnicas mais apropriadas para este tipo de implementação. Em particular, são descritas resumidamente as etapas de codificação das entradas e saídas, minimização de estados, codificação de estados e optimização lógica. Seguidamente, são ilustrados os processos de síntese manual de uma unidade de controlo e de optimização automática da sua componente combinatória. Os resultados da síntese com ferramentas de projecto assistido por computador, usando várias técnicas de codificação de estados, são também apresentados. Finalmente, é descrito um fluxo de projecto de unidades de controlo orientado para implementação em FPGAs da família XC6200.
- Capítulo 6 – "Projecto de Unidades de Controlo Virtuais" – começa por referir algumas das aplicações dos sistemas reconfiguráveis e por enquadrar as unidades de controlo virtuais neste tipo de sistemas. Seguidamente é apresentado o fluxo de projecto deste tipo de circuitos, com especial ênfase para as etapas de especificação, síntese e implementação. A placa de desenvolvimento *FireFly*™ utilizada neste trabalho é também aqui descrita. A arquitectura concebida para implementação de unidades de controlo virtuais na FPGA XC6200 e que permite alcançar os objectivos apresentados na secção 1.3 é apresentada neste capítulo. Finalmente é

descrita uma biblioteca VHDL de componentes parametrizáveis construída para facilitar o projecto de circuitos que utilizem a família XC6200 como tecnologia de implementação.

- Capítulo 7 – "Software para Controlo da Reconfiguração Dinâmica" – é dedicado à apresentação das ferramentas de software desenvolvidas para suportar o acesso ao hardware utilizado neste trabalho e a reconfiguração dinâmica dos circuitos construídos e implementados nas FPGAs da família XC6200. Mais concretamente, é apresentada uma biblioteca de classes escrita em C++ para processamento dos ficheiros produzidos pelas ferramentas de implementação e para (re)configuração dos dispositivos da família XC6200. Por último, são descritas a estrutura e a funcionalidade de um controlador de software (*device driver*) para a placa de desenvolvimento *FireFly*™ utilizada neste trabalho.
- Capítulo 8 – "Conclusões e Trabalho Futuro" – apresenta as conclusões, resume as contribuições originais resultantes deste trabalho e descreve algumas ideias para trabalho futuro.

Quanto aos anexos, o seu conteúdo é o seguinte:

- Anexo I – "Listagens das PLAs e Equações Multi-nível" – apresenta as listagens das descrições iniciais e optimizadas da parte combinatória das unidades de controlo utilizadas no capítulo 5 para ilustrar a optimização lógica automática de um circuito com as ferramentas *Espresso* e *misII*.
- Anexo II – "Diagramas Esquemáticos" – mostra os diagramas esquemáticos das unidades de controlo do capítulo 5 sintetizadas com as ferramentas *Leonardo Spectrum* e *Synopsys*.
- Anexo III – "Listagem da Biblioteca para Mapeamento na Tecnologia XC6200 da Xilinx" – apresenta a biblioteca "xc6200.genlib" construída para realizar o mapeamento na tecnologia XC6200 da Xilinx dos circuitos sintetizados com a ferramenta SIS. O fluxo de projecto baseado nesta ferramenta está descrito no capítulo 5.
- Anexo IV – "Listagens dos Módulos VHDL" – lista o código VHDL que descreve a arquitectura de unidades de controlo virtuais desenvolvida no âmbito deste trabalho e descrita no capítulo 6.
- Anexo V – "Código Fonte do Controlador da Placa de Desenvolvimento *FireFly*™" – lista o código fonte do controlador de software construído para a placa de desenvolvimento *FireFly*™ cuja estrutura é descrita resumidamente no capítulo 7.

A dissertação termina com as listas de referências e de acrónimos.

2 Dispositivos Lógicos Programáveis

Sumário

Este capítulo faz uma introdução aos dispositivos lógicos programáveis, os quais constituíram a tecnologia de implementação usada na componente prática do trabalho descrito nesta dissertação.

Em primeiro lugar, são apresentadas as escalas de integração (ou capacidades lógicas equivalentes) e os graus de programação nos quais são normalmente divididos os vários tipos de dispositivos que podem ser utilizados na implementação de sistemas digitais. Seguidamente, os dispositivos programáveis e não programáveis são classificados em subcategorias em função do seu método de construção, arquitectura básica e complexidade.

De seguida são descritas as tecnologias de programação utilizadas na construção dos dispositivos lógicos programáveis, nomeadamente, os fusíveis, os antifusíveis, os transístores de porta flutuante e as células de memória RAM estática. As secções seguintes descrevem as arquitecturas típicas dos três tipos de dispositivos mais comuns: os dispositivos lógicos programáveis elementares (PROMs, PLAs, PALs e GALs), os dispositivos lógicos programáveis complexos (CPLDs) e os agregados de células lógicas programáveis (FPGAs). Como as FPGAs são o tipo mais recente de dispositivo lógico programável, o que possui maiores capacidades e aquele que foi utilizado neste trabalho é-lhes atribuída uma atenção especial. Apesar dos dispositivos programáveis possuírem muitas vantagens sobre os não programáveis, também possuem os seus inconvenientes pelo que é feita uma análise comparativa das duas tecnologias.

Finalmente, é realizada uma descrição resumida da arquitectura da FPGA utilizada neste trabalho, a XC6200 da Xilinx. Em particular são apresentadas as suas características gerais, a estrutura, a constituição das células e da unidade funcional, os recursos de interligação e os registos de controlo. São também fornecidas algumas recomendações a ter em conta na implementação de projectos nesta família a fim de melhorar a qualidade final dos resultados obtidos.

2.1 Introdução

Um projectista de sistemas electrónicos digitais tem à sua disposição várias tecnologias para efectuar a sua implementação. Dois aspectos importantes na classificação destas tecnologias são a capacidade lógica implementável num único circuito integrado e o seu grau de programação. Quanto à primeira, a classificação divide-se normalmente em quatro categorias:

- *Small-Scale Integration* (SSI) – dispositivos contendo até 10 portas lógicas num único encapsulamento;
- *Medium-Scale Integration* (MSI) – dispositivos contendo entre 10 e 100 portas lógicas num único encapsulamento;
- *Large-Scale Integration* (LSI) – dispositivos contendo entre 100 e alguns milhares de portas lógicas num único encapsulamento;
- *Very-Large-Scale Integration* (VLSI) – dispositivos contendo centenas de milhar ou milhões de portas lógicas num único encapsulamento;

Se a função de um dispositivo for definida durante a sua fabricação, não podendo ser alterada pelo utilizador, diz-se que o dispositivo não é programável. Se por outro lado o dispositivo possuir uma funcionalidade genérica ou modificável, ou seja, se a sua função não for especificada durante a fabricação, sendo realizada à posteriori pelo utilizador[1], diz-se que o dispositivo é programável. Este tipo de dispositivos são também designados por dispositivos programáveis pelo utilizador ou no campo (*Field Programmable Devices – FPDs*). Um FPD é um circuito integrado genérico cuja função é definida pelo utilizador através do uso de ferramentas de desenvolvimento e programação adequadas. As vantagens principais dos FPDs relativamente aos dispositivos não programáveis são os reduzidos custos de projecto, a flexibilidade e a facilidade de reconversão, o que permite a sua utilização noutros projectos. Estas vantagens resultam da sua programação ser efectuada pelo utilizador.

A programação do dispositivo pode ser efectuada através de equipamento apropriado ou então no próprio sistema onde vai ser inserido. No segundo caso o dispositivo pode também ser designado por programável no circuito ou no sistema. Finalmente, se a funcionalidade do dispositivo puder ser alterada mais do que uma vez pelo utilizador, diz-se que é reprogramável.

Os dispositivos electrónicos integrados podem ser analógicos, digitais ou mistos (*mixed-signal*). No entanto, neste capítulo vamos considerar apenas os dispositivos lógicos utilizados na implementação de sistemas digitais.

As tecnologias mais comuns para implementação de sistemas digitais são os dispositivos lógicos não programáveis SSI/MSI, os dispositivos lógicos programáveis elementares (*Simple Programmable Logic Devices – SPLDs*), os dispositivos lógicos programáveis complexos (*Complex Programmable Logic Devices – CPLDs*), os circuitos integrados específicos da aplicação (*Application Specific Integrated Circuits – ASICs*) e os agregados de células lógicas programáveis (*Field Programmable Gate Arrays – FPGAs*) [Maxfield96, BroRos96, Trimberger94]. Em termos de capacidade lógica os ASICs são

[1] Neste contexto, o utilizador é aquele que usa o dispositivo para projectar um sistema.

geralmente dispositivos LSI ou VLSI. Os SPLDs, os CPLDs são dispositivos MSI e LSI respectivamente. As FPGAs começaram por ser dispositivos LSI, mas as mais recentes são já VLSI.

Os circuitos integrados SSI/MSI referidos acima e os ASICs são exemplos de dispositivos não programáveis. Por outro lado, os SPLDs, os CPLDs e as FPGAs são dispositivos lógicos programáveis pelo utilizador ou programáveis no campo (*Field Programmable Logic Devices – FPLDs*). A tecnologia mais apropriada para um sistema em particular depende de vários factores, entre eles a sua complexidade, o tempo de desenvolvimento pretendido e o número de unidades do sistema que se espera produzir [ChaMou94].

Na Figura 2.1 estão ilustradas as subcategorias em que os ASICs e os FPLDs se podem dividir.

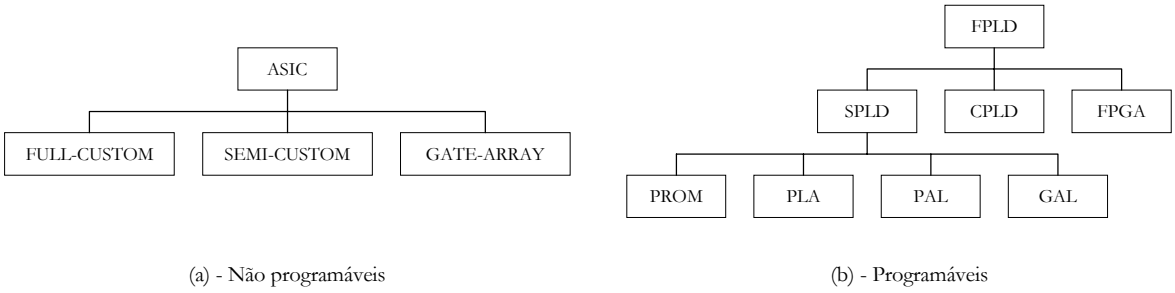


Figura 2.1 – Classificação dos circuitos integrados utilizados em sistemas digitais; (a) ASICs, (b) FPLDs e respectivas subcategorias.

Os ASICs e os FPLDs sofisticados mais recentes alteraram de forma significativa o projecto de sistemas digitais. Ao contrário das gerações anteriores de hardware em que os circuitos consistiam numa ou mais placas de circuito impresso contendo vários circuitos integrados SSI ou MSI com portas lógicas básicas e unidades funcionais e de armazenamento, tais como somadores e registos, a maioria dos sistemas actuais consistem em dispositivos de elevada densidade. Isto é verdade não só para circuitos complexos como processadores e memória mas também para circuitos mais simples como máquinas de estado, contadores, registos e descodificadores.

Para pequenas quantidades de lógica, a implementação pode ainda ser facilmente realizada com dispositivos lógicos SSI ou MSI. Cada um possui respectivamente algumas portas lógicas do mesmo tipo ou componentes com complexidade análoga à de um registo. No entanto, com a crescente complexidade dos sistemas e a existência de tecnologias que permitem elevados níveis de integração, este tipo de abordagem de implementação é cada vez menos utilizada.

Um SPLD é um dispositivo de uso geral, com uma capacidade lógica equivalente a dezenas ou centenas de circuitos SSI. A maioria dos SPLDs permite implementar funções lógicas na forma de soma de produtos, ou seja, com dois níveis de portas lógicas. Devido à sua estrutura, os SPLDs não são escaláveis e consequentemente desapropriados para a implementação de circuitos complexos ou com um elevado número de entradas ou saídas. Além disso, não permitem partilhar portas lógicas para implementar produtos ou somas com variáveis comuns, isto é, se dois termos de uma função lógica diferirem pelo menos numa variável, é necessário

realizar o seu cálculo separadamente. Este facto tem consequências negativas ao nível da área do circuito e do seu consumo de potência.

Os CPLDs são uma extensão dos SPLDs e permitem resolver o problema da escalabilidade através da integração de vários SPLDs num único circuito integrado. No entanto, o consumo de potência e os elevados requisitos de interligação no caso de um número elevado de blocos, limitam a sua dimensão, para além de que não resolvem o problema da partilha de recursos no cálculo de diferentes funções lógicas. Assim, para projectos complexos são necessários dispositivos escaláveis multi-nível, tais como as FPGAs.

Para implementar sistemas com milhares ou milhões de portas lógicas e destinados a mercados de grande dimensão são utilizados ASICs de elevada densidade. Um ASIC é concebido pelo respectivo fabricante para desempenhar uma dada função, não podendo esta ser alterada após a sua fabricação. O projecto de um ASIC é realizado normalmente ao nível do circuito ou ao nível lógico. Em ambos os casos, consiste basicamente na determinação das características dos seus componentes, essencialmente transístores e portas lógicas e na sua implantação e interligação na pastilha do circuito integrado. Este processo pode ser baseado em vários métodos (Figura 2.1), nomeadamente, completamente especializado (*full-custom*), semi-especializado (*semi-custom*) e em agregados de portas lógicas (*gate-array*). Os diversos métodos diferem no nível ao qual é realizado o projecto, no tipo de módulos primitivos utilizados e na flexibilidade com que esses módulos podem ser implantados na pastilha do circuito integrado.

O primeiro estilo desenvolvido para projecto de circuitos integrados VLSI foi o *full-custom*, o qual utiliza componentes ao nível do circuito, tais como transístores e conexões, como elementos primitivos. Desde que sejam respeitadas determinadas regras dependentes da tecnologia, o projectista tem liberdade total no que respeita à implantação e interligação destes componentes na pastilha do circuito integrado. A determinação das suas características, tais como velocidade, área e consumo de potência, é realizada ao nível do circuito. Este estilo de projecto é o mais flexível e o que permite maiores optimizações das características atrás referidas. No entanto, para circuitos complexos requer um tempo de projecto elevado e um conhecimento aprofundado da tecnologia e da operação dos componentes ao nível do circuito.

Para facilitar o processo de projecto foram introduzidos outros métodos de projecto conceptualmente mais simples mas também mais limitativos. A simplificação consegue-se com a realização do projecto a níveis de abstracção mais elevados normalmente com a utilização de primitivas mais complexas e pela restrição da forma como estas podem ser implantadas na pastilha de um circuito integrado.

No estilo *semi-custom*, uma biblioteca de células standard (*standard-cells*) é projectada ao nível lógico. Estas células são especificadas pela sua função e pelas suas características físicas. A utilização destas células ao nível lógico simplifica consideravelmente o processo de projecto, mas reduz a flexibilidade e o grau de optimização que se pode alcançar. Nalguns casos é utilizada uma combinação de *full-custom* com *semi-custom*, sendo as partes críticas do circuito projectadas em *full-custom*.

Finalmente, no método *gate-array*, implantam-se previamente na pastilha do circuito integrado e segundo uma estrutura regular, componentes básicos

(normalmente portas lógicas). O projecto consiste na definição das conexões entre estas portas lógicas de forma a implementar a funcionalidade desejada. Um dispositivo projectado segundo esta abordagem é também vulgarmente designado por MPGA (*Mask-Programmable Gate Array*) ou MPLD (*Mask-Programmable Logic Device*) porque a sua especialização para uma dada aplicação é realizada através de máscaras durante o processo de fabrico. As máscaras são usadas para criar as camadas de metalização responsáveis pela interligação das portas lógicas predefinidas. A Figura 2.2 ilustra uma secção de um *gate-array* constituída por portas lógicas NAND de três entradas rodeadas por canais verticais e horizontais de interligação.

Para além dos elevados tempos de projecto, construção e teste de um ASIC, os custos envolvidos na preparação das máscaras fazem com que estes sejam uma tecnologia de implementação competitiva somente para grandes volumes de fabricação.

Por outro lado, as FPGAs combinam as vantagens das MPGAs com as dos dispositivos programáveis. Tal como as MPGAs, as FPGAs integram num único circuito milhares ou milhões de portas lógicas numa estrutura multi-nível. Tal como os SPLDs e CPLDs, as FPGAs são programáveis pelo utilizador, reduzindo o tempo de desenvolvimento e o custo dos sistemas destinados a mercados de pequena dimensão. Além disso, são também apropriadas para o desenvolvimento rápido de protótipos.

No entanto, como veremos mais à frente, comparativamente com os ASICs, as FPGAs também possuem algumas desvantagens, em particular ao nível do desempenho e da densidade lógica, uma vez que os circuitos que controlam a sua programação contribuem para uma diminuição da velocidade máxima do circuito e ocupam uma área que não pode ser utilizada pelos componentes do sistema a implementar no dispositivo.

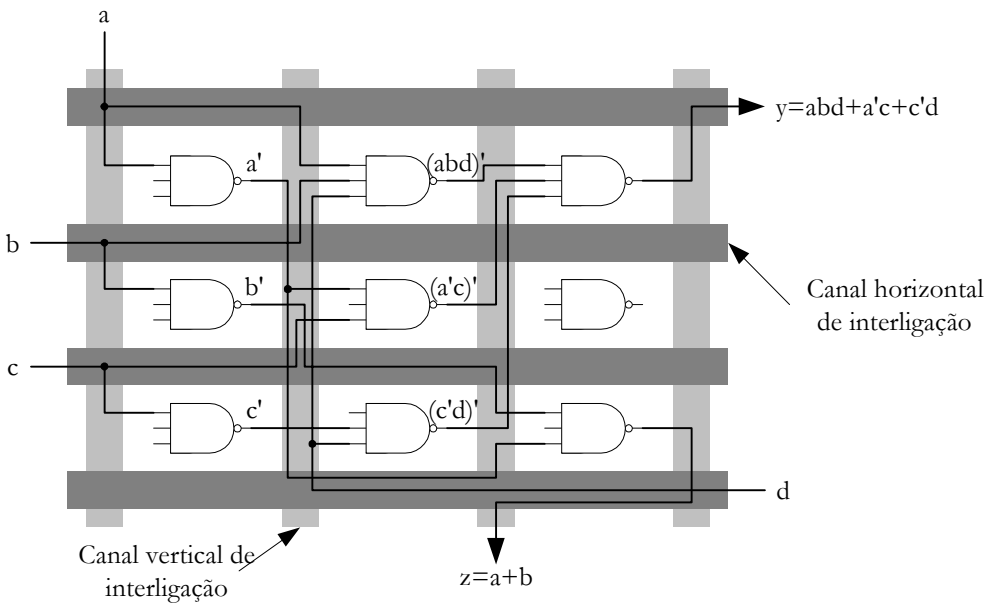


Figura 2.2 – Exemplo de um *gate-array*.

Na última década assistiu-se a uma forte evolução do mercado de FPLDs, existindo actualmente uma grande gama de produtos disponíveis. A Figura 2.3 ilustra a

variação das capacidades lógicas de cada categoria de FPLD em termos de portas lógicas equivalentes. Esta medida fornece uma estimativa do número de portas NAND de 2 entradas implementáveis num dado dispositivo. De notar que este valor é apenas uma aproximação, sendo por vezes manipulado pelos fabricantes de FPLDs para servir os seus próprios interesses. O diagrama serve de guia para seleccionar um tipo de dispositivo em particular de acordo com a capacidade lógica necessária para o sistema a projectar. No entanto, a escolha de um dado dispositivo depende também da adequação das suas características ao projecto em causa. Como iremos ver mais à frente, cada tipo de FPLD possui os seus campos de aplicação preferenciais. Por vezes a decisão é também condicionada pelas potencialidades das ferramentas de software utilizadas no projecto e implementação do circuito num dado dispositivo.

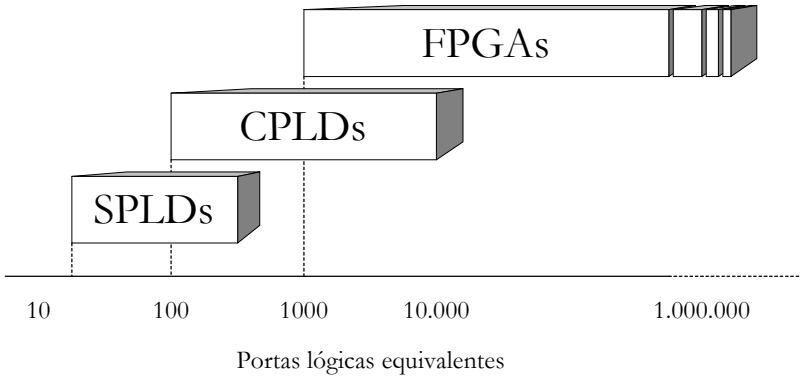


Figura 2.3 – Capacidades dos três tipos mais comuns de FPLDs em termos de portas lógicas equivalentes.

2.2 Tecnologias de Programação

A programação dos FPLDs pelos utilizadores é possível graças à utilização de vários componentes, cujo comportamento pode ser modificado após a fabricação do dispositivo. Exemplos destes componentes são o interruptor programável, o multiplexador e a tabela de verdade. As tecnologias de programação são o conjunto das técnicas físicas empregues na sua construção, sendo as mais comuns baseadas em fusíveis, antifusíveis, células e transístores EPROM (*Erasable Programmable Read Only Memory*) e EEPROM (*Electrically Erasable Programmable Read Only Memory*) e células SRAM (*Static Random Access Memory*). As próximas secções descrevem sucintamente cada uma destas tecnologias de programação.

2.2.1 Fusíveis

O primeiro interruptor programável desenvolvido foi o fusível usado nas PLAs. Apesar dos dispositivos de pequena capacidade ainda continuarem a usá-los, as novas tecnologias estão a substituí-los rapidamente. Os dispositivos baseados em fusíveis possuem um funcionamento semelhante aos fusíveis de protecção contra sobre-intensidades de corrente, nos quais através da aplicação duma corrente excessiva é provocada uma alteração acentuada das suas características eléctricas. As duas

variantes principais desta tecnologia baseiam-se em fusíveis laterais e verticais. Um fusível lateral consiste num fio construído com uma liga de tungsténio e titânio em série com um transístor bipolar (*Bipolar Junction Transistor – BJT*). A programação é efectuada através da aplicação de uma corrente suficientemente elevada para fundir o fio. Este tipo de ligação é inicialmente um curto-circuito, tornando-se num circuito aberto após a sua programação. Por outro lado, o díodo da junção base-emissor dum BJT constitui um fusível vertical. Este tipo de ligação começa por ser um circuito aberto porque o BJT comporta-se como dois díodos ligados em anti-série, impedindo o fluxo de corrente. No entanto, através da aplicação duma sequência de impulsos de corrente suficientemente elevados através do emissor do BJT, ocorre o efeito de avalanche o que provoca a fusão e desaparecimento da junção de emissor e a criação de um curto-circuito.

2.2.2 Antifusíveis

Em dispositivos de elevada densidade a tecnologia CMOS domina a indústria dos circuitos integrados tornando-se necessárias outras abordagens para a implementação de interruptores programáveis. Assim, como alternativa aos fusíveis, alguns FPLDs (predominantemente CPLDs e FPGAs) empregam a tecnologia antifusível na construção dos interruptores programáveis. Os antifusíveis são fabricados utilizando uma tecnologia CMOS modificada e comportam-se inicialmente como circuitos abertos que se transformam numa ligação de baixa resistência depois de programados. A sua estrutura é composta por um canal de silício amorfo entre duas camadas de metalização (Figura 2.4). Inicialmente (antes da programação) o silício amorfo é um isolador com uma resistência superior a 1GΩ. O utilizador pode programar a ligação antifusível através da aplicação de sinais de corrente relativamente elevada (cerca de 20 mA) às entradas do dispositivo. Esta corrente cria uma ligação através da transformação de silício amorfo isolador em polissilício condutor.

As duas vantagens principais desta tecnologia de programação são as dimensões reduzidas de um antifusível e a sua pequena resistência eléctrica depois de programado.

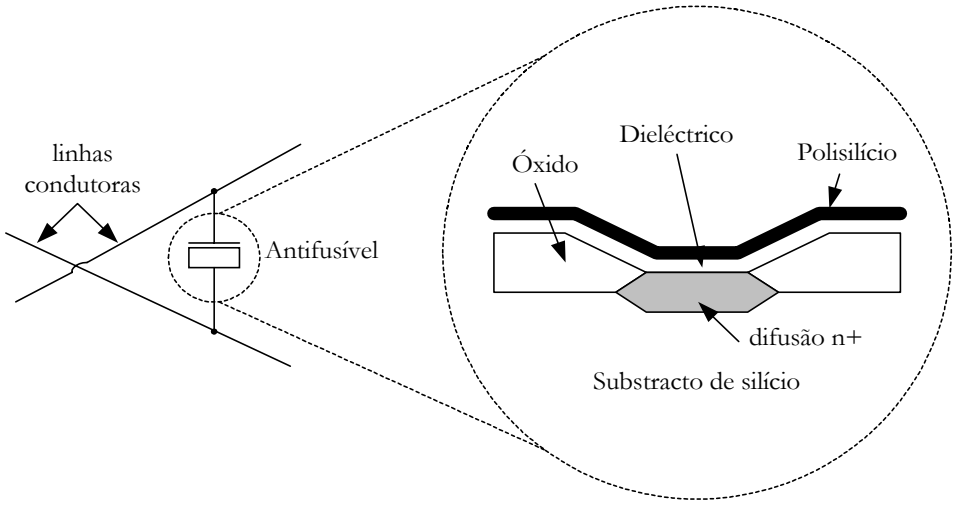


Figura 2.4 – Constituição de um antifusível.

Os dispositivos baseados nas tecnologias fusível e antifusível são programáveis apenas uma vez, normalmente conhecidos por OTP - *one-time programmable*, já que o processo de programação é irreversível, possibilitando apenas a modificação das ligações ainda não programadas. Por outro lado, são também não voláteis, ou seja, retêm a sua programação mesmo na ausência de tensão de alimentação.

2.2.3 Transístores de Porta Flutuante - EPROM e EEPROM

Outra tecnologia utilizada para construir interruptores programáveis baseia-se em transístores de porta flutuante, semelhantes aos usados em memórias EPROM e EEPROM. Esta tecnologia é mais utilizada nos CPLDs. Para permitir a implementação de funções *wired-AND*, os transístores são colocados entre as linhas de entrada e de saída duma matriz do CPLD, tal como ilustrado na Figura 2.5. Se uma entrada fizer parte do produto calculado numa dada linha de saída, pode controlar o seu valor através da activação do respectivo transístor. Os transístores relativos às restantes entradas são bloqueados por programação para que não afectem o valor da saída. No caso de transístores EEPROM, o esquema é semelhante ao da Figura 2.5.

Os dispositivos baseados nestas tecnologias são reprogramáveis e não voláteis. A diferença entre as duas tecnologias é que enquanto a programação dos dispositivos baseados em EEPROM pode ser feita no circuito, a dos dispositivos EPROM deve ser realizada em equipamento de programação específico.

Apesar das tecnologias de programação EPROM e EEPROM combinarem a não volatilidade com a reprogramabilidade, não são normalmente utilizadas na construção de FPGAs porque requerem um processo de fabricação mais complexo e as células ocupam mais espaço, sendo este um factor muito importante em dispositivos de elevada capacidade como as FPGAs. As FPGAs disponíveis comercialmente utilizam na sua generalidade como tecnologia de programação células SRAM ou antifusíveis.

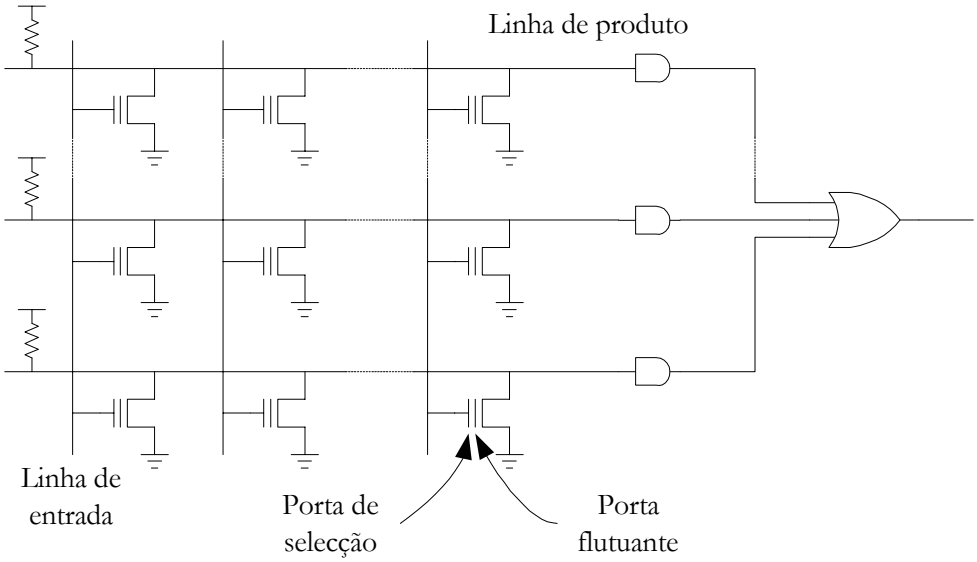


Figura 2.5 – Implementação de funções lógicas AND com transístores EPROM.

2.2.4 Células de Memória Estática - SRAM

A tecnologia de programação SRAM utiliza células de memória para configurar os blocos lógicos e os elementos responsáveis pelo controlo dos recursos de interligação do FPLD. A configuração de um dispositivo baseado em SRAM é realizada tipicamente através de um ou vários dos elementos seguintes:

- Interruptor programável – um transístor controlado por uma célula SRAM ligada à sua porta actua como um interruptor que pode ser usado para estabelecer as ligações entre duas linhas condutoras (Figura 2.6 (a));
- Multiplexador programável – num multiplexador de 2^K entradas controlado por K linhas de selecção, o estado das células SRAM determina qual das entradas é ligada à saída (Figura 2.6 (b)).
- Tabela de verdade – uma matriz de 2^K células SRAM endereçadas por K entradas implementa uma função lógica de K variáveis (Figura 2.6 (c)).

Uma vez que a configuração do dispositivo é baseada em células de memória SRAM, a sua programação pode ser realizada um número ilimitado de vezes. A reprogramabilidade permite ao fabricante do dispositivo testar todos os seus elementos e interligações através da realização de vários ciclos de programação em equipamento de teste apropriado. Como resultado, os utilizadores adquirem dispositivos completamente testados e em que a taxa de sucesso de programação é normalmente de 100%, tudo isto sem a necessidade de vectores de teste específicos da aplicação ou de técnicas de projecto que facilitem o teste do circuito. Ao nível do sistema, a reprogramabilidade pode ser utilizada para implementar no mesmo dispositivo reprogramável diferentes circuitos que não sejam necessários simultaneamente. De acordo com a operação do sistema, os diferentes circuitos vão sendo carregados a partir de um dispositivo de configuração externo. A reprogramabilidade permite também a realização de protótipos de forma interactiva, reutilizando o mesmo dispositivo nas várias iterações de projecto.

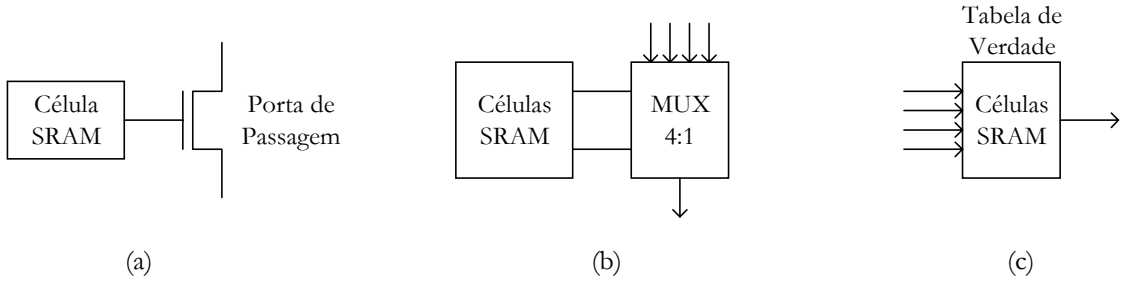


Figura 2.6 – Elementos programáveis controlados por células SRAM - (a) interruptor; (b) multiplexador; (c) tabela de verdade.

A maior desvantagem da tecnologia de programação SRAM é a elevada área que ocupa. Cada interruptor programável necessita de pelo menos cinco transístores para implementar a célula de memória SRAM e um para o interruptor propriamente dito. Na Figura 2.7 está ilustrada uma célula de memória típica constituída por cinco transístores. De notar que normalmente não existe uma área separada da pastilha do circuito integrado para as células de memória de configuração, estando estas

distribuídas juntamente com os elementos lógicos controlados. Uma vez que a memória de configuração não é em geral modificada frequentemente durante o funcionamento do circuito, a sua construção contempla a estabilidade e a densidade em vez da velocidade de operação. A tecnologia de programação SRAM possui duas grandes vantagens, a reprogramação rápida e o facto de poderem ser construídos com processos standard de fabricação de circuitos integrados.

Sendo a SRAM volátil, o dispositivo deve ser configurado cada vez que o sistema for ligado. Isto requer uma memória externa permanente para armazenar a informação de programação, normalmente do tipo PROM, EPROM ou EEPROM. Por este motivo, os dispositivos baseados em SRAM possuem lógica adicional para detectar o arranque do circuito e para se inicializarem automaticamente. A programação do dispositivo demora normalmente alguns milissegundos.

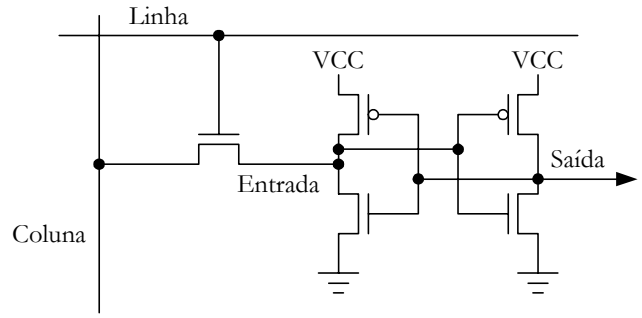


Figura 2.7 – Célula de memória SRAM com cinco transístores.

A Tabela 2.1 resume as características mais importantes do ponto de vista de utilização das tecnologias de programação empregues na construção dos FPLDs.

Tipo do interruptor	Tecnologia	Reprogramável	Volátil
Fusível	Bipolar	Não	Não
Antifusível	CMOS+	Não	Não
EPROM	UVCMOS	Sim (em equipamento específico)	Não
EEPROM	EECMOS	Sim (no circuito)	Não
SRAM	CMOS	Sim (no circuito)	Sim

Tabela 2.1 - Resumo das tecnologias de programação mais utilizadas em dispositivos lógicos programáveis.

2.3 Dispositivos Lógicos Programáveis Elementares

Os dispositivos lógicos programáveis elementares *(Simple Programmable Logic Devices – SPLD*) podem ser decompostos em quatro subcategorias: PROMs, PLAs, PALs e GALs. Estes dispositivos possuem geralmente uma pequena capacidade lógica, baixo custo e reduzidos tempos de propagação entre pinos. A sua estrutura interna é composta por duas matrizes de portas lógicas ou elementos programáveis (Figura 2.8). Os factores que distinguem as diferentes subcategorias são o número de matrizes

programáveis, o tipo de elementos utilizados na construção de cada matriz e a sua reprogramabilidade.

Para além da sua funcionalidade básica os SPLDs podem possuir outras características, tais como saídas com registo, *tri-state* ou realimentadas para as entradas. No primeiro caso, pode também ser possível programar o tipo de registo (*latch* S-R/D ou *flip-flop* D/T/JK). Adicionalmente, alguns dispositivos permitem programar os pinos como entradas, saídas ou bidireccionais. Na Figura 2.8 está ilustrada a estrutura geral de um SPLD.

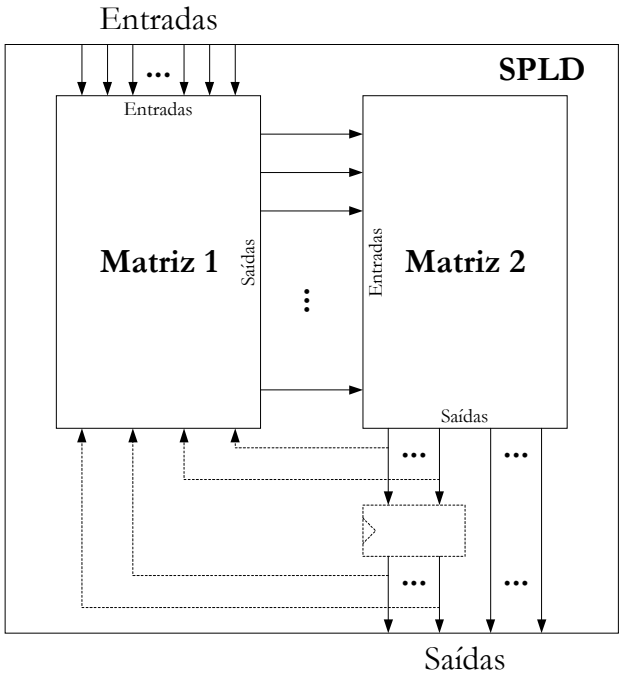


Figura 2.8 – Estrutura geral de um SPLD.

2.3.1 PROMs

O primeiro circuito integrado programável utilizado na implementação de circuitos digitais foi a memória só de leitura programável (*Programmable Read Only Memory - PROM*). Apesar de serem memórias, as PROMs são também FPLDs no sentido clássico, porque podem ser programadas pelo utilizador e usadas como tabelas de verdade de funções lógicas. No entanto, a sua programação pode ser realizada apenas uma vez. As linhas de endereço são utilizadas como entradas do circuito e as linhas de dados como saídas. A arquitectura interna de uma PROM é semelhante a um descodificador que alimenta um plano OR programável (Figura 2.9), podendo também ser vistas como uma matriz AND predefinida, seguida de uma matriz OR programável.

As PROMs são particularmente úteis para implementar funções com um grande número de produtos. No entanto, como as suas entradas são completamente descodificadas e a maior parte das funções lógicas numa das suas formas mais usuais, a soma de produtos, raramente possui muitos termos, as PROMs não são eficientes para implementar circuitos lógicos sendo portanto pouco utilizadas para este fim.

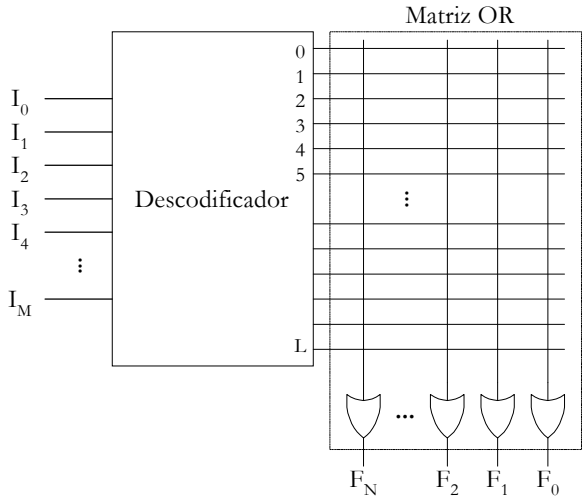


Figura 2.9 – Estrutura interna de uma PROM.

2.3.2 PLAs

O primeiro dispositivo desenvolvido especificamente para implementar circuitos lógicos foi a matriz lógica programável (*Programmable Logic Array – PLA*), introduzida pela Philips no início dos anos 70. Uma PLA consiste em dois níveis de portas lógicas, um plano AND seguido de um plano OR, ambos programáveis (Figura 2.10). Cada saída da matriz AND pode ser utilizada para calcular o produto entre quaisquer entradas ou os seus complementos. De forma análoga, cada saída da matriz OR pode ser utilizada para calcular o soma entre quaisquer saídas da matriz AND. Esta estrutura faz com que as PLAs sejam apropriadas para implementar funções lógicas na forma de soma de produtos, sendo também bastante versáteis uma vez que tanto os termos AND como OR podem possuir muitas entradas. Numa PLA, o número de funções AND é independente do número de entradas e o número de funções OR é independente do número de entradas e do número de funções AND.

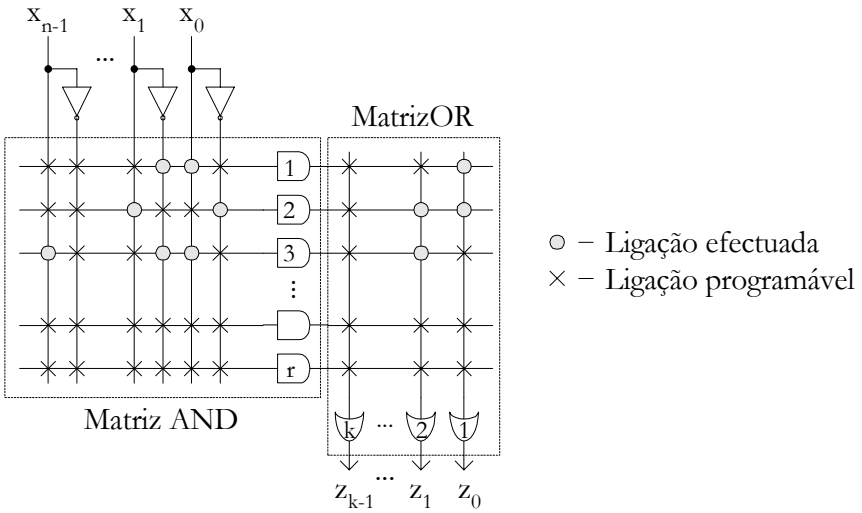


Figura 2.10 – Estrutura interna de uma PLA.

Tal como referido acima, nada obriga a que uma matriz AND seja seguida de uma matriz OR. Alguns dispositivos possuem duas matrizes NAND/NOR ou uma combinação dos dois tipos.

2.3.3 PALs

As PLAs quando surgiram, tinham alguns problemas, nomeadamente o seu elevado custo de fabricação e baixo desempenho. Ambos são devidos aos dois níveis de lógica programável que eram difíceis de fabricar e introduziam atrasos de propagação significativos. Além disso, muitas aplicações não necessitam que ambas as matrizes AND e OR sejam programáveis. Para ultrapassar estes problemas, a *Monolithic Memories* desenvolveu e propôs as PALs (*Programmable Array Logic*). Tal como ilustrado na Figura 2.11, uma PAL possui apenas um nível de lógica programável, sendo constituída por uma matriz AND programável seguida de uma matriz OR fixa. Para compensar a perda de generalidade imposta pela matriz OR fixa, as PALs estão disponíveis em diversos formatos com um número variável de entradas e saídas e portas OR de vários tamanhos. As PLAs são mais flexíveis do que as PALs, mas as últimas são mais rápidas porque as ligações fixas possuem um menor tempo de comutação do que as programáveis. Sendo mais rápidas e baratas, as PALs são o tipo mais frequente de SPLDs. Para implementar circuitos sequenciais, as PALs possuem normalmente flip-flops ligados às saídas da matriz OR. O aparecimento das PALs afectou profundamente o projecto de sistemas digitais, tendo também servido de base a algumas das arquitecturas recentes mais sofisticadas.

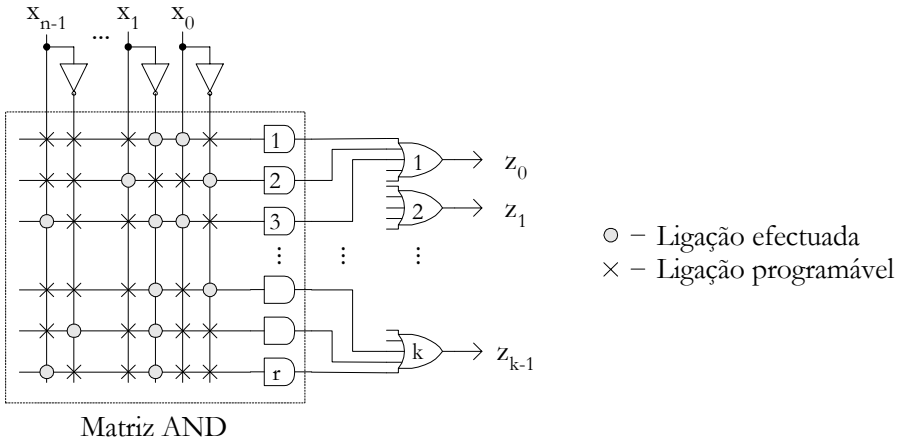


Figura 2.11 – Estrutura interna de uma PAL.

2.3.4 GALs

Os dispositivos GAL são versões sofisticadas de dispositivos lógicos programáveis e electricamente apagáveis (*EEPLD – Electrically Erasable Programmable Logic Device*) com uma estrutura semelhante às PALs mas com algumas características adicionais. Os dispositivos reprogramáveis possuem algumas vantagens sobre os baseados em fusíveis e antifusíveis, tais como a possibilidade de realização de testes mais rigorosos após a sua fabricação, através de um ou mais ciclos de programação

bem como a correcção de erros ou alterações do projecto através da reprogramação do dispositivo.

Existem também dispositivos que podem ser programados sem que seja necessária a remoção do circuito em que se encontram inseridos. Isto torna-se possível devido à utilização de memória de configuração do tipo SRAM ou FLASH.

2.4 Dispositivos Lógicos Programáveis Complexos

A evolução da tecnologia permitiu desenvolver dispositivos com maiores capacidades do que os SPLDs. No entanto, as arquitecturas dos SPLDs não são escaláveis porque o tamanho das matrizes programáveis cresce rapidamente à medida que o número de entradas aumenta. Assim, uma das formas possíveis para construir dispositivos de maior capacidade baseados em arquitecturas SPLD é incluir num único circuito integrado vários SPLDs interligados por um conjunto de conexões programáveis (Figura 2.12). Muitos dos FPLDs disponíveis actualmente possuem esta estrutura, sendo normalmente designados por dispositivos lógicos programáveis complexos (*Complex Programmable Logic Devices – CPLDs*). Estes dispositivos são bastante mais sofisticados do que um SPLD, por vezes também ao nível do bloco básico. As ligações programáveis podem ser baseadas em fusíveis, antifusíveis, transístores EPROM ou EEPROM ou células de memória SRAM.

Alguns CPLDs baseados em interruptores programáveis controlados por células SRAM permitem a utilização desta memória tanto para programação, como para blocos de memória utilizáveis pelo circuito lógico implementado no dispositivo, aumentando assim a sua versatilidade. As interconexões programáveis podem possuir dezenas a centenas de linhas, sendo impraticável a sua ligação a todos os blocos SPLD. Assim, o interface entre estes blocos e as interconexões é estabelecido através de multiplexadores programáveis.

Os CPLDs possuem tipicamente uma capacidade máxima equivalente a 50 SPLDs. Como é difícil estender estas arquitecturas para além deste limite, a construção de FPLDs com capacidades muito elevadas requer outras abordagens.

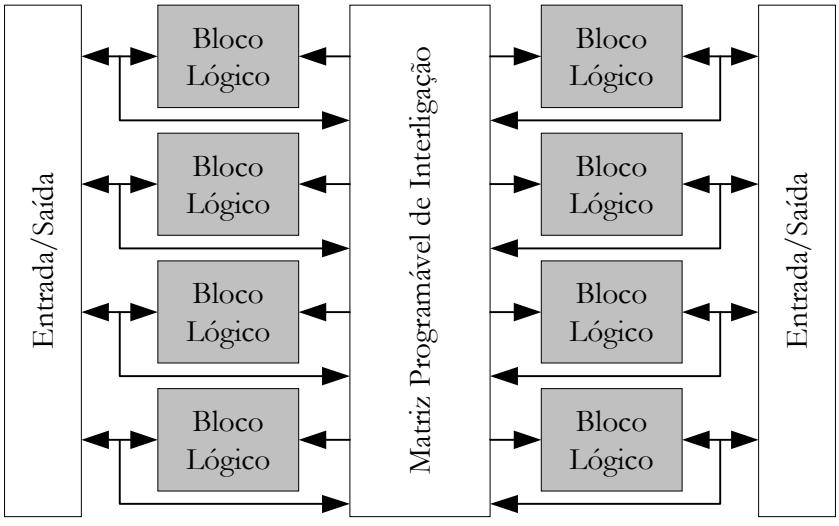


Figura 2.12 – Estrutura de um CPLD.

2.4.1 Aplicações do CPLDs

As suas elevadas velocidades de operação e uma grande gama de capacidades, tornam estes dispositivos úteis em muitas aplicações desde a implementação de lógica de interligação (*glue logic*) até à realização de protótipos de sistemas digitais simples. Um importante factor para o crescimento do mercado de CPLDs foi a conversão de projectos que utilizavam vários SPLDs em circuitos que utilizam um menor número de CPLDs. Os CPLDs podem ser utilizados para implementar projectos complexos tais como controladores gráficos, adaptadores de rede ou controladores de memória. Em geral, os circuitos que utilizem portas AND ou OR de muitas entradas e que não utilizem muitos flip-flops são bons candidatos para implementação em CPLDs. Um exemplo deste tipo de circuito são as máquinas de estados finitos. Uma vantagem importante da maioria dos CPLDs é sua reprogramabilidade permitindo alterações do projecto de forma simples. Nalguns casos, a reprogramação pode ser efectuada sem que seja preciso removê-lo do circuito, permitindo por exemplo trocar o protocolo de um equipamento de comunicações sem o desligar.

Normalmente, a partição de projectos em blocos tipo SPLD de um CPLD é feita de forma natural, produzindo circuitos com velocidades de operação bastante previsíveis, sendo esta uma das vantagens mais importantes das arquitecturas CPLD.

2.5 Agregados de Células Lógicas Programáveis

Os SPLDs e CPLDs são úteis para diversas aplicações, no entanto, a sua estrutura matricial de dois níveis limita a capacidade lógica destes dispositivos e consequentemente a complexidade dos circuitos neles implementáveis. Por outro lado, os ASICs são dispositivos muito genéricos e de elevada capacidade. Em particular, as MPGAs são os ASICs digitais disponíveis actualmente que posssuem um número mais elevado de componentes. Tal como descrito atrás, as MPGAs consistem numa matriz de portas lógicas pré-definidas em que a especialização para um dado circuito é realizada durante a fabricação através de máscaras para estabelecer a ligação entre as suas portas. As principais desvantagens das MPGAs são os elevados custos e tempo de desenvolvimento. Assim, do ponto de vista de tempo de desenvolvimento, grau de programação e capacidade existe uma grande lacuna entre SPLDs/CPLDs no limite inferior da capacidade e ASICs no limite superior.

Apesar das MPGAs não serem FPLDs, a sua estrutura serviu de motivação para o desenvolvimento de dispositivos equivalentes mas programáveis, as FPGAs. Estas surgiram nos meados dos anos 80 com o objectivo de preencher a lacuna existente entre os SPLDs/CPLDs e os ASICs. As FPGAs combinam várias características das MPGAs (ex. elevada capacidade e estrutura matricial multi-nível) com a possibilidade de programação após a fabricação.

De forma sucinta, uma FPGA é um circuito integrado VLSI, de uso geral e que pode ser programado pelo utilizador para implementar um sistema digital constituído por milhares ou milhões de portas lógicas. Ao contrário dos SPLDs que são dispositivos de dois níveis, as FPGAs permitem implementar lógica multi-nível e sistemas complexos num único circuito integrado. Uma FPGA consiste numa matriz composta por três tipos de elementos programáveis:

- Blocos lógicos contendo elementos combinatórios e/ou sequenciais;
- Recursos de interconexão controlados por elementos programáveis;
- Blocos de entrada/saída para interface com o exterior.

As várias arquitecturas de FPGAs diferem na forma como estes elementos são agrupados, na tecnologia de programação empregue no seu fabrico e na granulosidade dos blocos lógicos e de entrada/saída. A estrutura das arquitecturas de FPGAs disponíveis comercialmente pertence a uma das seguintes três categorias:

- Ilhas de blocos lógicos rodeados por recursos de interligação (Figura 2.13);
- Linhas de blocos lógicos e canais de interligação (Figura 2.14);
- Matriz de células também designada por *sea-of-gates* (Figura 2.15).

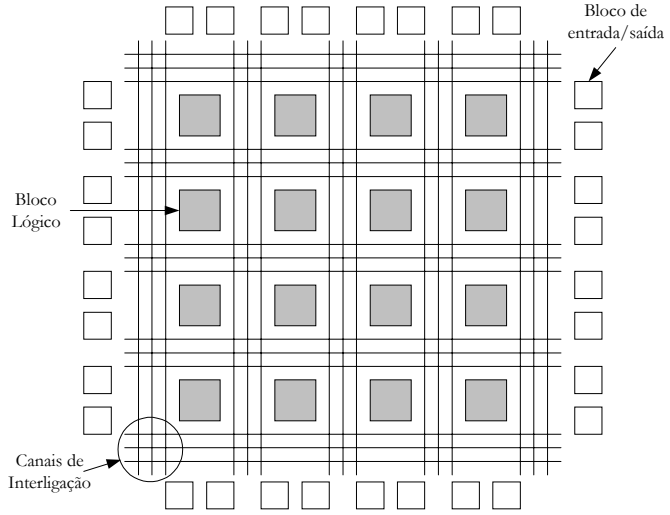


Figura 2.13 – Arquitectura de uma FPGA baseada em ilhas de blocos lógicos rodeados por recursos de interligação.

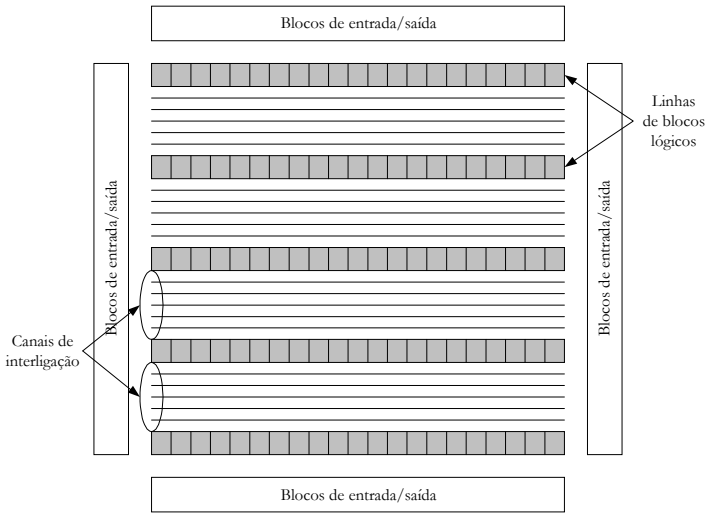


Figura 2.14 – Arquitectura de uma FPGA baseada em linhas de blocos lógicos e canais de interligação.

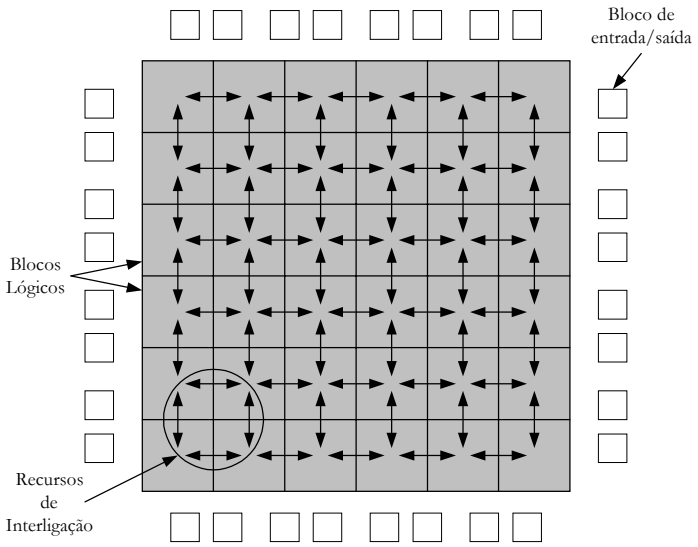


Figura 2.15 – Arquitectura de uma FPGA celular, também designada por *sea-of-gates*.

Enquanto os primeiros dois tipos possuem áreas apreciáveis destinadas a canais de interligação contendo um número considerável de linhas (>10), numa arquitectura celular, a comunicação é feita entre células adjacentes. No entanto, pode também existir um número reduzido de recursos de interligação com o comprimento de várias células permitindo efectuar de forma eficiente ligações mais longas.

A família XC4000 da Xilinx e a 54SX da Actel são exemplos de arquitecturas baseadas em ilhas e linhas de blocos lógicos respectivamente. A arquitectura XC6200 da Xilinx apresentada mais à frente é um exemplo de uma arquitectura celular.

Os recursos de interligação e os blocos de entrada/saída podem ser programados para realizar respectivamente as conexões internas e externas pretendidas.

Os blocos lógicos podem ser programados para implementar uma grande diversidade de funções lógicas. Normalmente, possuem também flip-flops para implementar lógica sequencial ou registos. Os três elementos principais utilizados na construção da parte combinatória de um bloco lógico são as portas lógicas, as tabelas de verdade (*lookup tables – LUTs*) e os multiplexadores. Estes elementos são muitas vezes combinados de forma a se obter uma estrutura compacta e com uma funcionalidade tão genérica quanto possível. Uma LUT de K entradas pode ser programada como uma tabela de verdade de uma função lógica de K variáveis. Um multiplexador de 2^N entradas pode implementar uma função de N variáveis.

A Figura 2.16 ilustra a implementação da função lógica " y = (a & b) | c " usando tabelas de verdade e multiplexadores. As LUTs possuem duas variantes de implementação. Assumindo LUTs de três entradas, a primeira consiste na utilização de um descodificador de 3 para 8 para seleccionar uma das 8 células SRAM que contém o valor de saída correspondente às entradas da tabela que implementa a função. Alternativamente, as saídas das células SRAM podem ser aplicadas a uma pirâmide de multiplexadores controlados pelas entradas da tabela, sendo a saída o valor fornecido pelo último multiplexador.

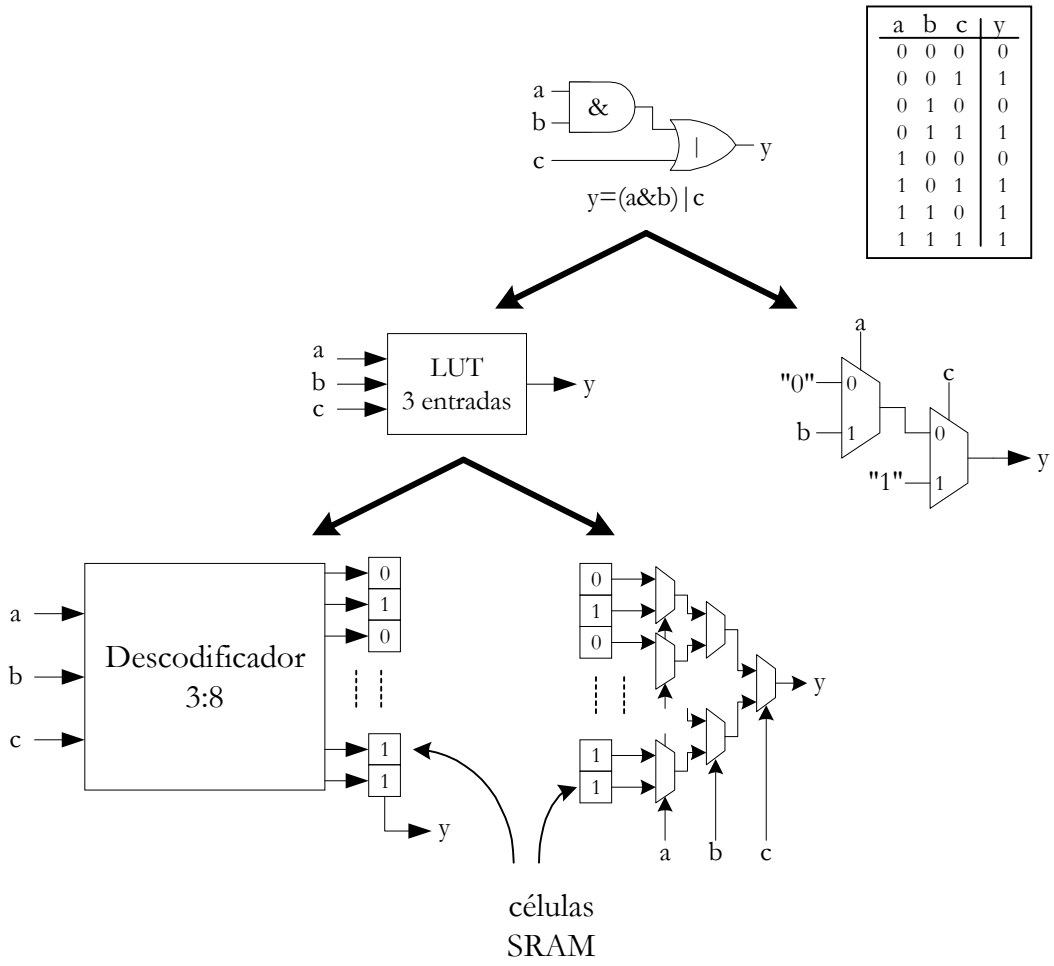


Figura 2.16 – Implementação de uma função lógica usando uma tabela de verdade ou multiplexadores.

A Figura 2.17 mostra um exemplo do bloco lógico usado na família XC4000 da Xilinx, no qual a implementação de funções Booleanas é feita com tabelas de verdade. Por outro lado, na Figura 2.18 estão ilustradas as estruturas dos blocos lógicos da família 54SX da Actel cujos elementos programáveis que implementam a parte combinatória são baseados em multiplexadores. Um factor que diferencia as várias arquitecturas de FPGAs é a granulosidade do seu bloco lógico. Enquanto o da Figura 2.17 é um exemplo de um bloco lógico de granulosidade média, os da Figura 2.18 são de granulosidade fina. Ao contrário das arquitecturas das MPGAs que são sempre de granulosidade fina (ao nível das portas lógicas primitivas e flip-flops), a granulosidade das arquitecturas de FPGAs é variável. Uma arquitectura de granulosidade fina comparativamente com uma arquitectura de granulosidade média/grossa, tem a vantagem de proporcionar uma taxa de utilização mais elevada dos seus blocos lógicos. No entanto, tem a desvantagem de necessitar de mais blocos para implementar uma dada função e consequentemente os sinais terem de atravessar um maior número de ligações programáveis, o que por sua vez provocará maiores atrasos. Assim a granulosidade de uma FPGA é um compromisso entre a generalidade do seu bloco lógico e o seu desempenho, sendo também influenciada pela sua tecnologia de programação.

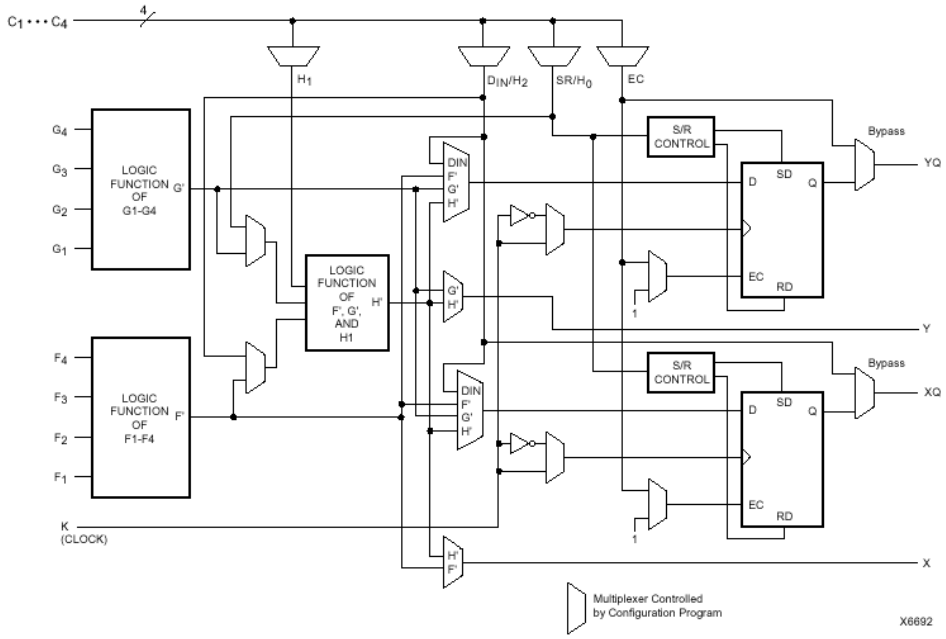


Figura 2.17 – Estrutura do bloco lógico configurável baseado em tabelas de verdade da família XC4000 da Xilinx.

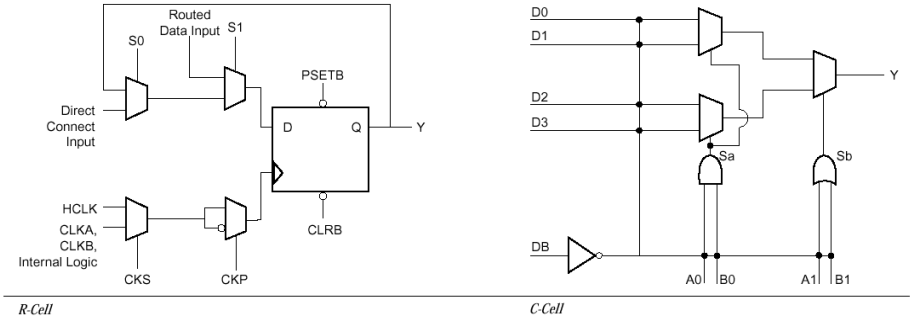


Figura 2.18 – Estrutura das células baseadas em multiplexadores da família 54SX da Actel.

Actualmente, a maioria das FPGAs disponíveis comercialmente emprega tecnologias de programação baseadas em SRAM e antifusível. No primeiro caso é utilizado um conjunto de células de memória que são carregadas durante a fase de programação com os valores binários que definem a funcionalidade dos blocos lógicos, dos blocos de entrada/saída e do estado dos elementos de controlo dos recursos de interligação. Estas células permanecem com o mesmo valor até ao carregamento de uma nova configuração. Esta tecnologia de programação apesar de requerer uma memória externa não volátil para armazenamento da configuração, permite que o mesmo circuito seja utilizado para diversas aplicações através do carregamento de uma configuração adequada. Por exemplo, uma FPGA pode ser inicialmente programada para efectuar o teste inicial do sistema onde se encontra inserida e posteriormente reprogramada para desempenhar a sua função no sistema.

Nem todas as FPGAs são baseadas em SRAM. A outra tecnologia utilizada na construção de FPGAs é baseada em antifusíveis. Na tecnologia antifusível, a

programação da FPGA é efectuada electricamente para fechar permanentemente alguns dos interruptores antifusíveis, de forma a estabelecer as conexões entre os blocos lógicos do dispositivo e assim definir a sua funcionalidade. As principais vantagens destas FPGAs relativamente às baseadas em SRAM é serem mais rápidas e não voláteis. Outra vantagem importante dos antifusíveis é o seu tamanho reduzido, tornando possível um grande número de pontos de interligação no dispositivo. No entanto, estas FPGAs possuem a desvantagem de ser programáveis apenas uma vez.

O exemplo da Figura 2.19 ilustra a aplicação numa FPGA de dois tipos de elementos programáveis controlados por células SRAM: um transístor de passagem para estabelecer a conexão entre duas linhas condutoras e um multiplexador para seleccionar as linhas que são aplicadas à entrada dos blocos lógicos. Mais concretamente, a figura mostra a ligação de um bloco lógico (representado pela porta AND no canto superior esquerdo) a outro através de dois transístores de passagem e um multiplexador, todos controlados por células SRAM. Em geral, uma FPGA pode usar transístores de passagem, multiplexadores ou ambos.

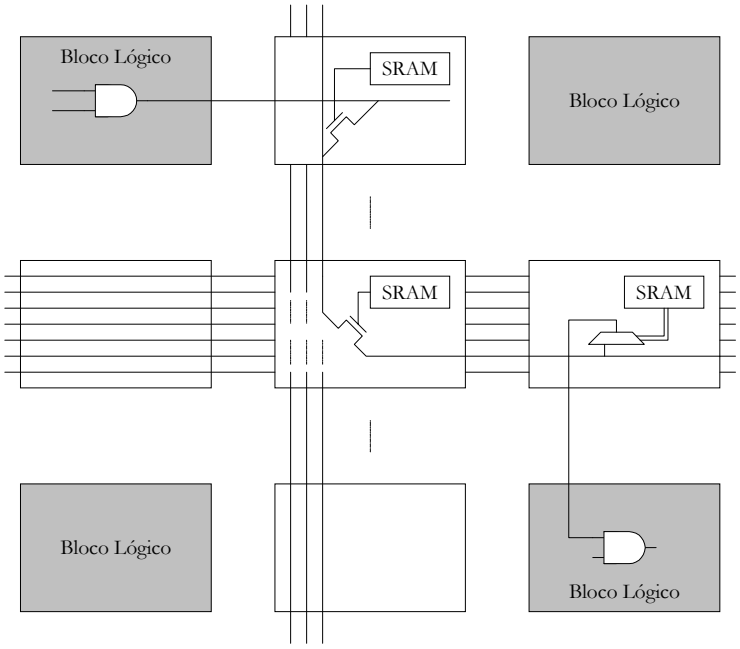


Figura 2.19 – Interruptores e multiplexadores controlados por células SRAM numa FPGA.

As FPGAs provocaram uma mudança nos métodos de implementação dos circuitos digitais já que são o único tipo de FPLD genérico de elevada capacidade e são actualmente um dos segmentos com maior crescimento na indústria dos semicondutores.

Infelizmente, as FPGAs também possuem alguns problemas. Em primeiro lugar, os atrasos das ligações não são tão previsíveis como nos SPLDs e CPLDs. Em segundo lugar, cada fabricante possui as suas próprias arquitecturas e disponibiliza as ferramentas de desenvolvimento para implementação de projectos nos seus dispositivos, dificultando a migração entre FPGAs de diferentes fabricantes. Em terceiro lugar, as ferramentas de síntese estão tradicionalmente mais orientadas para arquitecturas de granulosidade fina como as MPGAs, ou seja, produzem circuitos

constituídos por componentes ao nível das portas lógicas e flip-flops. Assim, os resultados produzidos por algumas ferramentas de síntese e implementação não tiram partido de todas as capacidades existentes no dispositivo utilizado, contribuindo para uma utilização deficiente dos seus recursos. Para contornar este problema é muitas vezes necessário instanciar de forma estrutural componentes que encapsulam uma funcionalidade específica da arquitectura utilizada, o que obriga o projectista a possuir um conhecimento aprofundado de alguns dos seus detalhes, impede o uso de ferramentas de síntese em todo o projecto e torna difícil a migração entre diferentes famílias de dispositivos. Por último, nas FPGAs é difícil prever os atrasos de propagação antes da interligação do circuito, sendo muitas vezes necessário realizar optimizações manuais bem como trabalhar a baixo nível para se obter os resultados pretendidos.

2.5.1 Aplicações das FPGAs

As FPGAs ganharam uma rápida aceitação na última década porque podem ser utilizadas numa grande variedade de aplicações, tais como implementação de lógica arbitrária, integração de vários SPLDs e CPLDs num único encapsulamento, controladores de dispositivos, circuitos de codificação e filtragem para telecomunicações, sistemas de pequena/média dimensão com blocos de memória SRAM, entre outros. As FPGAs podem também ser utilizadas na construção de protótipos de sistemas a implementar numa MPGA usando uma ou várias FPGAs de grande capacidade. De notar que em termos de capacidade, uma FPGA grande corresponde a uma MPGA pequena. Outra aplicação é a emulação de grandes sistemas de hardware através da utilização de várias FPGAs interligadas. Finalmente, uma aplicação das FPGAs que tem vindo a tornar-se muito popular é a computação reconfigurável, a qual consiste na utilização de dispositivos programáveis para executar em hardware algoritmos e aplicações tradicionalmente executadas em software num processador de uso geral. Isto permite combinar a versatilidade de uma solução programável com o desempenho do hardware dedicado.

Tal como mencionado atrás, os projectos são normalmente mapeados de forma natural nos blocos tipo SPLD de um CPLD. No entanto, um circuito mapeado numa FPGA deve ser partido em blocos lógicos que serão distribuídos pelo dispositivo e cuja granulosidade depende de uma arquitectura específica. Os tempos de atraso de um circuito dependem da estrutura dos blocos lógicos da FPGA, da organização das suas interligações e da forma como estes recursos são utilizados pelas ferramentas de CAD para implementação. Assim, o desempenho das FPGAs depende normalmente mais da sua arquitectura e das ferramentas de CAD utilizadas, do que no caso dos CPLDs.

2.6 Comparação entre os FPLDs e os MPLDs

Comparativamente com os MPLDs, os FPLDs possuem algumas vantagens importantes, nomeadamente o menor tempo de desenvolvimento de um produto e o custo inferior para pequenas quantidades. No entanto, os FPLDs também possuem algumas desvantagens, em particular ao nível da velocidade de operação e da densidade lógica do dispositivo. As próximas subsecções realizam uma análise comparativa entre

os dois tipos de dispositivos lógicos, tendo por base os seguintes parâmetros: velocidade, densidade, tempo de desenvolvimento, tempo para construção do protótipo e simulação, tempo para fabricação e desenvolvimento de testes, modificações futuras, risco de inventariação e custo.

2.6.1 Velocidade

Em muitas aplicações, as velocidades de operação dos FPLDs actuais variam entre as dezenas e as poucas centenas de MHz, sendo superiores às dos sistemas constituídos por circuitos integrados SSI e MSI mas inferiores às dos MPLDs. Nos sistemas com circuitos SSI/MSI a velocidade é na maior parte dos casos limitada pelos tempos de propagação dos sinais entre dispositivos. A menor velocidade dos FPLDs relativamente aos MPLDs resulta da capacidade de programação dos primeiros. Os elementos programáveis aumentam a resistência e a capacidade das ligações, o que por sua vez aumenta os tempos de propagação. Apesar desta desvantagem, a velocidade dos FPLDs é adequada para a maioria das aplicações. Além disso, com a inclusão na arquitectura dos FPLDs de componentes dedicados para fins específicos, tais como somadores, multiplicadores e memórias, pode-se melhorar substancialmente o seu desempenho através da redução dos tempos de propagação críticos (ex. propagação do *carry* de um somador). A velocidade de operação de um sistema baseado em FPLDs pode ser aumentada com a utilização de dispositivos fabricados com processos mais recentes e mais rápidos, sem que para tal seja necessário efectuar qualquer alteração de projecto [SalSma97]. No caso dos MPLDs, a situação é completamente diferente onde a utilização de novos processos requer a construção de novas máscaras, o que contribui para um aumento do tempo de desenvolvimento e do custo final do produto.

2.6.2 Densidade

A capacidade de programação dos FPLDs necessita de circuitos adicionais que ocupam área não utilizável pelos projectistas. Consequentemente, para a mesma quantidade de lógica disponível, um FPLD será geralmente maior e mais caro do que um MPLD. No entanto, nos MPLDs mais recentes, uma grande área do circuito não pode ser utilizada para implementação de componentes devido a limitações físicas impostas pelos pinos de entrada e saída. Assim, a utilização desta área desperdiçada para circuitos de programação pode não resultar num aumento do tamanho do FPLD [SalSma97]. Por outras palavras, quando o tamanho de um MPLD ou FPLD for estabelecido pelo número de pinos, as capacidades lógicas de ambos podem ser iguais. Este facto deve-se à utilização de processos submícron na fabricação dos FPLDs. Por este motivo, os fabricantes de MPLDs estão a concentrar-se nos dispositivos de muito alta capacidade (>5,000,000 de portas lógicas) e/ou de elevado desempenho, deixando os de menor capacidade para os FPLDs.

2.6.3 Tempo de Desenvolvimento

O desenvolvimento de FPLDs é acompanhado pela criação de ambientes para o projecto de sistemas que os utilizam. Este ambientes integram uma vasta gama de ferramentas para especificação, descrição, síntese, simulação e implementação do

projecto e reutilização de componentes previamente desenvolvidos e testados, o que contribui para uma diminuição do tempo total de projecto. Nas fases iniciais dos projectos baseados em FPLDs, em particular na especificação, modelação e subtarefas de síntese independente da tecnologia, podem ser utilizadas as mesmas ferramentas de desenvolvimento dos MPLDs e ASICs tradicionais. A lista de ligações resultante é posteriormente manipulada pelos algoritmos de síntese e implementação específicos do FPLD e que são disponibilizados pelos seus fabricantes ou vendedores de ferramentas de CAD. Para tirar partido das especificidades de cada FPLD, o projecto físico ou implementação pode também ser realizado manualmente e a baixo nível, o que contribui geralmente para uma melhor utilização dos recursos existentes mas à custa de um tempo de desenvolvimento mais elevado.

Na generalidade, estas ferramentas são de alto nível e acessíveis à maioria das empresas e instituições que utilizam FPLDs nos seus projectos. O tempo de desenvolvimento de um projecto baseado em FPLDs é gasto essencialmente na sua descrição, simulação e criação do protótipo, não sendo necessárias as fases de geração de vectores de teste, construção das máscaras, fabricação do circuito, encapsulamento e teste, as quais consomem grande parte do tempo de desenvolvimento de um MPLD. Isto faz com que o tempo de desenvolvimento para projectos baseados em FPLDs seja da ordem de dias ou semanas enquanto nos MPLDs é da ordem das semanas ou meses.

2.6.4 Tempo para Construção do Protótipo e Simulação

Enquanto o processo de fabricação de um MPLD demora semanas ou meses, desde a conclusão do projecto até à existência física de dispositivos, um FPLD requer somente a finalização do projecto. As modificações para a correcção de um erro de projecto ou realização de melhoramentos são rápidas e fáceis de realizar, resultando em ciclos curtos de projecto e na diminuição dos tempos de desenvolvimento para novos produtos baseados em FPLDs.

Para que o projecto de um MPLD seja correctamente validado, devem ser efectuadas simulações exaustivas antes da sua fabricação. Um problema inerente a qualquer simulação é o compromisso entre a rapidez e a precisão dos resultados obtidos. Por outro lado, nos FPLDs as simulações são muito mais simples devido ao conhecimento prévio dos modelos e das suas características temporais. Em muitos casos, a simulação é completamente substituída pela depuração do protótipo, o qual funciona à velocidade real do sistema e com precisão temporal absoluta. Um protótipo pode ser facilmente modificado e reinserido no sistema em minutos ou horas.

Os FPLDs permitem a construção de protótipos de baixo custo enquanto os MPLDs são mais económicos para grandes quantidades. Assim, em certos casos os FPLDs são utilizados durante o desenvolvimento sendo depois substituídos por MPLDs no produto final, desde que as quantidades a produzir o justifiquem. Normalmente não é necessário modificar o projecto quando se faz conversão de FPLD para MPLD, exceptuando os casos em que ocorrem problemas na verificação temporal. Alguns fabricantes fornecem versões programadas por máscara dos seus FPLDs combinando a flexibilidade e vantagens de ambos os métodos.

2.6.5 Tempo para Fabricação e Desenvolvimento de Testes

Para verificar o seu funcionamento, todos os circuitos integrados devem ser testados após a fabricação e encapsulamento, sendo o teste específico de cada dispositivo. Os MPLDs possuem três tipos de custos associados ao seu teste:

- Lógica integrada adicional para facilitar o teste;
- Geração de programas de teste para cada tipo de dispositivo;
- Teste dos dispositivos após a sua fabricação.

Devido à estrutura simples e regular dos FPLDs, a preparação dos programas de teste é relativamente simples. São também justificados os esforços na produção de programas de testes extensivos e de elevada qualidade pois poderão ser utilizados durante todo o tempo de vida de um FPLD. Os utilizadores não necessitam de desenvolver testes para os FPLDs dependentes do projecto porque os testes do fabricante garantem que o dispositivo funciona de acordo com as suas especificações para todos os projectos implementados. O utilizador necessita apenas de realizar a depuração do sistema projectado. As consequências da fabricação de ambas as categorias de circuitos são óbvias. Uma vez verificados os FPLDs podem ser fabricados em qualquer quantidade e disponibilizados como componentes completamente testados e prontos para a implementação de um projecto, enquanto os MPLDs requerem procedimentos de preparação antes da sua produção, que são específicos de cada projecto, o que contribui para um aumento do tempo de fabricação.

2.6.6 Modificações Futuras

Enquanto os MPLDs são programados durante o processo de fabricação, os FPLDs são programados electricamente após a sua fabricação. A programação pode demorar desde milissegundos a minutos e pode ser realizada com equipamento simples e de baixo custo. Em muitos casos a programação pode inclusive ser realizada no próprio sistema. Por oposição, qualquer modificação de um MPLD necessita de uma máscara cujo custo é considerável e deve ser amortizado pelo número total de unidades produzidas.

2.6.7 Risco de Inventariação

Uma importante vantagem dos FPLDs é o seu baixo risco de inventariação, semelhante ao dos componentes SSI e MSI, uma vez que através de programação, o mesmo dispositivo pode ser utilizado em diversos projectos. Tal não acontece com os MPLDs, nos quais a funcionalidade e as áreas de aplicação não podem ser alteradas a partir do momento que termina o seu processo de fabricação. Além disso, a decisão acerca do volume de MPLDs a produzir deve ser realizada com a devida antecedência de forma a permitir a sua fabricação e garantir que os dispositivos estarão disponíveis quando necessários e ao mesmo tempo com o mínimo de exactidão para evitar a fabricação de um número exagerado ou insuficiente de dispositivos. Em termos económicos e de atrasos de projecto, os FPLDs possuem geralmente um risco associado muito baixo. A construção de protótipos baseados em FPLDs possibilita a

correcção rápida de eventuais erros e permite que os projectistas adoptem nas etapas iniciais de desenvolvimento de um produto abordagens de projecto menos conservadoras e consequentemente mais arriscadas mas que podem também conduzir a melhores resultados finais.

2.6.8 Custo

Os factores descritos acima reflectem-se no custo dos dispositivos. A maior vantagem de um projecto baseado em MPLDs é o seu baixo custo em grandes quantidades. Os FPLDs possuem custos de concepção, desenvolvimento, modificação e teste muito inferiores. No entanto, como possuem uma maior área e uma menor densidade, o custo de produção unitário é superior. Assim, o volume de mercado de um produto determina a tecnologia de implementação mais apropriada para ser utilizada no seu projecto.

Resumindo, o baixo custo dos FPLDs torna-os bastante atractivos para o desenvolvimento de produtos com um volume de mercado que não justifique a sua implementação em MPLDs. O reduzido tempo de fabricação é um elemento essencial do sucesso dos FPLDs. No entanto, os atrasos provocados pelos interruptores programáveis, bem como a área por eles ocupada impedem que os FPLDs atinjam as velocidades e as densidades dos MPLDs. Contudo, os melhoramentos das arquitecturas e das ferramentas de CAD permitirão minimizar estas desvantagens. Além disso, o aumento da escala de integração faz com que nos circuitos, cuja dimensão mínima seja limitada pelo número de pinos de entrada/saída, a densidade se torne cada vez menos importante. Por estes motivos, acredita-se que esta será cada vez mais a tecnologia empregue na implementação de circuitos digitais.

2.7 A Família de FPGAs XC6200 da Xilinx

O trabalho desenvolvido no âmbito desta dissertação utilizou como tecnologia de implementação a família de FPGAs XC6200 da Xilinx [Xilinx97a]. Para facilitar a compreensão dos capítulos que descrevem o trabalho realizado, nas próximas subsecções são apresentadas de forma resumida as características mais relevantes destes dispositivos.

Em 1998, a Xilinx decidiu descontinuar a família XC6200. No entanto, devido às suas características únicas, tais como reconfiguração dinâmica parcial e interface com microprocessador, a comunidade académica e de investigação continuou a demonstrar bastante interesse pela XC6200, como se pode aliás comprovar pela análise das actas de algumas das conferências internacionais mais importantes da especialidade [FPL99, FCCM99, FPGA99]. Por este motivo, a Xilinx continuou a fornecer dispositivos desta família para efeitos de investigação.

2.7.1 Descrição

As FPGAs XC6200 da Xilinx são baseadas em SRAM, pelo que os dispositivos desta família podem ser reconfigurados rapidamente e um número ilimitado de vezes. Tal como outras famílias de FPGAs destina-se a implementar funções normalmente desempenhadas por um ASIC. A sua arquitectura é do tipo celular (*sea-of-gates)*, os seus

blocos constituintes de granulosidade fina e possui um esquema de interligação hierárquica de baixos atrasos. Além disso, suporta configurações totais ou parciais. O seu interface permite a ligação directa a um microprocessador ou microcontrolador através de um barramento paralelo, de forma a mapear a memória de configuração da FPGA no espaço de endereçamento do processador hospedeiro. Assim, do ponto de vista de interface, para o processador hospedeiro a FPGA comporta-se como uma memória estática convencional. Além do interface paralelo, existe também um interface série, o qual é normalmente utilizado para ligação a uma memória série do tipo PROM, EPROM ou EEPROM e responsável pela configuração inicial do dispositivo. No caso de projectos altamente estruturados, tais como certas unidades de execução, a capacidade lógica equivalente do dispositivo maior desta família (XC6264) pode atingir as 200,000 portas lógicas.

Os dispositivos XC6200 possuem também lógica adicional para suportar a sua reconfiguração total ou parcial sem suspender a sua operação. O processador hospedeiro pode ler ou escrever o conteúdo dos registos pertencentes ao circuito lógico implementado no dispositivo. As transferências de dados podem ser de 8, 16 ou 32 bits mesmo quando os bits do registo estão distribuídos ao longo de uma coluna de células. Além disso, as saídas das suas unidades funcionais podem ser lidas pelo processador através do mesmo interface que é usado para configuração. Estas características permitem às FPGAs XC6200 suportar o conceito de hardware virtual, através do qual os circuitos implementados e a executar na FPGA podem ser salvaguardados num dispositivo externo e temporariamente substituídos por outros para permitir a atribuição dos recursos da FPGA a diferentes tarefas. Diferentes secções do dispositivo podem ser reconfiguradas sem perturbar os circuitos que executam nas outras partes. Assim, uma FPGA XC6200 numa aplicação de coprocessamento pode ser partilhada no tempo por vários processos a executar no computador hospedeiro.

2.7.2 Estrutura e Recursos de Interligação

A estrutura da arquitectura XC6200 é regular, simétrica e hierárquica. No nível mais baixo da hierarquia existe uma matriz de células lógicas simples e configuráveis. Cada uma destas células consiste numa unidade funcional e uma área de interligação através da qual é realizada a comunicação entre as células (Figura 2.20). Cada célula pode ser ligada às suas vizinhas (horizontais e verticais) através de recursos de interligação designados por locais (Figura 2.21). No entanto, para permitir ligar de forma eficiente células não adjacentes, ou seja, para reduzir o atraso das ligações mais longas, as células são agrupadas hierarquicamente em blocos cada vez maiores, os quais possuem recursos de interligação próprios. Em primeiro lugar, são formados blocos de 4×4 células. As linhas existentes a este nível possuem o comprimento de 4 células e possibilitam a comunicação entre blocos vizinhos sem que para tal seja necessária a utilização dos recursos de interligação locais (Figura 2.22).

Seguidamente e de forma análoga ao caso anterior, estes blocos de 4×4 células são agrupados em blocos de 16×16 células, possuindo também este nível da hierarquia recursos de interligação específicos constituídos por linhas horizontais e verticais de

comprimento 16 células (Figura 2.23). Por último, os blocos de 16×16 células são agrupados até se atingir o tamanho pretendido para o dispositivo. As dimensões das FPGAs XC6216 e XC6264 são respectivamente 64×64 e 128×128 células. Existem também linhas com o comprimento total do dispositivo, que possibilitam a ligação entre células distantes e/ou blocos de entrada/saída localizados na sua extremidade (Figura 2.24). A vantagem de uma arquitectura de interligação hierárquica é a variação logarítmica em vez de linear dos atrasos de propagação em função da distância.

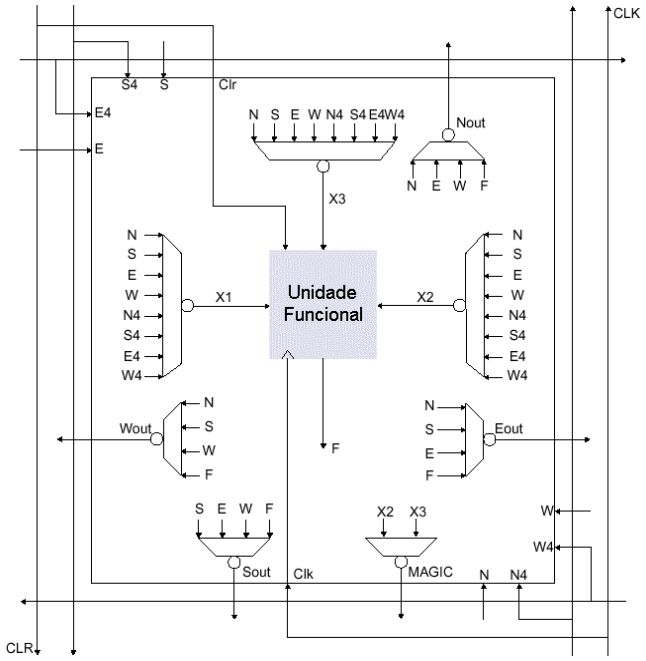


Figura 2.20 – Estrutura duma célula XC6200 constituída por uma unidade funcional e por recursos de interligação (linhas e multiplexadores).

Todos os recursos de interligação são unidireccionais. Os seus nomes dependem do seu sentido (Norte-N, Sul-S, Este-E, Oeste-W) e do seu comprimento em número de células (1, 4, 16, dispositivo-CL) – exemplos: S, N4, W16, ECL.

Para além dos recursos de interligação de cada nível hierárquico, existem também linhas globais e linhas especiais designadas pelo fabricante por *magic*, que proporcionam recursos de interligação adicionais. As quatro linhas globais (G1, G2, GClk e GClr) destinam-se essencialmente à distribuição de sinais de elevado *fan-out* e dos sinais de controlo (*clock e clear*) dos flip-flops existentes em cada célula. Estes sinais podem também ser aplicados aos respectivos flip-flops através de linhas não globais (locais e hierárquicas). No entanto, para evitar problemas de temporização e sempre que um dado sinal se destinar a controlar vários flip-flops dispersos pela FPGA é preferível a utilização das linhas globais, as quais garantem atrasos e desfasamentos reduzidos. As linhas *magic* possibilitam a realização de ângulos rectos nas ligações, ao contrário de todos os outros recursos de interligação que são rectilíneos. Tal como ilustrado na Figura 2.25, este tipo de recursos de interligação está organizado em torno dos blocos de 4×4 células.

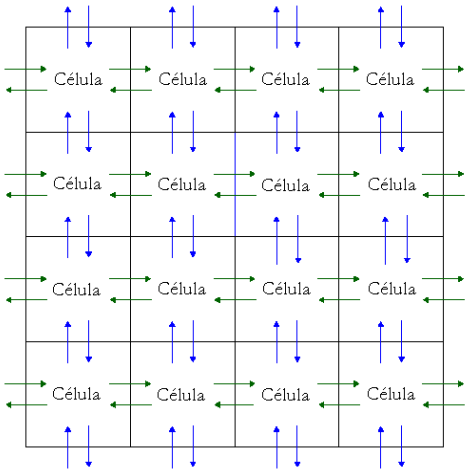


Figura 2.21 – Estrutura das interligações entre células vizinhas.

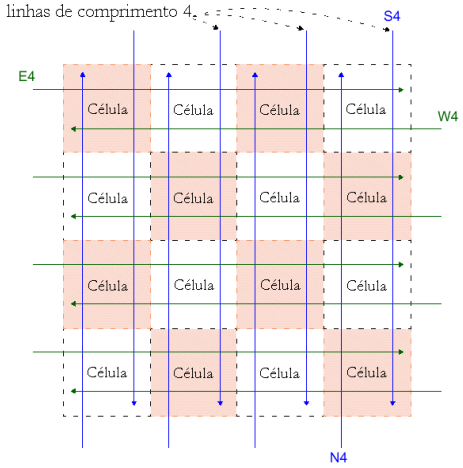


Figura 2.22 – Bloco de 4×4 células e respectivas interligações.

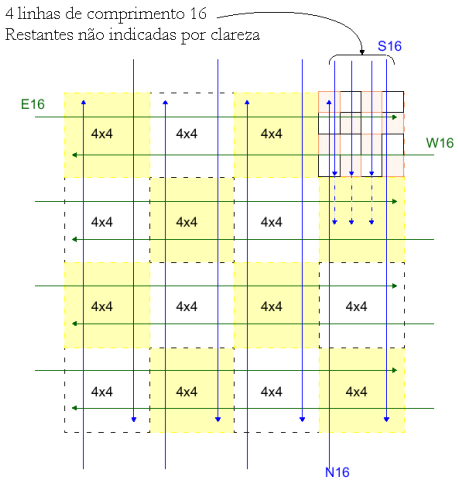


Figura 2.23 – Bloco de 16×16 células e respectivas interligações.

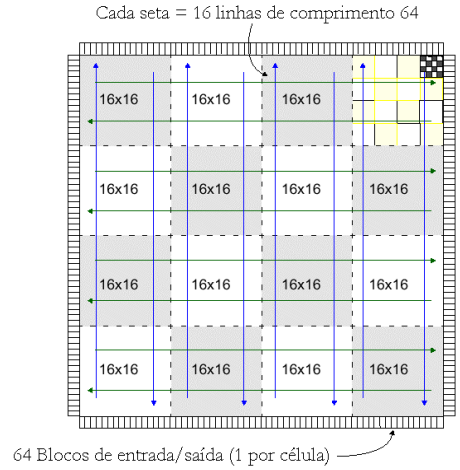


Figura 2.24 – Dispositivo XC6216 formado por 16 blocos de 16×16 células, 64 blocos de entrada/saída e recursos de interligação.

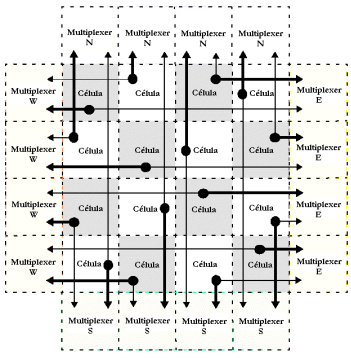


Figura 2.25 – Traçado das linhas *magic* num bloco de 4×4 células.

2.7.3 Unidade funcional

Cada unidade funcional contém cinco multiplexadores e um flip-flop tipo D (Figura 2.26). Os sinais *Clk* e *Clr* controlam o flip-flop e os três sinais X1, X2 e X3 são usados como entradas da função lógica. A estrutura da unidade funcional baseia-se no facto de qualquer função de duas variáveis Booleanas poder ser calculada usando multiplexadores de 2:1 se forem aplicadas às suas entradas as variáveis apropriadas ou os seus complementos.

Todos os multiplexadores no interior da célula são controlados pela memória de configuração da FPGA, à excepção do multiplexador controlado pela entrada X1. As entradas da célula X2 e X3 são transformadas nas saídas Y2 e Y3 dos respectivos multiplexadores, consoante o valor dos bits de configuração que controlam as suas entradas de selecção. O multiplexador RP protege o conteúdo do flip-flop através da realimentação do seu valor de saída para a entrada. Um flip-flop "protegido" só pode ser alterado a partir do interface de configuração. O multiplexador CS selecciona o sinal que controla a saída F da célula, entre uma saída combinatória ou sequencial.

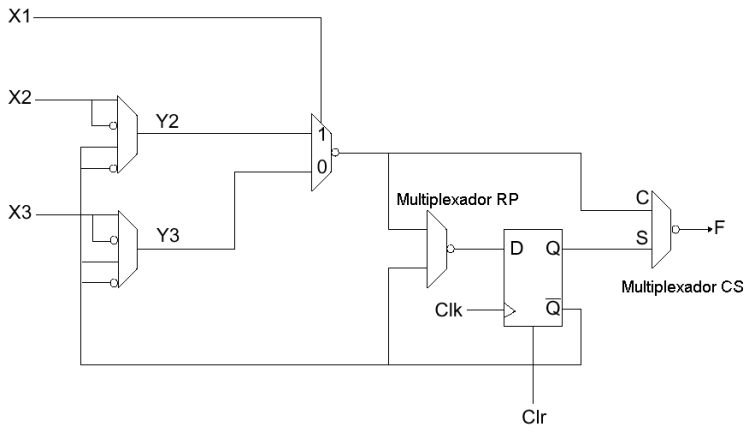


Figura 2.26 – Estrutura interna da unidade funcional.

Qualquer célula pode ser configurada para implementar uma função lógica de uma ou duas entradas, um multiplexador de 2:1, uma saída constante (0 ou 1) ou qualquer um dos casos anteriores com um flip-flop tipo D disparado por flanco.

Na Figura 2.27 são mostradas as várias configurações possíveis de uma célula. Algumas combinações não são apresentadas na figura uma vez que podem ser obtidas a partir de outra. Um multiplexador com a entrada SEL negada é um exemplo de uma primitiva desnecessária uma vez que as entradas A e B podem ser trocadas pela ferramenta de mapeamento.

As funções disponíveis em cada célula proporcionam um conjunto alvo bastante conveniente para as ferramentas de síntese que produzem circuitos ao nível das portas lógicas e flip-flops. Uma vez que cada célula possui um flip-flop, podem ser implementados projectos com um elevado número de registos. A arquitectura de pequena granulosidade permite um mapeamento fácil de projectos arbitrários em várias células, bem como um elevado grau de utilização das mesmas. A desvantagem das arquitecturas programáveis de pequena granulosidade é o grande número de células

lógicas necessárias para implementar uma dada funcionalidade e consequentemente os elevados atrasos provocados pelo atravessamento de muitos elementos programáveis. Em arquitecturas de granulosidade mais grossa, o circuito deve ser dividido em componentes de maior complexidade que serão mapeados em blocos lógicos configuráveis de maiores dimensões. Neste caso, uma desvantagem importante é o menor nível de utilização do bloco lógico.

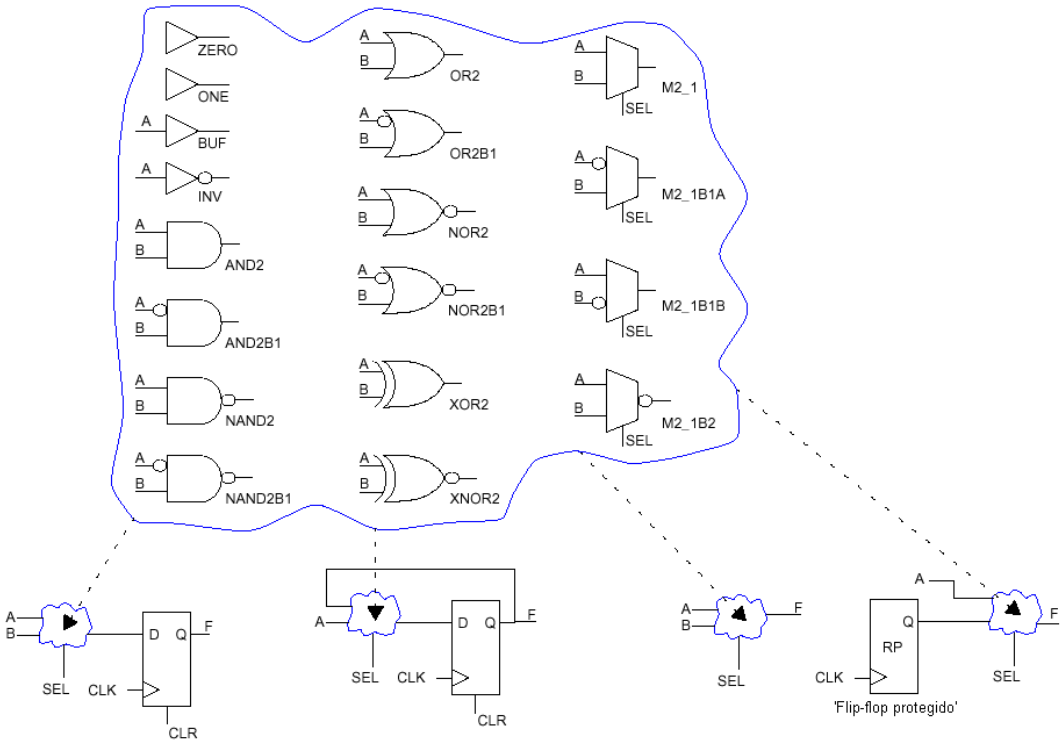


Figura 2.27 – Funções lógicas implementáveis na unidade funcional.

2.7.4 Registos de controlo

Para facilitar e acelerar a reconfiguração do dispositivo e o acesso aos registos de utilizador implementados nas células de uma FPGA XC6200, existem vários registos de controlo acessíveis a partir do interface de configuração. Estes registos estão localizados no mapa de endereços da FPGA, numa área especial dedicada a registos de controlo. De todos, os mais relevantes são designados pelo fabricante por *Map Register*, *Mask Register* e *Wildcard Registers*. A função de cada um deles é descrita sucintamente nas próximas subsecções.

Map Register

As FPGAs da família XC6200 possuem um mecanismo para mapear as saídas das células de uma coluna nos bits do barramento de dados externo. Isto permite seleccionar somente as células que implementam os bits do componente (normalmente um registo) ao qual se pretende aceder. Se este mecanismo não existisse, o processador hospedeiro teria de realizar operações complexas de deslocamento e máscara para

eliminar os bits correspondentes às células não pertencentes ao registo. Outra possibilidade seria o utilizador restringir a implantação dos componentes de forma a que os bits do registo ficassem em células adjacentes, o que nem sempre pode ser conseguido. A parte visível deste mecanismo está implementada na forma de um registo de mapeamento (*Map Register*), o qual possui um bit por cada linha de células da matriz. Antes de serem realizados acessos aos registos implementados nas células da matriz, o *Map Register* deve ser carregado com um valor apropriado.

Um valor lógico 0 num bit do *Map Register* indica que a célula da respectiva linha faz parte do registo ao qual se pretende aceder. O mapeamento das células de uma linha da matriz nos bits do barramento de dados começa pelo bit menos significativo. Assim, a primeira linha com um 0 no *Map Register* é associada ao bit 0 do barramento de dados externo, a segunda linha com um 0 no *Map Register* liga ao bit 1 e assim sucessivamente. Na Figura 2.28 é mostrado um exemplo de operação do *Map Register*.

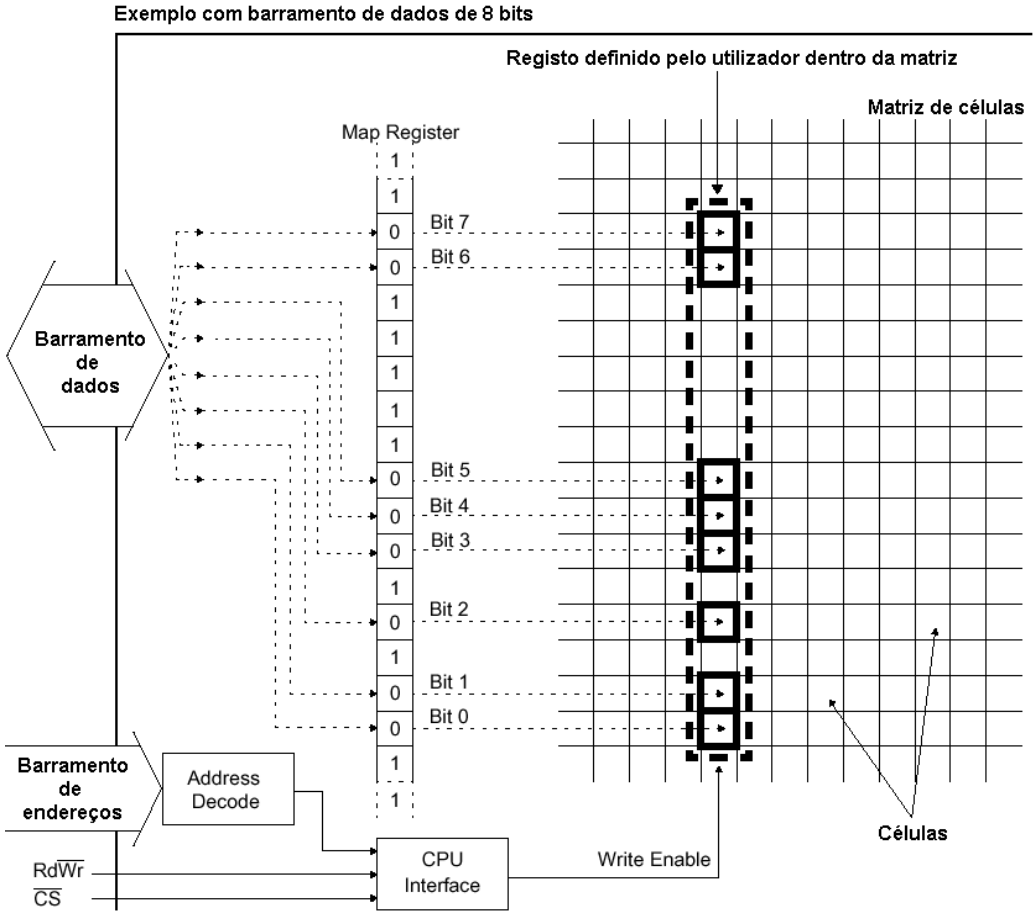


Figura 2.28 – Exemplo de operação do *Map Register*.

Mask Register

Entre o barramento de dados externo e as ligações internas existe uma unidade de mascaramento que pode ser usada para preservar o valor de determinados bits durante a configuração do dispositivo. Esta unidade é controlada por um registo de 32

bits designado por *Mask Register*. O número de bits deste registo que são significativos em cada momento corresponde à largura do barramento de dados externo nesse instante. Um valor lógico 1 num bit deste registo indica que o bit correspondente do barramento de dados não é relevante, ou seja, durante uma operação de escrita, os respectivos bits da memória de configuração preservam o seu valor.

O *Mask Register* não afecta o acesso aos registos das células. Nestes casos, o *Map Register* pode ser utilizado para impedir que certos bits sejam modificados. Finalmente, o *Mask Register* é também ignorado durante as operações de leitura e escrita nos registos de controlo.

Wildcard Registers

O interface de uma FPGA XC6200 contém circuitos adicionais que podem reduzir significativamente o número de transferências de dados durante a configuração do dispositivo. Estas unidades estão activas apenas nos ciclos de escrita da memória de configuração e inactivas durante todos os acessos aos registos de controlo. Estes circuitos são controlados por dois registos especiais designados por *Row Wildcard* e *Column Wildcard*.

O descodificador de endereços de linha é complementado com o registo *Row Wildcard*, o qual possui um bit por cada bit do campo que endereça as linhas de células da FPGA. Durante os ciclos de escrita, os bits que possuem o valor lógico 1 no registo de *Row Wildcard* indicam que o bit correspondente do endereço deve ser considerado como "*don't care*", ou seja, o descodificador selecciona todos os endereços independentemente deste bit. Quando o dispositivo é inicializado o registo está preenchido com zeros para que todos os bits de endereço sejam significativos. Por outras palavras, o registo *Row Wildcard* permite a escrita simultânea em várias células pertencentes à mesma coluna com os mesmos dados de configuração. Este procedimento é efectuado durante o teste do dispositivo para permitir o carregamento eficiente de sequências regulares na memória de configuração, mas é geralmente mais útil em projectos compostos por componentes regulares, pois permite a alteração simultânea de várias células.

De forma análoga, o descodificador de endereços de coluna possui igualmente um registo de *Column Wildcard*, o qual possibilita a escrita da mesma informação de configuração em várias colunas de células pertencentes à mesma linha.

Os registos *Mask Register* e *Wildcard Registers* podem ser utilizados em conjunto para realizar de forma eficiente operações complexas de reconfiguração de estruturas regulares no dispositivo.

À semelhança do *Mask Register*, o registo *Row Wildcard* é ignorado nas operações de acesso aos flip-flops das células, nas quais a descodificação das linhas é realizada pelo *Map Register*. Quanto ao registo *Column Wildcard* pode ser utilizado para carregar simultaneamente o mesmo valor num banco de registos distribuído por várias colunas.

A Figura 2.29 mostra alguns exemplos da utilização desta unidade, assumindo um barramento de dados de 8 bits e uma FPGA XC6216, na qual o número de linhas e colunas da matriz de células é 64. As saídas do descodificador de linha activam os circuitos de programação das células cuja configuração se pretende alterar.

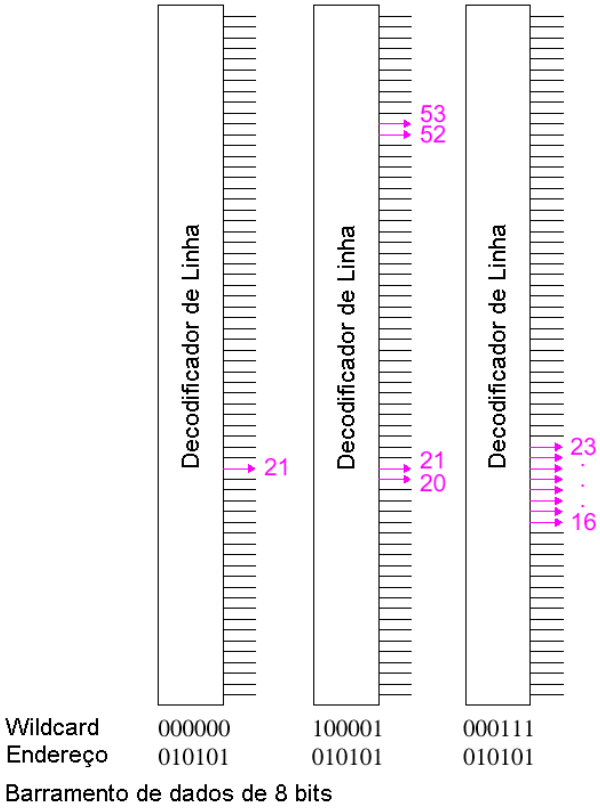


Figura 2.29 – Activação das linhas de saída do descodificador de linha em função do endereço e do valor do registo *Row Wildcard*.

2.7.5 Fluxo de Projecto

O fluxo de projecto para sistemas digitais destinados a serem implementados em dispositivos da família XC6200 consiste essencialmente no mesmo conjunto de etapas descritas no primeiro capítulo desta dissertação. Assim, nesta secção serão abordados apenas os aspectos particulares desta tecnologia. Os passos de descrição, síntese e validação podem ser realizados por aplicações standard de captura de esquemático, linguagens de descrição de hardware, síntese e simulação comercializadas pelas empresas que desenvolvem ferramentas de CAD, tais como *Viewlogic*, *Synopsys*, *Mentor Graphics*, etc.

A Figura 2.30 mostra uma visão simplificada do fluxo de projecto, assumindo como ponto de partida uma descrição estrutural do circuito. A elaboração de descrições estruturais deve ser baseada em bibliotecas que contêm as primitivas implementáveis numa unidade funcional, tais como portas lógicas, flip-flops e multiplexadores [Xilinx97b]. Para simplificar o projecto de circuitos mais complexos, existem também bibliotecas de macro-blocos com componentes mais elaborados, tais como somadores, multiplicadores, registos, contadores, filas, pilhas, etc. [Xilinx97b]. Podem também ser definidas pelo utilizador outras bibliotecas desde que sejam baseadas nas anteriores. No caso de descrições estruturais, um dos métodos mais utilizados consiste na preparação de uma descrição do circuito usando VHDL

estrutural e na sua conversão para uma lista de ligações EDIF (*Electronic Data Interchange Format*) através da aplicação de software VELAB (*Vhdl ELABorator*) [Xilinx97b].

Tal como descrito atrás, a simulação funcional é utilizada para averiguar a operação do circuito do ponto de vista lógico, antes da sua implementação, ou seja, sem qualquer informação temporal.

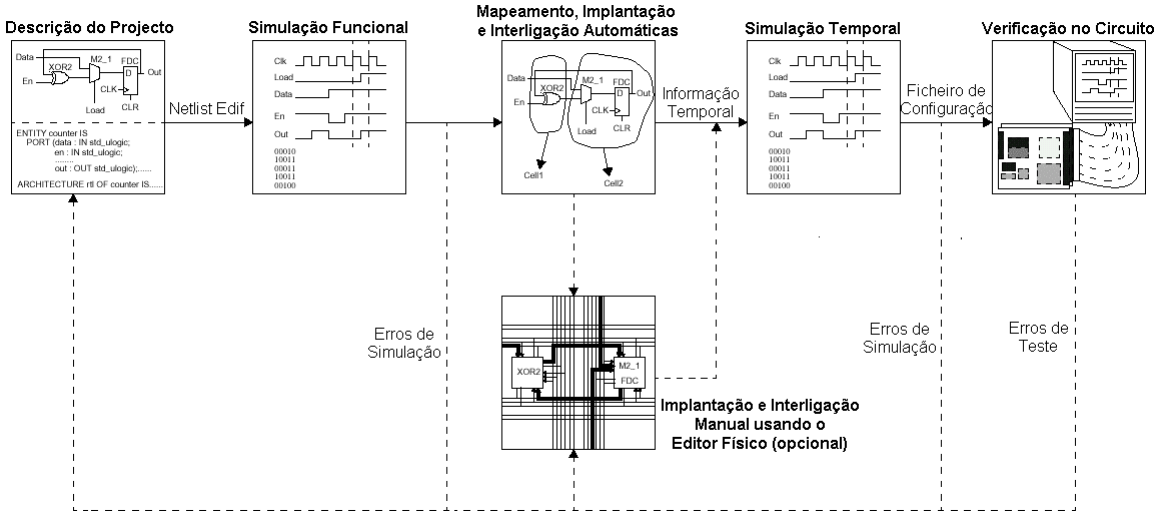


Figura 2.30 – Fluxo de projecto típico para a XC6200.

A implementação do circuito no dispositivo é suportada pela ferramenta *XACTstep Series 6000* [Xilinx97c], que tem como entrada a lista de ligações resultante das fases de descrição ou de síntese do projecto. Na Figura 2.31 é apresentado o seu interface gráfico. Mais concretamente, esta ferramenta realiza a última etapa da síntese lógica do circuito, ou seja, o mapeamento na tecnologia e as duas tarefas que constituem o projecto físico do circuito, a implantação e interligação dos seus componentes. Esta ferramenta pode implementar o circuito de forma automática, semi-automática, através do fornecimento de directivas, ou completamente manual. O método adoptado depende da optimização e do tempo de projecto pretendidos. O desenvolvimento incremental é também suportado através da repetição da implementação, somente para os blocos que sejam modificados ao longo das várias iterações de projecto.

Os atrasos dos componentes do circuito podem ser adicionados à lista de ligações para efeitos de simulação temporal.

O *XACTstep Series 6000* produz os seguintes ficheiros de saída:

- *.CAL – contém a informação para configuração da FPGA;
- *.SYM – contém informação acerca dos componentes do circuito implementados na FPGA, como por exemplo, tipo, localização e modos de acesso;
- *.RAL – contém informação para reconfiguração dos componentes que foram identificados como reconfiguráveis na fase de descrição do projecto.

Por uma questão de simplicidade de linguagem, a ferramenta *XACTstep Series 6000* passará a ser designada a partir de agora apenas por XACT6000.

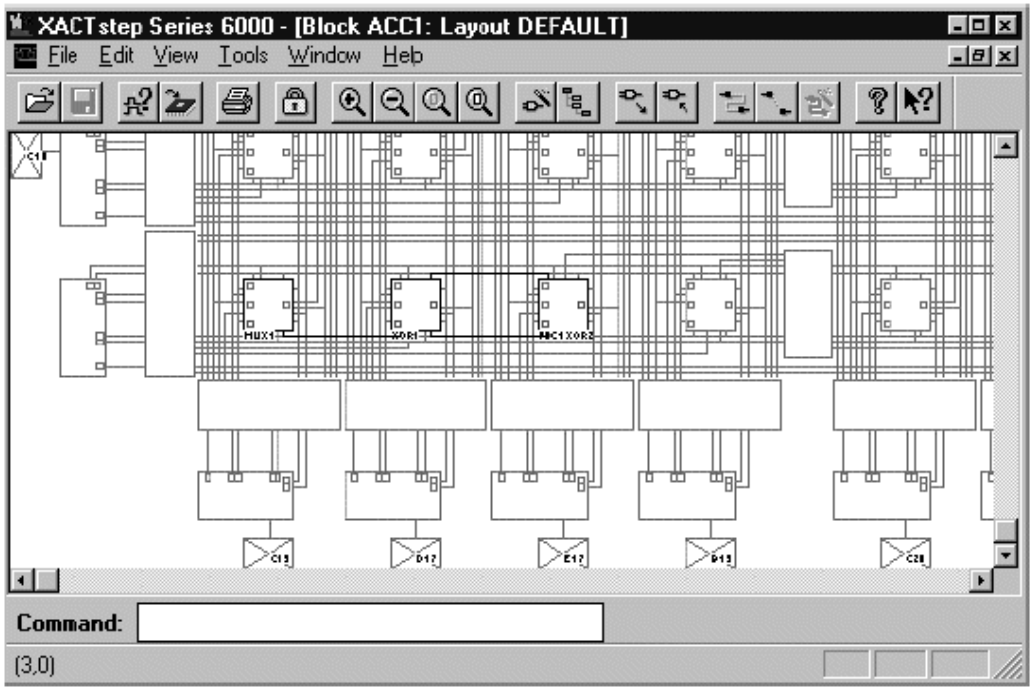


Figura 2.31 – Interface da ferramenta *XACTstep Series 6000* para implementação de circuitos na FPGA XC6200.

O primeiro ficheiro é sempre utilizado uma vez que contém a configuração que deve ser descarregada na FPGA para que esta inicie a sua operação. O segundo ficheiro é usado normalmente para efeitos de depuração do circuito. Por último, o terceiro tipo de ficheiro é utilizado para reconfigurar o dispositivo, embora o método utilizado possua bastantes limitações uma vez que todas as células que se pretende que sejam reconfiguráveis devem ser identificadas individualmente durante a descrição do projecto. Um método mais conveniente para configurar grandes áreas da FPGA baseia-se na definição de janelas de reconfiguração e no carregamento de um ficheiro CAL que programe todos os endereços relativos a essa janela. Este é aliás o método utilizado neste trabalho e que será descrito mais à frente.

2.7.6 Recomendações de Projecto

As arquitecturas das FPGAs são bastante variadas e por isso os projectistas devem adoptar diferentes estratégias e técnicas de projecto de forma a maximizar o desempenho e minimizar a área do circuito projectado. Tendo por base as sugestões publicadas em [WooTraHer98] e a experiência adquirida ao longo deste trabalho, são sumariadas abaixo as recomendações mais relevantes que devem ser tomadas em conta no projecto de circuitos que utilizem a XC6200 como tecnologia de implementação.

Considerações Arquitecturais

Comparada com outras famílias de FPGAs, tais como as séries XC4000 ou *Virtex* da Xilinx [Xilinx99] e a série *Flex* da Altera [Altera99], os recursos de interligação da XC6200 são limitados. Os projectos altamente estruturados e com baixa complexidade de interligação podem ser implementados eficientemente na XC6200,

sendo esta capacidade bem explorada pelas ferramentas de projecto. Nos restantes casos a melhor estratégia consiste em expressar os blocos do sistema numa forma comportamental de alto nível e passar essa descrição às ferramentas de síntese que geram as representações estruturais do sistema a projectar. Isto permite optimizar o circuito com base nas primitivas existentes na biblioteca da tecnologia alvo, o que por sua vez possibilita que as ferramentas responsáveis pela implementação do circuito utilizem eficientemente os recursos lógicos e de interligação disponíveis.

Sinais de Controlo dos Flip-flops

As quatro linhas globais de interligação da XC6200 possuem atrasos e desfasamentos reduzidos e podem ser utilizadas para distribuir sinais de controlo dos flip-flops espalhados por toda a FPGA e outros sinais de elevado *fan-out*. No entanto, quando um sinal passa por um flip-flop passará a usar um recurso de interligação geral (local ou hierárquico) do dispositivo. Os sinais de controlo têm normalmente de percorrer grandes distâncias ao longo do dispositivo e são aplicados a um grande número de células. Assim, os problemas de interligação podem ser minimizados através da utilização, sempre que possível, de uma interligação global para estes sinais. Este conceito está ilustrado na Figura 2.32 onde um sinal de controlo é distribuído por seis componentes do circuito. A implementação da Figura 2.32 (b) é preferível porque simplifica bastante a interligação. Embora utilize flip-flops adicionais, este é um custo que se pode desprezar comparando com o ganho em termos de recursos de interligação, até porque em muitos casos pode-se agrupar os flip-flops com outros componentes do circuito na mesma célula.

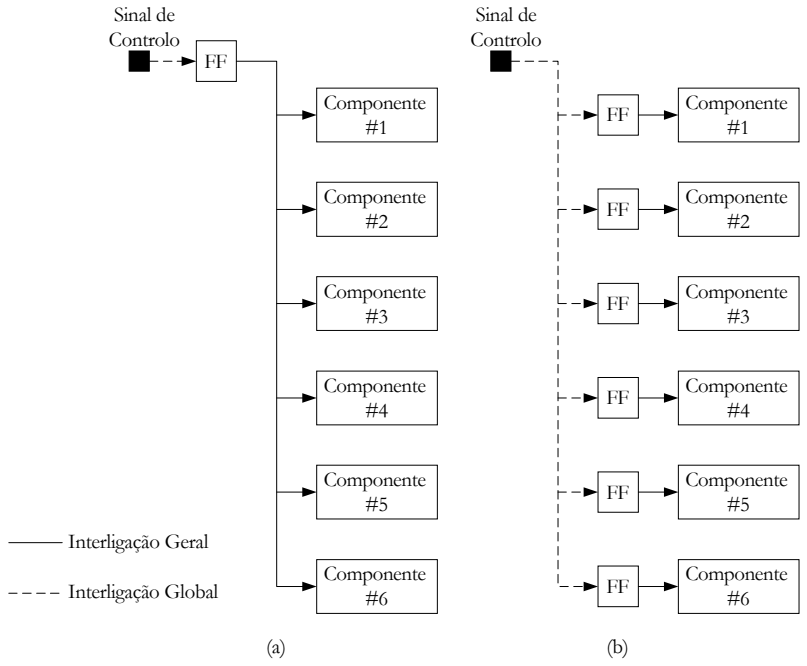


Figura 2.32 – Encaminhamento dos sinais de controlo dos flip-flops através de recursos de interligação (a) gerais; (b) globais.

Pipelining

Para melhorar o desempenho de um sistema os projectistas podem utilizar técnicas de *pipelining* através da adição de flip-flops para reduzir o comprimento dos caminhos críticos. Em projectos baseados na XC6200 pode-se usar o *pipelining* sem custos de hardware adicionais, uma vez que cada célula pode implementar simultaneamente uma função lógica de duas entradas e um flip-flop. Assim, em circuitos combinatórios onde sejam adicionados flip-flops para criar etapas de *pipeline*, as portas lógicas e os flip-flops podem ser agrupados na mesma unidade funcional de uma FPGA XC6200.

Recursos de Interligação e Implantação de Componentes

A estrutura matricial de uma FPGA XC6200 pode sugerir que seja possível implementar os componentes de um circuito em qualquer posição do dispositivo, mantendo as suas posições relativas e interligações. No entanto, a utilização dos recursos de interligação hierárquicos de comprimento 4, 16, 64 ou 128 células, das linhas *magic* e de controlo dos flip-flops das células, impõe algumas restrições à implantação dos componentes. Um circuito implementado com sucesso na extremidade de uma matriz de 4×4 ou 16×16 células poderá não ser implementável uma célula à esquerda ou à direita porque os recursos de interligação poderão não o permitir. Ou seja, dependendo do tipo de recursos utilizados por um componente, se as suas extremidades transpuserem os limites de um bloco de 4×4 ou 16×16 células, a utilização dos multiplexadores de controlo das ligações hierárquicas e de *magic* deixar-se-á de fazer da mesma forma, o que por sua vez implica a repetição da tarefa de interligação do circuito.

O mesmo problema coloca-se ao nível das linhas de controlo dos flip-flops, as quais são partilhadas por várias células alinhadas verticalmente. Isto significa que se um dado componente precisar, por exemplo, de um sinal de relógio específico, deve ocupar todas as células às quais esse sinal é aplicado e conter o seu multiplexador de controlo.

A solução trivial para estes problemas é utilizar apenas recursos de interligação locais. No entanto, como aumenta consideravelmente a dificuldade de implementação e os atrasos do circuito, para além de não resolver o problema dos sinais de controlo dos flip-flops, é também a solução menos aconselhável.

Assim, de uma forma geral, se pretendermos que um dado componente seja facilmente relocatável na FPGA, devemos assegurar que as suas dimensões sejam múltiplas do comprimento do maior recurso de interligação usado em ambas as direcções e ao mesmo tempo que as suas extremidades coincidam com os limites naturais de um bloco hierárquico da FPGA de forma a garantir que os multiplexadores de controlo dos recursos de interligação estejam contidos no componente ou pertençam às suas extremidades. Esta recomendação tem também vantagens importantes ao nível da reconfiguração dinâmica e parcial do dispositivo, uma vez que reduz a partilha de recursos entre os blocos que se pretende reconfigurar e aqueles que se pretende manter em funcionamento.

3 Modelos e Arquitecturas de Unidades de Controlo

Sumário

Neste capítulo é aprofundado o conceito de modelo apresentado no capítulo 1 desta dissertação. Mais concretamente, é apresentada uma caracterização baseada no domínio de aplicação dos modelos que complementa a classificação ortogonal baseada em níveis de granulosidade e tipos de representação atrás descrita. As características mais relevantes do sistema a projectar influenciam fortemente o tipo de formalismo utilizado na sua descrição e projecto. Assim, um dado modelo pode ser mais apropriado do que outro para descrever um determinado aspecto de um sistema, pelo que é importante classificá-los em termos da sua orientação preferencial, sendo as mais comuns as seguintes: o estado, as operações, a estrutura e os dados. No caso de um modelo cobrir mais do que um dos casos anteriores é designado por heterogéneo. Sendo os modelos orientados ao estado, os mais apropriados para descrever a funcionalidade das unidades de controlo, são a seguir apresentados alguns dos mais representativos, nomeadamente a máquina de estados finitos, a rede de Petri, a máquina de estados finitos hierárquica, a máquina de estados finitos paralela e hierárquica e a máquina de estados finitos virtual. A última parte deste capítulo é dedicada à discussão do conceito de arquitectura e à descrição das mais utilizadas na implementação de unidades de controlo, sendo apresentadas as seguintes: controlador, controlador hierárquico e controlador virtual. Ao contrário dos controladores ordinários (sem hierarquia), nos controladores hierárquicos e virtuais um dos aspectos importantes é a sua sincronização, pelo que também é dedicada alguma atenção a este problema. Na realidade, por uma questão de coerência com as noções e classificação de modelos apresentadas no capítulo 1, este capítulo deveria chamar-se "Modelos Comportamentais e Modelos Estruturais de Unidades de Controlo". No entanto, para simplificar a linguagem é utilizada a designação de modelo para modelo comportamental e de arquitectura para modelo estrutural.

3.1 Introdução

Tal como definido no capítulo 1, o projecto de um sistema é um processo composto por várias etapas, iniciando-se com a especificação, através da qual se define a sua funcionalidade e terminando com a depuração e teste necessários para averiguar se o funcionamento do sistema depois de construído corresponde ao descrito na sua especificação. Nas etapas intermédias de síntese são realizadas diversas tarefas com o objectivo de definir a sua constituição e realizar a sua implementação com base em componentes físicos já existentes ou desenvolvidos especificamente para o projecto em causa. A abordagem utilizada e a complexidade da etapa de especificação varia em função da dimensão e características do sistema a projectar. Consideremos como exemplo a seguinte questão relacionada com a especificação de um controlador de um elevador:

> Como descrever a sua funcionalidade com suficiente detalhe de forma a prever com exactidão a posição do elevador após terem sido premidos sequencialmente vários botões?

Uma das formas possíveis de responder a esta questão consiste na elaboração de um texto em linguagem natural (ex. Português) que descreva tão exaustivamente quanto possível o funcionamento do controlador. No entanto, a utilização de linguagem natural para especificar um sistema possui alguns problemas, nomeadamente o facto de serem normalmente ambíguas, incompletas, incapazes de descrever adequadamente detalhes importantes do sistema a projectar e também não permitirem a verificação da correcção da descrição quer através de métodos formais quer através de simulação. Por outro lado, estas limitações também se agravam à medida que aumenta a complexidade do sistema a especificar, podendo em certos casos a especificação tornar-se completamente ilegível. Assim, são necessárias abordagens mais precisas para especificação da funcionalidade do sistema.

A forma mais usual para alcançar o nível de precisão desejado é encarar o sistema como um conjunto de subsistemas ou blocos mais simples cada um representado por um modelo. Para este efeito existem várias formas de decomposição da funcionalidade global do sistema. O que basicamente distingue esses métodos são os tipos de blocos e as regras utilizadas na sua composição para descrever o funcionamento do sistema. Assim, cada tipo de modelo possui as suas primitivas básicas específicas, bem como regras de interligação. Para ser útil, um modelo deve ser:

- Formal, não devendo conter ambiguidade;
- Completo, de forma a poder descrever todo o sistema;
- Compreensível e natural para facilitar e não dificultar a compreensão do sistema;
- Flexível, uma vez que a certa altura do projecto poderá ser necessário modificar a funcionalidade do sistema;
- Extensível de forma a facilitar a adição de novas funcionalidades durante o projecto ou após a sua conclusão;

- Reutilizável para se poder integrar noutros projectos como um componente predefinido.

Um modelo consiste em objectos e regras de composição, é uma representação formal do sistema e é utilizado para descrever as suas características. Cada modelo permite decompor um sistema em blocos e gerar a sua especificação através da descrição desses blocos numa linguagem específica. Uma linguagem pode capturar diferentes modelos e um modelo pode ser capturado por diversas linguagens. O objectivo de um modelo é proporcionar uma visão abstracta do sistema. A Figura 3.1 mostra três modelos diferentes de um controlador de elevador. Na Figura 3.1 (a) encontra-se a descrição do seu funcionamento em linguagem natural, enquanto na Figura 3.1 (b) a representação é feita com um conjunto de instruções de programação e a Figura 3.1 (c) representa o controlador como uma máquina de estados.

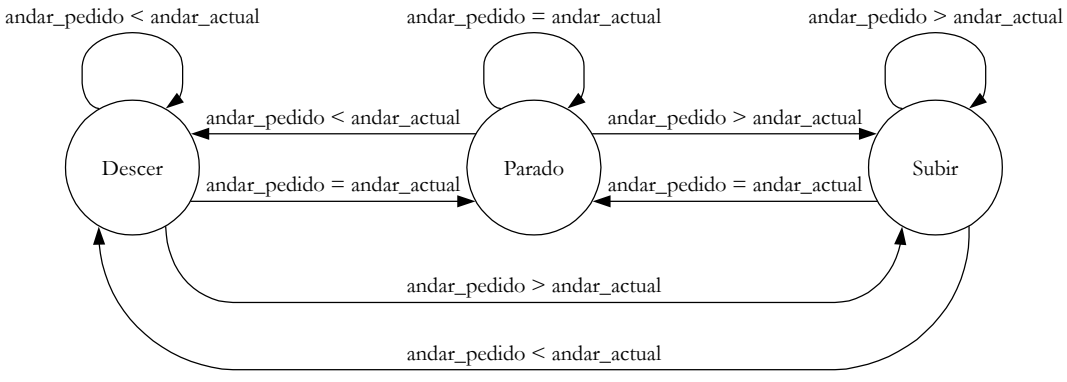
Se o elevador estiver parado e o andar onde tiver ocorrido o pedido for o andar actual do elevador, então este deve permanecer imóvel.

Se o elevador estiver parado e o andar onde tiver ocorrido o pedido estiver abaixo do andar actual do elevador então este deve descer até o andar requisitado.

Se o elevador estiver parado e o andar onde tiver ocorrido o pedido estiver acima do andar actual do elevador então este deve subir até o andar requisitado.

```
Loop

  If andar_pedido=andar_actual
    Acção = Parado
  Elsif andar_pedido<andar_actual
    Acção = Descer
  Elsif andar_pedido>andar_actual
    Acção = Subir

End Loop
```

(a) (b)



(c)

Figura 3.1 – Diferentes modelos de um controlador de elevador (a) linguagem natural (b) algorítmico (c) máquina de estados.

Cada um destes modelos é composto por um conjunto de objectos e respectivas interacções. Por exemplo, o modelo máquina de estados consiste num conjunto de estados e transições entre eles. Por outro lado, o modelo algorítmico consiste num conjunto de instruções que são executadas segundo uma sequência de controlo baseada num ciclo. A vantagem de possuirmos vários modelos disponíveis é cada um permitir representar o sistema de uma maneira diferente, o que possibilita a representação de diferentes características. Por exemplo, o modelo de máquina de estados é mais apropriado para representar o comportamento temporal do sistema, uma vez que permite expressar os estados e as transições causadas por eventos internos ou externos. Por outro lado, o modelo algorítmico não possui estados explícitos. No entanto, uma vez que pode especificar a relação entre entradas e saídas em termos de uma sequência de instruções, é apropriado para representar o sistema do ponto de vista procedimental. Tal como já foi referido no capítulo 1, de forma a realçar diferentes aspectos do sistema em diferentes fases do projecto, podem ser escolhidos diferentes modelos. Por exemplo, na fase de especificação, o projectista não possui conhecimentos para além da funcionalidade do sistema, pelo que tende a usar um modelo que não reflicta qualquer informação sobre a implementação. No entanto, na fase de implementação em que já existe informação disponível sobre os componentes do sistema, o projectista poderá utilizar um modelo que possa capturar a estrutura do sistema.

Por outro lado, diferentes domínios de aplicação requerem também diferentes modelos. Por exemplo, um sistema de tempo real e um sistema de bases de dados são modelados de forma diferente, uma vez que no primeiro se pretende realçar o comportamento temporal, enquanto no segundo a organização dos dados é a característica mais importante.

Tendo o projectista escolhido o modelo apropriado para especificar a funcionalidade do sistema, pode agora descrever em detalhe a operação do sistema. No entanto, o projecto não fica ainda completo, uma vez que o modelo não descreve a forma de como o sistema deve ser construído. Assim, o passo seguinte é a síntese do modelo, através da qual se realiza a sua transformação numa arquitectura, servindo para definir a implementação do modelo através da especificação do número e tipos de componentes bem como das suas interconexões. Como exemplo, na Figura 3.2 são apresentadas duas arquitecturas distintas que podem ser utilizadas para implementar o modelo de máquina de estados do controlador de elevador da Figura 3.1. A arquitectura da Figura 3.2 (a) é uma implementação ao nível lógico, que utiliza um registo para armazenar o estado actual e lógica combinatória para implementar as transições de estados e os valores dos sinais de saída. Na Figura 3.2 (b) pode ser visualizada a implementação ao nível do sistema, onde é feito o mapeamento do mesmo modelo de máquina de estados em software, usando uma variável num programa para representar o estado actual e instruções para calcular as transições de estado e os sinais de saída. Nesta arquitectura o programa é armazenado em memória e executado pelo processador. Ambas as arquitecturas implementam o circuito de controlo do elevador atrás descrito. No entanto, o exemplo da Figura 3.2 (b) é apresentado somente a título de exemplo, uma vez que neste trabalho estamos mais

interessados em implementações de circuitos de controlo em hardware e ao nível lógico (Figura 3.2 (a)), as quais passaremos a designar por unidades de controlo.

Tal como já foi referido no sumário, as designações de modelo e arquitectura são na realidade designações simplificadas para modelo comportamental e modelo estrutural, respectivamente. Assim, enquanto os modelos descrevem a operação do sistema, as arquitecturas descrevem a sua constituição. A síntese do sistema é o conjunto de tarefas que transformam um modelo numa arquitectura. No início do projecto somente a funcionalidade do sistema é conhecida. A função do projectista é descrever esta funcionalidade numa linguagem baseada nos modelos mais apropriados. À medida que o projecto evolui surgirá gradualmente uma arquitectura cujo nível de detalhe aumentará a cada etapa de projecto.

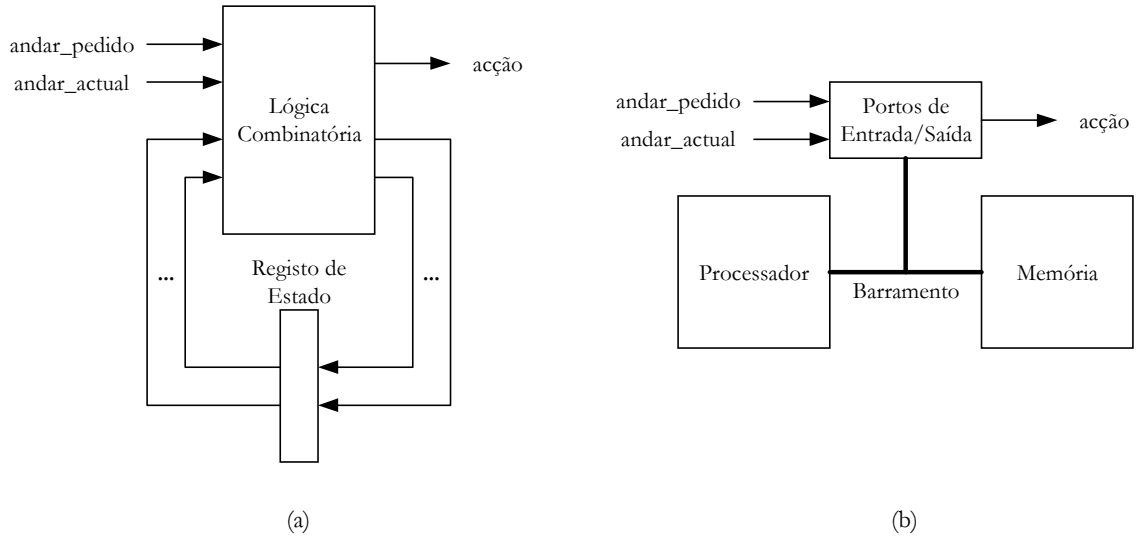


Figura 3.2 – Arquitecturas usadas na implementação de um controlador (a) implementação ao nível lógico (b) implementação ao nível do sistema.

Em geral existem arquitecturas mais eficientes para implementar um dado modelo do que outras. Adicionalmente, as tecnologias de implementação e fabricação poderão ter bastante influência na escolha de uma arquitectura. Assim, os projectistas têm de considerar várias implementações alternativas antes de completarem o processo de síntese.

Nas duas próximas secções deste capítulo vão ser considerados diversos modelos e apresentadas diversas arquitecturas úteis para o projecto de unidades de controlo.

3.2 Modelos

Os projectistas de sistemas utilizam vários modelos nas suas metodologias de projecto de hardware e software. Para além das duas formas de classificar um modelo descritas no capítulo 1, nomeadamente o tipo de representação e o nível de granulosidade, um modelo pode também ser classificado em termos do domínio de aplicação, ou seja, do tipo de características do sistema para as quais é mais adequado.

Segundo este ponto de vista, os modelos pertencem geralmente a uma das seguintes categorias [GajVahNarGon94]:

- Orientados aos estados;
- Orientados às operações;
- Orientados à estrutura;
- Orientados aos dados;
- Heterogéneos.

Enquanto as duas primeiras formas de classificação de modelos (tipo de representação e nível de granulosidade) são exclusivas dos modelos de hardware, esta é bastante mais geral, podendo ser aplicada a sistemas de hardware, software ou mistos como no caso dos sistemas embutidos.

Um modelo orientado aos estados, tal como uma máquina de estados finitos, representa o sistema como um conjunto de estados e um conjunto de transições entre eles disparadas por eventos externos. Esta classe de modelos é mais apropriada para sistemas de controlo, tais como sistemas reactivos de tempo real, onde o comportamento temporal do sistema é o aspecto mais importante do projecto.

Um modelo orientado às operações, tal como um grafo de fluxo de dados, descreve um sistema como um conjunto de operações relacionadas pelos dados ou dependências de execução. Este modelo é aplicado principalmente a sistemas de transformação, tais como de processamento digital de sinal, onde se realiza um conjunto de transformações a uma cadência fixa.

Um modelo orientado à estrutura (ex. diagrama de blocos) descreve os módulos físicos de um sistema e as suas interconexões. Ao contrário dos modelos orientados ao estado e às operações que reflectem principalmente a funcionalidade do sistema, os modelos orientados à estrutura concentram-se na constituição física do sistema. Alternativamente, os modelos orientados aos dados, tal como os diagramas entidade-relação, são úteis para representar o sistema como uma colecção de dados relacionados pelos seus atributos, tipo, etc. Este tipo de modelo é mais apropriado para sistemas de informação como bases de dados onde a função do sistema é menos importante do que a forma como os dados estão organizados. Finalmente, um projectista pode utilizar um modelo heterogéneo, ou seja, integrando várias características dos quatro tipos anteriores, sempre que precisar de representar um sistema complexo sob diferentes pontos de vista.

Como os modelos orientados ao estado são os mais indicados para o projecto de unidades de controlo, nas próximas secções são considerados alguns dos modelos mais populares deste tipo, começando pela máquina de estados finitos.

3.2.1 Máquina de Estados Finitos

Uma Máquina de Estados Finitos (*Finite State Machine – FSM*) [Booth67, Kohavi70, AleHan75] é um exemplo de um modelo orientado ao estado, sendo também o modelo mais popular para descrever unidades de controlo, uma vez que o seu comportamento temporal é representado naturalmente na forma de um conjunto de estados e de transições entre eles [GajVahNarGon94].

Basicamente, o modelo FSM consiste num conjunto de estados, num conjunto de transições entre estados e num conjunto de acções associadas aos estados ou transições. Formalmente uma FSM é definida pelo seguinte sêxtuplo:

$$f ::= <S, I, O, g : S \times I \rightarrow S, h : S \times I \rightarrow O, s_0>$$

onde $S = \{s_0, s_2, \ldots, s_U\}$ é o conjunto dos estados; $I = \{i_1, i_2, \ldots, i_J\}$ é o conjunto dos vectores de entrada definido a partir do conjunto dos sinais de entrada $X = \{x_1, x_2, \ldots, x_M\}$; $O = \{o_1, o_2, \ldots, o_K\}$ é o conjunto dos vectores de saída definido a partir do conjunto dos sinais de saída $Y = \{y_1, y_2, \ldots, y_N\}$; g é a função de transição que determina o estado seguinte a partir do estado actual e das entradas; h é a função de saída que determina as saídas também a partir do estado actual e das entradas; finalmente, s_0 é o estado inicial da FSM. De notar que uma FSM para além do estado inicial pode também possuir um ou vários estados finais que quando são alcançados a execução da FSM é dada por terminada. No entanto, como os estados finais são opcionais, não são em geral incluídos no tuplo de definição de uma FSM, pertencendo contudo ao conjunto S dos estados.

Na Figura 3.3 está representada graficamente uma FSM, na forma de um grafo dirigido e interpretado [MelSarTysYemZve98] normalmente designado por diagrama de transição de estados (*State Transition Diagram – STD*), que modela um controlador de elevador num edifício com três andares. Neste modelo, o conjunto de entradas $X = \{r_1, r_2, r_3\}$ representa o andar onde foi feito o pedido. Por exemplo, r_2 significa que houve um pedido do segundo andar. O conjunto de saídas $Y = \{d_2, d_1, n, u_1, u_2\}$ representa a direcção e o número de andares que o elevador se deve deslocar. Por exemplo, d_2 significa que o elevador deve descer dois andares, u_1 que o elevador deve subir um andar e *n* que deve permanecer imóvel. Neste caso, os conjuntos das entradas e saídas coincidem com os conjuntos dos respectivos vectores, ou seja, X = I e Y = O. Na Figura 3.3 podemos observar que se o elevador se encontrar no segundo andar (i. e. se o estado actual for s_2) e houver um pedido do primeiro andar, a saída deverá ser d_1.

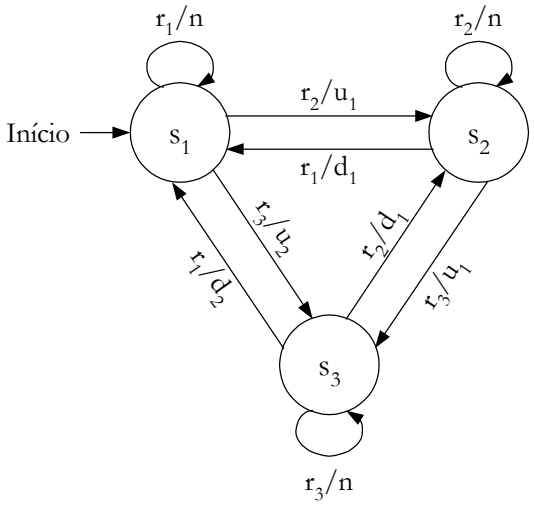


Figura 3.3 – Modelo FSM de *Mealy* para o controlador de elevador.

Existem dois tipos de FSMs que são bastante conhecidas e utilizadas: a FSM baseada em transições (Mealy) e FSM baseada em estados (Moore), as quais diferem essencialmente na definição da função de saída h. Numa FSM baseada em transições, as saídas dependem do estado actual e dos valores de entrada (h : S×I→O); numa FSM baseada em estados o valor das saídas depende somente do estado actual da FSM (h : S→O). Por outras palavras, numa FSM de Mealy as saídas estão associadas com as transições enquanto numa FSM de Moore as saídas estão associadas com os estados.

De notar que na Figura 3.3 foi utilizado o modelo baseado em transições para descrever o exemplo do controlador de elevador. O modelo baseado em estados para o mesmo controlador é mostrado na Figura 3.4 sendo neste caso o valor das saídas indicado no interior de cada estado.

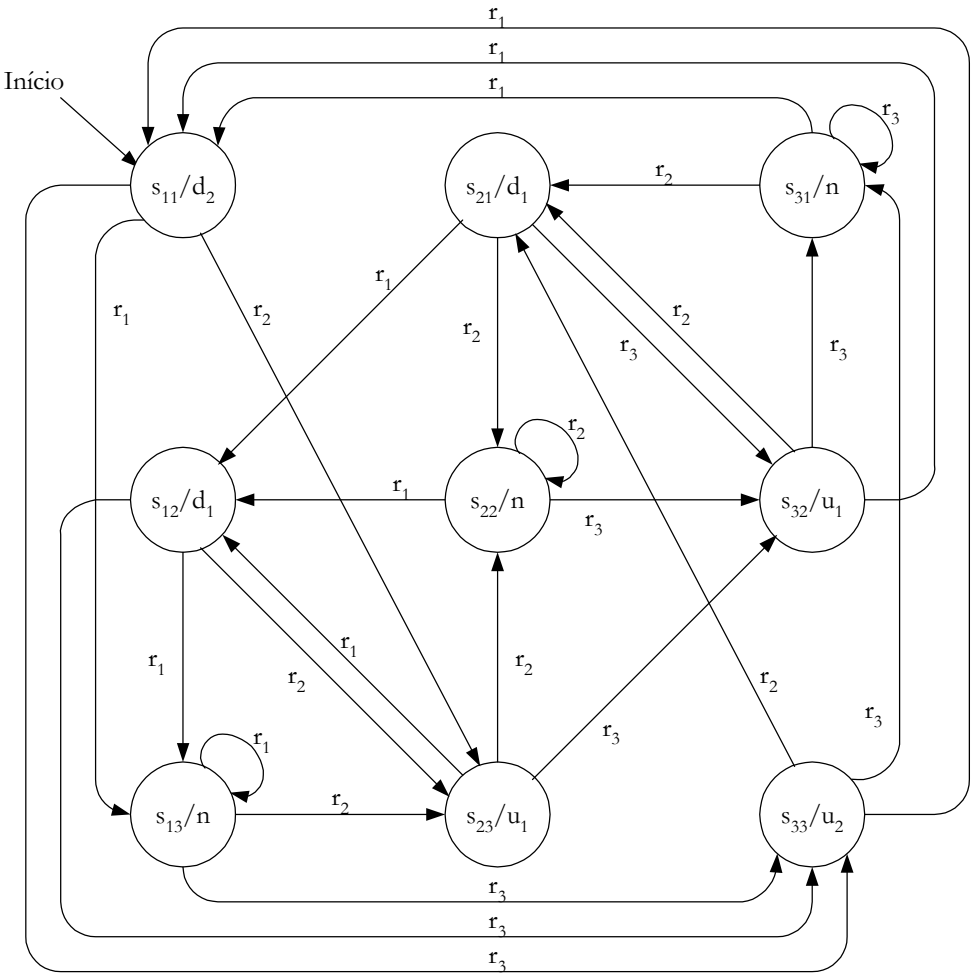


Figura 3.4 – Modelo FSM de *Moore* para o controlador de elevador.

Em termos práticos, a principal diferença entre estas duas variantes do modelo é que a FSM baseada em estados necessita geralmente de mais estados do que a baseada em transições. Isto deve-se ao facto de no modelo baseado em transições poderem partir do mesmo estado vários arcos cada um com um valor de saída diferente enquanto no modelo baseado em estados cada valor de saída distinto necessita do seu próprio estado, tal como no caso da Figura 3.4.

3.2.2 Máquina de Estados Finitos com Unidade de Execução

Nos casos em que a FSM necessite de operar sobre quantidades inteiras ou em vírgula flutuante que variem dentro de uma dada gama, pode ocorrer o problema da explosão do número de estados, uma vez que cada valor possível da quantidade requer o seu próprio estado. Isto poderá fazer com que a FSM possua um elevado número de estados. Por exemplo, um número inteiro de 16 bits pode representar 2^{16} ou 65536 estados diferentes. Existe uma forma relativamente simples de eliminar o problema da explosão dos estados, estendendo uma FSM com variáveis inteiras e/ou vírgula flutuante de forma a que cada variável substitua um número conveniente de estados [GajVahNarGon94]. Por exemplo, a introdução de uma variável de 16 bits, poderá reduzir o número de estados num modelo FSM de um factor de 65536. Este tipo de FSM estendida é chamada FSM com unidade de execução (*Finite State Machine with Datapath – FSMD*), sendo especificada a partir de [GajDutWuLin91]:

- Um conjunto de variáveis de armazenamento, V;
- Um conjunto de expressões, $E=\{f(x, y, z, \ldots) \mid x, y, z, \ldots \in V\}$;
- Um conjunto de operações de atribuição, $A=\{v \Leftarrow e \mid v \in V, e \in E\}$;
- Um conjunto de expressões de estado T, definidas como relações lógicas entre duas expressões do conjunto E, $T=\{Rel(a, b) \mid a, b \in E\}$.

Dadas estas definições uma FSMD pode ser definida como o sêxtuplo

$$<S, I \cup T, O \cup A, f, h, s_0>$$

onde relativamente ao modelo FSM, o conjunto dos vectores de entrada foi estendido para incluir as expressões de estado e o conjunto das saídas foi estendido para incluir as operações de atribuição. As funções f e h são agora definidas como os mapeamentos $S\times(I\cup T)\rightarrow S$ e $S\times(I\cup T)\rightarrow(O\cup A)$, respectivamente. Utilizando o modelo FSMD podemos descrever o exemplo do controlador de elevador da Figura 3.3 somente com um estado, tal como ilustrado na Figura 3.5, onde uma acção positiva corresponde a um movimento ascendente do elevador e o módulo ao número de andares que este se deve deslocar. Esta redução no número de estados é possível graças à utilização de uma variável para armazenar o valor do andar actual, eliminando assim a necessidade de um estado por andar.

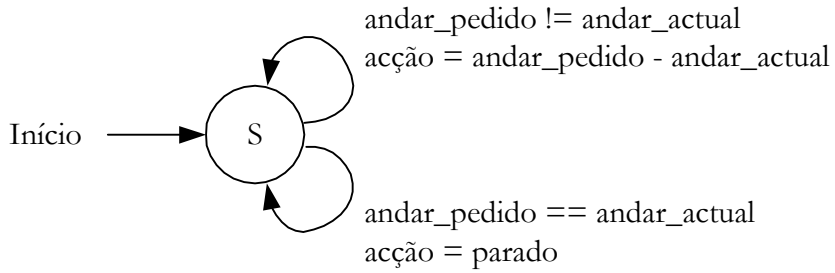


Figura 3.5 – Modelo FSMD do controlador de elevador.

Em geral, a FSM é apropriada para modelar sistemas predominantemente de controlo, enquanto a FSMD pode ser apropriada tanto para sistemas de controlo como de processamento. De facto, o modelo FSMD apesar de ser orientado ao estado, não é em rigor um modelo de unidades de controlo, mas sim um modelo de sistemas computacionais, podendo contudo ser utilizado para simplificar o projecto de uma unidade de controlo através do seu acoplamento a uma unidade de execução.

No entanto, nem o modelo FSM nem o FSMD são apropriados para sistemas complexos, já que nenhum dos dois suporta explicitamente concorrência e hierarquia. Sem o suporte explícito para concorrência um sistema complexo pode muito facilmente explodir em termos do número de estados. Vamos considerar como exemplo um sistema constituído por dois sub-sistemas concorrentes, cada um com 100 estados possíveis. Se tentarmos representar este sistema como uma única FSM ou FSMD, devemos representar todos os estados do sistema, os quais correspondem ao produto cartesiano entre os estados dos dois sub-sistemas e que são 100×100 = 10 000. Ao mesmo tempo, a ausência de hierarquia provocará o aumento do número de arcos. Se existirem por exemplo 100 estados, cada um com um arco correspondente à transição para um determinado estado, para um dado valor das entradas seriam precisos 100 arcos, ao contrário do que acontece num modelo hierárquico que agrupa os 100 estados num único, necessitando apenas de um arco. Assim, o problema dos modelos anteriores é tornarem-se incompreensíveis para os humanos quando o número de estados ou transições atinge valores da ordem das centenas.

3.2.3 Redes de Petri

O modelo de rede de Petri [Peterson81, Murata89, Reisig92] é outro tipo de modelo orientado ao estado que é útil para descrever e estudar sistemas constituídos por tarefas concorrentes e que interagem mutuamente.

O modelo de rede de Petri consiste num conjunto de lugares, um conjunto de transições e um conjunto de testemunhos. Os testemunhos encontram-se no interior dos lugares e circulam pela rede de Petri, sendo consumidos e produzidos sempre que ocorrer o disparo de uma transição. A sua função é simular as actividades dinâmicas e concorrentes do sistema.

Formalmente, uma rede de Petri é o quíntuplo [Murata89]:

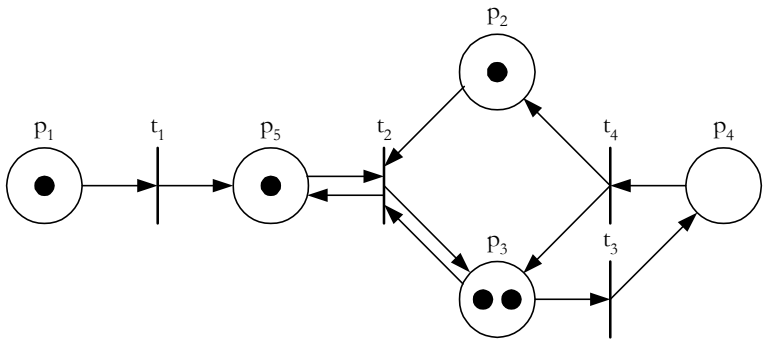
$$RP ::= <P, T, F, w, M_0>$$

onde $P=\{p_1, \ldots, p_M\}$ é um conjunto finito de lugares e $T=\{t_1, \ldots, t_N\}$ é um conjunto finito de transições, sendo P e T dois conjuntos disjuntos. $F \subseteq (P \times T) \cup (T \times P)$ é um conjunto de arcos entre os lugares e as transições, ou seja a relação de fluxo, $w: F \rightarrow N$, é uma função de ponderação que atribui pesos aos arcos de F e em que N é o conjunto dos números inteiros não negativos. Finalmente, $M_0: P \rightarrow N$ é a marcação inicial, ou seja, é o número inicial de testemunhos em cada lugar.

Na Figura 3.6 encontram-se as representações gráfica e textual de um exemplo de uma rede de Petri. Pode-se observar que a rede de Petri possui neste caso cinco

lugares (representados graficamente por círculos) e quatro transições (representadas graficamente por barras a cheio).

Um lugar p é entrada de uma transição t, quando é o primeiro elemento do par ordenado que define o arco entre p e t (p, t), caso contrário é o lugar de saída (t, p). No exemplo da Figura 3.6, os lugares p_2, p_3 e p_5 são entradas da transição t_2, e p_3 e p_5 lugares de saída da mesma transição. Uma transição sem qualquer lugar de entrada é chamada transição fonte e uma transição sem qualquer lugar de saída é denominada transição poço [Murata89]. A marcação inicial M_0 atribui um testemunho aos lugares p_1, p_2 e p_5 e dois testemunhos a p_3, sendo textualmente descrita por $M_0(p_1, p_2, p_3, p_4, p_5)=(1, 1, 2, 0, 1)$. Por defeito, o peso de um arco é 1.



Rede = (P, T, F, w, M_0)

P = {p_1, p_2, p_3, p_4, p_5}
T = {t_1, t_2, t_3, t_4}
F = {(p_1, t_1), (t_1, p_5), (p_5, t_2), (t_2, p_5), (p_2, t_2), (t_2, p_3), (p_3, t_2), (t_4, p_2), (t_4, p_3), (p_4, t_4), (p_3, t_3), (t_3, p_4)}
w = 1, $\forall$ f$\in$F
$M_0(p_1, p_2, p_3, p_4, p_5)$ = (1, 1, 2, 0, 1)

Figura 3.6 – Exemplo de uma rede de Petri.

Tal como mencionado acima, uma rede de Petri executa através do disparo de transições. Assim, para simular o comportamento dinâmico de um sistema, a marcação de uma rede de Petri altera-se segundo as seguintes regras de disparo de transições [Murata89]:

- Uma transição t é activada se cada lugar de entrada p de t estiver marcado com pelo menos w(p, t) testemunhos, onde w(p, t) é o peso do arco de p a t.
- Uma transição activa pode ou não disparar, dependendo da ocorrência ou não do evento associado;
- O disparo de uma transição t activa remove w(p, t) testemunhos de cada lugar de entrada p de t e adiciona w(t, p') testemunhos aos lugares p' de saída de t, onde w(t, p') é o peso do arco de t a p'.

Por exemplo, na Figura 3.6 após o disparo da transição t_2, assumindo que todos os arcos possuem peso 1, a marcação mudará para (1, 0, 2, 0, 1).

As redes de Petri são úteis porque permitem modelar de forma eficiente diversas características de um sistema. A Figura 3.7 ilustra os seguintes exemplos de características que podem ser representadas por uma rede de Petri:

- Sequenciação (Figura 3.7 (a)) – a transição t_1 é disparada após a transição t_2;
- Derivação não determinística (Figura 3.7 (b)) – apesar das transições t_1 e t_2 estarem ambas activas somente uma pode ser disparada;
- Sincronização (Figura 3.7 (c)) – a transição t_1 só pode ser disparada quando ambos os lugares de entrada possuírem testemunhos;
- Contenção de recursos (Figura 3.7 (d)) –as transições t_1 e t_2 competem pelo mesmo testemunho que reside num lugar de entrada comum a ambas;
- Concorrência (Figura 3.7 (e)) – as transições t_2 e t_3 podem ser disparadas em simultâneo. Mais concretamente, são modelados dois processos concorrentes, um produtor e um consumidor, sendo o testemunho existente no lugar central produzido por t_2 e consumido por t_3.

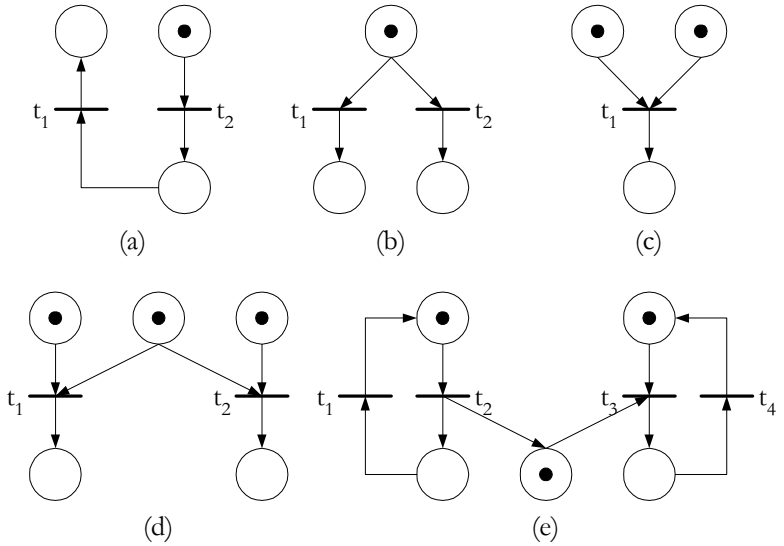


Figura 3.7 – Exemplos de redes de Petri que representam: (a) sequenciação, (b) derivação não determinística, (c) sincronização, (d) contenção de recursos, (e) concorrência.

Os modelos baseados em redes de Petri podem ser usados para testar e validar algumas propriedades importantes de um sistema, tais como a segurança e a vitalidade. Uma rede de Petri diz-se segura quando o número máximo de testemunhos em cada lugar é 1. Uma rede de Petri é limitada quando o número de testemunhos em cada lugar é limitado, ou seja o número de testemunhos da rede não pode crescer indefinidamente. De facto, não se pode construir uma rede de Petri na qual o número de testemunhos é ilimitado. Por outro lado, uma rede de Petri é viva se independentemente da marcação alcançada a partir de M_0, existir sempre pelo menos uma transição que pode disparar. Por esta razão, uma rede de Petri viva garante, independentemente da sequência de disparos escolhida, uma operação isenta de bloqueios (*deadlocks*).

Apesar das redes de Petri possuírem várias vantagens na modelação e análise de sistemas com concorrência, têm também limitações semelhantes às encontradas em modelos não hierárquicos como a FSM, podendo-se tornar rapidamente incompreensíveis à medida que a complexidade do sistema aumenta.

3.2.4 Máquina de Estados Finitos Hierárquica

A Máquina de Estados Finitos Hierárquica (*Hierarchical Finite State Machine – HFSM*) [SklFer98, Sklyarov99] é uma extensão do modelo FSM com suporte para descrições hierárquicas. Isto possibilita a construção modular de modelos de máquinas de estados, permitindo por um lado dividir uma FSM em máquinas mais pequenas através da decomposição descendente e sucessiva do modelo inicial e, por outro, construir FSMs mais complexas através da montagem ascendente e progressiva de outras mais simples. Tanto a decomposição descendente como a montagem ascendente possuem vantagens importantes no desenvolvimento de sistemas complexos. A primeira permite realizar o refinamento progressivo do modelo inicial e a abstracção dos detalhes considerados não relevantes em cada etapa do projecto. A segunda permite reutilizar componentes previamente desenvolvidos e testados como macro-blocos predefinidos em novos projectos.

Tal como uma FSM, o modelo HFSM consiste num conjunto de estados, num conjunto de transições e num conjunto de acções associadas aos estados ou transições. No entanto, ao contrário da FSM, existem neste caso dois tipos de estados: os atómicos e os macroestados. Os primeiros são em tudo idênticos aos encontrados numa FSM ordinária, não sendo portanto decomponíveis noutros estados e podendo no caso da máquina de Moore possuir acções associadas. Os macroestados são decomponíveis noutros estados, transições e acções, encapsulando portanto uma FSM pertencente a um nível hierárquico inferior. Quando ocorre uma transição para um macroestado, diz-se que a FSM actual invoca a FSM encapsulada por esse macroestado. Quando a FSM invocada termina a sua execução, isto é, quando transita para o seu estado final, diz-se que o controlo retorna à FSM invocadora.

A semântica das acções associadas aos macroestados numa máquina de Moore e às transições que deles partem numa máquina de Mealy pode levantar algumas dúvidas. Mais concretamente, não é claro se uma acção associada ao macroestado deva estar activa durante todo o intervalo de tempo em que este se encontrar activo, isto é, a execução se encontrar num dos estados da FSM que o descreve, ou somente durante o seu estado inicial. No caso de uma máquina de Mealy a situação é análoga. Assim, para evitar ambiguidades, ao contrário de um estado atómico, um macroestado não pode possuir acções associadas, devendo estas ser colocadas em cada um dos estados ou transições que o constituem. De notar que numa máquina de Mealy, as transições com partida num macroestado também não possuem acções associadas, devendo estas ser colocadas na transição para o estado final da FSM respectiva.

Uma HFSM consiste num conjunto de FSMs relacionadas hierarquicamente. No nível hierárquico superior, encontra-se a FSM principal. Cada um dos seus macroestados é ele próprio definido por uma HFSM. O processo de definição é aplicado sucessivamente aos diferentes níveis hierárquicos e termina quando mais nenhuma FSM possuir macroestados.

Enquanto numa FSM não é obrigatória a existência de um estado final, numa HFSM todas as FSMs que definem os seus macroestados devem possuir um estado de entrada (inicial) e um estado de saída (final) bem definidos. Consequentemente, se a FSM residente no nível de hierarquia mais elevado fizer parte de uma cadeia de invocações recursivas, ou seja, se for encapsulada por um macroestado em qualquer

nível hierárquico, deve também ela possuir um estado final. Assim, um estado final na sub-máquina principal de uma HFSM pode não ter exactamente o mesmo significado que numa FSM ordinária, uma vez que dependendo do nível a que esta esteja a executar podemos pretender que a mesma termine, reinicie a sua execução ou simplesmente retorne à FSM invocadora.

De notar que a mesma FSM pode ser associada a vários macroestados pertencentes a diversas FSMs distribuídas por vários níveis hierárquicos. Assim, por conveniência vamos definir o conjunto $F = \{f_1, f_2, \ldots, f_P\}$ das várias FSMs $f_1, f_2, \ldots, f_P$ que constituem a HFSM, sendo a FSM principal representada pelo elemento f_1.

Com base nos conceitos apresentados, uma HFSM é representada pela sua FSM principal (de nível superior), a qual é definida formalmente pelo seguinte tuplo:

$$f_1 ::= <S_1, T_1, I_1, O_1, g_1, h_1, s_{1b}, s_{1e}>$$

no caso geral, cada sub-máquina de uma HFSM é o tuplo:

$$f_p ::= <S_p, T_p, I_p, O_p, g_p, h_p, s_{pb}, s_{pe}>$$

onde $S_p = \{s_{p1}, s_{p2}, \ldots, s_{pU}\}$ é o conjunto dos estados atómicos; $T_p = \{t_{p1}, t_{p2}, \ldots, t_{pV}\}$ é o conjunto dos macroestados; $I_p = \{i_{p1}, i_{p2}, \ldots, i_{pJ}\}$ é o conjunto dos vectores de entrada definido a partir do conjunto dos sinais de entrada $X_p = \{x_{p1}, x_{p2}, \ldots, x_{pM}\}$; $O_p = \{o_{p1}, o_{p2}, \ldots, o_{pK}\}$ é o conjunto dos vectores de saída definido a partir do conjunto dos sinais de saída $Y_p = \{y_{p1}, y_{p2}, \ldots, y_{pN}\}$; g_p é a função de transição de estado, definida como o mapeamento $g_p : (S_p \cup T_p) \times I_p \rightarrow (S_p \cup T_p)$ e que determina o estado seguinte a partir do estado actual e das entradas; h_p é a função de saída, definida como o mapeamento $h_p : S_p \times I_p \rightarrow O_p$ e que determina as saídas também a partir do estado actual e das entradas; finalmente, s_{pb} e s_{pe} são respectivamente os estados inicial e final da FSM. De notar que $s_{pb}, s_{pe} \in S_p$.

Enquanto os conjuntos S_p e T_p e as funções g_p e h_p são específicas de cada FSM, os conjuntos X_p, Y_p, I_p e O_p podem ser comuns ou possuírem partes comuns a várias FSMs, já que diferentes FSMs podem ler as mesmas entradas e controlar as mesmas saídas.

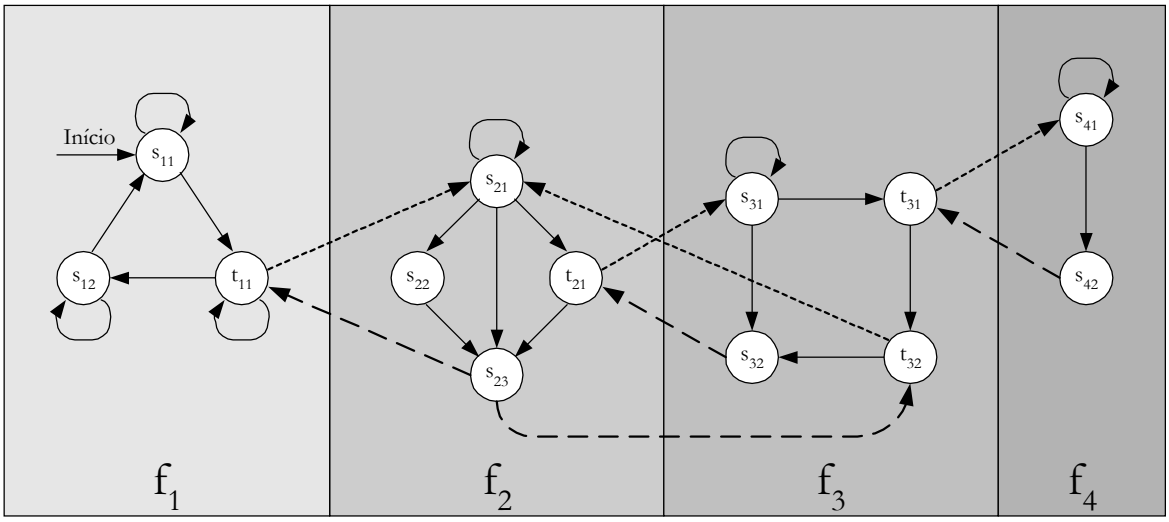
A relação entre uma dada FSM f_p e as FSMs por esta invocadas é estabelecida pelos seus macroestados e expressa através da seguinte expressão:

$$t_{pv} ::= f_{p'}$$

$$\text{em que } t_{pv} \in T_p \text{ e } f_{p'} \in F$$

Na Figura 3.8 estão ilustradas as representações gráfica e textual de uma HFSM. Por questões de simplicidade e uma vez que se pretende demonstrar somente as relações hierárquicas entre as diversas FSMs, os estados e as transições não foram etiquetados com os vectores de entrada e saída respectivos. De notar que no exemplo apresentado a FSM principal (f_1) não possui estado final.

Para concluir este tópico, gostaríamos de realçar que a definição formal aqui discutida difere significativamente da apresentada em [Sklyarov99], uma vez que utiliza uma abordagem multi-nível e recursiva em que cada macroestado é ele próprio definido como uma HFSM, enquanto em [Sklyarov99] é utilizada uma definição planar.



$F = \{f_1, f_2, f_3, f_4\}$ ----------► Invocação — — — —► Retorno

$S_1 = \{s_{11}, s_{12}\}$	$S_2 = \{s_{21}, s_{22}, s_{23}\}$	$S_3 = \{s_{31}, s_{32}\}$	$S_4 = \{s_{41}, s_{42}\}$
$T_1 = \{t_{11}\}$	$T_2 = \{t_{21}\}$	$T_3 = \{t_{31}, t_{32}\}$	$T_4 = \varnothing$
$t_{11} = f_2$	$t_{21} = f_3$	$t_{31} = f_4$	$s_{4b} = s_{41}$
$s_{1b} = s_{11}$	$s_{2b} = s_{21}$	$t_{32} = f_2$	$s_{4e} = s_{42}$
s_{1e} não existe	$s_{2e} = s_{23}$	$s_{3b} = s_{31}$	
		$s_{3e} = s_{32}$	

Figura 3.8 – Exemplo de um modelo HFSM.

3.2.5 Máquina de Estados Finitos Paralela

O modelo HFSM permite ultrapassar algumas das limitações dos modelos FSM e FSMD na representação de sistemas complexos, através de uma abordagem de construção progressiva e multi-nível e do agrupamento hierárquico dos estados e consequente diminuição do número de arcos. No entanto, uma HFSM não é ainda capaz de resolver o problema da explosão do número de estados que ocorre quando se descrevem sistemas constituídos por vários sub-sistemas concorrentes com modelos FSM tradicionais. Nestes casos são necessários modelos que suportem a descrição de concorrência de uma forma explícita. O modelo de máquina de estados finitos paralela (*Parallel Finite State Machine – PFSM*) [Sklyarov87] é um exemplo deste tipo de modelos.

Conceptualmente, uma PFSM consiste num conjunto de FSMs ordinárias relativamente autónomas, que funcionam em simultâneo e que interagem mutuamente para efeitos de transferência de informação e sincronização.

Formalmente, uma PFSM é um conjunto $F = \{f_1, f_2, \ldots, f_P\}$ de FSMs ordinárias. Ao contrário do modelo HFSM, no qual somente uma FSM se encontra

activa de cada vez, no modelo PFSM todas as FSMs se encontram activas simultaneamente. Cada FSM f_1, f_2, …, f_P é definida pelo sêxtuplo:

$$f_p ::= <S_p, I_p, O_p, g_p : S_p \times I_p \rightarrow S_p, h_p : S_p \times I_p \rightarrow O_p, s_{p0}>$$

Cada um dos elementos do tuplo é definido da mesma forma que numa FSM ordinária, ou seja, $S_p = \{s_{p0}, s_{p2}, \ldots, s_{pU}\}$ é o conjunto dos estados; $I_p = \{i_{p1}, i_{p2}, \ldots, i_{pJ}\}$ é o conjunto dos vectores de entrada definido a partir do conjunto dos sinais de entrada $X_p = \{x_{p1}, x_{p2}, \ldots, x_{pM}\}$; $O_p = \{o_{p1}, o_{p2}, \ldots, o_{pK}\}$ é o conjunto dos vectores de saída definido a partir do conjunto dos sinais de saída $Y_p = \{y_{p1}, y_{p2}, \ldots, y_{pN}\}$; g_p é a função de transição que determina o estado seguinte a partir do estado actual e das entradas; h_p é a função de saída que determina as saídas também a partir do estado actual e das entradas; finalmente, s_{p0} é o estado inicial da FSM.

A possível interacção entre as várias FSMs é representada pela existência de linhas de entrada e saída em comum, ou de forma equivalente pela seguinte relação entre os vectores de entrada e de saída de duas sub-máquinas distintas:

$$I_p \cap O_{p'} \neq \varnothing, \; p \neq p'$$

De notar que no caso da PFSM ser constituída por apenas uma FSM, a definição de PFSM reduz-se à definição da FSM ordinária.

Nesta secção não é apresentada nenhuma representação gráfica de uma PFSM uma vez que os STDs não suportam a descrição de estados e FSMs concorrentes. Na próxima secção é introduzida a linguagem *Statecharts*, a qual permite expressar de forma natural estes conceitos. Os Esquemas de Grafos Hierárquicos (*Hierarchical Graph Schemes – HGSs*) [Sklyarov84], apresentados no próximo capítulo, também suportam a descrição de hierarquia e paralelismo em modelos orientados ao estado.

3.2.6 Máquina de Estados Finitos Paralela e Hierárquica

A máquina de estados finitos paralela e hierárquica (*Hierarchical and Parallel Finite State Machine – HaPFSM*) é essencialmente uma extensão ao modelo FSM que adiciona suporte para hierarquia e paralelismo, combinando assim as características dos modelos HFSM e PFSM num só. Desta forma é possível eliminar a tendência para a explosão do número de estados e transições que pode ocorrer quando se descrevem sistemas hierárquicos e concorrentes com modelos FSM tradicionais [GajVahNarGon94, Rocha99].

Tal como uma FSM, o modelo HaPFSM consiste num conjunto de estados, um conjunto de transições e um conjunto de acções associadas aos estados ou às transições. No entanto, ao contrário da FSM, e tal como numa HFSM, os estados podem ser atómicos ou macroestados, ou seja, decomponíveis num conjunto de subestados, modelando assim a hierarquia. Além disso, de forma análoga a uma PFSM, cada estado pode também ser decomposto em subestados concorrentes que executam em paralelo. Neste modelo as transições podem ser estruturadas ou não estruturadas [GajVahNarGon94]. As primeiras só são permitidas entre dois estados no mesmo nível

da hierarquia enquanto as segundas podem ocorrer entre quaisquer dois estados independentemente da sua relação hierárquica.

Este modelo apesar de bastante poderoso é também consideravelmente mais complexo que os anteriores, pelo que a sua definição formal é também mais elaborada. Além disso, do ponto de vista prático poderá ser interessante considerar algumas simplificações ao modelo, que apesar de o tornarem mais restritivo, torná-lo-ão mais tratável, mas ainda assim adequado para muitas aplicações. Consideremos então as seguintes três variantes do modelo HaPFSM:

- Máquina de Estados Finitos Paralela e Hierárquica – esta é na realidade a variante à qual não são impostas quaisquer restrições. Para se distinguir das restantes vamos designá-la apenas por Máquina de Estados Finitos Generalizada (*Generalized Finite State Machine –GFSM*). Este modelo é composto inicialmente por um número qualquer de máquinas paralelas que podem invocar hierarquicamente outras máquinas. Por sua vez, estas podem também executar diversas máquinas em paralelo. Resumindo, numa GFSM podem existir várias HFSMs e PFSMs activas simultaneamente;
- Máquina de Estados Finitos Paralela Hierárquica (*Hierarchical Parallel Finite State Machine – HPFSM*) – nesta variante restringe-se a existência de paralelismo ao primeiro nível hierárquico. Por outras palavras, uma HPFSM é uma máquina paralela em que cada uma das suas sub-máquinas é uma máquina hierárquica. Se considerarmos uma estrutura em árvore que representa a operação de uma HPFSM, a máquina paralela ocupa a raiz da árvore. Neste caso só pode existir uma PFSM activa ou várias HFSMs activas em simultâneo;
- Máquina de Estados Finitos Hierárquica Paralela (*Parallel Hierarchical Finite State Machine – PHFSM*) – ao contrário do caso anterior em que a máquina paralela ocupava a raiz da árvore que representa a operação de uma HPFSM, numa PHFSM existe somente uma HFSM, sendo as folhas da árvore ocupadas pelas máquinas paralelas. Neste caso só pode existir uma HFSM ou PFSM activa em cada momento.

Com base nestes conceitos a definição formal das três variantes de HaPFSM resulta da combinação das definições formais de FSM, HFSM e PFSM.

Uma linguagem particularmente apropriada para capturar qualquer uma destas variantes é a *Statecharts* [Harel87], uma vez que pode suportar facilmente as noções de hierarquia, paralelismo e comunicação entre estados concorrentes. Os *Statecharts* utilizam transições não estruturadas e um sistema de comunicação por difusão, no qual os eventos emitidos por qualquer estado podem ser detectados por todos os outros.

A linguagem *Statecharts* é uma linguagem gráfica. A sua descrição será feita no próximo capítulo, sendo aqui apresentadas somente as ideias fundamentais para ilustrar os conceitos de hierarquia e paralelismo deste modelo.

A Figura 3.9 mostra um exemplo de uma HaPFSM descrita com *Statecharts*. Os rectângulos arredondados são utilizados para representar os estados em qualquer nível e o encapsulamento para expressar uma relação hierárquica entre estes estados. As linhas a tracejado entre estados representam concorrência e as setas denotam as

transições entre os estados, sendo cada uma etiquetada com um evento e opcionalmente com uma condição e/ou acção dentro de parêntesis. Na Figura 3.9 podemos também observar que o estado Y está decomposto em dois estados concorrentes, A e D. Por seu lado, o primeiro consiste em dois subestados B e C enquanto o segundo inclui os subestados E, F e G. Os pontos a cheio na figura indicam os pontos de entrada de cada estado.

De acordo com a linguagem *Statecharts*, quando o evento b ocorre, enquanto no estado C, o estado A transitará para o estado B. Se, por outro lado, o evento a ocorrer enquanto no estado B, transitará para o estado C mas somente se a condição P se verificar no instante da ocorrência. Durante a transição de B para C, a acção c associada com a transição será realizada.

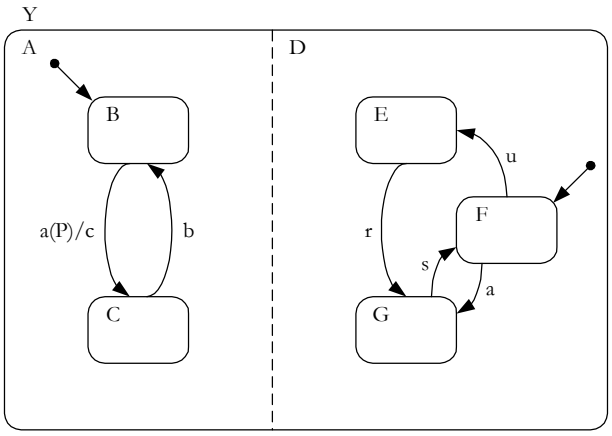


Figura 3.9 – Representação de uma HaPFSM com a linguagem *Statecharts*.

Devido ao suporte para hierarquia e paralelismo, o modelo HaPFSM é apropriado para representar sistemas de controlo complexos. No entanto, o problema deste modelo é o facto de que, à semelhança de outros modelos orientados ao estado, concentrar-se exclusivamente na modelação da componente de controlo, o que significa que só se pode associar com as suas transições ou estados, acções muito simples, tais como atribuições de valor. Consequentemente, o modelo HaPFSM não é apropriado para modelar certas características de sistemas complexos, que requerem estruturas de dados complexas ou podem realizar em cada estado acções complexas e arbitrárias. Para esses sistemas este modelo por si só será provavelmente insuficiente, sendo conveniente a sua combinação com outro mais apropriado.

3.2.7 Máquina de Estados Finitos Virtual

O modelo FSM e seus derivados, tais como o HFSM, PFSM e HaPFSM, usados na descrição de unidades de controlo são normalmente utilizados de forma inflexível. Isto significa que uma vez terminado o projecto as características do circuito resultante não são facilmente modificáveis, isto é, qualquer alteração ou adição de funcionalidade obriga à repetição de todos os passos de projecto. Em muitos casos é importante tornar o circuito projectado flexível e extensível de forma a permitir actualizações futuras, nalguns casos durante a sua fabricação mas cada vez mais depois da sua aquisição pelo cliente e muitas das vezes em pleno funcionamento. Como é a unidade

de controlo que estabelece a sequência de operações realizadas por um dado circuito ou equipamento e os modelos atrás descritos não permitem atingir estes objectivos, torna-se necessária a adopção de outros modelos mais apropriados.

Um dos modelos mais indicados para este fim é designado por Máquina de Estados Finitos Virtual (*Virtual Finite State Machine – VFSM*). Este modelo é baseado nos anteriores e diz-se que uma FSM é virtual quando a sua funcionalidade ou mais concretamente um ou vários parâmetros, tais como o conjunto dos estados e dos vectores de entrada/saída assim como as funções de transição e de saída podem ser alteradas facilmente, ou seja, sem que seja necessário repetir todos os passos de projecto. Se a modificação puder ser realizada sem que para tal seja necessário suspender a operação da máquina, a VFSM designa-se por VFSM dinâmica. Se, por outro lado, for necessário interromper a sua operação a VFSM designa-se por VFSM estática.

Quando comparada com uma FSM tradicional (não virtual) uma VFSM possui vantagens importantes, das quais se destacam:

- A flexibilidade, permitindo alterar o comportamento da unidade de controlo, bem como corrigir erros de projecto durante ou após a conclusão do seu desenvolvimento, sendo para tal necessários tempo e esforço mínimos;
- A extensibilidade, a qual torna possível adicionar funcionalidades ao circuito após o projecto ter sido concluído;
- A reutilização de hardware, fazendo com que os mesmos recursos físicos possam ser utilizados para implementar diferentes componentes da unidade de controlo que não sejam necessárias simultaneamente.

Conceptualmente a passagem de um modelo tradicional para um modelo virtual é bastante simples, sendo necessárias apenas algumas considerações bastantes abstractas. O aspecto fundamental das unidades de controlo virtuais reside nas suas arquitecturas de implementação, as quais serão discutidas mais à frente.

Apesar de teoricamente o modelo VFSM poder ser baseado em qualquer dos anteriores, na prática é preferível adoptar modelos hierárquicos. A razão é bastante simples; como estes decompõem a máquina global em sub-máquinas mais simples, relativamente independentes e na generalidade dos casos com um funcionamento em intervalos de tempo distintos, torna-se muito mais fácil alterar uma máquina inactiva num dado instante, do que uma em pleno funcionamento. Este facto deve-se à dificuldade de em muitos casos estabelecer a correspondência entre uma alteração ao nível comportamental e a respectiva alteração ao nível do circuito, devido às técnicas de síntese e optimização utilizadas no projecto do sistema.

A virtualização de uma FSM (HFSM/HaPFSM) pode ser realizada de várias formas, correspondendo cada uma à modificação dos elementos do seu tuplo de definição:

- Adição ou remoção de elementos do conjunto F, ou seja, de sub-máquinas dentro da HFSM;
- Alteração das funções g e h, ou seja das transições entre estados e das saídas respectivamente;

- Modificação dos conjuntos de vectores de entrada (I) e saída (O) usados pelas funções g e h;
- Adicionar ou remover estados dos conjuntos S e T (estados atómicos ou macroestados respectivamente);
- Alterar os estados iniciais e finais de cada sub-máquina.

Do ponto de vista formal uma VFSM possui a mesma representação que o modelo tradicional subjacente, com a excepção de que todos os elementos do tuplo de definição podem ser alterados dinamicamente.

A Figura 3.10 ilustra as representações gráfica e textual de uma VFSM obtida por modificação do exemplo da Figura 3.8, onde se mostram as seguintes transformações: (a) modificação do estado inicial, (b) alteração de um macroestado; (c) adição de um estado atómico; (d) remoção de um estado atómico; (e) substituição de uma sub-máquina.

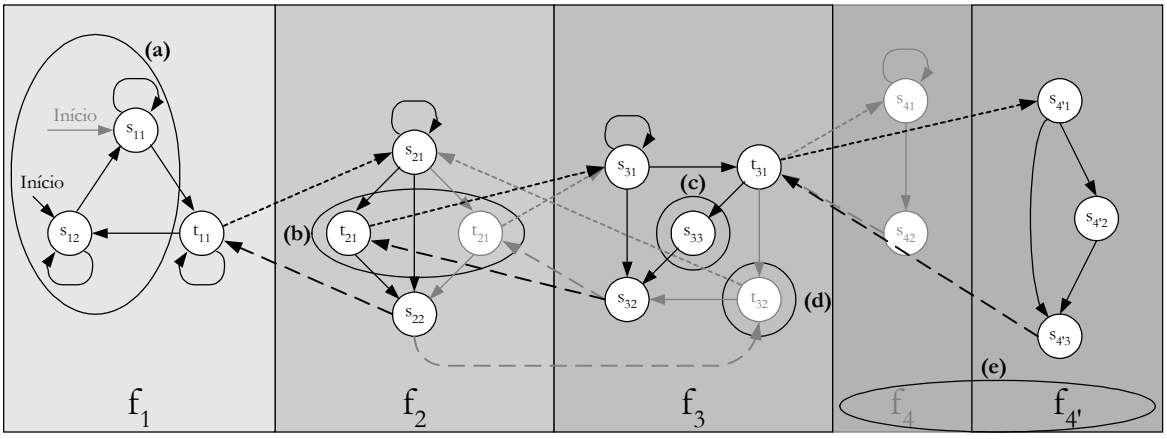


Figura 3.10 – Exemplo de um modelo VFSM.

3.3 Arquitecturas

Depois da funcionalidade do sistema e mais concretamente da unidade de controlo ter sido capturada usando um dos modelos comportamentais acima apresentados, o passo seguinte consiste na execução das tarefas de síntese do circuito, ou seja, na transformação do modelo comportamental numa arquitectura ou modelo estrutural que descreva a forma como o sistema deve ser implementado. Uma arquitectura deve descrever a estrutura do sistema, mais concretamente, o número e tipo de componentes, bem como as suas interconexões.

De uma forma geral, as arquitecturas de sistemas electrónicos podem variar desde simples unidades funcionais e controladores até processadores com uma elevada

capacidade de cálculo. As arquitecturas podem ser específicas da aplicação ou de uso geral. Exemplos das primeiras são os coprocessadores e os processadores específicos, tais como os processadores digitais de sinal (*Digital Signal Processors - DSPs*). Os exemplos mais populares das arquitecturas de uso geral são os processadores CISC (*Complex Instruction Set Computer*) e RISC (*Reduced Instruction Set Computer*).

Nas próximas subsecções vão ser apresentados exemplos de arquitecturas utilizadas para implementar unidades de controlo em hardware. Estas arquitecturas são consideradas dependentes da aplicação, uma vez que as unidades de controlo são normalmente específicas de cada projecto. As arquitecturas aqui apresentadas estão vocacionadas para implementar alguns dos modelos orientados ao estado atrás descritos.

3.3.1 Controlador

Uma das arquitecturas mais simples destinada a implementar modelos orientados ao estado é a do controlador. Esta implementa de forma directa o modelo FSM definido pelo sêxtuplo <S, I, O, g, h, s_0> [GajVahNarGon94]. Tal como mostrado na Figura 3.11, um controlador consiste num registo e dois circuitos ou blocos de lógica combinatória. O registo é normalmente designado por registo de estado e é utilizado para armazenar um dos estados de S. Os blocos combinatórios denominados por Função de Transição e Função de Saída implementam as funções g e h, respectivamente. A entradas e saídas do controlador são sinais Booleanos, sobre os quais são definidos os conjuntos I e O. O sinal de inicialização é responsável por carregar no registo de estado o estado inicial s_0. As transições entre estados são sincronizadas por um sinal de relógio.

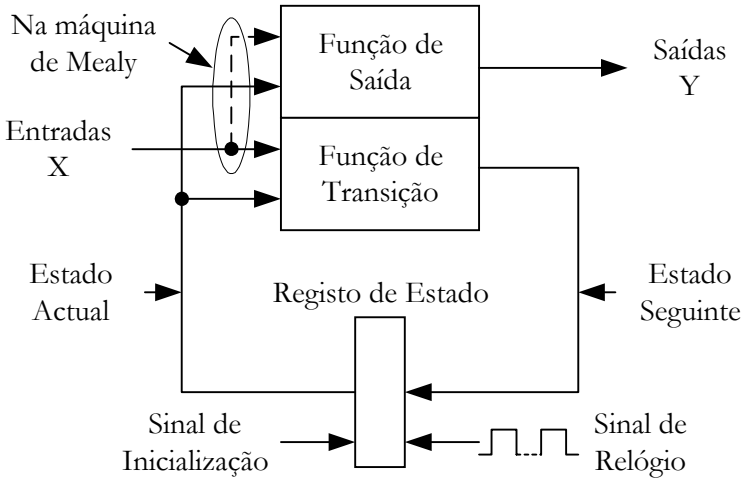


Figura 3.11 – Arquitectura de um controlador.

Tal como mencionado acima, existem dois tipos distintos de FSMs, as baseadas em transições (Mealy) e as baseadas em estados (Moore). Estes tipos diferem na forma como é definida a função h e afectam a ligação das entradas na arquitectura (ver Figura 3.11). Para controladores baseados em transições, h é definida como o mapeamento $S \times I \rightarrow O$, o que significa que a função de saída depende de duas variáveis, o estado

actual e as entradas. Por outro lado, para controladores baseados em estados, h é definida como o mapeamento S→O, o que significa que a função de saída depende somente do estado actual. Uma vez que tanto as entradas como as saídas são sinais Booleanos, esta arquitectura é apropriada para implementar controladores que não requeiram manipulações complexas de dados. A síntese do controlador consiste na minimização e codificação de estados, minimização lógica e mapeamento na tecnologia das funções de estado seguinte e de saída.

3.3.2 Controlador com Unidade de Execução

Tal como o próprio nome sugere, esta arquitectura combina um controlador com unidade de execução [GajVahNarGon94]. A Figura 3.12 ilustra o diagrama de blocos desta arquitectura, a qual implementa o modelo FSMD atrás descrito. De notar que, da mesma forma que uma FSMD não é em rigor um modelo de unidades de controlo, um controlador com unidade de execução não é uma arquitectura para implementação de unidades de controlo no sentido clássico. Contudo, esta é utilizada para facilitar a implementação de unidades de controlo sempre que estas necessitem de realizar manipulações complexas de dados que dificilmente seriam realizáveis por um controlador ordinário.

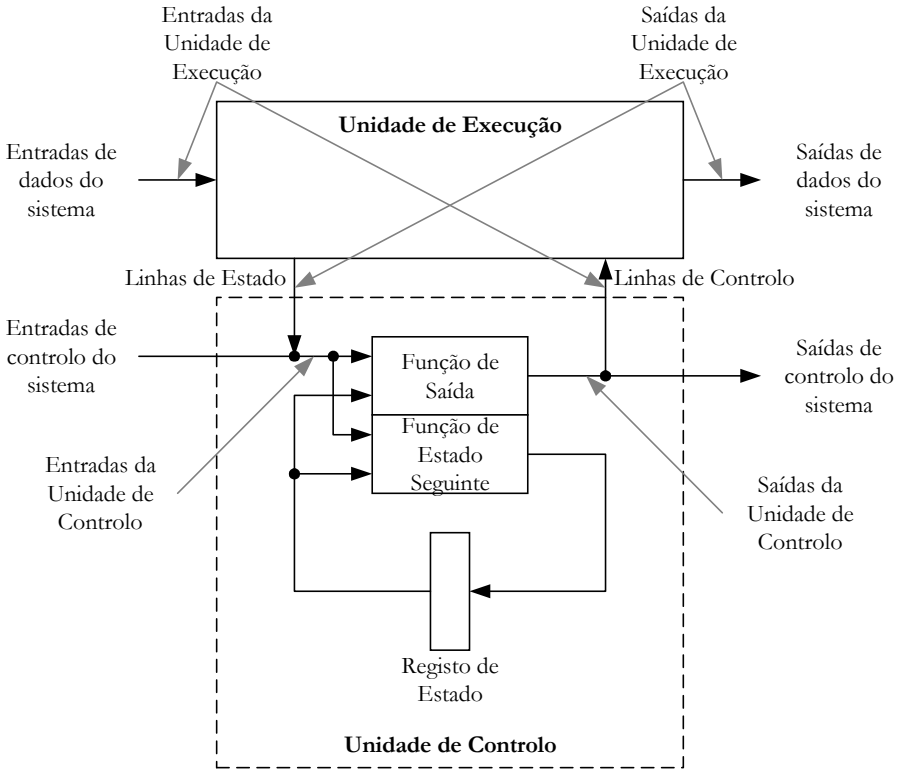


Figura 3.12 – Diagrama de blocos de um controlador com unidade de execução.

À semelhança de um controlador ordinário, um controlador com unidade de execução contém um registo de estado e dois blocos combinatórios que implementam as funções de transição de estado e de saída. Por seu lado, a unidade de execução contém unidades funcionais tais como unidades aritméticas e lógicas, multiplicadores,

elementos de armazenamento, multiplexadores e barramentos utilizados para interligar os diversos componentes.

As entradas da unidade de controlo consistem nos sinais de estado da unidade de execução e em sinais de entrada para controlo do sistema. As saídas da unidade de controlo são constituídas pelos sinais de controlo das unidades funcionais e de armazenamento da unidade de execução e por saídas de controlo do sistema. Todos os sinais anteriores são tipicamente compostos por um só bit. Esta arquitectura também possui normalmente linhas para entrada e saída de dados, as quais são aplicadas directamente à unidade de execução e usadas na ligação a memórias externas e outros componentes do sistema para armazenamento de dados e resultados. Ao contrário dos sinais de controlo anteriores, estes são normalmente multi-bit e organizados em barramentos.

Por ser geral, esta arquitectura é utilizada numa grande diversidade de aplicações desde simples controladores até coprocessadores. Adicionalmente, a arquitectura FSMD é também utilizada como arquitectura base para a construção dos processadores de uso geral, uma vez que cada processador inclui pelo menos uma unidade de controlo e uma de execução para além de outros componentes, tais como memórias *cache* de instruções e dados.

3.3.3 Controlador Hierárquico

Da mesma forma que o modelo HFSM é uma extensão do modelo FSM e que permite a descrição hierárquica do comportamento de uma unidade de controlo, um controlador hierárquico é uma extensão da arquitectura do controlador ordinário (sem hierarquia) e que suporta a implementação hierárquica de uma HFSM.

De notar que uma HFSM pode ser implementada usando a arquitectura de um controlador ordinário através da planificação da sua especificação hierárquica. No entanto, se for adoptado este método, a modularidade do comportamento conseguida durante a especificação é perdida durante a implementação, o que na prática se traduz numa diminuição da flexibilidade e extensibilidade do circuito resultante, as quais constituem dois objectivos importantes deste trabalho. Além disso, esta abordagem também não pode ser aplicada ao projecto de unidades de controlo com cadeias de invocação recursivas entre as sub-máquinas que a constituem.

Nos próximos parágrafos será apresentada a estrutura da arquitectura genérica de um controlador hierárquico bem como outros aspectos relacionados, de forma a tirar partido de todas as potencialidades de uma especificação hierárquica na implementação de uma unidade de controlo e a tornar o circuito projectado flexível e extensível. Consideremos para já a seguinte questão:

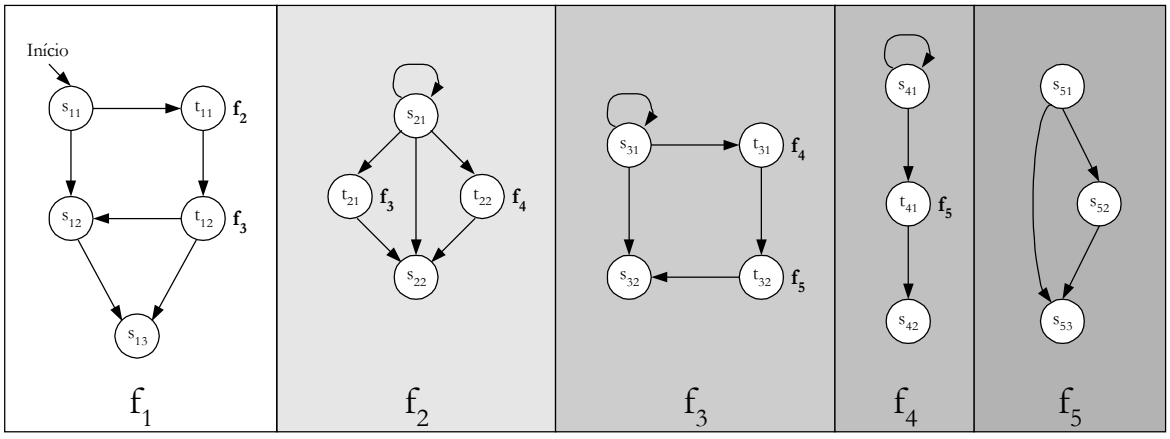
> Como realizar a comutação entre os vários níveis de uma HFSM de forma a executar a FSM que implementa um dado macroestado num novo nível hierárquico e retornar ao nível anterior quando esta terminar a sua execução?

Numa HFSM cada vez que é efectuada a transição para um macroestado deve na realidade ser efectuada a transição para o estado inicial da sub-máquina que o

descreve. A partir deste instante, o fluxo de execução é controlado pela FSM invocada. Quando esta terminar, a execução deve retornar ao estado que sucede ao macroestado da sub-máquina invocadora, pelo que é necessário salvaguardar algures o contexto em que ocorreu a invocação. De notar que este processo de invocação pode ser repetido um número indefinido de vezes sem que nenhuma das sub-máquinas invocadoras tenha retornado, podendo sempre que necessário interromper a sua execução e chamar uma nova FSM.

De facto, este processo é em tudo idêntico à execução de uma aplicação de software num computador de uso geral, pelo que podem ser aproveitados muitos dos conceitos de implementação aí utilizados. Assim, a comunicação entre os diversos níveis hierárquicos e a salvaguarda do contexto das sub-máquinas invocadoras deve ser realizado através de uma pilha de memória. Neste caso e ignorando a comunicação entre os diversos níveis hierárquicos, cada posição da pilha é utilizada como um registo de estado para esse nível, sendo a posição actual indexada por um registo especial de controlo designado por ponteiro da pilha. Esta abordagem foi inicialmente proposta em [Sklyarov84] e explorada posteriormente em [SklRoc96a, SklRoc96b]. Assim, a arquitectura de um controlador hierárquico baseia-se na do controlador ordinário, em que o registo de estado é substituído por uma pilha de registos de estado complementada com alguma lógica adicional para gerir as transições entre diferentes níveis hierárquicos.

Para ilustrar as operações realizadas na pilha de memória e no respectivo ponteiro durante as transições hierárquicas de uma HFSM, consideremos o exemplo da Figura 3.13, o qual não possui nenhuma invocação recursiva.



F = {f_1, f_2, f_3, f_4, f_5}

$S_1 = \{s_{11}, s_{12}, s_{13}\}$
$T_1 = \{t_{11}, t_{12}\}$
$t_{11} = f_2$
$t_{12} = f_3$
$s_{1b} = s_{11}$
$s_{1e} = s_{13}$

$S_2 = \{s_{21}, s_{22}\}$
$T_2 = \{t_{21}, t_{22}\}$
$t_{21} = f_3$
$t_{21} = f_4$
$s_{2b} = s_{21}$
$s_{2e} = s_{22}$

$S_3 = \{s_{31}, s_{32}\}$
$T_3 = \{t_{31}, t_{32}\}$
$t_{31} = f_4$
$t_{32} = f_5$
$s_{3b} = s_{31}$
$s_{3e} = s_{32}$

$S_4 = \{s_{41}, s_{42}\}$
$T_4 = \{t_{41}\}$
$t_{41} = f_5$
$s_{4b} = s_{41}$
$s_{4e} = s_{42}$

$S_5 = \{s_{51}, s_{52}, s_{53}\}$
$T_5 = \varnothing$
$s_{5b} = s_{51}$
$s_{5e} = s_{53}$

Figura 3.13 – Exemplo de uma HFSM sem recursividade.

O topo inicial da pilha é usado como registo de estado da FSM f_1 de nível 1, ou seja, da sub-máquina principal ou de entrada da HFSM. Enquanto não se transitar para

um macroestado é utilizado sempre o mesmo registo e a operação é semelhante à de uma FSM ordinária. Vamos agora supor que é alcançado o macroestado t_{11}. Neste caso deve ser executada a sub-máquina f_2, devendo portanto ser incrementado o ponteiro da pilha. Consequentemente, o topo anterior da pilha armazena o estado interrompido de f_1 (t_{11}) e o novo topo da pilha o estado de entrada de f_2 (s_{21}). A mesma sequência de passos pode ser aplicada a qualquer nível. Quando uma sub-máquina termina a sua execução deve ser realizada a sequência inversa de operações de forma a retornar ao estado interrompido da máquina invocadora, devendo portanto ser decrementado o ponteiro da pilha.

A Figura 3.14 mostra a relação entre os níveis hierárquicos de uma HFSM e os registos da pilha de memória, em que o registo R_i é utilizado como memória do nível L_i da HFSM.

A determinação das sub-máquinas que executam em cada nível hierárquico pode ser obtida com a ajuda do grafo G_h, o qual representa a árvore de invocações da HFSM. Na raiz da árvore encontra-se somente a sub-máquina principal f_1. Os restantes níveis são preenchidos por qualquer sub-máquina f_p do conjunto F que descrevem os macroestados. A relação entre os níveis $L_2 \ldots L_N$ e as sub-máquinas da HFSM é de muitos para muitos, uma vez que a mesma sub-máquina pode ser invocada por várias que executam a diferentes níveis hierárquicos e uma máquina que executa num dado nível pode invocar várias sub-máquinas. As folhas da árvore correspondem às FSMs que não possuem macroestados, ou seja, para as quais $T_p=\varnothing$.

Tal como ilustrado na Figura 3.14, as operações de incremento e decremento do ponteiro da pilha são executadas por intermédio da activação dos sinais y_+ e y_-, respectivamente.

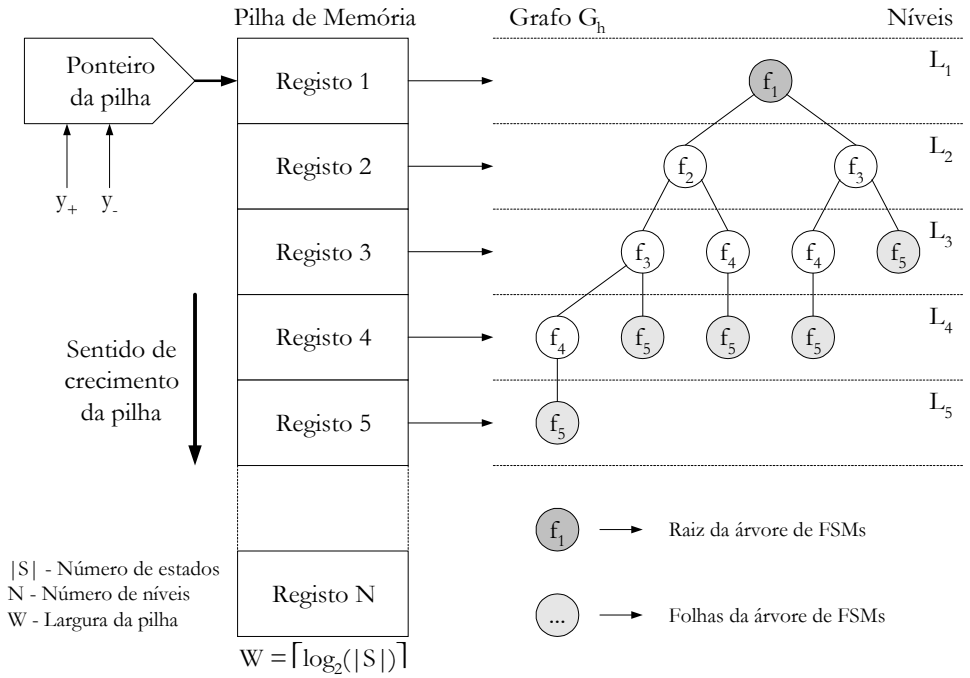


Figura 3.14 – Relações entre as FSMs que executam nos vários níveis hierárquicos de uma HFSM e os registos da pilha de memória.

Uma questão importante no projecto de um controlador hierárquico é a determinação das dimensões mínimas da pilha de memória. Enquanto num controlador ordinário as dimensões do seu elemento de armazenamento se restringem à largura do registo de estado, o qual depende do número de estados e da codificação utilizada, num controlador hierárquico é importante determinar também a profundidade da pilha de memória a fim de garantir que não ocorrem situações de saturação durante a operação da unidade de controlo.

Enquanto numa HFSM não recursiva é possível dar uma resposta exacta a este problema, no caso de uma HFSM com cadeias de invocação recursivas tal não acontece. Quando muito é possível obter estimativas baseadas em informações adicionais sobre as condições típicas em que o sistema vai funcionar. No entanto, para tornar o sistema mais seguro é recomendável em qualquer dos casos a incorporação de mecanismos para detecção de situações de saturação da pilha e que realizem as acções adequadas no caso de tal acontecer.

Apesar da determinação do comprimento da pilha poder ser feita com base no grafo G_h apresentado na Figura 3.14, no caso geral em que o número de níveis e de sub-máquinas pode ser elevado, este tipo de representação é altamente ineficiente. Assim, para determinar o número de níveis de uma HFSM não recursiva vamos construir o grafo dirigido G_i (ver Figura 3.15) que representa as invocações mútuas entre as diferentes FSMs que constituem a HFSM. Ao contrário do grafo multi-nível G_h apresentado na Figura 3.14, o grafo G_i é planar. Os vértices representam cada uma das FSMs f_p do conjunto F. Neste grafo existe um arco de f_p para $f_{p'}$ se f_p invocar $f_{p'}$. A presença de ciclos no grafo está relacionada com a existência de cadeias de invocação recursivas, o que não acontece neste caso. O tamanho total da pilha, isto é, o número de registos, não deve ser inferior ao comprimento do maior caminho que pode ser definido sobre G_i, o qual coincide com o número de níveis de G_h. Este caminho possui neste caso comprimento 5 e está indicado na Figura 3.15 a sombreado.

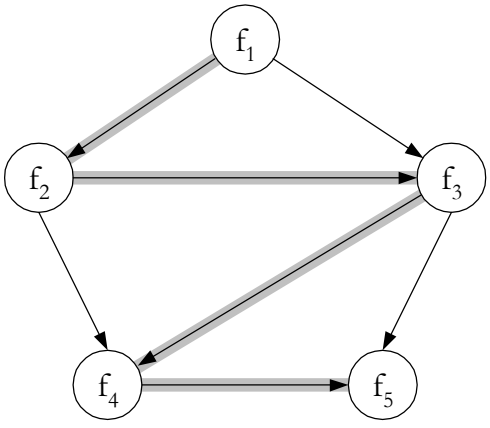
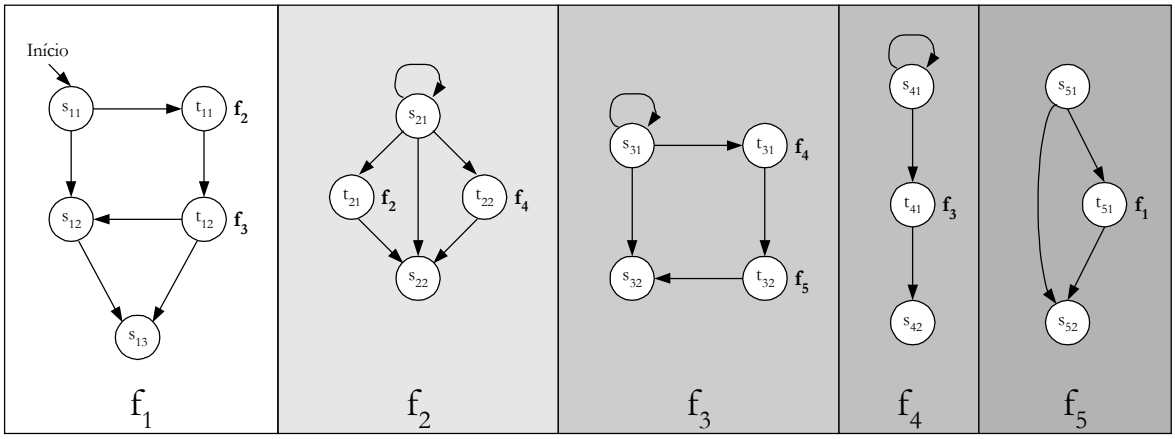


Figura 3.15 – Grafo de invocações numa HFSM sem recursividade.

Consideremos agora um exemplo de uma HFSM recursiva apresentado na Figura 3.16.

Devido à impossibilidade em determinar do ponto de vista estático o número de níveis mínimo que a pilha de memória deve possuir, não é possível neste caso construir o grafo G_h, uma vez que possuiria um número infinito de níveis.



$\mathbf{F = \{f_1, f_2, f_3, f_4, f_5\}}$

$S_1 = \{s_{11}, s_{12}, s_{13}\}$	$S_2 = \{s_{21}, s_{22}\}$	$S_3 = \{s_{31}, s_{32}\}$	$S_4 = \{s_{41}, s_{42}\}$	$S_5 = \{s_{51}, s_{52}\}$
$T_1 = \{t_{11}, t_{12}\}$	$T_2 = \{t_{21}, t_{22}\}$	$T_3 = \{t_{31}, t_{32}\}$	$T_4 = \{t_{41}\}$	$T_5 = \{t_{51}\}$
$t_{11} = f_2$	$t_{21} = f_3$	$t_{31} = f_4$	$t_{41} = f_3$	$t_{51} = f_1$
$t_{12} = f_3$	$t_{21} = f_4$	$t_{32} = f_5$	$s_{4b} = s_{41}$	$s_{5b} = s_{51}$
$s_{1b} = s_{11}$	$s_{2b} = s_{21}$	$s_{3b} = s_{31}$	$s_{4e} = s_{42}$	$s_{5e} = s_{52}$
$s_{1e} = s_{13}$	$s_{2e} = s_{22}$	$s_{3e} = s_{32}$		

Figura 3.16 – Exemplo de uma HFSM com recursividade.

No entanto, é sempre possível construir o grafo G_i, que tal como foi dito atrás representa as invocações mútuas entre as várias sub-máquinas da HFSM. A Figura 3.17 mostra este grafo para o caso da HFSM recursiva da Figura 3.16. Este permite visualizar as cadeias de invocação recursiva e que correspondem aos ciclos fechados que podem ser definidos sobre G_i. Neste caso podem ser identificadas três cadeias de invocações recursivas. O tipo de recursividade depende do número de nodos existentes em cada ciclo, podendo ser identificadas as seguintes situações: auto-recursividade (1 nodo) e recursividade distribuída (mais do que 1 nodo).

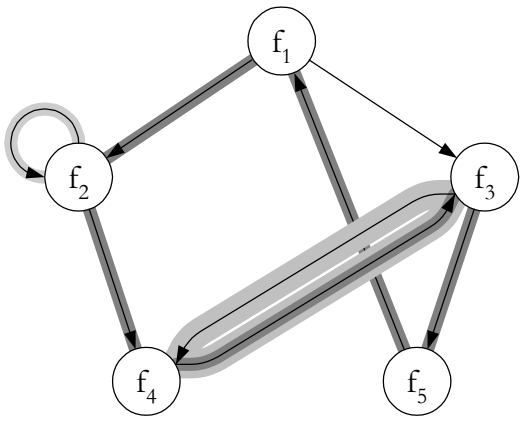


Figura 3.17 – Grafo de invocações numa HFSM com recursividade.

Finalmente, a arquitectura genérica de um controlador hierárquico está apresentada na Figura 3.18 e corresponde a uma extensão do controlador ordinário, em que o registo de estado é substituído por uma pilha de registos de estado e o bloco que implementa a função de transição é dividido em dois blocos, um que implementa as

transições ordinárias (entre estados pertencentes ao mesmo nível hierárquico) e outro que implementa as transições hierárquicas.

Os blocos das funções de transição e de saída podem ser implementados de forma monolítica ou modular. No primeiro caso, durante a síntese do circuito é feita a fusão das funções relativas a todas as sub-máquinas da HFSM, sendo o circuito resultante implementado por apenas um bloco. A principal vantagem desta abordagem é a possibilidade de partilha de recursos pelas funções lógicas que implementam a funcionalidade da unidade de controlo. No entanto, qualquer alteração do comportamento da unidade de controlo obriga à repetição de todo o processo de síntese. Por oposição, na abordagem modular as funções relativas a cada sub-máquina são sintetizadas e implementadas separadamente, resultando num circuito composto por tantos blocos quantas as FSMs do sistema. A vantagem desta técnica é a facilidade com que se pode modificar, adicionar ou remover FSMs sem que para tal seja necessário repetir todo o projecto desde o início. A principal desvantagem deve-se à lógica adicional responsável pela comutação e activação dos blocos em função da FSM que se encontra activa em cada momento.

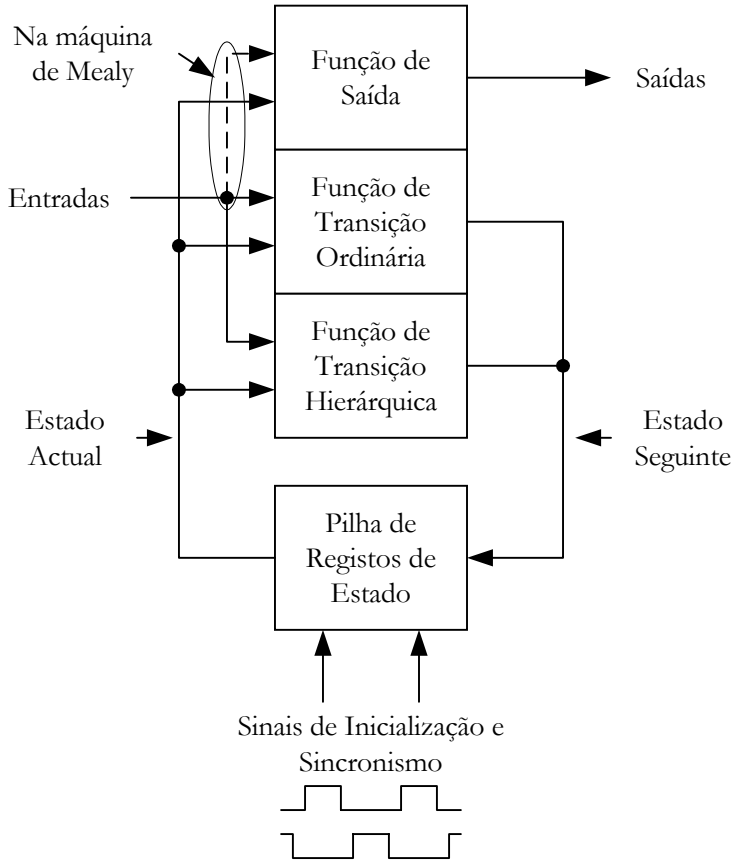


Figura 3.18 – Arquitectura genérica de um controlador hierárquico.

Além dos componentes representados na Figura 3.18 são também necessários registos de entrada e saída para garantir que as entradas são amostradas nos instantes correctos e que as saídas possuem a duração adequada, respectivamente. Este requisito adicional relativamente aos controladores ordinários, deve-se às transições hierárquicas,

as quais são responsáveis pela alteração das condições de entrada do circuito combinatório que calcula os sinais de saída. Este facto faz com que a sincronização de um controlador hierárquico seja mais complexa do que num controlador ordinário. Este assunto será abordado em pormenor mais à frente. Por uma questão de simplicidade, estes detalhes são omitidos na Figura 3.18.

Nas próximas secções vão ser consideradas duas arquitecturas distintas de controladores hierárquicos, uma reprogramável e outra não. Para além destas, em [Rocha99] é também apresentada uma arquitectura reprogramável e modular, a qual não é aqui discutida, pois no capítulo 6 será apresentada uma arquitectura reprogramável, modular e escalável. Além disso está optimizada para implementar unidades de controlo hierárquicas baseadas no modelo VFSM em dispositivos lógicos programáveis, mais concretamente, em FPGAs reconfiguráveis dinamicamente.

Controlador Hierárquico Não Reprogramável

O diagrama de blocos da arquitectura do controlador hierárquico não reprogramável está ilustrado na Figura 3.19. Neste caso, a função de transição ordinária é implementada pelo circuito combinatório, enquanto as transições hierárquicas são desempenhadas pelo circuito combinatório juntamente com o registo R_h e obviamente a pilha de registos.

Esta arquitectura é considerada não reprogramável porque é sintetizada de forma monolítica e toda a funcionalidade está embutida no circuito combinatório. Mesmo que a tecnologia de implementação seja reprogramável, qualquer alteração da lógica combinatória é bastante difícil de realizar, pelo menos sem repetir o processo de síntese e reconstruir todo o circuito. Por outras palavras, a reprogramabilidade de uma unidade de controlo não depende somente da tecnologia de implementação utilizada, mas também da facilidade com que o seu comportamento pode ser alterado. Para tornar esta arquitectura flexível, o circuito combinatório poderia ser construído com uma memória do tipo RAM. No entanto, tal como foi explicado no capítulo 2, geralmente esta não é uma forma eficiente de implementar funções lógicas. De notar que no caso de serem utilizados dispositivos lógicos programáveis que contenham blocos de memória ou tabelas de verdade de dimensões reduzidas para a implementação de funções lógicas, a síntese e implementação não consiste somente na determinação do conteúdo dessas memórias, mas também nas suas interligações. Estas podem afectar a estrutura global de todo o circuito combinatório e consequentemente dificultar a modificação da sua funcionalidade.

O código armazenado no registo R_h indica qual a próxima sub-máquina que deve ser executada. Cada sub-máquina do conjunto $F = \{f_1, f_2, \ldots, f_P\}$ é identificada pelo seu código binário $K(f_p)=(a_{pL} \ldots a_{p2}\ a_{p1})$, onde $a_{pl} \in \{0, 1, -\}$ e $l=1, \ldots, L$. Os elementos "0", "1", "-", representam respectivamente os valores lógicos zero, um, e *don't care*. O comprimento mínimo do código deverá ser $L \geq \lceil \log_2(|F| + 1) \rceil$, uma vez que o código (0 … 00) não é utilizado. A forma mais directa de atribuir os códigos é fazer $K(f_p)= p$.

Se não existirem transições hierárquicas, ou seja, se não existirem transições para macroestados, o circuito opera de forma análoga a um controlador ordinário. Quando for atingido um macroestado é realizada a seguinte sequência de operações:

- O código $K(f_p)$, sendo f_p a sub-máquina que implementa o macroestado, é armazenado no registo R_h através das entradas $z_1, \ldots, z_L$;
- O ponteiro da pilha é incrementado ($y_+=1$). Como resultado é seleccionado um novo registo de estado da pilha de memória. O registo anterior retém o estado da sub-máquina invocadora e que interrompeu a sua execução. O novo registo deve ser automaticamente inicializado a zero (0 … 00);
- O código $K(f_p)$ armazenado em R_h é aplicado às entradas $r_1, \ldots, r_L$ do circuito combinatório e, em conjunto com o código binário zero (0 … 00) aplicado às entradas $q_1, \ldots, q_W$, provoca a transição para o estado inicial de f_p. A partir deste momento, f_p será responsável pelo controlo da execução até terminar;
- Após a terminação de f_p é necessário decrementar o ponteiro da pilha ($y_-=1$) de forma a retornar ao estado interrompido da sub-máquina invocadora. O controlo passará então para o estado no qual f_p foi invocada.

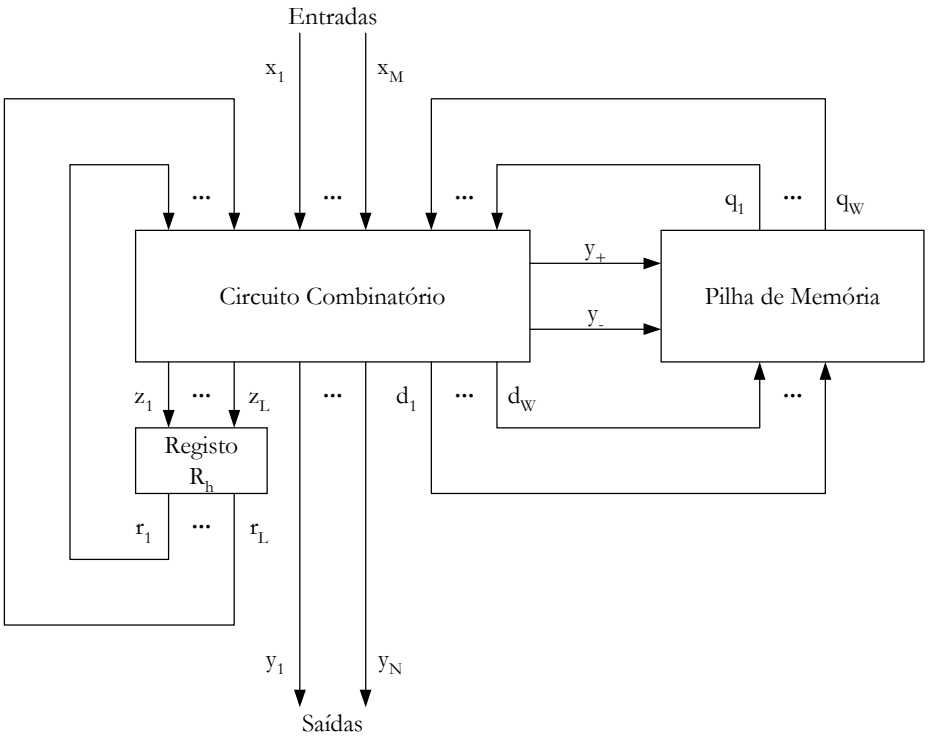


Figura 3.19 – Exemplo de uma arquitectura de um controlador hierárquico não reprogramável.

Controlador Hierárquico Reprogramável

A arquitectura do controlador hierárquico reprogramável é baseada na anterior e foi proposta em [RocSkl97a, RocSkl97b]. O seu diagrama de blocos encontra-se na Figura 3.20. A diferença entre as duas arquitecturas reside na substituição do registo R_h por um conversor de código, o qual actua como uma tabela que transforma o código da próxima sub-máquina a executar no seu estado inicial. Assim, as transições

hierárquicas são geradas pelo conversor de código enquanto as funções de transição ordinária e de saída continuam a ser implementadas pelo circuito combinatório.

Ao contrário da anterior, esta arquitectura é considerada reprogramável porque permite que certas alterações da funcionalidade sejam realizadas sem que para tal seja necessário voltar a projectar de raiz todo circuito. Apesar desta arquitectura ser também monolítica ao nível da implementação das funções de transição e de saída das várias sub-máquinas que a constituem, a sua funcionalidade está agora distribuída pelo circuito combinatório e pelo conversor de código. A dificuldade em modificar o primeiro é análoga ao caso anterior. No entanto, como o conversor de código é uma tabela e possui em princípio uma estrutura do tipo memória, é facilmente modificável, o que permite a realização de pequenas alterações ao nível das transições hierárquicas.

O funcionamento desta arquitectura pode ser descrito da seguinte forma:

- Em todas as transições para estados atómicos as saídas r_1, …, r_w do conversor de código são colocadas a 0 através da aplicação do código 0 às suas entradas z_1, …, z_L. O estado seguinte é fornecido pelas saídas c_1, …, c_w do circuito combinatório tal como num controlador ordinário (não hierárquico);
- No caso de uma transição para um macroestado descrito pela sub-máquina f_p, é realizada a seguinte sequência de operações:
  - O código $K(f_p)$ da sub-máquina f_p é aplicado às entradas z_1, …, z_L do conversor de código;
  - O ponteiro da pilha é incrementado ($y_+=1$), pelo que o topo da pilha é alterado e consequentemente passará a ser utilizado um novo registo de estado, ficando o código do estado da sub-máquina interrompida no topo anterior da pilha;
  - As saídas c_1, …, c_w do circuito combinatório são colocadas a 0. O código $K(f_p)$ é transformado pelo conversor de código no código do primeiro estado do macroestado descrito por f_p. A partir deste instante, a nova sub-máquina é responsável pelo controlo da execução até terminar, ou seja, até ser atingido o seu estado final;
  - Após a terminação da sub-máquina f_p que implementa o macroestado é necessário decrementar o ponteiro da pilha ($y_-=1$) de forma a retornar à sub-máquina invocadora. O controlo é então retomado pela sub-máquina interrompida no respectivo macroestado, transitando de imediato para o seu estado seguinte.

Esta arquitectura possui relativamente à anterior as seguintes vantagens:

- Uma vez que não existem linhas de entrada do circuito combinatório dedicadas à identificação do macroestado, o número total de ligações é inferior;
- O mapeamento dos códigos dos macroestados nos códigos dos estados de entrada das sub-máquinas que os implementam é realizado pelo conversor de código, o que faz com que seja possível realizar alterações nas transições hierárquicas sem que seja necessário alterar o circuito combinatório;

- Se o conversor de código for implementado usando componentes reprogramáveis baseados em memória RAM, as alterações podem ser realizadas facilmente e em certos casos até dinamicamente. De facto, esta arquitectura constitui uma primeira abordagem para a construção de unidades de controlo flexíveis. No entanto, a extensibilidade ainda não é alcançável com esta arquitectura uma vez que obriga a reprojectar todo o circuito combinatório.

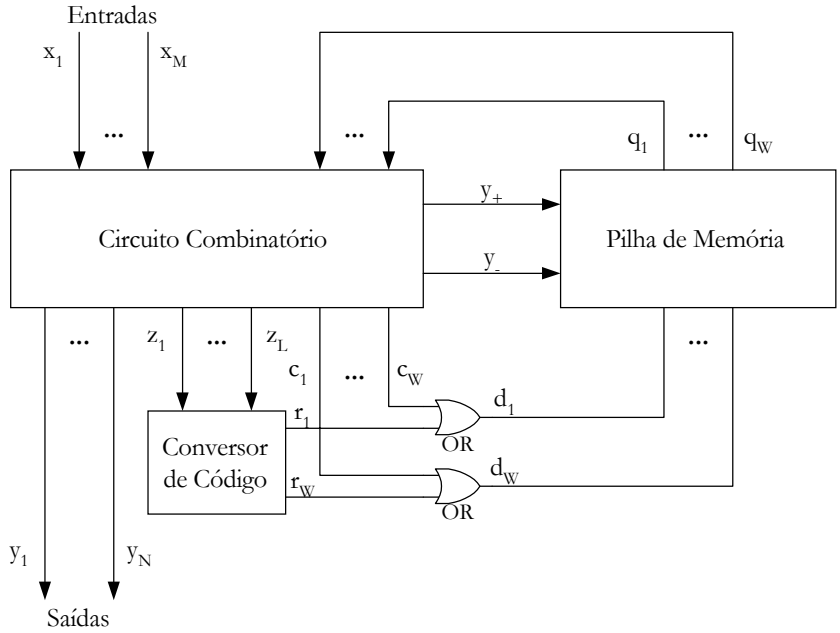


Figura 3.20 – Exemplo de uma arquitectura de um controlador hierárquico reprogramável.

Sincronização

Enquanto um controlador ordinário possui uma sincronização bastante simples, baseada somente num sinal de relógio responsável pelas transições de estado, um controlador hierárquico possui uma sincronização mais complexa, sendo necessário sincronizar os seguintes eventos:

- Transições de estados;
- Fixação dos sinais de entrada x_1, …, x_M;
- Fixação dos sinais de saída y_1, …, y_N;
- Fixação das entradas do conversor de código z_1, …, z_L;
- Incremento e decremento do ponteiro da pilha (sinais y_+, y_-).

Existem várias formas de realizar a sincronização, as quais dependem do modelo e da arquitectura utilizada. Aqui vão ser consideradas apenas duas relativas à arquitectura reprogramável apresentada acima, uma para o modelo de Moore e a outra para modelo de Mealy.

Sincronização de um Controlador Hierárquico de Moore

A sequência de sincronização de um controlador hierárquico reprogramável de Moore, apresentada em [Rocha99], é baseada no método de sincronização da arquitectura do controlador hierárquico não reprogramável descrito em [Sklyarov84], e está ilustrada na Figura 3.21. Na realidade, esta figura apresenta duas formas diferentes de sincronização, mas que produzem resultados equivalentes. A primeira é baseada em dois sinais de relógio com fases distintas, enquanto a segunda possui apenas um sinal de relógio complementado por impulsos de pequena duração e gerados de forma assíncrona.

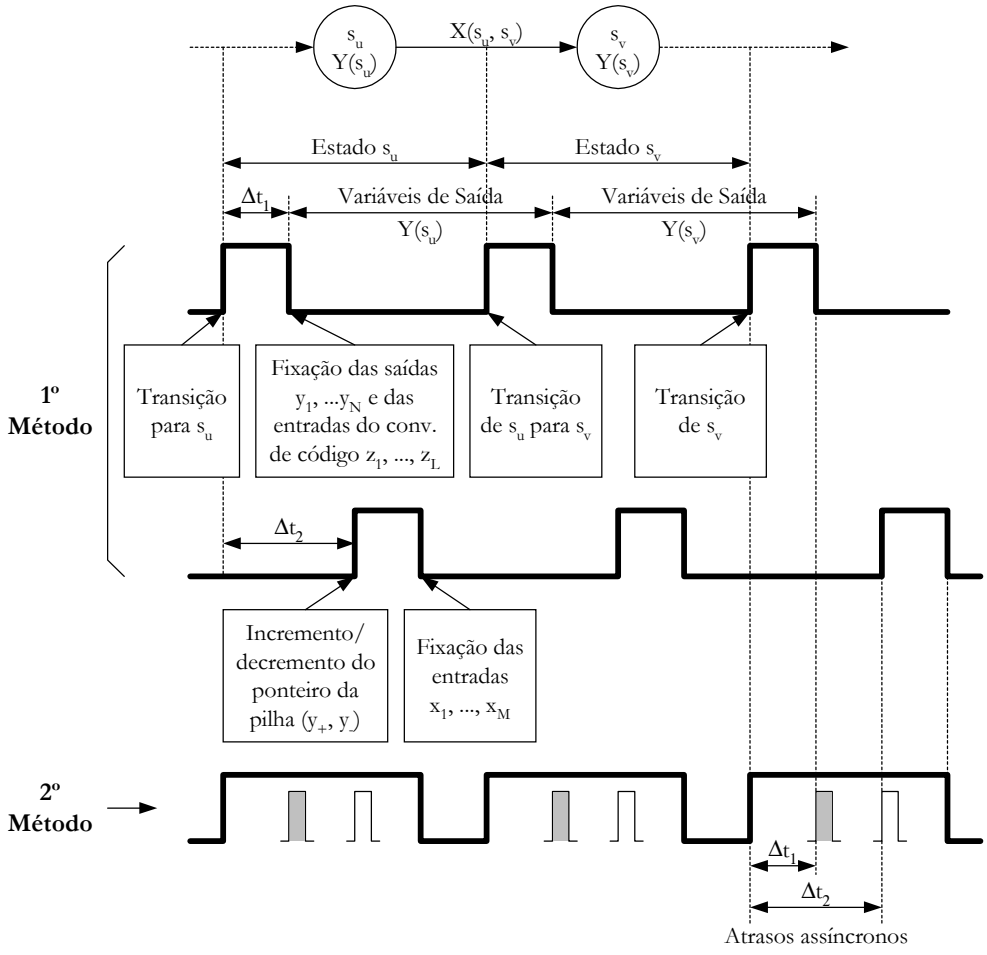


Figura 3.21 – Sequência de eventos de sincronização na arquitectura do controlador hierárquico reprogramável segundo modelo de Moore.

A transição entre estados é sincronizada pelo flanco ascendente do primeiro sinal de relógio da Figura 3.21, o qual é responsável pelo carregamento do novo estado na pilha de memória. Este procedimento não depende de s_u e s_v pertencerem ao mesmo ou a diferentes níveis hierárquicos. Após um tempo de atraso, o novo estado surgirá nas saídas da pilha, sendo então aplicado ao circuito combinatório. Este gera os sinais de saída y_1, …, y_N, e c_1, …, c_W, as entradas z_1, …, z_L do conversor de código e os sinais de controlo do ponteiro da pilha y_+ e y_-.

O flanco descendente do mesmo sinal de relógio é usado para fixar as variáveis de saída y_1, …, y_N e as entradas z_1, …, z_L do conversor de código. O intervalo de tempo Δt_1 deve ser superior à soma de todos os atrasos envolvidos na geração dos sinais acima mencionados de forma a garantir que são fixados os valores correctos.

A transição ascendente do segundo sinal de relógio é responsável pelo controlo do ponteiro da pilha de memória. Se um dos sinais y_+ ou y_- estiver activo, o ponteiro é incrementado ou decrementado, caso contrário permanece inalterado.

No flanco descendente do mesmo sinal são fixadas as entradas x_1, …, x_M que serão utilizadas pelo circuito combinatório para calcular o novo estado a carregar na pilha de memória no início do próximo ciclo de sincronização.

Esta sequência de passos de sincronização repete-se indefinidamente durante o funcionamento do controlador.

Na Figura 3.21 está também indicada a correspondência entre os eventos dos dois métodos de sincronização apresentados, o baseado em dois sinais de relógio desfasados e o baseado somente num sinal de relógio complementado com atrasos assíncronos.

Sincronização de um Controlador Hierárquico de Mealy

A sincronização do controlador hierárquico reprogramável de Mealy está ilustrada na Figura 3.22. Comparativamente ao caso anterior, a principal diferença reside na sequência pela qual devem ocorrer os eventos de sincronização, uma vez que neste caso as saídas não dependem só do estado actual mas também das entradas, pelo que após a transição de estado, a fixação das entradas deve ser feita antes das saídas.

A transição entre estados é sincronizada pelo flanco ascendente do primeiro sinal de relógio da Figura 3.22. O flanco descendente do mesmo sinal é agora utilizado para fixar os valores das entradas x_1, …, x_M. Após algum tempo, o circuito combinatório produz as saídas relativas ao estado actual e aos valores das entradas fixadas.

O flanco ascendente do segundo sinal de sincronização é usado para fixar os sinais de saída. O intervalo de tempo Δt_2-Δt_1 deve ser superior à soma dos atrasos envolvidos na geração dos sinais de forma a garantir que são fixados os valores correctos.

Por último, a transição descendente do segundo sinal de sincronização é responsável pelo controlo do ponteiro da pilha de memória de acordo com o valor dos sinais y_+ e y_-.

Esta sequência de passos de sincronização repete-se indefinidamente durante o funcionamento do controlador.

Tal como no caso anterior, a Figura 3.22 indica a correspondência entre os eventos dos dois métodos de sincronização apresentados, o baseado em dois sinais de relógio desfasados e o baseado somente num sinal de relógio complementado com atrasos assíncronos.

Neste caso, o método de sincronização baseado em atrasos assíncronos tem a vantagem da pilha de memória precisar apenas de um sinal de sincronização mas que é usado em ambos os flancos; no ascendente para efectuar a transição de estado e no

descendente para incrementar ou decrementar o ponteiro da pilha consoante o estado dos sinais y_+ e y_-, respectivamente.

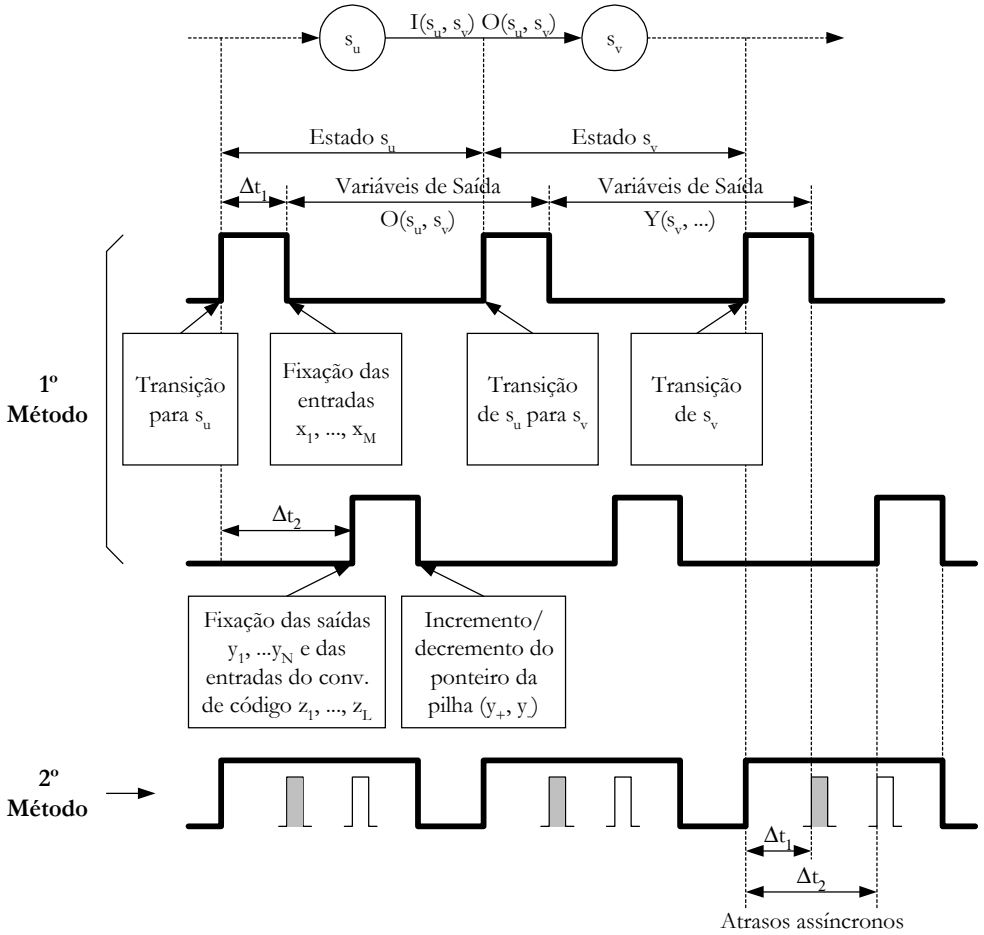


Figura 3.22 – Sequência de eventos de sincronização na arquitectura de controlador hierárquico reprogramável segundo modelo de Mealy.

3.3.4 Controlador Virtual

O controlador virtual é a arquitectura que implementa o modelo VFSM apresentado na secção 3.2.7. Existem várias variantes possíveis desta arquitectura, das quais será apresentada apenas a que serviu de base à implementação realizada no âmbito deste trabalho. Neste capítulo vai ser realizada uma discussão bastante resumida e simplificada da mesma, ficando para o capítulo 6 a sua apresentação completa.

Tal como foi referido atrás, por uma questão de eficiência, simplicidade de reconfiguração e previsibilidade de comportamento, os modelos VFSM devem ser preferencialmente baseados em modelos hierárquicos, pelo que a arquitectura aqui apresentada é uma extensão da arquitectura geral do controlador hierárquico atrás descrito. De facto, esta arquitectura resulta da modularização do controlador hierárquico com algumas características adicionais.

A sua estrutura de alto nível é apresentada na Figura 3.23. O aspecto mais importante é a partição do circuito em blocos mais simples e mutuamente exclusivos, chamados Elementos Reprogramáveis (ERs), que implementam as funções de saída e de transição de cada sub-máquina de forma disjunta. Para facilitar a reprogramação e a interacção entre os vários ERs, o seu interface externo está predefinido, ou seja, todas as suas ligações externas são fixas. A configuração interna de cada elemento reprogramável pode ser modificada de forma a implementar a funcionalidade pretendida.

À semelhança do controlador hierárquico, o controlador virtual possui também uma pilha de registos que actua como memória do circuito.

Apesar da estrutura, em particular das ligações, parecer bastante mais complexa que nos casos anteriores, a sua regularidade faz com que seja facilmente implementável. Além disso, dependendo da tecnologia de implementação utilizada, a configuração de cada ER pode ser alterada individualmente e dinamicamente, o que possibilita a reutilização dos mesmos recursos de hardware para implementar diferentes sub-máquinas. A substituição do conteúdo ou reconfiguração dos ERs deve ser feita de acordo com uma estratégia de escalonamento que permita reduzir os tempos de espera do circuito para reconfiguração.

Um pormenor importante desta arquitectura está relacionado com a distribuição das linhas de entrada e com o controlo das linhas de saída dos vários ERs. Idealmente, todos os ERs deviam poder ler todas as entradas e controlar todas as saídas. No entanto, no caso de um número elevado destas linhas é necessário utilizar multiplexadores ou outros elementos de comutação de forma a restringir o número de linhas aplicadas a cada ER.

A arquitectura é escalável, permitindo a existência de um número elevado de ERs, limitado apenas pela área do circuito e pelos atrasos dos elementos de comutação utilizados na distribuição das entradas e no controlo das saídas. No entanto, devido ao carácter virtual do circuito não são normalmente necessários muitos blocos de implementação.

Uma vez que as FSMs implementadas em cada bloco podem ser substituídas estática ou dinamicamente é fundamental identificar as que estão carregadas em cada ER num dado momento. Isto obriga à existência de dispositivos suplementares para monitorização e controlo bem como a dispositivos externos para armazenamento das configurações de todas as FSMs. Caso a FSM necessária num dado momento não esteja configurada num dos ERs, é necessário efectuar o seu carregamento a partir do dispositivo externo de armazenamento.

Devido à necessidade de reprogramação dos blocos de implementação das FSMs, a construção dos controladores virtuais deve ser realizada em dispositivos lógicos programáveis pelo utilizador (*Field Programmable Logic Devices – FPLDs*). Consoante se pretenda que a reconfiguração do circuito seja feita de forma estática ou dinâmica, a implementação deve ser feita em dispositivos reprogramáveis estática ou dinamicamente, respectivamente.

Devido aos procedimentos de verificação das FSMs implementadas em cada bloco, a sequência de controlo e de sincronização é mais complexa do que no caso dos controladores hierárquicos. Mais à frente será apresentada de forma detalhada a

sincronização e os circuitos de controlo de um controlador virtual. Como iremos ver, torna-se necessário realizar algumas optimizações de forma a minimizar a sobrecarga associada às transições hierárquicas, favorecendo o caso mais frequente, que são as transições não hierárquicas.

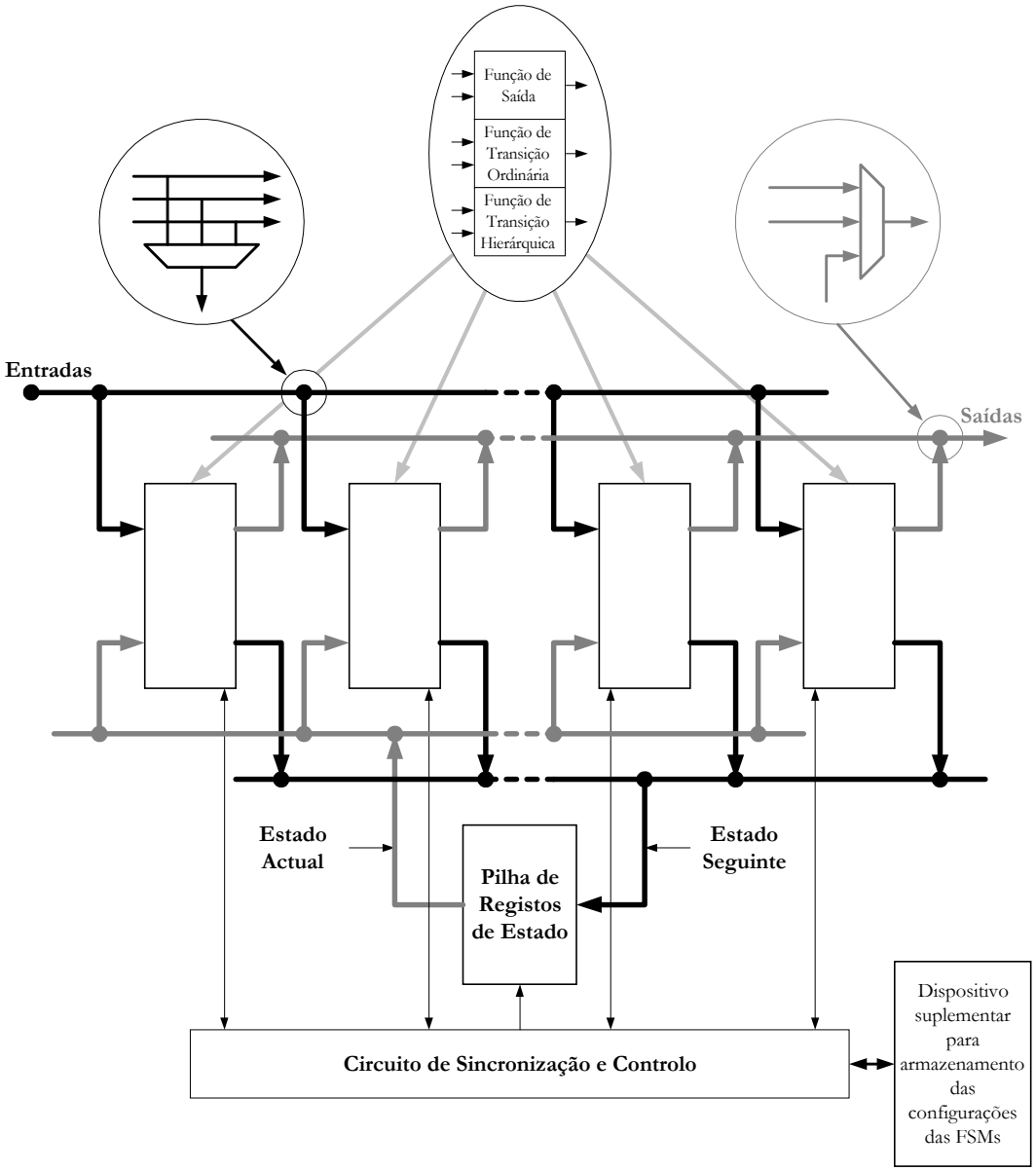


Figura 3.23 – Arquitectura genérica de um controlador virtual.

Uma vez que cada sub-máquina é uma entidade relativamente autónoma, pode ser sintetizada e implementada separadamente, o que possui duas vantagens importantes:

- O desenvolvimento e teste individual de cada sub-máquina e a possibilidade de a reutilizar noutros projectos que utilizem a mesma arquitectura base;
- A redução da complexidade associada às tarefas de síntese, optimização e implementação devido à divisão da unidade de controlo em entidades mais simples que são manipuladas separadamente. Mais concretamente, é possível

adoptar técnicas de codificação de estados e minimização lógica locais a cada sub-máquina.

Apesar do objectivo desta arquitectura ser o suporte da implementação de um conjunto de FSMs que executem sequencialmente, pode também suportar a implementação de algoritmos de controlo com concorrência desde que seja possível activar vários blocos em simultâneo. De facto as arquitecturas que possibilitam a implementação de unidades de controlo baseadas nos modelos paralelos atrás descritos, nomeadamente o PFSM e o HaPFSM, não foram abordadas intencionalmente por vários motivos:

- As arquitecturas propostas em [Sklyarov87, RocSklFer97] possuem algumas limitações. Mais concretamente, são consideradas em [Rocha99] pseudo-paralelas, uma vez que o ciclo principal de execução da máquina é dividido em sub-ciclos nos quais é executada cada uma das máquinas. Além disso, o circuito combinatório é partilhado pelas várias sub-máquinas o que faz com que estas não executem realmente em paralelo e dificulte a sua virtualização;
- Com base nos modelos PFSM e HaPFSM atrás definidos, em particular nas variantes identificadas, nomeadamente a GFSM, a HPFSM e a PHFSM é necessário mais algum trabalho de investigação para desenvolver as arquitecturas mais apropriadas para implementar cada um dos modelos;
- Finalmente, os objectivos deste trabalho estão mais direccionados para a implementação de controladores hierárquicos, pelo que as questões relacionadas com o paralelismo e a concorrência foram abordadas somente do ponto de vista teórico.

No último capítulo desta dissertação são apresentadas algumas ideias sobre trabalho futuro onde um dos aspectos principais é a investigação de arquitecturas que permitam a implementação eficiente de controladores paralelos baseados nos modelos GFSM, HPFSM e PHFSM e que possibilitem a sua virtualização.

4 Especificação de Unidades de Controlo

Sumário

Neste capítulo são apresentados alguns formalismos ou linguagens para especificação de unidades de controlo. De notar que a barreira que separa os modelos das linguagens de especificação é em muitos casos muito ténue. De facto, são raras as situações em que as características de uma linguagem de especificação permitem descrever exactamente todas as potencialidades de um dado modelo e vice-versa. Na maior parte dos casos as características de uma linguagem são um subconjunto ou superconjunto das necessárias para tirar partido de todas as potencialidades do modelo utilizado, pelo que as linguagens são em muitos casos também consideradas modelos.

Este capítulo começa por fazer o levantamento dos requisitos das linguagens de especificação de forma a poderem descrever de forma eficiente qualquer tipo de unidade de controlo baseada nos modelos orientados ao estado apresentados no capítulo anterior. Em particular são identificados os seguintes requisitos: transições de estados, concorrência, hierarquia, comunicação, sincronização e processamento de excepções.

Em termos de formalismos, são apresentados vários exemplos de linguagens gráficas, mais concretamente, os diagramas de transição de estado, as máquinas de estado algorítmicas, as redes de Petri (em particular uma variante designada por máquinas de estado), os esquemas de grafos, os esquemas de grafos hierárquicos e paralelos e finalmente, os *Statecharts*. Em termos de formalismos textuais são apresentadas as tabelas de transição de estado como um método equivalente aos diagramas de transição de estado e a linguagem de descrição de hardware VHDL.

A apresentação das diversas linguagens é acompanhada de um exemplo comum de uma máquina de venda automática de forma a se poder comparar os resultados obtidos. Este capítulo termina com a apresentação do processo de simulação em VHDL da máquina de venda automática e do seu módulo que contém os vectores de teste e que é normalmente designado por *testbench*.

4.1 Introdução

Tal como foi referido no capítulo anterior, um modelo pode ser capturado por várias linguagens e uma linguagem pode capturar diversos modelos. Neste capítulo vão ser apresentadas algumas das linguagens ou formalismos que podem ser usados para realizar a especificação de unidades de controlo baseadas nos modelos orientados ao estado discutidos no capítulo anterior. Como se verá ao longo deste capítulo, os formalismos existentes permitem descrever o comportamento de uma unidade de controlo de forma gráfica ou textual.

O requisito mínimo que as linguagens devem cumprir é serem capazes de representar de forma eficiente os estados, as transições e as acções a ambos associadas, os quais são elementos sempre presentes em qualquer modelo de unidade de controlo. Dependendo da complexidade e características do sistema a projectar e consequentemente do modelo adoptado, pode ser desejável que a linguagem utilizada suporte também a representação explícita de:

- Concorrência – em muitos casos a representação do comportamento de um sistema usando unicamente subcomportamentos sequenciais pode resultar em descrições complexas e pouco naturais e consequentemente difíceis de compreender. Nestas situações é mais fácil e razoável utilizar subcomportamentos concorrentes que colaboram e interagem entre si de forma a atingir a funcionalidade desejada.
- Hierarquia – o princípio "dividir para reinar" é a forma básica de lidar com a complexidade. A especificação hierárquica de um sistema possibilita a sua descrição como um conjunto de sub-sistemas mais pequenos e permite ao projectista concentrar-se num sub-sistema de cada vez. Existem dois tipos de hierarquia, a estrutural e a comportamental [GajVahNarGon94], que estão relacionadas com a zona de trabalho no plano de abstracção (ver capítulo 1). Enquanto a primeira está normalmente mais associada à montagem ascendente dos componentes do sistema, a segunda é geralmente mais utilizada na decomposição descendente da sua funcionalidade, embora nada obrigue que assim seja.
- Comunicação – se o comportamento de um sistema for descrito como um conjunto de subcomportamentos concorrentes ou processos, é necessário que estes comuniquem entre si de forma a alcançar a funcionalidade pretendida. Este tipo de comunicação é normalmente conceptualizada em termos dos paradigmas de memória partilhada ou de passagem de mensagens [GajVahNarGon94]. No paradigma de memória partilhada cada processo emissor escreve num meio partilhado, tal como uma variável global ou porto que pode ser lido por todos os processos receptores. No paradigma de passagem de mensagens, os dados são transferidos entre processos através de um meio abstracto chamado canal, por intermédio de primitivas do tipo enviar-receber.
- Sincronização – quando o comportamento de um sistema é descrito como um conjunto de subcomportamentos concorrentes ou processos, cada um

pode gerar dados ou eventos que necessitem de ser reconhecidos por outros processos. Neste caso os dados trocados entre processos ou as acções realizadas pelos vários processos podem necessitar de ser sincronizadas [GajVahNarGon94].

- Processamento de excepções – nalguns casos, a ocorrência de determinados eventos, tal como a activação de um sinal de reinicialização, obriga a que um comportamento activo seja terminado imediatamente e que seja executado um comportamento predefinido em vez de se esperar até que o primeiro conclua a sua execução.

De notar que nem todas as facilidades anteriores são suportadas pelas diversas linguagens.

Nas próximas secções vão ser descritas sumariamente as linguagens mais utilizadas na especificação de unidades de controlo. A sua apresentação vai ser feita usando um exemplo comum que ilustra algumas das características mais importantes de cada uma. Desta forma é possível comparar os vários formalismos de especificação em termos da sua eficiência e facilidade de utilização.

O exemplo considerado consiste numa máquina de venda automática que fornece latas de bebidas após ter recebido a quantia de 150 Escudos (Esc). A máquina aceita moedas de 50 Esc e 100 Esc, devendo ser introduzidas uma de cada vez. Se o comprador introduzir três moedas de 50 Esc, ou uma de 50 Esc e uma de 100 Esc recebe uma lata, mas se colocar duas moedas de 100 Esc recebe uma lata e 50 Esc de troco.

Após a colocação das moedas suficientes para comprar uma lata, deve ser premido o botão "Continuar" para concluir a compra. Existe também um botão denominado "Cancelar" que interrompe a operação e devolve ao cliente todas as moedas introduzidas até esse momento.

Para simplificar o problema vamos considerar que a máquina só vende um tipo de latas e que só pode ser adquirida uma de cada vez. Além disso, quando já tiver sido introduzido dinheiro suficiente, todas as moedas suplementares inseridas são automaticamente devolvidas ao cliente. Este só pode colocar mais moedas e efectuar uma nova compra quando tiver recebido a lata anterior e eventualmente o troco.

Independentemente do formalismo utilizado, a unidade de controlo possui um conjunto de estados que simbolizam a situação em que se encontra uma operação de venda, um conjunto de entradas que representam o valor lido dos sensores e botões do sistema e um conjunto de saídas que dependem do estado interno da unidade de controlo e são utilizadas para controlar os actuadores da máquina. Mais detalhadamente, cada um destes conjuntos possui os seguintes elementos cuja descrição é a seguinte:

- Estados
  - EInicial, E50, E100, E150, E200 – activados consoante o montante correspondente às moedas introduzidas até um dado momento;
  - ELata – activado após terem sido introduzidos 150 Esc e o botão "Continuar" tiver sido premido;

  - ELataTroco – activado após terem sido introduzidos 200 Esc e o botão "Continuar" tiver sido premido;
  - ECancelada – activado quando for premido o botão "Cancelar" durante uma operação de compra.
- Entradas
  - M50 – linha do sensor que detecta a introdução de moedas de 50 Esc;
  - M100 – linha do sensor que detecta a introdução de moedas de 100 Esc;
  - Continuar – linha do botão que simboliza a confirmação da compra após terem sido introduzidas as moedas necessárias;
  - Cancelar – linha do botão que simboliza o cancelamento de uma compra.
- Saídas
  - Lata – linha de activação do actuador que controla a libertação de uma lata;
  - Troco – linha de activação do actuador que controla a libertação de uma moeda de 50 Esc de troco;
  - Devolução – linha de activação do actuador que controla, no caso de cancelamento, a devolução das moedas introduzidas até ao momento;
  - Rejeição – linha de activação do actuador que controla o mecanismo de rejeição de moedas após terem sido introduzidas as suficientes para comprar uma lata.

Nas próximas secções vão ser apresentados vários formalismos e analisada a forma como estes capturam o comportamento da máquina de venda automática. Em particular são apresentados os diagramas de transição de estado, as tabelas de transição de estado, as máquinas de estado algorítmicas, uma variante das redes de Petri chamada máquinas de estado, os esquemas de grafos, os esquemas de grafos hierárquicos, os *Statecharts* e finalmente a linguagem de descrição de hardware VHDL.

4.2 Diagramas de Transição de Estado

O modelo de Máquina de Estados Finitos (*Finite State Machine – FSM*) descrito no capítulo anterior e definido pelo sêxtuplo $<S, I, O, g, h, s_0>$ é tradicionalmente capturado por um formalismo gráfico vulgarmente designado por Diagrama de Transição de Estado (*State Transition Diagram – STD*). Um STD é um multi-grafo dirigido e interpretado $G_t(V, A)$ [MelSarTysYemZve98], onde os elementos do conjunto dos vértices V possuem uma correspondência de um para um com os elementos do conjunto dos estados S. As transições de estado são representadas pelos elementos do conjunto dos arcos A, os quais possuem também uma correspondência de um para um com as transições especificadas pela função g. Em particular, existe um arco (v_m, v_n), em que v_m e v_n representam respectivamente os estados s_m, e s_n, se existir um vector de entrada $i_j \in I$ tal que $g(s_m, i_j) = s_n$, $\forall$ m, n = 1, 2, …, $|S|$.

No modelo de Mealy um arco com origem em $v_m \in V$ é etiquetado com $i_j/h(s_m, i_j)$, enquanto no modelo de Moore os arcos são etiquetados somente com o

vector de entrada i_j, sendo cada vértice v_m etiquetado com a função de saída correspondente $h(s_m)$. Em ambos os casos cada vértice é também normalmente etiquetado com um símbolo binário ou alfanumérico que representa o estado.

Na Figura 4.1 encontra-se o STD relativo ao exemplo da máquina de venda automática descrita na introdução deste capítulo. De notar que foi utilizado o modelo de Moore, uma vez que as saídas estão associadas aos estados.

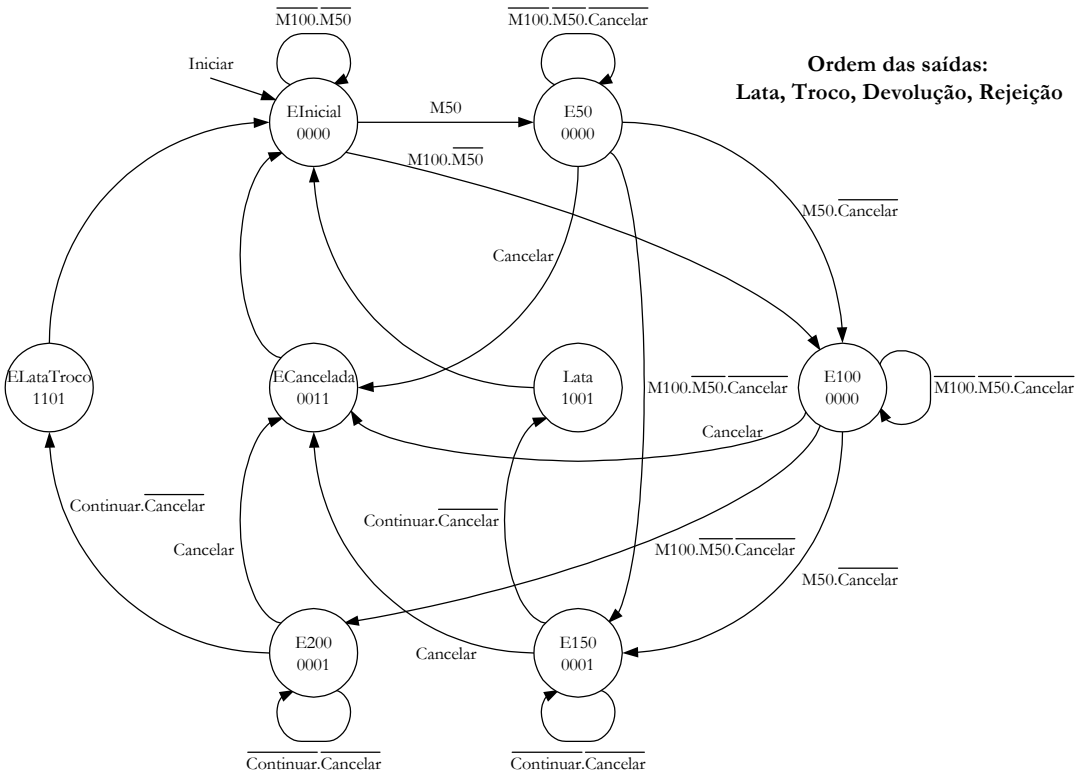


Figura 4.1 – Exemplo de um diagrama de transição de estado para descrição do comportamento da unidade de controlo da máquina de venda automática.

Para não sobrecarregar um STD, em muitos casos opta-se por não colocar os lacetes existentes na Figura 4.1, assumido-se que todas as condições não cobertas pelas transições a partir de um dado estado correspondem à permanência nesse estado.

Por vezes é conveniente representar os STDs de forma hierárquica através da sua partição em subdiagramas [Micheli94]. À excepção do subdiagrama da raiz, todos os outros possuem um estado de entrada e um estado de saída que são associados a um ou mais vértices de chamada em níveis hierarquicamente superiores e que representam macroestados. Cada transição para um macroestado é equivalente a uma transição para o estado de entrada do STD correspondente. Uma transição para o estado de saída corresponde a um retorno ao macroestado invocador. De facto, esta representação já foi utilizada na discussão das máquinas hierárquicas do capítulo anterior. Os diagramas hierárquicos são utilizados para realizar a composição ascendente de uma FSM de forma modular ou para efectuar a decomposição descendente de uma especificação, sendo particularmente úteis na elaboração e manipulação de especificações complexas.

Alternativamente, uma FSM pode também ser capturada textualmente por uma Tabela de Transição de Estado (*State Transition Table – STT*). Uma STT possui

normalmente quatro colunas designadas por "estado actual", "saídas", "entradas" e "estado seguinte". Na Tabela 4.1 encontra-se a STT correspondente ao exemplo da máquina de venda automática descrita pelo STD da Figura 4.1. Como se trata de uma máquina de Moore, cada estado só possui uma combinação ou vector de saída associado. O número de linhas por estado das colunas das "entradas" e do "estado seguinte" corresponde ao número de arcos que partem desse estado, incluindo o lacete. Na máquina de Mealy é usual trocar a ordem das colunas das "entradas" e das "saídas" para melhor expressar a dependência definida pelo mapeamento $h : S \times I \rightarrow O$. Na coluna das saídas os valores "1" e "0" significam respectivamente que para esse estado uma dada saída é ou não activada. Na coluna das entradas os símbolos "1" e "0" representam a dependência de uma transição da variável na forma normal ou complementada, respectivamente. O símbolo "-" significa que a transição não depende dessa variável.

Os STDs não são apropriados para especificar sistemas e mais concretamente unidades de controlo complexas porque não suportam a representação explícita de concorrência. Apesar de suportarem descrições hierárquicas, como veremos mais à frente, existem outros formalismos que o permitem fazer de forma mais eficaz.

Estado Actual	**Saídas** Lata Troco Devolução Rejeição	**Entradas** M50 M100 Continuar Cancelar	**Estado Seguinte**
EInicial	0 0 0 0	1 - - -	E50
		0 1 - -	E100
		0 0 - -	EInicial
E50	0 0 0 0	- - - 1	ECancelada
		1 - - 0	E100
		0 1 - 0	E150
		0 0 - 0	E50
E100	0 0 0 0	- - - 1	ECancelada
		1 - - 0	E150
		0 1 - 0	E200
		0 0 - 0	E100
E150	0 0 0 1	- - - 1	ECancelada
		- - 1 0	ELata
		- - 0 0	E150
E200	0 0 0 1	- - - 1	ECancelada
		- - 1 0	ELataTroco
		- - 0 0	E200
ELata	1 0 0 1	- - - -	EInicial
ELataTroco	1 1 0 1	- - - -	EInicial
ECancelada	0 0 1 1	- - - -	EInicial

Tabela 4.1 – Tabela de transição de estado que descreve o comportamento da unidade de controlo da máquina de venda automática.

4.3 Máquinas de Estado Algorítmicas

Os diagramas de transição de estados não são apropriados para descrever algoritmicamente o comportamento de uma unidade de controlo, nem para capturar de forma adequada estruturas sequenciais complexas [Katz94]. As Máquinas de Estado Algorítmicas (*Algorithmic State Machines – ASMs*) foram introduzidas por *Clare* em [Clare73] e são um método alternativo para descrever o comportamento de uma FSM de forma semelhante a um fluxograma.

Os diagramas ASM representam graficamente as funções de saída e de transição de uma FSM, sendo particularmente úteis para projectar unidades de controlo que implementem algoritmos. No final do projecto podem também ser usadas para efeitos de documentação.

Um diagrama ASM consiste num ou mais blocos ASM interligados. A estrutura de um bloco ASM está representada na Figura 4.2. Cada um descreve a operação da máquina de estados durante o intervalo de tempo em que o respectivo estado está activo e representa o estado actual, as saídas, as saídas condicionais e o estado seguinte para cada combinação das entradas. Consequentemente, as funções de saída e de transição são representadas estado a estado, devendo as ligações dos blocos ASM ser feitas de forma a que haja apenas um estado seguinte para cada estado e conjunto estável de entradas [Clare73].

O bloco ASM representado na Figura 4.2 possui um caminho de entrada e um número variável de caminhos de saída. Internamente é constituído por uma caixa de estado e uma rede composta por um número qualquer (inclusive 0) de caixas de decisão e de saídas condicionais.

Um estado é representado por uma caixa de estado, a qual possui a seguinte informação: um nome dentro de uma circunferência acima do canto superior esquerdo ou direito da referida caixa; um código, o qual provavelmente não está definido quando se inicia a descrição; uma lista de saídas escrita no interior da caixa de estado e escolhidas de acordo com a operação que se pretende realizar. A lista de saídas indica o conjunto de sinais que devem ser activados quando se entra num dado estado. É possível especificar se o sinal é activado imediatamente ou se deve ser activado apenas no próximo flanco do relógio. Normalmente os sinais cuja activação deve ser realizada imediatamente possuem o prefixo I, ao contrário dos sinais com atraso que não têm qualquer prefixo.

As caixas de decisão contêm entradas da máquina de estados e estabelecem as condições que controlam as transições de estados e a activação das saídas condicionais. Cada caixa contém uma expressão Booleana que determina o próximo bloco ASM activo. A caixa de decisão possui dois caminhos de saída. O caminho de saída "Verdadeiro" é normalmente indicado por um 1 ou T (*true*) e é escolhido quando a condição toma o valor lógico verdadeiro. Por outro lado, se a condição possuir o valor lógico falso é escolhido o caminho de saída "Falso", indicado por 0 ou F (*false*). A ordem pela qual estão ligadas as caixas de decisão dentro de um bloco ASM é irrelevante na determinação do próximo bloco activo.

A caixa de saídas condicionais contém, à semelhança da caixa de estado, uma lista de saídas, que ao contrário das anteriores não dependem só do estado mas também das condições de entrada. Os sinais desta lista podem também ser afectados

por prefixos como no caso anterior para especificar se a sua activação deve ser imediata ou atrasada um ciclo de relógio.

Enquanto a lista de saídas da caixa de estado é utilizada no modelo de Moore, a lista de saídas condicionais é utilizada no modelo de Mealy.

Para simplificar a construção de um esquema ASM a caixa que delimita um bloco ASM pode ser omitida porque um bloco é claramente identificado pelo conjunto das caixas de decisão e de saídas condicionais que existem entre um dado estado e o seu sucessor. Além disso, algumas caixas condicionais podem ser partilhadas por mais do que um estado.

O modelo ASM está bem documentado em [WinPro80, Green86].

Na Figura 4.3 está ilustrado o diagrama ASM da máquina de venda automática atrás apresentada. Da observação da figura e por comparação com o STD da Figura 4.1 pode-se concluir que este método possui a vantagem de descrever algoritmicamente o comportamento do referido sistema. No entanto, possui as mesmas limitações que os STDs na descrição de sistemas complexos, uma vez que não suporta explicitamente a especificação de hierarquia e concorrência. Mais à frente serão apresentados outros formalismos, os Esquemas de Grafos Hierárquicos e suas variantes, que combinam a vantagem de uma descrição algorítmica com a possibilidade de especificar de forma explícita a hierarquia e a concorrência.

Finalmente, se esta notação for utilizada tal como na definição inicial, em que se usam os blocos base delimitados, ou seja, sem as simplificações acima descritas o diagrama resultante é consideravelmente maior do que o STD equivalente.

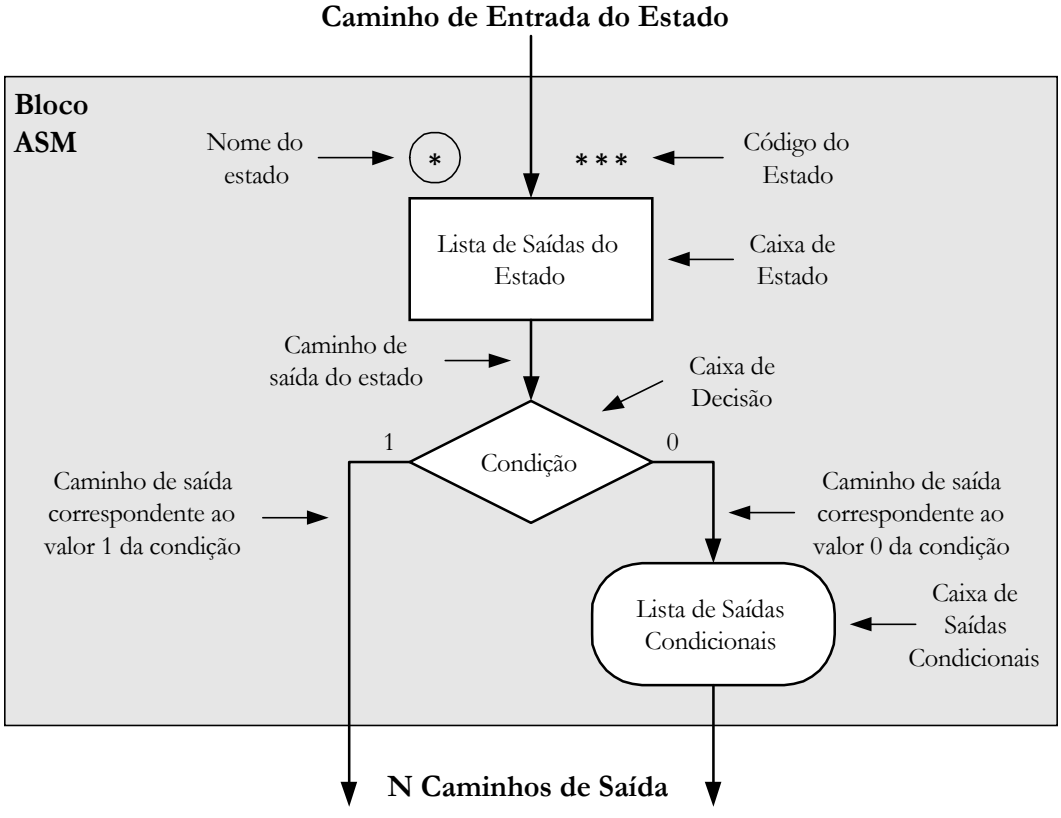


Figura 4.2 – Estrutura do bloco base usado na construção de máquinas de estado algorítmicas.

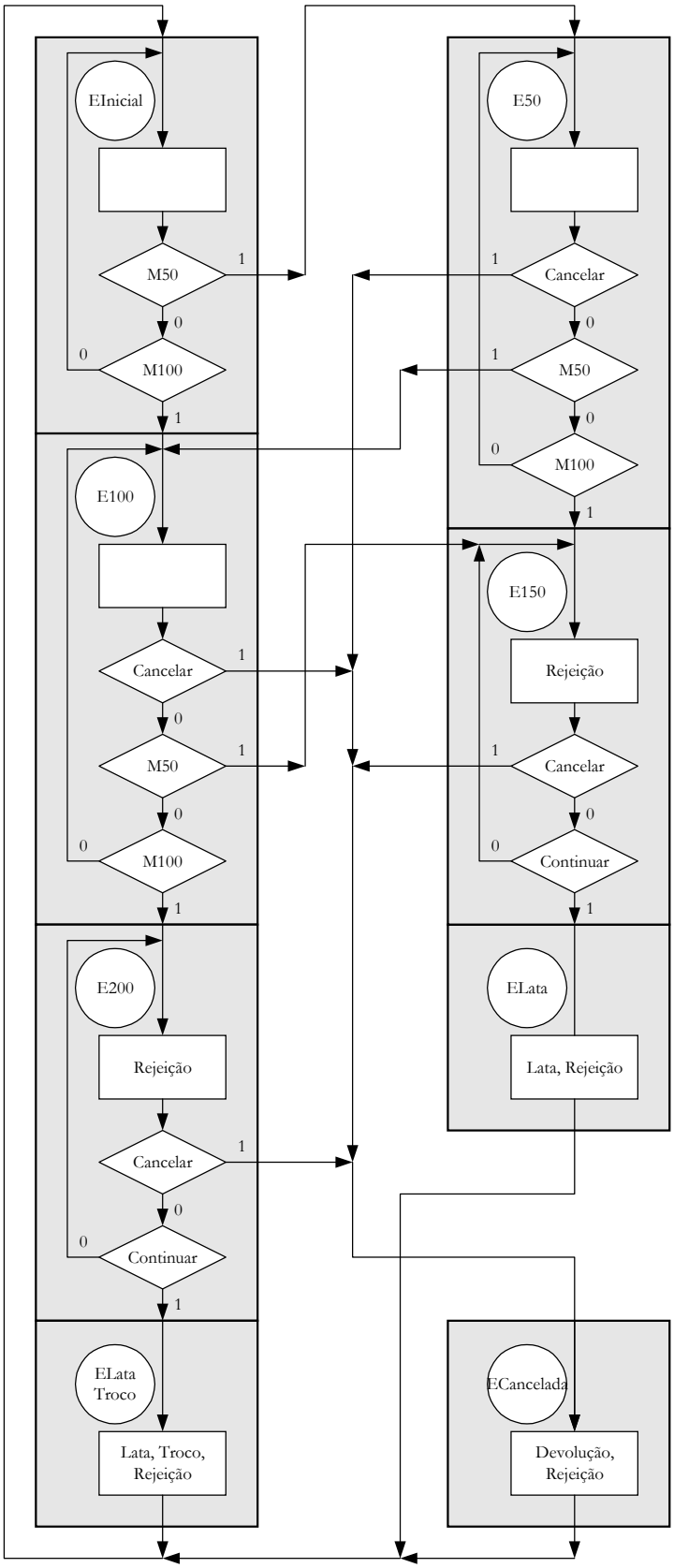


Figura 4.3 – Exemplo de uma máquina de estados algorítmica para descrição do comportamento da unidade de controlo da máquina de venda automática.

4.4 Redes de Petri

As redes de Petri foram já apresentadas no capítulo anterior como um modelo orientado ao estado, logo passível de ser utilizado na descrição do comportamento de unidades de controlo. No entanto, tal como foi referido atrás, a barreira entre as linguagens de especificação e os modelos é por vezes muito ténue, pelo que as redes de Petri são novamente aqui abordadas como uma linguagem de especificação e em particular para apresentar o exemplo da máquina de venda automática, cuja rede de Petri se encontra na Figura 4.4. Mais precisamente, nesta figura encontra-se um caso particular de uma rede de Petri, designada por máquina de estados e que pode ser usada para descrever qualquer FSM. Esta variante é caracterizada por possuir só um testemunho e cada transição ter apenas um arco de entrada e um arco de saída. Os lugares correspondem aos estados do STD da Figura 4.1. O estado inicial possui um testemunho. As transições estão etiquetadas com as condições responsáveis pela mudança de estado.

Para tornar a figura mais clara só são representadas as condições principais responsáveis por cada transição e não são indicadas quaisquer saídas activas quer nos lugares, quer nas transições. A Figura 4.4 pode ser completada de forma trivial a partir da análise do STD correspondente.

Sempre que um lugar possuir mais do que uma transição de saída diz-se, consoante as aplicações, que existe um conflito, decisão ou escolha. A máquina de estados permite a representação de conflitos mas não a sincronização de actividades paralelas. De facto o conflito só existe ao nível da configuração da rede de Petri, uma vez que numa FSM as transições devem ser etiquetadas de forma a que em cada momento só uma se encontre activa.

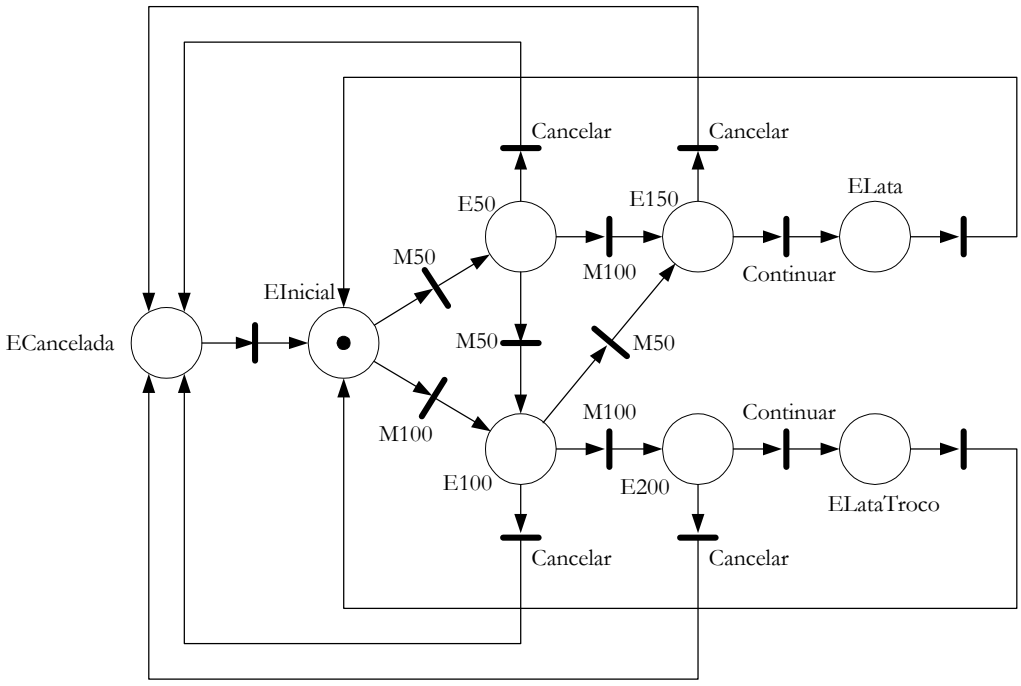


Figura 4.4 – Exemplo de uma rede de Petri para descrição do comportamento da unidade de controlo da máquina de venda automática.

4.5 Esquemas de Grafos

Os Esquemas de Grafos de um algoritmo (*Graph Schemes of algorithm – GSs*) são um método de especificação gráfica que consiste num grafo dirigido composto por diferentes tipos de nodos interligados por arcos que definem os fluxos de execução possíveis de um algoritmo. Os GSs foram propostos em [Baranov74] e são também apresentados em [Baranov94]. A sua construção é bastante semelhante à de um fluxograma. A Figura 4.5 ilustra os vários tipos de nodos que podem aparecer num GS. Cada GS é composto pelo menos por dois nodos rectangulares com os cantos arredondados: um inicial denominado "*Begin*" e um final denominado "*End*" que determinam respectivamente o início e o fim do algoritmo. Além destes, um GS é também constituído por um conjunto finito de nodos operacionais (rectangulares) e um conjunto finito de nodos condicionais (losangulares). Os primeiros são utilizados para activar uma lista de sinais de saída quando o respectivo nodo é activado. Os nodos losangulares representam um teste ao valor lógico de um sinal de entrada que determinará o caminho a seguir no grafo, influenciando o fluxo de execução do algoritmo.

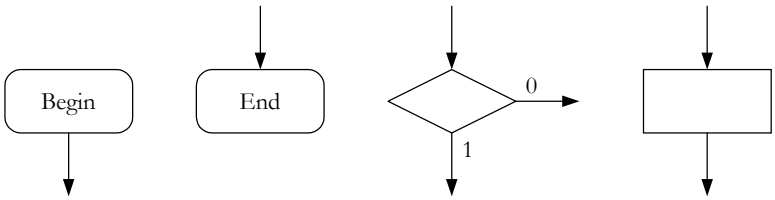


Figura 4.5 – Tipos de nodos de um GS.

Um GS possui a seguinte descrição formal [Baranov94]:

- Cada GS possui um ponto de entrada representado por um nodo rectangular com os cantos arredondados e marcado com a etiqueta *Begin* e um ponto de saída que é também identificado por um nodo semelhante ao anterior mas marcado com a etiqueta *End*;
- Os nodos operacionais são representados graficamente por rectângulos e possuem microinstruções do conjunto $O=\{o_1, o_2, \ldots, o_K\}$. Qualquer microinstrução o_k possui um subconjunto de microoperações do conjunto $Y=\{y_1, y_2, \ldots, y_N\}$. Uma microoperação é um sinal de saída da unidade de controlo que provoca uma acção simples na unidade de execução, tal como o carregamento de um registo ou o incremento de um contador. É possível utilizar a mesma microinstrução em nodos operacionais diferentes. De facto, a noção de microinstrução é em tudo idêntica à de vector de saída usada na definição de FSM, pelo que são usados os mesmos símbolos;
- Cada nodo condicional possui apenas um elemento do conjunto $X=\{x_1, x_2, \ldots, x_M\}$, que contém todas as condições lógicas. Uma condição lógica é um sinal de entrada, que comunica o resultado de um teste, como por exemplo o estado de um sensor ou o resultado de uma comparação. Diferentes nodos condicionais podem conter a mesma condição lógica;

- Todos os nodos possuem apenas uma entrada, à excepção do *Begin*, que não possui entrada. Todos os nodos operacionais possuem uma saída. O nodo *End* não possui saída. Um nodo condicional tem duas saídas marcadas com as etiquetas "0" (falso) e "1" (verdadeiro);
- As entradas e as saídas dos nodos são ligadas por linhas dirigidas (arcos) de uma saída para uma entrada, tal que:
  - Uma saída ligue apenas a uma entrada;
  - Uma entrada fique ligada a pelo menos uma saída;
  - Um nodo deve ficar ligado de forma a fazer parte de pelo menos um caminho desde o nodo *Begin* até ao nodo *End*.

Na Figura 4.6 está ilustrado o GS que descreve o comportamento da unidade de controlo da máquina de venda automática. Neste caso assume-se que quando é atingido o nodo "*End*" a execução é automaticamente reiniciada no nodo "*Begin*". De notar que apesar da semelhança visual entre os GSs e as ASMs, em particular se não forem usados os rectângulos delimitadores, os dois formalismos possuem diferenças fundamentais, das quais se realçam as seguintes:

- Enquanto nas ASMs está definido um bloco básico a ser usado na construção desse tipo de diagramas, nos GSs as entidades fundamentais são os vários tipos de nodos que podem ser interligados de qualquer forma desde que se respeitem as regras acima apresentadas;
- Ao contrário das ASMs, os GSs possuem nodos especiais de "*Begin*" e "*End*" que representam os pontos de entrada e de saída de um algoritmo;
- Nas ASMs existe uma correspondência de um para um entre as caixas de estado e os estados da FSM correspondente. No caso da máquina de Moore as saídas são especificadas no interior das caixas de estado enquanto na máquina de Mealy são colocadas nas caixas de saídas condicionais. Por oposição num GS não existe uma correspondência directa entre os nodos operacionais e os estados, devendo ser realizada a sua marcação com etiquetas que representam os estados. As regras de marcação dependem do tipo de modelo adoptado (Moore ou Mealy) e estão apresentadas em [Rocha99, Melo00]. Este é um dos motivos pelo qual se considera que os GS proporcionam uma descrição que é independente da implementação do circuito.

Tal como os STDs e as ASMs, os GSs também não suportam a representação explícita de concorrência e hierarquia, pelo que não são adequados para descrever sistemas complexos. No entanto, os Esquemas de Grafos Hierárquicos (*Hierarchical Graph Schemes - HGSs*), introduzidos em [Sklyarov84], suportam descrições hierárquicas baseadas no uso de macrooperações e de funções lógicas. Os Esquemas de Grafos Hierárquicos Paralelos (*Parallel Hierarchical Graph Schemes - PHGSs*), introduzidos em [Sklyarov87], além das descrições hierárquicas permitem que as macrooperações sejam invocadas em paralelo. Ambos são apropriados para descrever sistemas complexos e são descritos na próxima secção.

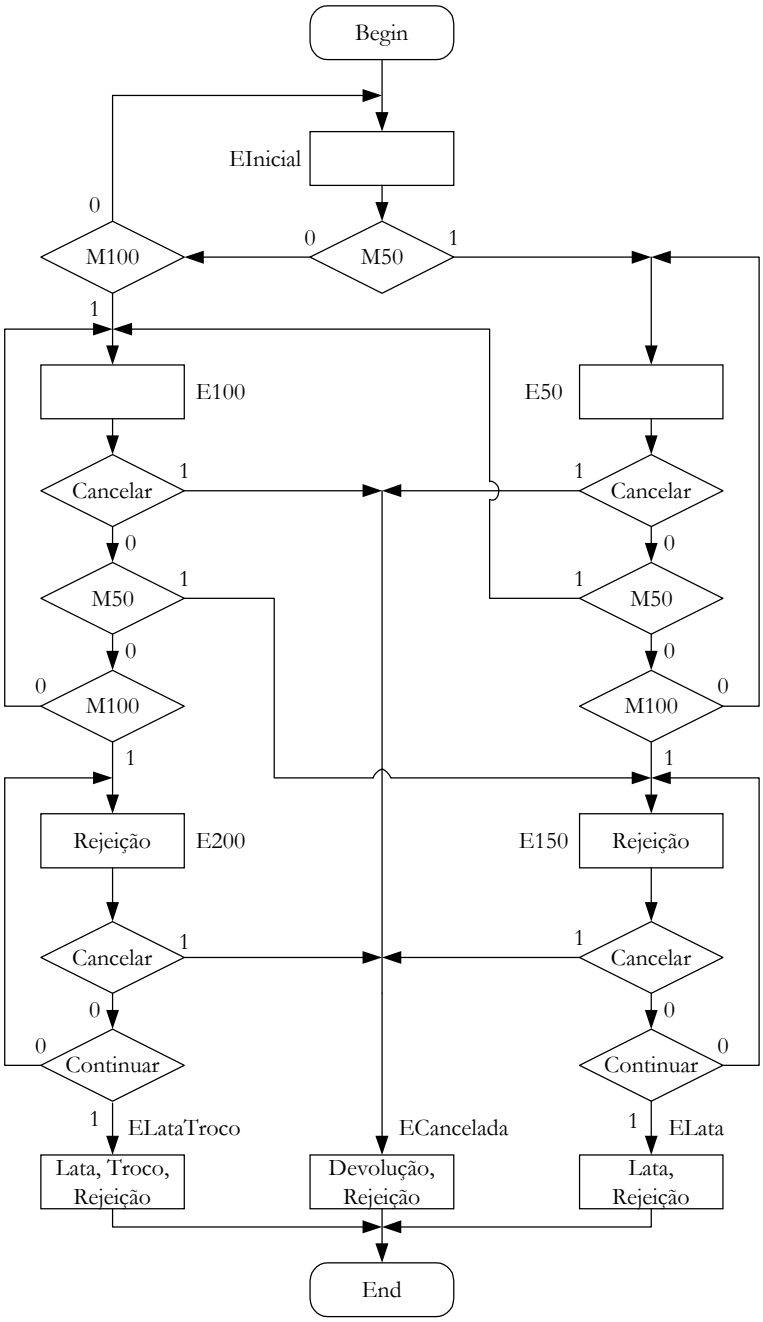


Figura 4.6 – Exemplo de um esquema de grafos para descrição do comportamento da unidade de controlo da máquina de venda automática.

4.6 Esquemas de Grafos Hierárquicos

Os Esquemas de Grafos Hierárquicos (*Hierarchical Graph Schemes – HGSs*) foram introduzidos em [Sklyarov84] e são basicamente GSs com as seguintes características adicionais:

- Os seus nodos operacionais podem conter quer microinstruções do conjunto $O=\{o_1, o_2, \ldots, o_K\}$, quer macroinstruções do conjunto $\Phi=\{\phi_1, \phi_1, \ldots, \phi_I\}$, ou ambas. Qualquer macroinstrução ϕ_i é constituída por

um subconjunto de macrooperações do conjunto $T=\{\tau_1, \tau_2, \ldots, \tau_J\}$. Cada macrooperação τ_j é descrita por outro HGS de nível hierárquico inferior. Por enquanto vamos considerar apenas processos sequenciais, isto é, sem paralelismo, pelo que cada macroinstrução inclui apenas uma macrooperação;

- Os seus nodos condicionais contêm apenas um elemento do conjunto $X \cup \Psi$, onde $X=\{x_1, x_2, \ldots, x_M\}$ é o conjunto de condições lógicas e $\Psi=\{\psi_1, \psi_2, \ldots, \psi_V\}$ é o conjunto de funções lógicas. Cada função lógica é calculada executando um conjunto predefinido de passos sequenciais descritos por um HGS de nível hierárquico inferior.

Consideremos o conjunto $\Theta=\{\theta_1, \theta_2, \ldots, \theta_A\}$ definido como $\Theta=T \cup \Psi$. Cada elemento $\theta_a \in \Theta$ corresponde a um HGS Γ_a que descreve o algoritmo que realiza θ_a (se $\theta_a \in T$) ou calcula θ_a (se $\theta_a \in \Psi$). Em ambos os casos o algoritmo é descrito por um HGS de nível hierárquico inferior. Vamos assumir que $T(\Gamma_a)$ é o subconjunto das macrooperações e $\Psi(\Gamma_a)$ é o subconjunto das funções lógicas que pertencem ao HGS Γ_a. Se $T(\Gamma_a) \cup \Psi(\Gamma_a)=\varnothing$ então o algoritmo possui apenas um nível de representação, sendo portanto um GS ordinário.

Um algoritmo de uma unidade de controlo descrito de forma hierárquica pode ser especificado por um conjunto de HGSs que descrevem a componente principal e todos os elementos do conjunto Θ. A componente principal é descrita pelo HGS Γ_1, a partir do qual se inicia a execução do algoritmo de controlo. Todos os restantes HGSs serão chamados por Γ_1, ou por outros HGSs descendentes de Γ_1.

A Figura 4.7 mostra a descrição do algoritmo de controlo do exemplo da máquina de venda automática. De notar que neste caso $T=\varnothing$ e $\Psi=\{\psi_1\}$, pelo que não é utilizado nenhum HGS que descreva uma macrooperação e ψ_1 é usada para calcular a função "Terminar". No entanto, como um HGS que representa uma macrooperação é em tudo idêntico ao HGS principal da Figura 4.7, não é apresentado nenhum exemplo adicional para ilustrar esta situação. À semelhança dos GSs assume-se que quando é atingido o nodo "*End*" do HGS principal, a execução é automaticamente reiniciada no nodo "*Begin*".

Algumas operações num HGS podem ser designadas como virtuais. Uma macrooperação ou função lógica é virtual se não for associada de forma permanente a um HGS durante o projecto da unidade de controlo. Desta forma, qualquer elemento virtual pode possuir no futuro diferentes implementações, pelo que um HGS virtual pode ser visto como uma parte modificável de um algoritmo de controlo.

Os Esquemas de Grafos Hierárquicos Paralelos (*Parallel Hierarchical Graph Schemes – PHGSs*) foram propostos em [Sklyarov87]. A diferença fundamental relativamente aos HGSs é a possibilidade de uma macroinstrução poder ser constituída por mais do que uma macrooperação, devendo todas ser executadas em paralelo. A transição de qualquer nodo operacional com uma microinstrução e/ou uma macroinstrução para o nodo seguinte é efectuada quando todas as suas componentes tiverem terminado.

Os HGSs possibilitam o desenvolvimento de algoritmos de controlo complexos de forma estruturada, uma vez que suportam a representação de hierarquia. Desta forma é possível realizar a concepção e teste de cada HGS separadamente e em certos casos efectuar a sua reutilização em projectos futuros. Por outro lado, como os PHGSs suportam a descrição de paralelismo, permitem ao contrário dos STDs, ASMs e GSs especificar de forma eficiente unidades de controlo compostas por subcomportamentos concorrentes. Os HGSs e PHGSs proporcionam também uma boa separação entre a funcionalidade da unidade de controlo e a sua implementação.

Finalmente, como suportam a especificação de operações virtuais são apropriados para descrever a funcionalidade de unidades ou circuitos de controlo virtuais, os quais são caracterizados por permitirem alterar o seu comportamento após o seu projecto ter sido concluído e em certos casos até dinamicamente, permitindo assim alcançar um dos objectivos deste trabalho.

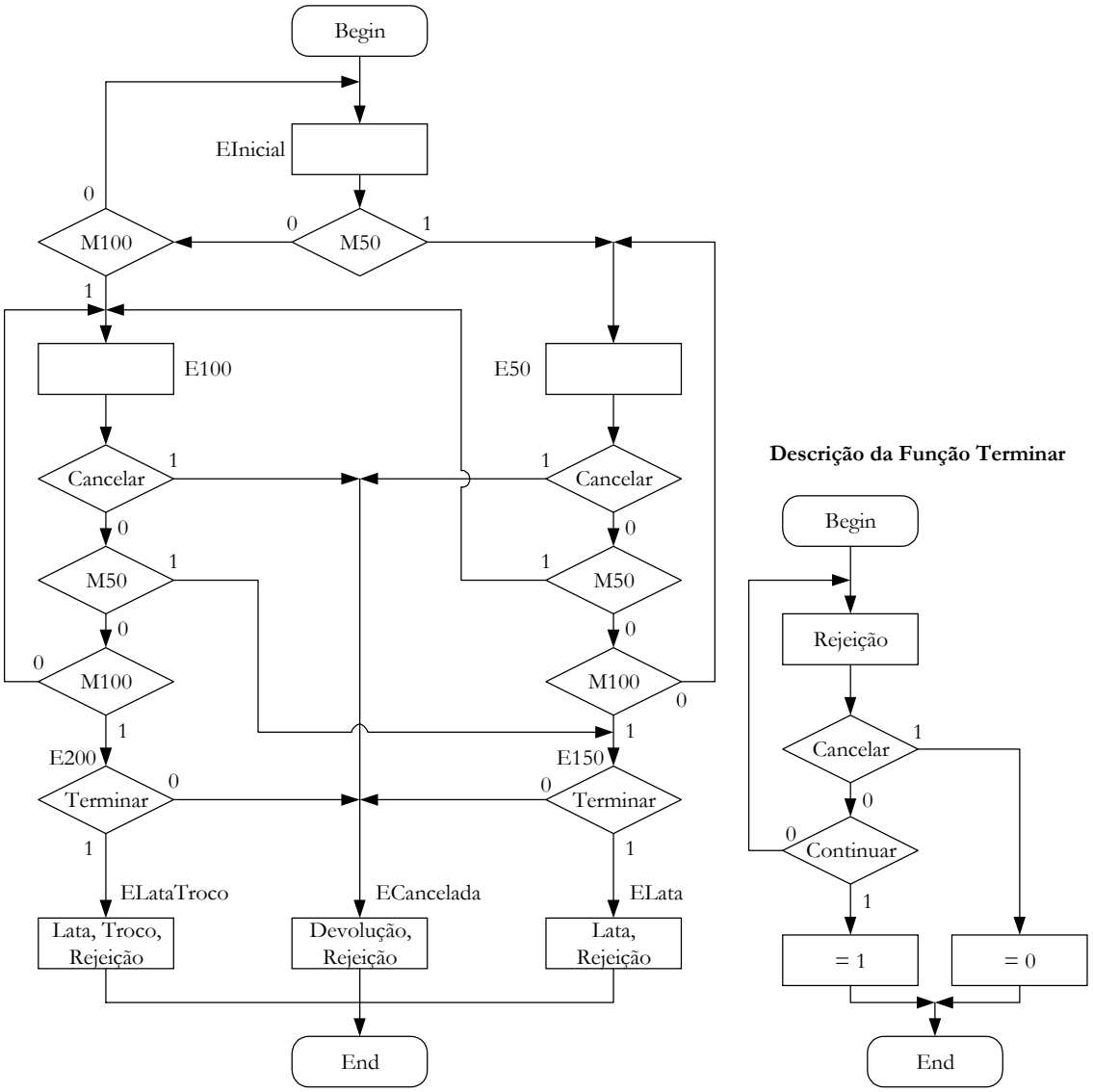


Figura 4.7 – Exemplo de um esquema de grafos hierárquico para descrição do comportamento da unidade de controlo da máquina de venda automática.

4.7 Statecharts

A linguagem *Statecharts* foi introduzida em [Harel87] como um formalismo visual para especificar o comportamento de sistemas reactivos complexos [DruHar89, Statemate90]. Com esta linguagem podem ser obtidas representações elegantes graças à extensão do modelo FSM tradicional, de forma a incluir três elementos adicionais muito importantes na descrição de sistemas de dimensão considerável: a hierarquia, a concorrência e a comunicação. De facto, este formalismo para além de ser uma linguagem é muitas vezes considerado também um modelo devido às potencialidades adicionais que possui relativamente aos modelos tradicionais de máquinas de estado.

Como o exemplo da máquina de venda automática não é apropriado para ilustrar todos os aspectos importantes da sintaxe e da semântica dos *Statecharts*, vai ser usado o exemplo da Figura 4.8 e que foi apresentado em [DruHar89]. Para tornar a figura mais clara, não são representadas quaisquer saídas.

Tal como o modelo FSM, os *Statecharts* baseiam-se em estados, eventos e condições. Os eventos e as condições são responsáveis pelas transições entre os estados. Os estados e as transições podem ser associados de várias formas com eventos de saída, chamados acções, as quais podem ser executadas durante o disparo de uma transição ou durante a entrada, saída, ou permanência num estado.

Os *Statecharts* combinam os modelos FSM de Moore e de Mealy e adicionam-lhes construções hierárquicas e concorrentes, para desta forma ultrapassar as limitações do modelo FSM convencional na especificação de sistemas reactivos complexos [DruHar89].

Os *Statecharts* são uma linguagem gráfica onde os estados são representados por rectângulos de cantos arredondados e que podem ser sucessivamente combinados em estados de níveis hierárquicos mais elevados. Alternativamente, os estados de qualquer nível podem ser divididos em estados de nível hierárquico inferior através da utilização de dois tipos diferentes de decomposição: sequencial (OR) e concorrente (AND).

A Figura 4.8 mostra um exemplo de um estado AND *A* composto por dois estados *B* e *C* separados por uma linha a tracejado, o que significa que quando o sistema se encontra em *A*, os estados *B* e *C* devem-se encontrar ambos activos. Por outras palavras, *B* e *C* são estados ortogonais. Por outro lado, *B* e *C* são estados OR, o que significa que quando o sistema se encontrar em *B* somente um dos estados *D*, *E* ou *F* deve estar activo. De forma semelhante, quando o sistema estiver em *C* um dos estados *G* ou *H* deve estar activo. Os estados (*D*, *E*, *F*) e os estados (*G*, *H*) são exclusivos. Assim, quando o sistema estiver no estado *A* existem as seguintes configurações possíveis (*D*, *G*); (*D*, *H*); (*E*, *G*); (*E*, *H*); (*F*, *G*); (*F*, *H*).

As setas com origem num ponto e que terminam nos estados *A*, *E* e *G* designam-se por "setas por defeito" e significam que estes são os estados iniciais de *N*, *B* e *C*, respectivamente, pelo que o estado inicial do sistema é o *A*, sendo a sua configuração inicial (*E*, *G*).

Nos *Statecharts* as transições não estão restringidas a um só nível hierárquico, podendo partir de um estado em qualquer nível para qualquer outro estado [DruHar89]. Na Figura 4.8 estão mostrados alguns exemplos. O evento *a* provoca a transição do estado *K* para o estado *L*. O evento *b* causa a transição do estado *J*, ou seja, do estado *L* ou *M* para o estado *K*. O evento *c* provoca a transição de *A*, ou seja,

de uma das configurações acima enumeradas para o estado *M*. No caso do evento *d*, a transição faz-se para o superestado *J*, ou por outras palavras para o seu estado inicial *L*, para o qual aponta a seta por defeito.

Em geral, as transições fazem-se de configurações para configurações com a possibilidade de componentes ortogonais quer nos estados de origem, quer nos estados de destino. O evento *f* provoca a transição da configuração (*F*, *H*) para o estado *P*, ou deste para a configuração (*D*, *H*).

Se a configuração activa for (*E*, *G*), o evento *m* provoca a transição simultânea de *E* para *F* e de *G* para *H*, enquanto o evento *p* causa a transição de *E* para *D* independentemente do que ocorra em *C*.

A concorrência e a independência nos *Statecharts* são possíveis graças à ortogonalidade dos estados [DruHar89].

As saídas podem estar associadas às transições, tal como numa FSM de Mealy, através da adição de uma etiqueta do tipo *a*/*o* ao longo de um arco, o que significa que o evento *a* é responsável pela activação da acção *o*. Esta acção pode também ser associada à entrada (*entry o*), à saída (*exit o*), ou à permanência num estado (*throughout o*), de forma análoga a uma FSM de Moore. Em qualquer dos casos *o* pode ser um evento externo (acção) ou interno que pode ser usado para sincronizar outras transições noutros estados ortogonais [DruHar89].

Os *Statecharts* permitem a especificação de informação temporal num estado. No entanto, como esta informação pode ser especificada em estados pertencentes a qualquer nível hierárquico e em qualquer componente ortogonal, os *Statecharts* possibilitam na realidade a especificação de restrições temporais globais [DruHar89].

Os *Statecharts* fornecem vários métodos de sincronização. Um evento que atinja a linha delimitadora de uma caixa de estado sincroniza esse estado. Por exemplo, o evento *e* na Figura 4.8 efectua a reinicialização do estado *A* para a sua configuração inicial (*E*, *G*). Outra forma de sincronização consiste na activação de uma variável interna que actuará como evento para sincronizar outras transições.

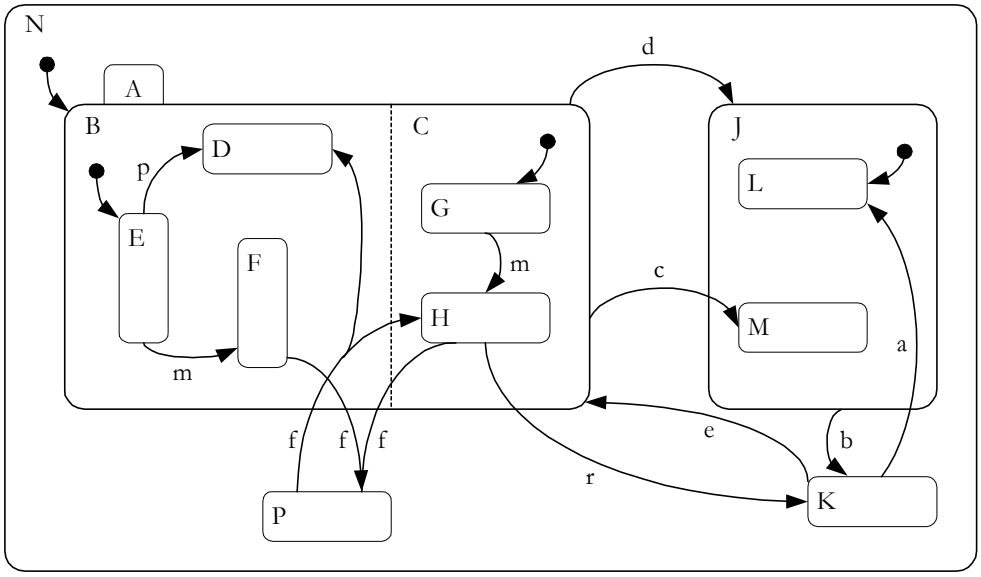


Figura 4.8 – Exemplo de um *Statechart* genérico que ilustra a capacidade de descrição de hierarquia e concorrência.

Os *Statecharts* possibilitam a descrição de sistemas reactivos complexos pois suportam descrições hierárquicas e concorrentes, especificações temporais e vários mecanismos de sincronização. No entanto, tal como qualquer outra linguagem ou modelo orientado ao estado, este formalismo é adequado somente para sistemas predominantemente de controlo, nos quais as operações realizadas sobre os dados estão associadas com as actividades contidas nos estados ou ao longo das transições [DruHar89]. Consequentemente os *Statecharts* não são apropriados para descrever sistemas complexos que requerem estruturas de dados complexas [GajVahNarGon94].

O diagrama *Statechart* da máquina de venda automática está apresentado na Figura 4.9. A principal vantagem resultante da utilização deste formalismo reside na redução do número de arcos do diagrama. Isto é possível graças à possibilidade do agrupamento hierárquico de estados, em particular, daqueles em que pode ocorrer o cancelamento de uma compra. De facto, de todas as descrições gráficas, esta é aquela que proporciona uma forma mais compacta para descrever o comportamento do exemplo utilizado. No entanto, relativamente ao outro formalismo apresentado que suporta hierarquia e concorrência (os Esquemas de Grafos Hierárquicos), os *Statecharts* têm a desvantagem de não proporcionarem uma descrição algorítmica. Além disso, são de mais difícil verificação e nem todos os aspectos são implementáveis, razão pela qual ainda não foi apresentada nenhuma arquitectura de hardware que permita tirar partido de todas as potencialidades do modelo de Máquina de Estados Finitos Concorrente Hierárquica baseada em *Statecharts* e discutida em [GajVahNarGon94]. Os HGSs têm relativamente aos *Statecharts* a desvantagem de serem muito mais restritivos ao nível to tipo de transições permitidas e no tipo de mecanismos de sincronização e concorrência que disponibiliza. No entanto, possuem a vantagem de suportar recursividade de forma natural, ao contrário dos *Statecharts* em que a forma mais intuitiva para especificar recursividade colide com o método de sincronização de superestados acima descrito.

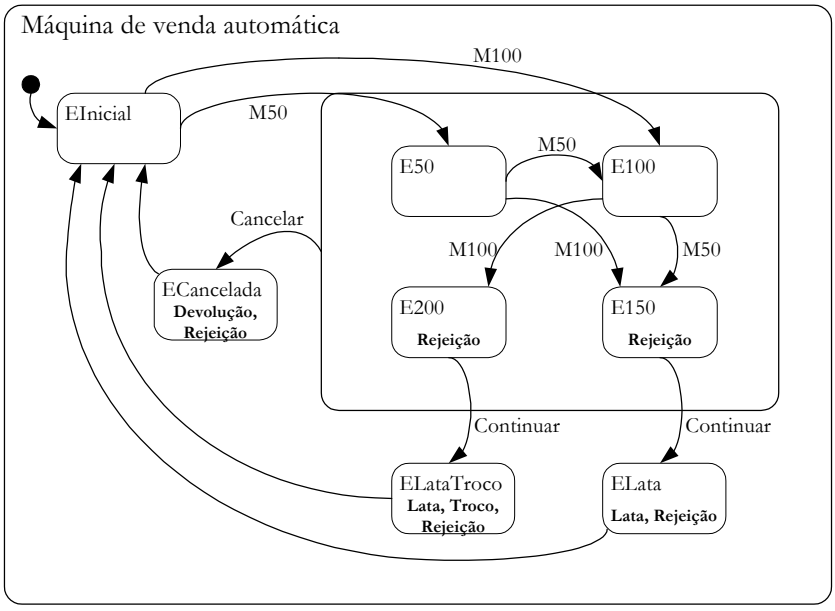


Figura 4.9 – Exemplo de *Statechart* para descrição do comportamento da unidade de controlo da máquina de venda automática.

4.8 VHDL

A linguagem VHDL (*VHSIC Hardware Description Language*) foi desenvolvida pelo Departamento de Defesa dos Estados Unidos no âmbito do programa *Very High Speed Integrated Circuits* e estandardizada pelo IEEE em 1987. Esta linguagem destina-se a suportar o desenvolvimento, documentação e intercâmbio de projectos de sistemas digitais. Desde o seu desenvolvimento tem sido bastante utilizada pelos projectistas como uma linguagem de descrição de hardware. Actualmente, existe uma grande variedade de ferramentas para captura, simulação, verificação e síntese, o que constitui uma das razões mais importantes para a sua grande aceitação e uso generalizado. Nesta secção vai ser feita uma breve introdução ao VHDL de forma a permitir compreender a descrição da máquina de venda automática que será à frente apresentada. Em [Ashenden96] é feita uma apresentação completa desta linguagem.

A principal abstracção do VHDL é a entidade, a qual é usada para representar um componente ou bloco de qualquer complexidade. Uma entidade realiza uma função específica, possui entradas e saídas bem definidas (chamados portos) e é identificada pela palavra chave *entity*. Na declaração da entidade além dos portos, podem também ser definidos outros parâmetros como por exemplo, informação temporal. A funcionalidade de uma entidade é especificada por uma arquitectura, a qual é identificada pela palavra chave *architecture*. Esta funcionalidade pode depender dos parâmetros definidos no interface da entidade. A sua descrição pode ser feita de forma comportamental usando construções semelhantes às existentes nas linguagens de programação usuais, de forma estrutural através da instanciação de componentes predefinidos ou na forma de fluxo de dados através de registos e barramentos. Finalmente pode também ser usada qualquer combinação das anteriores.

Em VHDL a hierarquia estrutural é suportada através da instanciação e inclusão de componentes predefinidos dentro de outros. A interligação dos diversos componentes é realizada por intermédio de sinais.

O VHDL permite descrever comportamentos concorrentes através da utilização de processos. Um processo é identificado pela palavra chave *process* e é composto por uma parte declarativa e um corpo. Um processo pode estar activo ou temporariamente suspenso. No primeiro caso, a sua execução é feita em paralelo com a dos restantes processos do sistema. Um processo pode ser tornado sensível a determinados eventos através da utilização de uma lista sensitiva. O corpo de um processo é sequencial, a execução é realizada em tempo zero e disparada pelos eventos declarados na lista sensitiva. No seu interior podem ser usadas funções, procedimentos e outras construções semelhantes às linguagens de programação tradicionais, além da atribuição de valores a sinais e variáveis.

O VHDL suporta hierarquia comportamental a dois níveis. No primeiro nível a especificação pode ser decomposta num conjunto de processos que executam de forma concorrente. No segundo nível pode ser feita a decomposição sequencial destes processos em procedimentos.

O VHDL possui dois tipos de objectos, nomeadamente os sinais e as variáveis, que podem ser utilizados para armazenar valores numéricos ou informação em geral. Os sinais possuem significado em hardware e diferem das variáveis no facto de possuírem uma componente temporal associada. A instrução *after* permite que as

instruções de atribuição de valor a um sinal efectuem o escalonamento futuro das actualizações. Os sinais podem ser utilizados em blocos sequenciais e concorrentes de VHDL, podem ser globais, mas só podem ser declarados em blocos concorrentes. Por outro lado, as variáveis são usadas principalmente para armazenar valores temporários, só podem ser utilizados em blocos sequenciais de VHDL e são locais ao bloco onde foram declaradas.

Para além das construções de programação acima referidas e que podem ser usadas dentro de um processo, o VHDL possui também uma grande gama de tipos de dados apropriados para criar modelos comportamentais de alto nível, tais como inteiro, real, enumerado, físico, matriz, registo e ponteiros. Os operadores lógicos, relacionais e aritméticos estão também disponíveis. No entanto, os últimos só podem ser aplicados aos tipos de dados inteiro e real.

A comunicação entre processos em VHDL é realizada através do paradigma de memória partilhada baseada em sinais, aos quais o processo emissor pode atribuir valores que podem lidos pelos processos receptores.

A sincronização pode ser feita de duas formas distintas. A primeira é baseada na lista sensitiva de um processo, que assegura que a execução do processo é iniciada quando ocorrer um evento num dos sinais da sua lista sensitiva. Consideremos como exemplo o seguinte processo:

```
P : process(x, y)
begin
   …
end process;
```

De acordo com esta definição, a execução do processo *P* estará suspensa até a ocorrência de um evento em qualquer dos sinais *x* ou *y*, permitindo assim sincronizar a execução do processo *P* com a execução de outros processos ou componentes que modifiquem os referidos sinais.

A segunda forma de sincronização utiliza a instrução *wait*, que suspende a execução do processo até que seja detectada a ocorrência de um evento num dos sinais especificados ou a condição especificada seja verdadeira. Por exemplo, a seguinte instrução *wait* assegura que a execução do processo será retomada somente quando ocorrer um evento nos sinais *x* ou *y* e quando *z* for igual a 1.

```
wait on x, y until (z = '1');
```

A instrução *wait* pode também ser utilizada da seguinte forma para suspender a execução de um processo por um tempo predefinido:

```
wait for 10 ns;
```

Finalmente, consideremos as seguintes expressões de atribuição de um valor a um sinal:

```
a <= b;
b <= a;
```

```
wait on a;
```

Como um processo executa em tempo zero, estas duas atribuições são realizadas em simultâneo, trocando os valores dos sinais *a* e *b*. De seguida, o processo em que se encontram fica suspenso até à ocorrência de um evento no sinal *a*.

O VHDL não suporta de forma explícita a especificação de transições de estado. Como veremos de seguida a descrição de máquinas de estado é baseada em processos. Na Figura 4.10 encontra-se a listagem VHDL que descreve o comportamento da unidade de controlo da máquina de venda automática. Os portos da entidade MAQUINA_VENDA consistem nas entradas e saídas da unidade de controlo e nos seus sinais de sincronismo e inicialização. Os estados estão enumerados na parte declarativa da arquitectura COMPORTAMENTAL, onde são também declarados os sinais que vão ser usados para armazenar o estado actual e o estado seguinte. A descrição de uma unidade de controlo em VHDL é normalmente baseada em dois processos [Skahill96]. O primeiro é responsável pela inicialização e pelas transições de estado sincronizadas pelos flancos do sinal de relógio, pelo que a sua lista sensitiva contém somente os sinais de sincronismo e de inicialização. O segundo processo determina o estado seguinte a partir do estado actual e das entradas e também as saídas em função do estado actual, uma vez que foi utilizado o modelo de Moore. A sua lista sensitiva é composta pelos sinais de entrada e pelo estado actual.

```
library ieee;
use ieee.std_logic_1164.all;

entity MAQUINA_VENDA is
    port(RELOGIO, INICIALIZACAO : in STD_LOGIC;
         M50, M100, CONTINUAR, CANCELAR : in STD_LOGIC;
         LATA, TROCO, DEVOLUCAO, REJEICAO : out STD_LOGIC);
end MAQUINA_VENDA;

architecture COMPORTAMENTAL of MAQUINA_VENDA is

    type ESTADO is (EINICIAL, E50, E100, E150, E200, ELATA, ELATATROCO, ECANCELADA);

    signal EACTUAL, ESEGUINTE : ESTADO;

begin

    PROC_SINCR : process(RELOGIO, INICIALIZACAO)
    begin
        if (INICIALIZACAO = '1') then
            EACTUAL <= EINICIAL;
        elsif (RELOGIO'event and RELOGIO = '1') then
            EACTUAL <= ESEGUINTE;
        end if;
    end process;

    PROC_TRANS : process(EACTUAL, M50, M100, CONTINUAR, CANCELAR)
    begin
        case EACTUAL is
            when EINICIAL =>
                LATA <= '0';
                TROCO <= '0';
                DEVOLUCAO <= '0';
                REJEICAO <= '0';
                if (M50 = '1') then
                    ESEGUINTE <= E50;
                elsif (M100 = '1') then
                    ESEGUINTE <= E100;
                else
                    ESEGUINTE <= EINICIAL;
```

```
            end if;

        when E50 =>
            LATA <= '0';
            TROCO <= '0';
            DEVOLUCAO <= '0';
            REJEICAO <= '0';
            if (CANCELAR = '1') then
                ESEGUINTE <= ECANCELADA;
            elsif (M50 = '1') then
                ESEGUINTE <= E100;
            elsif (M100 = '1') then
                ESEGUINTE <= E150;
            else
                ESEGUINTE <= E50;
            end if;

        when E100 =>
            LATA <= '0';
            TROCO <= '0';
            DEVOLUCAO <= '0';
            REJEICAO <= '0';
            if (CANCELAR = '1') then
                ESEGUINTE <= ECANCELADA;
            elsif (M50 = '1') then
                ESEGUINTE <= E150;
            elsif (M100 = '1') then
                ESEGUINTE <= E200;
            else
                ESEGUINTE <= E100;
            end if;

        when E150 =>
            LATA <= '0';
            TROCO <= '0';
            DEVOLUCAO <= '0';
            REJEICAO <= '1';
            if (CANCELAR = '1') then
                ESEGUINTE <= ECANCELADA;
            elsif (CONTINUAR = '1') then
                ESEGUINTE <= ELATA;
            else
                ESEGUINTE <= E150;
            end if;

        when E200 =>
            LATA <= '0';
            TROCO <= '0';
            DEVOLUCAO <= '0';
            REJEICAO <= '1';
            if (CANCELAR = '1') then
                ESEGUINTE <= ECANCELADA;
            elsif (CONTINUAR = '1') then
                ESEGUINTE <= ELATATROCO;
            else
                ESEGUINTE <= E200;
            end if;

        when ELATA =>
            LATA <= '1';
            TROCO <= '0';
            DEVOLUCAO <= '0';
            REJEICAO <= '1';
            ESEGUINTE <= EINICIAL;

        when ELATATROCO =>
            LATA <= '1';
            TROCO <= '1';
            DEVOLUCAO <= '0';
            REJEICAO <= '1';
            ESEGUINTE <= EINICIAL;

        when ECANCELADA =>
            LATA <= '0';
```

```
                TROCO <= '0';
                DEVOLUCAO <= '1';
                REJEICAO <= '1';
                ESEGUINTE <= EINICIAL;
        end case;
    end process;
end COMPORTAMENTAL;
```

Figura 4.10 – Listagem VHDL que descreve o comportamento da unidade de controlo da máquina de venda automática.

Na Figura 4.11 encontra-se a listagem do módulo VHDL utilizado para simular a entidade MAQUINA_VENDA da Figura 4.10. Este tipo de módulo é vulgarmente designado por *testbench* e é caracterizado por:

- Não possuir quaisquer entradas ou saídas;
- Instanciar as entidades que se pretender simular, bem como realizar a sua interligação;
- Gerar todos os sinais de entrada dessas entidades de acordo com um escalonamento temporal que permita simular as condições pretendidas.

A entidade da *testbench* foi chamada TESTE e a sua arquitectura ESTIMULOS.

O interface do componente MAQUINA_VENDA está definido na parte declarativa da arquitectura ESTIMULOS, onde também é feita a associação da sua instância MV com a arquitectura COMPORTAMENTAL da entidade MAQUINA_VENDA. Finalmente, nesta secção são declarados todos os sinais necessários e que coincidem com os portos da entidade MAQUINA_VENDA.

A geração dos sinais de entrada do componente MAQUINA_VENDA está distribuída por dois processos definidos no corpo da arquitectura ESTIMULOS. O primeiro (PROC_RELOG) é utilizado para gerar o sinal de relógio responsável pelas transições de estado. No segundo processo (PROC_ESTIM) são geradas as sequências de vectores de entrada que simulam as operações de compra. Inicialmente é realizado um procedimento de inicialização de forma a colocar a unidade de controlo num estado bem definido.

Seguidamente são realizadas três operações de compra: uma sem troco, uma com troco e uma com cancelamento, que podem ser descritas da seguinte forma:

- No primeiro caso são aplicados três impulsos ao sinal "M50" para simular a introdução de três moedas de 50 Esc, sendo de seguida aplicado um impulso ao sinal "Continuar" para concluir a compra, a qual termina com a activação da saída "Lata".
- No segundo caso são aplicados dois impulsos ao sinal "M100" para simular a introdução de duas moedas de 100 Esc, sendo de seguida aplicado um impulso ao sinal "Continuar" para concluir a compra, a qual termina com a activação das saídas "Lata" e "Troco".
- Finalmente, no terceiro e último caso são aplicados um impulso ao sinal "M100" e um impulso ao sinal "M50" para simular respectivamente a introdução de uma moeda de 100 Esc e uma de 50 Esc, sendo de seguida aplicado um impulso ao sinal "Cancelar" para abortar a compra, a qual termina com a activação da saída "Devolução".

A modificação de um sinal é seguida de um tempo de espera, que precede a alteração do sinal seguinte e durante o qual o processo é suspenso. De notar que os tempos utilizados são meramente simbólicos, não correspondendo minimamente a uma situação real.

Para evitar que a mesma moeda seja responsável por mais do que uma transição de estado, assume-se a existência de um circuito externo que garanta que as linhas de cada sensor só estão activas durante um ciclo de relógio consecutivo.

```
library ieee;
use ieee.std_logic_1164.all;

entity TESTE is
end TESTE;

architecture ESTIMULOS of TESTE is

    component MAQUINA_VENDA
        port(RELOGIO, INICIALIZACAO : in STD_LOGIC;
             M50, M100, CONTINUAR, CANCELAR : in STD_LOGIC;
             LATA, TROCO, DEVOLUCAO, REJEICAO : out STD_LOGIC);
    end component;

    signal RELOGIO, INICIALIZACAO : STD_LOGIC;
    signal M50, M100, CONTINUAR, CANCELAR : STD_LOGIC;
    signal LATA, TROCO, DEVOLUCAO, REJEICAO : STD_LOGIC;

    for MV : MAQUINA_VENDA use entity WORK.MAQUINA_VENDA(COMPORTAMENTAL);

begin

    MV : MAQUINA_VENDA port map(RELOGIO, INICIALIZACAO,
                                M50, M100, CONTINUAR, CANCELAR,
                                LATA, TROCO, DEVOLUCAO, REJEICAO);

    PROC_RELOG : process
        variable TEMP : std_logic := '1';
    begin
        TEMP := not TEMP;
        RELOGIO <= TEMP;
        wait for 10 ns;
    end process;

    PROC_ESTIM : process
    begin
        INICIALIZACAO <= '1';
        M50 <= '0';
        M100 <= '0';
        CONTINUAR <= '0';
        CANCELAR <= '0';
        wait for 30 ns;
        INICIALIZACAO <= '0';
        wait for 30 ns;

        M50 <= '1';
        wait for 20 ns;
        M50 <= '0';
        wait for 40 ns;
        M50 <= '1';
        wait for 20 ns;
        M50 <= '0';
        wait for 60 ns;
        M50 <= '1';
        wait for 20 ns;
        M50 <= '0';
        wait for 20 ns;
        CONTINUAR <= '1';
        wait for 20 ns;
```

```
        CONTINUAR <= '0';
        wait for 80 ns;

        M100 <= '1';
        wait for 20 ns;
        M100 <= '0';
        wait for 40 ns;
        M100 <= '1';
        wait for 20 ns;
        M100 <= '0';
        wait for 60 ns;
        CONTINUAR <= '1';
        wait for 20 ns;
        CONTINUAR <= '0';
        wait for 80 ns;

        M100 <= '1';
        wait for 20 ns;
        M100 <= '0';
        wait for 60 ns;
        M50 <= '1';
        wait for 20 ns;
        M50 <= '0';
        wait for 40 ns;
        CANCELAR <= '1';
        wait for 20 ns;
        CANCELAR <= '0';
        wait for 80 ns;
    end process;
end ESTIMULOS;
```

Figura 4.11 – Listagem VHDL da *testbench* da máquina de venda automática.

Na Figura 4.12 encontram-se as formas de onda resultantes da simulação da entidade MAQUINA_VENDA da Figura 4.10 com a *testbench* da Figura 4.11. O simulador utilizado foi o *VeriBest VHDL 15.01*. As três operações de compra acima referidas encontram-se assinaladas na figura. Além dos sinais de entrada e saída são também mostrados os estados internos da unidade de controlo ao longo da simulação.

Para concluir este capítulo, gostaríamos de referir que a linguagem de especificação adoptada e que será utilizada na generalidade dos capítulos que se seguem foi os Esquemas de Grafos Hierárquicos, uma vez que como suportam hierarquia, recursividade e operações virtuais, adaptam-se bem às características e objectivos deste trabalho.

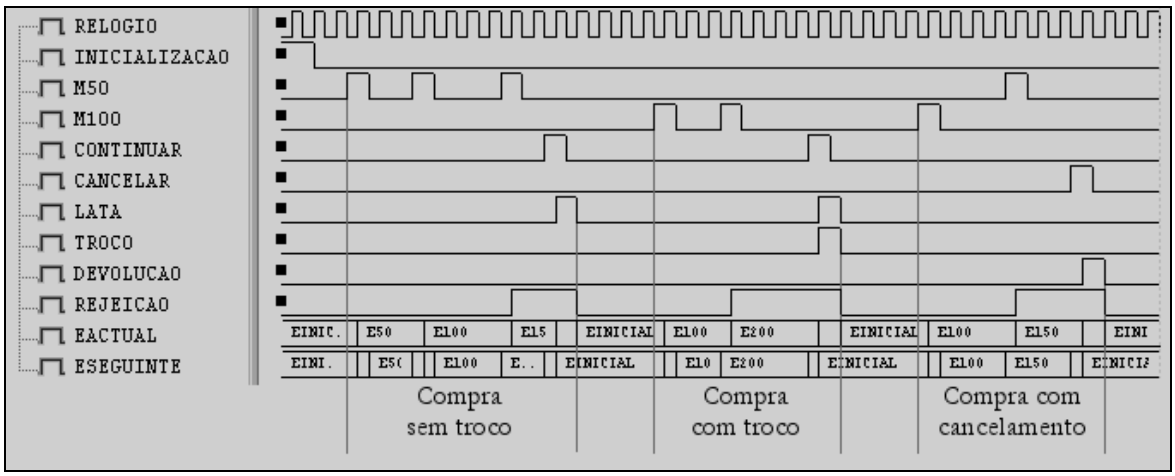


Figura 4.12 – Resultado da simulação da máquina de venda automática em VHDL.

5 Síntese de Unidades de Controlo

Sumário

Neste capítulo é feita uma descrição resumida do processo de síntese das unidades de controlo configuradas fisicamente (*hardwired*). Além da apresentação de alguns conceitos teóricos, são também objectivos deste capítulo a análise dos resultados obtidos com diferentes técnicas de síntese e o estabelecimento de uma metodologia de projecto para unidades de controlo a implementar em FPGAs da família XC6200 da Xilinx. Esta metodologia deve ter como ponto de partida uma descrição comportamental e permitir a utilização de ferramentas de desenvolvimento assistido por computador em todas as etapas de projecto, desde a especificação até à depuração. Nas secções iniciais são revistos alguns dos conceitos fundamentais da síntese lógica sequencial, mais concretamente, algumas definições e nomenclatura e as suas subtarefas mais importantes, nomeadamente a minimização de estados, a codificação de estados e a optimização lógica. No contexto da minimização de estados são apresentados os conceitos de estados equivalentes e de estados compatíveis. No âmbito da codificação de estados é referido um método heurístico que procura reduzir o número de literais das funções lógicas e maximizar a partilha de subexpressões comuns na parte combinatória de um circuito sequencial implementado com lógica multi-nível. A optimização lógica é também abordada embora de forma necessariamente muito superficial. Para analisar o impacto dos vários tipos de codificação de estados na área e frequência de funcionamento máxima do circuito foram realizadas várias experiências com diferentes ferramentas de síntese. Estas experiências serviram também para eleger o método de codificação de estados *one-hot* como o mais apropriado para as unidades de controlo a implementar em dispositivos da arquitectura XC6200. As suas maiores vantagens são a simplicidade do processo de síntese, o facto de proporcionar um bom desempenho do circuito e necessitar em geral de primitivas com um reduzido número de entradas, sendo este um factor importante em arquitecturas de granulosidade fina como a XC6200.

5.1 Introdução

A síntese de um sistema destinado a uma implementação em VLSI gera normalmente um conjunto de unidades de controlo e de unidades de execução. A função de cada uma delas já foi introduzida no capítulo 1 desta dissertação. Cada unidade de controlo ou controlador pode ser descrito usando vários modelos, dos quais a máquina de estados finitos (*Finite State Machine – FSM*) ou as suas variantes hierárquicas e/ou paralelas descritas no capítulo 3 são o exemplo mais comum. A síntese de uma unidade de controlo transforma a representação comportamental obtida após a sua especificação, num modelo estrutural constituído por um conjunto de componentes e suas interligações. Os componentes podem ser simples portas lógicas e flip-flops, ou alternativamente, dispositivos lógicos programáveis (*Programmable Logic Devices - PLDs*), tais como PROMs, PLAs, PALs, CPLDs e FPGAs. As vantagens dos dois últimos tipos de dispositivos e em particular das FPGAs são:

- Uma elevada capacidade lógica, possuindo um grande número de portas lógicas e flip-flops que podem ser ligados de forma arbitrária;
- A versatilidade e na maior parte dos casos a reprogramabilidade, permitindo a adopção de metodologias de projecto mais interactivas, o que se traduz em ciclos de projecto bastante curtos e na correcção fácil de erros.

Sendo a unidade de controlo um circuito sequencial modelado normalmente como uma FSM, ao seu processo de síntese dá-se o nome de síntese de lógica sequencial, correspondendo a um caso particular da síntese lógica com algumas etapas adicionais que serão discutidas mais à frente. No caso de ser utilizado o modelo FSM ordinário, a síntese de unidades de controlo deve converter a especificação funcional ou modelo comportamental numa estrutura de hardware constituída por um registo de estado e um circuito combinatório que gera o estado seguinte e as saídas do controlador. As tarefas envolvidas na criação dessa estrutura incluem a minimização de estados, a codificação de estados, a optimização lógica e mapeamento na tecnologia.

A minimização de estados reduz o número de estados de uma FSM através da substituição de estados equivalentes ou compatíveis por um único estado. Dois estados são equivalentes se a sequência de saídas para qualquer sequência de entradas não depender do estado que for escolhido como ponto de partida. A noção de estado compatível é mais complexa e será apresentada na secção 5.3.2. A minimização de estados é importante uma vez que o número de estados determina o tamanho do registo de estado e da lógica que implementa as funções de transição e de saída. Na secção 5.3 este assunto será abordado de forma mais detalhada.

A codificação de estados atribui códigos binários aos estados simbólicos da FSM. O principal objectivo é obter um conjunto de códigos que permita reduzir tanto quanto possível a quantidade de lógica combinatória da unidade de controlo. Os algoritmos utilizados para a codificação de estados dependem do tipo de implementação pretendida, nomeadamente lógica de dois níveis, tipicamente na forma de soma de produtos ou lógica multi-nível consistindo na interligação arbitrária de portas lógicas de uma dada biblioteca da tecnologia alvo. A codificação de estados será analisada na secção 5.4.

A optimização lógica é realizada após a codificação de estados para melhorar determinados parâmetros do circuito combinatório que implementa as funções de transição e de saída. Os parâmetros podem ser vários, desde a área do circuito, os seus atrasos ou a sua testabilidade. Em muitos casos os resultados obtidos resultam de uma solução de compromisso entre estes parâmetros.

Apesar de uma parte significativa da optimização de uma unidade de controlo ser realizada ao nível comportamental (ex. minimização de estados) usando modelos orientados ao estado como a FSM, esta pode não ser a melhor abordagem. Isto deve-se à falta de correlação entre certas optimizações que são realizadas a este nível e as correspondentes variações da área e atrasos do circuito. Um dos modelos onde é possível estabelecer esta correspondência é o de rede lógica síncrona, no qual se baseiam algumas técnicas de optimização recentes, como por exemplo a de *retiming* [Micheli94]. No entanto, somente o modelo FSM é extensivamente suportado pela maioria das ferramentas de síntese.

Enquanto a lógica combinatória de dois níveis é normalmente implementada numa PLA ou numa PAL, a lógica multi-nível pode ser implementada em dispositivos lógicos programáveis multi-nível como as FPGAs ou em ASICs construídos a partir de bibliotecas de lógica standard. O mapeamento na tecnologia transforma uma rede lógica independente da tecnologia, produzida pelas ferramentas de optimização lógica, num circuito com primitivas específicas da tecnologia de implementação utilizada.

A síntese de uma unidade de controlo pode ser realizada de forma manual ou automática. Com a crescente complexidade dos sistemas e consequentemente das suas unidades de controlo, devido à necessidade de tempos de projecto cada vez menores e à existência de ferramentas de síntese, o segundo método é cada vez mais utilizado. Neste capítulo vai ser ilustrado o processo manual de síntese de unidades de controlo, bem como apresentados alguns exemplos de circuitos sintetizados com o auxilio de ferramentas para este fim. Na Figura 5.1 estão ilustradas as tarefas normalmente realizadas na síntese manual de circuitos sequenciais. Além das que foram já referidas acima, tais como a minimização de estados, a codificação de estados e a optimização lógica, na síntese manual é também vulgar a construção de tabelas de transição de estados e de saída a partir da sua especificação (normalmente gráfica) baseada, por exemplo, num diagrama de transição de estados (*State Transition Diagram – STD*) ou em esquemas de grafos. Outras tarefas que fazem parte da síntese manual são a determinação das equações de transição e das saídas, a escolha do tipo de flip-flops utilizados para armazenar os estados e derivação das suas equações de excitação.

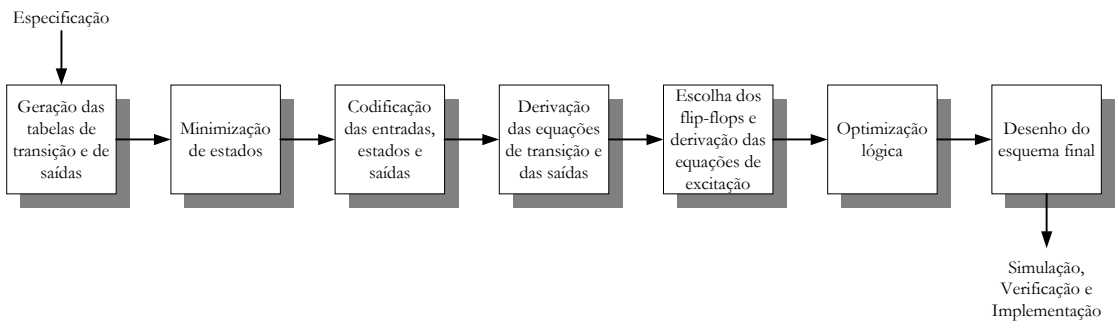


Figura 5.1 – Tarefas normalmente realizadas na síntese manual de circuitos sequenciais.

Como ponto de partida da síntese automática de uma unidade de controlo pode ser utilizada uma descrição gráfica, desde que se possua uma ferramenta apropriada para este fim, ou então uma descrição numa linguagem de descrição de hardware. Os métodos mais utilizados para especificação de circuitos sequenciais foram revistos no capítulo 4 desta dissertação.

As próximas secções descrevem resumidamente cada uma das etapas de síntese de uma unidade de controlo, com a excepção do mapeamento na tecnologia. Muitas das ideias apresentadas são um resumo de [Micheli94]. Ao longo deste capítulo vai ser utilizado, sempre que possível, o exemplo da unidade de controlo da máquina de venda automática (Figura 5.2) apresentado no capítulo anterior. Por conveniência, o seu STD é novamente apresentado na Figura 5.3. Os respectivos processos de síntese manual e automática serão ilustrados a partir da secção 5.6.

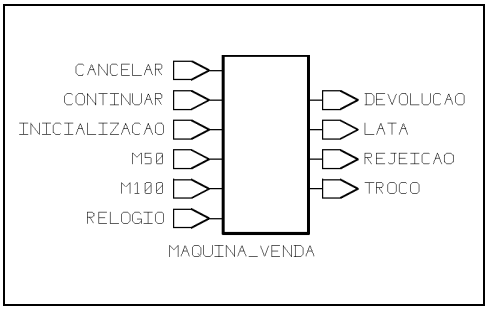


Figura 5.2 – Representação da unidade de controlo da máquina de venda automática do ponto de vista de entradas e saídas.

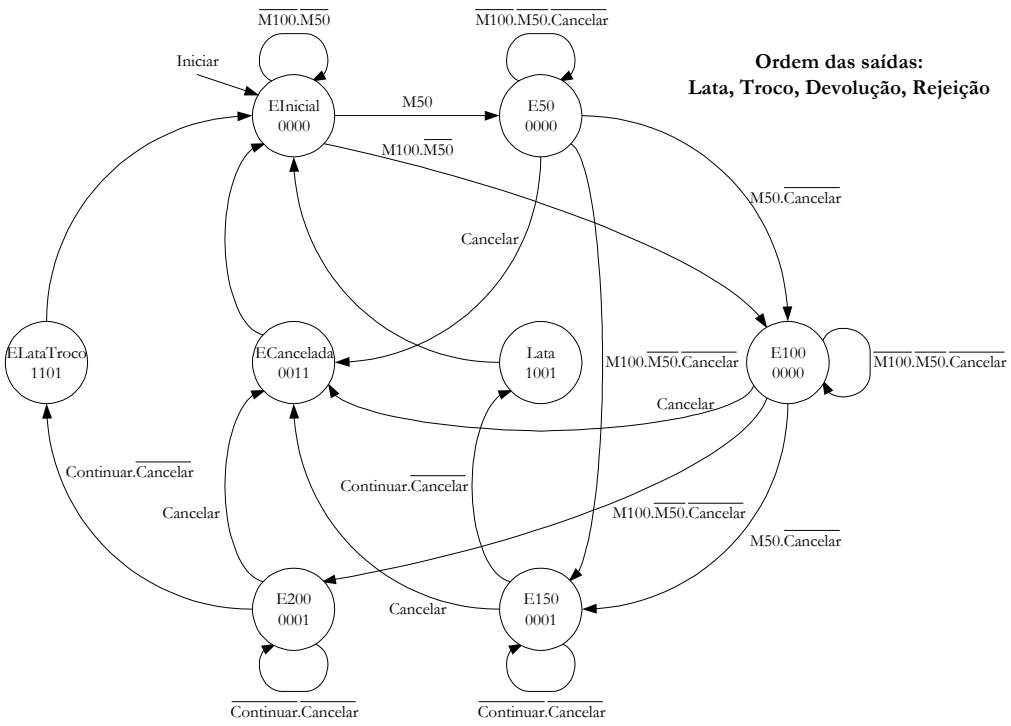


Figura 5.3 – Diagrama de transição de estados que descreve o comportamento da unidade de controlo da máquina de venda automática.

5.2 Codificação de Entradas e Saídas

Na maioria das situações as entradas e saídas de uma unidade de controlo são codificadas de forma a que cada linha represente apenas um sinal. No entanto, em certos casos pode ser vantajoso realizar a sua codificação de forma a que o número de linhas aplicadas ao núcleo da unidade de controlo seja minimizado. Para tal são necessários esquemas de redução ou codificação das entradas e de expansão ou descodificação das saídas. De notar que estas técnicas reflectem-se apenas na estrutura interna da unidade de controlo, uma vez que o seu interface com o exterior permanece inalterado. Em termos de implementação, estas técnicas traduzem-se na divisão da unidade de controlo em duas partes principais: um núcleo, semelhante às arquitecturas apresentadas no capítulo 3 e um ou vários circuitos de codificação/descodificação das entradas/saídas que permitem efectuar a sua minimização. Existem várias arquitecturas para este fim, das quais se destacam as seguintes:

- **Redução por multiplexagem do número de entradas** (Figura 5.4) - Na generalidade dos algoritmos de controlo, o número de entradas que afectam as transições de um dado estado é normalmente um subconjunto da totalidade das linhas de entrada da unidade de controlo. Assim, o número de entradas do seu núcleo pode ser reduzido através da utilização de um multiplexador controlado directamente ou indirectamente pelas variáveis de estado. Este multiplexador selecciona as entradas que são relevantes para cada estado e aplica-as ao núcleo. De notar, que no limite pode-se fazer com que cada transição dependa apenas de uma variável de entrada através da sua decomposição e inserção de estados adicionais. No entanto, esta abordagem faz com que o número de estados da unidade de controlo seja mais elevado e a execução do algoritmo de controlo mais demorada. Esta técnica será utilizada em 5.6 para facilitar a realização da optimização lógica do circuito;
- **Redução por codificação do número de entradas** (Figura 5.5) - Quando o valor de algumas variáveis de entrada é conhecido em determinados estados, o número de entradas pode ser reduzido através de um esquema de codificação. Este método tem relativamente ao anterior a vantagem de não necessitar de sinais adicionais para controlo do multiplexador. No entanto, possui a desvantagem de ter uma aplicabilidade mais reduzida;
- **Redução por descodificação do número de saídas** (Figura 5.6) - Sempre que exista um conjunto de saídas que sejam activadas de forma disjunta pode ser utilizado um esquema de descodificação de saídas que faz com que o número de variáveis controladas directamente pelo núcleo da unidade de controlo possa ser reduzido;
- **Fusão de saídas não ortogonais** – Dois sinais de saída são ortogonais se e só se existirem pelo menos dois vectores para os quais possuam valores distintos e diferentes de *don't care*. Dois ou mais sinais não ortogonais podem ser substituídos por apenas um.

A técnica de redução do número de entradas por multiplexagem é considerada em [Rocha99] enquanto as restantes são discutidas em [Melo00].

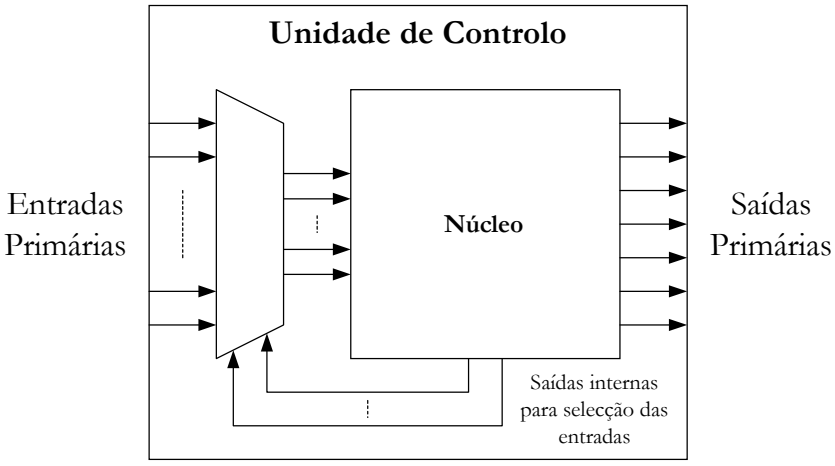


Figura 5.4 – Redução por multiplexagem do número de entradas aplicadas ao núcleo da unidade de controlo.

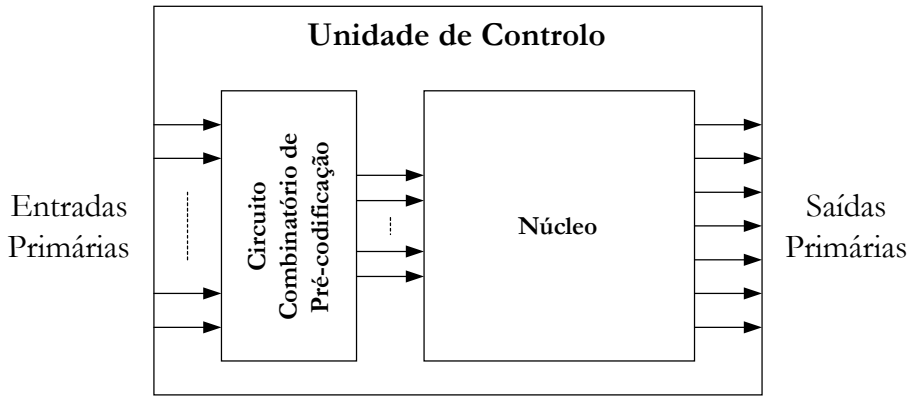


Figura 5.5 – Redução por codificação do número de entradas aplicadas ao núcleo da unidade de controlo.

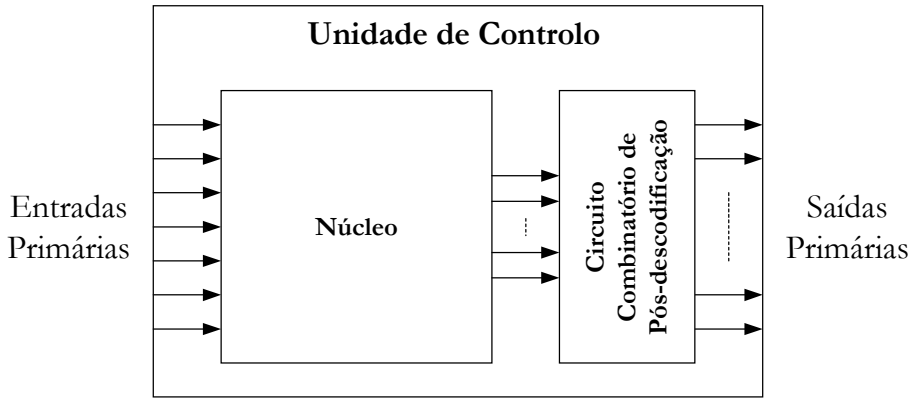


Figura 5.6 – Redução por descodificação do número de saídas controladas pelo núcleo da unidade de controlo.

5.3 Minimização de Estados

A minimização de estados tem como objectivo a redução do número de estados usados na descrição de uma unidade de controlo. No caso do modelo FSM ser capturado por um STD, a minimização de estados resulta numa diminuição do número de vértices do grafo e consequentemente do número de arcos. Nesta secção vamos considerar que a unidade de controlo é baseada no modelo FSM. Os conceitos e técnicas aqui discutidas são igualmente válidas para as variantes hierárquicas e/ou paralelas deste modelo (HFSM, HPFSM, PHFSM, GFSM e VFSM) uma vez que podem ser aplicadas separadamente a cada uma das sub-máquinas.

A redução do número de estados pode estar relacionada com a diminuição do número de elementos de armazenamento (quando os estados são codificados com um número mínimo de bits, o número de flip-flops corresponde ao valor inteiro resultante do arredondamento por excesso do logaritmo na base 2 do número de estados e que é normalmente expresso por $\lceil \log_2 |S| \rceil$). A redução de estados relaciona-se com a diminuição do número de transições e consequentemente com a redução do número de portas lógicas da componente combinatória do circuito.

A minimização de estados pode ser definida informalmente como a derivação de uma FSM com um comportamento semelhante à original mas com um número mínimo de estados. Uma definição mais precisa depende se estamos a considerar FSMs completamente especificadas ou não. Esta decisão afecta o formalismo, a complexidade do problema e os algoritmos utilizados. Consequentemente, a minimização de estados é descrita separadamente para ambos os casos nas próximas subsecções.

5.3.1 Minimização de Estados para FSMs Completamente Especificadas

Quando se considera uma FSM completamente especificada, a função de transição *g* e a função de saída *h* estão definidas para cada par (entrada, estado) em $I \times S$. Dois estados são equivalentes se as sequências dos vectores de saída da FSM inicializada em cada um dos estados coincidirem para qualquer sequência de entrada. A equivalência é verificada através do seguinte teorema:

> Dois estados de uma FSM são equivalentes se e só se para qualquer entrada possuírem saídas idênticas e os estados seguintes correspondentes forem equivalentes.

Uma vez que a equivalência é simétrica, reflexiva e transitiva, os estados podem ser divididos em classes de equivalência, sendo a partição resultante única. Uma implementação de uma FSM com um número mínimo de estados é aquela em que cada classe é representada por apenas um estado. A minimização do número de estados de uma FSM completamente especificada consiste na determinação das classes de equivalência. Estas podem ser derivadas por refinamento iterativo de uma partição do conjunto de estados. A partição inicial será designada por P_1 e a partição de ordem i por P_i, onde i = 1, 2, …, n.

As partições podem ser determinadas da seguinte forma:

- Em primeiro lugar, os blocos da partição P_1 contêm os estados cujas saídas sejam iguais para qualquer que seja o vector de entrada, satisfazendo assim uma condição necessária de equivalência;
- Seguidamente os blocos da partição são refinados iterativamente através da sua divisão, satisfazendo o requisito de que todos os estados num dado bloco de P_{i+1} devem possuir os estados seguintes no mesmo bloco de P_i para qualquer entrada possível;
- Quando a iteração convergir, isto é, quando $P_i = P_{i+1}$ para um valor de i, pelo teorema anterior os blocos da partição são classes de equivalência.

De notar que a convergência é sempre alcançada no máximo de n_s iterações em que n_s é a cardinalidade do conjunto dos estados. No caso limite em que sejam necessárias n_s iterações, obtém-se uma partição de grau 0 onde todos os blocos possuem somente um estado, isto é, nenhum par de estados é equivalente. A complexidade deste algoritmo é $O(n_s^2)$. Consideremos como exemplo de aplicação deste método o STD da Figura 5.7, cuja tabela de transição de estados é a seguinte:

Entrada	Estado Actual	Estado Seguinte	Saída
0	S_1	S_3	1
1	S_1	S_5	1
0	S_2	S_3	1
1	S_2	S_5	1
0	S_3	S_2	0
1	S_3	S_1	1
0	S_4	S_4	0
1	S_4	S_5	1
0	S_5	S_4	1
1	S_5	S_1	0

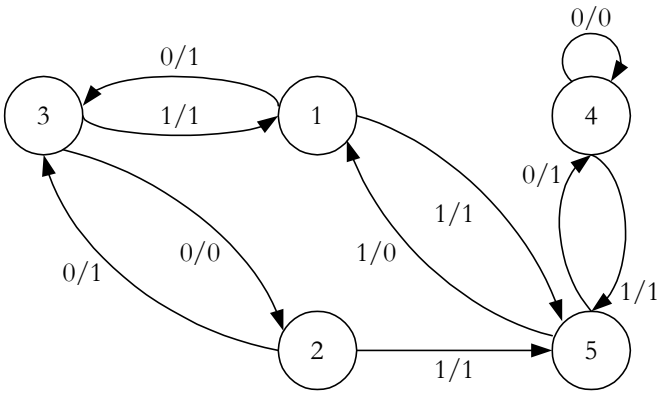


Figura 5.7 – Diagrama de transição de estados inicial (antes da minimização de estados) de uma FSM completamente especificada.

O conjunto dos estados pode ser inicialmente dividido de acordo com os valores das saídas, resultando na partição P_1:

$$P_1 = \{\{s_1, s_2\}, \{s_3, s_4\}, \{s_5\}\}$$

Seguidamente deve ser analisado cada bloco de P_1 para verificar se os estados seguintes correspondentes se encontram num único bloco quaisquer que sejam as entradas. Daqui conclui-se que os estados seguintes de s_1 e s_2 coincidem, enquanto os estados seguintes de s_3 e s_4 estão em blocos diferentes, pelo que o bloco {s3, s4} deve ser dividido, dando origem à partição P_2:

$$P_2 = \{\{s_1, s_2\}, \{s_3\}, \{s_4\}, \{s_5\}\}$$

Quando se analisa os blocos de P_2 conclui-se que não é possível realizar mais nenhum refinamento porque os estados seguintes de s_1 e s_2 coincidem. Consequentemente existem quatro classes de estados equivalentes. Na tabela seguinte o bloco $\{s_1, s_2\}$ é representado por s_{12}. O diagrama de estados correspondente é mostrado na Figura 5.8. De notar que a unidade de controlo da máquina de venda automática, apesar de ser um exemplo de uma FSM completamente especificada, não foi utilizada uma vez que já se encontra minimizada em termos do número de estados.

Entrada	Estado Actual	Estado Seguinte	Saída
0	S_{12}	S_3	1
1	S_{12}	S_5	1
0	S_3	S_{12}	0
1	S_3	S_{12}	1
0	S_4	S_4	0
1	S_4	S_5	1
0	S_5	S_4	1
1	S_5	S_{12}	0

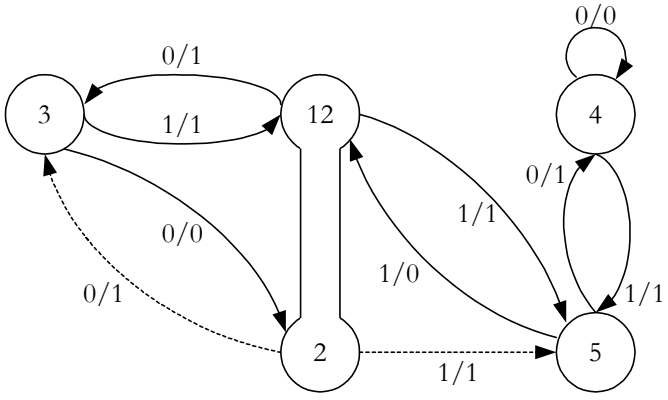


Figura 5.8 – Diagrama de transição de estados final (após a minimização de estados).

5.3.2 Minimização de Estados para FSMs Não Completamente Especificadas

No caso de FSMs não completamente especificadas a função de transição *g* e/ou a função de saída *h* não estão definidas para alguns pares (entrada, estado). As condições *don't care* representam as transições e as saídas não especificadas e modelam o conhecimento prévio de que alguns vectores de entrada não podem ocorrer em certos estados ou que algumas saídas não são observadas ou são irrelevantes nos respectivos estados sobre determinadas condições de entrada.

Diz-se que uma sequência de entrada é aplicável se não levar a nenhuma transição não especificada. Dois estados são compatíveis se as sequências de saída da FSM inicializada em cada um dos estados coincidirem sempre que ambas forem especificadas e para qualquer sequência de entrada aplicável. O seguinte teorema define em que condições dois estados são compatíveis, aplicando-se portanto a FSMs não completamente especificadas:

> Dois estados de uma FSM são compatíveis se e só se para qualquer entrada as correspondentes funções de saídas forem coincidentes quando ambas são especificadas e quando os estados seguintes correspondentes são compatíveis quando ambos também forem especificados.

Este teorema constitui a base dos procedimentos iterativos para determinação das classes de estados compatíveis, sendo correspondentes aos utilizados para determinar estados equivalentes em FSMs completamente especificadas. No entanto, relativamente ao caso de FSMs completamente especificadas existem duas diferenças fundamentais.

- Em primeiro lugar, uma relação de compatibilidade é diferente de uma relação de equivalência porque é simétrica, reflexiva mas não transitiva. Assim, uma classe de estados compatíveis é definida como aquela em que todos os estados são compatíveis dois a dois. As classes máximas de estados compatíveis não formam uma partição do conjunto dos estados, uma vez que se podem sobrepor. Isto faz com que possam existir múltiplas soluções para o problema. A intratabilidade do problema resulta deste facto.
- Em segundo lugar, a selecção de um número adequado de classes de compatibilidade que cubra o conjunto dos estados é complicada pelas implicações entre as próprias classes porque a compatibilidade de dois ou mais estados pode requerer que outros também sejam compatíveis. Assim, a selecção de uma classe de compatibilidade para ser representada por um só estado pode implicar que outra classe também tenha de ser escolhida. Um conjunto de classes de compatibilidade diz-se fechado, ou possui a propriedade do fecho, quando todas as classes de compatibilidade implicadas estão nesse conjunto ou estão contidas nas classes desse conjunto.

Consideremos a FSM não completamente especificada da Figura 5.9 (a).

Por uma questão de simplicidade, somente a função de saída não está completamente especificada. A respectiva tabela de transição de estados é a seguinte:

Entrada	Estado Actual	Estado Seguinte	Saída
0	S_1	S_3	1
1	S_1	S_5	*
0	S_2	S_3	*
1	S_2	S_5	1
0	S_3	S_2	0
1	S_3	S_1	1
0	S_4	S_4	0
1	S_4	S_5	1
0	S_5	S_4	1
1	S_5	S_1	0

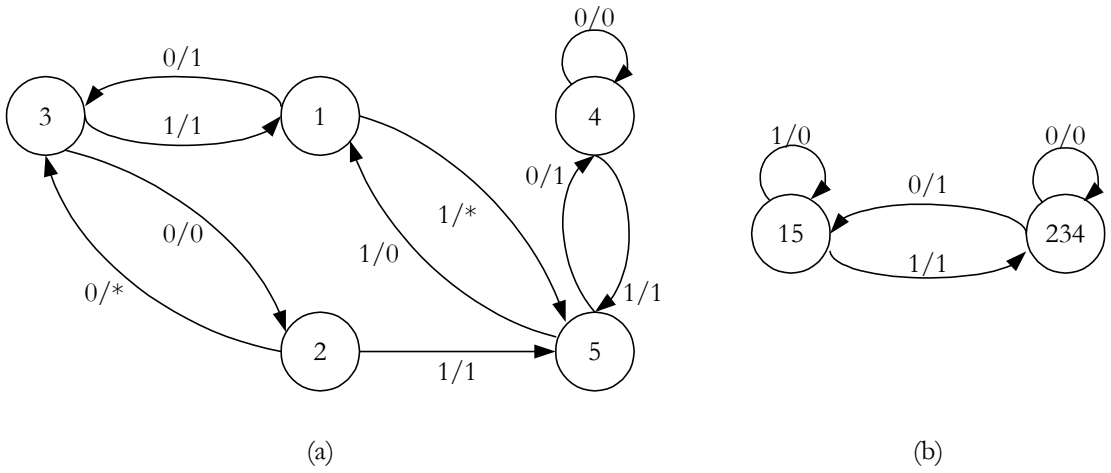


Figura 5.9 – Diagramas de transição de estados de uma FSM não completamente especificada (a) antes da minimização de estados; (b) após a minimização de estados.

De notar que se os valores *don't care* forem substituídos por "1s", obtém-se a tabela apresentada no exemplo utilizado na minimização de FSMs completamente especificadas e que pode ser minimizada para quatro estados. Outras escolhas para os valores *don't care* resultam noutras FSMs completamente especificadas. Infelizmente o número de FSMs completamente especificadas que resultam desta substituição cresce exponencialmente com o aumento do número de *don't cares*.

Consideremos então a compatibilidade entre os vários pares de estados:

- O par $\{s_1, s_2\}$ é compatível;
- A compatibilidade do par $\{s_2, s_3\}$ depende da compatibilidade do par $\{s_1, s_5\}$;
- O par $\{s_1, s_3\}$ não é compatível o que mostra a ausência de transitividade numa relação de compatibilidade.

Na tabela seguinte são listados os pares de estados compatíveis e incompatíveis.

	Pares	Pares Implicados
Compatível	$\{s_1, s_2\}$	
Compatível	$\{s_1, s_5\}$	$\{s_3, s_4\}$
Compatível	$\{s_2, s_4\}$	$\{s_3, s_4\}$
Compatível	$\{s_2, s_3\}$	$\{s_1, s_5\}$
Compatível	$\{s_3, s_4\}$	$\{s_2, s_4\}$, $\{s_1, s_5\}$
Incompatível	$\{s_1, s_3\}$	
Incompatível	$\{s_1, s_4\}$	
Incompatível	$\{s_2, s_5\}$	
Incompatível	$\{s_3, s_5\}$	
Incompatível	$\{s_4, s_5\}$	

As classes máximas de compatibilidade, isto é, compostas pelo maior número possível de estados compatíveis, são as seguintes:

Classes	Classes Implicadas
$\{s_1, s_2\}$	
$\{s_1, s_5\}$	$\{s_3, s_4\}$
$\{s_2, s_3, s_4\}$	$\{s_1, s_5\}$

A minimização do número de estados de uma FSM não completamente especificada consiste na selecção de classes de compatibilidade suficientes que satisfaçam a propriedade de fecho de forma a que os estados sejam cobertos. Assim, a minimização de estados pode ser formulada como um problema de *binate covering* e resolvido de forma exacta ou heurística pelos respectivos algoritmos [Micheli94]. Os estados de uma implementação mínima corresponderão às classes seleccionadas e o seu número à cardinalidade de uma cobertura mínima.

No exemplo apresentado existem três classes máximas de estados compatíveis. No entanto, uma cobertura fechada mínima envolve apenas $\{s_1, s_5\}$ e $\{s_2, s_3, s_4\}$, sendo a sua cardinalidade 2. O STD minimizado é mostrado na Figura 5.9 (b).

5.4 Codificação de Estados

A codificação de estados consiste na determinação da representação binária dos estados simbólicos de uma FSM. No caso mais geral, este problema pode consistir também na escolha dos flip-flops usados para o seu armazenamento (ex. D, T, JK). Aqui vão ser considerados apenas os flip-flops do tipo D, uma vez que são os mais utilizados e aqueles que normalmente existem nos blocos lógicos das arquitecturas de FPGAs. A codificação afecta a área do circuito e o seu desempenho. A maioria das técnicas conhecidas para codificação de estados têm como objectivo a redução das métricas de complexidade do circuito, as quais se relacionam directamente com a sua área mas não obrigatoriamente com o seu desempenho. A complexidade do circuito está relacionada com o número de bits de armazenamento n_b usados para representar os estados, isto é, com o comprimento de codificação e com o tamanho da

componente combinatória que implementa as funções de transição e de saída. As medidas da complexidade desta diferem consideravelmente quando se consideram implementações com lógica de dois níveis ou multi-nível. Por esta razão, as técnicas de codificação de estados têm sido desenvolvidas separadamente para cada um dos casos.

O trabalho inicial sobre codificação de estados focou-se na utilização de códigos de comprimento mínimo para representar o conjunto dos estados, isto é, $n_b = \lceil \log_2 |n_s| \rceil$, em que n_s é a cardinalidade do conjunto dos estados. Em muitos casos são utilizadas codificações arbitrárias (aleatórias) ou predefinidas como a binária ou a de *gray*. A vantagem destes tipos de codificação reside na simplicidade do processo de atribuição. No entanto, não existe nenhuma garantia de que a área ou os atrasos do circuito resultante sejam óptimos nem tão pouco uma aproximação razoável.

Nos casos das codificações de estado binária e de *gray*, os códigos são atribuídos por ordem crescente e pela sequência em que são enumerados ou surgem no STD. A diferença é que enquanto no primeiro caso é utilizado um contador binário no segundo é usado um contador de *gray*. A Figura 5.10 ilustra a atribuição dos códigos de estado binários à FSM que descreve o comportamento da unidade de controlo da máquina de venda automática, enquanto na Figura 5.11 é mostrada a codificação de *gray* para o mesmo exemplo. Como a FSM possui 8 estados são precisos pelo menos 3 bits para realizar a sua codificação. Na maior parte das situações a codificação de *gray* fornece melhores resultados devido à maior probabilidade de adjacência entre estados consecutivos, que como veremos na próxima secção é uma das heurísticas utilizadas para codificar os estados de FSMs destinadas a implementações com dois níveis de portas lógicas. Nas duas próximas secções são resumidos os princípios que regem a codificação de estados para FSMs destinadas a implementações de dois níveis e multi-nível, respectivamente. É dada uma maior atenção ao segundo caso devido ao carácter multi-nível das FPGAs.

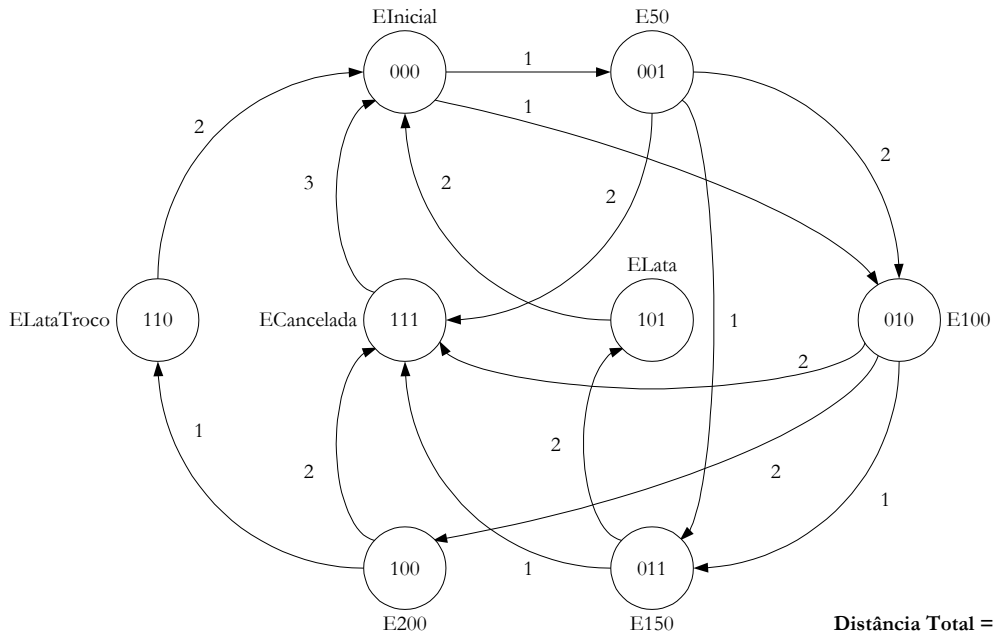


Figura 5.10 – Codificação de estados binária para a FSM da unidade de controlo da máquina de venda automática.

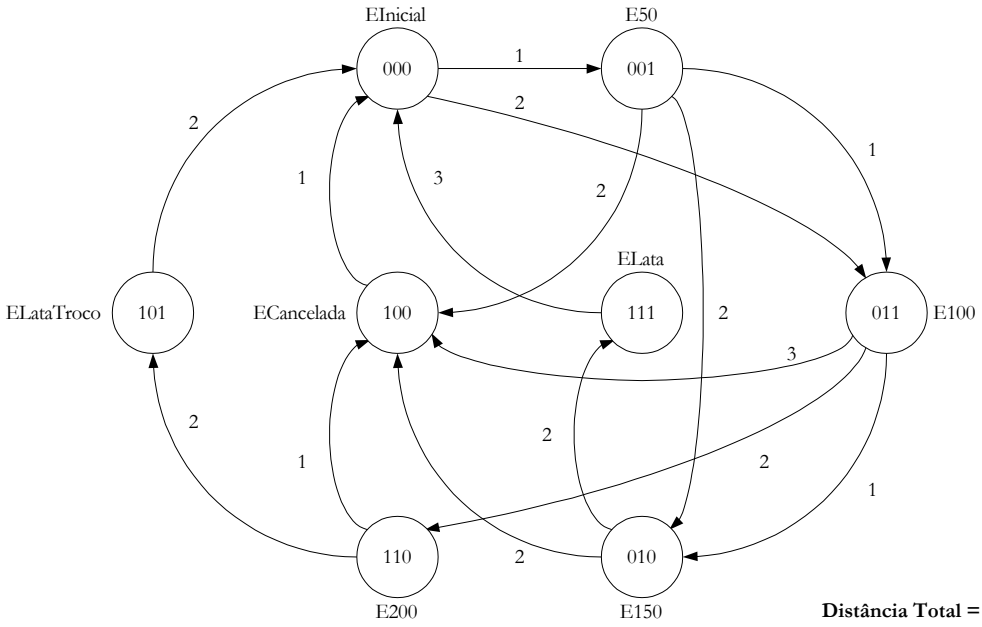


Figura 5.11 – Codificação de estados de *gray* para a FSM da unidade de controlo da máquina de venda automática.

5.4.1 Codificação de Estados para Circuitos de Dois Níveis

Os circuitos de dois níveis têm sido objecto de investigação intensa nas últimas décadas. A complexidade de um circuito representado na forma de somas de produtos está relacionada com o número de entradas, saídas e termos de produto. Para implementações baseadas em PLAs estes números podem ser usados para calcular a área do circuito e o comprimento físico do caminho mais longo, o qual está relacionado com o atraso do caminho crítico.

A escolha de uma codificação afecta o seu comprimento e o tamanho da componente combinatória do circuito. No total existem $2^{nb}!/(2^{nb}-n_s)!$ codificações possíveis, sendo a escolha da melhor uma tarefa bastante complexa e que geralmente não pode ser resolvida de forma exaustiva devido ao número total de codificações possíveis. De notar que o tamanho das representações na forma de soma de produtos é invariante relativamente à permutação e complemento dos bits de codificação, o que faz com que o número de códigos relevantes possa ser reduzido para $(2^{nb}-1)!/(2^{nb}-n_s)!n_b!$.

A maioria dos métodos heurísticos clássicos para codificação de estados são baseados em estratégias de minimização da distância de *Hamming* entre os códigos de estados consecutivos. Isto significa que o código atribuído a um dado estado deve ser tal que as respectivas variáveis sejam o mais próximas possível das que representam os estados anteriores. Este critério fundamenta-se numa relação directa que existe entre uma distância reduzida dos códigos de estados consecutivos e uma representação na forma de soma de produtos de tamanho mínimo. Idealmente, a distância de *Hamming* entre os códigos de estados consecutivos devia ser 1.

Como na maioria das FSMs é difícil fazer com que todos os pares de estados ligados por uma transição possuam códigos adjacentes, uma abordagem bastante

comum é minimizar a soma das distâncias entre os códigos desses estados. Este método passará a ser designado a partir de agora por Distância Total Mínima – DTM.

A Figura 5.12 ilustra a aplicação desta heurística à FSM da máquina de venda automática que tem sido usada como exemplo. A distância total entre os códigos dos estados consecutivos é 21. Esta distância é menor do que as obtidas nos casos de uma codificação binária e de *gray*, que foram 25 e 27, respectivamente (ver Figura 5.10 e seguinte). O impacto destas e de outras codificações de estado na área do circuito será analisada mais à frente. Existem métodos exactos e heurísticos mais recentes que podem ser utilizados para atribuir códigos aos estados simbólicos de uma FSM implementada na forma de soma de produtos. No entanto, estas técnicas não são aqui abordadas uma vez que neste capítulo se pretender realçar os métodos mais apropriados para circuitos multi-nível.

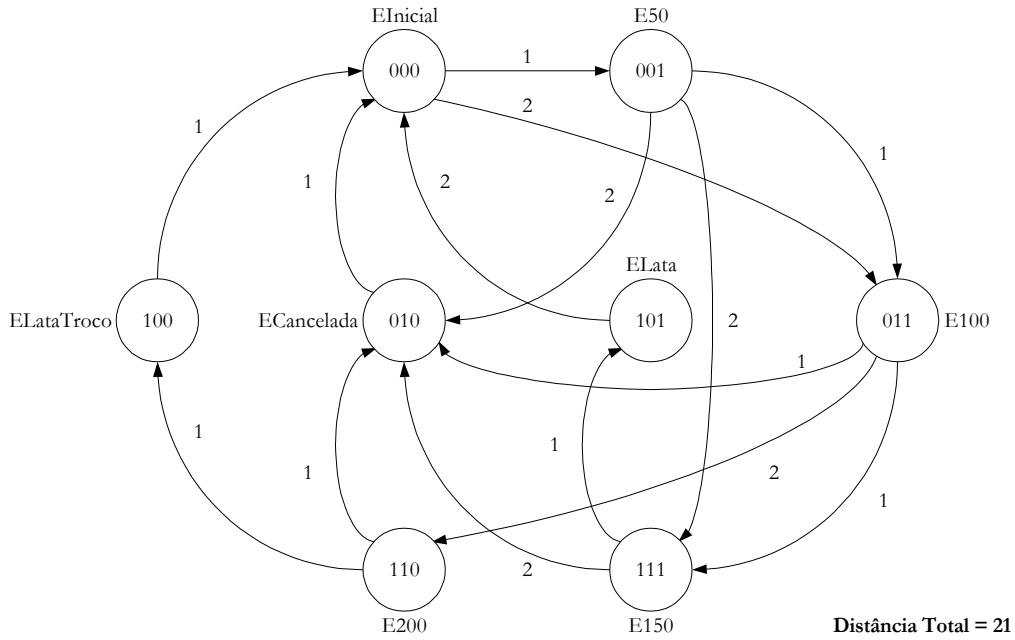


Figura 5.12 – Codificação de estados de DTM para a FSM da unidade de controlo da máquina de venda automática.

5.4.2 Codificação de Estados para Circuitos Multi-nível

As técnicas de codificação de estados para circuitos multi-nível utilizam para a componente combinatória do circuito sequencial o modelo de rede lógica [Micheli94]. A medida da área total do circuito está relacionada com o número de bits de codificação, ou seja, de flip-flops e com o número de literais da rede lógica. O atraso corresponde ao comprimento do caminho crítico da rede. Até à data foram desenvolvidos apenas métodos heurísticos para a determinação dos códigos dos estados de forma a optimizar as estimativas da área ocupada pelo circuito.

A abordagem mais simples baseia-se na determinação inicial de uma boa codificação de estados para um modelo lógico de dois níveis, seguida da reestruturação do circuito usando algoritmos de optimização combinatória. Apesar da escolha dos

códigos dos estados ser feita considerando um modelo diferente, os resultados experimentais têm-se revelado bastante satisfatórios. Este facto parece ser indicador de muitas FSMs possuírem funções de transição e de saída que podem ser implementadas eficientemente com poucos níveis de lógica.

A dificuldade da codificação de estados para modelos lógicos multi-nível resulta da grande variedade de transformações disponíveis para optimizar uma rede lógica e do problema em estimar o número de literais. Consequentemente, os investigadores têm considerado técnicas de codificação em conjunto com uma transformação lógica em particular. Nesta secção vai ser resumido um método cujo objectivo é a extracção de cubos comuns e que se baseia na determinação de um critério de proximidade dos códigos. Os detalhes deste método podem ser consultados em [AshDevNew92]. Por exemplo, quando dois ou mais estados possuem uma transição para o mesmo estado seguinte é conveniente fazer com que a distância entre os códigos correspondentes seja pequena uma vez que isso se relaciona com o tamanho do cubo comum que pode ser extraído.

Para ilustrar esta situação vamos considerar uma FSM com o conjunto de estados $S=\{s_1, s_2, s_3, s_4, s_5\}$. Vamos assumir que dois estados s_1 e s_2 possuem transições para o mesmo estado seguinte s_3 dependentes de $\bar{i}$ e i respectivamente. Vamos também assumir que os estados são codificados com três bits e que os estados s_1 e s_2 possuem códigos adjacentes (distância unitária) nomeadamente 000 e 001. Estes códigos correspondem aos cubos $\bar{a}.\bar{b}.\bar{c}$ e $\bar{a}.\bar{b}.c$ respectivamente, onde {a, b, c} são as variáveis de estado. A transição para o estado s_3 pode ser escrita como $\bar{i}.\bar{a}.\bar{b}.\bar{c} + i.\bar{a}.\bar{b}.c$, ou de forma simplificada $\bar{a}.\bar{b}.(\bar{i}.\bar{c} + i.c)$. De notar que se s_2 fosse codificado como 111, não poderia ser extraído nenhum cubo e no caso do código 011 o cubo extraído seria menor.

O problema da codificação é modelado por um grafo completo onde os vértices possuem uma correspondência de um para um com os estados e as arestas são etiquetadas com valores numéricos que representam pesos. Estes pesos significam a proximidade pretendida para os códigos dos respectivos pares de estados e são determinados por varrimento sistemático de todos os pares. A codificação de estados é determinada mergulhando este grafo num espaço Booleano de dimensões apropriadas. Uma vez que este processo é um problema intratável, são usados algoritmos heurísticos para determinar uma codificação onde a distância entre dois códigos está relacionada com o peso da aresta que une os respectivos estados (quanto maior for o peso, menor será a distância).

No entanto, existem ainda dois problemas por resolver. Em primeiro lugar, quando se consideram os códigos de dois estados com transições para o mesmo estado seguinte, a dimensão do cubo comum pode ser determinada pela distância dos códigos, mas o número de possíveis extracções de cubos depende da codificação dos estados seguintes. Consequentemente, o ganho em área não pode ser relacionado directamente com as transições e os pesos são apenas uma indicação imprecisa do possível ganho global que resulta da extracção dos cubos comuns. Em segundo lugar, as extracções de cubos comuns interage mutuamente.

Para fazer frente a estas dificuldades em [AshDevNew92] foram propostos dois algoritmos heurísticos. Ambos usam um grafo completo e etiquetado com pesos em

que a diferença reside na forma como estes são determinados. No primeiro algoritmo, designado por "orientado ao *fan-out*", aos pares de estados que possuam transições para o mesmo estado seguinte são atribuídos pesos elevados às respectivas arestas (para se obter códigos próximos). Os pesos são calculados por uma fórmula complexa que tem em consideração os padrões de saída. Esta abordagem procura maximizar o tamanho dos cubos comuns na função de estado seguinte codificada. No segundo algoritmo, chamado "orientado ao *fan-in*", aos pares de estados com transições dos mesmos estados anteriores são atribuídos pesos elevados. Mais uma vez os pesos são determinados por uma regra complexa que tem em consideração os padrões de entrada. Esta estratégia procura maximizar o número de cubos comuns na função de estado seguinte codificada. Para exemplificar o algoritmo "orientado ao *fan-out*" consideremos agora a seguinte tabela de transição de estados cujo STD correspondente está mostrado na Figura 5.13 (a):

Entrada	**Estado Actual**	**Estado Seguinte**	**Saída**
0	S_1	S_3	0
1	S_1	S_3	0
0	S_2	S_3	0
1	S_2	S_1	1
0	S_3	S_5	0
1	S_3	S_4	1
0	S_4	S_2	1
1	S_4	S_3	0
0	S_5	S_2	1
1	S_5	S_5	0

Em primeiro lugar deve ser construído um grafo completo de cinco vértices e de seguida determinados os pesos das arestas (Figura 5.13 (b)). Por uma questão de simplicidade vamos considerar apenas pesos binários, em que 1 representa o peso mais elevado. Vamos tomar como exemplo o par de estados $\{s_1, s_2\}$ em que ambos possuem uma transição para o estado s_3. Nesta circunstâncias deve ser atribuído um peso de 1 à aresta $\{v_1, v_2\}$. Através do varrimento de todos os pares de estados conclui-se que às arestas $\{\{v_1, v_2\}, \{v_2, v_4\}, \{v_1, v_4\}, \{v_4, v_5\}, \{v_3, v_5\}\}$ devem ser atribuídos pesos unitários e às restantes peso 0. A codificação representada pela seguinte matriz, na qual cada linha corresponde ao código de um estado, reflecte os requisitos de proximidade especificados pelos pesos:

$$E = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

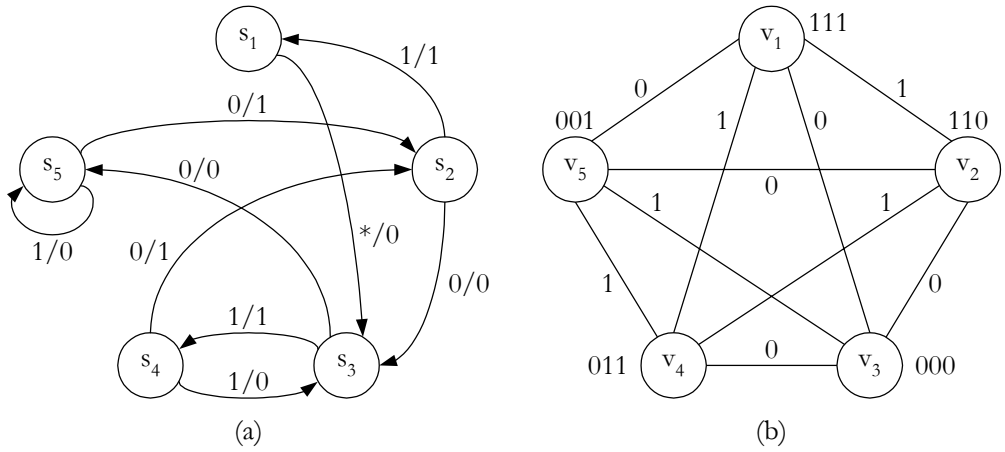


Figura 5.13 – Exemplo de diagrama de transição de estados (a) e respectivo grafo para atribuição de estados baseada no algoritmo orientado ao *fan-out* (b).

Substituindo os códigos na tabela e eliminando as linhas que não contribuem para as funções de transição e de saída codificadas obtém-se:

Entrada	**Estado Actual**	**Estado Seguinte**	**Saída**
1	110	111	1
0	000	001	0
1	000	011	1
0	011	110	1
0	001	110	1
1	001	001	0

Vamos assumir agora que *i* representa a variável de entrada; *a*, *b*, *c* as variáveis de estado e f_a, f_b, f_c as funções de transição codificadas. Consideremos a expressão $f_a = i.a.b.\bar{c} + \bar{i}.\bar{a}.b.c + \bar{i}.\bar{a}.\bar{b}.c$ que pode ser reescrita como $f_a = i.a.b.\bar{c} + \bar{i}.\bar{a}.c.(b + \bar{b}) = i.a.b.\bar{c} + \bar{i}.\bar{a}.c$. O cubo comum $\bar{i}.\bar{a}.c$ está relacionado com os códigos de {s_4, s_5} que são adjacentes. De notar que o cubo $\bar{i}.\bar{a}.c$ não está em conjunção com nenhuma expressão porque neste caso particular as entradas primárias coincidem para as transições de {s_4, s_5}.

Vamos agora assumir que ao estado s_5 é atribuído o código 101 correspondendo ao cubo $a.\bar{b}.c$. Neste caso $f_a = i.a.b.\bar{c} + \bar{i}.\bar{a}.b.c + \bar{i}.a.\bar{b}.c$ que pode ser reescrita como $f_a = i.a.b.\bar{c} + \bar{i}.c.(\bar{a}.b + a.\bar{b})$. Esta expressão é maior, isto é, possui mais literais do que a anterior devido à maior distância dos códigos de {s_4, s_5}.

Para detectar potenciais cubos comuns foram propostas outras regras, bem como fórmulas de avaliação mais precisas [DuHacLinNew91]. Todas têm em comum a análise dos pares de transições para o estado seguinte que podem ser representados simbolicamente como $i_1.s_1+i_2.s_2$, onde i_1, i_2 são vectores de entrada (cubos) e $\{s_1, s_2\} \in S$ é qualquer par de estados. Enquanto o caso geral foi descrito acima, casos mais específicos ocorrem quando $i_1 = i_2$ ou $i_1 \subseteq i_2$. Em qualquer dos casos é conveniente

conseguir uma distância unitária nos códigos de s_1 e s_2, uma vez que se consegue uma maior redução de literais.

Consideremos mais uma vez a primeira expressão de f_a do exemplo anterior. Uma vez que $i_1 = i_2 = \bar{i}$ e a distância de codificação de {s_4, s_5} é 1, o cubo comum $\bar{i}.\bar{a}.c.(b + \bar{b}) = \bar{i}.\bar{a}.c$ não está em conjunção com nenhuma expressão. Consideremos agora as transições dos estados {s_1, s_2} para o estado s_3 dependentes das entradas i_1 e i_2 respectivamente, em que $i_1 \subseteq i_2$ (por exemplo, i_1=i.j e i_2=i). Assumindo que aos estados {s_1, s_2} são atribuídos códigos adjacentes, tais como 000 ($\bar{a}.\bar{b}.\bar{c}$) e 001 ($\bar{a}.\bar{b}.c$) respectivamente, então a transição pode ser expressa como $i.j.\bar{a}.\bar{b}.\bar{c} + i.\bar{a}.\bar{b}.c$ e simplificada para $i.\bar{a}.\bar{b}.(j.\bar{c} + c) = i.\bar{a}.\bar{b}.(j + c)$.

É importante notar que a extracção de cubos comuns reduz o número de literais, mas também existem outras transformações que permitem alcançar o mesmo objectivo, pelo que esta técnica de codificação explora apenas um método para reduzir a complexidade de uma rede multi-nível. Por este motivo, a estimativa do tamanho de uma rede multi-nível óptima considerando apenas a extracção de cubos, pode não ser muito precisa. Os resultados experimentais têm mostrado que esta técnica é adequada, principalmente devido ao facto das funções de transição e de saída serem relativamente planas não necessitando portanto de muitos níveis de lógica.

Para concluir esta secção, a Figura 5.14 ilustra a atribuição dos estados da FSM que descreve a unidade de controlo da máquina de venda automática usando uma combinação dos algoritmos orientados ao *fan-in* e ao *fan-out* e que será a partir deste momento designada por Distância Mínima entre os Estados de Entrada e de Saída - DMEES. As bandas a cheio e as linhas representam, respectivamente, os estados cujos códigos devem ser mantidos próximos segundo os algoritmos orientados ao *fan-out* e ao *fan-in*. A comparação da área do circuito para esta codificação de estado e as apresentadas acima será feita na secção 5.7.

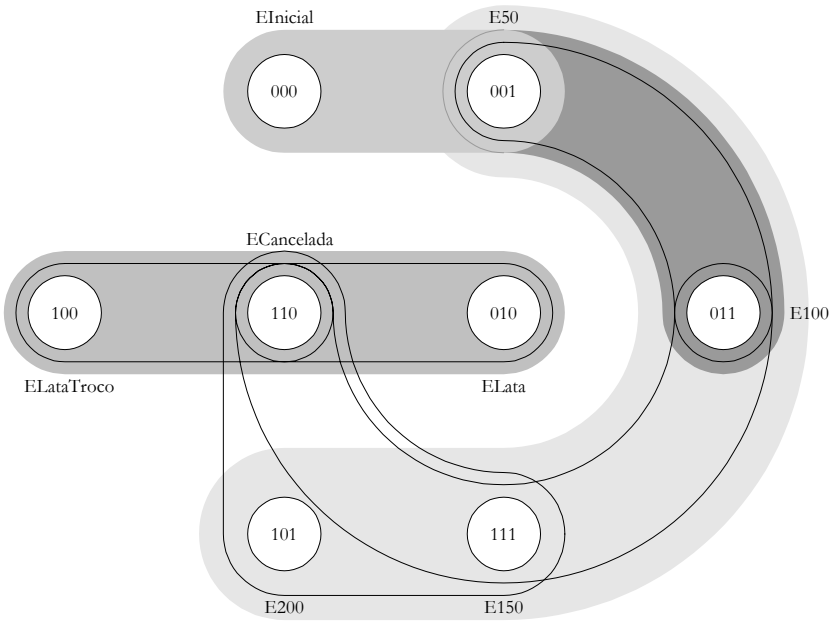


Figura 5.14 – Codificação de estados DMEES para a FSM da máquina de venda automática.

5.4.3 Codificação de Estados One-Hot

Se o número de estados de uma unidade de controlo não for elevado e a tecnologia de implementação disponibilizar um número razoável de flip-flops, a implementação pode ser feita usando um flip-flop por estado. Este método de codificação resulta numa correspondência directa entre o STD ou o GS e o respectivo circuito. Em cada instante só existe um flip-flop no estado "1", estando todos os outros a "0", pelo que esta implementação é normalmente conhecida por *one-hot*. A inicialização do sistema consiste no carregamento de um "1" no flip-flop que representa o estado inicial e de um "0" nos restantes. Com a ajuda da Figura 5.15 vamos agora analisar a correspondência entre os nodos de um STD e de um GS e os elementos lógicos que os implementam.

O caso mais simples é quando um estado possui apenas um predecessor e um sucessor (Figura 5.15 (a)) sendo implementado por apenas um flip-flop. No caso mais geral, um estado possui vários predecessores e vários sucessores (Figura 5.15 (b)). A transição para os últimos é determinada pelos valores dos sinais de entrada ou condições lógicas. Na Figura 5.15 (c) é mostrada a implementação de um nodo condicional de um GS, o qual é um caso particular da situação anterior. Na Figura 5.15 (d) é mostrada a forma como deve ser calculada uma saída, usando uma porta lógica OR, no caso de ser activada em mais do que um estado. Finalmente, na Figura 5.15 (e) estão indicados os pontos do circuito de onde devem ser derivadas as saídas no caso das máquinas de Moore e de Mealy.

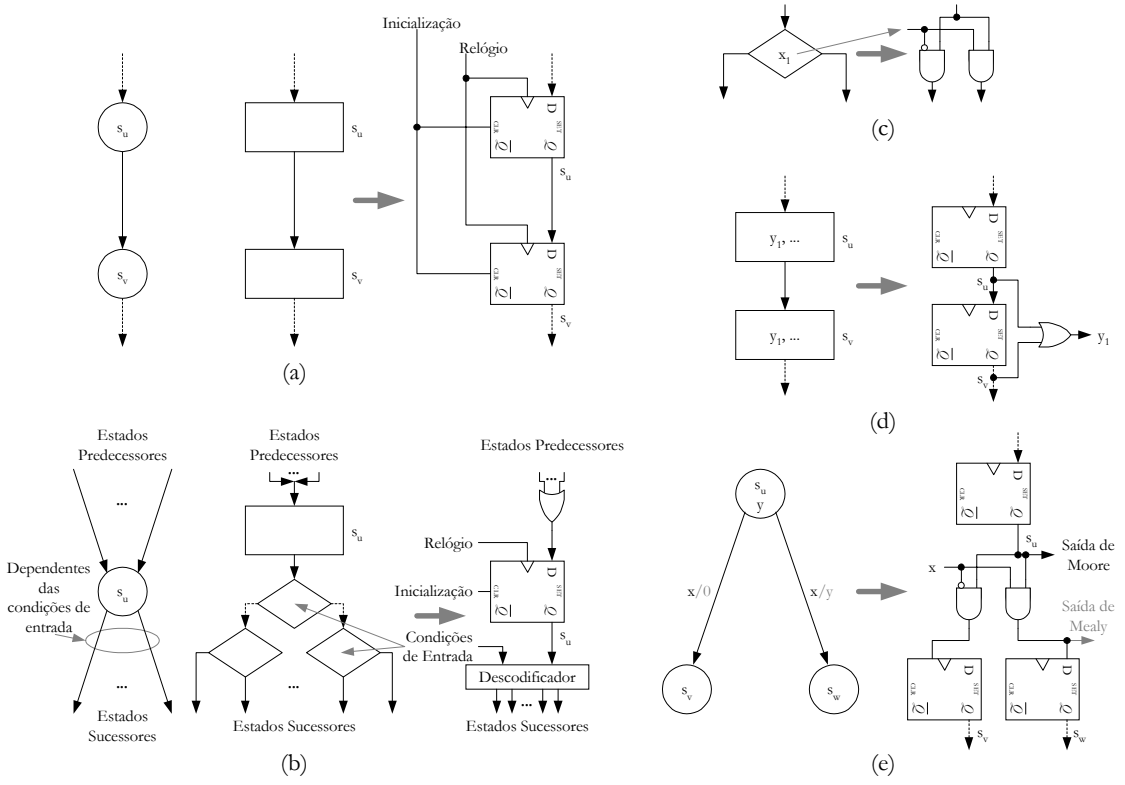


Figura 5.15 – Regras para conversão de uma descrição comportamental baseada em STDs ou GSs num circuito usando codificação de estados *one-hot*.

Para *n* estados são necessários n flip-flops, em vez de $\lceil \log_2 n \rceil$ como no caso de uma codificação de estados de comprimento mínimo. Este factor pode impedir a utilização desta técnica em circuitos com um elevado número de estados.

As vantagens deste método de codificação são o reduzido tempo de projecto e o facto de necessitar de portas lógicas com um menor número de entradas do que a implementação binária correspondente. Esta técnica é adequada para a implementação de unidades de controlo em FPGAs da família XC6200, devido à granulosidade fina destes dispositivos e à redução do número médio de entradas por porta lógica.

Na Figura 5.17 é apresentado o esquema da unidade de controlo baseada em codificação de estados *one-hot* da máquina de venda automática descrita no capítulo anterior e cujo GS é novamente apresentado na Figura 5.16.

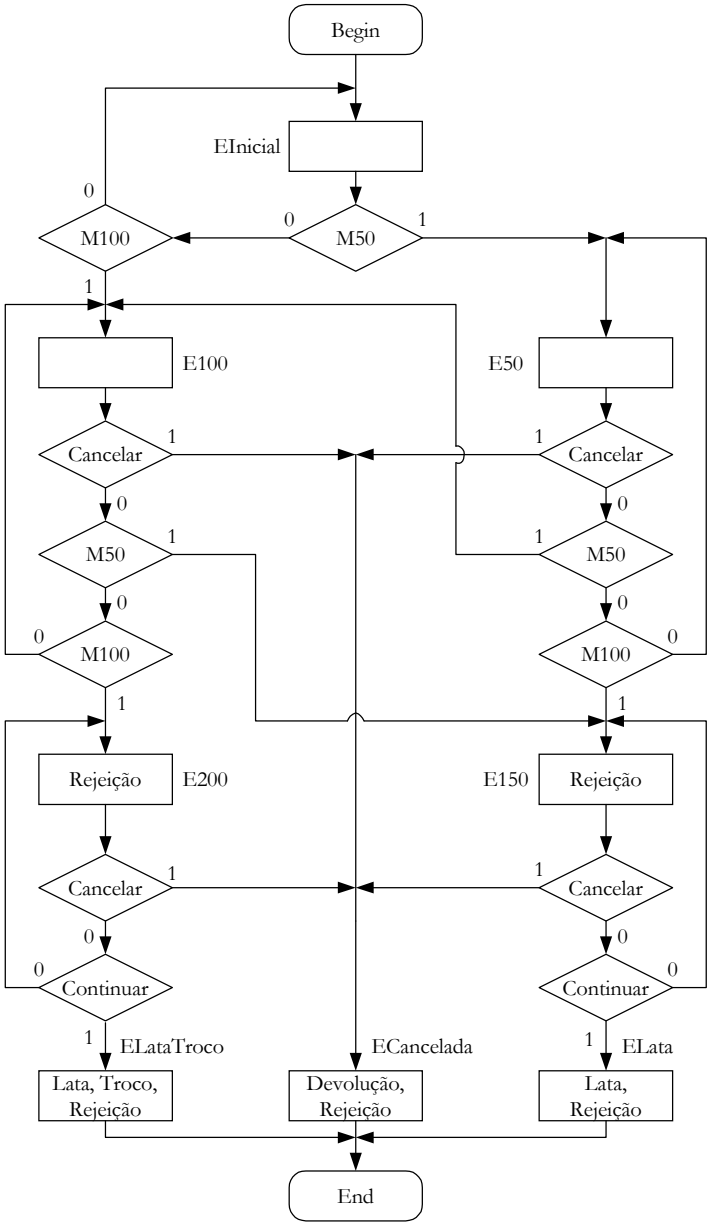


Figura 5.16 – Esquema de grafos que descreve o comportamento da unidade de controlo da máquina de venda automática apresentada no capítulo anterior.

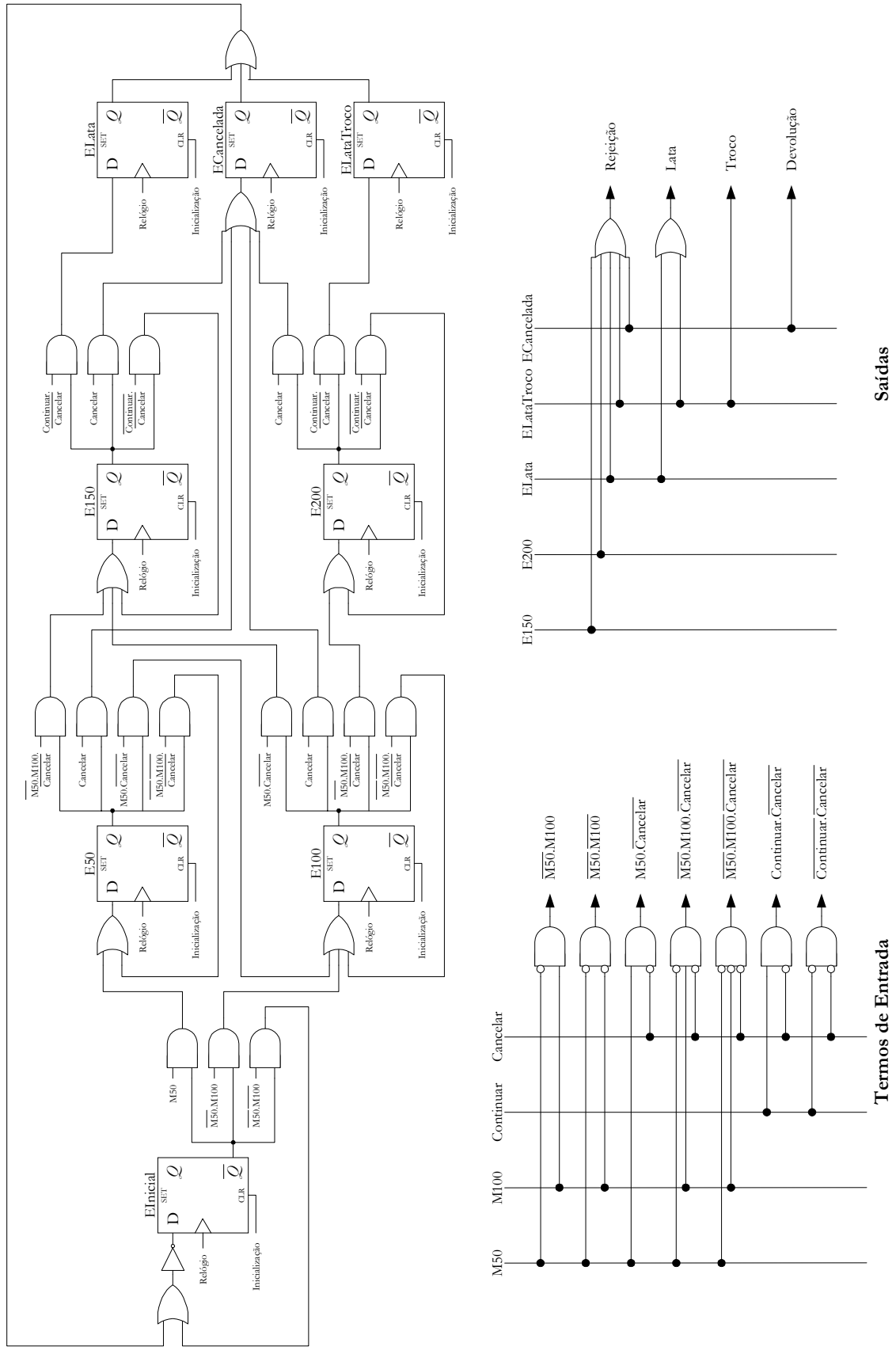


Figura 5.17 – Esquema da unidade de controlo da máquina de venda automática baseada em codificação de estados *one-hot*.

5.5 Optimização Lógica

A optimização lógica é realizada após a codificação de estados e tem como objectivo melhorar uma ou várias características do circuito que está a ser projectado. Alguns dos objectivos mais importantes são:

- Reduzir a área ocupada pelo circuito;
- Diminuir o comprimento do caminho crítico o qual está relacionado com a máxima frequência de funcionamento;
- Melhorar a testabilidade do circuito de forma a aumentar o número de falhas detectáveis com um dado conjunto de vectores de teste.

A optimização de circuitos lógicos tem sido uma área de investigação bastante activa nas últimas décadas. As técnicas utilizadas são bastante variadas e por vezes complexas, pelo que neste pequeno resumo será feita apenas uma breve descrição dos princípios subjacentes e algumas referências aos métodos e ferramentas mais utilizadas.

As técnicas empregues dependem se estamos a considerar uma implementação com dois níveis de lógica, isto é, na forma de soma de produtos ou produto de somas, ou por outro lado, uma implementação multi-nível constituída por um determinado conjunto de funções lógicas combinadas de forma arbitrária. Numa implementação com dois níveis de lógica, por exemplo na forma de soma de produtos, procura-se minimizar o número de produtos, diminuir o número de literais de cada produto e no caso de circuitos com múltiplas saídas maximizar a partilha dos produtos pelas várias funções. Por outro lado, numa implementação multi-nível pretende-se minimizar o número de literais das equações lógicas e maximizar o número e o tamanho de subexpressões comuns. Devido aos diferentes objectivos que regem a optimização de circuitos destinados a implementações de dois níveis e multi-nível os algoritmos e as ferramentas aplicáveis a cada um dos casos têm sido desenvolvidos separadamente.

Enquanto numa implementação de dois níveis, a optimização lógica é também sinónimo de minimização, numa implementação multi-nível tal não é verdade, uma vez que a partilha de subexpressões apesar de reduzir a área do circuito pode em certos casos aumentar o número de níveis do circuito o que faz com que os seus atrasos sejam maiores. Assim, no caso de circuitos multi-nível a optimização baseia-se muitas vezes num compromisso entre a área e o comprimento do caminho crítico.

Os algoritmos utilizados em ambos os casos podem ser exactos ou heurísticos. Enquanto os primeiros garantem uma solução óptima para o problema em termos do parâmetro que se pretende optimizar (área, atrasos, testabilidade, etc.), os segundos fornecem uma solução que é na maior parte das situações uma boa aproximação à solução óptima, podendo em certos casos até coincidir. A vantagem dos algoritmos exactos é a qualidade dos resultados obtidos. Contudo, nem sempre são utilizados devido aos elevados recursos ou tempo de processamento de que necessitam. Por outro lado, os algoritmos heurísticos apesar de não produzirem uma solução óptima têm a vantagem de necessitarem tipicamente de menos recursos e serem mais rápidos que os anteriores.

A optimização de um circuito combinatório pode ser realizada manualmente ou com o auxílio de ferramentas apropriadas. A primeira abordagem está restringida a

problemas de pequena complexidade, isto é, limitados a poucas variáveis, devido à incapacidade do ser humano em trabalhar eficientemente com grandes volumes de informação ou ao elevado tempo necessário até à obtenção de resultados. Por outro lado, as ferramentas de optimização lógica possuem capacidade para lidar com problemas de complexidade bastante variável. De todas as ferramentas disponíveis vamos destacar duas: o *Espresso* [BraHacMcMSan84] e o *misII* [BraRudSanWan87]. Ambas foram desenvolvidas em *Berkeley* como ferramentas de investigação e vão ser utilizadas na secção 5.7 para optimizar a parte combinatória da unidade de controlo da máquina de venda automática, permitindo assim com um esforço reduzido avaliar o impacto de várias codificações de estado na área ocupada pelo circuito. O *Espresso* destina-se a realizar a optimização de circuitos com dois níveis de lógica na forma de soma de produtos. O *misII* é uma ferramenta interactiva para optimizar circuitos multi-nível. A optimização manual de circuitos com dois níveis de lógica é normalmente feita com mapas de *Karnaugh*, pelo método de *Quine-McCluskey* ou variantes deste.

Os circuitos combinatórios são muito frequentemente implementados com redes multi-nível de portas lógicas. A granulosidade fina das redes multi-nível proporciona vários graus de liberdade no projecto de circuitos digitais que podem ser explorados para optimizar a área ou os atrasos bem como satisfazer restrições especificas, tais como diferentes requisitos temporais em diferentes caminhos entre as entradas e as saídas. Por este motivo as redes multi-nível são normalmente preferidas às implementações de dois níveis. Uma rede multi-nível é optimizada através de várias transformações, tais como a eliminação, a decomposição, a extracção, a simplificação e a substituição [Micheli94]. Infelizmente, a desvantagem da flexibilidade da implementação de funções combinatórias como redes multi-nível é a dificuldade de modelação e optimização das próprias redes. Até agora foram propostos poucos métodos exactos, possuindo todos eles uma complexidade computacional elevada. Devido a este facto, os métodos exactos não são considerados práticos hoje em dia. As técnicas para optimização de circuitos multi-nível baseiam-se normalmente em diferentes modelos, dos quais se destacam:

- Os modelos algébricos em que as funções Booleanas são representadas por expressões algébricas na forma de polinómios multi-lineares de coeficientes unitários. Estes são manipulados e optimizados de acordo com as regras da álgebra polinomial e ignorando características específicas da álgebra Booleana. Assim, à custa de uma diminuição da qualidade dos resultados, consegue-se simplificar e acelerar a procura de subexpressões comuns;
- Os modelos Booleanos que ao contrário do anterior exploram todas as potencialidades da álgebra de *Boole*. Cada nodo da rede lógica está associado a uma função Booleana e a um conjunto local de *don't cares*. A transformação fundamental é a simplificação Booleana de uma ou mais funções locais.

Na síntese de unidades de controlo a optimização lógica desempenha um papel muito importante, principalmente no caso de ser utilizada uma codificação de estados de comprimento mínimo. Na codificação de estados *one-hot* a optimização é trivial

devido à descodificação completa dos estados e à diminuição do número de variáveis envolvidas em cada transição.

5.6 Síntese Manual para Circuitos de Dois Níveis

Nesta secção vão ser ilustrados os procedimentos típicos da síntese manual de uma unidade de controlo implementada com lógica de dois níveis. Isto significa que as variáveis de entrada dos flip-flops que armazenam o estado e os sinais de saída do circuito são determinados com funções lógicas descritas, optimizadas e implementadas na forma de soma de produtos. Para o efeito, vai ser utilizado o exemplo da unidade de controlo da máquina de venda automática apresentado no capítulo anterior. De notar que a síntese manual só é viável em termos de tempo de projecto e produz resultados aceitáveis do ponto de vista de optimização do circuito quando a dimensão do problema não é muito elevada, isto é, quando o número de estados, transições, entradas e saídas é relativamente reduzido. O parâmetro mais crítico é o número de entradas, uma vez que por cada entrada adicional a dimensão do problema duplica.

A dimensão do problema adoptado encontra-se na fronteira da complexidade entre os circuitos que podem ser facilmente sintetizados manualmente e aqueles em que o seu processo de síntese só pode ser concluído em tempo útil recorrendo a ferramentas de projecto assistido por computador. Mesmo assim, para facilitar a optimização da área do circuito recorrendo aos mapas de *Karnaugh* é aplicada a técnica de redução do número de entradas por multiplexagem descrito em 5.2. A aplicação desta técnica é possível porque as entradas "M50" e "Continuar" não são necessárias simultaneamente (a descrição de cada uma delas encontra-se no capítulo anterior). Assim, à custa de uma variável de saída adicional "y_1" é possível realizar a multiplexagem das entradas. A variável de saída deste multiplexador (que é uma variável de entrada do núcleo da unidade de controlo) passará a ser designada a partir de agora por "x_1". Para variável de selecção do multiplexador pode ser utilizada a saída "Rejeição", uma vez que é a partir do momento que esta é activada que somente as variáveis de entrada "Continuar" e "Cancelar" são relevantes para o funcionamento da unidade de controlo.

Uma implementação multi-nível pode ser obtida a partir de uma de dois níveis através da aplicação de técnicas de minimização lógica ao circuito resultante. Contudo, isto não será feito aqui uma vez que mais à frente serão utilizadas ferramentas de síntese e optimização para este fim.

Uma vez que a codificação das entradas e saídas já está definida e a especificação apresentada no capítulo anterior já se encontra minimizada em termos do número de estados, a próxima fase importante da síntese da unidade de controlo é a codificação de estados. Neste caso vai ser utilizada uma codificação de comprimento mínimo baseada na heurística DTM. Como a FSM possui oito estados são necessários pelo menos três para realizar a sua codificação.

5.6.1 Codificação de Estados

Para realizar a codificação de estados vai ser utilizada a heurística da distância total mínima atrás descrita. A Tabela 5.1 resulta da substituição dos estados simbólicos

da tabela de transição de estados apresentada no capítulo anterior por códigos binários atribuídos com base no método de codificação adoptado. A transformação das variáveis de entrada resultante da redução atrás descrita é já visível nesta tabela.

Estado Actual	**Saídas** Lata Troco Devolução Rejeição/Y_1	**Entradas** X_1 M100 Cancelar	**Estado Seguinte**
000	0 0 0 0	1 - -	001
		0 1 -	011
		0 0 -	000
001	0 0 0 0	- - 1	010
		1 - 0	011
		0 1 0	111
		0 0 0	001
011	0 0 0 0	- - 1	010
		1 - 0	111
		0 1 0	110
		0 0 0	011
111	0 0 0 1	- - 1	010
		1 - 0	101
		0 - 0	111
110	0 0 0 1	- - 1	010
		1 - 0	100
		0 - 0	110
101	1 0 0 1	- - -	000
100	1 1 0 1	- - -	000
010	0 0 1 1	- - -	000

Tabela 5.1 – Tabela de transição com os estados codificados que descreve o comportamento da unidade de controlo da máquina de venda automática.

5.6.2 Equações de Estado Seguinte e das Saídas

A partir da Tabela 5.1 pode ser construída a Tabela 5.2 que ilustra para cada par (estado actual codificado, entradas) os valores das variáveis de estado no próximo ciclo de relógio. Da análise da Tabela 5.2 resultam as seguintes equações de estado seguinte para cada uma das variáveis de estado (Q_1, Q_2, Q_3).

$$
\begin{aligned}
Q_1^+ = {} & \overline{Q_3}\cdot\overline{Q_2}\cdot\overline{Q_1}\cdot x_1 + \overline{Q_3}\cdot\overline{Q_2}\cdot\overline{Q_1}\cdot\overline{x_1}\cdot \mathrm{M100} + \\
& + \overline{Q_3}\cdot\overline{Q_2}\cdot Q_1\cdot x_1\cdot\overline{\mathrm{Cancelar}} + \overline{Q_3}\cdot\overline{Q_2}\cdot Q_1\cdot\overline{x_1}\cdot \mathrm{M100}\cdot\overline{\mathrm{Cancelar}} + \overline{Q_3}\cdot\overline{Q_2}\cdot Q_1\cdot\overline{x_1}\cdot\overline{\mathrm{M100}}\cdot\overline{\mathrm{Cancelar}} + \\
& + \overline{Q_3}\cdot Q_2\cdot Q_1\cdot x_1\cdot\overline{\mathrm{Cancelar}} + \overline{Q_3}\cdot Q_2\cdot Q_1\cdot\overline{x_1}\cdot\overline{\mathrm{M100}}\cdot\overline{\mathrm{Cancelar}} + \\
& + Q_3\cdot Q_2\cdot Q_1\cdot x_1\cdot\overline{\mathrm{Cancelar}} + Q_3\cdot Q_2\cdot Q_1\cdot\overline{x_1}\cdot\overline{\mathrm{Cancelar}}
\end{aligned}
$$

$$Q_2^+ = \overline{Q_3}\cdot\overline{Q_2}\cdot\overline{Q_1}\cdot\overline{x_1}\cdot M100 + \overline{Q_3}\cdot\overline{Q_2}\cdot Q_1\cdot Cancelar +$$
$$+\overline{Q_3}\cdot\overline{Q_2}\cdot Q_1\cdot x_1\cdot\overline{Cancelar} + \overline{Q_3}\cdot Q_2\cdot\overline{Q_1}\cdot\overline{x_1}\cdot M100\cdot\overline{Cancelar} + Q_3\cdot\overline{Q_2}\cdot\overline{Q_1}\cdot\overline{x_1}\cdot\overline{M100}\cdot\overline{Cancelar} +$$
$$+\overline{Q_3}\cdot Q_2\cdot Q_1\cdot x_1\cdot\overline{Cancelar} + \overline{Q_3}\cdot Q_2\cdot Q_1\cdot\overline{x_1}\cdot M100\cdot\overline{Cancelar} + Q_3\cdot Q_2\cdot\overline{Q_1}\cdot\overline{x_1}\cdot\overline{M100}\cdot\overline{Cancelar} +$$
$$+Q_3\cdot Q_2\cdot Q_1\cdot Cancelar + Q_3\cdot Q_2\cdot Q_1\cdot\overline{x_1}\cdot\overline{Cancelar} +$$
$$+Q_3\cdot Q_2\cdot\overline{Q_1}\cdot Cancelar + Q_3\cdot Q_2\cdot\overline{Q_1}\cdot\overline{x_1}\cdot\overline{Cancelar}$$

$$Q_3^+ = \overline{Q_3}\cdot\overline{Q_2}\cdot Q_1\cdot\overline{x_1}\cdot M100\cdot\overline{Cancelar} +$$
$$+\overline{Q_3}\cdot Q_2\cdot Q_1\cdot x_1\cdot\overline{Cancelar} + \overline{Q_3}\cdot Q_2\cdot Q_1\cdot\overline{x_1}\cdot M100\cdot\overline{Cancelar} +$$
$$+Q_3\cdot Q_2\cdot Q_1\cdot x_1\cdot\overline{Cancelar} + Q_3\cdot Q_2\cdot Q_1\cdot\overline{x_1}\cdot\overline{Cancelar} +$$
$$+Q_3\cdot Q_2\cdot\overline{Q_1}\cdot x_1\cdot\overline{Cancelar} + Q_3\cdot Q_2\cdot\overline{Q_1}\cdot\overline{x_1}\cdot\overline{Cancelar}$$

Estado Actual $Q_3\ Q_2\ Q_1$	**Entradas**	**Estado Seguinte $Q_3^+\ Q_2^+\ Q_1^+$**
000	x_1	001
	$\overline{x_1}.M100$	011
	$\overline{x_1}.\overline{M100}$	000
001	Cancelar	010
	$x_1\cdot\overline{Cancelar}$	011
	$\overline{x_1}\cdot M100\cdot\overline{Cancelar}$	111
	$\overline{x_1}\cdot\overline{M100}\cdot\overline{Cancelar}$	001
011	Cancelar	010
	$x_1\cdot\overline{Cancelar}$	111
	$\overline{x_1}\cdot M100\cdot\overline{Cancelar}$	110
	$\overline{x_1}\cdot\overline{M100}\cdot\overline{Cancelar}$	011
111	Cancelar	010
	$x_1\cdot\overline{Cancelar}$	101
	$\overline{x_1}\cdot\overline{Cancelar}$	111
110	Cancelar	010
	$x_1\cdot\overline{Cancelar}$	100
	$\overline{x_1}\cdot\overline{Cancelar}$	110
101	1	000
100	1	000
010	1	000

Tabela 5.2 – Tabela de transição com os estados codificados.

A partir da Tabela 5.1 pode também ser extraída a Tabela 5.3 que representa as saídas que são activadas em cada estado da unidade de controlo. As equações das saídas que resultam da análise da Tabela 5.3 são as seguintes:

$$Lata = Q_3\cdot\overline{Q_2}\cdot Q_1 + Q_3\cdot\overline{Q_2}\cdot\overline{Q_1}$$

$$\text{Troco} = Q_3 \cdot \overline{Q_2} \cdot \overline{Q_1}$$

$$\text{Devolução} = \overline{Q_3} \cdot Q_2 \cdot \overline{Q_1}$$

$$\text{Rejeição} = y_1 = Q_3 \cdot Q_2 \cdot Q_1 + Q_3 \cdot Q_2 \cdot \overline{Q_1} + Q_3 \cdot \overline{Q_2} \cdot Q_1 + Q_3 \cdot \overline{Q_2} \cdot \overline{Q_1} + \overline{Q_3} \cdot Q_2 \cdot \overline{Q_1}$$

Estado Actual Q_3 Q_2 Q_1	**Saídas**
000	0
001	0
011	0
111	Rejeição, y_1
110	Rejeição, y_1
101	Lata, Rejeição, y_1
100	Lata, Troco, Rejeição, y_1
010	Devolução, Rejeição, y_1

Tabela 5.3 – Tabela de saídas.

5.6.3 Escolha dos Elementos de Memória e Equações de Excitação dos Flip-flops

Os flip-flops utilizados na implementação de circuitos sequenciais são normalmente de um dos tipos D, SR, JK ou T. A utilização de flip-flops do tipo JK resulta normalmente em circuitos mais simples devido à introdução de *don't cares* nas respectivas equações de excitação. Contudo, como possuem mais ligações que os flip-flops tipo D, raramente são utilizados em circuitos VLSI. Nas FPGAs este aspecto é ainda mais importante, uma vez que a quantidade de recursos de interligação é um dos aspectos críticos, razão pela qual a generalidade das arquitecturas de FPGA possui flip-flops do tipo D nas suas células ou blocos lógicos.

Como num flip-flop tipo D, $Q^+ = D$, a passagem da Tabela 5.2 para a Tabela 5.4 onde estão representados os valores de excitação de cada flip-flop para cada par (estado actual codificado, entradas) é imediata.

A equações de excitação dos três flip-flops usados para armazenar o estado são as seguintes:

$$\begin{aligned}
D_1 = {} & \overline{Q_3} \cdot \overline{Q_2} \cdot \overline{Q_1} \cdot x_1 + \overline{Q_3} \cdot \overline{Q_2} \cdot \overline{Q_1} \cdot \overline{x_1} \cdot M100 + \\
& + \overline{Q_3} \cdot \overline{Q_2} \cdot Q_1 \cdot x_1 \cdot \overline{\text{Cancelar}} + \overline{Q_3} \cdot \overline{Q_2} \cdot Q_1 \cdot \overline{x_1} \cdot M100 \cdot \overline{\text{Cancelar}} + \overline{Q_3} \cdot \overline{Q_2} \cdot Q_1 \cdot \overline{x_1} \cdot \overline{M100} \cdot \overline{\text{Cancelar}} + \\
& + \overline{Q_3} \cdot Q_2 \cdot Q_1 \cdot x_1 \cdot \overline{\text{Cancelar}} + \overline{Q_3} \cdot Q_2 \cdot Q_1 \cdot \overline{x_1} \cdot \overline{M100} \cdot \overline{\text{Cancelar}} + \\
& + Q_3 \cdot Q_2 \cdot Q_1 \cdot x_1 \cdot \overline{\text{Cancelar}} + Q_3 \cdot Q_2 \cdot Q_1 \cdot \overline{x_1} \cdot \overline{\text{Cancelar}}
\end{aligned}$$

$$D_2 = \overline{Q_3}\cdot\overline{Q_2}\cdot\overline{Q_1}\cdot\overline{x_1}\cdot M100 + \overline{Q_3}\cdot\overline{Q_2}\cdot Q_1\cdot Cancelar +$$
$$+\overline{Q_3}\cdot\overline{Q_2}\cdot Q_1\cdot x_1\cdot\overline{Cancelar} + \overline{Q_3}\cdot\overline{Q_2}\cdot Q_1\cdot\overline{x_1}\cdot M100\cdot\overline{Cancelar} + \overline{Q_3}\cdot Q_2\cdot Q_1\cdot Cancelar +$$
$$+\overline{Q_3}\cdot Q_2\cdot Q_1\cdot x_1\cdot\overline{Cancelar} + \overline{Q_3}\cdot Q_2\cdot Q_1\cdot\overline{x_1}\cdot M100\cdot\overline{Cancelar} + \overline{Q_3}\cdot Q_2\cdot Q_1\cdot\overline{x_1}\cdot\overline{M100}\cdot\overline{Cancelar} +$$
$$+Q_3\cdot Q_2\cdot Q_1\cdot Cancelar + Q_3\cdot Q_2\cdot Q_1\cdot\overline{x_1}\cdot\overline{Cancelar} +$$
$$+Q_3\cdot Q_2\cdot\overline{Q_1}\cdot Cancelar + Q_3\cdot Q_2\cdot\overline{Q_1}\cdot\overline{x_1}\cdot\overline{Cancelar}$$

$$D_3 = \overline{Q_3}\cdot\overline{Q_2}\cdot Q_1\cdot\overline{x_1}\cdot M100\cdot\overline{Cancelar} +$$
$$+\overline{Q_3}\cdot Q_2\cdot Q_1\cdot x_1\cdot\overline{Cancelar} + \overline{Q_3}\cdot Q_2\cdot Q_1\cdot\overline{x_1}\cdot M100\cdot\overline{Cancelar} +$$
$$+Q_3\cdot Q_2\cdot Q_1\cdot x_1\cdot\overline{Cancelar} + Q_3\cdot Q_2\cdot Q_1\cdot\overline{x_1}\cdot\overline{Cancelar} +$$
$$+Q_3\cdot Q_2\cdot\overline{Q_1}\cdot x_1\cdot\overline{Cancelar} + Q_3\cdot Q_2\cdot\overline{Q_1}\cdot\overline{x_1}\cdot\overline{Cancelar}$$

Estado Actual $Q_3\ Q_2\ Q_1$	Entradas	Estado Seguinte $D_3\ D_2\ D_1$
000	x_1	001
	$\overline{x_1}.M100$	011
	$\overline{x_1}.\overline{M100}$	000
001	Cancelar	010
	$x_1\cdot\overline{Cancelar}$	011
	$\overline{x_1}\cdot M100\cdot\overline{Cancelar}$	111
	$\overline{x_1}\cdot\overline{M100}\cdot\overline{Cancelar}$	001
011	Cancelar	010
	$x_1\cdot\overline{Cancelar}$	111
	$\overline{x_1}\cdot M100\cdot\overline{Cancelar}$	110
	$\overline{x_1}\cdot\overline{M100}\cdot\overline{Cancelar}$	011
111	Cancelar	010
	$x_1\cdot\overline{Cancelar}$	101
	$\overline{x_1}\cdot\overline{Cancelar}$	111
110	Cancelar	010
	$x_1\cdot\overline{Cancelar}$	100
	$\overline{x_1}\cdot\overline{Cancelar}$	110
101	1	000
100	1	000
010	1	000

Tabela 5.4 – Tabela de excitação dos flip-flops.

5.6.4 Minimização Lógica

A minimização das equações lógicas destinadas a implementações de dois níveis pode ser feita recorrendo a vários métodos. Um dos métodos mais conhecidos, mas que possui enormes limitações ao nível do número máximo de variáveis envolvidas,

são os mapas de *Karnaugh*. No entanto, a utilização da técnica de redução do número de variáveis aplicadas ao núcleo da unidade de controlo tornou possível a utilização deste método.

Equação de Excitação do Flip-flop 1

A equação de excitação do flip-flop 1 foi já determinada em 5.6.3, sendo repetida aqui por conveniência.

$$D_1 = \overline{Q_3} \cdot \overline{Q_2} \cdot \overline{Q_1} \cdot x_1 + \overline{Q_3} \cdot \overline{Q_2} \cdot \overline{Q_1} \cdot \overline{x_1} \cdot M100 +$$
$$+ \overline{Q_3} \cdot \overline{Q_2} \cdot Q_1 \cdot x_1 \cdot \overline{Cancelar} + \overline{Q_3} \cdot \overline{Q_2} \cdot Q_1 \cdot \overline{x_1} \cdot M100 \cdot \overline{Cancelar} + \overline{Q_3} \cdot \overline{Q_2} \cdot Q_1 \cdot \overline{x_1} \cdot \overline{M100} \cdot \overline{Cancelar} +$$
$$+ \overline{Q_3} \cdot Q_2 \cdot Q_1 \cdot x_1 \cdot \overline{Cancelar} + \overline{Q_3} \cdot Q_2 \cdot Q_1 \cdot \overline{x_1} \cdot \overline{M100} \cdot \overline{Cancelar} +$$
$$+ Q_3 \cdot Q_2 \cdot Q_1 \cdot x_1 \cdot \overline{Cancelar} + Q_3 \cdot Q_2 \cdot Q_1 \cdot \overline{x_1} \cdot \overline{Cancelar}$$

O mapa de *Karnaugh* construído a partir desta equação está representado na Figura 5.18.

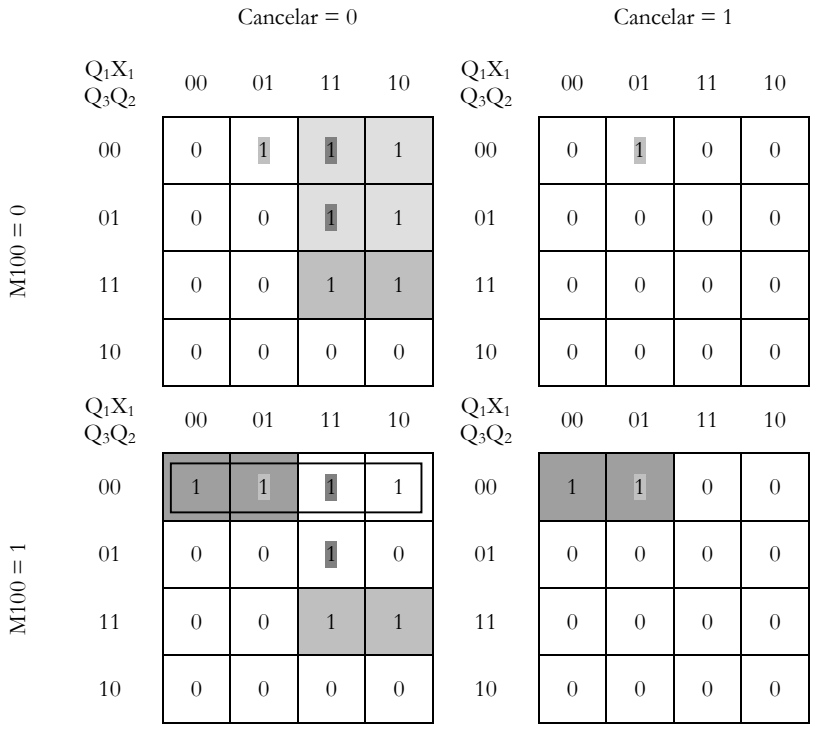


Figura 5.18 – Mapa de *Karnaugh* para minimização da equação de excitação do flip-flop 1.

Após simplificação obtém-se a seguinte equação minimizada:

$$D_1 = \overline{Q_3} \cdot \overline{Q_2} \cdot \overline{Q_1} \cdot x_1 + \overline{Q_3} \cdot Q_1 \cdot x_1 \cdot \overline{Cancelar} + \overline{Q_3} \cdot Q_1 \cdot \overline{M100} \cdot \overline{Cancelar} + Q_3 \cdot Q_2 \cdot Q_1 \cdot \overline{Cancelar} +$$
$$+ \overline{Q_3} \cdot \overline{Q_2} \cdot M100 \cdot \overline{Cancelar} + \overline{Q_3} \cdot \overline{Q_2} \cdot \overline{Q_1} \cdot M100$$

Equação de Excitação do Flip-flop 2

A equação de excitação do flip-flop 2 foi já determinada em 5.6.3, sendo repetida aqui por conveniência.

$$D_2 = \overline{Q_3}\cdot\overline{Q_2}\cdot\overline{Q_1}\cdot\overline{x_1}\cdot M100 + \overline{Q_3}\cdot\overline{Q_2}\cdot Q_1\cdot Cancelar +$$
$$+\overline{Q_3}\cdot\overline{Q_2}\cdot Q_1\cdot x_1\cdot\overline{Cancelar} + \overline{Q_3}\cdot Q_2\cdot\overline{Q_1}\cdot\overline{x_1}\cdot M100\cdot\overline{Cancelar} + \overline{Q_3}\cdot Q_2\cdot Q_1\cdot Cancelar +$$
$$+\overline{Q_3}\cdot Q_2\cdot Q_1\cdot x_1\cdot\overline{Cancelar} + Q_3\cdot\overline{Q_2}\cdot\overline{Q_1}\cdot\overline{x_1}\cdot M100\cdot\overline{Cancelar} + Q_3\cdot\overline{Q_2}\cdot Q_1\cdot\overline{x_1}\cdot\overline{M100}\cdot\overline{Cancelar} +$$
$$+Q_3\cdot Q_2\cdot Q_1\cdot Cancelar + Q_3\cdot Q_2\cdot Q_1\cdot\overline{x_1}\cdot\overline{Cancelar} +$$
$$+Q_3\cdot Q_2\cdot\overline{Q_1}\cdot Cancelar + Q_3\cdot Q_2\cdot\overline{Q_1}\cdot\overline{x_1}\cdot\overline{Cancelar}$$

O mapa de *Karnaugh* construído a partir desta equação está representado na Figura 5.19.

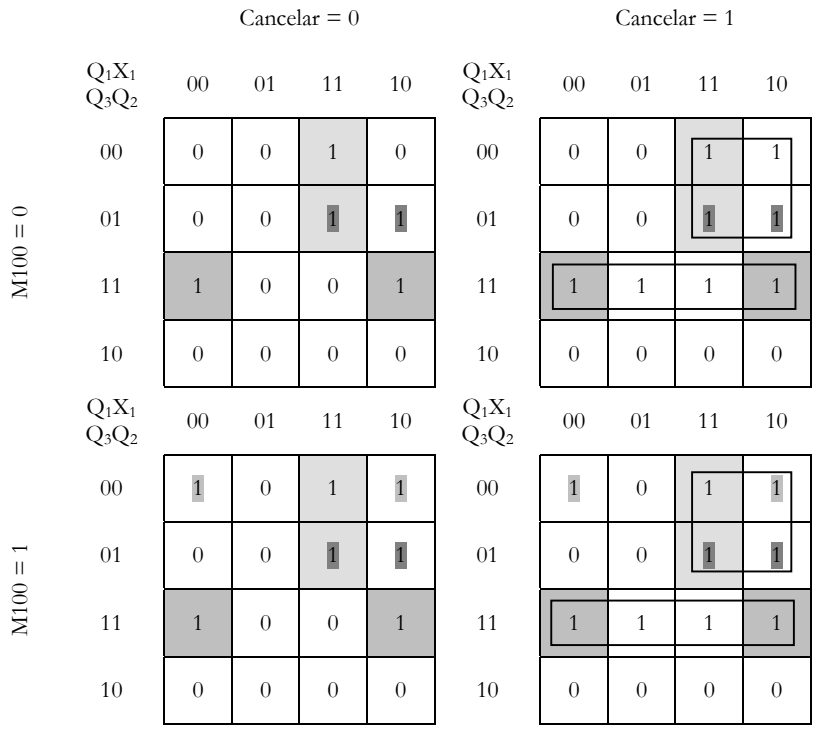


Figura 5.19 – Mapa de *Karnaugh* para minimização da equação de excitação do flip-flop 2.

Após simplificação obtém-se a seguinte equação minimizada:

$$D_2 = \overline{Q_3}\cdot Q_1\cdot x_1 + \overline{Q_3}\cdot Q_2\cdot Q_1 + Q_3\cdot Q_2\cdot\overline{x_1} + \overline{Q_3}\cdot\overline{Q_2}\cdot\overline{x_1}\cdot M100 +$$
$$+\overline{Q_3}\cdot Q_1\cdot Cancelar + Q_3\cdot Q_2\cdot Cancelar$$

Equação de Excitação do Flip-flop 3

A equação de excitação do flip-flop 3 foi já determinada em 5.6.3, sendo repetida aqui por conveniência.

$$
\begin{aligned}
D_3 = {} & \overline{Q_3} \cdot \overline{Q_2} \cdot Q_1 \cdot \overline{x_1} \cdot M100 \cdot \overline{Cancelar} + \\
& + \overline{Q_3} \cdot Q_2 \cdot Q_1 \cdot x_1 \cdot \overline{Cancelar} + \overline{Q_3} \cdot Q_2 \cdot Q_1 \cdot \overline{x_1} \cdot M100 \cdot \overline{Cancelar} + \\
& + Q_3 \cdot Q_2 \cdot Q_1 \cdot x_1 \cdot \overline{Cancelar} + Q_3 \cdot Q_2 \cdot Q_1 \cdot \overline{x_1} \cdot \overline{Cancelar} + \\
& + Q_3 \cdot Q_2 \cdot \overline{Q_1} \cdot x_1 \cdot \overline{Cancelar} + Q_3 \cdot Q_2 \cdot \overline{Q_1} \cdot \overline{x_1} \cdot \overline{Cancelar}
\end{aligned}
$$

O mapa de *Karnaugh* construído a partir desta equação está representado na Figura 5.20.

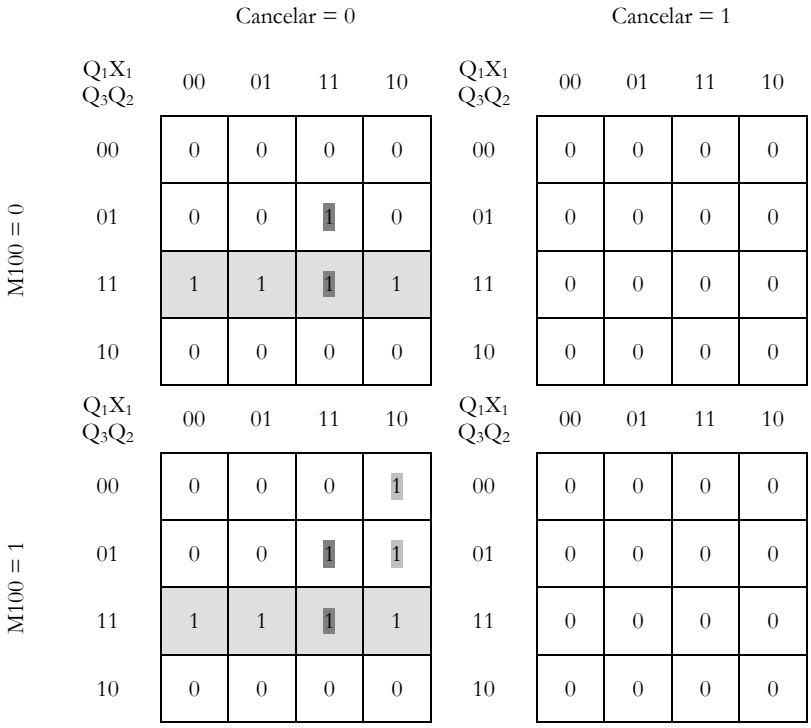


Figura 5.20 – Mapa de *Karnaugh* para minimização da equação de excitação do flip-flop 3.

Após simplificação obtém-se a seguinte equação minimizada:

$$
D_3 = Q_2 \cdot Q_1 \cdot x_1 \cdot \overline{Cancelar} + Q_3 \cdot Q_2 \cdot \overline{Cancelar} + \overline{Q_3} \cdot Q_1 \cdot \overline{x_1} \cdot M100 \cdot \overline{Cancelar}
$$

Após a minimização das equações de excitação dos flip-flops, o próximo passo é a minimização das equações de saída que será realizada na próxima secção.

Equações das saídas

A equação da saída "Lata" pode ser minimizada directamente obtendo-se o seguinte resultado:

$$\text{Lata} = Q_3 \cdot \overline{Q_2} \cdot Q_1 + Q_3 \cdot \overline{Q_2} \cdot \overline{Q_1} = Q_3 \cdot \overline{Q_2}$$

Nas equações das saídas "Troco" e "Devolução" não pode ser realizada qualquer simplificação, obtendo-se o seguinte resultado final:

$$\text{Troco} = Q_3 \cdot \overline{Q_2} \cdot \overline{Q_1}$$

$$\text{Devolução} = \overline{Q_3} \cdot Q_2 \cdot \overline{Q_1}$$

Por último, para simplificar a equação da saída (primária) "Rejeição" e da saída de controlo "y1" é vantajoso recorrer a um mapa de *Karnaugh*, que neste caso possui apenas três variáveis. Uma vez que se trata de uma máquina de Moore, as saídas só dependem das variáveis de estado. A equação inicial é a seguinte:

$$\text{Rejeição} = y_1 = Q_3 \cdot Q_2 \cdot Q_1 + Q_3 \cdot Q_2 \cdot \overline{Q_1} + Q_3 \cdot \overline{Q_2} \cdot Q_1 + Q_3 \cdot \overline{Q_2} \cdot \overline{Q_1} + \overline{Q_3} \cdot Q_2 \cdot \overline{Q_1}$$

Na
Figura 5.21 é mostrado o mapa de *Karnaugh* correspondente.

Q_3 \ Q_2Q_1	00	01	11	10
0	0	0	0	1
1	1	1	1	1

Figura 5.21 – Mapa de *Karnaugh* para minimização da equação das saídas "Rejeição" e "y_1".

Após simplificação obtém-se a seguinte equação minimizada:

$$\text{Rejeição} = y_1 = Q_3 + Q_2 \cdot \overline{Q_1}$$

5.6.5 Desenho do Esquema Final do Circuito

Após a obtenção das equações minimizadas de excitação dos flip-flops e das saídas pode ser desenhado o esquema final do circuito, o qual é mostrado na Figura 5.22.

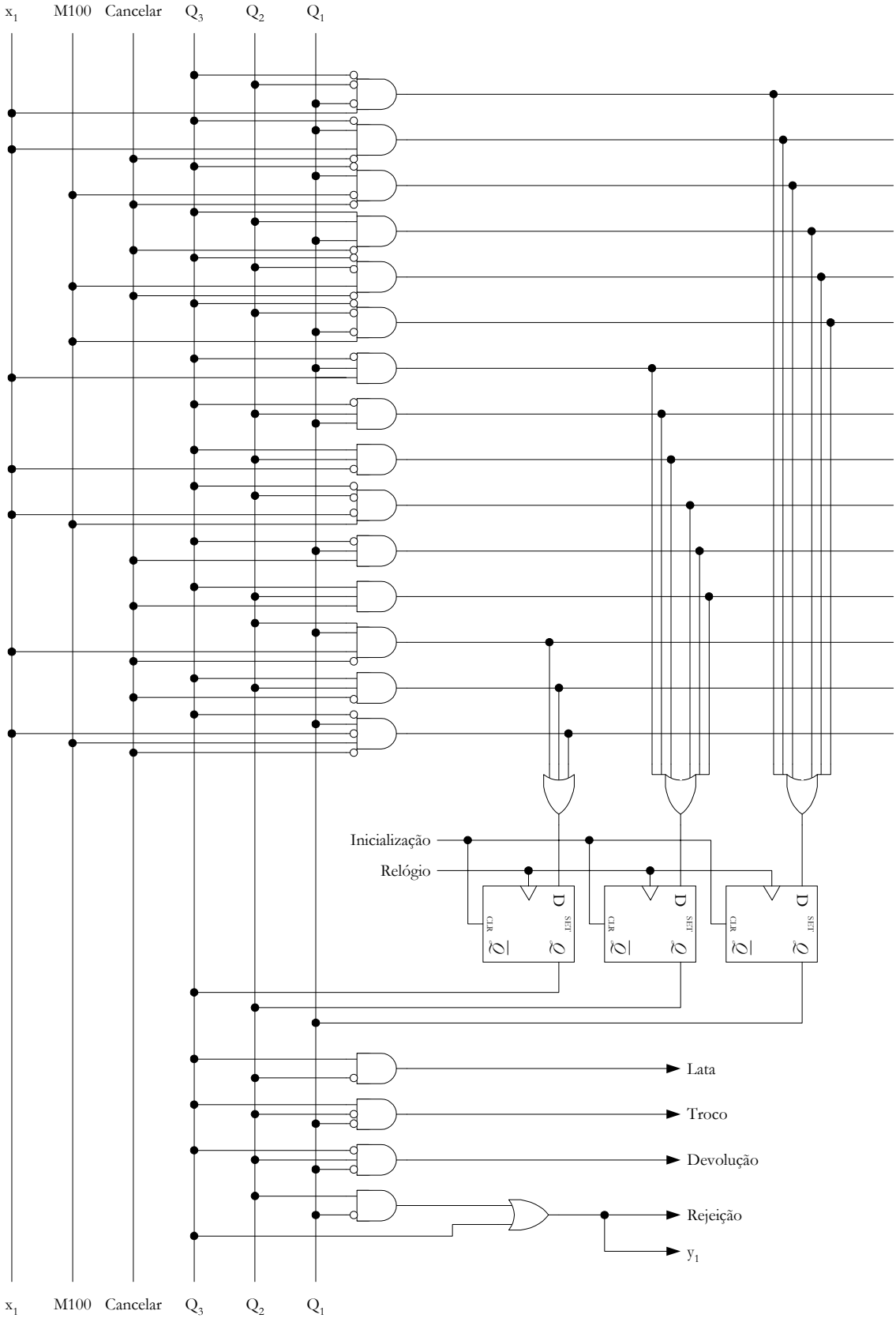


Figura 5.22 – Esquema final da unidade de controlo da máquina de venda automática sintetizada com codificação de estados DTM e implementada com dois níveis de lógica.

5.7 Optimização Automática da Componente Combinatória

Na síntese manual de uma unidade de controlo uma parte significativa do tempo de projecto é utilizado na optimização lógica do circuito. Isto faz com que a análise de diferentes abordagens de projecto e em particular o estudo do impacto de várias codificações de estado na complexidade da componente combinatória do circuito não possa ser realizada de forma eficiente. Assim, pode ser conveniente adoptar uma estratégia de projecto mista em que após a minimização e codificação de estados as equações de excitação dos flip-flops e das saídas possam ser obtidas rapidamente através da utilização de ferramentas de optimização lógica. De notar que só é razoável aplicar este método a unidades de controlo com um número reduzido de estados, ou seja, nos casos em que as etapas anteriores à optimização lógica possam ser realizadas rapidamente. De facto, a abordagem mais geral e que pode ser usada em todos os casos é aquela em que são utilizadas ferramentas de projecto assistido por computador em todas as fases de desenvolvimento de uma unidade de controlo.

Mesmo no caso de um fluxo de projecto completamente assistido por computador é em geral conveniente a intervenção do projectista em determinadas etapas para tomar decisões e estabelecer restrições que permitam melhorar a qualidade dos resultados produzidos pelas ferramentas. Este processo vai ser tratado na próxima secção. Para já vai ser ilustrada uma forma de realizar automaticamente a optimização lógica de um circuito combinatório que pode ou não fazer parte de uma unidade de controlo. Para este efeito podem ser utilizadas várias ferramentas, das quais destacamos duas que já foram referidas em 5.5:

- O *Espresso* para optimização de circuitos de dois níveis na forma de soma de produtos;
- O *misII* para optimização de circuitos multi-nível.

Uma das vantagens destas ferramentas é aceitarem a mesma descrição de entrada. O seu formato é semelhante a uma tabela de verdade que descreve as interligações de uma PLA (Figura 5.23), diferindo apenas o tipo dos ficheiros de saída. Enquanto o *Espresso* produz uma descrição de uma PLA semelhante à de entrada mas optimizada, o *misII* gera um conjunto de equações em que o resultado de uma pode ser utilizado como variável de entrada de outras, reflectindo assim o carácter multi-nível do circuito (Figura 5.24).

Estas ferramentas foram utilizadas para optimizar a componente combinatória da unidade de controlo da máquina de venda automática e analisar o impacto de várias codificações de estado atribuídas manualmente na área do circuito. Para tal foram utilizadas quatro codificações diferentes: binária, *gray*, DTM e DMEES. As suas listagens encontram-se na Tabela 5.5. Na Tabela 5.6 são resumidos os parâmetros do circuito relacionados com a sua área: a área da PLA (para implementações de dois níveis) e o número de literais (para implementações multi-nível). Tal como seria de esperar, as codificações DTM e DMEES são aquelas que fornecem os melhores resultados. As listagens completas dos ficheiros de entrada utilizados e dos produzidos pelas ferramentas de optimização são apresentadas no anexo I.

```
.i 7
.o 7
0001---             0010000
00001--             0100000
00000--             0000000
001---1             1110000
0011--0             0100000
00101-0             0110000
00100-0             0010000
010---1             1110000
0101--0             0110000
01001-0             1000000
01000-0             0100000
011---1             1110001
011--10             1010001
011--00             0110001
100---1             1110001
100--10             1100001
100--00             1000001
101----             0001001
110----             0001101
111----             0000011
.e
```

Figura 5.23 – Exemplo de uma descrição de um circuito combinatório no formato de entrada aceite pelas ferramentas de optimização utilizadas.

```
D3 = Q2*!Cancelar*[19] + Q1*!Cancelar*![18] + Q3*Q2*!Cancelar;
D2 = Q1*!D1*!Rejeicao + Q2*!D3*!Devolucao + Q1*[19] + !Continuar*D3 + ![18];
D1 = Q1*!Cancelar*!Rejeicao*[18] + Q1*D3*Rejeicao + !Q1*[19] + !Q1*![18] + !Q2*D3;
Lata = Q3*!Q2;
Troco = !Q1*Lata;
Devolucao = !Q3*Q2*!Q1;
Rejeicao = Devolucao + Q3;
[17] = Rejeicao + !M100;
[18] = [17] + M50;
[19] = M50*!Rejeicao;
```

Figura 5.24 – Exemplo de uma descrição de um circuito combinatório optimizado no formato de saída do *misII*.

	Tipos de Codificação			
Estados	*Binária*	*Gray*	*DTM*	*DMEES*
EInicial	000	000	000	000
E50	001	001	001	001
E100	010	011	011	011
E150	011	010	111	111
E200	100	110	110	101
ELata	101	111	101	010
ELataTroco	110	101	100	100
ECancelada	111	100	010	110

Tabela 5.5 – Listagem das codificações de estado utilizadas para optimização automática da componente combinatória da unidade de controlo da máquina de venda automática.

	Tipos de Codificação			
Estados	*Binária*	*Gray*	*DTM*	*DMEES*
Área da PLA	92	79	78	73
Nº de Literais	54	57	48	48

Tabela 5.6 – Resultados obtidos com os optimizadores lógicos Espresso e *misII* a partir das codificações de estado da Tabela 5.5.

5.8 Síntese Lógica Sequencial Assistida por Computador

Quando o número de estados, transições ou sinais de entrada e de saída de uma unidade de controlo é elevado a síntese manual torna-se num processo bastante complexo, moroso e sujeito a erros. Devido também ao número de variáveis envolvidas a solução conseguida pode não ser óptima nem tão pouco uma boa aproximação. Além disso, devido ao demasiado tempo que requer, torna proibitiva a exploração de várias alternativas ou abordagens de projecto. De notar que na síntese de circuitos sequenciais qualquer alteração ao nível das transições, estados e sua codificação obriga geralmente a repetir todos os passos seguintes do projecto.

Mesmo em exemplos simples, como o caso da máquina de venda automática atrás descrita, a exploração manual de várias alternativas de projecto requer um tempo considerável. Devido a estes factos, à complexidade crescente dos sistemas e à necessidade de tempos de projecto cada vez menores, as ferramentas de síntese lógica são cada vez mais importantes e utilizadas. Nesta secção vão ser dados alguns exemplos da sua utilização. Além disso, pretende-se também alcançar os seguintes objectivos:

- Comparar a qualidade dos resultados produzidos por diversas ferramentas de síntese em termos da área do circuito e frequência máxima de funcionamento;
- Analisar o impacto de várias técnicas de codificação de estados nos parâmetros referidos no ponto anterior;
- Com base nos resultados dos pontos anteriores seleccionar as abordagens e as técnicas de projecto mais adequadas de forma a estabelecer um fluxo de projecto de unidades de controlo destinadas a implementar nas FPGAs XC6200, que tenha como ponto de partida uma descrição comportamental e possa ser realizada por ferramentas de projecto assistido por computador e com o mínimo de intervenção do projectista.

Para atingir estes objectivos foram utilizadas as seguintes três ferramentas:

- *Leonardo Spectrum* da *Exemplar Logic*, versão 1999.1j a executar sobre o sistema operativo *Microsoft Windows 2000 Professional*;
- *Synopsys Design Analyzer* a executar sobre o sistema operativo *HPUX*;
- SIS (*Sequential Interactive Synthesis*) versão 1.2 a executar sobre o sistema operativo *RedHat Linux 6.2 – Kernel 2.2.14*.

Enquanto as duas primeiras são ferramentas comerciais, a última é um conjunto de aplicações de domínio académico que foram desenvolvidas na Universidade de *Berkeley* para efeitos de investigação. Como o objectivo desta secção não é fazer uma avaliação exaustiva das referidas ferramentas, à semelhança das secções anteriores vai ser utilizado somente o exemplo da unidade de controlo da máquina de venda automática.

A principal vantagem da generalidade das aplicações comerciais é a facilidade de utilização uma vez que possuem interfaces gráficos com o utilizador. Por oposição, a ferramenta de investigação utilizada possui um interface menos amigável baseado em

comandos. Contudo, possui as vantagens de ser de domínio público e de proporcionar um maior controlo sobre os processos de síntese e optimização através de uma utilização interactiva e da disponibilização de várias aplicações e de um número elevado de comandos. No entanto, obriga o utilizador a possuir um conhecimento razoável dos algoritmos utilizados de forma a tirar partido de todas as suas potencialidades e a aplicar os mais adequados a cada situação.

Como exemplo da flexibilidade do SIS pode-se citar as aplicações de minimização e de codificação de estados que implementam vários algoritmos exactos e heurísticos que o utilizador pode invocar interactivamente, analisar as saídas produzidas e escolher aquele que fornecer melhores resultados.

Nas próximas secções são apresentados e analisados os circuitos obtidos para a unidade de controlo da máquina de venda automática com cada uma das ferramentas enumeradas acima.

5.8.1 Leonardo Spectrum

A primeira aplicação utilizada para sintetizar a unidade de controlo da máquina de venda automática foi o *Leonardo Spectrum* da *Exemplar Logic* versão 1999.1j. O ponto de partida para utilizar esta ferramenta foi a criação de um projecto e a inclusão do ficheiro VHDL que foi apresentado no capítulo anterior e que descreve o comportamento do circuito que se pretende sintetizar.

Os tipos de codificação de estados disponibilizados por esta ferramenta são os seguintes: binária, *gray*, *one-hot*, *two-hot* e aleatória. Para cada um destes tipos de codificação sintetizou-se duas vezes o circuito, uma em que o objectivo era reduzir a sua área (designada por optimização da área) e a outra em que se pretendia reduzir os atrasos do mesmo, ou seja, a maximizar a frequência limite de funcionamento (designada por optimização dos atrasos).

Para realizar o mapeamento na tecnologia foi utilizada uma biblioteca predefinida para ASICs que contém primitivas do tipo portas lógicas, flip-flops e multiplexadores. À excepção do número de entradas das portas lógicas, as primitivas existentes nesta biblioteca são idênticas às funções implementáveis nas células de uma FPGA XC6200. De notar, no entanto que um circuito sintetizado e mapeado em componentes desta biblioteca não pode ser passado às ferramentas de implementação da XC6200 devido à diferença acima referida e a discrepâncias nos nomes das primitivas. No entanto, com a ajuda de um utilitário que faz parte desta aplicação é possível construir bibliotecas para mapeamento na tecnologia completamente personalizadas. Neste caso, optou-se por não o fazer uma vez que foi realizado um procedimento análogo com o SIS e que como veremos mais à frente, este último sintetiza circuitos mais eficientes em termos de área e atrasos do que o *Leonardo Spectrum*.

Os diagramas esquemáticos dos circuitos sintetizados utilizando optimização da área estão apresentados no anexo II. A tabela e o gráfico da Figura 5.25 resumem os resultados obtidos em termos da área do circuito e da frequência máxima de funcionamento para cada tipo de codificação usando este tipo de optimização. Os circuitos obtidos usando optimização dos atrasos encontram-se também no anexo II. Os respectivos resultados são resumidos na Figura 5.26.

Tipo de Codificação	Área do Circuito	Freq. de Funcionamento (MHz)
Binária	298	255,5
Gray	332	377,1
One-Hot	221	528,9
Two-Hot	464	322,0
Aleatória	300	274,1

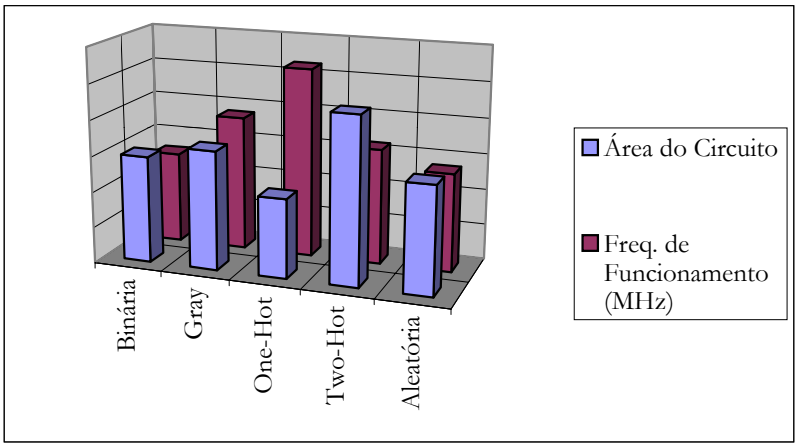


Figura 5.25 – Sumário dos resultados da síntese da unidade de controlo da máquina de venda automática obtidos com o *Leonardo Spectrum* em termos de área do circuito e frequência de funcionamento utilizando optimização da área.

Tipo de Codificação	Área do Circuito	Freq. de Funcionamento (MHz)
Binária	263	272,9
Gray	333	361,9
One-Hot	234	613,5
Two-Hot	466	382,9
Aleatória	236	216,7

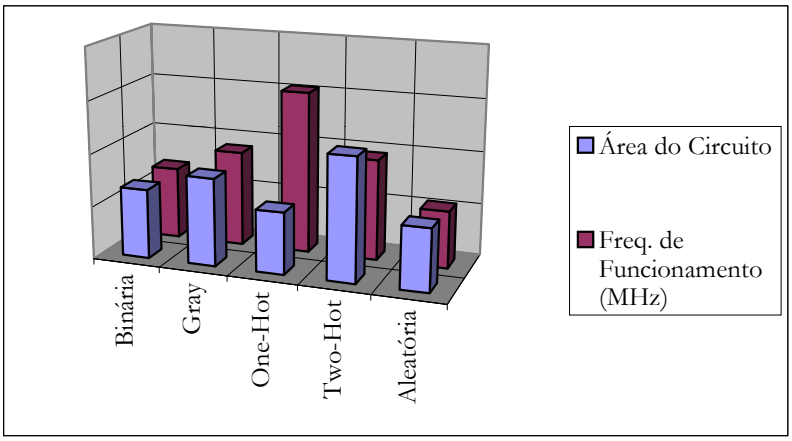


Figura 5.26 – Sumário dos resultados da síntese da unidade de controlo da máquina de venda automática obtidos com o *Leonardo Spectrum* em termos de área do circuito e frequência de funcionamento utilizando optimização dos atrasos.

Da análise dos resultados conclui-se que o método de codificação de estados *one-hot* é aquele fornece melhores resultados em qualquer uma das optimizações. Em termos de área do circuito o pior método de codificação é o *two-hot*, o que é natural, uma vez que está mais vocacionado para circuitos sequenciais com um elevado número de estados. Um circuito baseado neste tipo de codificação utiliza mais flip-flops do que um que use uma codificação de comprimento mínimo mas menos flip-flops do que no caso de *one-hot*. Desta forma é possível atingir um compromisso aceitável entre o número de flip-flops e o número de variáveis envolvidas em cada transição de estados.

Através de uma análise comparativa dos resultados de ambos os tipos de optimização conclui-se que a área e a frequência máxima de funcionamento nem sempre são coerentes com o tipo de optimização escolhida. Além disso, da observação dos diagramas esquemáticos (ver anexo II) parece razoável concluir que a complexidade dos circuitos sintetizados parece ser exagerada para o sistema em causa, razão pela qual esta ferramenta parece não ser muito eficiente em termos de optimização lógica. Finalmente, também não é capaz de escolher uma codificação de estados que reduza os custos da área do circuito sintetizado. Em conjunto, estes três factores levaram à síntese do mesmo circuito com outras ferramentas, cujos resultados serão apresentados nas próximas secções.

5.8.2 Synopsys

Devido aos resultados pouco convincentes obtidos com o *Leonardo Spectrum* optou-se por sintetizar a unidade de controlo da máquina de venda automática com uma ferramenta conceituada como o *Synopsys*.

Tal como no caso anterior, foi utilizada a descrição comportamental em VHDL da unidade de controlo apresentada no capítulo anterior como ponto de partida do processo de síntese.

À semelhança da ferramenta anterior foi também utilizada uma biblioteca de ASICs para mapeamento na tecnologia e que possui primitivas análogas às implementáveis numa célula da FPGA XC6200.

Neste caso foram analisadas as seguintes codificações de estado:

- Binária;
- *Gray*;
- *One-hot*;
- Manual1 (corresponde ao método de codificação baseado na heurística DTM atrás descrita);
- Manual2 (corresponde ao método de codificação baseado na heurística DMEES atrás descrita);
- Automática1 (codificação determinada automaticamente pela aplicação segundo critérios próprios de optimização);
- Automática2 (semelhante à anterior mas com a restrição do código do estado inicial ser 000, de forma a facilitar a inicialização da máquina).

As listagens destas codificações são apresentadas na Tabela 5.7.

	Tipos de Codificação						
Estados	*Binária*	*Gray*	*One-Hot*	*Manual1*	*Manual2*	*Automatica1*	*Automatica2*
EInicial	000	000	10000000	000	000	111	000
E50	001	001	01000000	001	001	101	101
E100	010	011	00100000	011	011	100	001
E150	011	010	00010000	111	111	001	111
E200	100	110	00001000	110	101	000	011
ELata	101	111	00000100	101	010	011	010
ELataTroco	110	101	00000010	100	100	010	110
ECancelada	111	100	00000001	010	110	110	100

Tabela 5.7 – Listagem das codificações de estado utilizadas no *Synopsys*.

Os diagramas esquemáticos dos circuitos sintetizados são apresentados no anexo II. Os resultados da área e frequência de funcionamento máxima de cada um destes circuitos estão resumidos na Figura 5.27. Para facilitar a análise dos resultados a área do circuito é dividida em duas componentes: a área da parte combinatória (portas lógicas) e a área da parte não combinatória (flip-flops). Tal como se suspeitava, a análise dos resultados obtidos confirma as suspeitas da baixa qualidade dos circuitos sintetizados com o *Leonardo Spectrum*, principalmente ao nível da elevada complexidade do circuito e consequentemente da área por ele ocupada.

De notar que os valores das frequências de funcionamento não podem ser comparados entre diferentes ferramentas, a menos que se utilize a mesma biblioteca para mapeamento na tecnologia, uma vez que os valores obtidos são altamente dependentes dos parâmetros dos seus componentes.

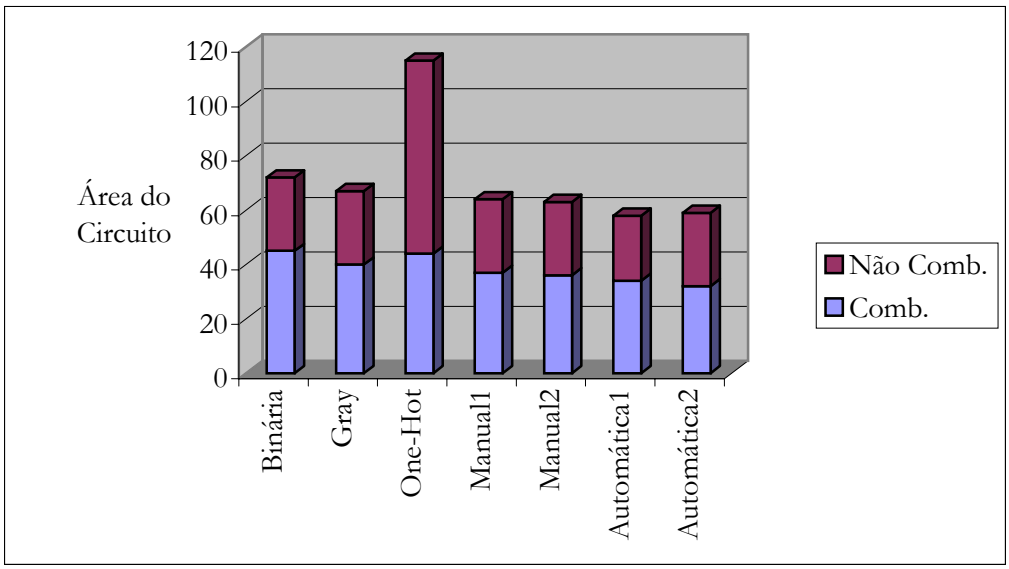
De todas as codificações, aquela que forneceu melhores resultados quer em termos de área do circuito, quer em termos de frequência de funcionamento foi a automática sem restrições (Automática1). Do ponto de vista de área a segunda melhor codificação é a Automática2, enquanto a *one-hot* possui a segunda melhor frequência de funcionamento. Os piores resultados correspondem à codificação binária, o que pode ser explicado pelo facto de possuir uma distância total elevada (menor adjacência entre os códigos dos estados consecutivos).

Os valores da área do circuito para codificação de estados *one-hot* devem ser analisados com o devido cuidado, uma vez que para a tecnologia de implementação utilizada neste trabalho (arquitectura de FPGAs XC6200) o custo em termos de área de um flip-flop é exactamente igual ao de uma porta lógica de duas entradas ou um multiplexador de 2 para 1.

Com esta ferramenta já foi possível analisar de uma forma mais precisa os méritos e os problemas dos vários tipos de codificação de estados. Contudo, uma medida mais realista da área ocupada pelo circuito só vai ser possível obter com a próxima ferramenta, o SIS, para a qual foi desenvolvida uma biblioteca para mapeamento na tecnologia que possui as primitivas implementáveis na tecnologia utilizada neste trabalho.

Outro aspecto que vai ser possível avaliar é a implementação dos circuitos no dispositivo alvo, isto é, se as tarefas de implantação e interligação podem ser concluídas facilmente. De notar que não é razoável medir o tempo de implementação devido à simplicidade do exemplo utilizado. Finalmente, vai também ser possível obter medidas mais precisas da frequência de funcionamento máxima do circuito.

Tipo	Comprimento	Área do Circuito			Freq. de Funcionamento (MHz)
		Comb.	Não Comb.	Total	
Binária	3	45	27	72	131,6
Gray	3	40	27	67	195,7
One-Hot	8	44	71	115	200,8
Manual1	3	37	27	64	156,5
Manual2	3	36	27	63	173,3
Automática1	3	34	24	58	213,2
Automática2	3	32	27	59	174,2



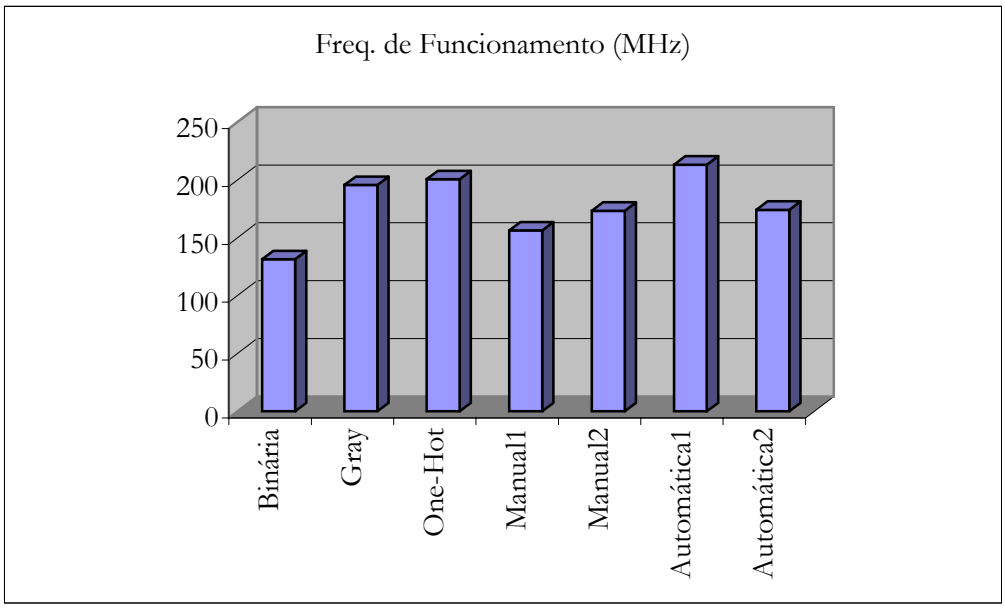


Figura 5.27 – Sumário dos resultados da síntese da unidade de controlo da máquina de venda automática obtidos com o *Synopsys* em termos de área do circuito e frequência de funcionamento.

5.8.3 SIS – Sequential Interactive Synthesis

Para concluir a apresentação da sequência das ferramentas de desenvolvimento utilizadas para sintetizar a unidade de controlo da máquina de venda automática vão ser agora mostrados os resultados obtidos com o SIS (*Sequential Interactive Synthesis*) [Sis92]. Tal como já foi referido acima, esta ferramenta foi desenvolvida em *Berkeley* como um conjunto de aplicações de investigação. As suas vantagens principais são:

- ser de domínio público e disponibilizar o código fonte,
- possuir os formatos abertos de entrada e saída das aplicações, o que a torna extensível e facilita a interacção com outras ferramentas;
- por comparação com os resultados obtidos nas secções anteriores, os algoritmos implementados podem ainda hoje ser considerados bastante actuais e até avançados.

A sua maior desvantagem relativamente às anteriores é o interface, uma vez que toda a interacção com o utilizador é realizada por intermédio de comandos. No entanto, para utilizadores experientes esta é também uma vantagem importante por permitir a elaboração de ficheiros de comandos que permitem automatizar determinadas tarefas e assim conseguir uma utilização mais eficiente da ferramenta.

Além das capacidades de síntese de circuitos sequenciais síncronos e assíncronos através de ferramentas e comandos para minimização e codificação de estados e manipulação de diagramas de transição de estados e grafos de transição de sinal, incorpora as ferramentas de minimização lógica de dois níveis "*Espresso*" e multi-nível "*misII*" já utilizadas na secção 5.7 para realizar a optimização automática da parte combinatória de uma unidade de controlo. Na execução das experiências cujos resultados são apresentados nesta secção são relevantes apenas a aplicação de codificação de estados para circuitos multi-nível "*Jedi*" e os comandos de minimização lógica, já que o modelo comportamental da unidade de controlo da máquina de venda automática encontra-se já minimizado em termos do número de estados.

O ponto de partida para utilizar esta ferramenta na síntese de uma unidade de controlo é a elaboração de um ficheiro no formato *kiss2* que descreve o seu comportamento e que é basicamente uma tabela de transição de estados em que podem ser utilizados os símbolos "0", "1" e "-" nos vectores de entrada e de saída. Opcionalmente, esta descrição pode ser inserida num ficheiro no formato "*blif*" (*Berkeley Logic Interchange Format*) que pode ser utilizado para atribuir nomes às entradas e saídas do circuito ou interligar a unidade de controlo com outros componentes. A descrição "*kiss2*" da unidade de controlo da máquina de venda automática é apresentada na Figura 5.28 e resulta de pequenas alterações realizadas na tabela de transição de estados apresentada no capítulo anterior.

Na realização das experiências foram utilizados dois tipos de codificação de estados: automática de comprimento mínimo e *one-hot*. No primeiro caso os códigos dos estados foram atribuídos automaticamente pela aplicação "*Jedi*", tendo-se realizado dois tipos de optimização: área e atrasos. A listagem das codificações utilizadas encontra-se na Tabela 5.8.

Neste caso não foi possível obter esquemáticos dos circuitos gerados devido a alguns problemas das ferramentas utilizadas para converter os modelos em VHDL estrutural numa representação gráfica. Ao contrário das ferramentas anteriores foi possível completar o projecto da unidade de controlo desde a sua especificação até à sua implementação numa FPGA XC6200. O modelo estrutural resultante da síntese foi simulado com a aplicação *VeriBest VHDL 15.01* tendo-se obtido resultados análogos aos apresentados no capítulo anterior.

```
.model MaqVenda
.inputs M50 M100 Continuar Cancelar
.outputs Lata Troco Devolucao Rejeicao
.start_kiss
.i 4
.o 4
.r EInicial
1---    EInicial       E50            0000
01--    EInicial       E100           0000
00--    EInicial       EInicial       0000
---1    E50            ECancelada     0000
1--0    E50            E100           0000
01-0    E50            E150           0000
00-0    E50            E50            0000
---1    E100           ECancelada     0000
1--0    E100           E150           0000
01-0    E100           E200           0000
00-0    E100           E100           0000
---1    E150           ECancelada     0001
--10    E150           ELata          0001
--00    E150           E150           0001
---1    E200           ECancelada     0001
--10    E200           ELataTroco     0001
--00    E200           E200           0001
----    ELata          EInicial       1001
----    ELataTroco     EInicial       1101
----    ECancelada     EInicial       0011
.end_kiss
.end
```

Figura 5.28 – Descrição comportamental da unidade de controlo da máquina de venda automática no formato *kiss2* inserido num ficheiro *blif*.

	Tipos de Codificação	
Estados	*Automatica*	*One-Hot*
EInicial	111	10000000
E50	110	01000000
E100	100	00100000
E150	000	00010000
E200	001	00001000
ELata	011	00000100
ELataTroco	101	00000010
ECancelada	010	00000001

Tabela 5.8 – Listagem das codificações de estado utilizadas no SIS.

A Figura 5.29 mostra as representações físicas dos três circuitos obtidos após a implementação com as ferramentas de mapeamento, implantação e interligação da FPGA XC6200 ((a) optimização da área; (b) optimização dos atrasos; (c) *one-hot*). As duas primeiras opções de síntese utilizam codificações de comprimento mínimo.

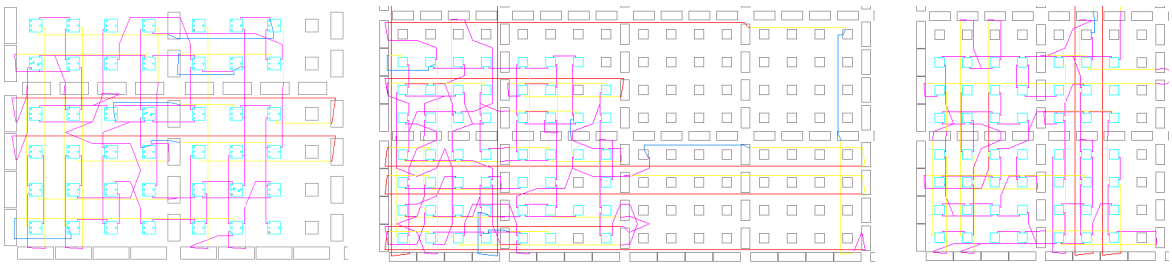


Figura 5.29 – Representação física da unidade de controlo após a implementação com o XACT6000. Circuito sintetizado com (a) optimização da área; (b) optimização dos atrasos; (c) codificação de estados *one-hot*.

Da observação da Figura 5.29 é fácil concluir que o circuito sintetizado com optimização dos atrasos é aquele que ocupa uma área maior. Os resultados obtidos em termos de área do circuito e frequência máxima de funcionamento para cada uma das opções de síntese encontram-se resumidos na Figura 5.30. De notar que ao contrário do que sucedeu com o *Leonardo Spectrum*, neste caso existe coerência entre os parâmetros do circuito e o tipo de optimização escolhida. Em termos de área, o circuito que utiliza codificação de estados *one-hot* é pior do que o optimizado para área, mas melhor do que o sintetizado com optimização dos atrasos.

A análise temporal do circuito foi realizada após a sua implementação, pelo que os valores são reais, ao contrário do que acontece com os casos anteriores em que os resultados obtidos devem ser considerados apenas como primeiras aproximações. A diferença de uma ordem de grandeza a menos nas frequências de funcionamento, relativamente aos circuitos sintetizados com as ferramentas anteriores e mapeados em componentes de bibliotecas para ASICs, deve-se aos atrasos provocados pelos elementos de comutação da FPGA.

Em termos de frequência de funcionamento a codificação de estados *one-hot* é aquela que fornece os melhores resultados, podendo proporcionar um bom compromisso entre a frequência de funcionamento e a área ocupada, para além de possuir um processo de síntese bastante simples. Além disso, uma vez que cada transição de estado só depende de uma variável de estado, o número médio de entradas por porta lógica é menor do que quando se usa uma codificação de comprimento mínimo, que é um factor bastante importante em tecnologias de implementação de pequena granulosidade e com poucas entradas por porta lógica, como é o caso da arquitectura de FPGAs XC6200. Durante o projecto desta unidade de controlo constatou-se ainda que no circuito sintetizado com optimização dos atrasos só foi possível concluir com sucesso a sua implementação após algumas iterações, o que não sucedeu com os restantes dois circuitos.

A maior desvantagem da codificação de estados *one-hot* é necessitar de um elevado número de flip-flops. Contudo, a partir do momento em que o número de primitivas combinatórias utilizadas num circuito é superior ao número de flip-flops e em cada célula pode ser combinada uma porta lógica com um flip-flop esta desvantagem é anulada. De notar que na família XC6200 é impossível utilizar separadamente as componentes combinatória e sequencial de uma célula. Outro argumento a favor da utilização da codificação de estados *one-hot* em unidades de

controlo virtuais é o facto de somente uma parte do circuito estar implementada em hardware, o que contribui para uma diminuição do número de flip-flops necessários.

Com base nos resultados obtidos, o método de codificação escolhido para implementar unidades de controlo na FPGA XC6200 foi o *one-hot*. De notar que estas conclusões foram baseadas apenas num exemplo de uma FSM e que conclusões mais precisas só seriam possíveis obter com a utilização de um conjunto adequado de máquinas de teste, também vulgarmente designadas por *benchmarks*. No entanto, este não era o objectivo deste capítulo. Além disso, estas conclusões são também suportadas pelos resultados publicados em [OliLauSkl98, Sklyarov00].

Tipo de Codificação	Área do Circuito	Freq. de Funcionamento (MHz)
Automática (área)	42	13,4
Automática (atraso)	56	18,3
One-Hot	49	22,3

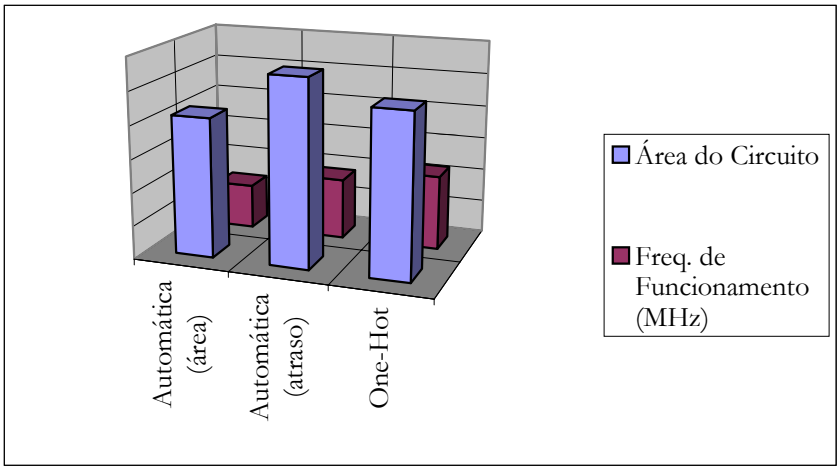


Figura 5.30 – Sumário dos resultados da síntese da unidade de controlo da máquina de venda automática obtidos com o *SIS* em termos de área do circuito e frequência de funcionamento para diferentes opções de síntese.

5.9 Fluxo de Projecto de Unidades de Controlo Orientado para a XC6200

Nesta secção vai ser apresentado o fluxo de projecto concebido para desenvolver unidades de controlo a implementar em FPGAs da família XC6200 da Xilinx. A sua representação gráfica encontra-se na Figura 5.31. Aqui pretende-se realçar as tarefas envolvidas na síntese sequencial e lógica do circuito uma vez que são aquelas para as quais não existe qualquer suporte das ferramentas tradicionalmente utilizadas no projecto de sistemas baseados nesta tecnologia de implementação e que foram descritas no capítulo 2 desta dissertação.

Neste fluxo de projecto a função principal é desempenhada pelo SIS em conjunto com alguns ficheiros standard de comandos. Tal como já foi descrito em

5.8.3, o ponto de partida é a preparação de um ficheiro que contenha a descrição comportamental da unidade de controlo no formato *kiss2*. Alternativamente podem ser utilizados outros formatos desde que possam ser convertidos neste de forma manual ou automática. Opcionalmente pode ser realizada a simulação do modelo para verificar se o comportamento descrito é o desejado. A etapa seguinte é a leitura deste modelo pelo SIS, o qual realizará com o auxílio de um conjunto de ficheiros standard, a síntese e optimização do circuito. Alguns exemplos das tarefas desempenhadas por esta ferramenta são: a minimização de estados, a codificação de estados, a optimização lógica e o mapeamento na tecnologia. Além de um conjunto considerável de comandos internos, esta ferramenta integra também várias aplicações das quais se destacam: *Espresso* (minimização lógica para implementações de dois níveis), *misII* (minimização lógica para implementações multi-nível), *stamina* e *sred* (minimização de estados), *Nova* (codificação de estados para implementações de dois níveis) e *Jedi* (codificação de estados para implementações multi-nível).

Para mapeamento na tecnologia foi construída a biblioteca "xc6200.genlib" que contém a descrição de todas as portas lógicas, multiplexadores e flip-flops implementáveis numa célula da arquitectura XC6200. A sua listagem encontra-se no anexo III desta dissertação. Em rigor, o mapeamento na tecnologia é realizado em duas etapas, a primeira pelo SIS e a segunda pelo XACT6000. A função mais importante desempenhada pela segunda é o agrupamento de duas primitivas na mesma célula sempre que possível, reduzindo quer a área do circuito, isto é o número de células ocupadas, quer o número de interruptores programáveis que os sinais atravessam, diminuindo desta forma os atrasos.

Após a realização das tarefas enumeradas acima, a ferramenta produz um ficheiro de saída em formato *blif* e que é essencialmente uma lista de ligações com alguns dados adicionais, tais como a informação temporal e a codificação de estados utilizada. Sobre este ficheiro é realizado um pós-processamento, cujo objectivo é gerar uma descrição em VHDL estrutural da unidade de controlo sintetizada que possa ser utilizada nas ferramentas tradicionais de projecto da XC6200 (VELAB e XACT6000).

Para simplificar as etapas de síntese e de pós-processamento foram criados alguns ficheiros de comandos e reutilizados outros que são distribuídos com a ferramenta. Segundo os autores da ferramenta os ficheiros standard permitem obter bons resultados para a generalidade dos circuitos. Contudo os melhores resultados são normalmente obtidos através de uma utilização interactiva da aplicação, sendo no entanto necessário um conhecimento aprofundado dos comandos disponíveis e dos algoritmos que estes implementam. Alternativamente, a síntese pode também ser realizada com a aplicação HGS2VHDL [LauSkl99], mas como esta só permite a utilização de estados *one-hot*, a primeira abordagem possui a vantagem de permitir explorar uma gama maior de alternativas de projecto. As fases seguintes fazem já parte do fluxo de projecto normal suportado pelas ferramentas da FPGA XC6200, o qual já foi descrito no capítulo 2 desta dissertação. Os detalhes relativos à especificação e às fases posteriores à implementação, em particular a depuração, não são aqui consideradas uma vez que já foram abordadas em [Melo00].

Para concluir este capítulo é importante referir que a síntese de uma unidade de controlo hierárquica baseada numa das arquitecturas apresentadas no capítulo 3

corresponde a uma extensão do processo aqui discutido, uma vez que é necessário gerar alguns sinais adicionais (ex. sinais para controlo da pilha de memória) [Rocha99]. No entanto, este assunto não é aqui abordado porque a arquitectura concebida e apresentada no próximo capítulo permite simplificar este processo.

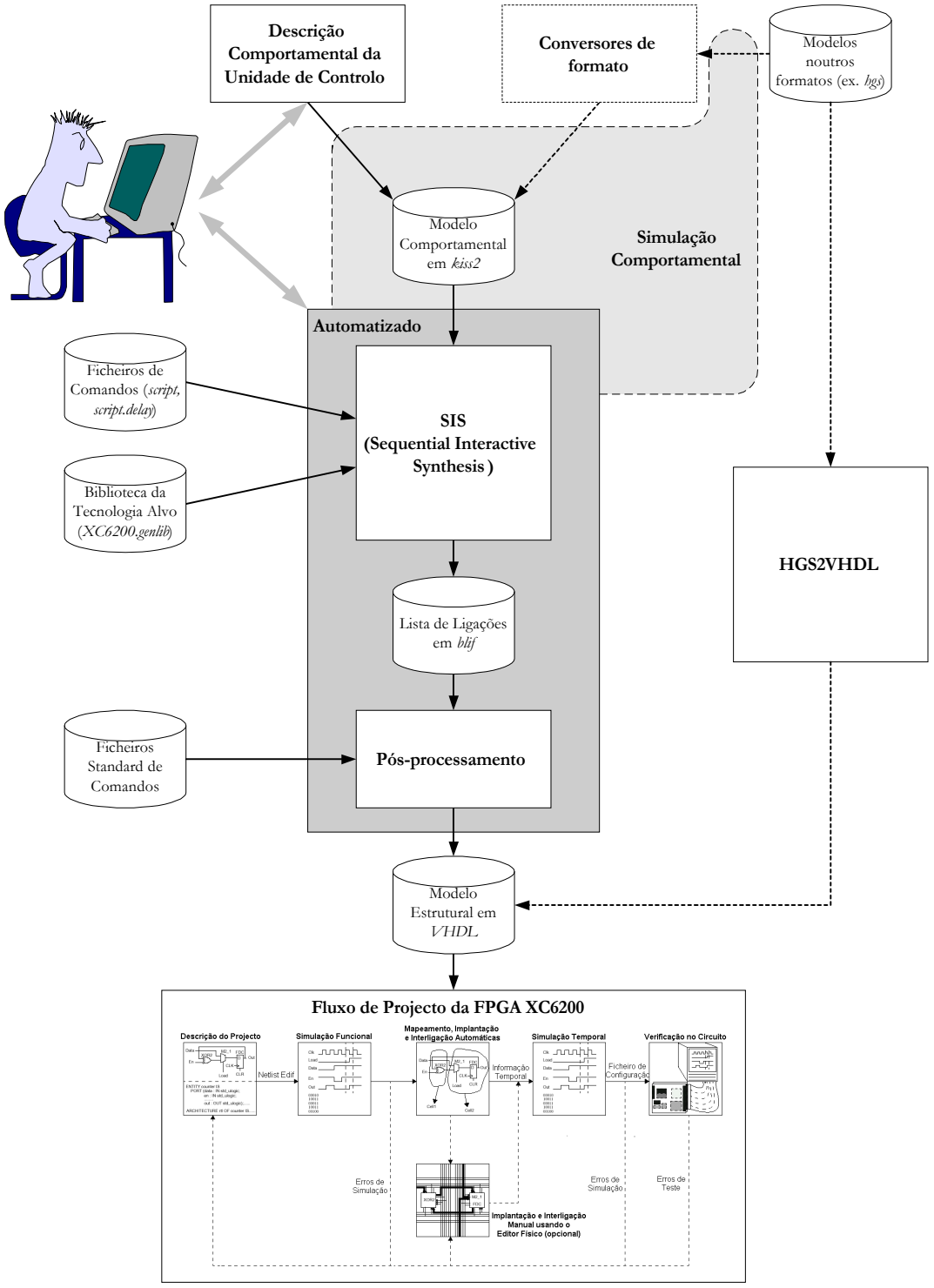


Figura 5.31 – Fluxo de projecto baseado no SIS para unidades de controlo destinadas a implementar em dispositivos da família de FPGAs XC6200 da Xilinx.

6 Projecto de Unidades de Controlo Virtuais

Sumário

Este capítulo é dedicado ao projecto de unidades de controlo virtuais a implementar na FPGA XC6200 da Xilinx. Este tipo de circuitos pode ser utilizado em sistemas reconfiguráveis, sempre que for necessário tornar uma unidade de controlo flexível e extensível, ou efectuar a sua construção com recursos de hardware limitados. Os domínios de aplicação e o seu fluxo de projecto são apresentados na parte inicial deste capítulo. A placa de desenvolvimento *FireFly*™ utilizada neste trabalho é também descrita de forma resumida.

O passos de projecto aqui abordados são a especificação, a síntese e a implementação. O método adoptado para a especificação utiliza os esquemas de grafos hierárquicos como formalismo de descrição comportamental. A síntese é baseada em codificação de estados *one-hot*, uma vez que o dispositivo utilizado é de granulosidade fina e possui um elevado número de flip-flops, sendo possível com esta técnica alcançar uma utilização eficiente das suas células. A implementação é baseada numa estrutura predefinida, parametrizável e optimizada para a FPGA XC6200. Esta estrutura simplifica o projecto de unidades de controlo virtuais e a sua reconfiguração através do estabelecimento de algumas restrições. Além disso, é perfeitamente escalável, podendo ser utilizada para construir circuitos de complexidade variável. O método de sincronização desenvolvido é mais complexo do que o apresentado atrás, mas permite melhorar o desempenho do circuito através da optimização do caso mais frequente durante o seu funcionamento, que são as transições não hierárquicas.

Na última parte deste capítulo é apresentada uma biblioteca de componentes descritos em VHDL que se destina a complementar as bibliotecas fornecidas pela Xilinx juntamente com as ferramentas de desenvolvimento da família XC6200. A maior parte desde componentes são parametrizáveis o que significa que algumas características podem ser facilmente especificadas durante a sua instanciação. Além disso, alguns deles podem substituir com vantagem os disponibilizados pela Xilinx.

6.1 Introdução

O aparecimento na última década de dispositivos lógicos de elevada capacidade e programáveis pelo utilizador abriu caminho ao surgimento de uma nova classe de circuitos: os sistemas reconfiguráveis. Estes podem ser usados para construir sistemas computacionais modificáveis e permitem combinar as vantagens de uma solução programável com o elevado desempenho de uma implementação em hardware.

Outra utilização possível para os sistemas reconfiguráveis é a construção de circuitos em que a quantidade de recursos de hardware disponíveis para uma implementação integral não seja suficiente e onde nem todos os sub-sistemas sejam necessários em simultâneo, sendo portanto preferível optar por uma implementação parcial em conjunto com a reconfiguração dinâmica do sistema durante a sua operação. Neste caso, devido à analogia com os sistemas de memória virtual, os sistemas reconfiguráveis podem também ser designados por sistemas de hardware virtual e os respectivos circuitos de controlo por unidades de controlo virtuais.

Ao contrário dos sistemas de hardware ou software tradicionais que estão optimizados para uma dada classe de aplicações, os sistemas reconfiguráveis têm também a vantagem de poderem ser optimizados para cada instância de uma aplicação, o que pode resultar em ganhos ao nível da área e desempenho do circuito [SinHogMcA96]. As aplicações dos sistemas reconfiguráveis têm sido apresentadas em algumas das conferências mais importantes da especialidade [FPL98, FPL99, FCCM99, FPGA99]. Contudo, as suas potencialidades não foram ainda devidamente aproveitadas devido à inexistência de dispositivos lógicos programáveis verdadeiramente adequados e à escassez de ferramentas de projecto apropriadas [Mayer97, AdaBam99].

Alguns dos domínios de aplicação mais relevantes são as aplicações multimédia [SinLeeLuKurBagFil98], o processamento de sinal [ZhaPap96, MalFraAlvAmo98], o processamento de imagem e de vídeo [SinBel94, WooTraHer98, AdaRoeBam99], as redes neuronais [EldHut94, Pam96], a geração de imagens em formato *PostScript* ™ [SinPatBurDal97, MacPatSin99], os processadores com um conjunto de instruções modificável dinamicamente [WirHut95], o reconhecimento de padrões [SidMeiPra99a], os algoritmos genéticos [GraNel95, SidMeiPra99b], a construção de *plug-ins* para aplicações de software [LudSloSin99], entre outros.

No entanto, os sistemas reconfiguráveis também possuem desvantagens. Para além de exigirem uma forma de raciocínio diferente e novas metodologias de projecto, a sua maior desvantagem está relacionada com o tempo necessário para efectuar a reconfiguração. Se durante esta não for realizado nenhum processamento útil, o desempenho do sistema é prejudicado, podendo anular todas as vantagens resultantes da utilização de uma implementação flexível em hardware, proporcionada por este novo paradigma de projecto de sistemas digitais.

A modificação de um sistema reconfigurável pode consistir tanto na alteração da(s) sua(s) unidades(s) de execução como da(s) unidades(s) de controlo, ou ambas. A complexidade da(s) unidade(s) de controlo eventualmente existentes em cada um destes sistemas é bastante variável, podendo em certos casos até nem existir. No entanto, a sua presença é praticamente obrigatória nas situações em que for necessário

realizar sequências complexas de operações que dependem de eventos em sinais de entrada ou de condições internas do circuito.

Como as unidades de controlo são específicas de cada projecto, usualmente pouco modulares e bastante irregulares, a aplicação das arquitecturas tradicionais às variantes virtuais não pode ser realizada directamente, uma vez que torna a sua reconfiguração parcial bastante difícil, senão impossível. Assim, é necessário estabelecer um conjunto de restrições e desenvolver arquitecturas, que sem comprometerem a generalidade do circuito, simplifiquem o seu projecto e reconfiguração.

O objectivo deste capítulo é precisamente a apresentação de uma arquitectura que permita atingir estes objectivos. Tal como já foi explicado no capítulo 3, os modelos e arquitecturas utilizadas no projecto de unidades de controlo virtuais devem ser preferencialmente hierárquicos, pelo que o formalismo utilizado na sua especificação deve suportar a realização de descrições multi-nível. Por outro lado, as técnicas de síntese e as arquitecturas de implementação devem ser adequadas às características do dispositivo alvo.

Devido à necessidade de reprogramabilidade do circuito, os dispositivos normalmente utilizados para implementação destes sistemas são as FPGAs reconfiguráveis dinamicamente, das quais a arquitectura XC6200 da Xilinx utilizada neste trabalho é provavelmente o exemplo mais conhecido. Tal como já foi referido no capítulo 2, esta FPGA é também reconfigurável parcialmente. Além disso, é de contexto único o que significa que só possui um elemento de configuração (célula SRAM) por cada elemento configurável (flip-flop ou multiplexador), ou seja, tem apenas uma página de configuração. Por oposição, um dispositivo de contexto múltiplo possui várias páginas de configuração, estando em cada momento activa apenas uma.

Um dos métodos que tem sido apontado para diminuir o tempo de reconfiguração necessário nos sistemas reconfiguráveis e nas unidades de controlo virtuais consiste na realização em simultâneo de configuração e processamento. Isto pode ser conseguido com a utilização de dispositivos de contexto único reconfiguráveis parcialmente ou com dispositivos de contexto múltiplo. Como no segundo caso a comutação entre páginas de configuração pode ser realizada rapidamente, as inactivas podem ser reconfiguradas ao mesmo tempo que as unidades funcionais ou células do dispositivo controladas pela página activa efectuam trabalho útil.

Outro método proposto para reduzir o tempo de reconfiguração baseia-se na compressão prévia da informação de configuração e na sua transferência e descompressão por hardware dedicado no interior do dispositivo [Hauck98, LiHau99].

Neste capítulo, não são abordadas as etapas de verificação, depuração e teste de unidades de controlo virtuais e a etapa de especificação é tratada apenas superficialmente, uma vez que já foram objecto de estudo em [Melo00]. O método de especificação utilizado baseia-se nos esquemas de grafos hierárquicos apresentados no capítulo 4. Este método permite especificar hierarquicamente qualquer algoritmo de controlo e suporta a descrição de recursividade. A partição do algoritmo em sub-

algoritmos necessária à virtualização resulta naturalmente da especificação baseada em HGSs.

Em [Sklyarov99] é apresentado um exemplo de aplicação das unidades de controlo virtuais para implementação de operações em árvores binárias (inserção, remoção, procura, etc.). Este exemplo é interessante porque permite tirar partido simultaneamente da especificação hierárquica e recursiva destas operações e da implementação virtual, uma vez que somente as operações necessárias num dado instante precisam de estar carregadas em hardware.

Esta abordagem pode também ser utilizada nos casos em que se pretende construir um circuito ou sistema de controlo flexível e extensível mas com requisitos de desempenho que não podem ser cumpridos com as arquitecturas de software tradicionais. Além disso, como os processadores de uso geral estão normalmente mais vocacionados para a manipulação de palavras multi-bit, podem ser altamente ineficientes para operar em sinais de um só bit, os quais são a generalidade das entradas e saídas de um circuito ou sistema de controlo digital.

Como veremos ao longo deste capítulo, a arquitectura concebida pode ser encarada como uma estrutura predefinida e orientada para a implementação de unidades de controlo complexas, hierárquicas, flexíveis, extensíveis e reconfiguráveis. A reutilização pode ser conseguida de duas maneiras distintas: ao nível comportamental, em que cada um dos sub-algoritmos pode ser usado em diferentes projectos, e ao nível físico noutros circuitos que utilizem a mesma estrutura base.

O fluxo de projecto de unidades de controlo virtuais a implementar na família de FPGAs XC6200 está ilustrado na Figura 6.1. A partir de uma descrição em HGSs realizada pelo utilizador é gerado um ou vários ficheiros que contêm a descrição de cada sub-algoritmo. A etapa seguinte consiste na conversão de cada sub-algoritmo numa descrição em VHDL estrutural. Este processo de síntese está documentado em [LauSkl99]. Seguidamente o código gerado deve ser compilado e a lista de ligações resultante processada pelas ferramentas de projecto físico da FPGA XC6200 de acordo com o fluxo de projecto apresentado no capítulo 2.

De todos os ficheiros produzidos pelas ferramentas de implementação apenas o de configuração (*.cal) é utilizado, uma vez que cada sub-algoritmo é processado individualmente pelas ferramentas de síntese e implementação. No entanto, para assegurar que o carregamento de uma configuração de um sub-algoritmo afecta apenas as células que lhe dizem respeito, é necessário realizar a filtragem deste ficheiro. Os métodos que o realizam estão implementados na biblioteca de classes apresentada no próximo capítulo. Para acelerar a leitura do ficheiro de configuração, isto é, evitar a interpretação do ficheiro de texto gerado pela aplicação XACT 6000, a configuração é armazenada num ficheiro binário. Este em conjunto com o ficheiro binário que possui a configuração da estrutura predefinida que implementa a arquitectura desenvolvida são utilizados para efeitos de depuração ou execução da unidade de controlo.

A depuração é realizada pela aplicação *GraphBuilder* [Melo00] enquanto a execução é suportada por um pequeno utilitário, desenvolvido para o efeito (*FireFly™ Manager*) [OliMelSkl99]. Esta aplicação permite também configurar todos os aspectos da placa de desenvolvimento utilizada neste trabalho e que será descrita na secção 6.1.1.

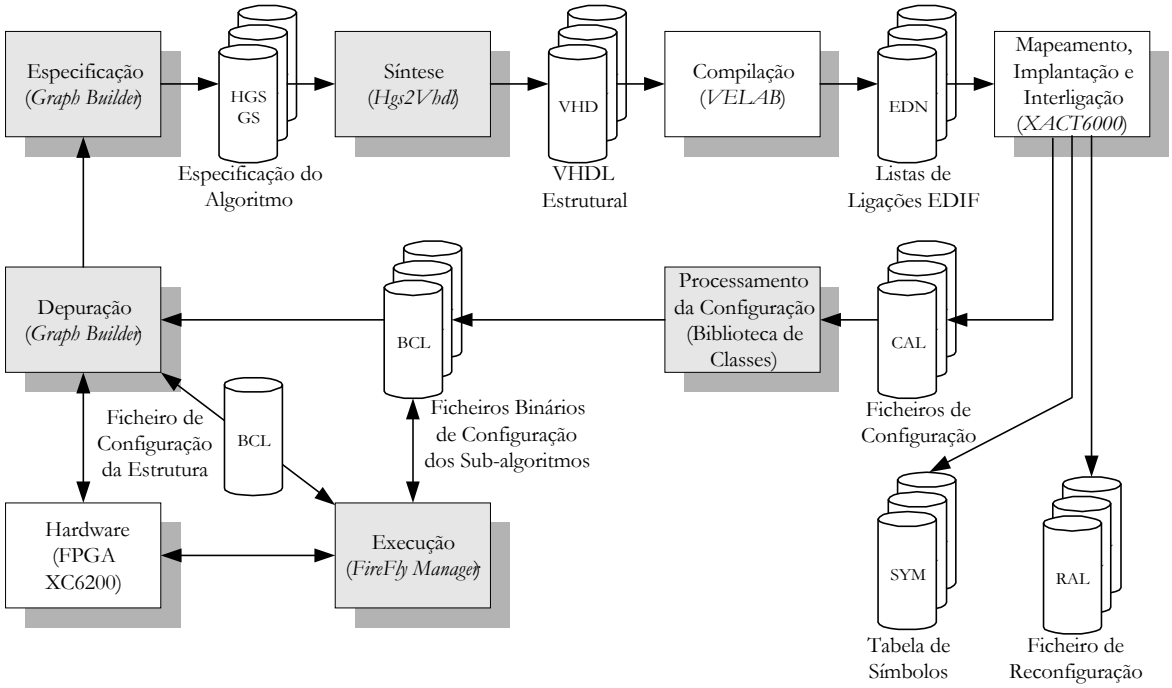


Figura 6.1 – Fluxo de projecto para unidades de controlo virtuais baseadas na FPGA XC6200.

6.1.1 Plataforma de Desenvolvimento

Neste trabalho foi utilizada uma placa de desenvolvimento adquirida à *Annapolis Micro Systems* e cuja designação comercial é *FireFly Development System*™ [NisGuc97]. Esta placa destina-se a ser ligada a um computador hospedeiro através do barramento PCI (*Peripheral Component Interconnect*). A Figura 6.2 ilustra o seu diagrama de blocos, cujos componentes principais são:

- Uma FPGA reconfigurável dinâmica e parcialmente XC6216/XC6264;
- Uma FPGA XC4013 da Xilinx para implementação do interface com o barramento PCI;
- 512/2048 Kbytes de RAM estática divididos em dois bancos individualmente controláveis pela FPGA XC6200 ou pelo computador hospedeiro através da FPGA XC4013;
- Conectores de entrada/saída para interface com o exterior e para efeitos de expansão;
- Outros componentes para geração de sinais de relógio e controlo do consumo de potência da FPGA XC6200.

A memória de configuração da FPGA XC6200 é mapeada no espaço de endereçamento do computador hospedeiro, o que possibilita o acesso a todas as suas funcionalidades (as mais importantes foram descritas no capítulo 2 desta dissertação).

Esta placa pode ser programada para gerar interrupções para o computador hospedeiro nas seguintes três situações:

- No final de um valor programado de ciclos de relógio;

- Quando for ultrapassado o limite preestabelecido do consumo de potência da FPGA XC6200;
- Sempre que necessário a partir da FPGA XC6200, podendo esta interrupção ser utilizada, por exemplo, para iniciar um pedido de reconfiguração.

Na Figura 6.3 encontra-se a fotografia da placa de desenvolvimento *FireFly*™, onde são visíveis todos os componentes acima referidos.

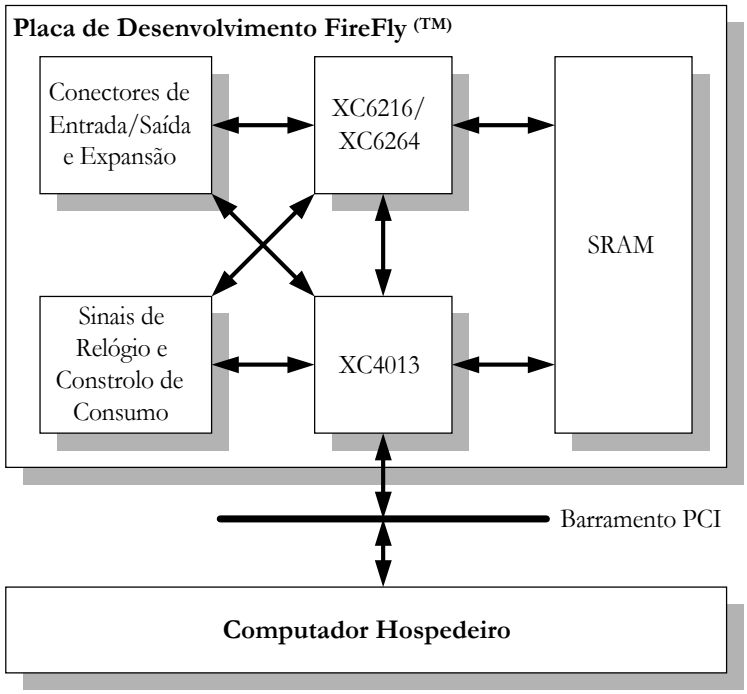


Figura 6.2 – Diagrama de blocos da placa de desenvolvimento *FireFly*™.

Figura 6.3 – Vista da implantação de componentes da placa de desenvolvimento *FireFly*™.

6.2 Especificação

Os Esquemas de Grafos (*GSs – Graph Schemes*) e suas variantes, em particular os hierárquicos (HGSs), foram o formalismo adoptado para especificar unidades de controlo virtuais. Esta escolha foi motivada pelas seguintes características:

- Proporcionam uma descrição algorítmica;
- Permitem a realização de descrições hierárquicas (HGSs);
- Suportam a especificação de operações paralelas (PHGSs);
- Possibilitam a descrição de algoritmos recursivos (HGSs);
- São adequados para especificar operações virtuais (Figura 6.4), as quais podem possuir várias implementações que são escolhidas dinamicamente.

A descrição de hierarquia e paralelismo são também duas das potencialidades dos *Statecharts*. No entanto, a escolha não recaiu sobre os últimos uma vez que os restantes requisitos são também muito importantes.

Contudo, os GSs e suas variantes têm relativamente aos *Statecharts* algumas limitações em particular ao nível das transições de estados e da sincronização. Apesar disso, este problema pode ser ultrapassado com algumas extensões aos GSs.

De notar que alguns destes conceitos foram já discutidos no capítulo 4, tendo sido aqui incluídos para reunir num único sítio os aspectos mais importantes do projecto de unidades de controlo virtuais.

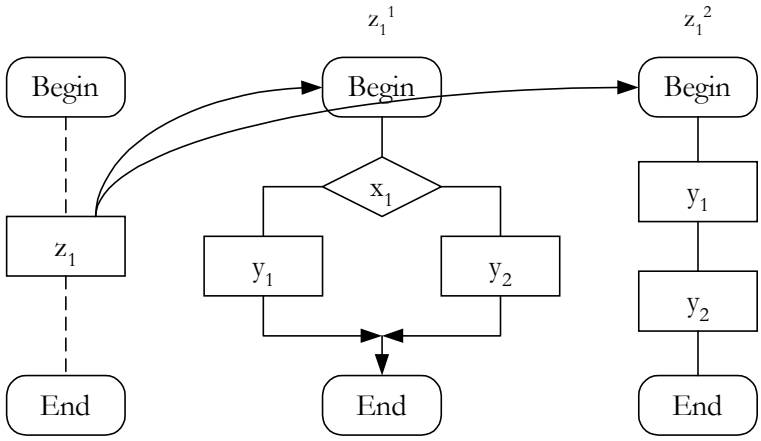


Figura 6.4 – Especificação de operações virtuais com HGSs.

6.3 Síntese

Tal como já foi discutido no capítulo anterior, a síntese de unidades de controlo destinadas à implementação em FPGAs com arquitecturas de granulosidade fina e com um elevado número de flip-flops pode utilizar com vantagens a codificação de estados *one-hot*. Os benefícios estão relacionados com a simplicidade do processo de síntese e com uma utilização mais eficiente dos recursos lógicos do dispositivo. Além disso, esta técnica de codificação de estados proporciona um método de conversão directa entre a

especificação de uma unidade de controlo baseada em HGSs e o respectivo circuito, remetendo todas as optimizações para o nível comportamental.

Estas optimizações são discutidas em [Baranov94, Baranov98, Melo00] e consistem na redução do número de nodos operacionais e condicionais da descrição e na minimização do número de condições lógicas e de microoperações do circuito. A redução do número de nodos baseia-se em técnicas de simplificação lógica e na determinação de estados equivalentes. A minimização de condições lógicas assenta num possível conhecimento prévio do valor das entradas do circuito nos seus vários estados. Por último, a minimização do número de microoperações baseia-se nas relações de dependência que eventualmente possam existir entre elas.

A Figura 6.5 resume as regras utilizadas na síntese de unidades de controlo que utilizam codificação de estados *one-hot* e que já foram discutidas no capítulo anterior.

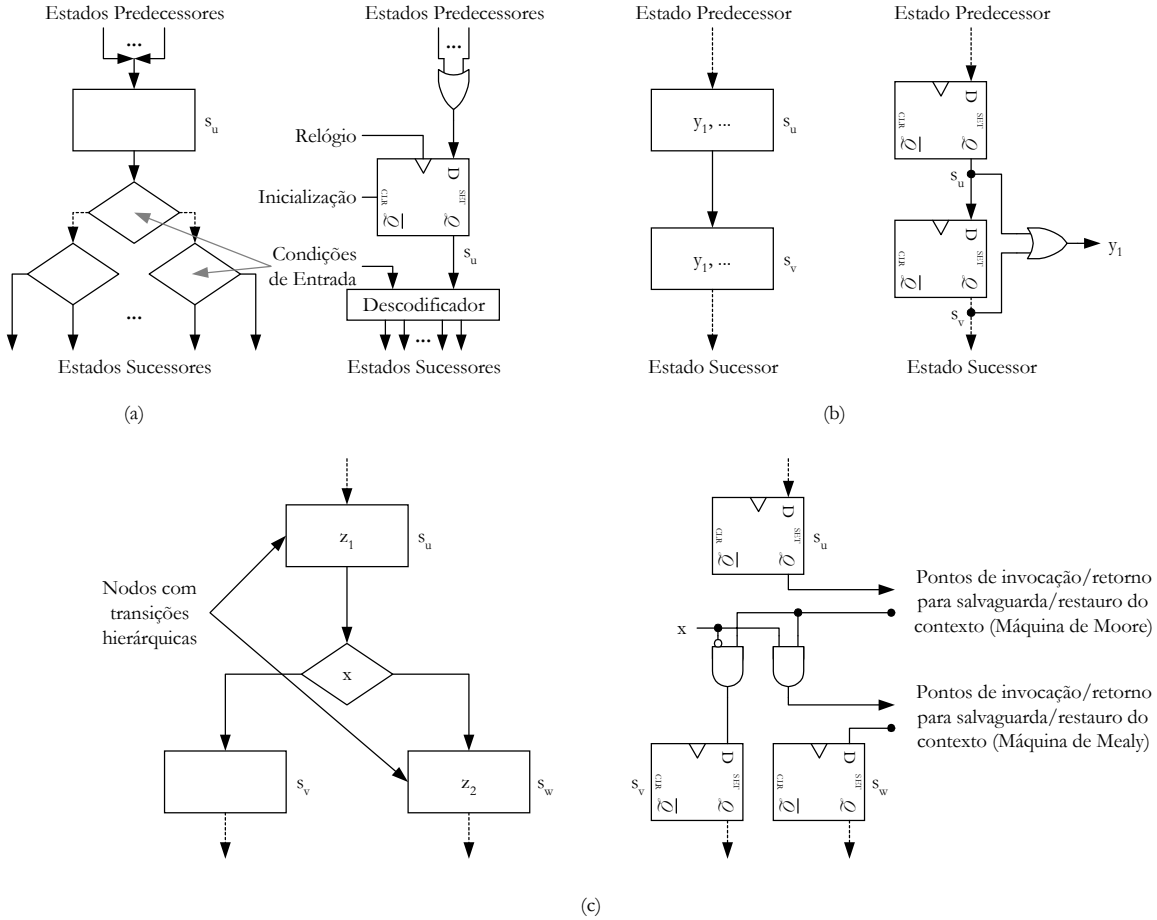


Figura 6.5 – Síntese de unidades de controlo hierárquicas baseadas em codificação de estados *one-hot* a partir de uma especificação em HGSs.

A Figura 6.5 (a) ilustra simultaneamente a conversão de um conjunto de nodos condicionais num descodificador ou desmultiplexador e a utilização de portas OR de *N* entradas para implementar a convergência de *N* caminhos para o mesmo nodo de um HGS. Por outro lado a Figura 6.5 (b) exemplifica a determinação de uma saída activada em dois estados utilizando uma porta lógica OR de 2 entradas.

Finalmente, a Figura 6.5 (c) apresenta um método de síntese que pode ser utilizado no caso de transições hierárquicas em que a cadeia de flip-flops e portas lógicas é interrompida nos estados ou transições em que ocorram invocações/retornos de sub-algoritmos. Isto deve ser feito no caso de uma implementação hierárquica para salvaguardar/restaurar o estado interrompido na/da pilha de memória. Desta forma é possível executar o novo sub-algoritmo num novo nível hierárquico e permite no caso de algoritmos recursivos a reutilização do mesmo circuito em diferentes níveis da hierarquia.

Enquanto numa máquina hierárquica de Moore os pontos de invocação estão sempre situados a seguir a um flip-flop, nas máquinas de Mealy situam-se após as portas lógicas ou descodificadores que os sucedem.

6.4 Implementação

Um dos objectivos principais deste trabalho é o desenvolvimento de uma arquitectura para implementação de unidades de controlo virtuais. A abordagem adoptada consiste numa estrutura predefinida, parametrizável e composta por vários tipos de componentes. O diagrama de blocos da arquitectura desenvolvida encontra-se na Figura 6.6. Os componentes podem ser divididos em quatro grupos principais: bloco de implementação, lógica de mapeamento, registos de entrada/saída e pilha de memória. A funcionalidade de alguns destes componentes, mais concretamente os pertencentes ao bloco de implementação, pode ser modificada por reconfiguração ou reprogramação.

A concepção desta arquitectura foi realizada de forma a simplificar o projecto deste tipo de circuitos, pelo que os componentes responsáveis pelos procedimentos necessários na generalidade das aplicações possuem funcionalidade e interligações fixas. Exemplos deste tipo de procedimentos são as transições hierárquicas, a verificação se o próximo sub-algoritmo a executar está carregado em hardware e a geração de sinais de sincronização. Os componentes que os realizam pertencem à lógica de mapeamento, ao circuito de sincronização e à pilha de memória.

No contexto desta arquitectura, um componente diz-se reprogramável se as alterações do seu comportamento puderem ser realizadas sem modificar a sua estrutura lógica. O conversor de código é um exemplo de um componente deste tipo, uma vez que, como veremos mais à frente, pode ser implementado com uma tabela ou memória cujas entradas são programáveis. Os componentes reconfiguráveis são mais versáteis, permitindo a modificação da sua estrutura lógica para alterar a sua funcionalidade. O elemento reconfigurável existente no interior do bloco de implementação é um exemplo deste tipo de componentes.

Para simplificar a reconfiguração dinâmica, as ligações entre os componentes reconfiguráveis e os restantes estão predefinidas.

A arquitectura desenvolvida foi implementada numa FPGA XC6216 existente na placa de desenvolvimento *FireFly*™ apresentada atrás. A sua descrição foi realizada em VHDL estrutural, tendo a parametrização sido realizada através de atributos. Mais à frente será mostrado um exemplo que ilustra a utilização de atributos para este efeito. A sua implementação foi realizada de acordo com o fluxo de projecto da XC6200 e

descrito no capítulo 2. Devido a limitações da FPGA utilizada, a pilha de memória foi colocada na memória externa existente na placa de desenvolvimento *FireFly*™.

Apesar da complexidade aparente da arquitectura, a sua estrutura é relativamente simples. Para o mostrar vamos dividi-la em duas partes, separadas pelo tracejado da Figura 6.6.

A parte superior opera essencialmente com estados, transições ordinárias e cálculo de saídas, sendo portanto análoga a uma unidade de controlo tradicional (não hierárquica). A função desempenhada pelo codificador e descodificador será analisada mais à frente, bastando para já referir que são componentes muito simples, sendo portanto a área por eles ocupada reduzida. A pilha de memória, os registos de entrada/saída e o circuito de sincronização são necessários em qualquer implementação hierárquica, pelo que relativamente às arquitecturas apresentadas no capítulo 3 não constituem uma sobrecarga adicional.

Os componentes abaixo da linha a tracejado são utilizados durante as transições hierárquicas e armazenam, na sua generalidade, códigos de sub-algoritmos. O conversor de código apesar de possuir uma função ligeiramente diferente da apresentada no capítulo 3, possui uma complexidade e estrutura semelhantes.

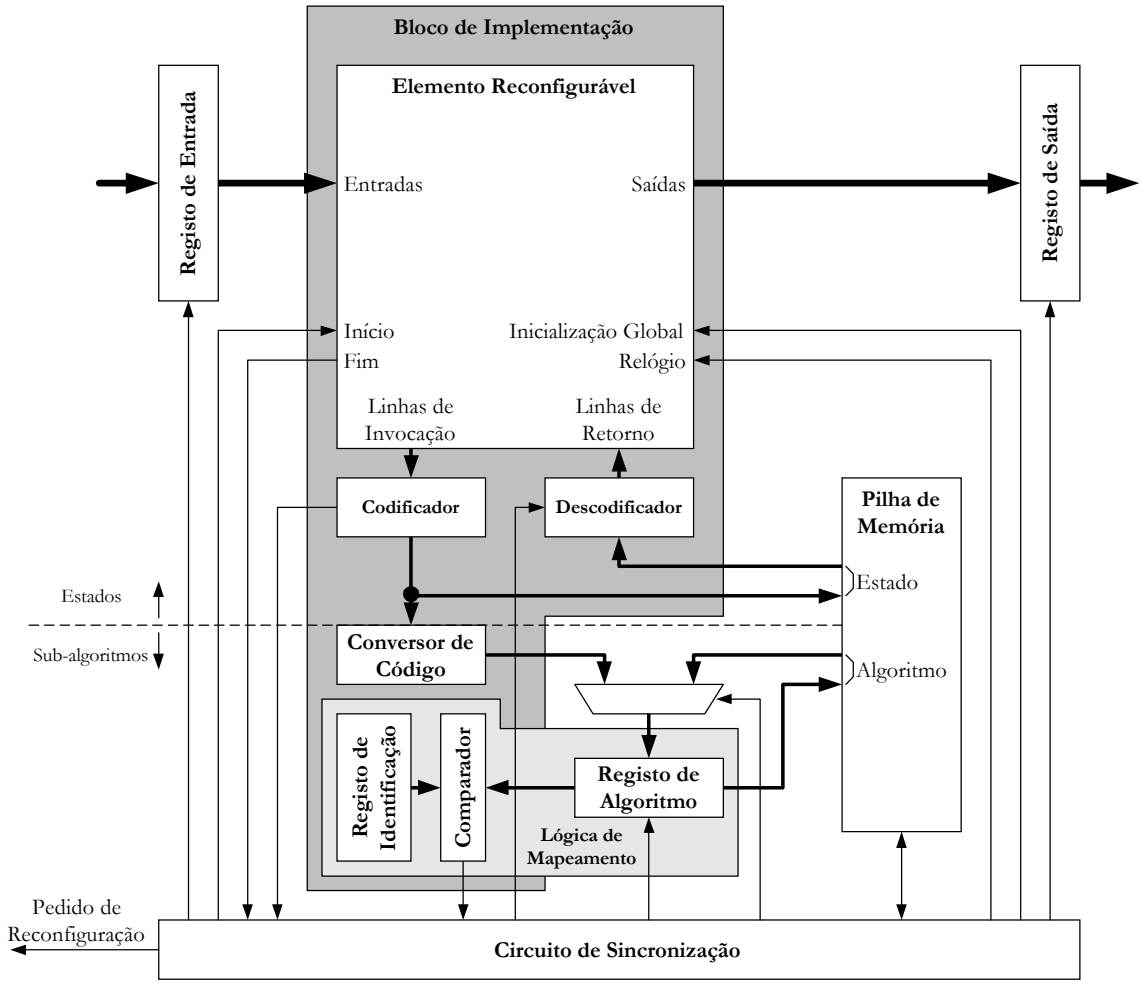


Figura 6.6 – Estrutura predefinida para implementação de unidades de controlo virtuais e hierárquicas baseadas em codificação de estados *one-hot*.

Finalmente, os restantes componentes (dois registos, um comparador e um multiplexador) possuem uma estrutura trivial e podem ser implementados eficientemente no dispositivo utilizado. As próximas subsecções descrevem cada um dos componentes da arquitectura desenvolvida.

6.4.1 Elemento Reconfigurável

O elemento reconfigurável define uma janela na matriz de células da FPGA que pode ser utilizada para implementar algoritmos de controlo modificáveis. Infelizmente, devido à forma como está definido o mapeamento entre os bits de configuração e os respectivos recursos controlados, não existe na arquitectura XC6200 nenhuma área rectangular da matriz de células da FPGA em que todos os recursos lógicos e de interligação sejam controlados por um conjunto de bits contíguos no espaço de endereçamento da memória de configuração. Este facto possui essencialmente duas desvantagens. Em primeiro lugar contribui negativamente para a eficiência com que os dados de configuração podem ser transferidos e carregados na FPGA. Em segundo lugar exige que se tomem precauções especiais para que sejam alterados apenas os endereços de memória relativos aos recursos que se quer reconfigurar, de forma a não corromper a configuração das restantes células.

O tamanho do elemento reconfigurável não é fixo, isto é, pode ser modificado por parametrização através da alteração do valor de atributos em VHDL. A altura e a largura da janela definem, respectivamente, o número máximo de linhas de entrada/saída e o número máximo de transições hierárquicas por sub-algoritmo.

Para uma dada aplicação, o tamanho da janela deve ser suficiente para implementar o algoritmo de maior complexidade esperado. No entanto, após a implementação da estrutura e a conclusão do projecto da unidade de controlo é sempre possível acomodar algoritmos mais complexos através da sua decomposição em algoritmos mais simples. Contudo, como veremos mais à frente na secção dedicada ao circuito de sincronização, este facto tem desvantagens ao nível da sobrecarga introduzida pelas transições hierárquicas, pelo que a forma como um algoritmo de controlo é decomposto em sub-algoritmos tem implicações importantes no desempenho do circuito.

Para simplificar a modificação dinâmica da unidade de controlo, mais concretamente do seu elemento reconfigurável, foram impostas algumas restrições ao seu formato e interface bem como no tipo de recursos de interligação que pode utilizar. As restrições impostas mais relevantes foram as seguintes:

- O número, tipo e localização dos portos de entrada e saída do bloco reconfigurável estão preestabelecidos;
- Os tipos de recursos de interligação utilizáveis pela estrutura e pelos sub-algoritmos sobre o bloco reconfigurável são disjuntos (ver Figura 6.7). Enquanto a primeira só pode usar recursos globais e de comprimento 16, cada sub-algoritmo só pode usar recursos locais e de comprimento 4 (para uma discussão sobre o tipo de recursos existentes na arquitectura XC6200 pode ser consultado o capítulo 2 desta dissertação);
- cada bloco reconfigurável deve estar contido em si próprio, isto é, todos os recursos de interligação utilizados por um sub-algoritmo devem ser

controlados por interruptores ou multiplexadores programáveis localizados no seu interior ou, quando muito, na sua extremidade.

De facto, as convenções estabelecidas para a utilização dos recursos de interligação e a arquitectura do dispositivo, em particular ao nível do comprimento das linhas de inicialização dos flip-flops (que estão alinhadas verticalmente ao longo de 16 células), faz com que o tamanho do elemento reconfigurável deva ser (4×n) × (16×n) células.

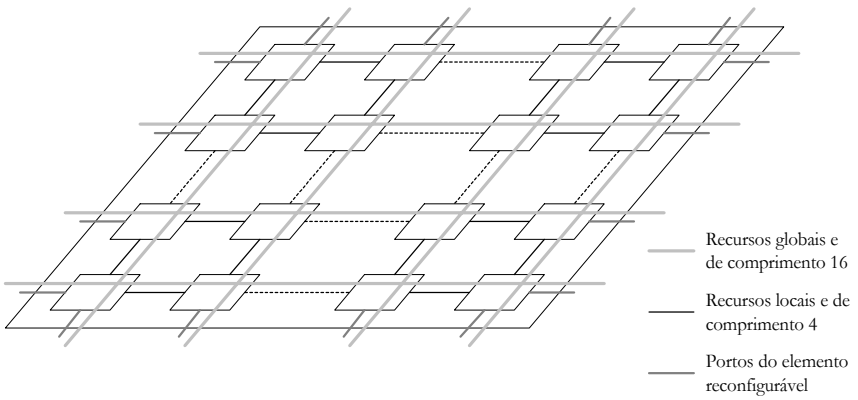


Figura 6.7 – Distribuição dos recursos de interligação entre a estrutura predefinida e os sub-algoritmos implementados no elemento reconfigurável.

Existe uma linha especial de saída do elemento reconfigurável que é realimentada para as entradas da mesma para permitir a implementação de funções lógicas descritas por um HGS.

Para assegurar que o circuito funciona correctamente e para simplificar a tarefa de implementação do mesmo, isto é, as subtarefas de implantação e interligação, as linhas de relógio e de inicialização são encaminhadas usando recursos de interligação globais, as quais são distribuídas pelo dispositivo numa configuração de baixos atrasos e desfasamentos. A Figura 6.8 ilustra o interface do elemento reconfigurável, o qual é constituído pelos seguintes sinais:

- Relógio – responsável pelas transições de estado da unidade de controlo;
- Inicialização Global – carrega o valor lógico "0" em todos os flip-flops utilizados no elemento reconfigurável;
- Entradas – linhas de entrada da unidade de controlo, ou seja, as condições lógicas das quais depende o fluxo de execução do sub-algoritmo de controlo;
- Saídas – linhas de saída da unidade de controlo, ou seja, as microoperações que dependem do estado actual e eventualmente das condições lógicas de entrada;
- Início – sinal activado pelo circuito de sincronização no princípio da execução de um sub-algoritmo de controlo;
- Fim – sinal activado pelo sub-algoritmo de controlo quando este termina a sua execução. Este sinal é aplicado ao circuito de sincronização que realiza as acções adequadas de forma a retornar ao sub-algoritmo invocador;

- Linhas de invocação – linhas de saída activadas pelo sub-algoritmo de controlo durante uma invocação hierárquica;
- Linhas de retorno – linhas de entrada activadas pelo circuito de sincronização durante um retorno hierárquico e de acordo com o estado do sub-algoritmo interrompido durante a invocação correspondente.

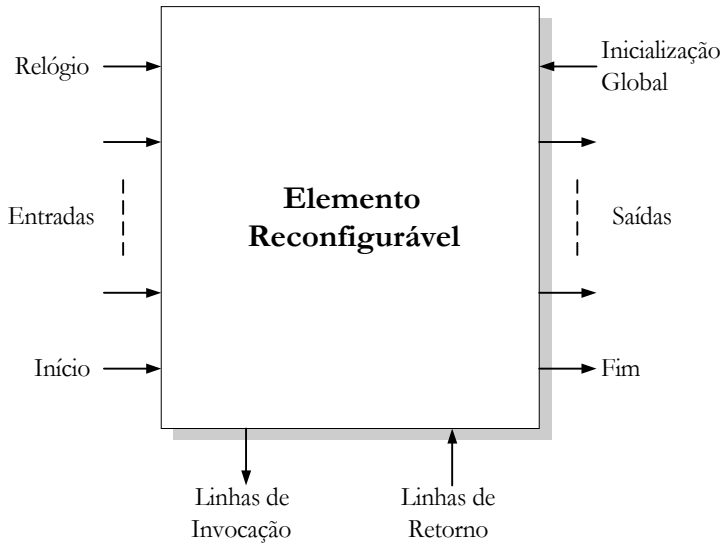


Figura 6.8 – Interface do elemento reconfigurável para implementação de sub-algoritmos modificáveis.

De facto, o interface podia ser simplificado se os sinais de "Início" e de "Inicialização Global" fossem combinados num só. No entanto, devido ao número limitado de linhas globais existentes na família XC6200, à possibilidade da existência de múltiplos elementos reconfiguráveis e à necessidade de inicializar cada um individualmente, tal não é possível. No caso de uma linha de inicialização por elemento reconfigurável, a sua activação provocaria o carregamento do valor lógico "0" em todos os flip-flops à excepção do primeiro e que representa o estado inicial, onde era carregado o valor lógico "1". Na realidade a situação ideal seria a existência de duas linhas globais e independentes para cada elemento, de forma a aplicar os sinais de relógio e de inicialização somente aos elementos necessários, reduzindo assim o consumo de potência global do circuito.

Na implementação actual e com os valores por defeito, cada elemento reconfigurável é composto por $12 \times 32 = 384$ células e possui 8 portos para transições hierárquicas (4 para invocação e 4 para retorno), 32 portos de entrada e 32 portos de saída. De acordo com os resultados publicados em [OliLauSkl98] um elemento reconfigurável com estas dimensões pode ser utilizado para implementar um sub-algoritmo de controlo descrito por um HGS composto por 100 a 150 nodos, consoante o tipo (condicional/operacional) e o número de saídas activadas em cada nodo operacional.

No entanto, tal como já foi referido o seu tamanho é parametrizável, o que permite o redimensionamento em função da complexidade dos sub-algoritmos de uma dada aplicação.

Descrição VHDL e Representação Física

A descrição do elemento reconfigurável foi feita em VHDL. A listagem completa deste módulo encontra-se no ficheiro *"Rec_Elem.vhd"* do anexo IV. A declaração da entidade é apresentada na Figura 6.9. Além dos sinais declarados na secção dos portos formais, a entidade contém também os valores que permitem variar a sua dimensão e que são declarados na zona dos parâmetros genéricos formais.

```
entity REC_ELEM is

        generic(N_IN_OUT        : integer := 16;
                N_CALLS         : integer := 4);

        port(GR_CLK             : in    STD_LOGIC;
             GR_CLR             : in    STD_LOGIC;
             START              : in    STD_LOGIC;
             FINISH             : out   STD_LOGIC;
             OH_CURR_ST         : out   STD_LOGIC_VECTOR((N_CALLS-1) downto 0);
             OH_PREV_ST         : in    STD_LOGIC_VECTOR((N_CALLS-1) downto 0);
             INPUTS             : in    STD_LOGIC_VECTOR((N_IN_OUT-1) downto 0);
             OUTPUTS            : out   STD_LOGIC_VECTOR((N_IN_OUT-1) downto 0));

end REC_ELEM;
```

Figura 6.9 – Listagem VHDL para definição da entidade que implementa o elemento reconfigurável.

A descrição VHDL foi processada de acordo com o fluxo de projecto da FPGA XC6200 e que foi apresentado no capítulo 2 desta dissertação, tendo-se obtido a representação física da Figura 6.10. Os pontos de entrada e saída de cada elemento reconfigurável são fixados com a ajuda de *buffers* e de componentes que fornecem um valor lógico fixo, neste caso "0".

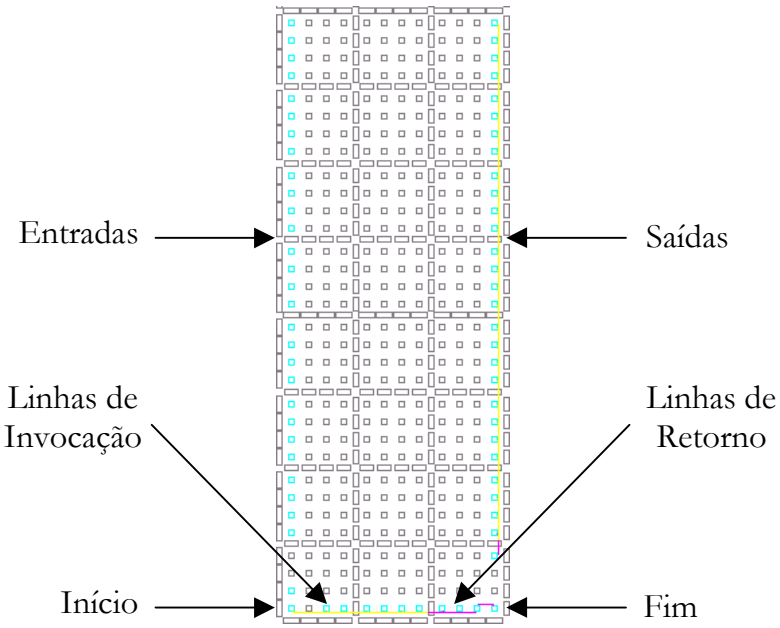


Figura 6.10 – Representação física do elemento reconfigurável após implementação numa FPGA XC6216 com a ferramenta XACT6000.

6.4.2 Codificador e Descodificador

Os sub-algoritmos implementados no elemento reconfigurável utilizam codificação de estados *one-hot*. Para reduzir o tamanho da pilha de memória é colocado entre as linhas de invocação e retorno do elemento reconfigurável um par codificador/descodificador que converte, durante as transições hierárquicas, os estados *one-hot* em binário e vice-versa (Figura 6.11). O codificador só está activo durante as invocações hierárquicas, e notifica o circuito de sincronização através do sinal "Invocação". O descodificador realiza a operação inversa, isto é, converte os estados binários em estados *one-hot* e é activado pelo circuito de sincronização durante o retorno de um sub-algoritmo através do sinal "Retorno".

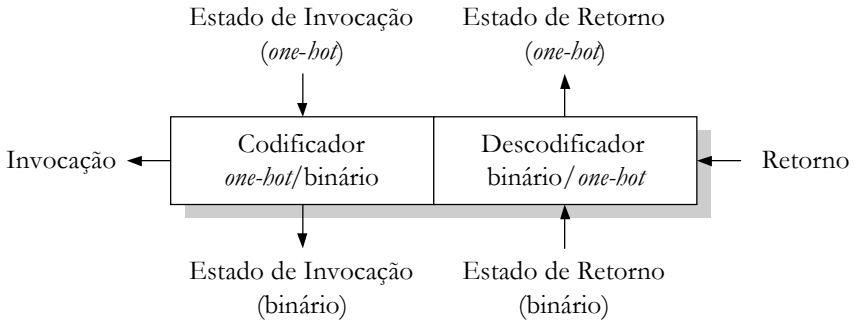


Figura 6.11 – Codificador e descodificador de estados *one-hot*/binário.

Descrição VHDL e Representação Física

As descrições do codificador e do descodificador estão embutidas no módulo VHDL "*Single_Block.vhd*" que é apresentado no anexo IV. O diagrama esquemático de ambos está ilustrado na Figura 6.12.

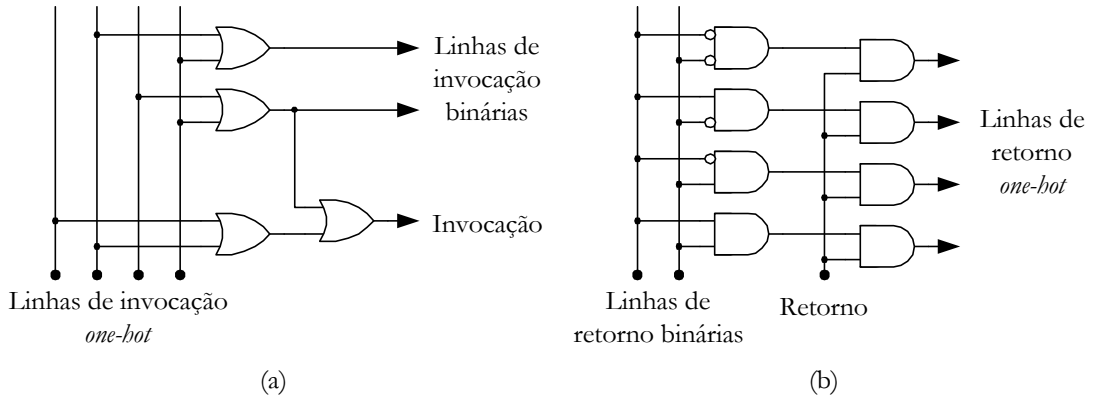


Figura 6.12 – Diagrama esquemático do (a) codificador e (b) descodificador de estados *one-hot*/binário.

Tal como no caso do elemento reconfigurável, após o processamento da descrição VHDL por parte das ferramentas de implementação, obtém-se a representação física da Figura 6.13.

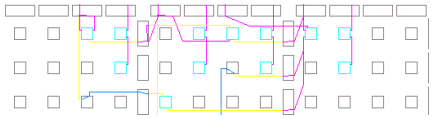


Figura 6.13 – Representação física do codificador e descodificador dos estados *one-hot*/binário e vice-versa.

6.4.3 Conversor de Código

O conversor de código é uma tabela de verdade que transforma o código binário de um estado de invocação no código do próximo sub-algoritmo a ser executado. O seu endereçamento é efectuado pelas saídas do codificador de estados *one-hot*/binário. Este componente permite programar de forma eficiente e flexível as invocações de um sub-algoritmo.

Do ponto de vista da unidade de controlo, este componente comporta-se como se fosse unicamente de leitura, sendo a sua reprogramação efectuada pela entidade que realiza a reconfiguração do circuito.

A sua estrutura é muito regular e necessita de um número reduzido de ciclos de relógio para programar todas as suas entradas, sem que para tal seja necessário configurar quaisquer recursos de interligação. No caso de quatro transições hierárquicas por sub-algoritmo e um número máximo de 64 sub-algoritmos, são necessárias apenas cinco operações de escrita (uma para programar o *Map Register* e as restantes para programar cada uma das entradas da tabela). O número de células necessárias depende da implementação ou não de capacidades de leitura a partir do interface de configuração (*read-back*). No primeiro caso são necessárias 42 células enquanto no segundo apenas 30.

Enquanto o conversor de código apresentado no contexto da arquitectura do controlador hierárquico reprogramável do capítulo 3 transformava o código do próximo sub-algoritmo a executar no seu estado inicial, este fornece o código do sub-algoritmo invocado com base no código do estado do sub-algoritmo invocador. Apesar da primeira abordagem ter vantagens ao nível da flexibilidade do circuito, uma vez que se pode programar transições entre quaisquer dois estados, a segunda alternativa torna os sub-algoritmos mais modulares e mais fáceis de desenvolver, modificar e reutilizar individualmente, o que permite alcançar três dos objectivos mais importantes deste trabalho.

Descrição VHDL e Representação Física

À semelhança do codificador/descodificador anteriores, este componente está instanciado no módulo "*Single_Block.vhd*" que é apresentado no anexo IV. A sua implementação é feita no módulo *"Preg4N.vhd"* da biblioteca VHDL de componentes que é descrita na secção 6.5 deste capítulo. A estrutura deste componente está representada na Figura 6.14.

Após a implementação do conversor de código, obtém-se a representação física da Figura 6.15. Os registos estão alinhados verticalmente e intercalados com os multiplexadores de forma a utilizar o melhor possível os recursos de interligação disponíveis e a simplificar a tarefa das ferramentas de implementação.

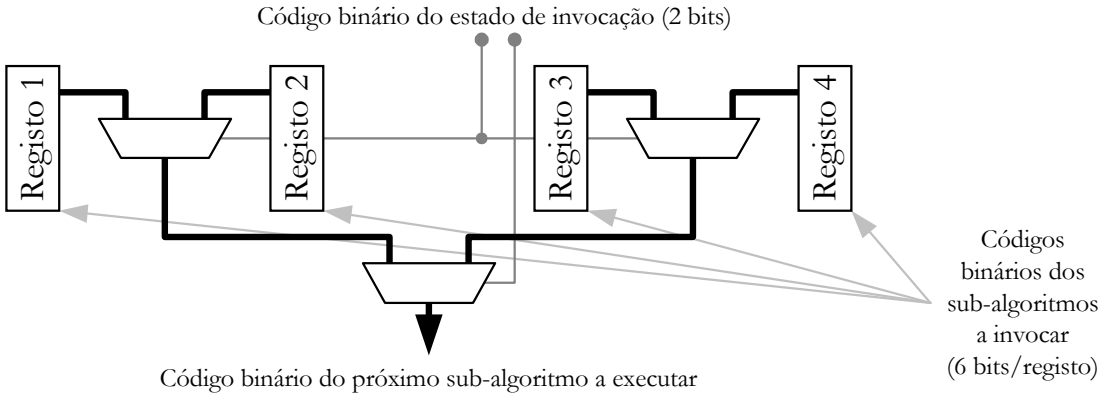


Figura 6.14 – Diagrama esquemático do conversor de código.

O espaçamento de uma célula entre os bits de cada registo e dos multiplexadores deve-se a limitações impostas pelo interface com a memória externa da placa de desenvolvimento e que como veremos mais à frente é utilizada para implementar a pilha de memória da unidade de controlo hierárquica.

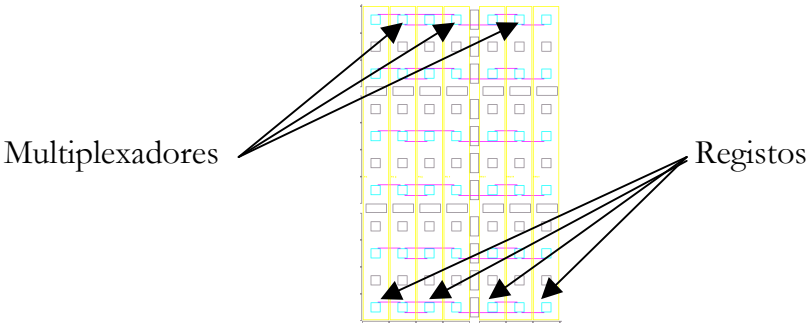


Figura 6.15 – Representação física do conversor de código.

6.4.4 Lógica de Mapeamento

A lógica de mapeamento é responsável por identificar o sub-algoritmo e o respectivo bloco activo, verificar durante transições hierárquicas se o próximo sub-algoritmo a executar se encontra carregado num dos elementos reconfiguráveis existentes e finalmente iniciar ou retomar a sua execução consoante se trate de uma invocação ou retorno, respectivamente.

Registo de Algoritmo

O Registo de Algoritmo (RA) é utilizado para armazenar o código do sub-algoritmo que está a ser executado. Independentemente do número de elementos reconfiguráveis existentes, só existe um registo deste tipo, uma vez que se assume a ausência de sub-algoritmos a executarem de forma concorrente.

O seu valor é carregado quer do conversor de código (durante a invocação de um sub-algoritmo) quer da pilha de memória (durante o retorno de um sub-algoritmo). A inicialização do circuito carrega o valor lógico "0" em todos os bits deste registo pelo que o sub-algoritmo principal deve possuir o código 0.

Registo de Identificação

O Registo de Identificação (RI) armazena o código do sub-algoritmo implementado num dado elemento reconfigurável. Ao contrário do RA, existe um RI para cada elemento reconfigurável. O seu valor é comparado com a saída do RA durante as transições hierárquicas. Se os valores forem iguais significa que o sub-algoritmo requisitado foi previamente carregado nesse elemento reconfigurável e a execução da unidade de controlo pode prosseguir. No caso oposto, o circuito de sincronização interrompe a execução até que o sub-algoritmo requisitado seja carregado. O circuito de sincronização efectua esta verificação antes de um sub-algoritmo iniciar a sua execução e sempre que a retomar após um retorno, permitindo desta forma a utilização de apenas um elemento reconfigurável.

A interligação dos diversos elementos que constituem a lógica de mapeamento é ilustrada na Figura 6.16. As saídas desta são aplicadas às entradas de selecção dos multiplexadores responsáveis pela comutação dos vários elementos reconfiguráveis da arquitectura.

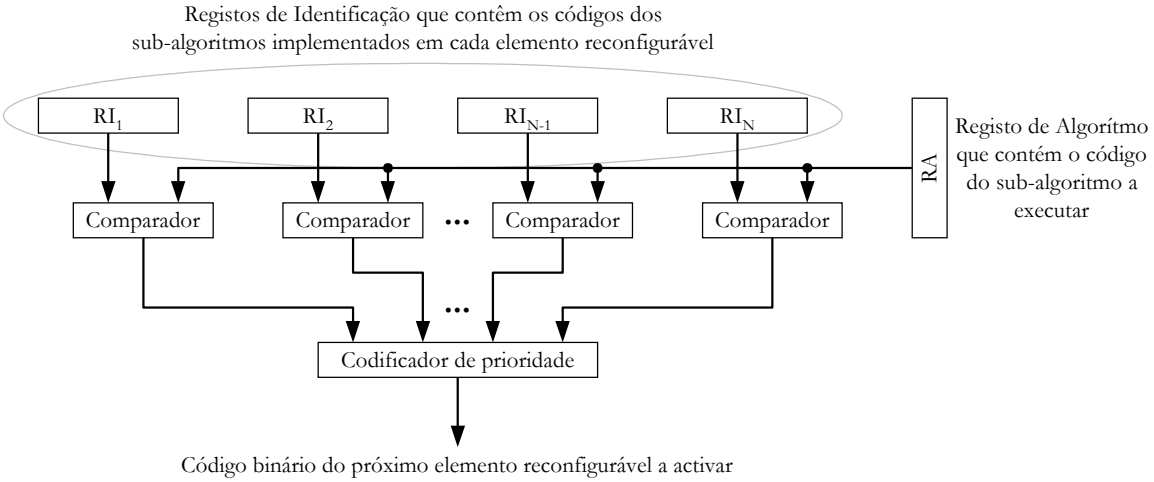


Figura 6.16 – Interligações entre os diversos elementos que constituem a lógica de mapeamento.

Na Figura 6.17 é mostrado um Bloco de Implementação (BI), o qual constitui a célula base usada na construção da estrutura predefinida para implementar unidades de controlo virtuais nas FPGAs da família XC6200. A sua descrição é feita no ficheiro "*Single_Block.vhd*" apresentado no anexo IV e é constituído pelos seguintes componentes:

- Um elemento reconfigurável para implementar os sub-algoritmos propriamente ditos;
- Um par codificador/descodificador para converter os estados *one-hot* em binário e vice-versa;
- Um conversor de código para transformar os estados de invocação nos códigos dos sub-algoritmos invocados;
- Um registo de identificação e respectivo comparador para verificar se o sub-algoritmo invocado se encontra carregado neste BI.

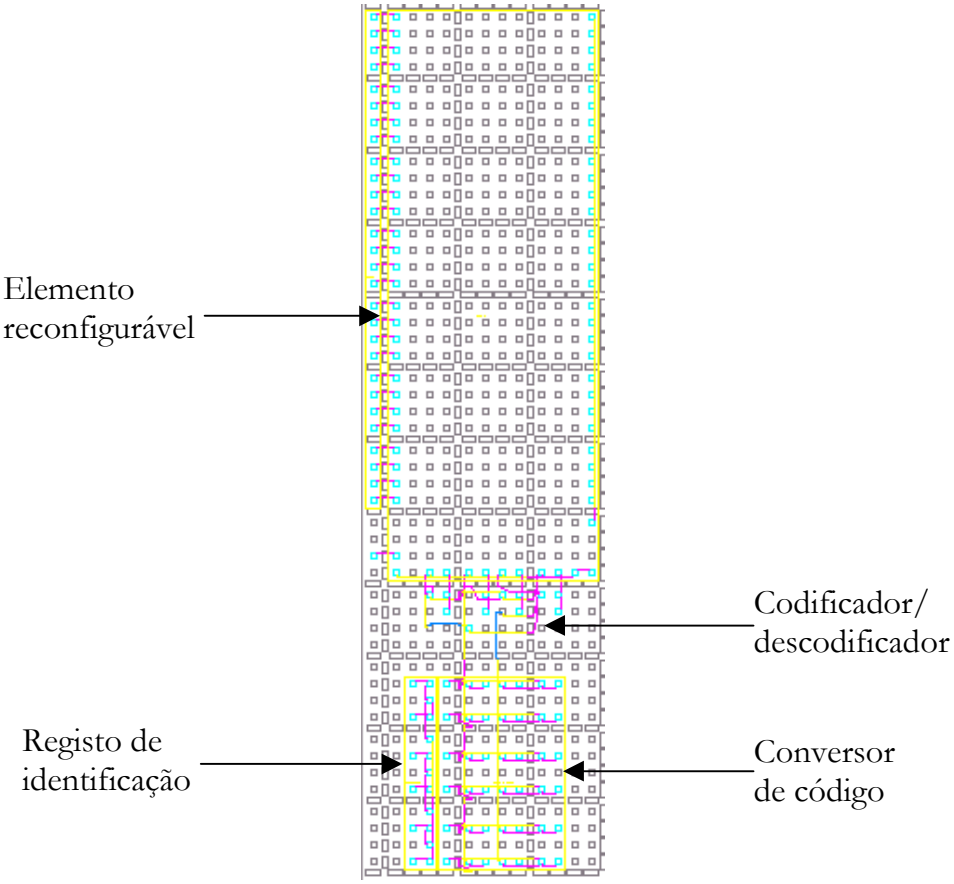


Figura 6.17 – Representação física de um bloco de implementação.

Assim, os dados para configuração de cada sub-algoritmo devem incluir:

- a sua identificação;
- o conteúdo do elemento reconfigurável;
- a informação para programação do conversor de código.

6.4.5 Pilha de Memória

Tal como foi discutido no capítulo 4, a pilha de memória é utilizada como elemento de armazenamento da unidade de controlo, substituindo o tradicional registo de estado de forma a implementar hierarquicamente algoritmos de controlo.

A pilha de memória só é utilizada durante as transições de estado hierárquicas. Todos os estados onde ocorram apenas transições ordinárias são armazenados nos flip-flops do elemento reconfigurável e são considerados um atributo interno de cada sub-algoritmo. Desta forma é possível melhorar o desempenho do circuito e diminuir a largura da pilha de memória.

Cada entrada ou registo da pilha de memória contém o código do sub-algoritmo interrompido, que é um identificador global, e o código do estado invocador, que é um identificador local. Em conjunto, estes dois campos identificam de forma única qualquer estado invocador e possibilitam o projecto individual de cada sub-algoritmo, o que pode simplificar consideravelmente as etapas de síntese e implementação. Além disso, tornam a unidade de controlo completamente extensível.

Assim, a largura, ou número de bits por registo da pilha de memória deve ser igual ou superior a $\lceil \log_2(P) \rceil + \lceil \log_2(V) \rceil$, em que P e V representam, respectivamente, o número máximo de sub-algoritmos da unidade de controlo e o número máximo de invocações hierárquicas por sub-algoritmo. Como o número máximo de sub-algoritmos estipulado foi 64, podendo cada um fazer 4 transições hierárquicas, cada entrada deve possuir 8 bits (1 *byte*).

O comprimento ou profundidade da pilha de memória deve ser maior ou igual ao número máximo possível de níveis hierárquicos. Neste caso optou-se por fixar este valor em 256, uma vez que para as experiências realizadas e para um grande número de aplicações, este é um valor mais do que suficiente e constitui actualmente uma quantidade de memória irrisória (256 *bytes*).

Quando ocorre uma situação de saturação da pilha, a execução da unidade de controlo é interrompida, sendo activado um bit de estado. Opcionalmente, pode também ser gerada uma interrupção para notificar o controlador de reconfiguração, neste caso uma aplicação de software, para que este leia o conteúdo da pilha de memória de forma a determinar a causa da saturação.

Descrição VHDL e Representação Física

A descrição da pilha de memória é feita no ficheiro "*Stack.vhd*" que se encontra no anexo IV. A definição da entidade é apresentada na Figura 6.18. Todos os sinais de entrada de um bit (*CLK, N_STACK_RD, STACK_WR, STACK_OPAD_EN, SP_INC, , SP_DEC* e *SP_RESET*) são controlados pelo circuito de sincronização. Os sinais multi-bit *STACK_IN* e *STACK_OUT* são, respectivamente, as entradas e saídas de dados. Finalmente, o sinal de estado *SP_LIMIT* é activado quando ocorrem situações de saturação da pilha de memória.

```
entity STACK is

       port(CLK              : in   STD_LOGIC;
            N_STACK_RD       : in   STD_LOGIC;
            STACK_WR         : in   STD_LOGIC;
            STACK_OPAD_EN    : in   STD_LOGIC;
            SP_INC           : in   STD_LOGIC;
            SP_DEC           : in   STD_LOGIC;
            SP_RESET         : in   STD_LOGIC;
            STACK_IN         : in   STD_LOGIC_VECTOR(7 downto 0);
            STACK_OUT        : out  STD_LOGIC_VECTOR(7 downto 0);
            SP_LIMIT         : out  STD_LOGIC);

end STACK;
```

Figura 6.18 – Listagem VHDL da definição da entidade que representa a pilha de memória.

Na Figura 6.19 encontra-se a representação física do controlador da pilha de memória. A memória propriamente dita não está implementada no interior da FPGA mas sim nos circuitos integrados SRAM existentes na placa de desenvolvimento utilizada. Consequentemente, a localização dos pinos indicada na Figura 6.19 é específica da plataforma utilizada. A abordagem da implementação dos bits de memória nas células da FPGA não foi a escolhida porque necessitava de um elevado número de células. Contudo, a solução adoptada também não é a mais desejável do ponto de vista do número de circuitos integrados necessários e desempenho do

sistema. Felizmente, as famílias de FPGAs mais recentes, como por exemplo a *Virtex* da Xilinx, possuem já blocos de memória com a dimensão apropriada para permitir implementar num único circuito integrado todos os componentes necessários à construção de uma unidade de controlo hierárquica. Como veremos no último capítulo desta dissertação, um dos possíveis pontos de trabalho futuro é a avaliação da adequação das características das novas famílias de FPGAs para a actualização da implementação realizada no âmbito deste trabalho.

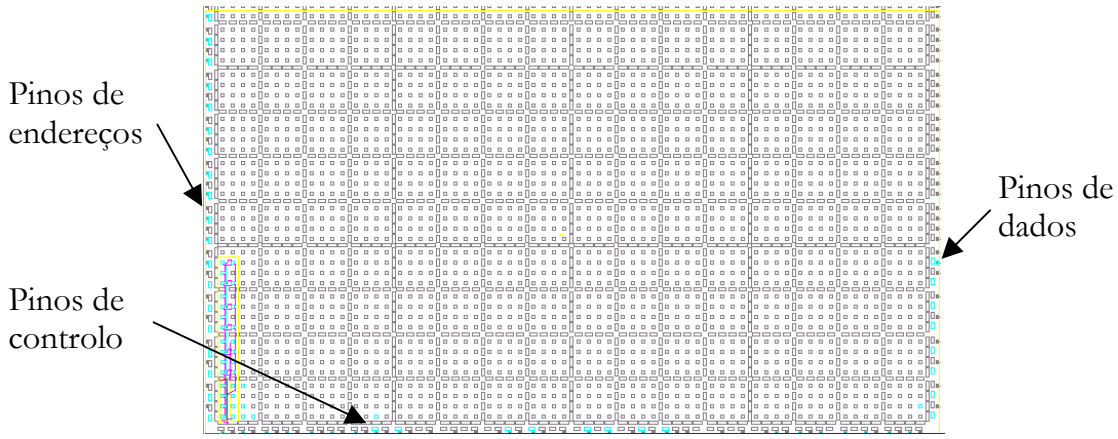


Figura 6.19 – Representação física do controlador da pilha de memória.

6.4.6 Circuito de Sincronização

O circuito de sincronização estabelece a sequência dos eventos da unidade de controlo e assegura o seu correcto funcionamento através do controlo de diversas operações, tais como a fixação das saídas, a salvaguarda e restauro do contexto necessários durante as transições hierárquicas e a monitorização de condições de excepção, como por exemplo a ausência de um sub-algoritmo necessário num dos elementos reconfiguráveis ou a saturação da pilha de memória.

Este circuito é ele próprio uma pequena unidade de controlo descrita por uma FSM ordinária (não hierárquica) e implementada usando codificação de estados *one-hot*. Ao contrário dos sub-algoritmos implementados nos elementos reconfiguráveis, pretende-se que a sua funcionalidade seja fixa para a generalidade das aplicações.

O circuito de sincronização activa indicadores no caso da ocorrência de uma das condições de excepção acima referidas. Além disso, está acoplado a uma unidade de geração de interrupções que pode ser utilizada para gerar notificações para a aplicação executada no computador hospedeiro que controla a reconfiguração do circuito. Para processar estas interrupções foi desenvolvido um controlador de dispositivo (*device driver*) que será apresentado no próximo capítulo, o qual permite que o processamento possa ser efectuado a partir de uma aplicação de utilizador.

Na Figura 6.20 é mostrada a especificação algorítmica simplificada do circuito de sincronização. Uma especificação mais detalhada com a indicação de todos os sinais activados é mostrada na Figura 6.21. Esta descrição é baseada nos Esquemas de Grafos apresentados no capítulo 4 desta dissertação com algumas extensões propostas em [OliSkl99] e que permitem a activação permanente (*Set*) e a desactivação (*Reset*) de

microoperações. A especificação de informação temporal proposta em [OliSkl99] não é utilizada assumindo-se que todos os estados possuem a mesma duração.

O diagrama da Figura 6.21 pode ser dividido nas seguintes partes: inicialização, verificação de sub-algoritmo, transição ordinária, invocação, retorno, saturação da pilha de memória e fim de execução.

Como se pode concluir da análise da Figura 6.21, uma transição não hierárquica necessita de dois ciclos de relógio, os quais são usados para realizar a transição de estado e fixar as microoperações de saída. As transições hierárquicas necessitam de quatro ciclos de relógio adicionais para salvaguardar ou restaurar o contexto e verificar se o próximo sub-algoritmo a executar está carregado num dos blocos reconfiguráveis. Desta forma pretende-se optimizar a situação mais frequente durante a execução da unidade de controlo que são as transições ordinárias.

Descrição VHDL e Representação Física

A descrição do circuito de sincronização encontra-se no módulo "*Sync.vhd*" apresentado no anexo IV desta dissertação. A função dos seus sinais de entrada e saída pode ser descrita resumidamente da seguinte forma:

- *CLK* – sinal de entrada de relógio do circuito de sincronização;
- RESET – sinal de entrada para inicialização do circuito de sincronização;
- *CALL* – sinal de entrada activado pelo codificador de estados durante uma invocação;
- *FINISH* – sinal de entrada activado pelo estado final de um sub-algoritmo simbolizando a sua terminação;
- *GR_OK* – sinal de entrada activado pela lógica de mapeamento que significa que o próximo sub-algoritmo a executar está carregado numa das células de implementação;
- *SP_LIMIT* – sinal de entrada activado pela pilha de memória no caso de saturação da mesma;
- *GR_FAULT_INT* – sinal de saída aplicado ao circuito gerador de interrupções, sendo activado no caso do próximo sub-algoritmo a executar não estar carregado em nenhuma das células de implementação;
- *STACK_OV_INT* – sinal de saída aplicado ao circuito gerador de interrupções, sendo activado no caso de saturação da pilha de memória;
- *GR_CLK* – sinal de saída aplicado aos elementos reconfiguráveis para sincronizar as transições de estado dos sub-algoritmos;
- *GR_CLR* – sinal de saída aplicado aos elementos reconfiguráveis para inicialização dos sub-algoritmos;
- *CURR_GR_CLK* – sinal de saída para controlo do carregamento do registo de algoritmo;
- *CURR_GR_CLR* – sinal de saída para inicialização do registo de algoritmo;
- *PREV_ST_CLK* – sinal de saída para controlo do carregamento de um registo auxiliar que armazena o estado anterior lido da pilha de memória;
- *STACK_WR* – sinal de saída que controla as operações de escrita da pilha de memória;

- *N_STACK_RD* – sinal de saída que controla as operações de leitura da pilha de memória;
- *STACK_OPAD_EN* – sinal de saída que activa os controladores do barramento de dados da memória;
- *SP_INC* – sinal de saída para incremento do ponteiro da pilha de memória;
- *SP_DEC* – sinal de saída para decremento do ponteiro da pilha de memória;
- *SP_RESET* – sinal de saída para inicialização do ponteiro da pilha de memória;
- *OUT_CLK* – sinal de saída para fixação das microoperações da unidade de controlo;
- *OUT_CLR* – sinal de saída para inicialização das microoperações da unidade de controlo;
- *GR_FAULT_FLAG* – sinal de saída activado enquanto o próximo sub-algoritmo a executar não estiver carregado numa das células de implementação;
- *STACK_OV_FLAG* – sinal de saída activado quando ocorrer a saturação da pilha de memória;
- *START* – sinal de saída activado no início da execução de um sub-algoritmo;
- *RET* – sinal de saída para activação do descodificador de endereços durante o retorno de um sub-algoritmo.

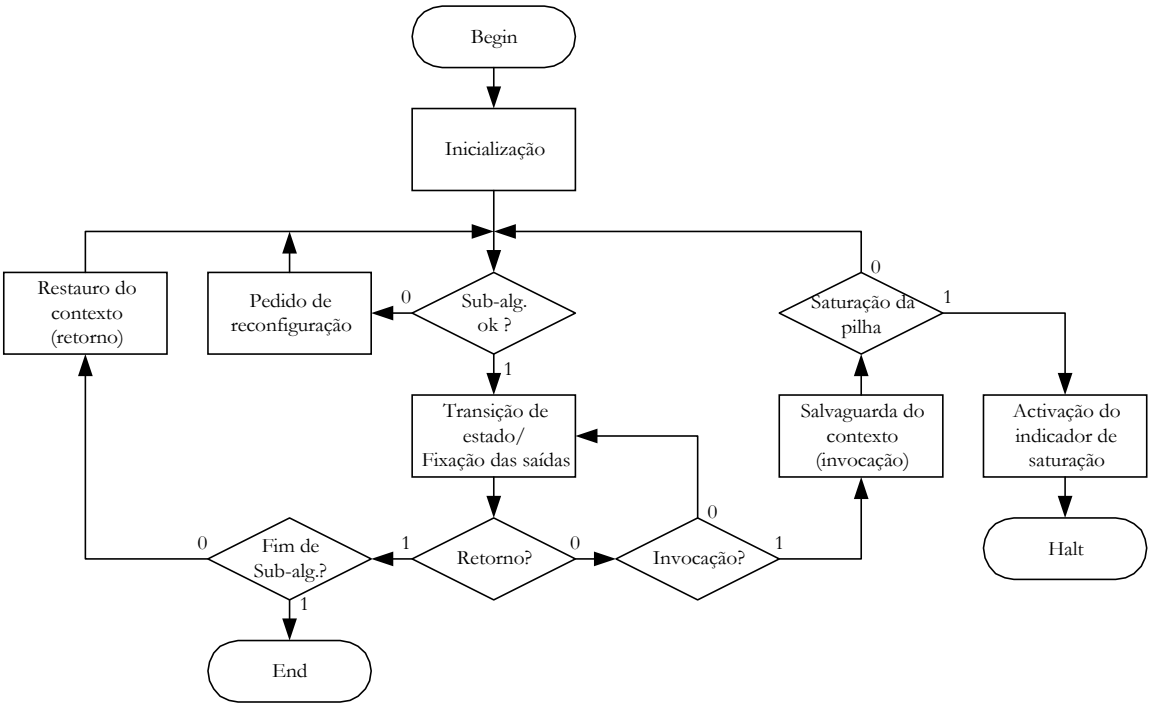


Figura 6.20 – Especificação simplificada do circuito de sincronização.

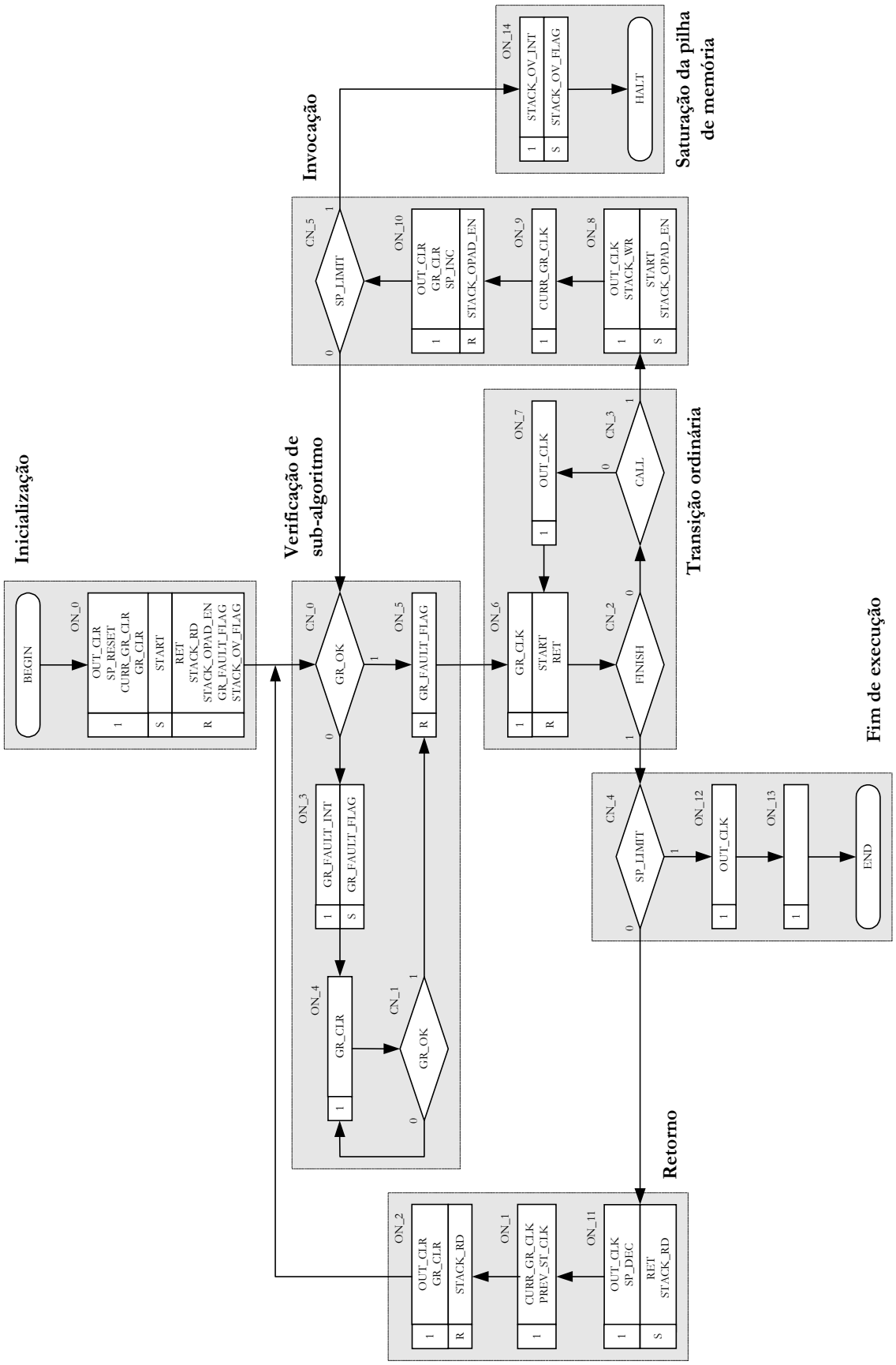


Figura 6.21 – Especificação detalhada do circuito de sincronização.

Na Figura 6.22 está ilustrada a representação física do circuito de sincronização após a sua implementação. A área considerável que ocupa resulta da necessidade do alinhamento entre os portos do circuito de sincronização e os recursos controlados, de forma a simplificar a tarefa das ferramentas de implementação e a cumprir as convenções relacionadas com a utilização dos recursos de interligação sobre os elementos reconfiguráveis.

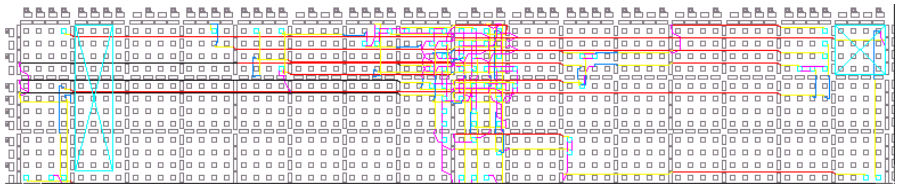


Figura 6.22 – Representação física do circuito de sincronização.

6.4.7 Interligação de Múltiplos Blocos de Implementação

A arquitectura desenvolvida é perfeitamente escalável permitindo a interligação de um número considerável de blocos de implementação, sendo limitada apenas pela capacidade lógica do dispositivo utilizado e pelos tempos de atraso introduzidos pelos multiplexadores usados na comutação dos blocos. A Figura 6.23 mostra o esquema de interligação de dois blocos de implementação. A sua descrição encontra-se no ficheiro *Dual_Block.vhd* apresentado no anexo IV. A Figura 6.24 ilustra a sua representação física.

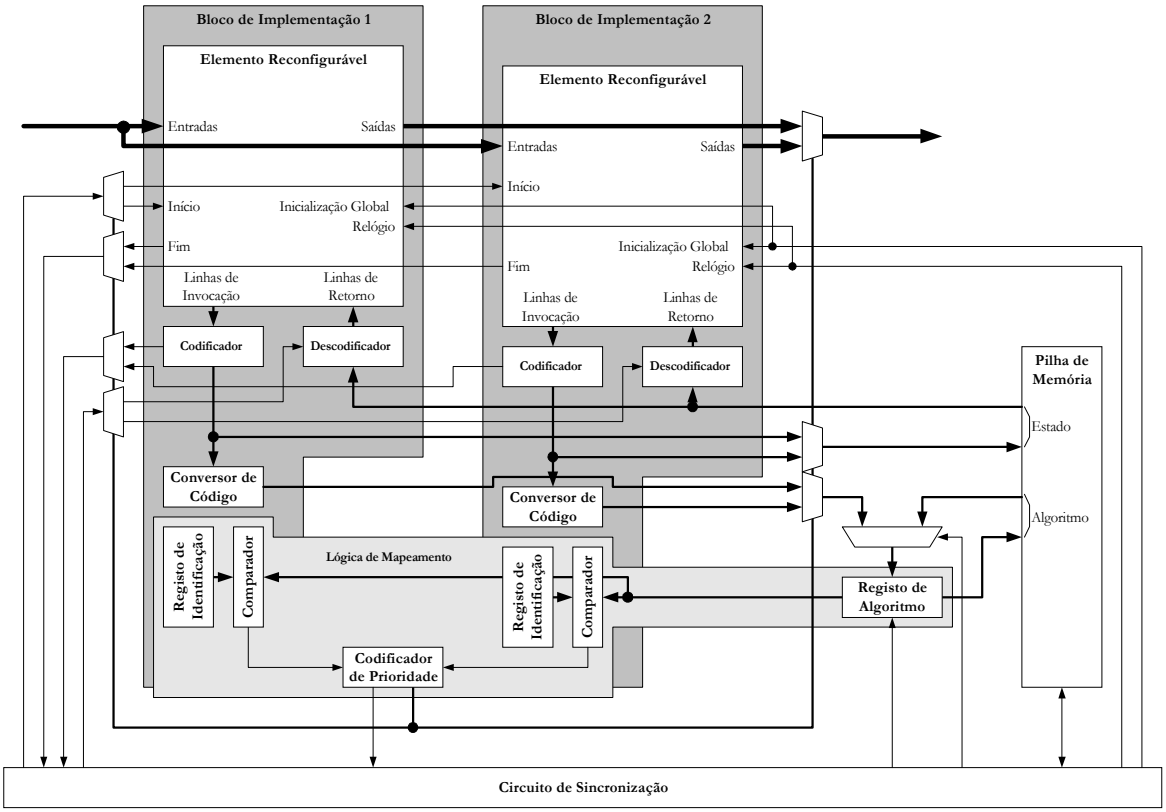


Figura 6.23 – Interligação de dois blocos de implementação.

À excepção de uma parte da lógica de mapeamento, da pilha de memória e do circuito de sincronização, todos os componentes são replicados em cada bloco de implementação, o que contribui para uma boa escalabilidade da arquitectura. Vamos agora analisar resumidamente a Figura 6.23. As entradas da unidade de controlo são aplicadas directamente aos elementos reconfiguráveis dos dois blocos de implementação. Tal como já foi referido no capítulo 3, no caso de um número elevado de entradas é conveniente a utilização de elementos de comutação para reduzir o número de entradas aplicadas a cada elemento. As saídas de estado da pilha de memória e as linhas de relógio e de inicialização global são também aplicadas directamente a cada bloco de implementação. Todos os restantes sinais são comutados com o auxilio de multiplexadores ou desmultiplexadores, uma vez que se assume a inexistência de sub-algoritmos a executar em paralelo. Estes elementos de comutação são controlados pela lógica de mapeamento que define o bloco de implementação que contém o sub-algoritmo a executar em cada instante. Esta arquitectura pode ser facilmente convertida em paralela hierárquica através da separação dos vários blocos de implementação, da atribuição de uma pilha de memória a cada um e da alteração do circuito de sincronização. Este é um dos pontos de trabalho futuro apresentado no último capítulo desta dissertação. Para simular a implementação em dispositivos de contexto múltiplo e para avaliar futuramente o impacto do tempo de reconfiguração no desempenho do circuito, foram implementados numa FPGA XC6216 quatro blocos de implementação. A sua descrição é feita nos ficheiros *Quad_Block.vhd* e *VCU.vhd* do anexo IV. A correspondente representação física está ilustrada na Figura 6.25. A FPGA XC6216 é composta por uma matriz de 64×64 células, sendo a sua capacidade lógica equivalente de 16K portas lógicas. A complexidade máxima do algoritmo de controlo implementável nestas circunstâncias possui os seguintes parâmetros: 32 entradas, 32 saídas, 64 sub-algoritmos, 4 transições hierárquicas por sub-algoritmo, 256 níveis hierárquicos e 100 a 150 nodos por sub-algoritmo.

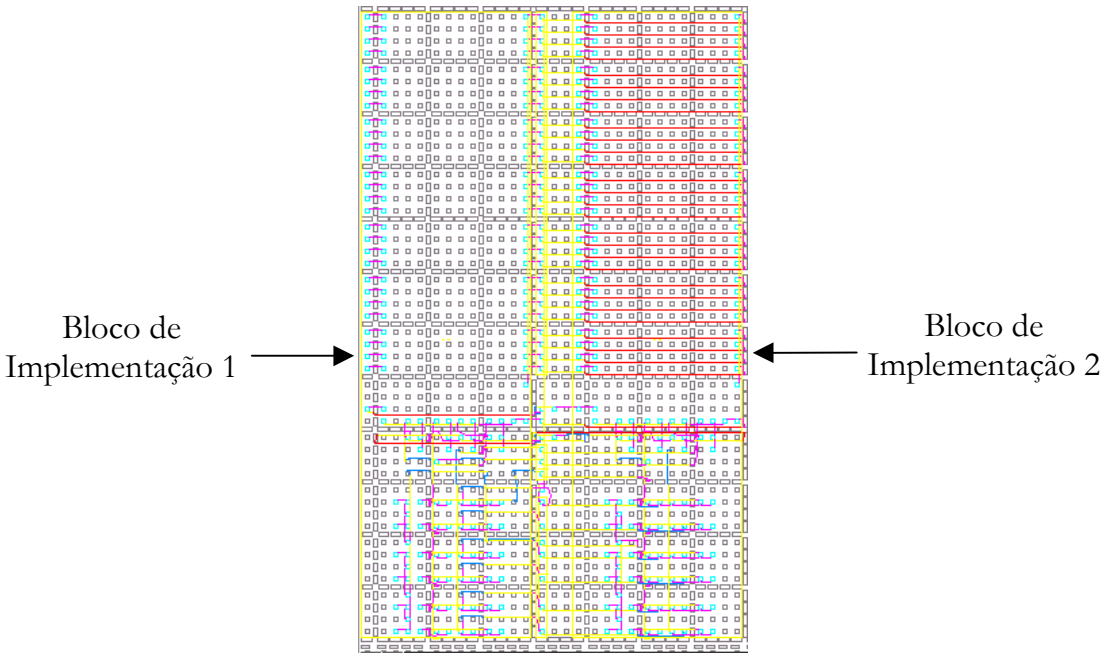


Figura 6.24 – Representação física da interligação de dois blocos de implementação.

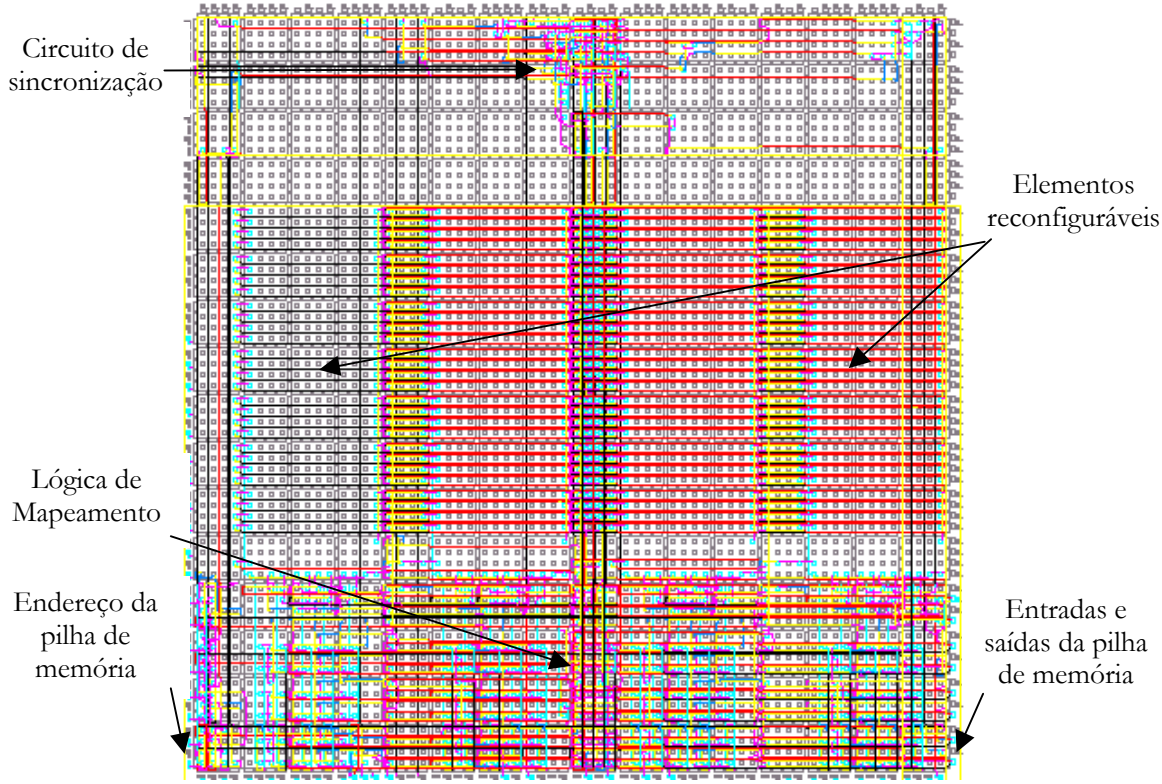


Figura 6.25 – Representação física global da estrutura predefinida implementada numa FPGA XC6216.

6.4.8 Implementação em Dispositivos de Contexto Múltiplo e Utilização em Sistemas Integrados

A implementação realizada na FPGA XC6200 serviu para validar a arquitectura desenvolvida. Contudo, da análise da sua estrutura e representação física pode-se concluir que existe um número considerável de células que são usadas em funções permanentes e outras que não são de todo utilizadas. A existência das segundas deve-se à necessidade de alinhamento entre determinados componentes de forma a simplificar o trabalho das ferramentas de implementação, diminuir o seu tempo de execução e tornar mais previsíveis os resultados obtidos. Só desta forma é possível cumprir as convenções estabelecidas na secção 6.4.1 relacionadas com a utilização dos recursos de interligação na zona dos elementos reconfiguráveis. Esta foi a abordagem escolhida, uma vez que a utilização de atributos para restringir o processo de interligação revelou-se bastante complexa e ineficiente.

A implementação desta arquitectura pode, em princípio, ser realizada mais eficientemente em dispositivos de múltiplo contexto, tal como ilustrado na Figura 6.26. Actualmente ainda não existe nenhum dispositivo deste tipo disponível comercialmente existindo apenas protótipos de investigação como o apresentado em [Nec98].

Ao contrário das arquitecturas tradicionais de FPGAs que só possuem um contexto ou página de configuração, num dispositivo de contexto múltiplo os recursos lógicos são controlados por uma das várias páginas de configuração disponíveis. No

caso particular da arquitectura desenvolvida, os recursos lógicos seriam utilizados para construir somente um bloco de implementação. Os vários sub-algoritmos seriam implementados nas diversas páginas de configuração. Cada uma deveria conter toda a informação necessária para programar um sub-algoritmo, nomeadamente a sua identificação e as configurações do elemento reconfigurável e do conversor de código (Figura 6.26). A transição entre diferentes sub-algoritmos seria realizada pela comutação de páginas de configuração. Obviamente alguns componentes, tais como a pilha de memória e a sincronização devem ser implementados num contexto comum.

Finalmente, a área ocupada pela arquitectura pode ser reduzida através da construção de blocos dedicados a circuitos de controlo, em que todas as ligações e componentes não modificáveis sejam implementados de forma permanente durante a construção do dispositivo lógico programável. De facto, uma situação análoga já existe actualmente com a integração em FPGAs de blocos de memória e de recursos dedicados à construção de circuitos aritméticos. Com o advento dos sistemas integrados (*Systems on Chip - SoC*) esta abordagem poderá fazer sentido para simplificar o projecto estruturado de unidades de controlo hierárquicas, complexas e reconfiguráveis. No entanto, reconhece-se que a estrutura a implementar deve ser tão geral quanto possível de forma a ser aplicável a uma grande diversidade de aplicações.

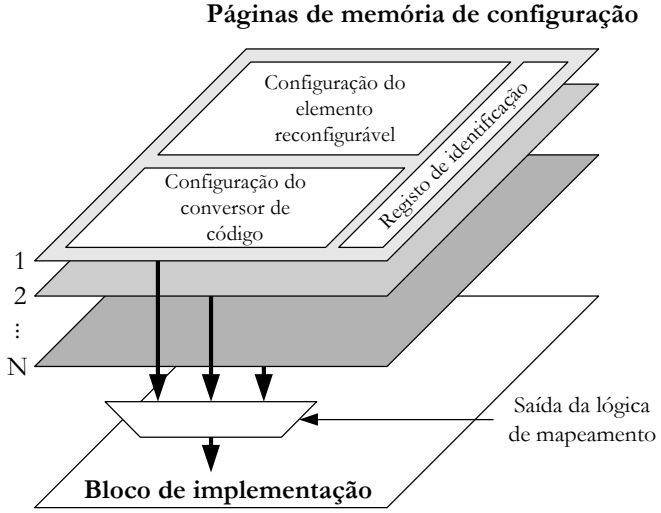


Figura 6.26 – Implementação da arquitectura desenvolvida em dispositivos de contexto múltiplo.

6.5 Biblioteca VHDL de Componentes

Para simplificar o desenvolvimento de projectos destinados a serem implementados em FPGAs da família XC6200 foi construída uma biblioteca VHDL de componentes descritos de forma estrutural e optimizados para esta arquitectura. Alguns destes componentes são invariantes, no entanto, a maior parte deles é parametrizável, o que significa que o seu tamanho e outras características podem ser facilmente modificáveis durante a sua instanciação nos módulos onde forem utilizados. Por exemplo, a biblioteca possui matrizes com um número variável de componentes do mesmo tipo, portas lógicas com um número parametrizável de entradas e registos

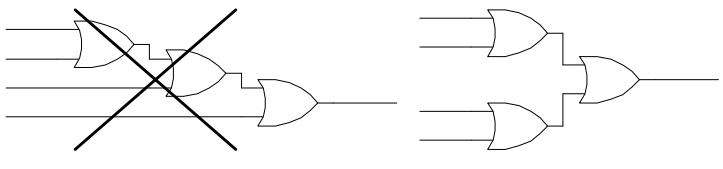
em que se pode modificar o número de bits bem como o espaçamento entre as células de cada um.

A biblioteca desenvolvida é baseada na PRIMS fornecida pela Xilinx, a qual contém todos os componentes implementáveis numa célula de uma FPGA XC6200 e que foram apresentados no capítulo 2 desta dissertação. Além disso, destina-se a complementar a biblioteca MACROS distribuída com o compilador de VHDL VELAB [Xilinx97b] e que possui componentes mais complexos, tais como acumuladores, subtractores, contadores, conversores série-paralelo e paralelo-série, etc. Adicionalmente, pode em certos casos substituir com vantagem qualquer uma das anteriores uma vez que permite implementar eficientemente portas lógicas com qualquer número de entradas, construir registos bastante flexíveis e converter directamente descrições comportamentais de unidades de controlo descritas por GSs/HGSs num circuito baseado em codificação *one-hot* e destinado a ser implementado na arquitectura XC6200.

A título de exemplo são mostradas abaixo as listagens VHDL de três dos componentes mais representativos desta biblioteca: uma porta lógica OR de N entradas, um registo de N bits com flip-flops "protegidos", ambos parametrizáveis e um componente que implementa um nodo condicional de um GS/HGS.

As portas lógicas de N entradas parametrizáveis são muito úteis, em particular as OR que, como vimos no capítulo anterior, são bastante utilizadas na construção de unidades de controlo descritas GSs/HGSs e baseadas em codificação *one-hot* para implementar a convergência de caminhos e o cálculo das saídas activadas em vários estados. Na Figura 6.28 é apresentada a descrição de uma porta lógica OR de N entradas parametrizável. A interligação das portas lógicas OR de 2 entradas que a constituem garante que se obtém uma configuração óptima em termos de tempos de propagação, uma vez que não é utilizada a ligação trivial em cascata (Figura 6.27).

Na descrição dos componentes são usados atributos, os quais têm como objectivo a especificação de restrições e o fornecimento de directivas ou informação que possa ser útil em fases posteriores do projecto em particular na sua implementação e depuração. Os atributos podem entre outras coisas, atribuir um nome a um componente, fixar a sua posição, estipular a sua forma, especificar restrições temporais importantes que devam ser respeitadas e indicar o valor inicial de registos constituídos por flip-flops "protegidos", os quais só podem ser escritos a partir do interface de configuração. Em [Xilinx97c] encontra-se uma descrição completa de todos os atributos utilizáveis na família XC6200.



Configuração Incorrecta Configuração Correcta

(a) (b)

Figura 6.27 – Duas configurações possíveis para construir uma porta lógica OR de 4 entradas com três portas de duas entradas (a) incorrecta (b) correcta.

No caso da porta lógica OR de N entradas apresentada é utilizado o atributo *FLATTEN* que planifica a relação hierárquica existente entre os componentes instanciados num dado módulo, de forma a ser possível colocar na mesma célula uma porta lógica e um flip-flop de diferentes componentes, melhorando assim os resultados obtidos na fase de implementação, isto é, a área ocupada pela implantação dos componentes e os tempos de propagação resultantes da sua interligação.

```
--------------------------------------------------------------------------
Porta lógica OR de N entradas parametrizável
--------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library PRIMS;
use PRIMS.XC6000_COMPONENTS.ALL;

entity ORN is
       generic(N      : integer := 3);

       port(I         : in    STD_LOGIC_VECTOR((N-1) downto 0);
            O         : out   STD_LOGIC);

       attribute FLATTEN      of ORN  : entity is "";
end ORN;

architecture STRUCT of ORN is

       signal OR_O    : STD_LOGIC_VECTOR(((2*N)-2) downto 0);

begin

       OR_O((N-1) downto 0) <= I((N-1) downto 0);

       G       : for J in N downto 2 generate
       begin
              B        : OR2   port map(I0    => OR_O(2*(N-J)),
                                        I1    => OR_O((2*(N-J))+1),
                                        O     => OR_O((2*(N-J))+J));
       end generate;

       O       <= OR_O((2*N)-2);

end STRUCT;
```

Figura 6.28 – Listagem VHDL da porta lógica OR de N entradas parametrizável.

Na Figura 6.29 é apresentada a listagem VHDL do componente *COND_NODE* que implementa um nodo condicional de um GS/HGS. Os seus portos são a condição lógica, o ponto de entrada do nodo e as duas saídas relativas a cada um dos valores lógicos da condição. Este componente é essencialmente um desmultiplexador controlado pela condição lógica. À semelhança do caso anterior é utilizado o atributo *FLATTEN* para que as suas portas lógicas possam ser combinadas na mesma célula da FPGA com flip-flops de outros componentes.

```
--------------------------------------------------------------------------
-- Componente que implementa um nodo condicional de um GS/HGS
--------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library PRIMS;
use PRIMS.XC6000_COMPONENTS.ALL;
```

```
entity COND_NODE is
       port(I          : in    STD_LOGIC;
            COND       : in    STD_LOGIC;
            O_0        : out   STD_LOGIC;
            O_1        : out   STD_LOGIC);

       attribute FLATTEN      of COND_NODE   : entity is "";
end COND_NODE;

architecture STRUCT of COND_NODE is

begin

       AND_0   : AND2B1       port map(I0    => COND,
                                       I1    => I,
                                       O     => O_0);

       AND_1   : AND2         port map(I0    => COND,
                                       I1    => I,
                                       O     => O_1);

end STRUCT;
```

Figura 6.29 – Listagem VHDL do componente *COND_NODE* que implementa um nodo condicional de um GS/HGS.

Finalmente, na Figura 6.30 é apresentada a listagem VHDL do componente *PREGN*, que é um registo de N bits parametrizável constituído por flip-flops "protegidos", os quais podem ser alterados somente a partir do interface de configuração de uma FPGA XC6200. A parametrização possui três graus de liberdade: o número de bits, o espaçamento entre os flip-flops que constituem o registo e o seu valor inicial.

O atributo *OVERLAP* permite, no caso dos flip-flops não estarem colocados em células adjacentes, utilizar os espaços livres para colocar primitivas de outros componentes.

```
--------------------------------------------------------------------------
-- Registo de n bits parametrizável, alinhado verticalmente e
-- constituído por flip-flops "protegidos"
--------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library PRIMS;
use PRIMS.XC6000_COMPONENTS.ALL;

entity PREGN is
       generic(N      : integer := 8;
               OS     : integer := 1;
               IV     : integer := 0);

       port(CLK       : in    STD_LOGIC;
            Q         : out   STD_LOGIC_VECTOR((N-1) downto 0));

       attribute BBOX         of PREGN       : entity       is "TALL";
       attribute OVERLAP      of PREGN       : entity       is "YES";
end PREGN;

architecture STRUCT of PREGN is

begin

       G : for J in (N-1) downto 0 generate
              attribute RLOC         of B    : label is "X0Y,J*OS,";
```

```
            attribute REGNAME      of B    : label is "REG";
            attribute REGBIT       of B    : label is ",J,";
            attribute INIT         of B    : label is ",(IV/(2**J))%2,";
      begin
            B       : RPFD  port map(C      => CLK,
                                     Q      => Q(J));
      end generate;

end STRUCT;
```

Figura 6.30 – Listagem VHDL do registo de N bits parametrizável constituído por flip-flops "protegidos".

Cada componente da biblioteca está implementado num ficheiro VHDL separado. Além disso, existem ficheiros adicionais chamados *packages* que dividem os componentes em módulos de acordo com a sua funcionalidade e facilitam a inclusão do seu protótipo ou definição do interface nos projectos que os utilizam.

De forma geral, uma *package* em VHDL permite encapsular definições, componentes e subprogramas que podem ser incluídos noutras descrições em VHDL. Esta possibilidade permite construir bibliotecas de declarações, procedimentos e funções, o que contribui para a modularidade e reutilização das descrições ou modelos escritos nesta linguagem.

Para concluir este capítulo são apresentadas as tabelas que contêm os componentes de cada *package* da biblioteca, bem como uma descrição resumida de cada um. Todas as *packages* são gerais e podem ser utilizadas com qualquer FPGA da família XC6200, exceptuando a *package RAM_Pack* que contém macros de acesso a memória SRAM externa e que só pode ser utilizada com a placa de desenvolvimento *FireFly*™ descrita na secção 6.1.1. A utilização destas macros com outras placas de desenvolvimento requer a modificação dos atributos que definem os pinos utilizados no interface com a memória externa.

Array_Pack.vhd

Nome do Módulo	Descrição
N_Buf.vhd	Matriz de N *buffers* parametrizável
N_Gnd.vhd	Matriz de N "0s" parametrizável
N_Inv.vhd	Matriz de N inversores parametrizável
N_M2_1.vhd	Matriz de N multiplexadores de 2 para 1 parametrizável
N_Vcc.vhd	Matriz de N "1s" parametrizável

Tabela 6.1 – Constituição da *package Array_Pack.vhd*.

Cmp_Pack.vhd

Nome do Módulo	Descrição
Eq_CompN.vhd	Comparador de igualdade de N bits parametrizável

Tabela 6.2 – Constituição da *package Cmp_Pack.vhd*.

Cnt_Pack.vhd

Nome do Módulo	Descrição
U_B_CntN.vhd	Contador binário crescente de N bits parametrizável
U_B_CntN_1.vhd	Contador binário crescente de N bits parametrizável com uma célula de separação entre as saídas
Ud_B_CntN_1.vhd	Contador binário crescente/decrescente de N bits parametrizável com uma célula de separação entre as saídas

Tabela 6.3 – Constituição da *package Cnt_Pack.vhd.*

Cod_Pack.vhd

Nome do Módulo	Descrição
Cod4_2.vhd	Codificador binário de 4 para 2 linhas
Cod8_3.vhd	Codificador binário de 8 para 3 linhas
Cod16_4.vhd	Codificador binário de 16 para 4 linhas
Cod32_5.vhd	Codificador binário de 32 para 5 linhas
Pr_Cod4_2.vhd	Codificador binário de prioridade de 4 para 2 linhas

Tabela 6.4 – Constituição da *package Cod_Pack.vhd.*

Dec_Pack.vhd

Nome do Módulo	Descrição
Dec2_4.vhd	Descodificador binário de 2 para 4 linhas
Dec3_8.vhd	Descodificador binário de 3 para 8 linhas
Dec4_16.vhd	Descodificador binário de 4 para 16 linhas
Dec5_32.vhd	Descodificador binário de 16 para 32 linhas

Tabela 6.5 – Constituição da *package Dec_Pack.vhd.*

Gates_Pack.vhd

Nome do Módulo	Descrição
AndN.vhd	Porta lógica AND de N entradas parametrizável
NandN.vhd	Porta lógica NAND de N entradas parametrizável
NorN.vhd	Porta lógica NOR de N entradas parametrizável
OrN.vhd	Porta lógica OR de N entradas parametrizável
XnorN.vhd	Porta lógica XNOR de N entradas parametrizável
XorN.vhd	Porta lógica XOR de N entradas parametrizável

Tabela 6.6 – Constituição da *package Gates_Pack.vhd.*

Gs_Pack.vhd

Nome do Módulo	Descrição
Cond_Node.vhd	Nodo condicional de um GS/HGS
Init_Op_Node.vhd	Nodo operacional inicial de um GS/HGS (durante a inicialização é carregado com o valor lógico "1")
Op_Node.vhd	Nodo operacional de um GS/HGS (durante a inicialização é carregado com o valor lógico "0")

Tabela 6.7 – Constituição da *package Gs_Pack.vhd*.

Lat_FF_Pack.vhd

Nome do Módulo	Descrição
FTC.vhd	Flip-flop tipo T com linha de inicialização
SR_Latch.vhd	*Latch* tipo SR com portas lógicas NOR

Tabela 6.8 – Constituição da *package Lat_FF_Pack.vhd*.

Misc_Pack.vhd

Nome do Módulo	Descrição
DelayN.vhd	Atraso programável
A_Stk_CtrlN.vhd	Controlador de pilha de memória assíncrono de N bits parametrizável
Stk_CtrlN_1.vhd	Controlador de pilha de memória síncrono de N bits parametrizável

Tabela 6.9 – Constituição da *package Misc_Pack.vhd*.

Ram_Pack.vhd

Nome do Módulo	Descrição
A_Ram_32.vhd	Macro bloco assíncrono para acesso de leitura/escrita de 32 bits da memória SRAM externa e com mecanismo de prevenção de conflitos no barramento
RW_Ram_32.vhd	Macro bloco síncrono para acesso de leitura/escrita de 32 bits da memória SRAM externa e com mecanismo de prevenção de conflitos no barramento
Rd_Ram_32.vhd	Macro bloco síncrono para acesso de leitura de 32 bits da memória SRAM externa e com mecanismo de prevenção de conflitos no barramento

Tabela 6.10 – Constituição da *package Ram_Pack.vhd*.

Reg_Pack.vhd

Nome do Módulo	Descrição
Preg_MuxN.vhd	Registo de N bits com flip-flops "protegidos", espaçamento entre bits parametrizável, valor inicial programável e multiplexadores nas saídas (útil para construir bancos de registos)
Preg_XnorN.vhd	Registo de N bits com flip-flops "protegidos", espaçamento entre bits parametrizável, valor inicial programável e portas lógicas XNOR nas saídas
Preg4N.vhd	Banco de 4 registos de N bits com flip-flops "protegidos", espaçamento entre bits parametrizável e valor inicial programável
PregN.vhd	Registo de N bits com flip-flops "protegidos", espaçamento entre bits parametrizável e valor inicial programável
Reg_MuxN.vhd	Registo de N bits com flip-flops tipo D, multiplexadores de 2:1 e espaçamento entre bits parametrizável (útil para construir bancos de registos)
RegN.vhd	Registo de N bits com flip-flops tipo D e espaçamento entre bits parametrizável

Tabela 6.11 – Constituição da *package Reg_Pack.vhd*.

7 Software para Controlo da Reconfiguração Dinâmica

Sumário

Este capítulo começa por descrever de forma resumida as várias opções de controlo do processo de reconfiguração dinâmica de uma FPGA, nomeadamente, a auto-reconfiguração, o controlo externo por hardware e o controlo externo por software. Cada um destes métodos possui vantagens e inconvenientes que são também discutidos sumariamente. De todos eles, a reconfiguração controlada por software é um dos mais utilizados. Devido às deficiências encontradas nas ferramentas de software construídas pela Xilinx e pela Annapolis Micro Systems e distribuídas juntamente com a placa de desenvolvimento *FireFly*™ utilizada neste trabalho, optou-se por desenvolver os seguintes módulos de software:

- Biblioteca de classes escrita em C++ e disponibilizada na forma de *DLLs* para sistemas operativos Microsoft Windows de 32 bits. A funcionalidade implementada permite a leitura e processamento de todos os ficheiros criados pelas ferramentas de implementação da família de FPGAs XC6200, a sua (re)configuração e o controlo de todos os aspectos da placa *FireFly*™;
- Controlador de software (*device driver*) para Microsoft Windows 98/2000 da placa de desenvolvimento *FireFly*™ de forma a permitir o controlo de todas as suas características a partir de uma aplicação de utilizador a executar num dos sistemas operativos referidos acima.

O desenvolvimento deste módulos teve várias vantagens das quais se destacam:

- Um conhecimento mais aprofundado da arquitectura da FPGA XC6200 e da placa de desenvolvimento *FireFly*™;
- Um maior controlo de todos os componentes do sistema, o que facilita a introdução de optimizações.

7.1 Introdução

A reconfiguração dinâmica de uma FPGA pode ser controlada de várias formas, das quais se destacam as seguintes:

- **Auto-reconfiguração** – a própria FPGA estabelece a sequência de operações necessárias para a sua reconfiguração. Apesar da informação de configuração estar normalmente armazenada externamente (à excepção dos dispositivos de contexto múltiplo) a FPGA controla completamente a sua reconfiguração, gerando todos os sinais necessários, actuando assim como mestre do sistema;
- **Controlo externo por hardware** – a reconfiguração dinâmica da FPGA é controlada por componentes externos dedicados e optimizados para esta função e que permitem a transferência eficiente da informação de configuração de um dispositivo de armazenamento externo para a memória interna da FPGA, actuando a FPGA como escravo do sistema;
- **Controlo externo por software** – o controlo do processo de reconfiguração é realizado por uma aplicação de software a executar num microprocessador ou microcontrolador de uso geral, estando a FPGA mapeada numa região do seu espaço de endereçamento de memória ou de entrada/saída. À semelhança do caso anterior, a FPGA também actua como escravo do sistema.

Todas estas formas de controlo da reconfiguração dinâmica têm vantagens e inconvenientes. Do ponto de vista da eficiência e desempenho, a primeira é provavelmente a melhor, mas também aquela que utiliza mais recursos internos da FPGA e que requer mais cuidados para garantir que o circuito de controlo não se auto-destrói. Apesar desta abordagem permitir a diminuição do número de componentes externos, tem também a desvantagem de utilizar recursos reconfiguráveis para implementar circuitos cuja funcionalidade se pretende em princípio manter fixa. Além disso, requer que a memória de configuração da FPGA possa ser acedida pelos próprios recursos lógicos controlados. A arquitectura de FPGAs XC6200 dispõe desta capacidade.

Por oposição, o controlo por software é o mais fácil de implementar, mas também aquele que é menos eficiente devido à sobrecarga introduzida pelo software. Este método é bastante utilizado em aplicações que utilizem a FPGA como coprocessador reconfigurável.

Finalmente, o controlo externo por hardware anula a desvantagem da utilização de células dinamicamente reconfiguráveis em funções fixas mas também aumenta o número de dispositivos do sistema. Relativamente ao software tem a vantagem de permitir uma transferência mais eficiente da configuração. Contudo, esta vantagem é mais óbvia nos casos em que todos os recursos lógicos e de interligação de um grupo de células adjacentes sejam controlados por endereços contíguos da memória de configuração, o que não acontece na arquitectura XC6200. Ao contrário do que a Xilinx procura transmitir na documentação da arquitectura XC6200, o mapeamento

entre os bits da memória de configuração e os recursos lógicos controlados é bastante complexo e em certos casos até irregular, o que dificulta a reconfiguração dinâmica do dispositivo.

Para suportar a reconfiguração dinâmica de um sistema têm sido propostas várias abordagens e arquitecturas sendo a maior parte baseada em técnicas de auto-reconfiguração [RobLys99, McGLys99] e de controlo externo por software [Brebner96, ShiLukChe98].

Neste capítulo é apresentada de forma resumida a contribuição efectuada no âmbito deste trabalho para a reconfiguração de FPGAs controlada por software e que pode ser dividida em duas componentes principais:

- Uma biblioteca orientada por objectos, escrita em C++ [Stroustrup95] e que permite o carregamento de ficheiros de configuração, a depuração, a transferência de dados e a reconfiguração de FPGAs da família XC6200. O controlo de todas funcionalidades da placa de desenvolvimento *FireFly*™ é também suportado. Esta biblioteca é também extensível a outras placas que possuam uma ou várias FPGAs XC6200;
- Um controlador de software para a placa de desenvolvimento *FireFly*™, que permite o controlo de todas as suas funcionalidades a partir de uma aplicação de utilizador, bem como a introdução de optimizações na transferência de dados entre o hardware e a aplicação.

Cada um destes módulos vai ser apresentado de forma resumida nas próximas secções.

7.2 A Biblioteca IMPARLIB

A biblioteca IMPARLIB (*IMProved Access and Reconfiguration LIBrary*) foi construída para simplificar as operações de acesso e reconfiguração do hardware utilizado neste trabalho: a placa de desenvolvimento *FireFly*™ apresentada no capítulo anterior e que possui interface PCI, dois bancos de memória SRAM e duas FPGAs da Xilinx, uma XC4013 para implementação do interface PCI e uma FPGA reconfigurável dinâmica e parcialmente da família XC6200 (XC6216 ou XC6264).

O desenvolvimento desta biblioteca foi motivado por algumas deficiências ao nível da estrutura e carências funcionais detectadas nas bibliotecas RALLIB e PCIRAL [Xilinx97c] desenvolvidas pela Xilinx e pela Annapolis Micro Systems e distribuídas com a placa *FireFly*™. Estas bibliotecas são orientadas por objectos, foram escritas em C++ e são distribuídas em código fonte. As tarefas que permitem realizar são: a leitura e o processamento dos ficheiros produzidos pela ferramenta de implementação XACT6000, o carregamento dos ficheiros de configuração em dispositivos da família XC6200 e a reconfiguração das suas células. Finalmente, são também disponibilizadas classes para controlar diversas características da placa *FireFly*™, tais como os bancos de memória SRAM, os parâmetros do gerador de sinais de relógio, as interrupções e o consumo de potência. Na biblioteca IMPARLIB procurou-se eliminar as deficiências detectadas através de uma melhoria da sua estrutura e da integração de funcionalidades que nas bibliotecas anteriores tinham de ser implementadas pela aplicação.

Do ponto de vista da estrutura, os melhoramentos introduzidos consistiram essencialmente numa definição mais cuidadosa das classes e num aproveitamento mais efectivo das potencialidades proporcionadas por uma linguagem orientada por objectos como o C++. No entanto, houve também o cuidado em não utilizar construções que pudessem afectar seriamente o desempenho. Assim, a estrutura das bibliotecas RALLIB e PCIRAL foi alterada de forma a aumentar a reutilização das suas classes e a melhorar o encapsulamento dos objectos que elas representam. Por exemplo, as funções utilizadas para ler/escrever registos específicos da família de FPGAs XC6200 estão agora numa classe que encapsula os dispositivos desta família, em vez de nas classes específicas do sistema de desenvolvimento *FireFly*™. Por outro lado, a biblioteca contém funções que suportam tarefas comuns, tais como a leitura/escrita de componentes (símbolos) implementados na FPGA, bem como a sua reconfiguração.

Além das deficiências detectadas, verificou-se também que a inclusão desta biblioteca nos projectos que dela necessitavam era feita através de um número considerável de ficheiros com o seu código fonte. Este procedimento possui vários inconvenientes, sendo o maior, a necessidade de recompilar a biblioteca cada vez que é criado e compilado um novo projecto.

Apesar de um módulo estático pré-compilado permitir evitar a recompilação da biblioteca cada vez que esta for inserida num novo projecto, não permite ainda resolver o problema da actualização das aplicações previamente desenvolvidas após a realização de uma alteração ou correcção de um erro na biblioteca, sendo neste caso necessário voltar a compilar todas as aplicações que dela dependam. Para resolver este problema, optou-se por utilizar bibliotecas de associação dinâmica à aplicação (*Dynamic Link Libraries - DLLs*) que são carregadas automaticamente pelo sistema operativo e ligadas às respectivas aplicações no início da sua execução. Esta abordagem além de facilitar a realização de modificações e melhoramentos futuros, facilita a partilha das bibliotecas entre diferentes aplicações e diminui a memória ocupada quando existem várias aplicações que utilizam a mesma biblioteca a executar simultaneamente.

A biblioteca IMPARLIB foi escrita em C++ e compilada com o Microsoft Visual C++ 6.0. As *DLLs* resultantes podem ser utilizadas nos sistemas operativos Windows 95/98/NT4/2000. No seu desenvolvimento foram utilizadas algumas classes da biblioteca MFC (*Microsoft Foundation Classes*), mais concretamente, algumas estruturas de dados, os mecanismos para serialização de objectos e de informação dinâmica sobre os tipos de dados, etc. Uma desvantagem desta abordagem é a dificuldade em portar o código fonte para outras plataformas. No entanto, neste caso não foi possível evitá-la porque a biblioteca IMPARLIB inclui elementos gráficos, tais como caixas de diálogo para configurar os registos de controlo da FPGA e da placa *FireFly*™ e a MFC disponibiliza as classes apropriadas para os representar. A biblioteca IMPARLIB está dividida nos seguintes módulos, cujas dependências estão ilustradas na Figura 7.1:

- BoardInterface.dll
- Board.dll
- Xc6200.dll
- CalFile.dll
- RalLib.dll
- FireFly.dll

Nas subsecções seguintes são apresentados cada um destes módulos. Na sua representação gráfica vai ser utilizada a linguagem UML (*Unified Modeling Language*) [BooRumJac99] embora de uma forma simplificada para não sobrecarregar os diagramas. Mais concretamente, não são indicados os tipos dos atributos das classes, nem dos parâmetros formais e dos valores de retorno dos seus métodos. Além disso, nos diagramas são apresentados apenas os elementos (atributos e métodos) públicos das classes uma vez que são os mais relevantes para uma utilização regular.

O código fonte desta biblioteca não é apresentado em anexo essencialmente por dois motivos:

- Apesar de já ter sido testada na generalidade, para garantir a inexistência de erros é ainda necessário realizar alguns testes mais rigorosos;
- Como possui um número elevado de linhas de código, contribuiria para um aumento significativo do volume desta dissertação.

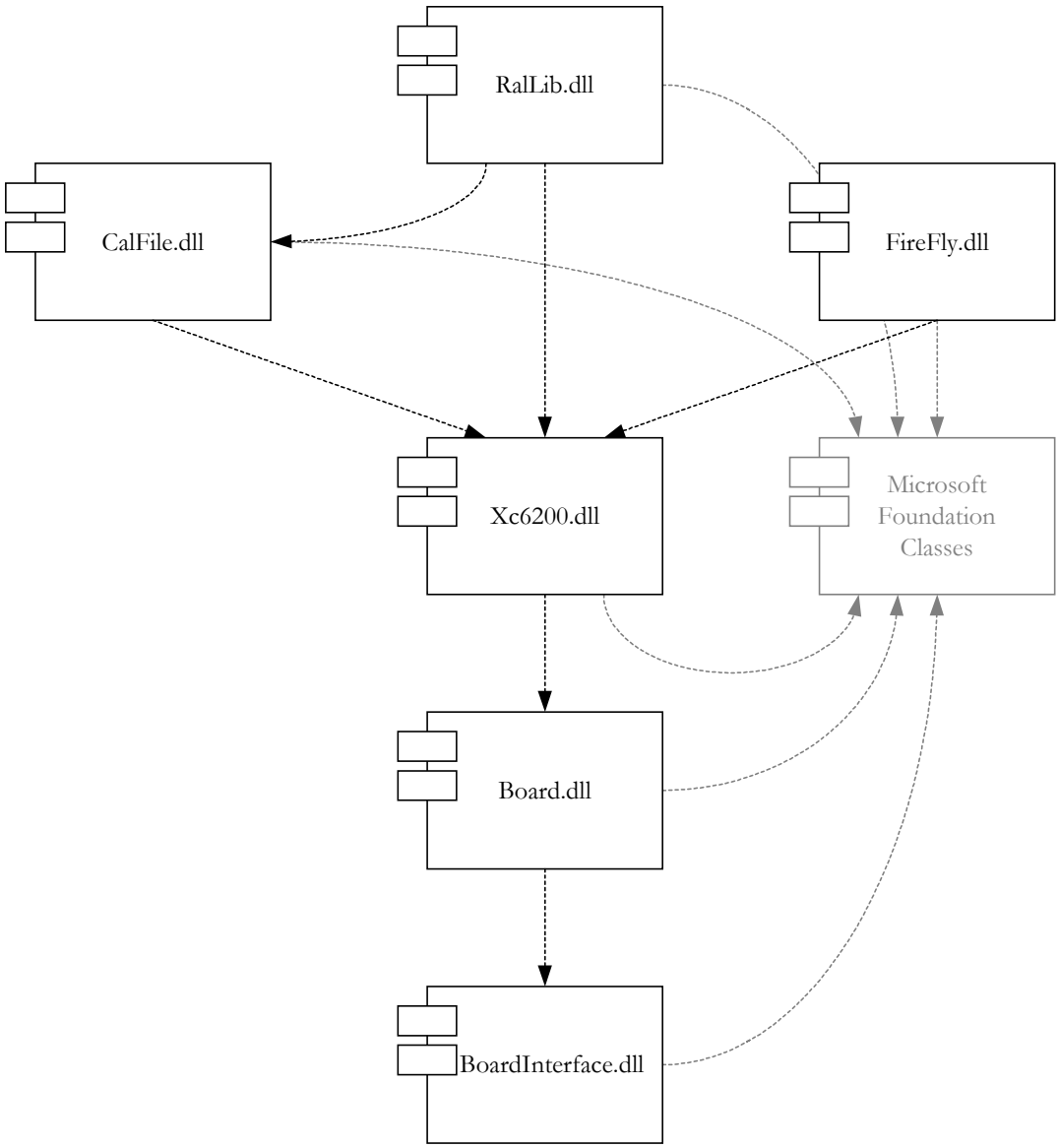


Figura 7.1 – Constituição da biblioteca IMPARLIB e dependências entre os seus módulos.

7.2.1 Módulo BoardInterface.dll

O módulo *BoardInterface.dll* possui classes que permitem abstrair diferentes tipos de interfaces normalmente utilizados em placas de hardware. Actualmente apenas é suportado o interface PCI, uma vez que é aquele que existe na placa de desenvolvimento *FireFly*™. As classes implementadas neste módulo são as seguintes (Figura 7.2): *CBoardInterface*, *CPciInterface*, *CBoardInterfaceException*. A classe *CBoardInterface* tem como objectivo estabelecer um tipo comum a partir do qual todos os interfaces devem ser derivados. De notar que, por ser abstracta, esta classe não pode ser instanciada. A classe *CPciInterface* é uma especialização da classe *CBoardInterface* em que todos os métodos são implementados partindo do pressuposto de que uma placa interligada ao barramento PCI deve ser acedida através de um controlador de software (*device driver*). Na secção 7.3 vai ser apresentado um controlador de software para a placa de desenvolvimento *FireFly*™ que trabalha em estreita cooperação com esta classe. Por último, a classe *CBoardInterfaceException* é utilizada no processamento de condições de excepção que podem ocorrer no acesso ao hardware, tais como um erro de detecção, inicialização ou abertura do dispositivo. A partir do momento em que todos os interfaces são derivados de um tipo comum é possível, através dos mecanismos de polimorfismo disponíveis nas linguagens orientadas por objectos [Stroustrup95], que os vários interfaces sejam tratados uniformemente e até que a mesma placa possa ser acedida através de diferentes tipos de interface. Esta abordagem é útil, por exemplo, na depuração remota de um circuito num laboratório de ensino com um número reduzido de placas de desenvolvimento. Para tal, basta derivar outro tipo de interface (ex. *CRemoteInterface*) da classe *CBoardInterface*, o qual faz parte de uma aplicação cliente-servidor que permite o acesso remoto à máquina que possui o hardware (servidor) a partir da máquina que vai executar a aplicação de depuração (cliente). A Figura 7.3 ilustra a relação de herança entre as classes *CBoardInterface* e *CPciInterface* e os métodos de cada uma destas classes. Em todos os diagramas apresentados as partes a cinzento representam componentes da biblioteca MFC. Resumidamente, os métodos implementados fornecem as seguintes funcionalidades:

- Obtenção de informações sobre o hardware (ex. endereços de memória);
- Escrita/leitura de posições de memória e portos de entrada/saída;
- Registo de uma função para processamento de interrupções.

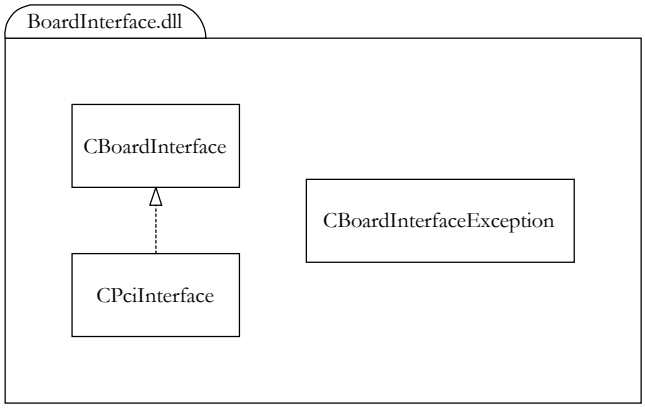


Figura 7.2 – Constituição do módulo *BoardInterface.dll*.

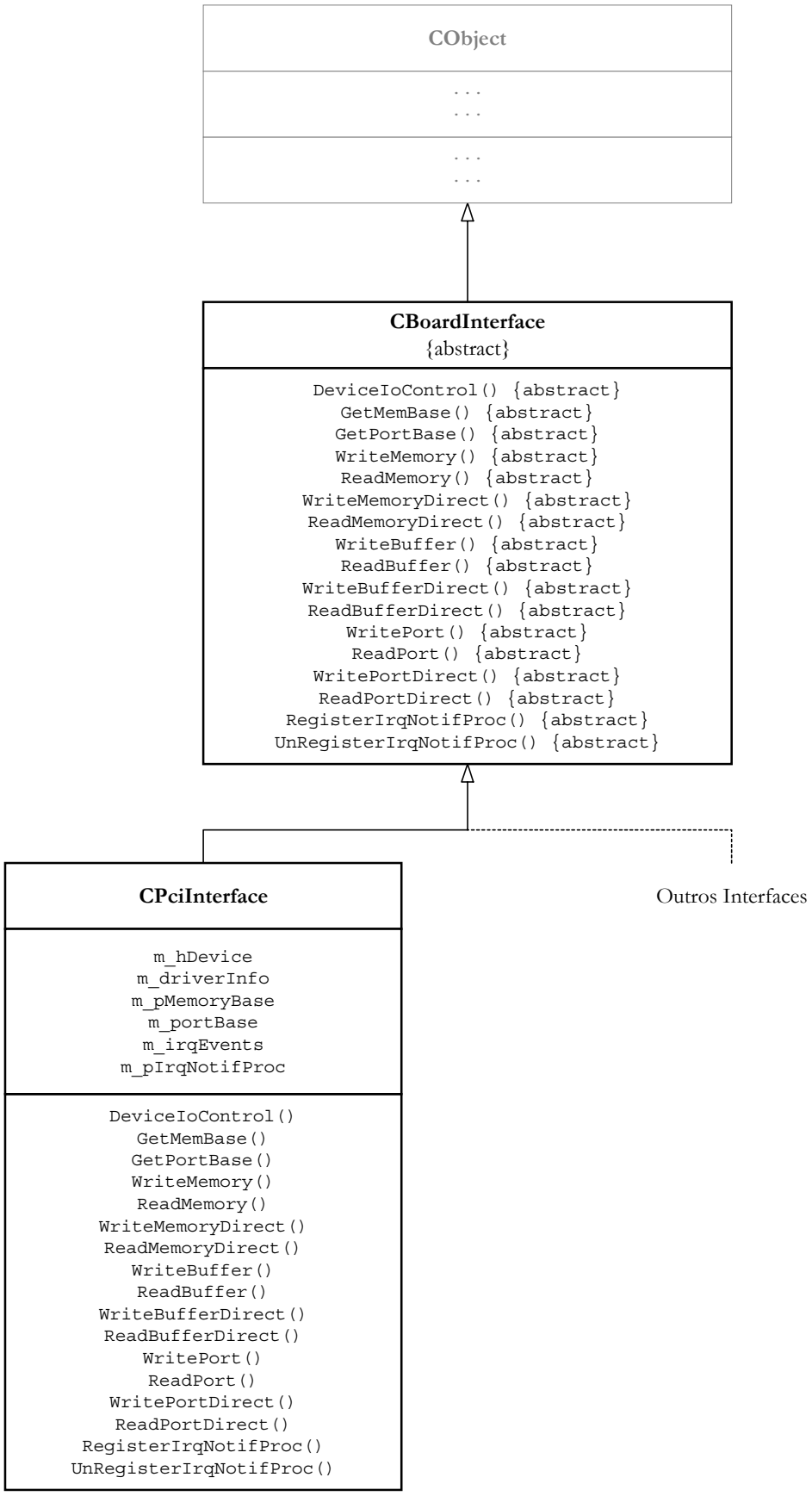


Figura 7.3 – Atributos, métodos e relações de herança das classes *CBoardInterface* e *CPciInterface*.

7.2.2 Módulo Board.dll

O objectivo do módulo *Board.dll* é fornecer uma classe base a partir da qual possam ser derivadas outras classes que representam placas de desenvolvimento específicas, possibilitando assim a utilização de técnicas de polimorfismo. A Figura 7.4 mostra as classes implementadas neste módulo, que são a *CBoard* e *CBoardException*. A classe *CBoard* é abstracta pelo que não podem ser criados objectos deste tipo. À semelhança do módulo anterior, este também possui uma classe dedicada ao tratamento de condições de excepção (*CBoardException*), como por exemplo a criação de um objecto com um interface inválido.

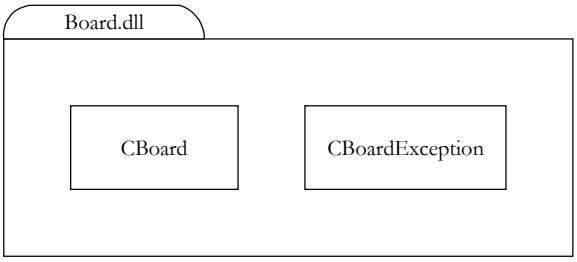


Figura 7.4 – Constituição do módulo *Board.dll*.

A Figura 7.5 ilustra as relações de hierarquia e de associação da classe *CBoard*. Cada placa deve possuir obrigatoriamente um interface, o qual é representado na classe *CBoard* pelo atributo *m_pBoardInterface*. O método *Setup()* permite a utilização de técnicas de polimorfismo na configuração dos parâmetros específicos de cada placa.

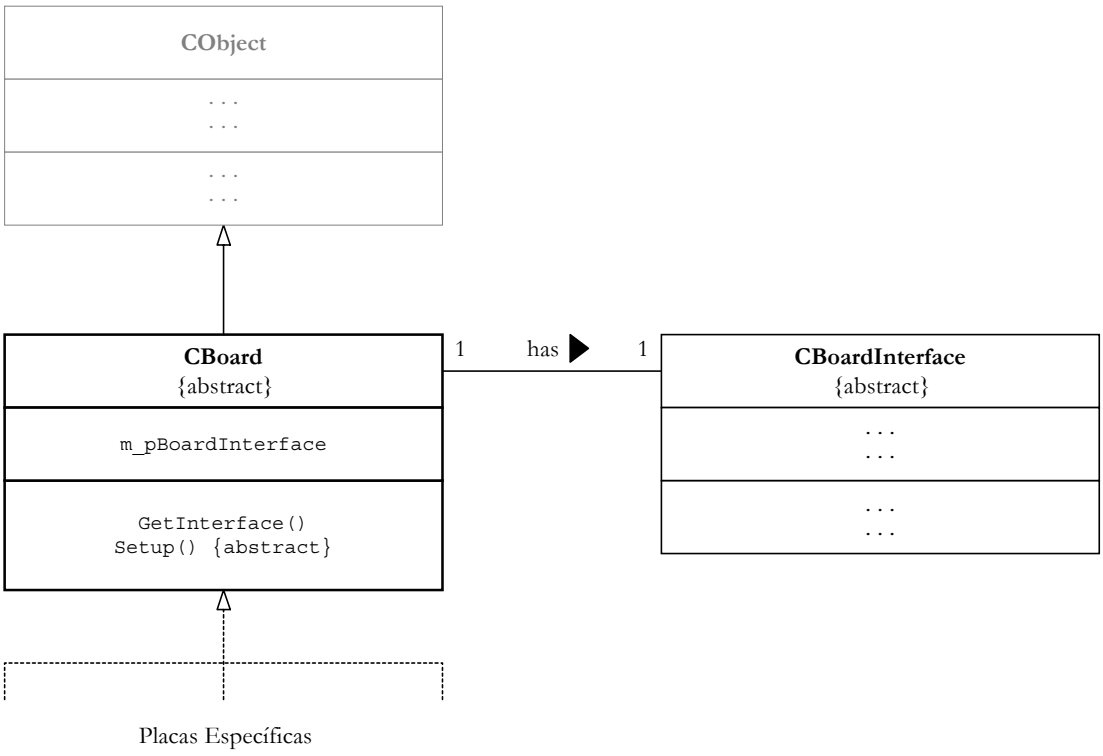


Figura 7.5 – Atributos, métodos e relações de herança e de associação das classes *CBoard* e *CBoardInterface*.

7.2.3 Módulo Xc6200.dll

As classes deste módulo têm como objectivo criar abstracções de software para as várias características da família de FPGAs XC6200, permitindo realizar facilmente todas as suas operações de acesso e configuração. Além disso, são também disponibilizados controlos que permitem alterar, graficamente, os registos de configuração. As classes implementadas neste módulo foram as seguintes (Figura 7.6):

- *CXc6200*, *CXc6216*, *CXc6264*, *CMapRegister*;
- *CXcAddrBus*, *CXc16BitAddrBus*, *CXc18BitAddrBus*;
- *CXc6200Board*;
- *CLoadDesignDlg*, *CXc6200SetupSheet*, *CConfigPage*, *CRegistersPage*.

O factor mais importante que diferencia a implementação realizada no âmbito deste trabalho da biblioteca PCIRAL construída pela Xilinx é a separação clara entre as classes que representam os dispositivos desta família de FPGAs e as classes que representam a placa de desenvolvimento *FireFly*™. Desta forma é possível reutilizar as primeiras para diferentes fins e em diferentes sistemas de desenvolvimento. A classe principal deste módulo é a *CXc6200* que encapsula todos os aspectos comuns da arquitectura XC6200. Todos os pormenores específicos de cada dispositivo são definidos nas classes *CXc6216* e *CXc6264*. Na Figura 7.7 estão ilustrados os atributos, os métodos e as relações de herança e de associação de cada uma destas classes. Os métodos incluídos permitem configurar todos os aspectos desta arquitectura. Para simplificar as operações de leitura/escrita de uma FPGA, cada objecto que a representa pode possuir uma placa associada. Uma característica inovadora que foi incluída foi a capacidade de detecção automática da FPGA existente numa dada placa.

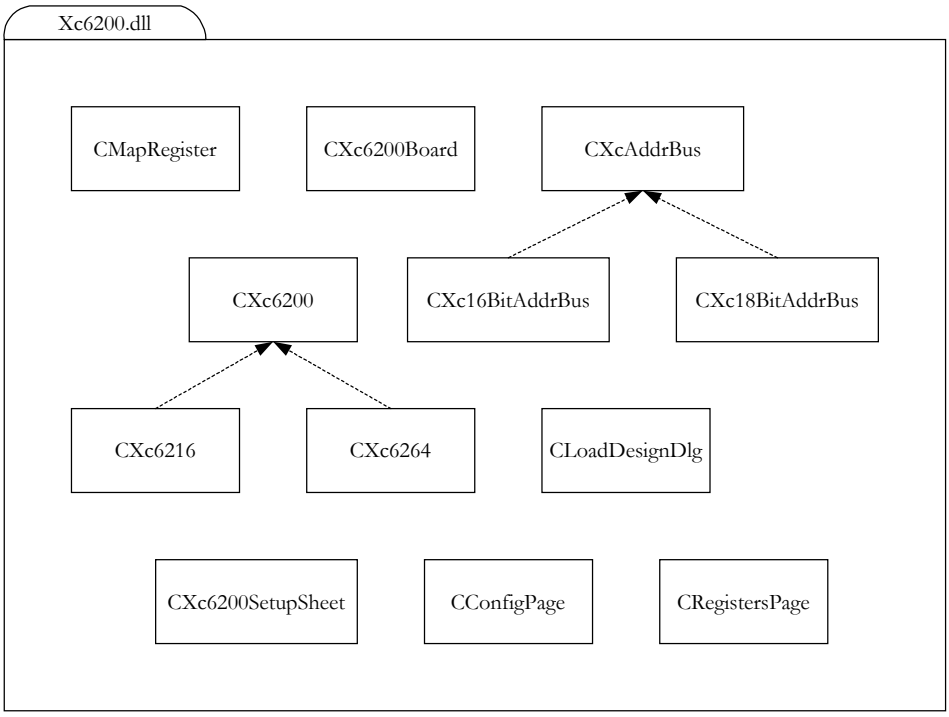


Figura 7.6 – Constituição do módulo *Xc6200.dll*.

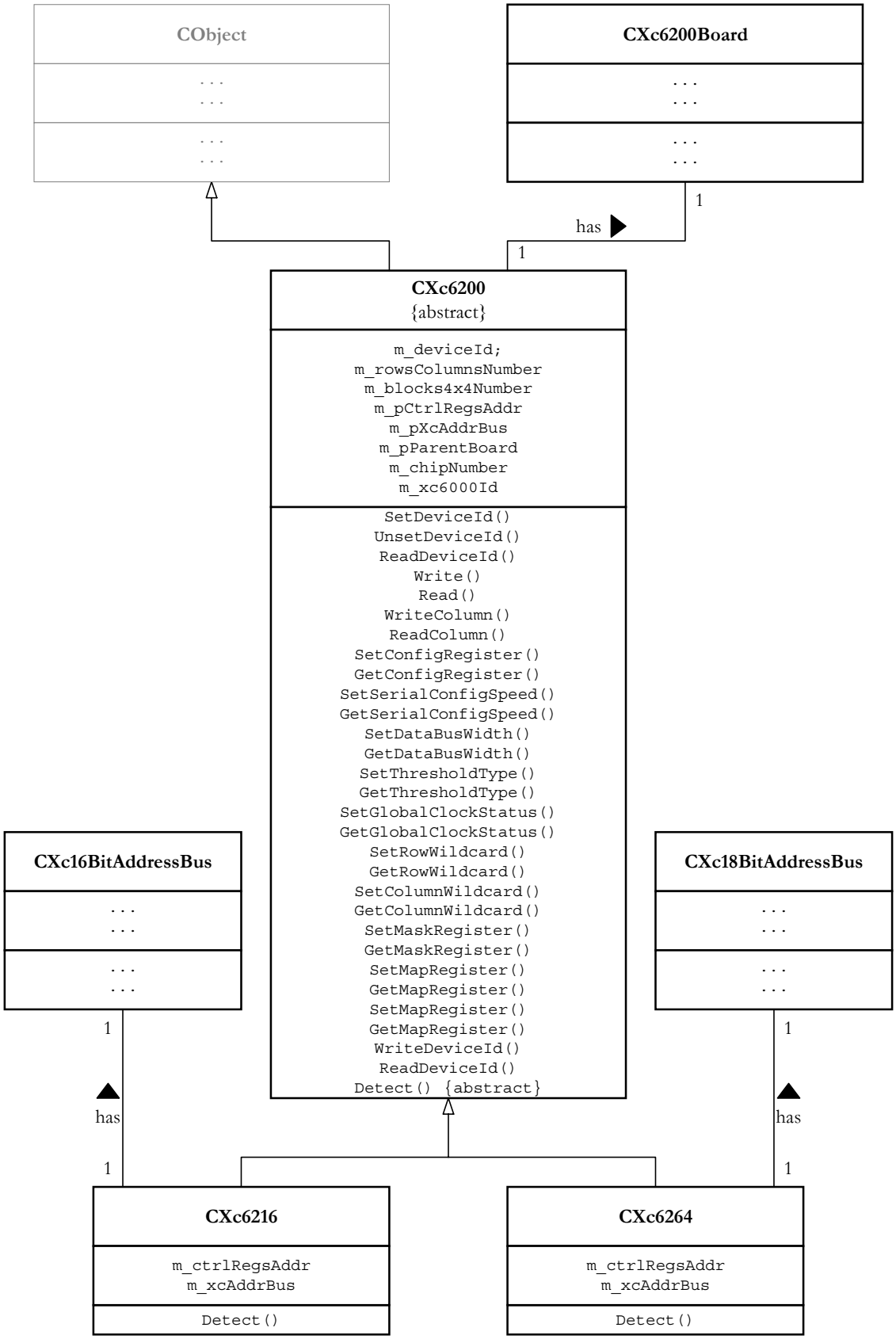


Figura 7.7 – Atributos, métodos e relações de herança e de associação das classes *CXc6200*, *CXc6216* e *CXc6264*.

As classes *CXcAddrBus* e suas derivadas (*CXc16BitAddrBus* e *CXc18BitAddrBus*) ilustradas no diagrama da Figura 7.8 representam, respectivamente, o barramento de endereços das FPGAs XC6216 e XC6264. A necessidade de definição destas classes deve-se à diferença da largura do barramento de endereços das duas FPGAs (Figura 7.9). Desta forma é possível uniformizar a manipulação dos campos de endereços.

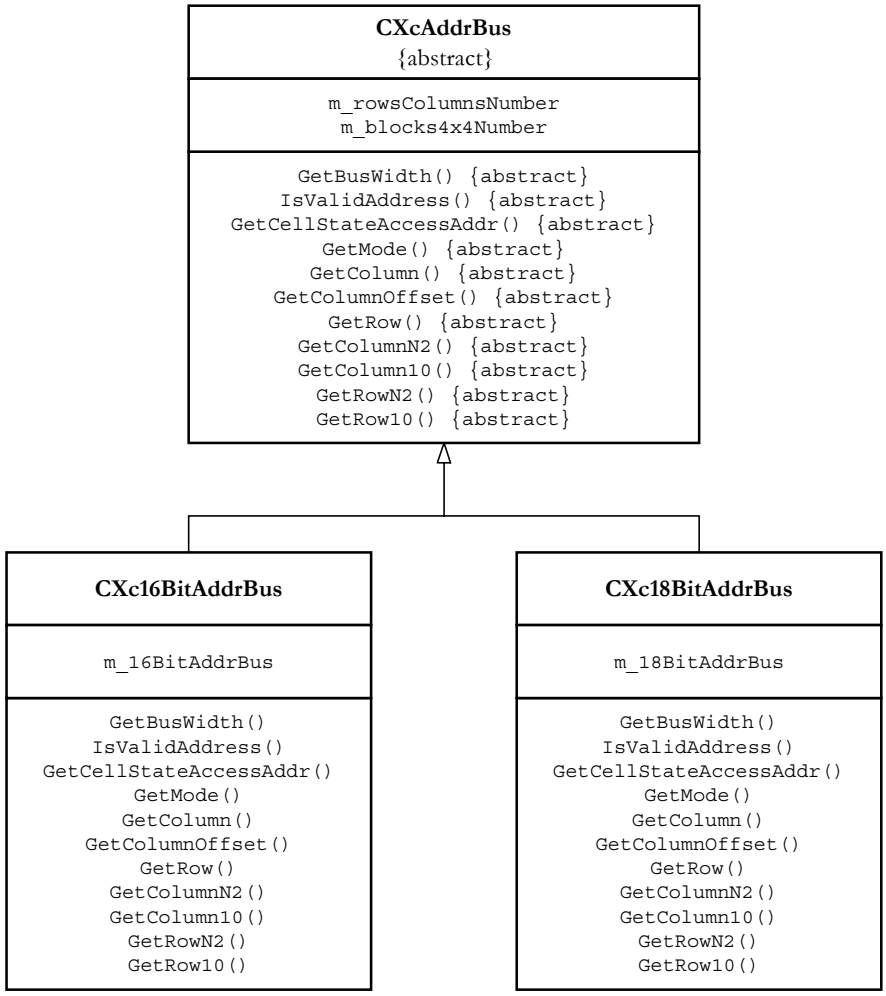


Figura 7.8 – Atributos, métodos e relações de herança das classes *CXcAddrBus, CXc16BitAddrBus* e *CXc18BitAddrBus*.

Modo	Coluna	*Offset* de Coluna	Linha
15:14	13:8	7:6	5:0

(a)

Modo	Coluna	*Offset* de Coluna	Linha
17:16	15:9	8:7	6:0

(b)

Figura 7.9 – Estrutura dos campos de endereçamento nas FPGAs (a) XC6216; (b) XC6264.

A classe *CMapRegister* (Figura 7.10) representa o registo de mapeamento existente nas FPGAs da família XC6200. Este registo é utilizado para mapear eficientemente as entradas e saídas das células no barramento de dados do dispositivo. Para uma descrição mais completa pode ser consultado o capítulo 2 desta dissertação.

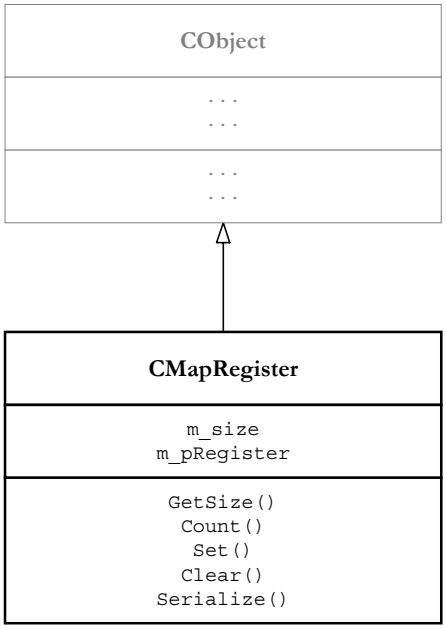


Figura 7.10 – Atributos, métodos e relações de herança da classe *CMapRegister*.

A classe *CXc6200Board* (Figura 7.11) é uma especialização da classe *CBoard* que suporta a representação de placas que possuam uma ou mais FPGAs desta família. As funções disponibilizadas permitem realizar operações de leitura/escrita nestas FPGAs, bem como obter ponteiros para acesso directo a objectos do tipo *CXc6216*/*CXc6264*.

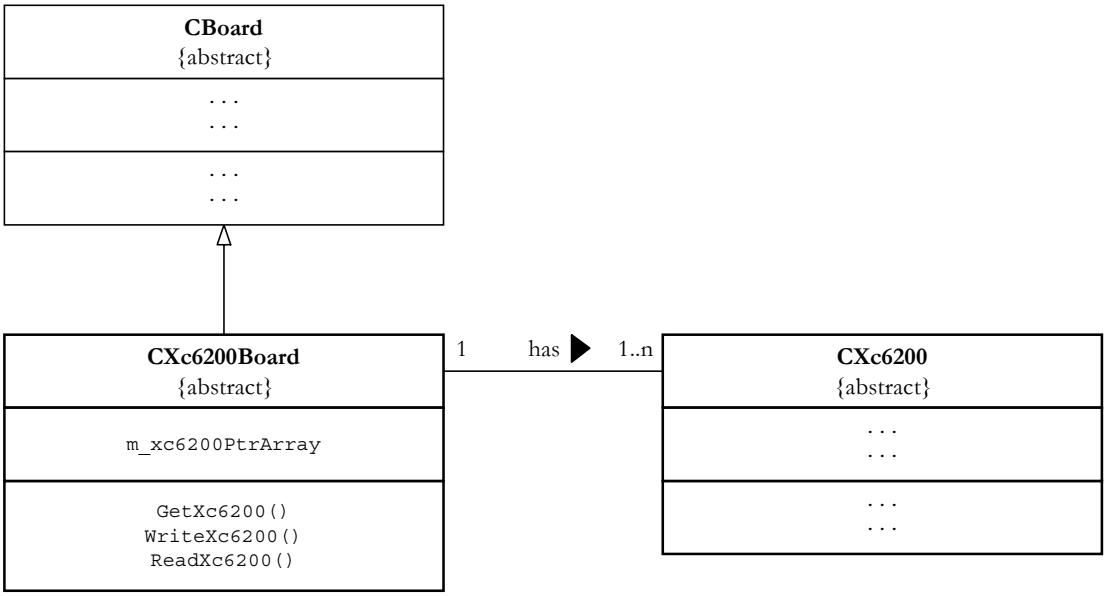


Figura 7.11 – Atributos, métodos e relações de herança e de associação das classes *CBoard*, *CXc6200Board* e *CXc6200*.

Para concluir a apresentação deste módulo, falta resumir as classes que representam os controlos gráficos implementados e que são as seguintes (Figura 7.12):

- *CLoadDesignDlg* – simplifica a abertura de um projecto, proporcionando controlos para navegar no sistema de ficheiros, bem como especificar o tipo de ficheiros que se pretende carregar (CAL/SYM/RAL);
- *CXc6200SetupSheet* – representa a janela que engloba as páginas de configuração descritas nos próximos dois pontos;
- *CConfigPage* – usada para configurar vários parâmetros da FPGA, tais como a velocidade de programação série, a largura do barramento de dados, etc. A Figura 7.13 mostra o aspecto visual deste controlo;
- *CRegistersPage* – permite configurar vários registos de controlo da FPGA, tais como o *Map Register*, o *Mask Register* e os *Wildcard Registers*.

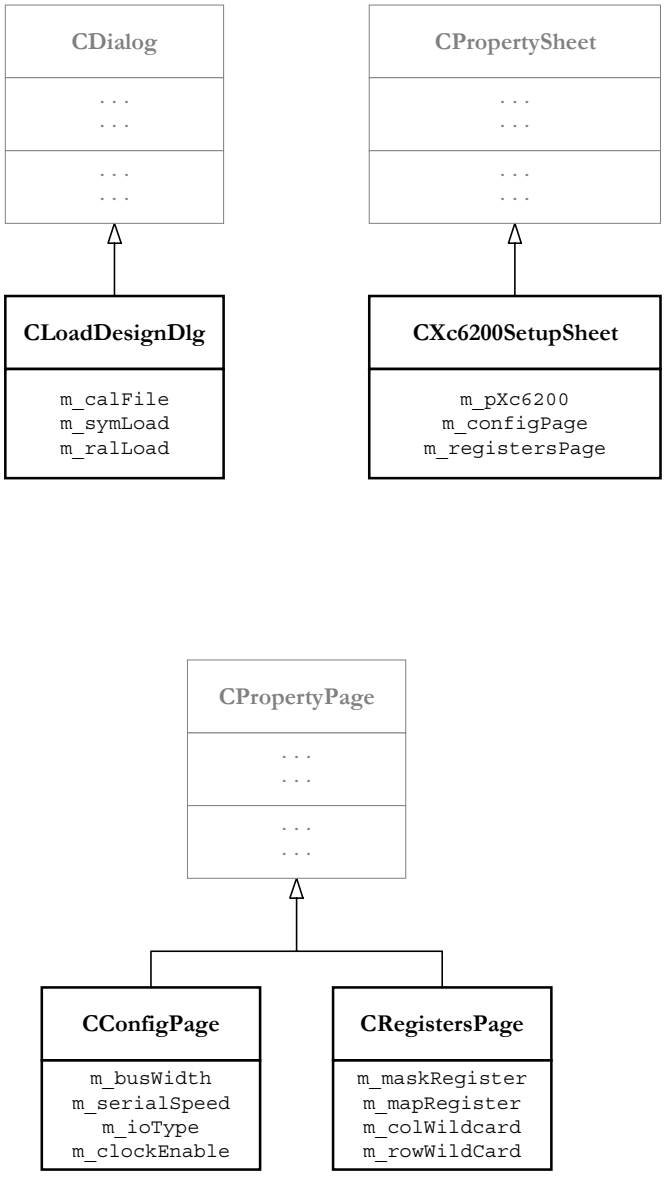


Figura 7.12 – Atributos, métodos e relações de herança das classes *CLoadDesignDlg, CXc6200SetupSheet, CConfigPage* e *CRegistersPage*.

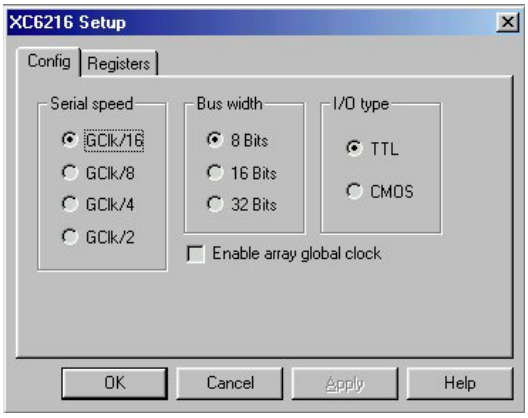


Figura 7.13 – Página para configuração dos parâmetros de interface de uma FPGA XC6200.

7.2.4 Módulo CalFile.dll

As classes implementadas neste módulo permitem ler os ficheiros de configuração (*.CAL) produzidos pela ferramenta de implementação XACT6000 e efectuar o seu carregamento no dispositivo alvo. Estes ficheiros possuem um formato de texto e estão organizados por linhas, cada uma contendo um comentário ou um par de valores hexadecimais (endereço, dados) para programação da FPGA (Figura 7.14).

```
# XACT6000 Version 1.1.7
#  Build number 7
#  Build Date Dec 10 1997
# mux WEST at 0x0 0x0, byte 0x0
0x4000  0xb0
# mux WEST at 0x0 0x1, byte 0x0
0x4001  0x70
# iob LEFT 0x1, byte 0x1
0x4141  0x3
# mux WEST at 0x0 0x2, byte 0x0
0x4002  0xb0
```

Figura 7.14 – Secção de um ficheiro de configuração (*.CAL).

As classes que constituem este módulo são mostradas na Figura 7.15. O melhoramento introduzido relativamente à implementação efectuada pela Xilinx na biblioteca PCIRAL é a separação clara entre as operações realizadas sobre os ficheiros de configuração e as classes de acesso à placa *FireFly*™. Para isso foi construída a classe *CCalFile* que representa os ficheiros deste tipo. Desta forma é possível a sua utilização com diferentes placas que possuam FPGAs desta família. As funcionalidades implementadas nesta classe permitem, além da leitura do ficheiro e do seu carregamento no dispositivo alvo, a realização das seguintes tarefas:

- Filtragem de endereços para garantir que o carregamento de um ficheiro de configuração na FPGA afecta somente as áreas que se pretende efectivamente modificar;
- Transformação do ficheiro de configuração ao nível dos endereços de forma a alterar a área do dispositivo onde o circuito vai ser carregado;

- Conversão para formato binário de forma a permitir uma leitura mais rápida e eficiente (uma vez que não é necessário realizar a sua interpretação).

De notar que as duas primeiras funcionalidades são bastante dificultadas pela forma como é feito o mapeamento das células de configuração no espaço de endereçamento da FPGA.

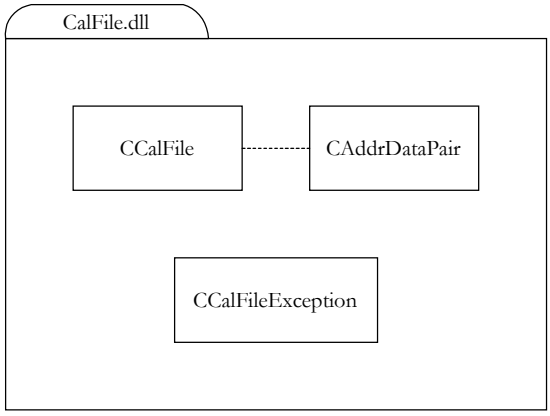


Figura 7.15 – Constituição do módulo *CalFile.dll*.

A Figura 7.16 ilustra as relações de herança e de associação existentes entre as classes deste módulo. A classe *CCalFile* é derivada da *CObject* para possibilitar a utilização dos mecanismos de serialização, úteis na leitura e escrita de ficheiros binários. Por outro lado, esta classe é constituída por uma matriz de estruturas *CAddrDataPair* que armazenam cada um dos pares (endereço, dados) do ficheiro de configuração.

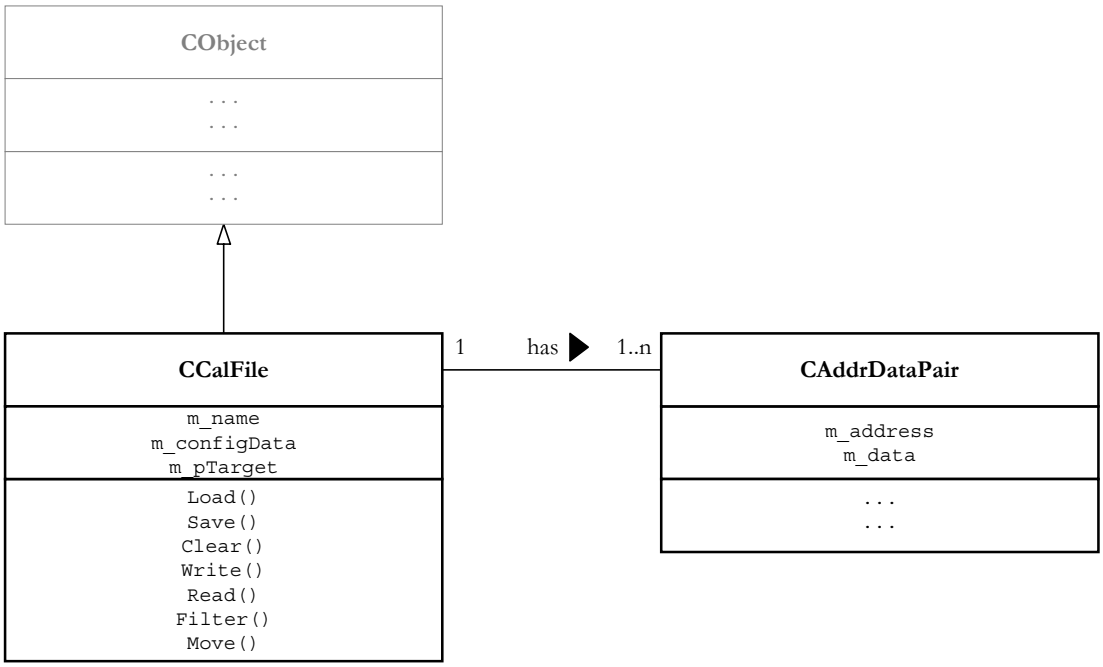


Figura 7.16 – Atributos, métodos e relações de herança e de associação das classes *CCalFIle* e *CAddrDataPair*.

7.2.5 Módulo RalLib.dll

O módulo *RalLib.dll* possui classes para representar as tabelas de símbolos e de reconfiguração geradas pela ferramenta de implementação XACT6000 e armazenadas nos ficheiros com a extensões SYM e RAL respectivamente.

O primeiro é um ficheiro de texto que possui informação sobre os componentes implementados na FPGA. Esta informação inclui o seu nome, o número de entradas e os respectivos nomes, o nome da saída (se existir), a sua localização, o tipo de lógica (combinatória/sequencial) e o modo de acesso permitido (só leitura/só escrita/ambas). Este ficheiro pode ser utilizado juntamente com as classes implementadas neste módulo para acesso de alto nível ao hardware ou para efeitos de depuração do circuito. O ficheiro que armazena a tabela de reconfiguração é binário e contém todas as configurações possíveis para os componentes que foram identificados como reconfiguráveis na lista de ligações fornecida à ferramenta de implementação. As classes implementadas neste módulo permitem a reconfiguração do dispositivo baseada neste ficheiro. Em [Xilinx97c] existem informações mais detalhadas sobre o formato destes ficheiros. Na Figura 7.17 é mostrada uma secção de um ficheiro de símbolos (*.SYM). As classes implementadas neste módulo estão ilustradas na Figura 7.18. No diagrama da Figura 7.19 são representadas as classes utilizadas no processamento dos ficheiros de símbolos, nomeadamente a *CRalSymTable*, a *CRalSymRecord*, a *CRalSimple*Record e a *CRalComposite*Record.

```
# Registo simples de lógica combinatória
s CELLS\STACK_MEM\CTRL\CNT\INV_0 0x1 FF_Q_0 0x1 INV_O_0 0x0 0x0 xc
# Registo simples de lógica sequencial
s CELLS\STACK_MEM\CTRL\CNT\FF_0 0x3 INV_O_0 CLK CLR 0x1 FF_Q_0 0x0 0x0 rs
# Registo composto
c CELLS\STACK_MEM\CTRL\CNT\REG 0x0 0xffffaaaa 0xffffffff r
```

Figura 7.17 – Secção de um ficheiro de símbolos (*.SYM).

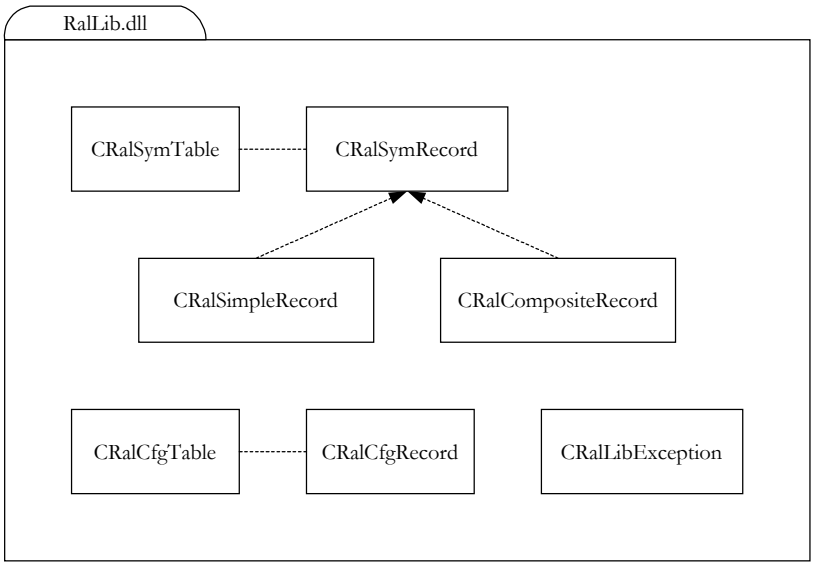


Figura 7.18 – Constituição do módulo *RalLib.dll*.

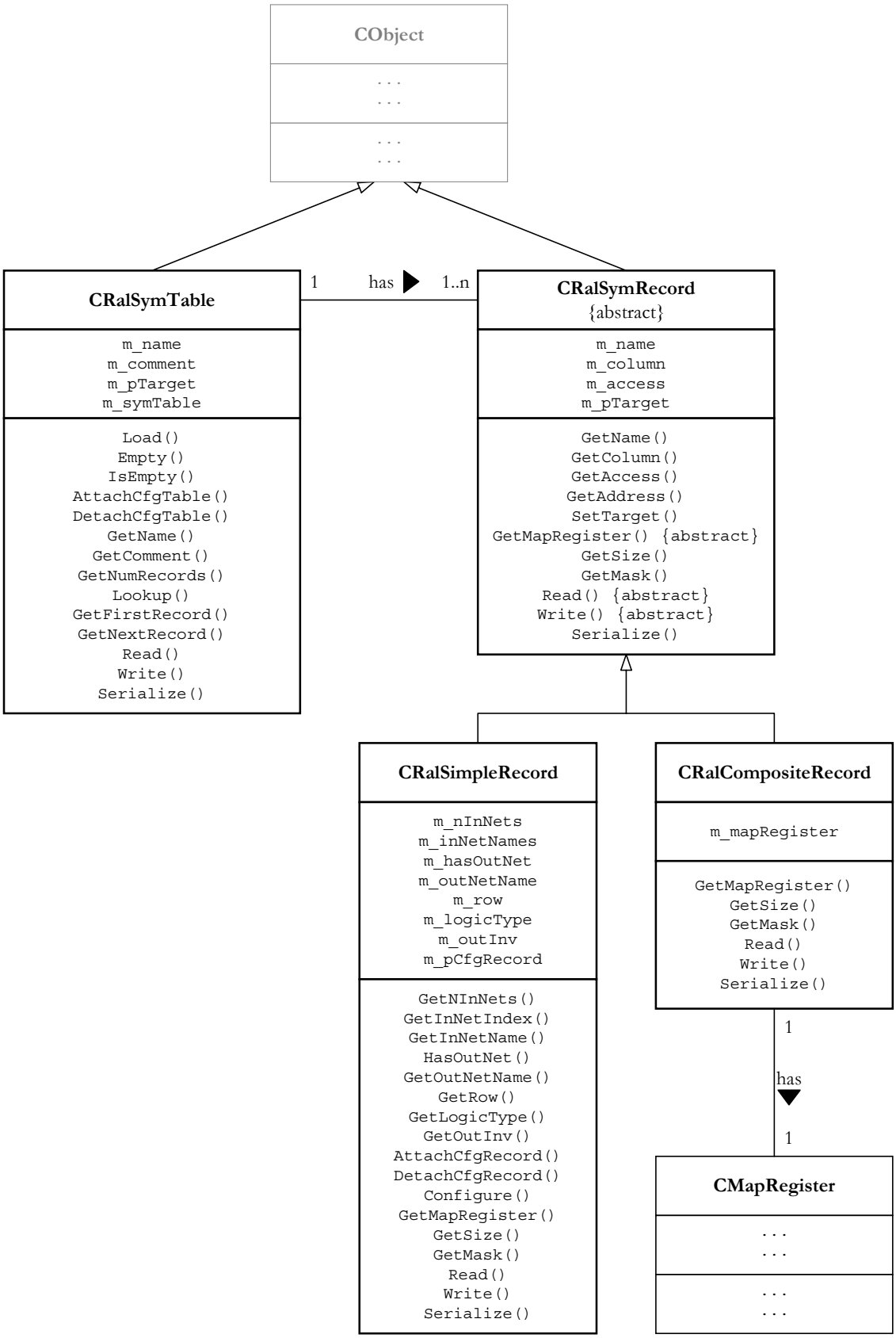


Figura 7.19 – Atributos, métodos e relações de herança e de associação das classes *CRalSymTable*, *CRalSymRecord*, *CRalSimpleRecord* e *CRalCompositeRecord*.

A classe *CRalSymTable*, representa a tabela de símbolos gerada pela aplicação XACT6000, a qual é constituída por registos simples ou compostos (Figura 7.17) que são representados, respectivamente, por objectos do tipo *CRalSimpleRecord* e *CRalCompositeRecord*. A classe *CRalSymRecord*, é utilizada para implementar os aspectos comuns aos dois tipos de registos.

Relativamente à biblioteca RALLIB fornecida pela Xilinx, foram introduzidos essencialmente dois melhoramentos: a capacidade para leitura e escrita directa de células de uma FPGA representadas na tabela de símbolos e a possibilidade de associação de uma tabela de reconfiguração à tabela de símbolos, o que simplifica imenso o processo de reconfiguração. Desta forma, algumas das tarefas que anteriormente necessitavam de ser implementadas pela aplicação que utilizava a biblioteca RALLIB, estão agora completamente encapsuladas na IMPARLIB.

Para concluir a discussão deste módulo falta apresentar as classes *CRalCfgTable* e *CRalCfgRecord* cujas relações de herança e de associação estão apresentadas na Figura 7.20. Um objecto da classe *CRalCfgTable* armazena o conteúdo de um ficheiro RAL, ou seja, a tabela de reconfiguração de um projecto. Esta tabela é constituída por vários registos, os quais são instâncias da classe *CRalCfgRecord*. Estas classes destinam-se a operar em estreita cooperação com as que representam a tabela de símbolos.

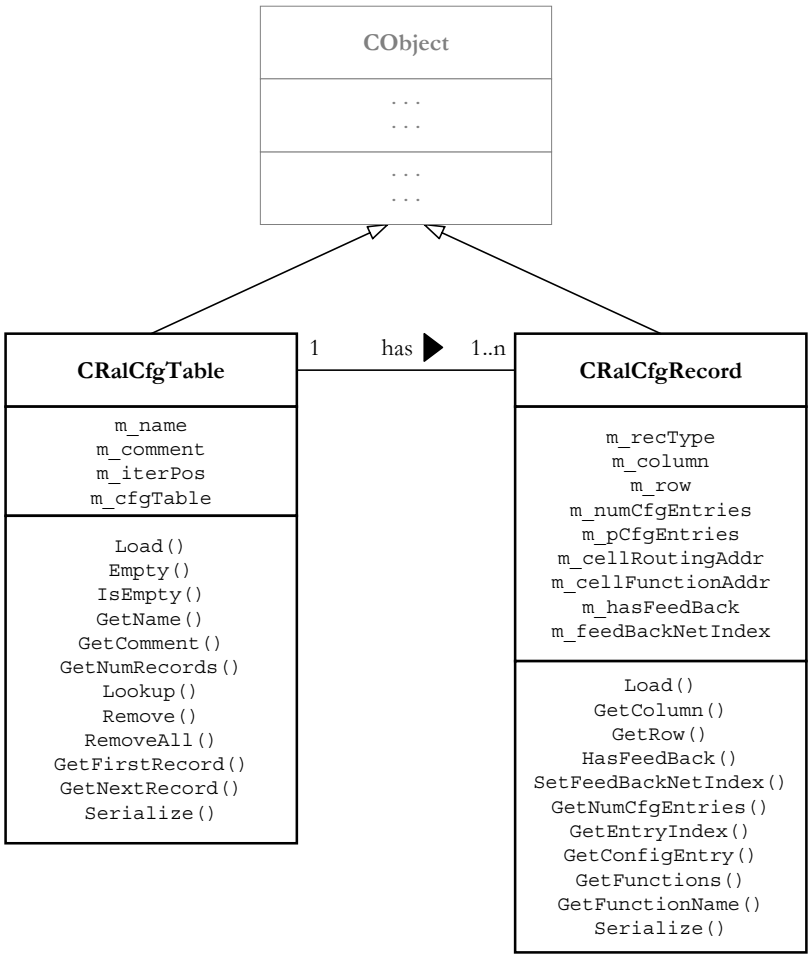


Figura 7.20 – Atributos, métodos e relações de herança e de associação das classes *CRalCfgTable* e *CRalCfgRecord*.

7.2.6 Módulo FireFly.dll

O último módulo desta biblioteca é o *FireFly.dll*. As classes que o constituem são as seguintes (Figura 7.21):

- *CFireFlyBoard*;
- *CClockAdvDlg*;
- *CLedCtrl*;
- *CFireflySetupSheet*, *CMemoryPage*, *CClockPage*, *CPowerPage* e *CInterruptsPage*.

A classe principal deste módulo é a *CFireFlyBoard*. Esta classe é derivada da *CXc6200Board* apresentada acima, uma vez que a placa *FireFly*™ é um tipo particular de uma placa que possui uma FPGA XC6200. Os parâmetros configuráveis desta placa podem ser divididos nos seguintes quatro grupos:

- Gestão dos bancos de memória SRAM;
- Geração dos sinais de relógio;
- Monitorização do consumo de potência;
- Controlo de interrupções.

Os métodos disponibilizados na classe *CFireFlyBoard* permitem o controlo por programação de todos estes parâmetros. Os atributos, métodos e relação de herança desta classe com a *CXc6200Board* estão representados na Figura 7.22.

Tal como já foi referido atrás, relativamente à biblioteca PCIRAL implementada pela Xilinx, a diferença fundamental foi a mudança para uma classe separada das funções responsáveis pelo carregamento dos ficheiros de configuração. Além disso, o interface com o controlador de software passou para a classe *CPciInterface*, uma vez que uma parte significativa desta classe não depende da placa que está a ser utilizada.

Além da classe *CFireFlyBoard*, são também disponibilizadas outras que representam controlos gráficos. Estes podem ser utilizadas para realizar a configuração da placa de forma gráfica e utilizam os serviços da *CFireflyBoard*. As relações de herança destas classes são representadas na Figura 7.23. Todas elas são derivadas de classes da biblioteca MFC. Na Figura 7.24 e seguintes são mostrados os vários controlos disponibilizados para a configuração visual dos parâmetros da placa *FireFly*™.

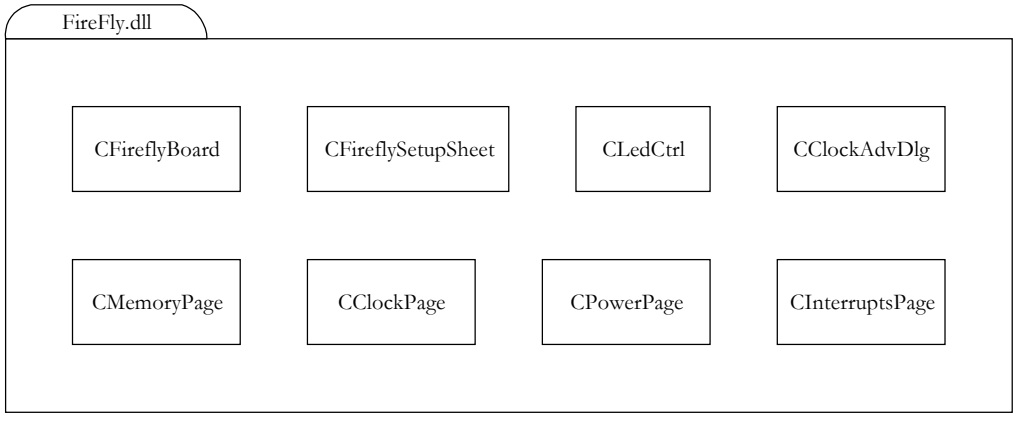


Figura 7.21 – Constituição do módulo *FireFly.dll*.

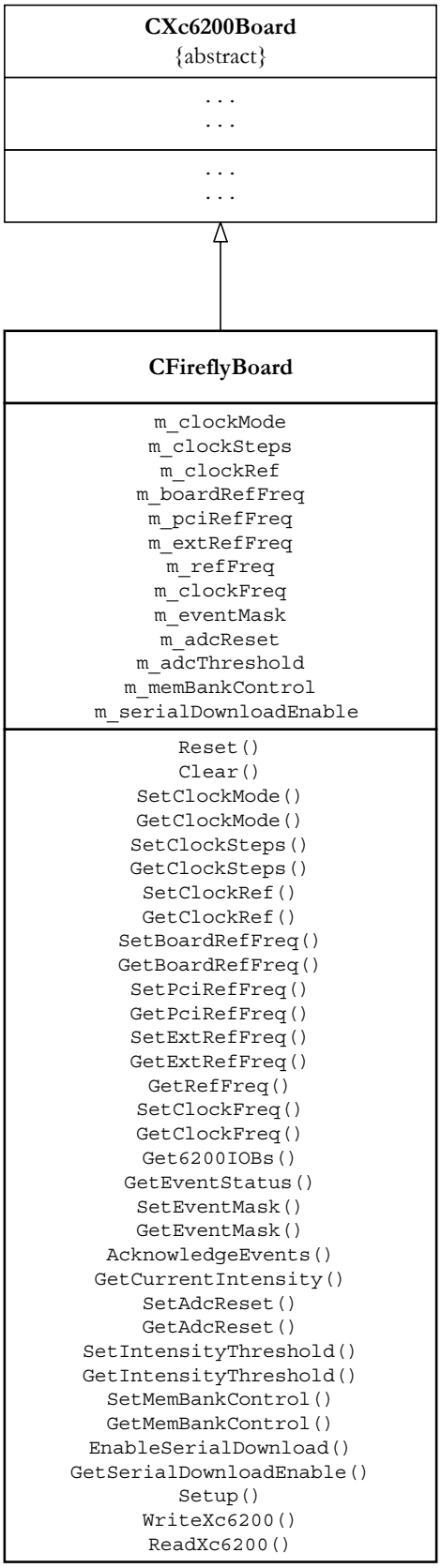


Figura 7.22 – Atributos, métodos e relações de herança das classes *CXc6200Board*, *CFireflyBoard*.

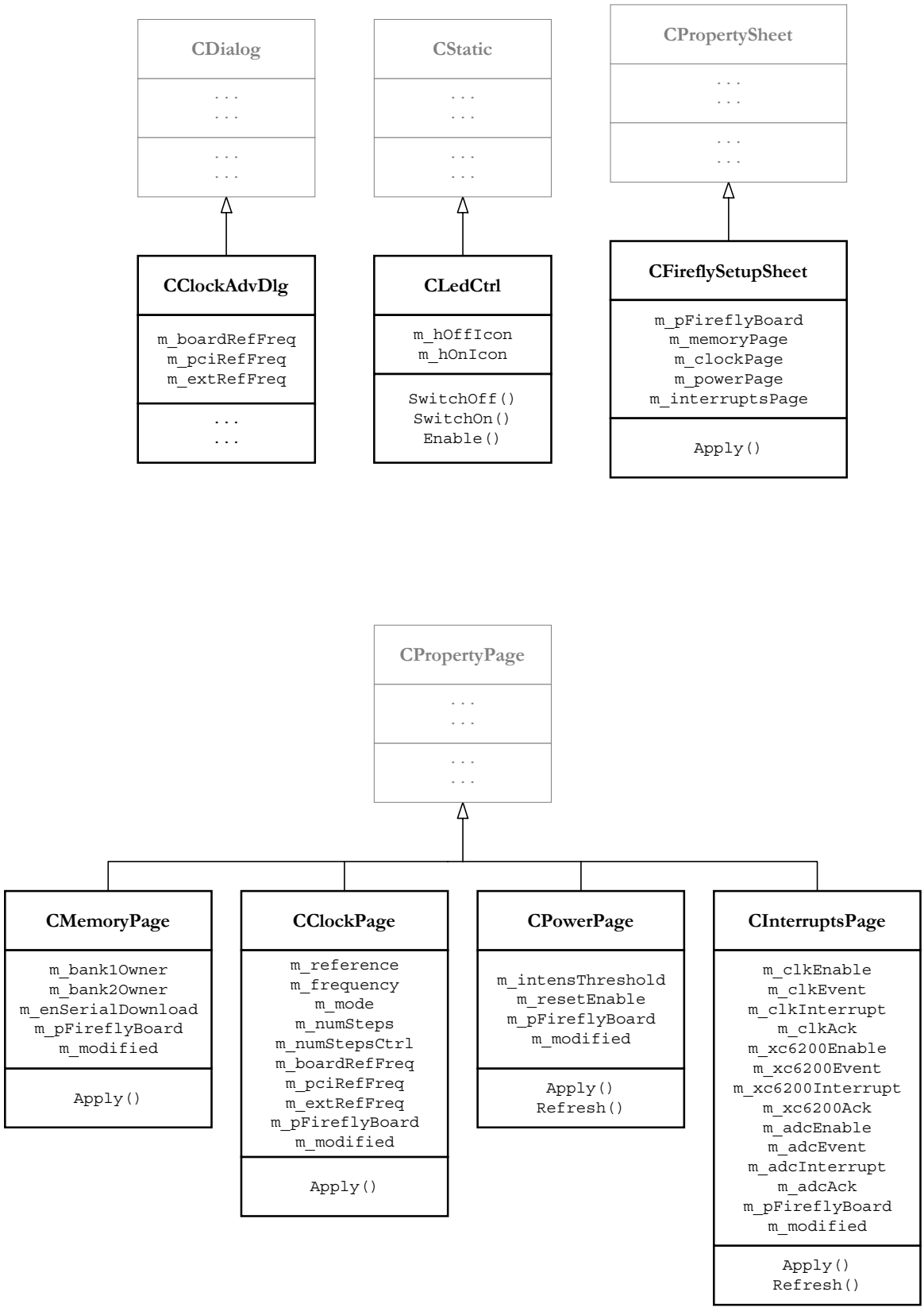


Figura 7.23 – Atributos, métodos e relações de herança e de associação das classes *CClockAdvDlg*, *CLedCtrl*, *CFireflySetupSheet*, *CMemoryPage*, *CClockPage*, *CPowerPage* e *CInterruptsPage*.

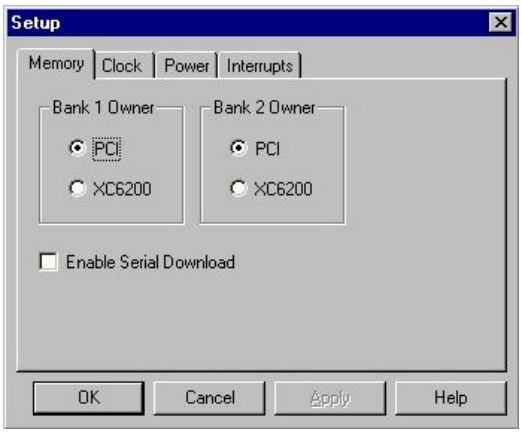


Figura 7.24 – Página para configuração dos parâmetros relativos aos bancos de memória da placa *FireFly*™.

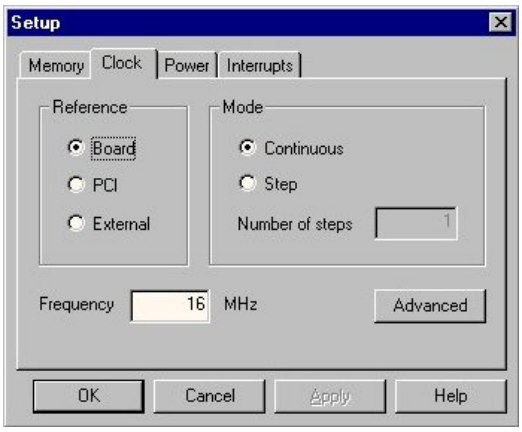


Figura 7.25 – Página para configuração dos parâmetros relativos ao gerador dos sinais de relógio da placa *FireFly*™.

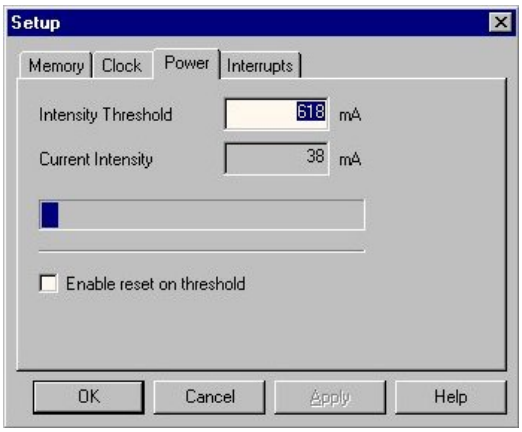


Figura 7.26 – Página para configuração dos parâmetros relativos ao controlo do consumo de potência da placa *FireFly*™.

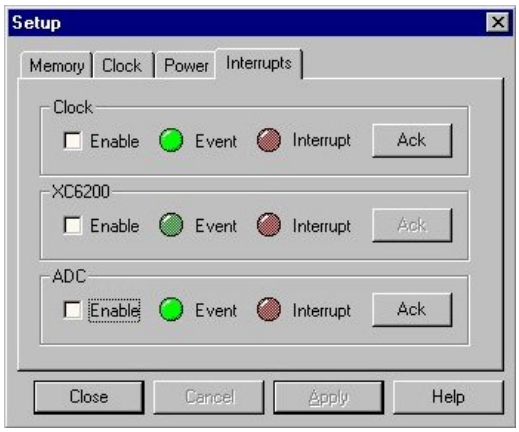


Figura 7.27 – Página para configuração dos parâmetros relativos às interrupções da placa *FireFly*™.

7.3 Controlador de Software para a Placa FireFly™

No âmbito deste trabalho foi também desenvolvido um controlador de software para a placa de desenvolvimento *FireFly*™. Este controlador implementa os mecanismos de *plug-n-play* e é baseado na arquitectura WDM (*Windows Driver Model*), o que permite a sua instalação e execução nos sistemas operativos Microsoft Windows 98 e Windows 2000, sem quaisquer modificações ao nível do código fonte. O seu desenvolvimento foi motivado pela incapacidade do controlador desenvolvido pela Xilinx e distribuído com a placa *FireFly*™ em processar correctamente as interrupções geradas por esta. A vantagem mais importante resultante do desenvolvimento deste módulo é a facilidade com que podem ser introduzidas optimizações ao nível da comunicação entre o hardware e a aplicação de controlo. Do ponto de vista de uma aplicação, as funcionalidades implementadas mais relevantes foram as seguintes:

- Abertura e fecho do dispositivo com as funções *CreateFile* e *CloseFile* da API (*Application Program Interface*) de Win32;
- Acesso à memória e ao espaço de endereçamento de entrada/saída com as funções *WriteFile* e *ReadFile* de Win32;
- Mapeamento da memória SRAM da placa *FireFly*™ no espaço de endereçamento da aplicação de utilizador para optimização dos acessos;
- Gestão de interrupções a partir de uma aplicação de utilizador;
- Outras funcionalidades (ex. informação sobre os endereços base da placa) acessíveis com a função *DeviceIoControl* de Win32.

O processamento de interrupções a partir de uma aplicação de utilizador é realizado em conjunto com a classe *CPciBoardInterface* apresentada acima. Qualquer aplicação que pretenda ser notificada da ocorrência de uma interrupção necessita apenas de registar uma função que será invocada por um objecto da classe *CPciBoardInterface*. De notar que, a forma como este processo está implementado possui uma certa sobrecarga associada. No entanto, uma das vantagens em desenvolver um controlador é precisamente a possibilidade de transferir para núcleo do sistema operativo funções com restrições temporais críticas. Na Figura 7.28 é ilustrada de forma simplificada a organização dos vários módulos que constituem o controlador desenvolvido, assim como as posições relativas das várias partes que constituem o sistema: a placa de desenvolvimento *FireFly*™, o seu controlador de software, o sistema operativo e, finalmente, a aplicação de utilizador, que pode ser um programa que utilize a placa como coprocessador reconfigurável, uma ferramenta de depuração, etc.

Os ficheiros *PciBoardIoCtrl.h* e *BoardIoTypes.h* contêm, respectivamente, as definições dos códigos de controlo e das estruturas de dados utilizadas na transferência de dados entre a aplicação e o controlador, pelo que são partilhados por ambos. A função e o conteúdo dos restantes módulos pode ser resumida da seguinte forma:

- *FireflyMain* – contém funções de inicialização e de suporte aos outros módulos que servem para manipular os pacotes com pedidos de entrada/saída (*I/O Request Packets – IRPs*) construídos pelo sistema, na sequência de uma chamada realizada no contexto de uma aplicação;
- *FireflyRW* – contém as rotinas chamadas pelo sistema operativo na sequência de uma invocação das funções *ReadFile*/*WriteFile* de *Win32* a partir de uma aplicação de utilizador com um identificador deste dispositivo;
- *FireflyDevCtrl* – contém as rotinas chamadas pelo sistema operativo na sequência de uma invocação da função *DeviceIoControl* de *Win32* a partir de uma aplicação de utilizador com um identificador deste dispositivo;
- *FireflyPnp* – contém as rotinas chamadas pelo gestor de *plug-n-play* do sistema operativo utilizadas para adicionar e remover dispositivos bem como iniciar ou terminar o seu funcionamento;
- *FireflyPower* – contém as rotinas chamadas pelo gestor de consumo de potência do sistema operativo utilizadas para suspender ou retomar o funcionamento do hardware;

- *FireflyIo* – contém as rotinas chamadas pelo sub-sistema de entrada/saída do sistema operativo que são responsáveis por executar de forma sequencial os vários pedidos de operações de entrada/saída armazenados numa lista interna de *IRPs*.

Para terminar este capítulo, na Figura 7.29 e seguinte são mostradas as páginas de propriedades do controlador desenvolvido a executar em Windows 98. No anexo V encontra-se listado o código fonte de todos os módulos apresentados na Figura 7.28.

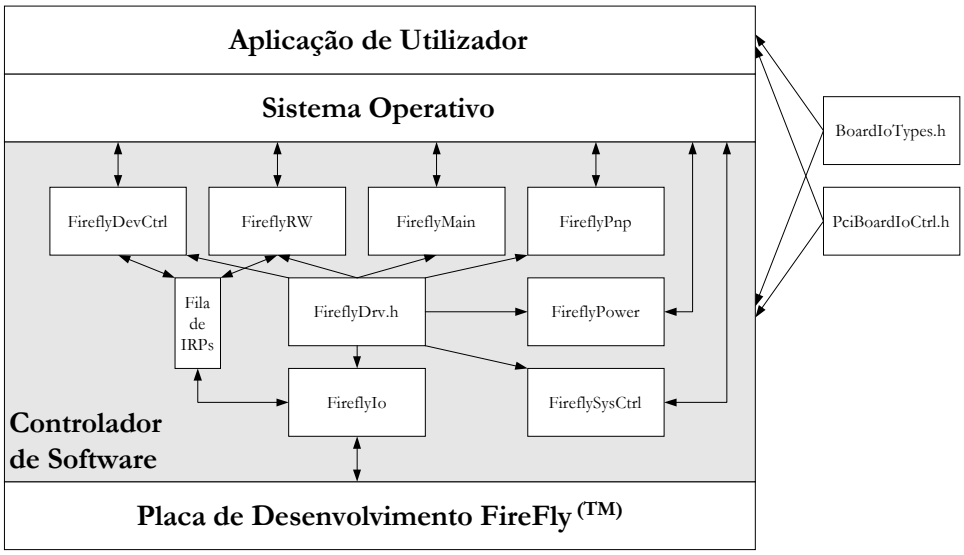


Figura 7.28 – Representação simplificada dos módulos que constituem o controlador de software desenvolvido para a placa de desenvolvimento *FireFly*™.

Figura 7.29 – Página de informação geral sobre o controlador desenvolvido.

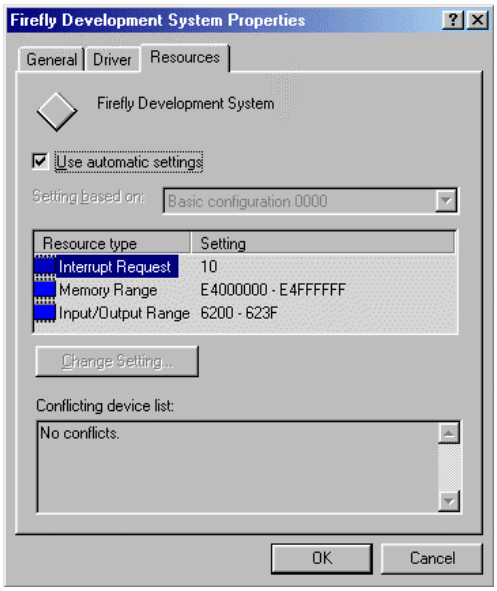


Figura 7.30 – Página de informação sobre os recursos utilizados pela placa *FireFly*™.

8 Conclusões e Trabalho Futuro

Sumário

Este capítulo conclui a dissertação com a apresentação de um resumo da abordagem proposta para o projecto de unidades de controlo virtuais a implementar em FPGAs reconfiguráveis dinamicamente da família XC6200 da Xilinx.

Seguidamente é apresentada uma compilação de todas as contribuições originais resultantes deste trabalho.

Finalmente, são discutidas algumas ideias para trabalho futuro. As direcções propostas são essencialmente duas. A primeira está relacionada com o projecto de unidades de controlo virtuais em famílias de FPGAs mais recentes, em particular a *Virtex* da Xilinx, ou outras com potencialidades equivalentes, principalmente ao nível da existência de blocos de memória integrados. Em segundo lugar são discutidas algumas ideias sobre a implementação de unidades de controlo paralelas, hierárquicas e virtuais baseadas nos modelos HPFSM, PHFSM e GFSM descritos no capítulo 3.

8.1 Conclusões

As unidades de controlo virtuais podem ser um dos elementos dos sistemas reconfiguráveis. Devido à sua especificidade, irregularidade e falta de modularidade possuem requisitos de projecto especiais de forma a simplificar a sua reconfiguração.

Esta dissertação procurou cobrir os aspectos mais importantes do projecto de unidades de controlo virtuais. Ao longo dos vários capítulos tentou-se fornecer o máximo de informação possível, tendo sido abordados os seguintes temas: o projecto de sistemas digitais, os dispositivos lógicos programáveis, o papel de uma unidade de controlo num sistema electrónico digital, os seus modelos e arquitecturas de implementação, os processos de especificação e de síntese e, finalmente, os aspectos específicos do projecto e da reconfiguração de unidades de controlo virtuais. A abordagem de projecto proposta nesta dissertação para alcançar os objectivos descritos no primeiro capítulo, nomeadamente, a flexibilidade, a extensibilidade, a reconfigurabilidade e a reutilização, baseia-se nos seguintes aspectos:

- Adopção de um modelo multi-nível baseado em máquinas de estados finitos hierárquicas, o qual possibilita o projecto estruturado de unidades de controlo e permite concentrar todos os esforços em cada nível de abstracção separadamente.
- Especificação hierárquica do comportamento da unidade de controlo com a linguagem HGS, a qual permite tirar partido de todas as capacidades do modelo HFSM e tem a vantagem de proporcionar uma descrição algorítmica independente da implementação. A decomposição hierárquica de um algoritmo em sub-algoritmos permite também em casos especiais a sua reutilização em projectos futuros.
- Utilização de técnicas de síntese apropriadas ao dispositivo alvo. Um dos aspectos mais importantes da síntese sequencial é a codificação de estados. Para permitir uma utilização eficiente dos recursos da FPGA XC6200, a técnica escolhida foi a *one-hot*.
- Concepção de uma arquitectura baseada numa estrutura predefinida, multi-nível, modular, parametrizável e escalável que possa de forma simples ser adaptada a unidades de controlo de diferentes complexidades. Esta assenta na arquitectura do controlador hierárquico com pilha de memória, à qual foram acrescentadas algumas funcionalidades para simplificar o projecto, tal como codificação/descodificação de estados binário↔*one-hot*.
- Implementação da arquitectura desenvolvida em dispositivos reconfiguráveis dinâmica e parcialmente para permitir a reutilização dos mesmos recursos em diferentes partes de um algoritmo de controlo. Assim, é possível implementar unidades de controlo complexas com recursos de hardware limitados e modificar ou estender o seu comportamento sempre que necessário.
- Estabelecimento das restrições adequadas que simplifiquem a partilha dos recursos lógicos e de interligação do dispositivo entre a estrutura predefinida e as partes dependentes da aplicação.

8.2 Contribuições Originais

As contribuições originais resultantes deste trabalho apresentadas ao longo desta dissertação podem ser resumidas da seguinte forma:

- Apresentação de uma taxonomia dos modelos usados no projecto de sistemas digitais baseada numa representação ortogonal entre os níveis de granulosidade dos componentes utilizados e o tipo de representação de cada um. Esta taxonomia pretende unificar e tornar mais claras as diferentes classificações publicadas pelos diversos autores e que são muitas vezes utilizadas de forma pouco consistente.
- Elaboração da definição formal de uma máquina de estados finitos hierárquica baseada numa estrutura multi-nível e recursiva. A definição é baseada no facto de uma máquina hierárquica ser constituída por um conjunto de estados atómicos e um conjunto de macroestados. A interligação entre os vários níveis hierárquicos é realizada pelos macroestados, em que cada um é ele próprio definido como uma máquina hierárquica.
- Investigação de uma arquitectura para implementação de unidades de controlo virtuais que seja escalável, parametrizável, e que encapsule todas as funcionalidades normalmente necessárias em qualquer unidade de controlo independentemente da aplicação alvo. Desta forma é possível simplificar o processo de projecto, uma vez que só é necessário conceber e implementar as partes específicas da aplicação. A arquitectura proposta permite alcançar os objectivos deste trabalho, nomeadamente a flexibilidade a extensibilidade e a reutilização.
- Construção de uma biblioteca de componentes descritos em VHDL e parametrizáveis de forma a simplificar o projecto de sistemas a implementar na família de FPGAs XC6200 da Xilinx.
- Desenvolvimento de software baseado em metodologias orientadas por objectos para abstracção do dispositivo lógico e da placa de desenvolvimento utilizada, processamento e carregamento dos ficheiros de configuração e reconfiguração do circuito. Mais concretamente, foi desenvolvida uma biblioteca de classes em C++ e um controlador de software para a placa de desenvolvimento *FireFly*™.

8.3 Trabalho Futuro

As propostas relativas ao trabalho futuro dividem-se essencialmente em dois pontos principais, que são:

- A averiguação da possibilidade de implementação da arquitectura desenvolvida num único dispositivo lógico programável de elevada capacidade, em princípio uma FPGA, de forma a minimizar o número de circuito integrados do sistema e a maximizar a sua velocidade de operação;
- A investigação de arquitecturas eficientes para implementação de unidades de controlo paralelas, hierárquicas e virtuais.

Uma das limitações encontradas no desenvolvimento da arquitectura apresentada nesta dissertação foi a impossibilidade de alojar todos os seus componentes no interior da FPGA. Mais concretamente, a pilha de memória teve de ser implementada em circuitos integrados SRAM externos devido à quantidade considerável de células e recursos de comutação e descodificação que seriam necessários se a sua implementação fosse realizada no interior da FPGA XC6200.

O aparecimento de FPGAs com blocos de memória de dimensão razoável, 256 a 4Kbytes na arquitectura *Virtex* da Xilinx [Xilinx99], tornou possível a integração de todos os componentes de uma unidade de controlo hierárquica num único encapsulamento. No entanto, como esta família em termos de reconfiguração é bastante diferente da XC6200, será necessário fazer um estudo para averiguar a sua adequação à implementação de unidades de controlo virtuais.

O segundo tópico passível de trabalho futuro está relacionado com a implementação eficiente de unidades de controlo paralelas, hierárquicas e eventualmente virtuais. Tendo por base as noções apresentadas no capítulo 3, a implementação de hierarquia e paralelismo numa unidade de controlo pode ser baseada em três variantes do modelo HaPFSM, nomeadamente o GFSM, o HPFSM e o PHFSM. O primeiro é o mais geral, mas também o de implementação mais complexa, uma vez que a hierarquia e o paralelismo podem ser combinados de qualquer forma, podendo existir num dado momento um número elevado de máquinas activas. Do ponto de vista de representação gráfica, a relação de invocações pode ser descrita por uma árvore, na qual podem existir vários nodos activos simultaneamente. No entanto, pode também acontecer que num dado instante o número de máquinas activas seja reduzido, pelo que interessa minimizar a quantidade de hardware subaproveitado. Assim, poderá ser interessante investigar a possibilidade de utilização de uma matriz de blocos de implementação semelhantes ao apresentado no capítulo 6, mas em que podem ser activados vários simultaneamente e em que a pilha de memória é partilhada e utilizada somente durante as transições hierárquicas. Contudo, esta abordagem necessita de uma sincronização mais complexa de forma a detectar as transições hierárquicas das diversas máquinas em intervalos de tempo diferentes. De notar que esta técnica difere da proposta em [Sklyarov87] no facto das máquinas poderem, com a excepção das transições hierárquicas, funcionar separadamente, isto é transitarem de estado e calcularem as suas saídas independentemente das restantes. Os modelos HPFSM e PHFSM são mais restritivos mas também mais fáceis de implementar, podendo ainda assim ser adequados para muitas aplicações. O modelo HPFSM pode ser implementado com um conjunto de máquinas hierárquicas, cada uma com a sua pilha de memória. A execução das diversas máquinas paralelas pode eventualmente ser sincronizada. Cada uma efectua transições hierárquicas independentemente das restantes. Finalmente, o modelo PHFSM pode ser implementado apenas com uma pilha de memória e um banco de registos que é utilizado quando for invocada uma máquina paralela. O número de registos não deve ser inferior ao número máximo de máquinas paralelas que podem ser invocadas simultaneamente. A apresentação destas ideias para trabalho futuro encerra o corpo da dissertação. A seguir são apresentados os anexos e as listas de referências e de acrónimos.

Anexo I – Listagens das PLAs e Equações Multi-nível

Sumário

Neste anexo são apresentadas as listagens dos vários ficheiros utilizados na optimização automática da componente combinatória da unidade de controlo da máquina de venda automática e cujos resultados foram resumidos no capítulo 5.

Tal como já foi aí referido foram utilizadas as seguintes codificações de estado:

- Binária;
- *Gray;*
- Distância total mínima (DTM);
- Distância mínima entre os estados de entrada e de saída (DMEES).

Para cada uma das codificações de estado acima enumeradas são apresentadas as seguintes listagens:

- Descrição inicial (sem qualquer optimização);
- PLA optimizada (descrição de dois níveis minimizada);
- Equações multi-nível (descrição multi-nível minimizada).

A optimização da descrição inicial para uma implementação de dois níveis foi realizada com a aplicação *Espresso* [BraHacMcMSan84] enquanto para uma implementação multi-nível foi usado o *misII* [BraRudSanWan87].

Codificação de Estados Binária

Descrição Inicial

```
.i 7
.o 7
.ilb Q3 Q2 Q1 M50 M100 Continuar Cancelar
.ob D3 D2 D1 Lata Troco Devolucao Rejeicao
.type fr

000     1---    001     0000
000     01--    010     0000
000     00--    000     0000

001     ---1    111     0000
001     1--0    010     0000
001     01-0    011     0000
001     00-0    001     0000

010     ---1    111     0000
010     1--0    011     0000
010     01-0    100     0000
010     00-0    010     0000

011     ---1    111     0001
011     --10    101     0001
011     --00    011     0001

100     ---1    111     0001
100     --10    110     0001
100     --00    100     0001

101     ----    000     1001

110     ----    000     1101

111     ----    000     0011

.e
```

PLA Optimizada

```
# espresso -Dcheck -Dexact -s -t MV_EntDir.pla
# UC Berkeley, Espresso Version #2.3, Release date 01/31/88
# READ         Time was 0.00 sec, cost is c=19(19) in=88 out=42 tot=130
# COMPL        Time was 0.00 sec, cost is c=20(20) in=93 out=98 tot=191
# PLA is MV_EntDir.pla with 7 inputs and 7 outputs
# ON-set cost is  c=19(19) in=88 out=42 tot=130
# OFF-set cost is c=20(20) in=93 out=98 tot=191
# DC-set cost is  c=0(0) in=0 out=0 tot=0
# PRIMES       Time was 0.01 sec, cost is c=35(35) in=130 out=61 tot=191
# ESSENTIALS   Time was 0.00 sec, cost is c=16(16) in=57 out=33 tot=90
# PI-TABLE     Time was 0.00 sec, cost is c=10(10) in=37 out=18 tot=55
# MINCOV       Time was 0.00 sec, cost is c=35(35) in=130 out=61 tot=191
# MV_REDUCE    Time was 0.00 sec, cost is c=17(17) in=61 out=31 tot=92
# RAISE_IN     Time was 0.01 sec, cost is c=17(0) in=61 out=31 tot=92
# VERIFY       Time was 0.00 sec, cost is c=17(0) in=61 out=31 tot=92
# READ          1 call(s) for 0.00 sec ( 0.0%)
# COMPL         1 call(s) for 0.00 sec ( 0.0%)
# MV_REDUCE     1 call(s) for 0.00 sec ( 0.0%)
# RAISE_IN      1 call(s) for 0.01 sec (90.9%)
# VERIFY        1 call(s) for 0.00 sec ( 0.0%)
# exact Time was 0.02 sec, cost is c=17(0) in=61 out=31 tot=92
.i 7
.o 7
.ilb Q3 Q2 Q1 M50 M100 Continuar Cancelar
.ob D3 D2 D1 Lata Troco Devolucao Rejeicao
```

```
.p 17
01001-- 1000000
100--1- 0100000
00-01-- 0100000
0011--- 0100000
010-0-- 0100000
0101--- 0100000
100---1 0110000
101---- 0001001
0-10--- 0010000
111---- 0000011
0-01--- 0010000
011--1- 1010001
100---- 1000001
110---- 0001101
011--0- 0110001
0-1---1 1110000
01----1 1110000
.e
# WRITE        Time was 0.00 sec, cost is c=17(0) in=61 out=31 tot=92
```

Equações Multi-nível

```
INORDER = Q3 Q2 Q1 M50 M100 Continuar Cancelar;
OUTORDER = D3 D2 D1 Lata Troco Devolucao Rejeicao;
D3 = Q2*!M50*M100*!Rejeicao + Q2*Cancelar*!Rejeicao + !Q3*Continuar*Rejeicao
 + Q3*!Q2*!Lata + !Q3*Q1*Cancelar;
D2 = !M50*M100*!D3*!Rejeicao + M50*!D1*!Rejeicao + !Q3*Q2*!D3 + !Q2*Continuar*
D3 + Cancelar*D3;
D1 = Q1*!M50*!Rejeicao + !Q1*M50*!Rejeicao + !Q3*Rejeicao + Cancelar*D3;
Lata = Q3*!Q2*Q1 + Troco;
Troco = Q3*Q2*!Q1;
Devolucao = Q3*Q2*Q1;
Rejeicao = Q2*Q1 + Q3;
```

Codificação de Estados de Gray

Descrição Inicial

```
.i 7
.o 7
.ilb Q3 Q2 Q1 M50 M100 Continuar Cancelar
.ob D3 D2 D1 Lata Troco Devolucao Rejeicao
.type fr

000     1---    001     0000
000     01--    011     0000
000     00--    000     0000

001     ---1    100     0000
001     1--0    011     0000
001     01-0    010     0000
001     00-0    001     0000

011     ---1    100     0000
011     1--0    010     0000
011     01-0    110     0000
011     00-0    011     0000

010     ---1    100     0001
010     --10    111     0001
010     --00    010     0001

110     ---1    100     0001
110     --10    101     0001
110     --00    110     0001

111     ----    000     1001
```

```
101     ----    000     1101

100     ----    000     0011

.e
```

PLA Optimizada

```
# espresso -Dcheck -Dexact -s -t MV_EntDir.pla
# UC Berkeley, Espresso Version #2.3, Release date 01/31/88
# READ          Time was 0.00 sec, cost is c=19(19) in=88 out=37 tot=125
# COMPL         Time was 0.02 sec, cost is c=20(20) in=93 out=103 tot=196
# PLA is MV_EntDir.pla with 7 inputs and 7 outputs
# ON-set cost is  c=19(19) in=88 out=37 tot=125
# OFF-set cost is c=20(20) in=93 out=103 tot=196
# DC-set cost is  c=0(0) in=0 out=0 tot=0
# PRIMES        Time was 0.00 sec, cost is c=35(35) in=145 out=58 tot=203
# ESSENTIALS    Time was 0.01 sec, cost is c=7(7) in=24 out=14 tot=38
# PI-TABLE      Time was 0.00 sec, cost is c=25(25) in=107 out=41 tot=148
# MINCOV        Time was 0.00 sec, cost is c=35(35) in=145 out=58 tot=203
# MV_REDUCE     Time was 0.00 sec, cost is c=15(15) in=59 out=23 tot=82
# RAISE_IN      Time was 0.00 sec, cost is c=15(0) in=56 out=23 tot=79
# MV_REDUCE     Time was 0.00 sec, cost is c=15(0) in=56 out=23 tot=79
# VERIFY        Time was 0.01 sec, cost is c=15(0) in=56 out=23 tot=79
# READ           1 call(s) for 0.00 sec ( 0.0%)
# COMPL          1 call(s) for 0.02 sec (64.5%)
# MV_REDUCE      2 call(s) for 0.00 sec ( 0.0%)
# RAISE_IN       1 call(s) for 0.00 sec ( 0.0%)
# VERIFY         1 call(s) for 0.01 sec (32.2%)
# exact Time was 0.04 sec, cost is c=15(0) in=56 out=23 tot=79
.i 7
.o 7
.ilb Q3 Q2 Q1 M50 M100 Continuar Cancelar
.ob D3 D2 D1 Lata Troco Devolucao Rejeicao
.p 15
01101-- 1000000
110---- 1000000
0-100-0 0010000
00001-- 0110000
0011--0 0110000
0001--- 0010000
0--01-0 0100000
101---- 0000100
-10--00 0100001
0-1---1 1000000
-10--10 1010001
100---- 0000011
-10---1 1000001
01----0 0100000
1-1---- 0001001
.e
# WRITE         Time was 0.00 sec, cost is c=15(0) in=56 out=23 tot=79
```

Equações Multi-nível

```
INORDER = Q3 Q2 Q1 M50 M100 Continuar Cancelar;
OUTORDER = D3 D2 D1 Lata Troco Devolucao Rejeicao;
D3 = Q2*!M50*!D1*!Rejeicao + Q1*Cancelar*!Rejeicao + Q3*Q2*!Lata + !Q3*Q2*
Cancelar + D1*Rejeicao;
D2 = !M50*M100*!D3*!Rejeicao + Q2*!Cancelar*!D1*!Lata + Q1*M50*D1 + !Q3*Q2*D1
;
D1 = Q1*!Cancelar*!Rejeicao*![14] + !Q2*M50*!Cancelar*!Rejeicao + Q2*!Q1*
Continuar*!Cancelar + !Q1*!Rejeicao*[14];
Lata = Q1*Rejeicao;
Troco = Q3*!Q2*Q1;
Devolucao = Q3*!Q2*!Q1;
Rejeicao = Q2*!Q1 + Q3;
[14] = M100 + M50;
```

Codificação de Estados Distância Total Mínima (DTM)

Descrição Inicial

```
.i 7
.o 7
.ilb Q3 Q2 Q1 M50 M100 Continuar Cancelar
.ob D3 D2 D1 Lata Troco Devolucao Rejeicao
.type fr

000     1---    001     0000
000     01--    011     0000
000     00--    000     0000

001     ---1    010     0000
001     1--0    011     0000
001     01-0    111     0000
001     00-0    001     0000

011     ---1    010     0000
011     1--0    111     0000
011     01-0    110     0000
011     00-0    011     0000

111     ---1    010     0001
111     --10    101     0001
111     --00    111     0001

110     ---1    010     0001
110     --10    100     0001
110     --00    110     0001

101     ----    000     1001

100     ----    000     1101

010     ----    000     0011

.e
```

PLA Optimizada

```
# espresso -Dcheck -Dexact -s -t MV_EntDir.pla
# UC Berkeley, Espresso Version #2.3, Release date 01/31/88
# READ         Time was 0.00 sec, cost is c=19(19) in=88 out=41 tot=129
# COMPL        Time was 0.01 sec, cost is c=20(20) in=93 out=99 tot=192
# PLA is MV_EntDir.pla with 7 inputs and 7 outputs
# ON-set cost is  c=19(19) in=88 out=41 tot=129
# OFF-set cost is c=20(20) in=93 out=99 tot=192
# DC-set cost is  c=0(0) in=0 out=0 tot=0
# PRIMES       Time was 0.01 sec, cost is c=39(39) in=151 out=69 tot=220
# ESSENTIALS   Time was 0.01 sec, cost is c=8(8) in=25 out=16 tot=41
# PI-TABLE     Time was 0.00 sec, cost is c=28(28) in=120 out=50 tot=170
# MINCOV       Time was 0.00 sec, cost is c=39(39) in=151 out=69 tot=220
# MV_REDUCE    Time was 0.00 sec, cost is c=15(15) in=56 out=23 tot=79
# RAISE_IN     Time was 0.01 sec, cost is c=15(0) in=55 out=23 tot=78
# MV_REDUCE    Time was 0.00 sec, cost is c=15(0) in=55 out=23 tot=78
# VERIFY       Time was 0.00 sec, cost is c=15(0) in=55 out=23 tot=78
# READ          1 call(s) for 0.00 sec ( 0.0%)
# COMPL         1 call(s) for 0.01 sec (47.6%)
# MV_REDUCE     2 call(s) for 0.00 sec ( 0.0%)
# RAISE_IN      1 call(s) for 0.01 sec (47.6%)
# VERIFY        1 call(s) for 0.00 sec ( 0.0%)
# exact Time was 0.04 sec, cost is c=15(0) in=55 out=23 tot=78
.i 7
.o 7
.ilb Q3 Q2 Q1 M50 M100 Continuar Cancelar
.ob D3 D2 D1 Lata Troco Devolucao Rejeicao
```

```
.p 15
00001-- 0110000
-111--0 1000000
0-101-0 1100000
0001--- 0010000
011-0-0 0110000
100---- 0000100
111---0 0010000
001---0 0010000
11---0- 0100000
0-11--0 0110000
0-1---1 0100000
010---- 0000011
11----1 0100001
11----0 1000001
10----- 0001001
.e
# WRITE        Time was 0.00 sec, cost is c=15(0) in=55 out=23 tot=78
```

Equações Multi-nível

```
INORDER = Q3 Q2 Q1 M50 M100 Continuar Cancelar;
OUTORDER = D3 D2 D1 Lata Troco Devolucao Rejeicao;
D3 = Q2*!Cancelar*[19] + Q1*!Cancelar*![18] + Q3*Q2*!Cancelar;
D2 = Q1*!D1*!Rejeicao + Q2*!D3*!Devolucao + Q1*[19] + !Continuar*D3 + ![18];
D1 = Q1*!Cancelar*!Rejeicao*[18] + Q1*D3*Rejeicao + !Q1*[19] + !Q1*![18] + !Q2*
D3;
Lata = Q3*!Q2;
Troco = !Q1*Lata;
Devolucao = !Q3*Q2*!Q1;
Rejeicao = Devolucao + Q3;
[17] = Rejeicao + !M100;
[18] = [17] + M50;
[19] = M50*!Rejeicao;
```

Codificação de Estados Distância Mínima entre os Estados de Entrada e de Saída (DMEES)

Descrição Inicial

```
.i 7
.o 7
.ilb Q3 Q2 Q1 M50 M100 Continuar Cancelar
.ob D3 D2 D1 Lata Troco Devolucao Rejeicao
.type fr

000     1---    001     0000
000     01--    011     0000
000     00--    000     0000

001     ---1    110     0000
001     1--0    011     0000
001     01-0    111     0000
001     00-0    001     0000

011     ---1    110     0000
011     1--0    111     0000
011     01-0    101     0000
011     00-0    011     0000

111     ---1    110     0001
111     --10    010     0001
111     --00    111     0001

101     ---1    110     0001
101     --10    100     0001
101     --00    101     0001
```

```
010     ----    000     1001

100     ----    000     1101

110     ----    000     0011

.e
```

PLA Optimizada

```
# espresso -Dcheck -Dexact -s -t MV_EntDir.pla
# UC Berkeley, Espresso Version #2.3, Release date 01/31/88
# READ          Time was 0.01 sec, cost is c=19(19) in=88 out=44 tot=132
# COMPL         Time was 0.00 sec, cost is c=20(20) in=93 out=96 tot=189
# PLA is MV_EntDir.pla with 7 inputs and 7 outputs
# ON-set cost is  c=19(19) in=88 out=44 tot=132
# OFF-set cost is c=20(20) in=93 out=96 tot=189
# DC-set cost is  c=0(0) in=0 out=0 tot=0
# PRIMES        Time was 0.00 sec, cost is c=45(45) in=176 out=81 tot=257
# ESSENTIALS    Time was 0.01 sec, cost is c=8(8) in=24 out=15 tot=39
# PI-TABLE      Time was 0.01 sec, cost is c=33(33) in=142 out=60 tot=202
# MINCOV        Time was 0.00 sec, cost is c=45(45) in=176 out=81 tot=257
# MV_REDUCE     Time was 0.00 sec, cost is c=15(15) in=53 out=23 tot=76
# RAISE_IN      Time was 0.00 sec, cost is c=15(0) in=50 out=23 tot=73
# MV_REDUCE     Time was 0.00 sec, cost is c=15(0) in=50 out=23 tot=73
# VERIFY        Time was 0.00 sec, cost is c=15(0) in=50 out=23 tot=73
# READ           1 call(s) for 0.01 sec (90.9%)
# COMPL          1 call(s) for 0.00 sec ( 0.0%)
# MV_REDUCE      2 call(s) for 0.00 sec ( 0.0%)
# RAISE_IN       1 call(s) for 0.00 sec ( 0.0%)
# VERIFY         1 call(s) for 0.00 sec ( 0.0%)
# exact Time was 0.03 sec, cost is c=15(0) in=50 out=23 tot=73
.i 7
.o 7
.ilb Q3 Q2 Q1 M50 M100 Continuar Cancelar
.ob D3 D2 D1 Lata Troco Devolucao Rejeicao
.p 15
000-1-- 0010000
-11-0-- 0100000
00-01-- 0100000
0001--- 0010000
1-1--00 1010000
0-101-- 1000000
0111--- 1000000
110---- 0000011
100---- 0001101
0-1---0 0010000
0-11--- 0100000
010---- 0001001
101---- 1000001
111---- 0100001
--1---1 1100000
.e
# WRITE         Time was 0.00 sec, cost is c=15(0) in=50 out=23 tot=73
```

Equações Multi-nível

```
INORDER = Q3 Q2 Q1 M50 M100 Continuar Cancelar;
OUTORDER = D3 D2 D1 Lata Troco Devolucao Rejeicao;
D3 = Q1*!M50*![15] + !Q2*Q1*Rejeicao + Q2*[17] + Q3*D1 + Q1*Cancelar;
D2 = !Q2*!M50*![15] + Q2*Q1*[15] + Q1*[17] + Cancelar*D3;
D1 = Q1*!Cancelar*!Rejeicao + Q1*!Continuar*!Cancelar + !Q1*[17] + !Q1*![15];
Lata = Q2*!Q1*!Devolucao + Troco;
Troco = Q3*!Q2*!Q1;
Devolucao = Q3*Q2*!Q1;
Rejeicao = Lata + Q3;
[15] = Rejeicao + !M100;
[17] = M50*!Rejeicao;
```

Anexo II – Diagramas Esquemáticos

Sumário

Neste anexo são mostradas as representações estruturais ao nível lógico, também designadas por diagramas esquemáticos, dos circuitos resultantes da síntese assistida por computador da unidade de controlo da máquina de venda automática e cujos resultados foram apresentados no capítulo 5. Tal como foi aí referido, foram usadas três ferramentas, o *Leonardo Spectrum*, o *Synopsys* e o *SIS*. Relativamente à primeira foram utilizadas as seguintes codificações de estado:

- Binária
- *Gray*
- *One-hot*
- *Two-hot*
- Aleatória

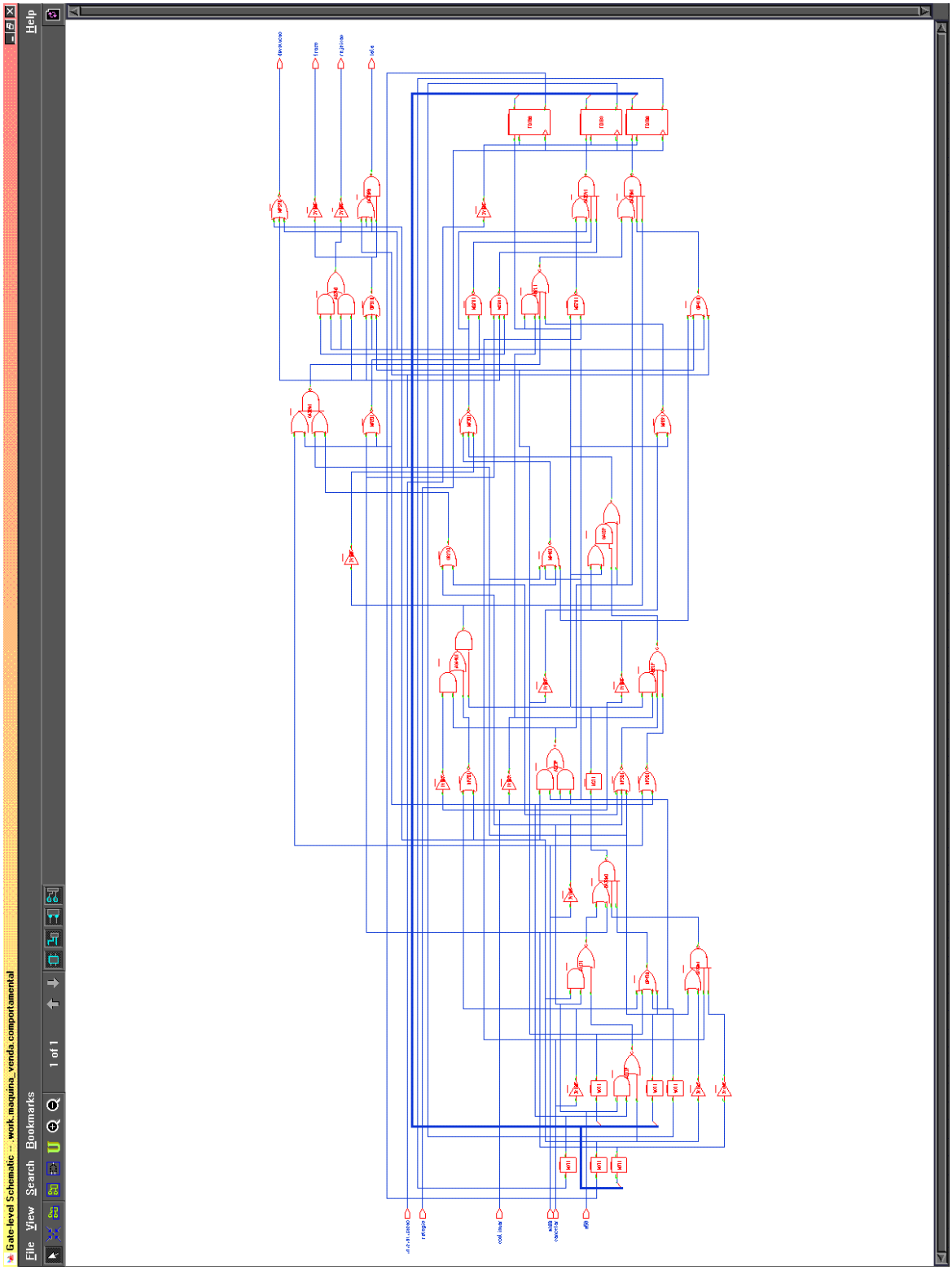
Para cada uma destas codificações a unidade de controlo foi sintetizada duas vezes, uma com optimização da área e a outra com optimização dos atrasos. Por outro lado, com o *Synopsys* foram usadas as seguintes codificações de estado:

- Binária
- *Gray*
- *One-hot*
- Manual1 (distância total mínima – DTM)
- Manual2 (distância mínima entre os estados de entrada e de saída – DMEES)
- Automática1
- Automática2 (automática com a restrição do estado inicial ser "000")

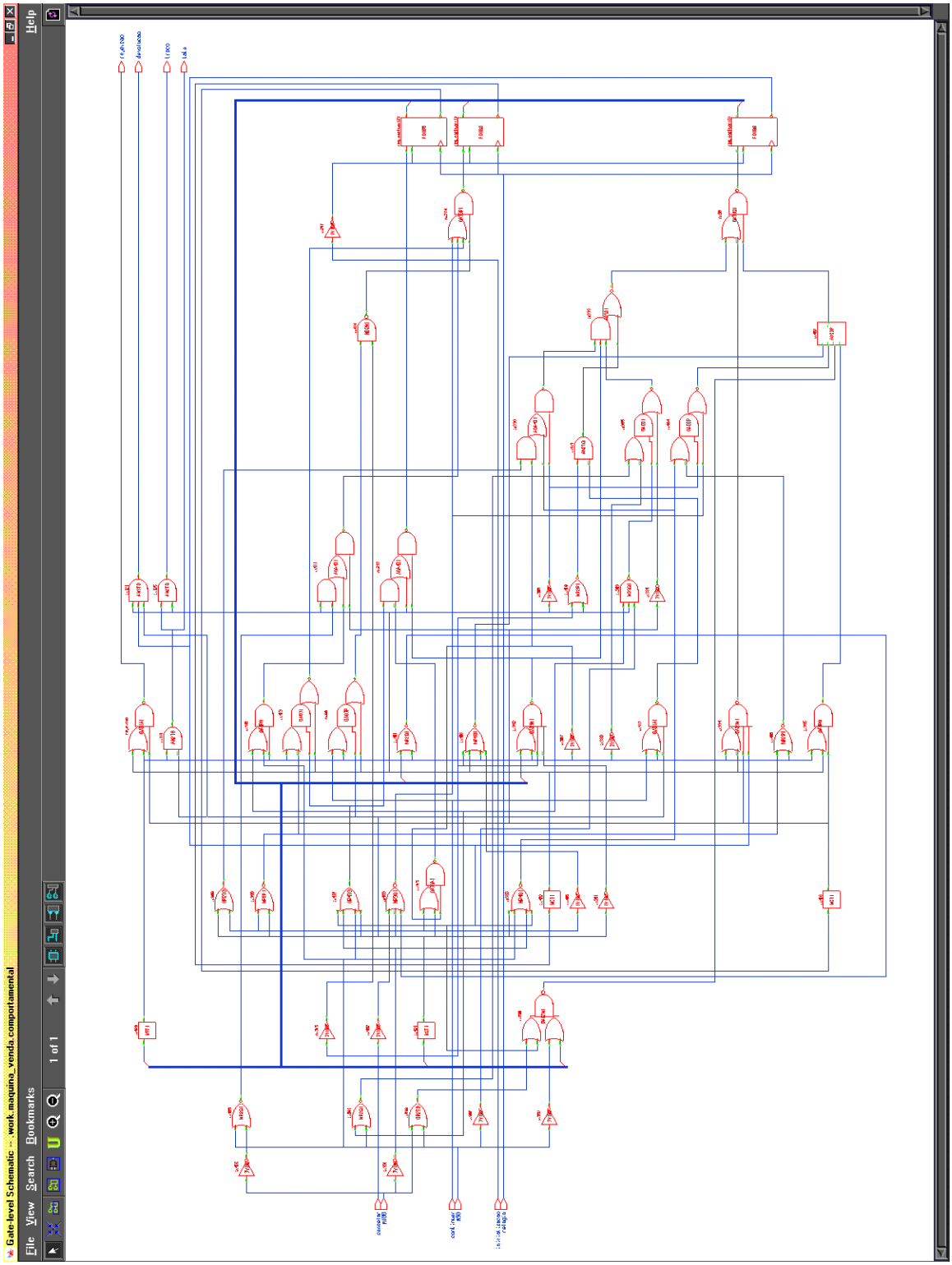
Relativamente ao *SIS* não é mostrado nenhum diagrama esquemático devido à impossibilidade em converter os resultados produzidos para este formato.

Leonardo Spectrum

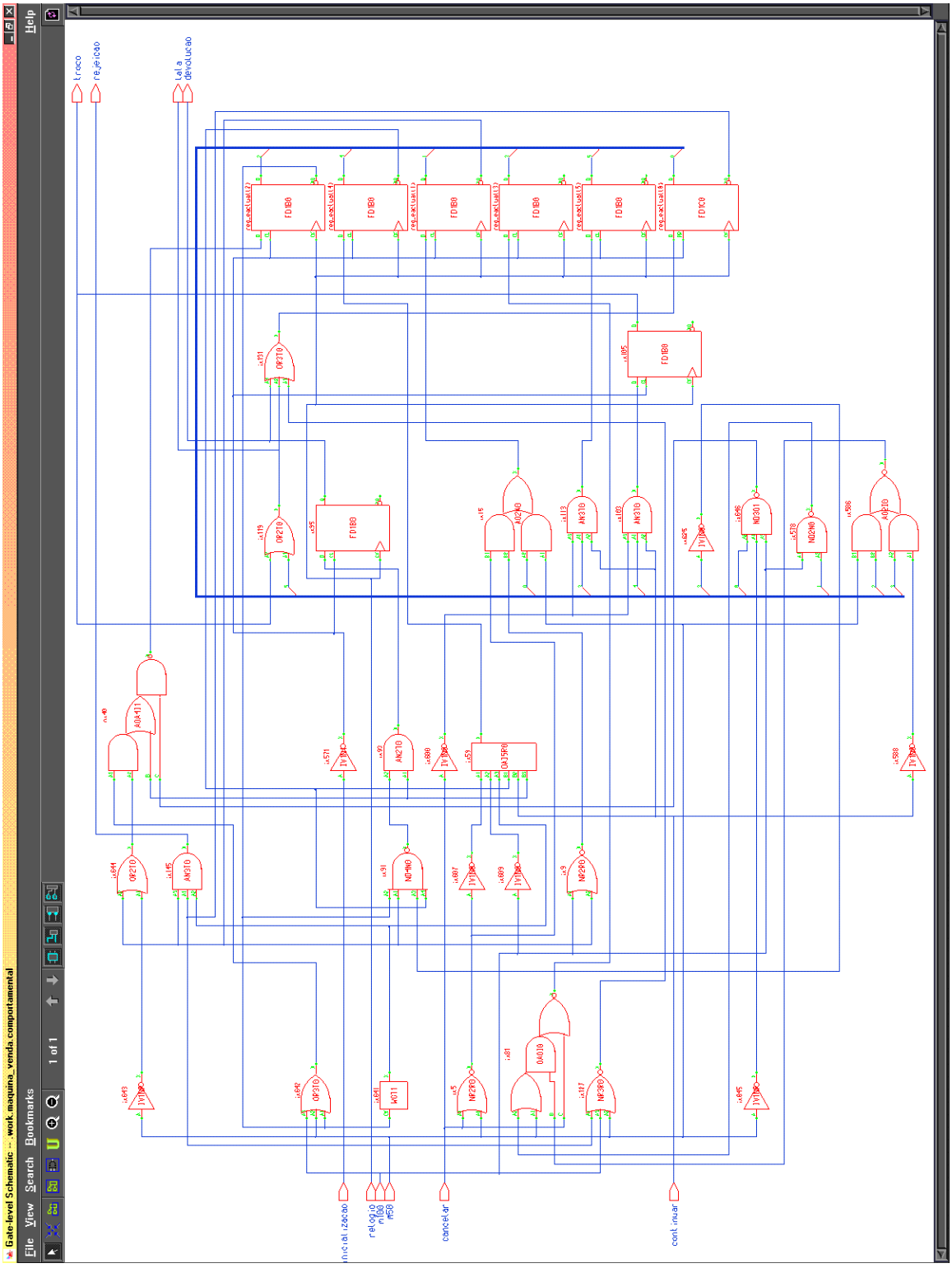
Codificação de Estados Binária com Optimização da Área



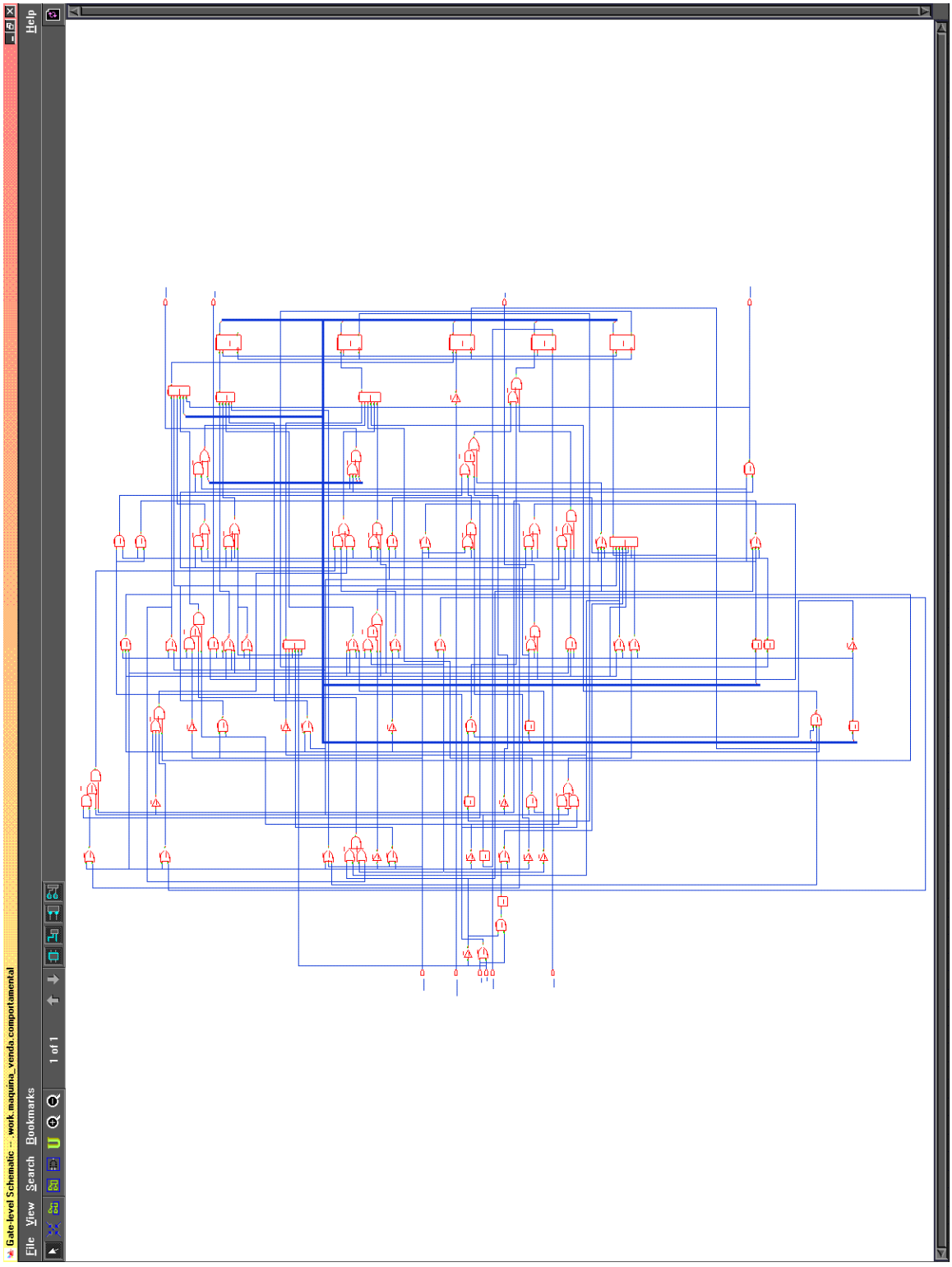
Codificação de Estados de Gray com Optimização da Área



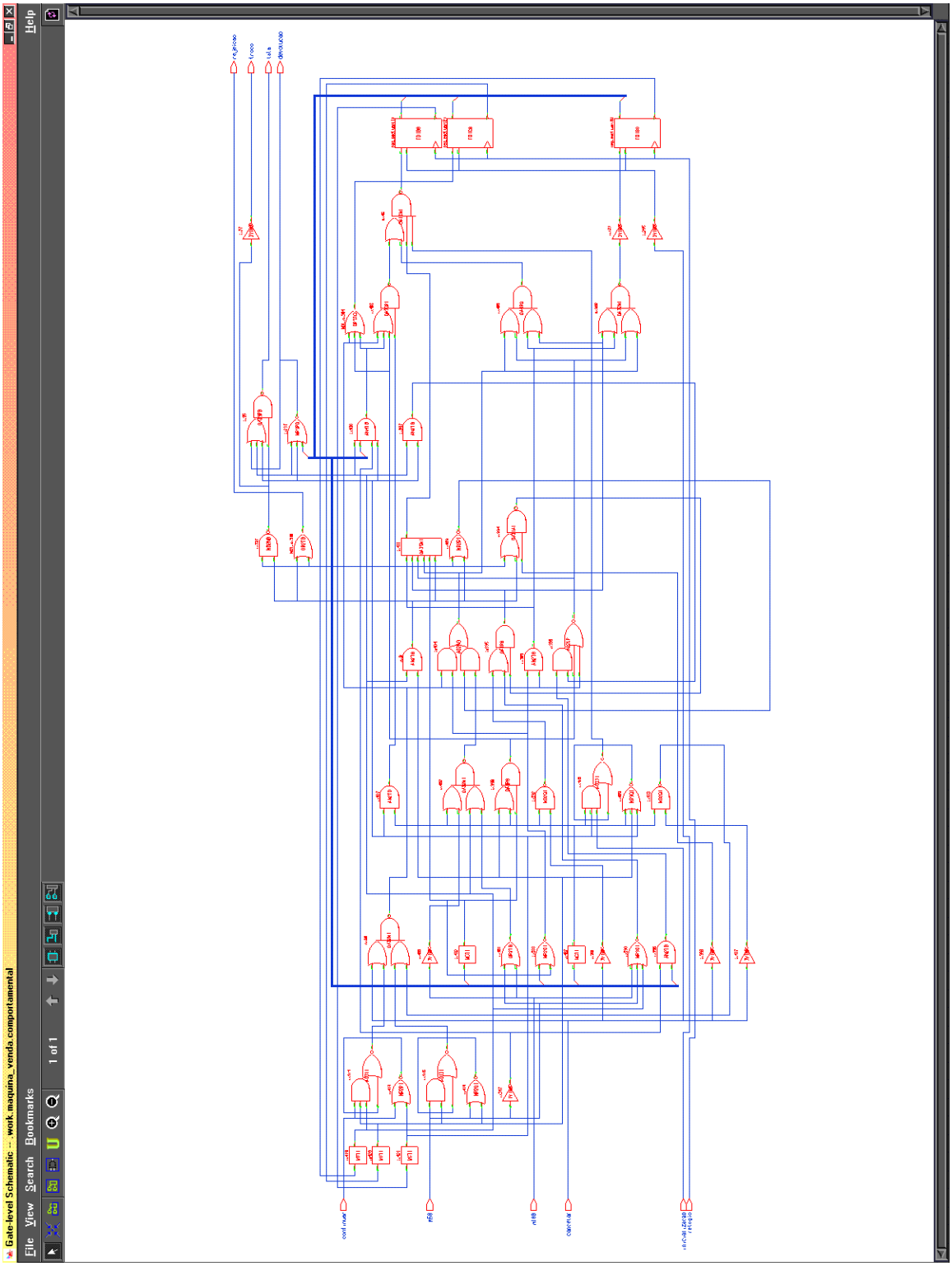
Codificação de Estados One-hot com Optimização da Área



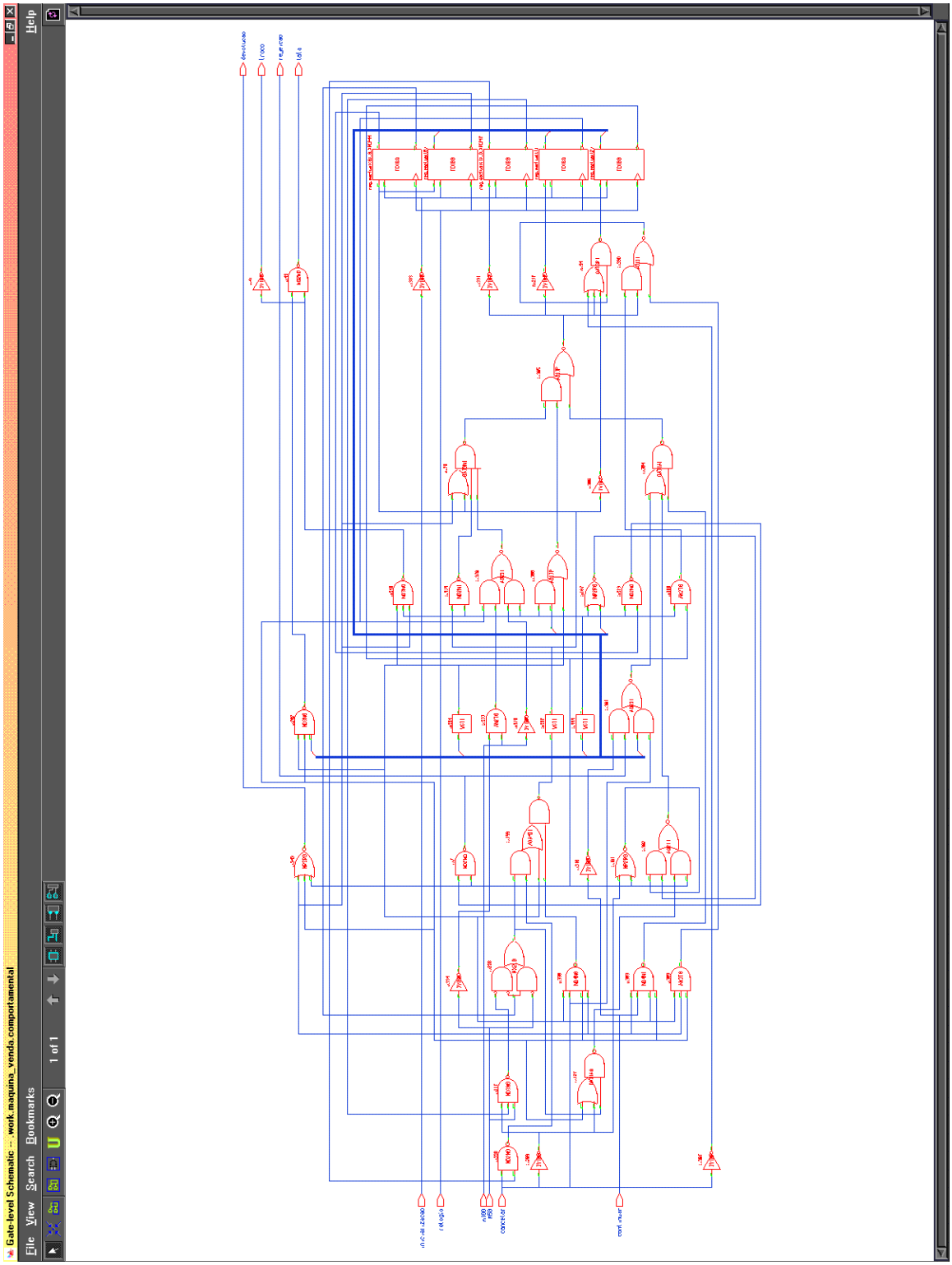
Codificação de Estados Two-hot com Optimização da Área



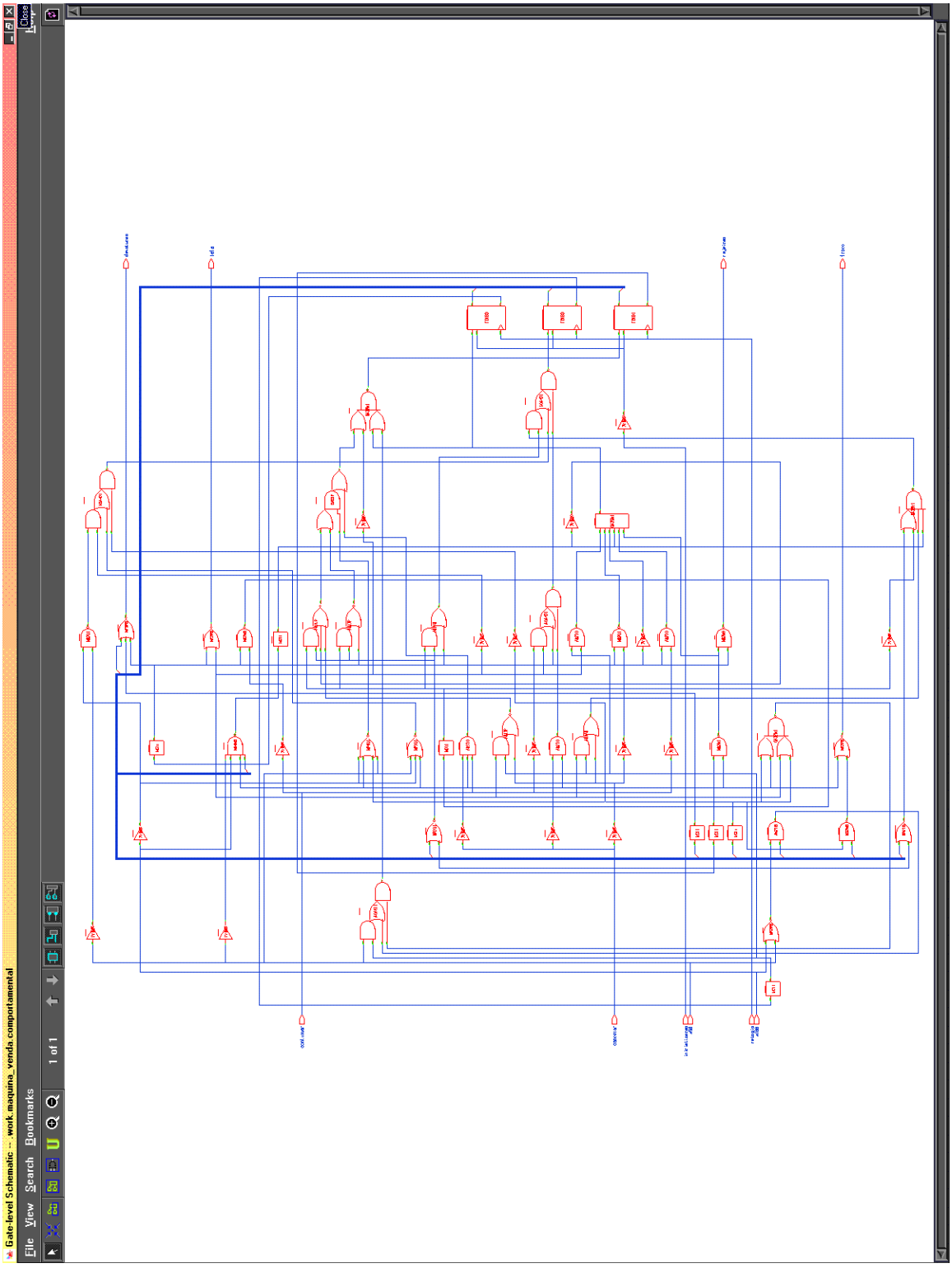
Codificação de Estados Aleatória com Optimização da Área



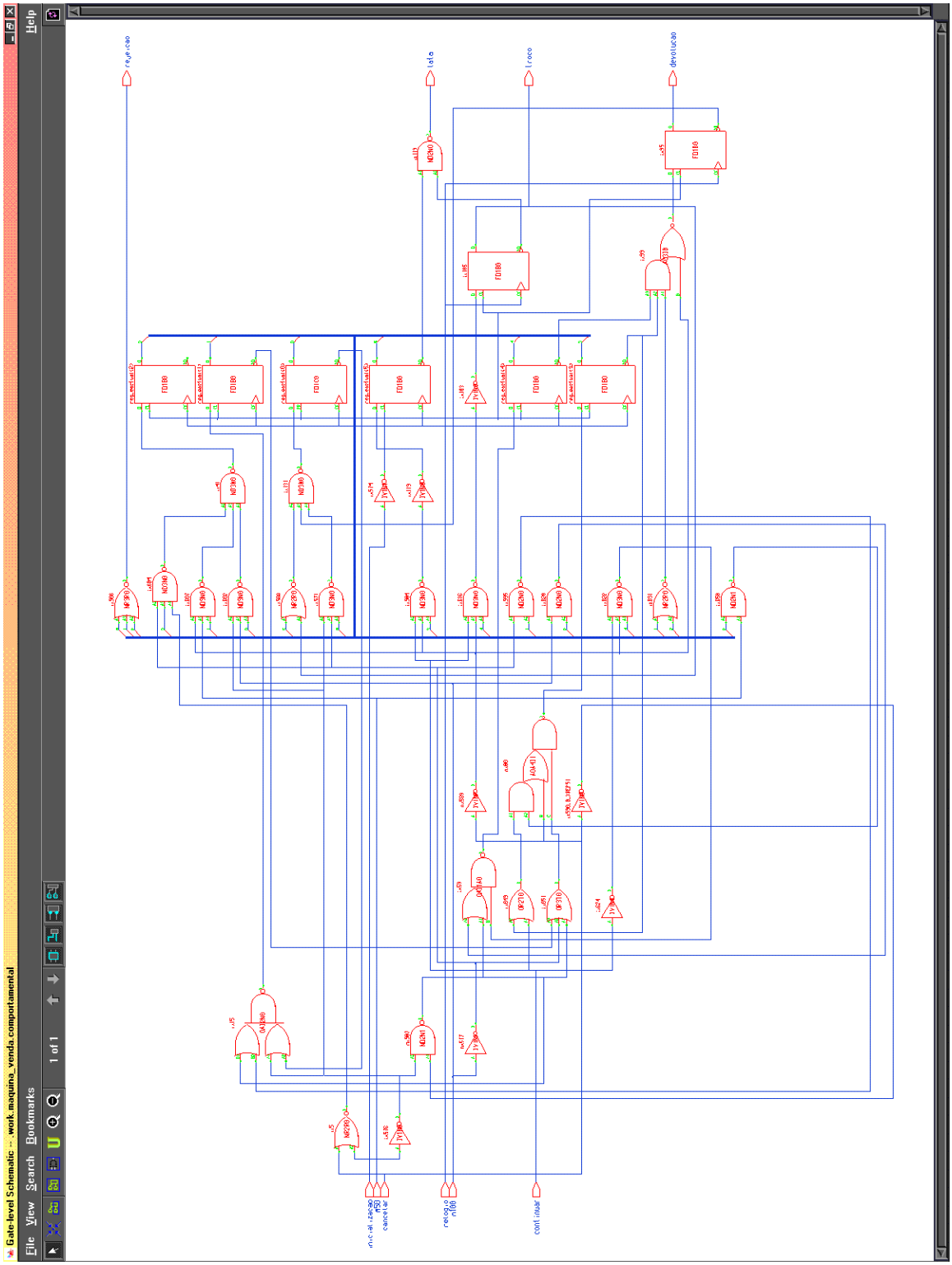
Codificação de Estados Binária com Optimização dos Atrasos



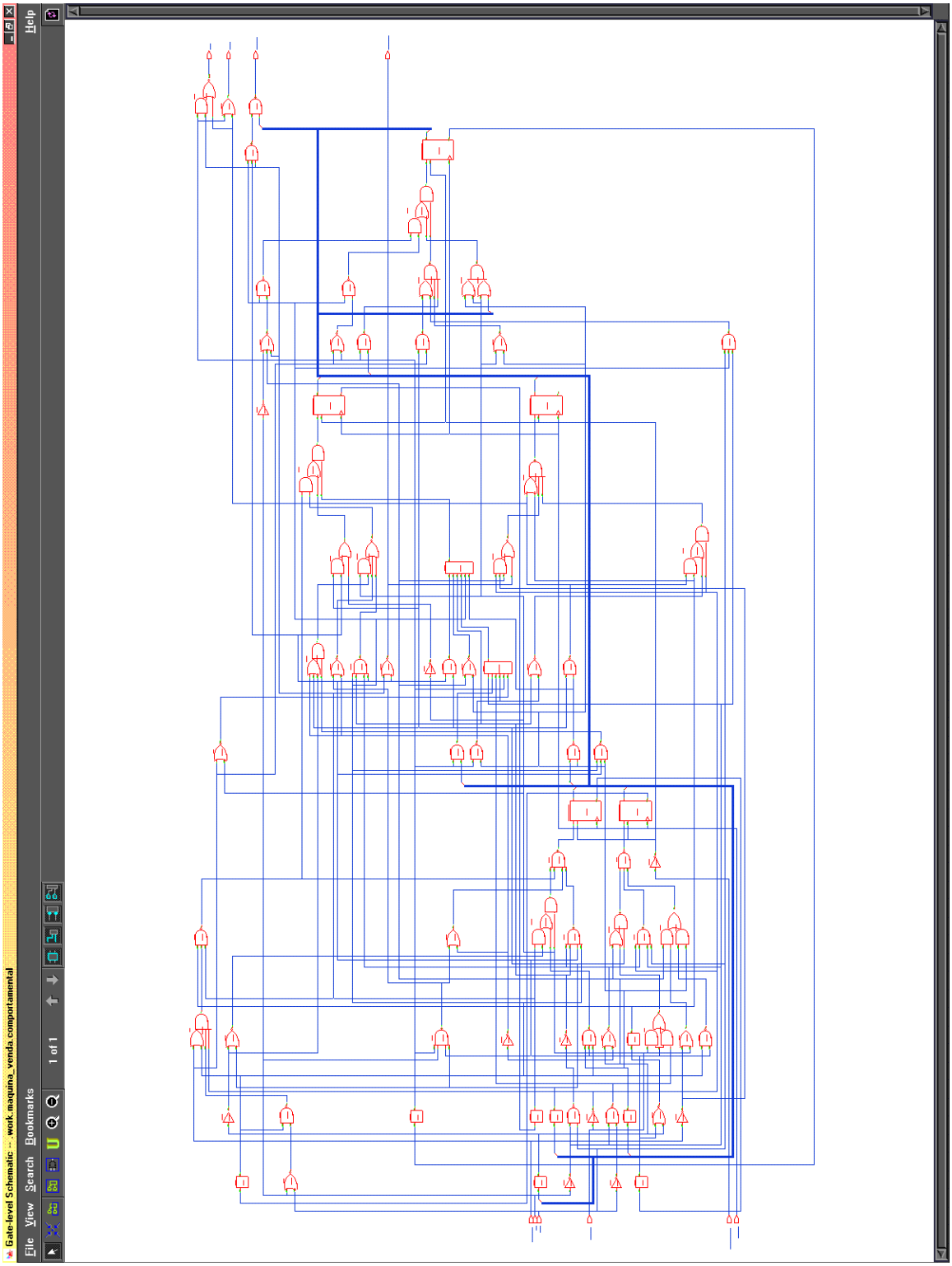
Codificação de Estados de Gray com Optimização dos Atrasos



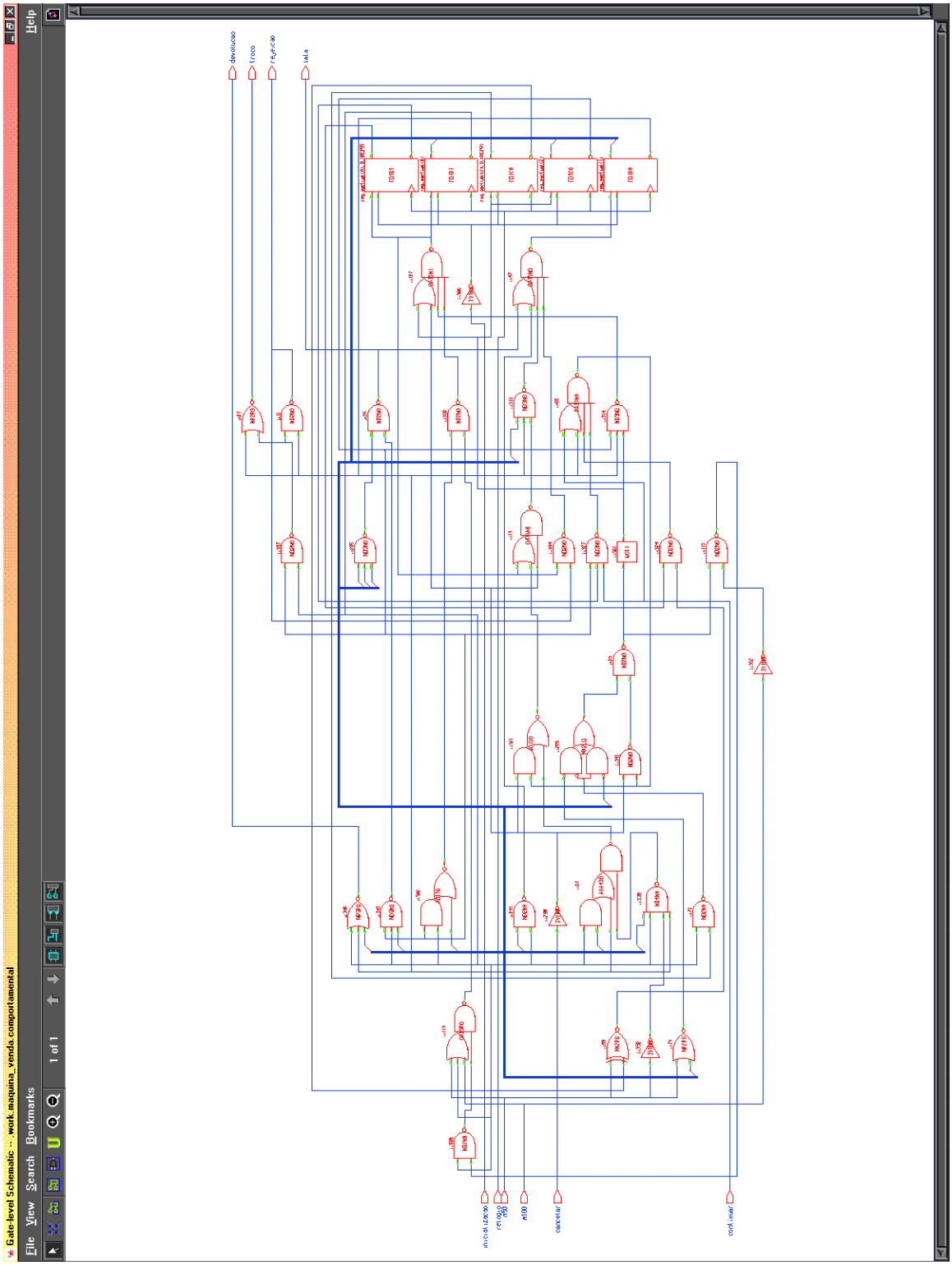
Codificação de Estados One-hot com Optimização dos Atrasos



Codificação de Estados Two-hot com Optimização dos Atrasos

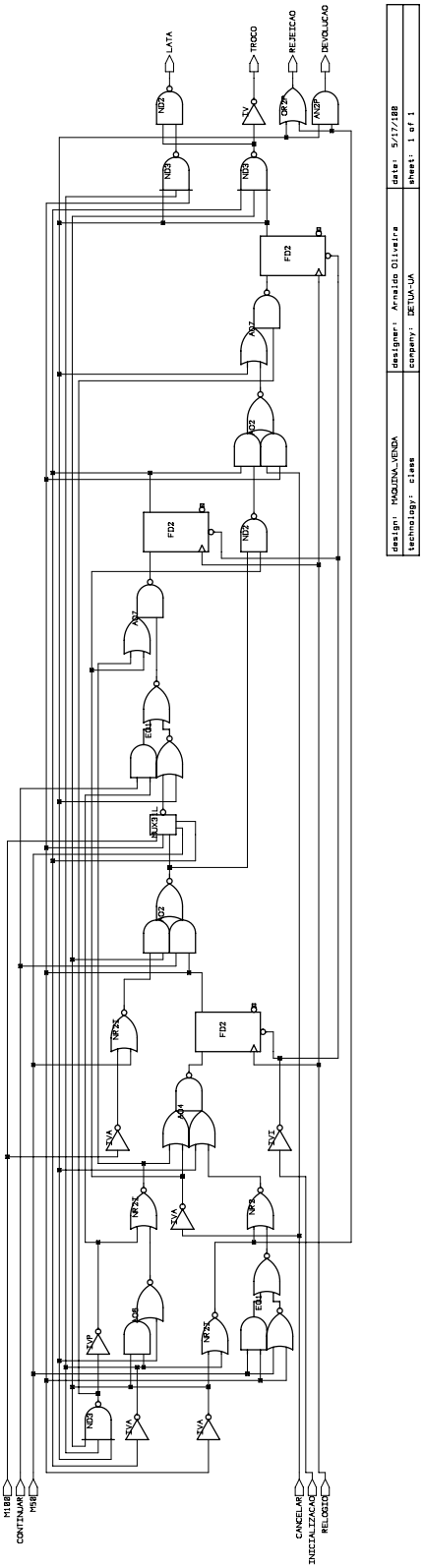


Codificação de Estados Aleatória com Optimização dos Atrasos

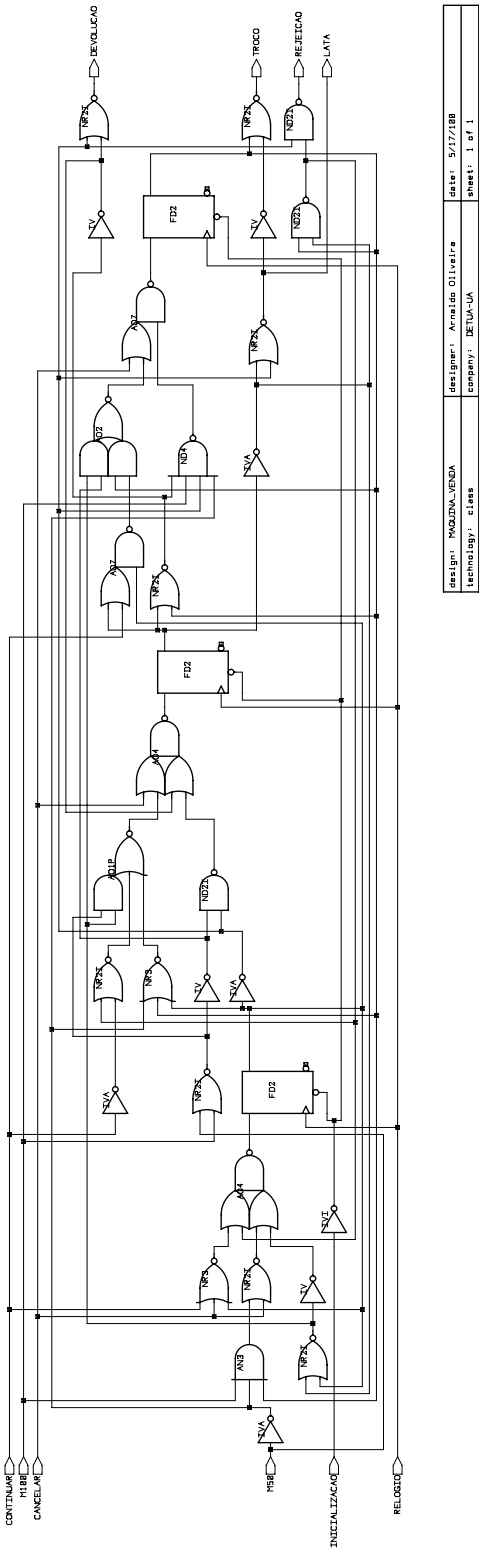


Synopsys

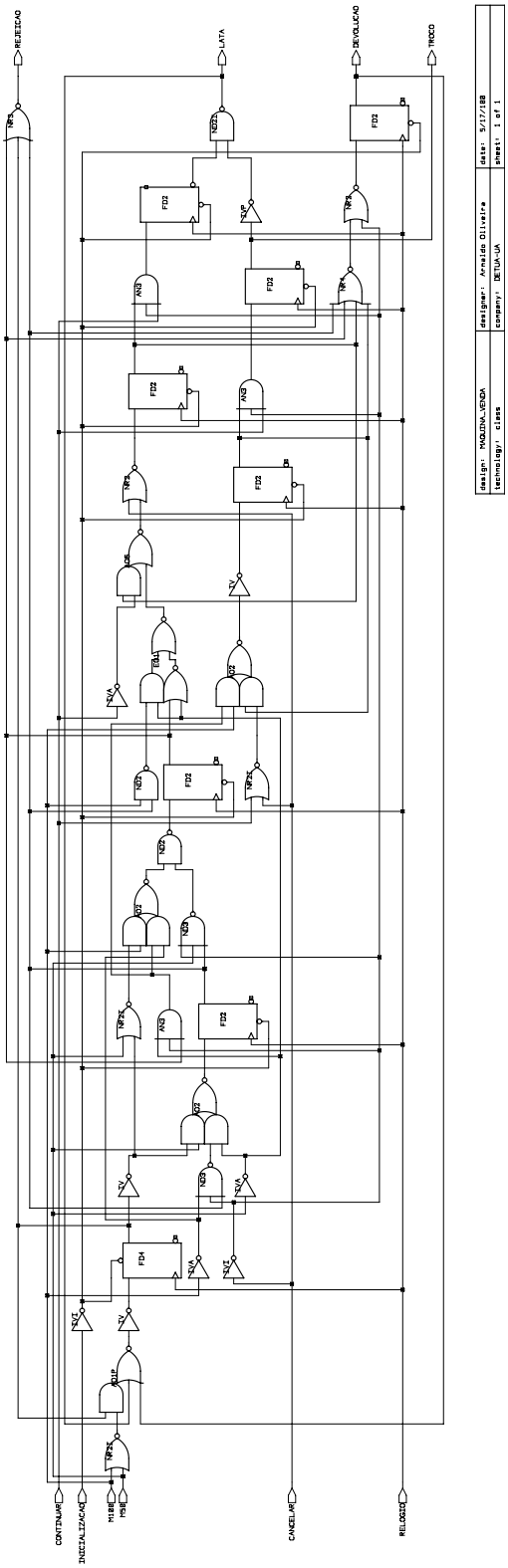
Codificação de Estados Binária



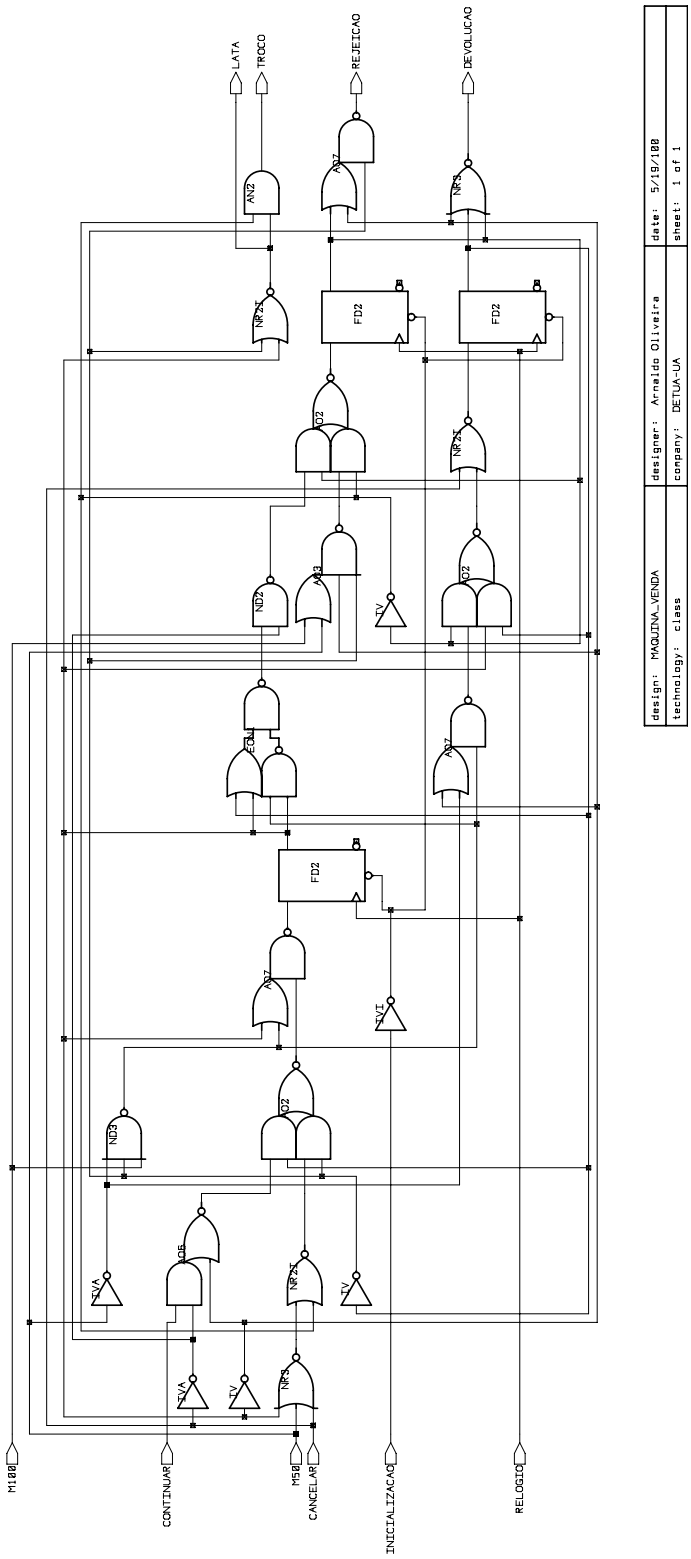
Codificação de Estados de Gray



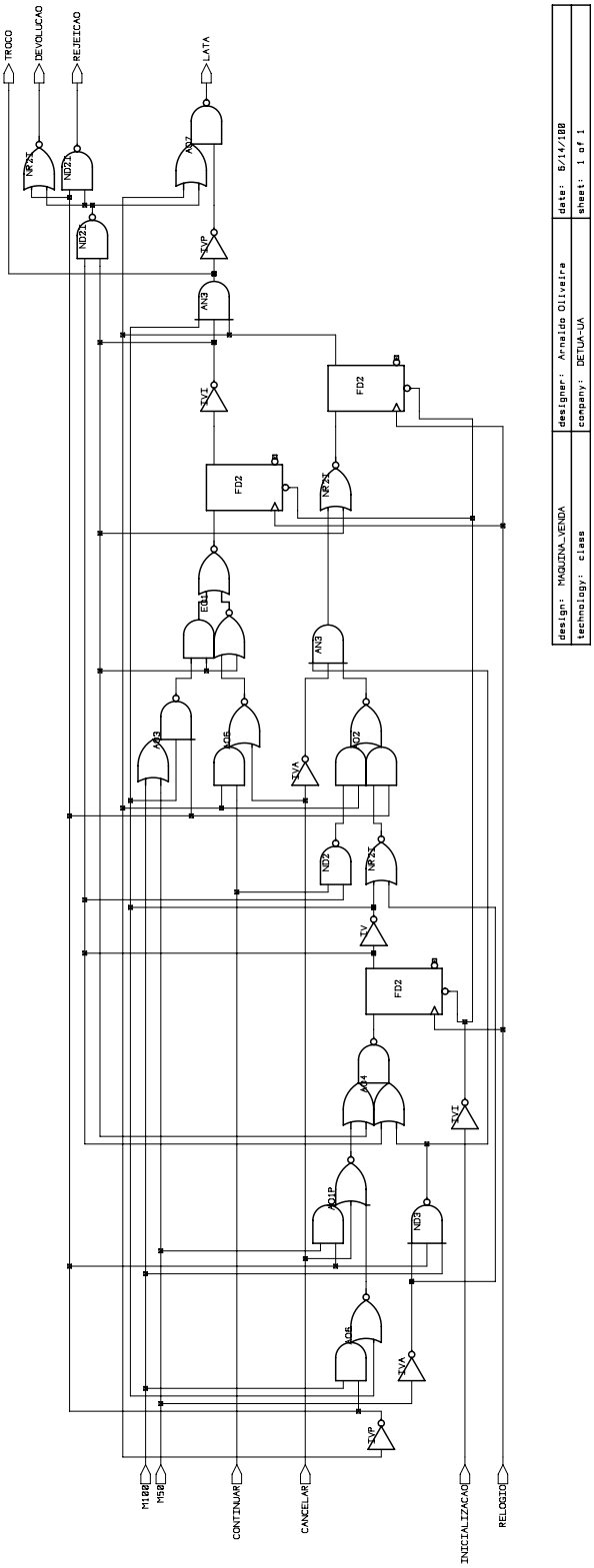
Codificação de Estados One-hot



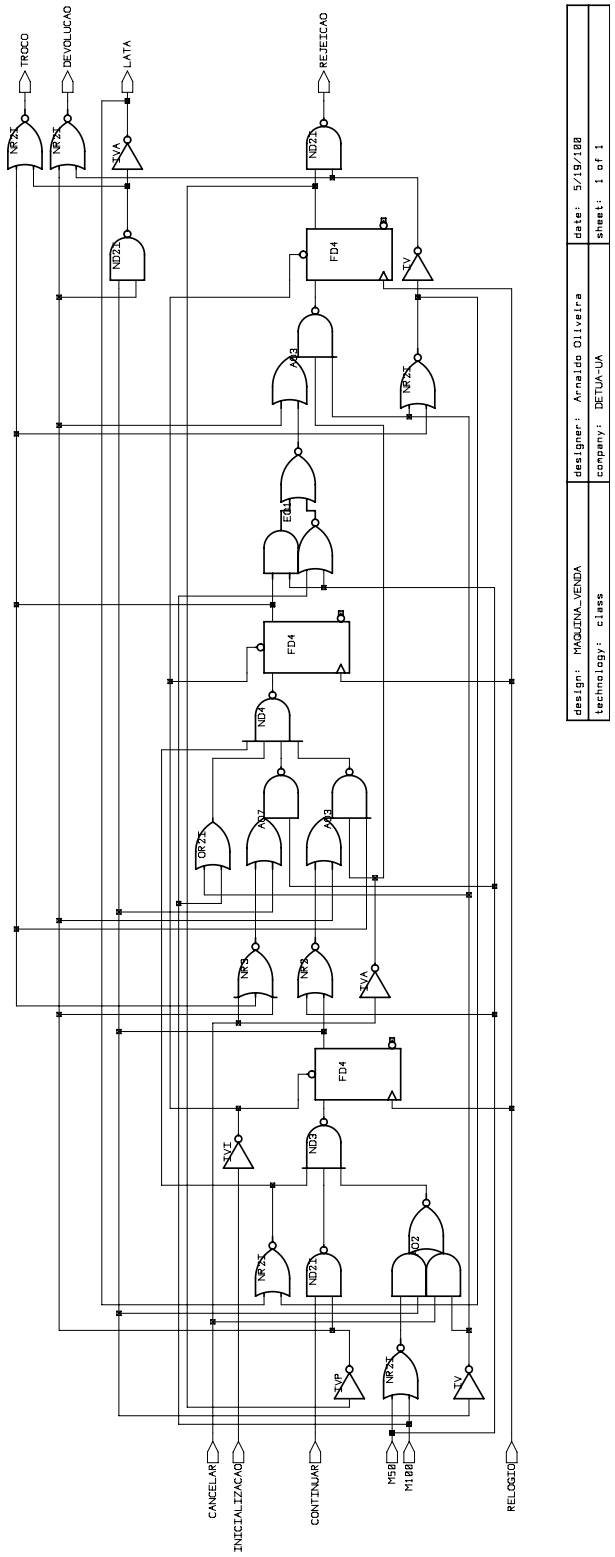
Codificação de Estados Manual1 (distância total mínima – DTM)



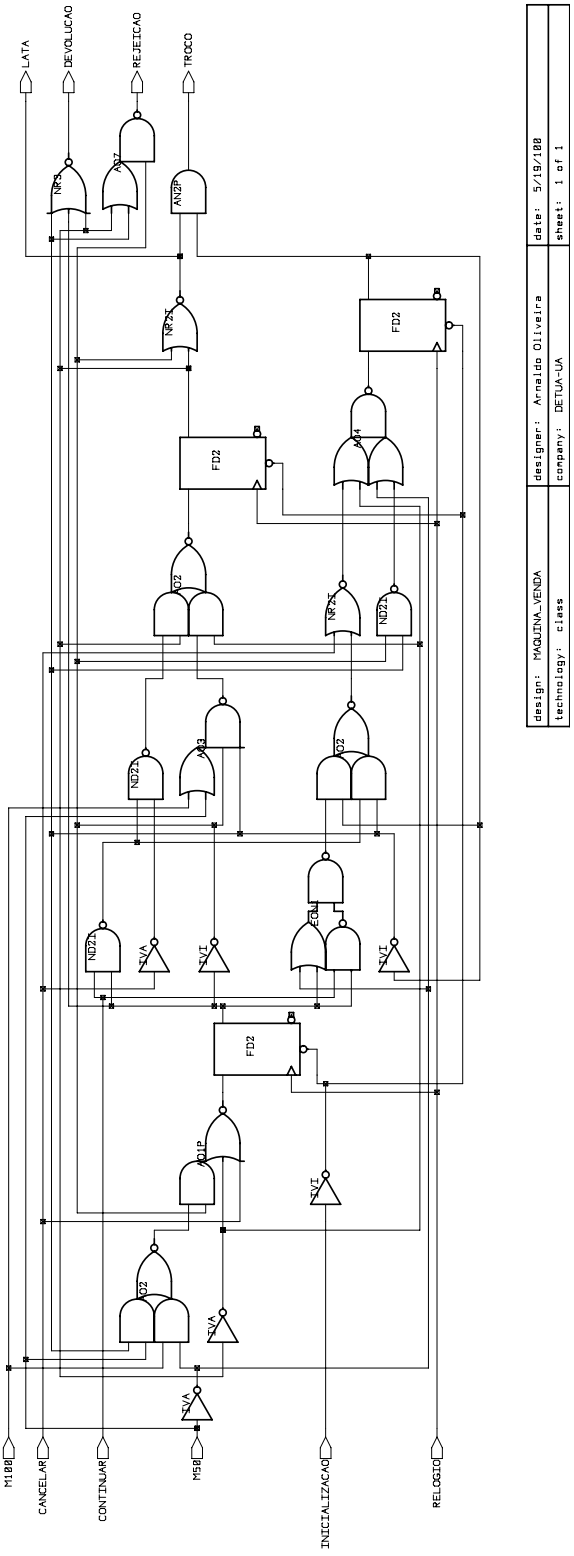
Codificação de Estados Manual2 (distância mínima entre os estados de entrada e de saída – DMEES)



Codificação de Estados Automática1 (completamente automática)



Codificação de Estados Automática2 (automática com a restrição do estado inicial ser "000")



Anexo III – Listagem da Biblioteca para Mapeamento na Tecnologia XC6200 da Xilinx

Sumário

Neste anexo é apresentado o conteúdo da biblioteca "xc6200.genlib" que foi construída para ser utilizada juntamente com o SIS na síntese de circuitos digitais cuja tecnologia de implementação seja a arquitectura de FPGAs XC6200 da Xilinx. Desta forma é possível realizar todos os passos da síntese lógica e sequencial de um circuito, tais como a optimização lógica, a minimização de estados, a codificação de estados e o mapeamento na tecnologia. Esta biblioteca está escrita no formato *genlib* [Sis92].

Como o circuito gerado é constituído apenas por componentes conhecidos das ferramentas de implementação da arquitectura XC6200, a lista de ligações produzida pode ser lida e processada por estas a fim de serem realizadas as tarefas de implantação e interligação do circuito num dispositivo específico desta família.

Os componentes existentes nesta biblioteca correspondem às primitivas implementáveis na parte combinatória e na parte sequencial de uma célula desta arquitectura de FPGAs e que são:

- Qualquer tipo de porta lógica de duas entradas;
- Qualquer tipo de multiplexador de 2 para 1;
- Flip-flop tipo D com entrada de inicialização a "0".

A separação entre as componentes combinatória e sequencial de uma célula é possível porque a ferramenta de implementação XACT6000 também realiza parte do mapeamento na tecnologia através do agrupamento, sempre que possível, de vários componentes da lista de ligações de entrada numa única célula da FPGA.

XC6200.genlib

```
GATE GND        1       O = CONST0;

GATE VCC        1       O = CONST1;

GATE BUF        1       O = I;
PIN I NONINV    1.0     999.0   0.1 0.1 0.1 0.1

GATE INV        1       O = !I;
PIN I INV       1.0     999.0   0.1 0.1 0.1 0.1

GATE AND2       1       O = (I0 * I1);
PIN I0 NONINV   1.0     999.0   0.1 0.1 0.1 0.1
PIN I1 NONINV   1.0     999.0   0.1 0.1 0.1 0.1

GATE AND2B1     1       O = (!I0 * I1);
PIN I0 INV      1.0     999.0   0.1 0.1 0.1 0.1
PIN I1 NONINV   1.0     999.0   0.1 0.1 0.1 0.1

GATE AND2B2     1       O = (!I0 * !I1);
PIN I0 INV      1.0     999.0   0.1 0.1 0.1 0.1
PIN I1 INV      1.0     999.0   0.1 0.1 0.1 0.1

GATE NAND2      1       O = !(I0 * I1);
PIN I0 INV      1.0     999.0   0.1 0.1 0.1 0.1
PIN I1 INV      1.0     999.0   0.1 0.1 0.1 0.1

GATE NAND2B1    1       O = !(!I0 * I1);
PIN I0 NONINV   1.0     999.0   0.1 0.1 0.1 0.1
PIN I1 INV      1.0     999.0   0.1 0.1 0.1 0.1

GATE NAND2B2    1       O = !(!I0 * !I1);
PIN I0 NONINV   1.0     999.0   0.1 0.1 0.1 0.1
PIN I1 NONINV   1.0     999.0   0.1 0.1 0.1 0.1

GATE OR2        1       O = (I0 + I1);
PIN I0 NONINV   1.0     999.0   0.1 0.1 0.1 0.1
PIN I1 NONINV   1.0     999.0   0.1 0.1 0.1 0.1

GATE OR2B1      1       O = (!I0 + I1);
PIN I0 INV      1.0     999.0   0.1 0.1 0.1 0.1
PIN I1 NONINV   1.0     999.0   0.1 0.1 0.1 0.1

GATE OR2B2      1       O = (!I0 + !I1);
PIN I0 INV      1.0     999.0   0.1 0.1 0.1 0.1
PIN I1 INV      1.0     999.0   0.1 0.1 0.1 0.1

GATE NOR2       1       O = !(I0 + I1);
PIN I0 INV      1.0     999.0   0.1 0.1 0.1 0.1
PIN I1 INV      1.0     999.0   0.1 0.1 0.1 0.1

GATE NOR2B1     1       O = !(!I0 + I1);
PIN I0 NONINV   1.0     999.0   0.1 0.1 0.1 0.1
PIN I1 INV      1.0     999.0   0.1 0.1 0.1 0.1

GATE NOR2B2     1       O = !(!I0 + !I1);
PIN I0 NONINV   1.0     999.0   0.1 0.1 0.1 0.1
PIN I1 NONINV   1.0     999.0   0.1 0.1 0.1 0.1

GATE XOR2       1       O = ((!I0 * I1) + (I0 * !I1));
PIN I0 UNKNOWN          1.0     999.0   0.1 0.1 0.1 0.1
PIN I1 UNKNOWN          1.0     999.0   0.1 0.1 0.1 0.1

GATE XNOR2      1       O = ((!I0 * !I1) + (I0 * I1));
PIN I0 UNKNOWN          1.0     999.0   0.1 0.1 0.1 0.1
PIN I1 UNKNOWN          1.0     999.0   0.1 0.1 0.1 0.1

GATE M2_1       1       O = ((D0 * S0) + (D1 * !S0));
PIN D0 NONINV   1.0     999.0   0.1 0.1 0.1 0.1
PIN D1 NONINV   1.0     999.0   0.1 0.1 0.1 0.1
```

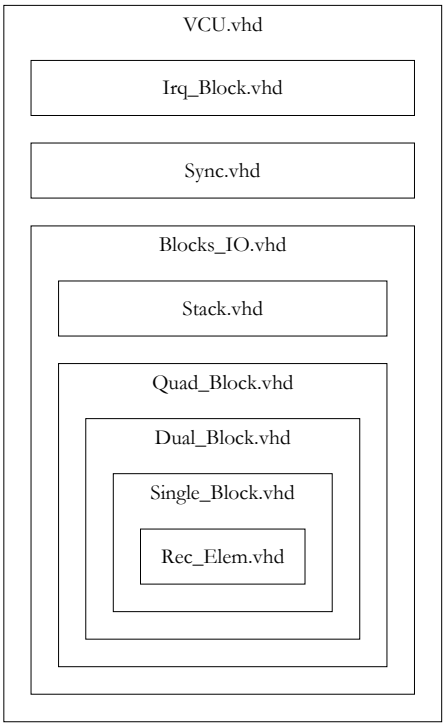
```
PIN S0 UNKNOWN          1.0     999.0   0.1 0.1 0.1 0.1

GATE M2_1B1A    1       O = ((!D0 * S0) + (D1 * !S0));
PIN D0 INV      1.0     999.0   0.1 0.1 0.1 0.1
PIN D1 NONINV   1.0     999.0   0.1 0.1 0.1 0.1
PIN S0 UNKNOWN          1.0     999.0   0.1 0.1 0.1 0.1

GATE M2_1B1B    1       O = ((D0 * S0) + (!D1 * !S0));
PIN D0 NONINV   1.0     999.0   0.1 0.1 0.1 0.1
PIN D1 INV      1.0     999.0   0.1 0.1 0.1 0.1
PIN S0 UNKNOWN          1.0     999.0   0.1 0.1 0.1 0.1

GATE M2_1B2     1       O = !((D0 * S0) + (D1 * !S0));
PIN D0 INV      1.0     999.0   0.1 0.1 0.1 0.1
PIN D1 INV      1.0     999.0   0.1 0.1 0.1 0.1
PIN S0 UNKNOWN          1.0     999.0   0.1 0.1 0.1 0.1

LATCH FDC       1       Q = D;
PIN D NONINV    1.0     999.0   0.1 0.1 0.1 0.1
SEQ Q ANY RISING_EDGE
CONTROL C       1.0     999.0   0.1 0.1 0.1 0.1
```

Anexo IV – Listagens dos Módulos VHDL

Sumário

Neste anexo são apresentadas as listagens VHDL completas de todos os módulos que constituem a estrutura parametrizável predefinida e que pode ser utilizada para implementar unidades de controlo virtuais na família de FPGAs reconfiguráveis dinâmica e parcialmente XC6200 da Xilinx.

A descrição funcional desta estrutura foi realizada no capítulo 6 desta dissertação. As relações de inclusão entre os vários módulos são apresentadas na figura seguinte.



Rec_Elem.vhd

```
--------------------------------------------------------------------------------
--
-- Reconfigurable element
--
--------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library PRIMS;
use PRIMS.XC6000_COMPONENTS.ALL;

library XC6200LIB;
use XC6200LIB.GS_PACK.ALL;
use XC6200LIB.GATES_PACK.ALL;


entity REC_ELEM is

        generic(N_IN_OUT       : integer := 16;
                N_CALLS        : integer := 4);

        port(GR_CLK            : in    STD_LOGIC;
             GR_CLR            : in    STD_LOGIC;
             START             : in    STD_LOGIC;
             FINISH            : out   STD_LOGIC;
             OH_CURR_ST        : out   STD_LOGIC_VECTOR((N_CALLS-1) downto 0);
             OH_PREV_ST        : in    STD_LOGIC_VECTOR((N_CALLS-1) downto 0);
             INPUTS            : in    STD_LOGIC_VECTOR((N_IN_OUT-1) downto 0);
             OUTPUTS           : out   STD_LOGIC_VECTOR((N_IN_OUT-1) downto 0));

-- Entity attributes -----------------------------------------------------------

        attribute OVERLAP      of REC_ELEM : entity   is "NO";
        attribute RTMAX        of REC_ELEM : entity   is "X4Y4";

end REC_ELEM;


architecture STRUCT of REC_ELEM is

-- Components definition -------------------------------------------------------

        component ORN is
                generic(N      : integer := 3);

                port(I         : in    STD_LOGIC_VECTOR((N-1) downto 0);
                     O         : out   STD_LOGIC);
        end component;

-- Architecture attributes -----------------------------------------------------

        attribute RLOC  of ST_BUF      : label  is "X0Y1";
        attribute RLOC  of FIN_BUF     : label  is "X,(2*N_CALLS)+3,Y0";

        attribute RLOC  of C_GND       : label  is "X,(2*N_CALLS)+2,Y0";
        attribute RLOC  of O_GND       : label  is "X,(2*N_CALLS)+3,Y3";

-- Entity internal signals -----------------------------------------------------

        signal ST_BUF_O, FIN_BUF_I     : STD_LOGIC;

        signal C_BUF_I, R_BUF_O        : STD_LOGIC_VECTOR((N_CALLS-1) downto 0);
        signal IN_BUF_O, OUT_BUF_I     : STD_LOGIC_VECTOR((N_IN_OUT-1) downto 0);

        signal ON_I                    : STD_LOGIC_VECTOR(4 downto 0);
        signal ON_O                    : STD_LOGIC_VECTOR(4 downto 0);

        signal CN_COND                 : STD_LOGIC_VECTOR(1 downto 0);
```

```
        signal CN_I                  : STD_LOGIC_VECTOR(1 downto 0);
        signal CN_O_0                : STD_LOGIC_VECTOR(1 downto 0);
        signal CN_O_1                : STD_LOGIC_VECTOR(1 downto 0);

        signal C_GND_O, O_GND_O      : STD_LOGIC;

-- Implementation ------------------------------------------------------------

begin

-- Components for entry and exit points --------------------------------------

        ST_BUF          : BUF   port map(I     => START,
                                         O     => ST_BUF_O);

        FIN_BUF         : BUF   port map(I     => FIN_BUF_I,
                                         O     => FINISH);

        C_BUF           : for J in (N_CALLS-1) downto 0 generate
                attribute RLOC of B : label  is "X,(2*J)+2,Y0";
        begin
                B       : BUF   port map(I     => C_BUF_I(J),
                                         O     => OH_CURR_ST(J));
        end generate;

        R_BUF           : for J in (N_CALLS-1) downto 0 generate
                attribute RLOC of B    : label  is "X,(2*J)+3,Y0";
        begin
                B       : BUF   port map(I     => OH_PREV_ST(J),
                                         O     => R_BUF_O(J));
        end generate;

        IN_BUF          : for J in (N_IN_OUT-1) downto 0 generate
                attribute RLOC of B    : label  is "X0Y,J+4,";
        begin
                B       : BUF   port map(I     => INPUTS(J),
                                         O     => IN_BUF_O(J));
        end generate;

        OUT_BUF         : for J in (N_IN_OUT-1) downto 0 generate
                attribute RLOC of B    : label  is "X,(2*N_CALLS)+3,Y,J+4,";
        begin
                B       : BUF   port map(I     => OUT_BUF_I(J),
                                         O     => OUTPUTS(J));
        end generate;

-- Nodes instantiation -------------------------------------------------

        ONG             : for J in 4 downto 0 generate
        begin
                I       : OP_NODE      port map(I      => ON_I(J),
                                                CLK    => GR_CLK,
                                                RESET  => GR_CLR,
                                                O      => ON_O(J));
        end generate;

        CNG             : for J in 1 downto 0 generate
        begin
                I       : COND_NODE    port map(I     => CN_I(J),
                                                COND  => CN_COND(J),
                                                O_0   => CN_O_0(J),
                                                O_1   => CN_O_1(J));
        end generate;

-- Conditions ----------------------------------------------------------

        CN_COND(0)      <= IN_BUF_O(1);
        CN_COND(1)      <= IN_BUF_O(2);

-- Connections between nodes -------------------------------------------

        ON_I(0)         <= ST_BUF_O;
        ON_I(1)         <= CN_O_0(0);
        ON_I(2)         <= CN_O_1(0);
```

```
        ON_I(3)         <= CN_O_1(1);
        ON_I(4)         <= R_BUF_O(0);

        CN_I(0)         <= ON_O(0);
        CN_I(1)         <= ON_O(2);

-- Control outputs assignment --------------------------------------

        C_BUF_I(0)      <= ON_O(3);

        FIN_OR          : ORN   generic map(N  => 3)
                                port map(I(0)  => ON_O(1),
                                         I(1)  => CN_O_0(1),
                                         I(2)  => ON_O(4),
                                         O     => FIN_BUF_I);

-- Standard outputs assignment -------------------------------------

        OUT_1_OR        : ORN   generic map(N  => 2)
                                port map(I(0)  => ON_O(0),
                                         I(1)  => ON_O(2),
                                         O     => OUT_BUF_I(1));

        OUT_BUF_I(2)    <= ON_O(1);

        OUT_3_OR        : ORN   generic map(N  => 2)
                                port map(I(0)  => ON_O(2),
                                         I(1)  => ON_O(4),
                                         O     => OUT_BUF_I(3));

-- Connect to ground unused output pins ----------------------------

        C_GND           : GND   port map(GROUND => C_GND_O);

        CALL_GND        : for J in (N_CALLS-1) downto 1 generate
        begin
                C_BUF_I(J)      <= C_GND_O;
        end generate;

        O_GND           : GND   port map(GROUND => O_GND_O);

        OUT_BUF_I(0)    <= O_GND_O;

        OUT_GND         : for J in (N_IN_OUT-1) downto 4 generate
        begin
                OUT_BUF_I(J)    <= O_GND_O;
        end generate;

end STRUCT;
```

Single_Block.vhd

```
--------------------------------------------------------------------------------
--
-- Single implementation block
--
--------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library PRIMS;
use PRIMS.XC6000_COMPONENTS.ALL;

library XC6200LIB;
use XC6200LIB.ARRAY_PACK.N_BUF;
use XC6200LIB.REG_PACK.PREG4N;
use XC6200LIB.REG_PACK.PREG_EQ_COMPN;

library WORK;
use WORK.REC_ELEM;


entity SINGLE_BLOCK is

        generic(N_IN_OUT       : integer := 16;
                N_G_LINES      : integer := 6;
                N_ST_LINES     : integer := 2;
                N_SP_CELLS     : integer := 2);

        port(START     : in    STD_LOGIC;
             FINISH    : out   STD_LOGIC;
             CALL      : out   STD_LOGIC;
             RET       : in    STD_LOGIC;
             CURR_ST   : out   STD_LOGIC_VECTOR((N_ST_LINES-1) downto 0);
             PREV_ST   : in    STD_LOGIC_VECTOR((N_ST_LINES-1) downto 0);
             CURR_GR   : in    STD_LOGIC_VECTOR((N_G_LINES-1) downto 0);
             NEXT_GR   : out   STD_LOGIC_VECTOR((N_G_LINES-1) downto 0);
             INPUTS    : in    STD_LOGIC_VECTOR((N_IN_OUT-1) downto 0);
             OUTPUTS   : out   STD_LOGIC_VECTOR((N_IN_OUT-1) downto 0);
             GR_OK     : out   STD_LOGIC;
             CLK1      : in    STD_LOGIC;
             GR_CLK    : in    STD_LOGIC;
             GR_CLR    : in    STD_LOGIC);

-- Entity attributes -----------------------------------------------------------

        attribute OVERLAP      of SINGLE_BLOCK        : entity       is "YES";
        attribute RTMAX        of SINGLE_BLOCK        : entity       is "X4Y4";

end SINGLE_BLOCK;


architecture STRUCT of SINGLE_BLOCK is

-- Components declaration ------------------------------------------------------

        component N_BUF is
                generic(N      : integer := 8;
                        OS     : integer := 1);
                port(I         : in    STD_LOGIC_VECTOR((N-1) downto 0);
                     O         : out   STD_LOGIC_VECTOR((N-1) downto 0));
        end component;

        component PREG4N is
                generic(N      : integer := 8;
                        OS     : integer := 1);
                port(CLK       : in    STD_LOGIC;
                     SEL       : in    STD_LOGIC_VECTOR(1 downto 0);
                     Q         : out   STD_LOGIC_VECTOR((N-1) downto 0));
        end component;
```

```
        component PREG_EQ_COMPN is
                generic(N       : integer := 8;
                        LS      : integer := 1;
                        IV      : integer := 0);
                port(CLK        : in    STD_LOGIC;
                     I          : in    STD_LOGIC_VECTOR((N-1) downto 0);
                     O          : out   STD_LOGIC);
        end component;

        component REC_ELEM is
                generic(N_IN_OUT        : integer := 16;
                        N_CALLS         : integer := 4);

                port(GR_CLK             : in    STD_LOGIC;
                     GR_CLR             : in    STD_LOGIC;
                     START              : in    STD_LOGIC;
                     FINISH             : out   STD_LOGIC;
                     OH_CURR_ST         : out   STD_LOGIC_VECTOR((N_CALLS-1) downto 0);
                     OH_PREV_ST         : in    STD_LOGIC_VECTOR((N_CALLS-1) downto 0);
                     INPUTS             : in    STD_LOGIC_VECTOR((N_IN_OUT-1) downto 0);
                     OUTPUTS            : out   STD_LOGIC_VECTOR((N_IN_OUT-1) downto 0));
        end component;

-- Entity internal signals ---------------------------------------------------

        signal IN_BUF_O                 : STD_LOGIC_VECTOR((N_IN_OUT-1) downto 0);
        signal OH_PREV_ST               : STD_LOGIC_VECTOR(((2**N_ST_LINES)-1) downto 0);
        signal DEC_OUT                  : STD_LOGIC_VECTOR(((2**N_ST_LINES)-1) downto 0);
        signal OH_CURR_ST               : STD_LOGIC_VECTOR(((2**N_ST_LINES)-1) downto 0);
        signal COD_OUT                  : STD_LOGIC_VECTOR((N_ST_LINES-1) downto 0);
        signal START_BUF_O, CALL_AUX    : STD_LOGIC;

-- Architecture attributes ---------------------------------------------------

        attribute RLOC of IN_BUF    : label is "X3Y20";

        attribute RLOC of START_BUF : label is "X3Y17";

        attribute RLOC of R_BLK     : label is "X4Y16";
        attribute BBOX of R_BLK     : label is "X,((2*(2**N_ST_LINES))+4),Y,(N_IN_OUT+4),";

        attribute RLOC of GR_MAP_REG: label is "X7Y0";

        attribute RLOC of GR_ID_REG : label is "X5Y0";

        attribute RLOC of DEC_AND0  : label is "X7Y14";
        attribute RLOC of DEC_AND1  : label is "X9Y14";
        attribute RLOC of DEC_AND2  : label is "X11Y14";
        attribute RLOC of DEC_AND3  : label is "X13Y14";
        attribute RLOC of DEC_AND4  : label is "X7Y15";
        attribute RLOC of DEC_AND5  : label is "X9Y15";
        attribute RLOC of DEC_AND6  : label is "X11Y15";
        attribute RLOC of DEC_AND7  : label is "X13Y15";

        attribute RLOC of COD_OR0   : label is "X10Y15";
        attribute RLOC of COD_OR1   : label is "X12Y15";
        attribute RLOC of COD_OR2   : label is "X6Y15";
        attribute RLOC of COD_OR3   : label is "X8Y13";

-- Implementation ------------------------------------------------------------

begin

-- Input buffer --------------------------------------------------------

        IN_BUF          : N_BUF         generic map(N  => N_IN_OUT,
                                                    OS  => 1)
                                        port map(I     => INPUTS,
                                                 O     => IN_BUF_O);

-- Start buffer --------------------------------------------------------

        START_BUF       : BUF           port map(I     => START,
```

```
                                        O     => START_BUF_O);

-- Reconfigurable block ---------------------------------------

       R_BLK      : REC_ELEM          generic map(N_IN_OUT   => N_IN_OUT,
                                                  N_CALLS    => 2**N_ST_LINES)
                                      port map(GR_CLK        => GR_CLK,
                                               GR_CLR        => GR_CLR,
                                               START         => START_BUF_O,
                                               FINISH        => FINISH,
                                               OH_CURR_ST    => OH_CURR_ST,
                                               OH_PREV_ST    => OH_PREV_ST,
                                               INPUTS        => IN_BUF_O,
                                               OUTPUTS       => OUTPUTS);

-- Code converter ---------------------------------------------

       GR_MAP_REG     : PREG4N       generic map(N  => N_G_LINES,
                                                 OS => N_SP_CELLS)
                                     port map(CLK   => CLK1,
                                              SEL   => COD_OUT,
                                              Q     => NEXT_GR);

-- Graph identifcation register-------------------------------------

       GR_ID_REG      : PREG_EQ_COMPN       generic map(N  => N_G_LINES,
                                                        LS => N_SP_CELLS,
                                                        IV => 2**(N_G_LINES-1))
                                            port map(CLK   => CLK1,
                                                     I     => CURR_GR,
                                                     O     => GR_OK);

-- Decoder ----------------------------------------------------

       DEC_AND0       : AND2B2       port map(I0    => PREV_ST(0),
                                              I1    => PREV_ST(1),
                                              O     => DEC_OUT(0));

       DEC_AND1       : AND2B1       port map(I0    => PREV_ST(1),
                                              I1    => PREV_ST(0),
                                              O     => DEC_OUT(1));

       DEC_AND2       : AND2B1       port map(I0    => PREV_ST(0),
                                              I1    => PREV_ST(1),
                                              O     => DEC_OUT(2));

       DEC_AND3       : AND2         port map(I0    => PREV_ST(0),
                                              I1    => PREV_ST(1),
                                              O     => DEC_OUT(3));

       DEC_AND4       : AND2         port map(I0    => DEC_OUT(0),
                                              I1    => RET,
                                              O     => OH_PREV_ST(0));

       DEC_AND5       : AND2         port map(I0    => DEC_OUT(1),
                                              I1    => RET,
                                              O     => OH_PREV_ST(1));

       DEC_AND6       : AND2         port map(I0    => DEC_OUT(2),
                                              I1    => RET,
                                              O     => OH_PREV_ST(2));

       DEC_AND7       : AND2         port map(I0    => DEC_OUT(3),
                                              I1    => RET,
                                              O     => OH_PREV_ST(3));

-- Coder ------------------------------------------------------

       COD_OR0        : OR2          port map(I0    => OH_CURR_ST(1),
                                              I1    => OH_CURR_ST(3),
                                              O     => COD_OUT(0));

       COD_OR1        : OR2          port map(I0    => OH_CURR_ST(2),
                                              I1    => OH_CURR_ST(3),
```

```
                                          O     => COD_OUT(1));

    COD_OR2        : OR2          port map(I0    => OH_CURR_ST(0),
                                          I1    => OH_CURR_ST(1),
                                          O     => CALL_AUX);

    COD_OR3        : OR2          port map(I0    => CALL_AUX,
                                          I1    => COD_OUT(1),
                                          O     => CALL);

    CURR_ST((N_ST_LINES-1) downto 0) <= COD_OUT((N_ST_LINES-1) downto 0);

end STRUCT;
```

Dual_Block.vhd

```
--------------------------------------------------------------------------------
--
-- Dual implementation block
--
--------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library PRIMS;
use PRIMS.XC6000_COMPONENTS.ALL;

library XC6200LIB;
use XC6200LIB.ARRAY_PACK.N_M2_1;

library WORK;
use WORK.SINGLE_BLOCK;


entity DUAL_BLOCK is

        generic(N_IN_OUT       : integer := 16;
                N_G_LINES      : integer := 6;
                N_ST_LINES     : integer := 2;
                N_SP_CELLS     : integer := 2);

        port(START             : in    STD_LOGIC;
             FINISH            : out   STD_LOGIC;
             CALL              : out   STD_LOGIC;
             RET               : in    STD_LOGIC;
             CURR_ST           : out   STD_LOGIC_VECTOR((N_ST_LINES-1) downto 0);
             PREV_ST           : in    STD_LOGIC_VECTOR((N_ST_LINES-1) downto 0);
             CURR_GR           : in    STD_LOGIC_VECTOR((N_G_LINES-1) downto 0);
             NEXT_GR           : out   STD_LOGIC_VECTOR((N_G_LINES-1) downto 0);
             INPUTS            : in    STD_LOGIC_VECTOR((N_IN_OUT-1) downto 0);
             OUTPUTS           : out   STD_LOGIC_VECTOR((N_IN_OUT-1) downto 0);
             GR_OK             : out   STD_LOGIC;
             MUX_SEL_IN        : in    STD_LOGIC;
             MUX_SEL_OUT       : out   STD_LOGIC;
             CLK1              : in    STD_LOGIC;
             GR_CLK            : in    STD_LOGIC;
             GR_CLR            : in    STD_LOGIC);

-- Entity attributes -----------------------------------------------------------

        attribute OVERLAP      of DUAL_BLOCK  : entity       is "YES";
        attribute RTMAX        of DUAL_BLOCK  : entity       is "X16Y4";

end DUAL_BLOCK;


architecture STRUCT of DUAL_BLOCK is

-- Components declaration ------------------------------------------------------

        component N_M2_1 is
                generic(N      : integer := 8;
                        OS     : integer := 1);

                port(S         : in    STD_LOGIC;
                     I0        : in    STD_LOGIC_VECTOR((N-1) downto 0);
                     I1        : in    STD_LOGIC_VECTOR((N-1) downto 0);
                     O         : out   STD_LOGIC_VECTOR((N-1) downto 0));
        end component;

        component SINGLE_BLOCK is
                generic(N_IN_OUT       : integer := 16;
                        N_G_LINES      : integer := 5;
                        N_ST_LINES     : integer := 2;
```

```
                    N_SP_CELLS     : integer := 2);

            port(START     : in    STD_LOGIC;
                 FINISH    : out   STD_LOGIC;
                 CALL      : out   STD_LOGIC;
                 RET       : in    STD_LOGIC;
                 CURR_ST   : out   STD_LOGIC_VECTOR((N_ST_LINES-1) downto 0);
                 PREV_ST   : in    STD_LOGIC_VECTOR((N_ST_LINES-1) downto 0);
                 CURR_GR   : in    STD_LOGIC_VECTOR((N_G_LINES-1) downto 0);
                 NEXT_GR   : out   STD_LOGIC_VECTOR((N_G_LINES-1) downto 0);
                 INPUTS    : in    STD_LOGIC_VECTOR((N_IN_OUT-1) downto 0);
                 OUTPUTS   : out   STD_LOGIC_VECTOR((N_IN_OUT-1) downto 0);
                 GR_OK     : out   STD_LOGIC;
                 CLK1      : in    STD_LOGIC;
                 GR_CLK    : in    STD_LOGIC;
                 GR_CLR    : in    STD_LOGIC);
      end component;

-- Internal signals ------------------------------------------------------

      signal START_0, START_1       : STD_LOGIC;

      signal FINISH_0, FINISH_1     : STD_LOGIC;

      signal CALL_0, CALL_1         : STD_LOGIC;

      signal RET_0, RET_1           : STD_LOGIC;

      signal CURR_ST_0, CURR_ST_1   : STD_LOGIC_VECTOR((N_ST_LINES-1) downto 0);

      signal NEXT_GR_0, NEXT_GR_1   : STD_LOGIC_VECTOR((N_G_LINES-1) downto 0);

      signal OUTPUTS_0, OUTPUTS_1   : STD_LOGIC_VECTOR((N_IN_OUT-1) downto 0);

      signal GR_OK_0, GR_OK_1       : STD_LOGIC;

      signal M_SEL_BUF_O            : STD_LOGIC;

-- Architecture attributes -----------------------------------------------

      attribute RLOC        of CELL_0      : label  is "X0Y0";
      attribute RLOC        of CELL_1      : label  is "X16Y0";

      attribute RLOC        of M_FINISH    : label  is "X16Y16";

      attribute RLOC        of M_CALL      : label  is "X16Y13";

      attribute RLOC        of M_CURR_ST   : label  is "X16Y12";

      attribute RLOC        of M_NEXT_GR   : label  is "X16Y0";

      attribute RLOC        of M_OUT       : label  is "X16Y20";

      attribute RLOC        of M_SEL_BUF   : label  is "X16Y11";
      attribute RLOC        of GR_OK_OR    : label  is "X16Y7";

      attribute RLOC        of START_EN_0  : label  is "X16Y17";
      attribute RLOC        of START_EN_1  : label  is "X17Y17";

      attribute RLOC        of RET_EN_0    : label  is "X16Y15";
      attribute RLOC        of RET_EN_1    : label  is "X17Y15";

-- Implementation --------------------------------------------------------

begin

-- Implementation cell 0 -------------------------------------------

      CELL_0  : SINGLE_BLOCK generic map(N_IN_OUT   => N_IN_OUT,
                                         N_G_LINES  => N_G_LINES,
                                         N_ST_LINES => N_ST_LINES,
                                         N_SP_CELLS => N_SP_CELLS)

                              port map(START         => START_0,
```

```
                                    FINISH        => FINISH_0,
                                    CALL          => CALL_0,
                                    RET           => RET_0,
                                    CURR_ST       => CURR_ST_0,
                                    PREV_ST       => PREV_ST,
                                    CURR_GR       => CURR_GR,
                                    NEXT_GR       => NEXT_GR_0,
                                    INPUTS        => INPUTS,
                                    OUTPUTS       => OUTPUTS_0,
                                    GR_OK         => GR_OK_0,
                                    CLK1          => CLK1,
                                    GR_CLK        => GR_CLK,
                                    GR_CLR        => GR_CLR);

-- Implementation cell 1 ------------------------------------------

       CELL_1  : SINGLE_BLOCK generic map(N_IN_OUT   => N_IN_OUT,
                                          N_G_LINES  => N_G_LINES,
                                          N_ST_LINES => N_ST_LINES,
                                          N_SP_CELLS => N_SP_CELLS)

                               port map(START        => START_1,
                                        FINISH       => FINISH_1,
                                        CALL         => CALL_1,
                                        RET          => RET_1,
                                        CURR_ST      => CURR_ST_1,
                                        PREV_ST      => PREV_ST,
                                        CURR_GR      => CURR_GR,
                                        NEXT_GR      => NEXT_GR_1,
                                        INPUTS       => INPUTS,
                                        OUTPUTS      => OUTPUTS_1,
                                        GR_OK        => GR_OK_1,
                                        CLK1         => CLK1,
                                        GR_CLK       => GR_CLK,
                                        GR_CLR       => GR_CLR);

-- Finish multiplexer ---------------------------------------------

       M_FINISH       : M2_1         port map(S0    => M_SEL_BUF_O,
                                              D0    => FINISH_0,
                                              D1    => FINISH_1,
                                              O     => FINISH);

-- Call multiplexer -----------------------------------------------

       M_CALL         : M2_1         port map(S0    => M_SEL_BUF_O,
                                              D0    => CALL_0,
                                              D1    => CALL_1,
                                              O     => CALL);

-- Current state multiplexer --------------------------------------

       M_CURR_ST      : N_M2_1       generic map(N  => N_ST_LINES,
                                                  OS => N_SP_CELLS)

                                     port map(S     => M_SEL_BUF_O,
                                              I0    => CURR_ST_0,
                                              I1    => CURR_ST_1,
                                              O     => CURR_ST);

-- Next graph multiplexer -----------------------------------------

       M_NEXT_GR      : N_M2_1       generic map(N  => N_G_LINES,
                                                  OS => N_SP_CELLS)

                                     port map(S     => M_SEL_BUF_O,
                                              I0    => NEXT_GR_0,
                                              I1    => NEXT_GR_1,
                                              O     => NEXT_GR);

-- Output multiplexer ---------------------------------------------

       M_OUT          : N_M2_1       generic map(N  => N_IN_OUT,
                                                  OS => 1)
```

```
                                    port map(S      => M_SEL_BUF_O,
                                             I0     => OUTPUTS_0,
                                             I1     => OUTPUTS_1,
                                             O      => OUTPUTS);

-- Other components -------------------------------------------------

        M_SEL_BUF       : BUF           port map(I      => MUX_SEL_IN,
                                                 O      => M_SEL_BUF_O);

        GR_OK_OR        : OR2           port map(I0     => GR_OK_0,
                                                 I1     => GR_OK_1,
                                                 O      => GR_OK);

        MUX_SEL_OUT     <= GR_OK_1;

        START_EN_0      : AND2B1        port map(I0     => M_SEL_BUF_O,
                                                 I1     => START,
                                                 O      => START_0);

        START_EN_1      : AND2          port map(I0     => M_SEL_BUF_O,
                                                 I1     => START,
                                                 O      => START_1);

        RET_EN_0        : AND2B1        port map(I0     => M_SEL_BUF_O,
                                                 I1     => RET,
                                                 O      => RET_0);

        RET_EN_1        : AND2          port map(I0     => M_SEL_BUF_O,
                                                 I1     => RET,
                                                 O      => RET_1);

end STRUCT;
```

Quad_Block.vhd

```
--------------------------------------------------------------------------------
--
-- Quad implementation block
--
--------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library PRIMS;
use PRIMS.XC6000_COMPONENTS.ALL;

library XC6200LIB;
use XC6200LIB.ARRAY_PACK.N_M2_1;

library WORK;
use WORK.DUAL_BLOCK;


entity QUAD_BLOCK is

        generic(N_IN_OUT       : integer := 16;
                N_G_LINES      : integer := 6;
                N_ST_LINES     : integer := 2;
                N_SP_CELLS     : integer := 2);

        port(START             : in    STD_LOGIC;
             FINISH            : out   STD_LOGIC;
             CALL              : out   STD_LOGIC;
             RET               : in    STD_LOGIC;
             CURR_ST           : out   STD_LOGIC_VECTOR((N_ST_LINES-1) downto 0);
             PREV_ST           : in    STD_LOGIC_VECTOR((N_ST_LINES-1) downto 0);
             CURR_GR           : in    STD_LOGIC_VECTOR((N_G_LINES-1) downto 0);
             NEXT_GR           : out   STD_LOGIC_VECTOR((N_G_LINES-1) downto 0);
             INPUTS            : in    STD_LOGIC_VECTOR((N_IN_OUT-1) downto 0);
             OUTPUTS           : out   STD_LOGIC_VECTOR((N_IN_OUT-1) downto 0);
             GR_OK             : out   STD_LOGIC;
             MUX_SEL_IN        : in    STD_LOGIC_VECTOR(1 downto 0);
             MUX_SEL_OUT       : out   STD_LOGIC_VECTOR(1 downto 0);
             CLK1              : in    STD_LOGIC;
             GR_CLK            : in    STD_LOGIC;
             GR_CLR            : in    STD_LOGIC);

-- Entity attributes -----------------------------------------------------------

        attribute OVERLAP      of QUAD_BLOCK  : entity       is "YES";
        attribute RTMAX        of QUAD_BLOCK  : entity       is "X16Y4";

end QUAD_BLOCK;


architecture STRUCT of QUAD_BLOCK is

-- Components declaration ------------------------------------------------------

        component N_M2_1 is
                generic(N      : integer := 8;
                        OS     : integer := 1);

                port(S         : in    STD_LOGIC;
                     I0        : in    STD_LOGIC_VECTOR((N-1) downto 0);
                     I1        : in    STD_LOGIC_VECTOR((N-1) downto 0);
                     O         : out   STD_LOGIC_VECTOR((N-1) downto 0));
        end component;

        component DUAL_BLOCK is
                generic(N_IN_OUT       : integer := 16;
                        N_G_LINES      : integer := 6;
                        N_ST_LINES     : integer := 2;
```

```
                    N_SP_CELLS     : integer := 2);

            port(START             : in    STD_LOGIC;
                 FINISH            : out   STD_LOGIC;
                 CALL              : out   STD_LOGIC;
                 RET               : in    STD_LOGIC;
                 CURR_ST           : out   STD_LOGIC_VECTOR((N_ST_LINES-1) downto 0);
                 PREV_ST           : in    STD_LOGIC_VECTOR((N_ST_LINES-1) downto 0);
                 CURR_GR           : in    STD_LOGIC_VECTOR((N_G_LINES-1) downto 0);
                 NEXT_GR           : out   STD_LOGIC_VECTOR((N_G_LINES-1) downto 0);
                 INPUTS            : in    STD_LOGIC_VECTOR((N_IN_OUT-1) downto 0);
                 OUTPUTS           : out   STD_LOGIC_VECTOR((N_IN_OUT-1) downto 0);
                 GR_OK             : out   STD_LOGIC;
                 MUX_SEL_IN        : in    STD_LOGIC;
                 MUX_SEL_OUT       : out   STD_LOGIC;
                 CLK1              : in    STD_LOGIC;
                 GR_CLK            : in    STD_LOGIC;
                 GR_CLR            : in    STD_LOGIC);
      end component;

-- Entity internal signals ---------------------------------------------------

      signal START_0, START_1              : STD_LOGIC;

      signal FINISH_0, FINISH_1            : STD_LOGIC;

      signal CALL_0, CALL_1                : STD_LOGIC;

      signal RET_0, RET_1                  : STD_LOGIC;

      signal CURR_ST_0, CURR_ST_1          : STD_LOGIC_VECTOR((N_ST_LINES-1) downto 0);

      signal NEXT_GR_0, NEXT_GR_1          : STD_LOGIC_VECTOR((N_G_LINES-1) downto 0);

      signal OUTPUTS_0, OUTPUTS_1          : STD_LOGIC_VECTOR((N_IN_OUT-1) downto 0);

      signal GR_OK_0, GR_OK_1              : STD_LOGIC;

      signal M_SEL_BUF_O                   : STD_LOGIC;

      signal M_SEL_OUT_0, M_SEL_OUT_1      : STD_LOGIC;

-- Architecture attributes ---------------------------------------------------

      attribute RLOC  of DUAL_0     : label  is "X0Y0";
      attribute RLOC  of DUAL_1     : label  is "X32Y0";

      attribute RLOC  of M_FINISH   : label  is "X32Y16";

      attribute RLOC  of M_CALL     : label  is "X33Y13";

      attribute RLOC  of M_CURR_ST  : label  is "X32Y12";

      attribute RLOC  of M_NEXT_GR  : label  is "X32Y0";

      attribute RLOC  of M_OUT      : label  is "X32Y20";

      attribute RLOC  of M_SEL_BUF  : label  is "X32Y11";
      attribute RLOC  of GR_OK_OR   : label  is "X34Y7";
      attribute RLOC  of M_SEL_MUX  : label  is "X32Y7";

      attribute RLOC  of START_EN_0 : label  is "X32Y17";
      attribute RLOC  of START_EN_1 : label  is "X34Y17";

      attribute RLOC  of RET_EN_0   : label  is "X32Y15";
      attribute RLOC  of RET_EN_1   : label  is "X34Y15";

-- Implementation ------------------------------------------------------------

begin

-- Implementation dual cell 0 ------------------------------------

      DUAL_0  : DUAL_BLOCK   generic map(N_IN_OUT   => N_IN_OUT,
```

```
                                    N_G_LINES     => N_G_LINES,
                                    N_ST_LINES    => N_ST_LINES,
                                    N_SP_CELLS    => N_SP_CELLS)

                                port map(START         => START_0,
                                         FINISH        => FINISH_0,
                                         CALL          => CALL_0,
                                         RET           => RET_0,
                                         CURR_ST       => CURR_ST_0,
                                         PREV_ST       => PREV_ST,
                                         CURR_GR       => CURR_GR,
                                         NEXT_GR       => NEXT_GR_0,
                                         INPUTS        => INPUTS,
                                         OUTPUTS       => OUTPUTS_0,
                                         GR_OK         => GR_OK_0,
                                         MUX_SEL_IN    => MUX_SEL_IN(0),
                                         MUX_SEL_OUT   => M_SEL_OUT_0,
                                         CLK1          => CLK1,
                                         GR_CLK        => GR_CLK,
                                         GR_CLR        => GR_CLR);

-- Implementation dual cell 1 -------------------------------------

       DUAL_1  : DUAL_BLOCK   generic map(N_IN_OUT   => N_IN_OUT,
                                          N_G_LINES  => N_G_LINES,
                                          N_ST_LINES => N_ST_LINES,
                                          N_SP_CELLS => N_SP_CELLS)

                                port map(START         => START_1,
                                         FINISH        => FINISH_1,
                                         CALL          => CALL_1,
                                         RET           => RET_1,
                                         CURR_ST       => CURR_ST_1,
                                         PREV_ST       => PREV_ST,
                                         CURR_GR       => CURR_GR,
                                         NEXT_GR       => NEXT_GR_1,
                                         INPUTS        => INPUTS,
                                         OUTPUTS       => OUTPUTS_1,
                                         GR_OK         => GR_OK_1,
                                         MUX_SEL_IN    => MUX_SEL_IN(0),
                                         MUX_SEL_OUT   => M_SEL_OUT_1,
                                         CLK1          => CLK1,
                                         GR_CLK        => GR_CLK,
                                         GR_CLR        => GR_CLR);

-- Finish multiplexer ---------------------------------------------

       M_FINISH       : M2_1         port map(S0    => M_SEL_BUF_O,
                                              D0    => FINISH_0,
                                              D1    => FINISH_1,
                                              O     => FINISH);

-- Call multiplexer -----------------------------------------------

       M_CALL         : M2_1         port map(S0    => M_SEL_BUF_O,
                                              D0    => CALL_0,
                                              D1    => CALL_1,
                                              O     => CALL);

-- Current state multiplexer --------------------------------------

       M_CURR_ST      : N_M2_1       generic map(N  => N_ST_LINES,
                                                 OS => N_SP_CELLS)

                                     port map(S     => M_SEL_BUF_O,
                                              I0    => CURR_ST_0,
                                              I1    => CURR_ST_1,
                                              O     => CURR_ST);

-- Next graph multiplexer -----------------------------------------

       M_NEXT_GR      : N_M2_1       generic map(N  => N_G_LINES,
                                                 OS => N_SP_CELLS)
```

```
                                    port map(S     => M_SEL_BUF_O,
                                             I0    => NEXT_GR_0,
                                             I1    => NEXT_GR_1,
                                             O     => NEXT_GR);

-- Output multiplexer ------------------------------------------------

       M_OUT          : N_M2_1       generic map(N  => N_IN_OUT,
                                                 OS => 1)

                                    port map(S     => M_SEL_BUF_O,
                                             I0    => OUTPUTS_0,
                                             I1    => OUTPUTS_1,
                                             O     => OUTPUTS);

-- Other components --------------------------------------------------

       M_SEL_BUF      : BUF          port map(I     => MUX_SEL_IN(1),
                                             O     => M_SEL_BUF_O);

       GR_OK_OR       : OR2          port map(I0    => GR_OK_0,
                                             I1    => GR_OK_1,
                                             O     => GR_OK);

       MUX_SEL_OUT(1) <= GR_OK_1;

       M_SEL_MUX      : M2_1         port map(S0    => GR_OK_1,
                                             D0    => M_SEL_OUT_0,
                                             D1    => M_SEL_OUT_1,
                                             O     => MUX_SEL_OUT(0));

       START_EN_0     : AND2B1       port map(I0    => M_SEL_BUF_O,
                                             I1    => START,
                                             O     => START_0);

       START_EN_1     : AND2         port map(I0    => M_SEL_BUF_O,
                                             I1    => START,
                                             O     => START_1);

       RET_EN_0       : AND2B1       port map(I0    => M_SEL_BUF_O,
                                             I1    => RET,
                                             O     => RET_0);

       RET_EN_1       : AND2         port map(I0    => M_SEL_BUF_O,
                                             I1    => RET,
                                             O     => RET_1);

end STRUCT;
```

Stack.vhd

```
--------------------------------------------------------------------------------
--
-- 8 bits, 256 levels stack built in SRAM
--
--------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library PRIMS;
use PRIMS.XC6000_COMPONENTS.ALL;

library XC6200LIB;
use XC6200LIB.GATES_PACK.NORN;
use XC6200LIB.MISC_PACK.STK_CTRLN_1;
use XC6200LIB.RAM_PACK.RW_RAM8;


entity STACK is

        port(CLK                : in    STD_LOGIC;
             N_STACK_RD         : in    STD_LOGIC;
             STACK_WR           : in    STD_LOGIC;
             STACK_OPAD_EN      : in    STD_LOGIC;
             SP_INC             : in    STD_LOGIC;
             SP_DEC             : in    STD_LOGIC;
             SP_RESET           : in    STD_LOGIC;
             STACK_IN           : in    STD_LOGIC_VECTOR(7 downto 0);
             STACK_OUT          : out   STD_LOGIC_VECTOR(7 downto 0);
             SP_LIMIT           : out   STD_LOGIC);

-- Entity attributes -----------------------------------------------------------

        attribute OVERLAP      of STACK       : entity       is "YES";
        attribute RTDEFER      of STACK       : entity       is "";

end STACK;


architecture STRUCT of STACK is

-- Components declaration ------------------------------------------------------

        component STK_CTRLN_1 is
                generic(N       : integer := 8);

                port(CLK        : in    STD_LOGIC;
                     CLR        : in    STD_LOGIC;
                     INC        : in    STD_LOGIC;
                     DEC        : in    STD_LOGIC;
                     Q          : out   STD_LOGIC_VECTOR((N-1) downto 0));
        end component;

        component NORN is
                generic(N       : integer := 8);

                port(I          : in    STD_LOGIC_VECTOR((N-1) downto 0);
                     O          : out   STD_LOGIC);
        end component;

-- Entity internal signals -----------------------------------------------------

        signal RD, WR, CLK_INV_O       : STD_LOGIC;
        signal ADDRESS                 : STD_LOGIC_VECTOR(7 downto 0);

-- Architecture attributes -----------------------------------------------------

        attribute RLOC         of WR_INV      : label        is "X14Y0";
        attribute RLOC         of CLK_INV     : label        is "X3Y0";
```

```
        attribute RLOC         of SRAM        : label        is "X0Y0";
        attribute RLOC         of CTRL        : label        is "X0Y0";

-- Implementation -----------------------------------------------------------

begin

        RD      <= N_STACK_RD;

        WR_INV  : INV          port map(I     => STACK_WR,
                                        O     => WR);

        CLK_INV : INV          port map(I     => CLK,
                                        O     => CLK_INV_O);

-- SRAM access macro --------------------------------------------------

        SRAM    : RW_RAM8      port map(RD            => RD,
                                        WR            => WR,
                                        OPAD_EN       => STACK_OPAD_EN,
                                        ADDRESS       => ADDRESS,
                                        DIN           => STACK_IN,
                                        DOUT          => STACK_OUT);

-- Stack controller ---------------------------------------------------

        CTRL    : STK_CTRLN_1  generic map(N  => 8)

                               port map(CLK   => CLK_INV_O,
                                        CLR   => SP_RESET,
                                        INC   => SP_INC,
                                        DEC   => SP_DEC,
                                        Q     => ADDRESS);

-- Stack pointer zero detector ----------------------------------------

        Z_DET   : NORN         generic map(N  => 8)

                               port map(I     => ADDRESS(7 downto 0),
                                        O     => SP_LIMIT);

end STRUCT;
```

Blocks_IO.vhd

```
--------------------------------------------------------------------------------
--
-- Implementation blocks with I/O registers
--
--------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library PRIMS;
use PRIMS.XC6000_COMPONENTS.ALL;

library XC6200LIB;
use XC6200LIB.REG_PACK.PREGN;
use XC6200LIB.REG_PACK.REGN;
use XC6200LIB.REG_PACK.REG_MUXN;

library WORK;
use WORK.QUAD_BLOCK;
use WORK.STACK;


entity BLOCKS_IO is

        generic(N_IN_OUT       : integer := 16;
                N_G_LINES      : integer := 6;
                N_ST_LINES     : integer := 2;
                N_SP_CELLS     : integer := 2);

        port(CLK1              : in    STD_LOGIC;
             CLK2              : in    STD_LOGIC;
             START             : in    STD_LOGIC;
             FINISH            : out   STD_LOGIC;
             CALL              : out   STD_LOGIC;
             RET               : in    STD_LOGIC;
             GR_OK             : out   STD_LOGIC;
             SP_LIMIT          : out   STD_LOGIC;
             STACK_WR          : in    STD_LOGIC;
             N_STACK_RD        : in    STD_LOGIC;
             STACK_OPAD_EN     : in    STD_LOGIC;
             SP_INC            : in    STD_LOGIC;
             SP_DEC            : in    STD_LOGIC;
             SP_RESET          : in    STD_LOGIC;
             CURR_GR_CLK       : in    STD_LOGIC;
             CURR_GR_CLR       : in    STD_LOGIC;
             PREV_ST_CLK       : in    STD_LOGIC;
             IN_CLK            : in    STD_LOGIC;
             OUT_CLK           : in    STD_LOGIC;
             OUT_CLR           : in    STD_LOGIC;
             GR_CLK            : in    STD_LOGIC;
             GR_CLR            : in    STD_LOGIC);

-- Entity attributes -----------------------------------------------------------

        attribute OVERLAP      of BLOCKS_IO   : entity       is "YES";
        attribute RTMAX        of BLOCKS_IO   : entity       is "X0Y16";

end BLOCKS_IO;


architecture STRUCT of BLOCKS_IO is

-- Components declaration ------------------------------------------------------

        component PREGN is
                generic(N      : integer := 8;
                        OS     : integer := 1;
                        IV     : integer := 0);
```

```
            port(CLK        : in    STD_LOGIC;
                 Q          : out   STD_LOGIC_VECTOR((N-1) downto 0));
        end component;

        component REGN is
            generic(N       : integer := 8;
                    OS      : integer := 1);

            port(D          : in    STD_LOGIC_VECTOR((N-1) downto 0);
                 CLK        : in    STD_LOGIC;
                 CLR        : in    STD_LOGIC;
                 Q          : out   STD_LOGIC_VECTOR((N-1) downto 0));
        end component;

        component REG_MUXN is
            generic(N       : integer := 8;
                    OS      : integer := 1);

            port(CLK        : in    STD_LOGIC;
                 CLR        : in    STD_LOGIC;
                 SEL        : in    STD_LOGIC;
                 I0         : in    STD_LOGIC_VECTOR((N-1) downto 0);
                 I1         : in    STD_LOGIC_VECTOR((N-1) downto 0);
                 Q          : out   STD_LOGIC_VECTOR((N-1) downto 0));
        end component;

        component QUAD_BLOCK is
            generic(N_IN_OUT       : integer := 16;
                    N_G_LINES      : integer := 6;
                    N_ST_LINES     : integer := 2;
                    N_SP_CELLS     : integer := 2);

            port(START             : in    STD_LOGIC;
                 FINISH            : out   STD_LOGIC;
                 CALL              : out   STD_LOGIC;
                 RET               : in    STD_LOGIC;
                 CURR_ST           : out   STD_LOGIC_VECTOR((N_ST_LINES-1) downto 0);
                 PREV_ST           : in    STD_LOGIC_VECTOR((N_ST_LINES-1) downto 0);
                 CURR_GR           : in    STD_LOGIC_VECTOR((N_G_LINES-1) downto 0);
                 NEXT_GR           : out   STD_LOGIC_VECTOR((N_G_LINES-1) downto 0);
                 INPUTS            : in    STD_LOGIC_VECTOR((N_IN_OUT-1) downto 0);
                 OUTPUTS           : out   STD_LOGIC_VECTOR((N_IN_OUT-1) downto 0);
                 GR_OK             : out   STD_LOGIC;
                 MUX_SEL_IN        : in    STD_LOGIC_VECTOR(1 downto 0);
                 MUX_SEL_OUT       : out   STD_LOGIC_VECTOR(1 downto 0);
                 CLK1              : in    STD_LOGIC;
                 GR_CLK            : in    STD_LOGIC;
                 GR_CLR            : in    STD_LOGIC);
        end component;

-- Entity internal signals ---------------------------------------------------

        signal MUX_SEL                      : STD_LOGIC_VECTOR(1 downto 0);

        signal INPUTS                       : STD_LOGIC_VECTOR((N_IN_OUT-1) downto 0);
        signal OUTPUTS                      : STD_LOGIC_VECTOR((N_IN_OUT-1) downto 0);

        signal ACT_BLK_GND_O                : STD_LOGIC;

        signal LF_FF_O, LF_CLR_GND_O        : STD_LOGIC;
        signal P_ST_CLR_GND_O               : STD_LOGIC;

        signal PREV_ST, CURR_ST             : STD_LOGIC_VECTOR((N_ST_LINES-1) downto 0);

        signal PREV_GR, CURR_GR, NEXT_GR    : STD_LOGIC_VECTOR((N_G_LINES-1) downto 0);

        signal STACK_IN, STACK_OUT          : STD_LOGIC_VECTOR(7 downto 0);

-- Architecture attributes ---------------------------------------------------

        attribute RLOC of CELLS             : label        is "X0Y0";

        attribute RLOC of ACT_BLK_REG       : label        is "X31Y4";
```

```
        attribute RLOC of IN_REG             : label        is "X34Y21";
        attribute RLOC of OUT_REG            : label        is "X33Y21";

        attribute RLOC of LF_FF              : label        is "X33Y15";
        attribute RLOC of LF_BUF             : label        is "X34Y20";

        attribute RLOC of CURR_GR_REG        : label        is "X35Y0";
        attribute RLOC of PREV_ST_REG        : label        is "X63Y12";

        attribute RLOC of STACK_MEM          : label        is "X0Y0";

-- Implementation -----------------------------------------------------------

begin

-- Implementation cells -----------------------------------------------------

        CELLS   : QUAD_BLOCK   generic map(N_IN_OUT   => N_IN_OUT,
                                           N_G_LINES  => N_G_LINES,
                                           N_ST_LINES => N_ST_LINES,
                                           N_SP_CELLS => N_SP_CELLS)

                               port map(START         => START,
                                        FINISH        => FINISH,
                                        CALL          => CALL,
                                        RET           => RET,
                                        CURR_ST       => CURR_ST,
                                        PREV_ST       => PREV_ST,
                                        CURR_GR       => CURR_GR,
                                        NEXT_GR       => NEXT_GR,
                                        INPUTS        => INPUTS,
                                        OUTPUTS       => OUTPUTS,
                                        GR_OK         => GR_OK,
                                        MUX_SEL_IN    => MUX_SEL,
                                        MUX_SEL_OUT   => MUX_SEL,
                                        CLK1          => CLK1,
                                        GR_CLK        => GR_CLK,
                                        GR_CLR        => GR_CLR);

-- Active block register ----------------------------------------------------

        ACT_BLK_GND    : GND          port map(GROUND=> ACT_BLK_GND_O);

        ACT_BLK_REG    : REGN         generic map(N  => 2,
                                                  OS => 1)

                                      port map(D     => MUX_SEL,
                                               CLK   => CLK1,
                                               CLR   => ACT_BLK_GND_O,
                                               Q     => open);

-- Input register -----------------------------------------------------------

        IN_REG  : PREGN               generic map(N  => N_IN_OUT-1,
                                                  OS => 1,
                                                  IV => 0)

                                      port map(CLK   => IN_CLK,
                                               Q((N_IN_OUT-2) downto 0) =>
                                                  INPUTS((N_IN_OUT-1) downto 1));

-- Output register ----------------------------------------------------------

        OUT_REG : REGN                generic map(N  => N_IN_OUT-1,
                                                  OS => 1)

                                      port map(D((N_IN_OUT-2) downto 0) =>
                                                  OUTPUTS((N_IN_OUT-1) downto 1),
                                               CLK    => OUT_CLK,
                                               CLR    => OUT_CLR,
                                               Q      => open);

-- Logical function output flip-flop ----------------------------------------
```

```
        LF_CLR_GND      : GND           port map(GROUND        => LF_CLR_GND_O);

        LF_FF           : FDC           port map(D             => OUTPUTS(0),
                                                 C             => OUT_CLK,
                                                 CLR           => LF_CLR_GND_O,
                                                 Q             => LF_FF_O);

        LF_BUF          : BUF           port map(I             => LF_FF_O,
                                                 O             => INPUTS(0));

-- Current graph register ------------------------------------------

        CURR_GR_REG     : REG_MUXN      generic map(N          => N_G_LINES,
                                                    OS         => N_SP_CELLS)

                                        port map(CLK           => CURR_GR_CLK,
                                                 CLR           => CURR_GR_CLR,
                                                 SEL           => RET,
                                                 I0            => NEXT_GR,
                                                 I1            => PREV_GR,
                                                 Q             => CURR_GR);

-- Previous state register -----------------------------------------

        P_ST_CLR_GND: GND               port map(GROUND        => P_ST_CLR_GND_O);

        PREV_ST_REG     : REGN          generic map(N          => N_ST_LINES,
                                                    OS         => N_SP_CELLS)

                                        port map(D(1 downto 0) => STACK_OUT(7 downto 6),
                                                 CLK           => PREV_ST_CLK,
                                                 CLR           => P_ST_CLR_GND_O,
                                                 Q             => PREV_ST);

-- Stack -----------------------------------------------------------

        STACK_MEM       : STACK         port map(CLK           => CLK2,
                                                 N_STACK_RD    => N_STACK_RD,
                                                 STACK_WR      => STACK_WR,
                                                 STACK_OPAD_EN => STACK_OPAD_EN,
                                                 SP_INC        => SP_INC,
                                                 SP_DEC        => SP_DEC,
                                                 SP_RESET      => SP_RESET,
                                                 STACK_IN      => STACK_IN,
                                                 STACK_OUT     => STACK_OUT,
                                                 SP_LIMIT      => SP_LIMIT);

        STACK_IN(7 downto 6)                    <= CURR_ST(1 downto 0);
        STACK_IN((N_G_LINES-1) downto 0)        <= CURR_GR((N_G_LINES-1) downto 0);

        PREV_GR((N_G_LINES-1) downto 0)         <= STACK_OUT((N_G_LINES-1) downto 0);

end STRUCT;
```

Sync.vhd

```
--------------------------------------------------------------------------------
--
-- Synchronization circuit for the virtual control unit
--
--------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library XC6200LIB;
use XC6200LIB.GS_PACK.ALL;
use XC6200LIB.GATES_PACK.ALL;
use XC6200LIB.LAT_FF_PACK.ALL;


entity SYNC is

       port(CLK              : in    STD_LOGIC;
            RESET            : in    STD_LOGIC;
            CALL             : in    STD_LOGIC;
            FINISH           : in    STD_LOGIC;
            GR_OK            : in    STD_LOGIC;
            SP_LIMIT         : in    STD_LOGIC;
            GR_FAULT_INT     : out   STD_LOGIC;
            STACK_OV_INT     : out   STD_LOGIC;
            GR_CLK           : out   STD_LOGIC;
            GR_CLR           : out   STD_LOGIC;
            CURR_GR_CLK      : out   STD_LOGIC;
            CURR_GR_CLR      : out   STD_LOGIC;
            PREV_ST_CLK      : out   STD_LOGIC;
            STACK_WR         : out   STD_LOGIC;
            N_STACK_RD       : out   STD_LOGIC;
            STACK_OPAD_EN    : out   STD_LOGIC;
            SP_INC           : out   STD_LOGIC;
            SP_DEC           : out   STD_LOGIC;
            SP_RESET         : out   STD_LOGIC;
            OUT_CLK          : out   STD_LOGIC;
            OUT_CLR          : out   STD_LOGIC;
            GR_FAULT_FLAG    : out   STD_LOGIC;
            STACK_OV_FLAG    : out   STD_LOGIC;
            START            : out   STD_LOGIC;
            RET              : out   STD_LOGIC);

-- Entity attributes -----------------------------------------------------------

       attribute OVERLAP     of SYNC        : entity       is "YES";
       attribute RTMAX       of SYNC        : entity       is "X0Y4";
       attribute RESERVE     of SYNC        : entity       is "X0Y0,X63Y51";
       attribute RESERVE     of SYNC        : entity       is "X4Y53,X6Y63";
       attribute RESERVE     of SYNC        : entity       is "X60Y60,X63Y63";

end SYNC;


architecture STRUCT of SYNC is

-- Components definition -------------------------------------------------------

       component ORN is
              generic(N      : integer := 8);

              port(I         : in    STD_LOGIC_VECTOR((N-1) downto 0);
                   O         : out   STD_LOGIC);
       end component;

-- Internal signals ------------------------------------------------------------

       signal ON_I                   : STD_LOGIC_VECTOR(14 downto 0);
       signal ON_O                   : STD_LOGIC_VECTOR(14 downto 0);
```

```
        signal CN_COND                : STD_LOGIC_VECTOR(5 downto 0);
        signal CN_I                   : STD_LOGIC_VECTOR(5 downto 0);
        signal CN_O_0                 : STD_LOGIC_VECTOR(5 downto 0);
        signal CN_O_1                 : STD_LOGIC_VECTOR(5 downto 0);

        signal L_I_S                  : STD_LOGIC_VECTOR(5 downto 0);
        signal L_I_R                  : STD_LOGIC_VECTOR(5 downto 0);
        signal L_O_Q                  : STD_LOGIC_VECTOR(5 downto 0);
        signal L_O_NQ                 : STD_LOGIC_VECTOR(5 downto 0);

        signal GR_OK_BUF_O            : STD_LOGIC;
        signal FINISH_BUF_O           : STD_LOGIC;
        signal CALL_BUF_O             : STD_LOGIC;
        signal SP_LIMIT_BUF_O         : STD_LOGIC;

        signal GR_CLR_OR_O            : STD_LOGIC;
        signal CURR_GR_CLK_OR_O       : STD_LOGIC;
        signal OUT_CLK_OR_O           : STD_LOGIC;
        signal OUT_CLR_OR_O           : STD_LOGIC;

-- Architecture attributes ------------------------------------------------------

        attribute RLOC        of GR_OK_BUF          : label        is "X34Y52";
        attribute RLOC        of FINISH_BUF         : label        is "X32Y52";
        attribute RLOC        of CALL_BUF           : label        is "X33Y52";
        attribute RLOC        of SP_LIMIT_BUF       : label        is "X2Y52";

        attribute RLOC        of GR_FAULT_INT_BUF   : label        is "X59Y62";
        attribute RLOC        of STACK_OV_INT_BUF   : label        is "X59Y63";
        attribute RLOC        of GR_CLK_BUF         : label        is "X17Y63";
        attribute RLOC        of GR_CLR_BUF         : label        is "X19Y63";
        attribute RLOC        of CURR_GR_CLK_BUF    : label        is "X35Y63";
        attribute RLOC        of CURR_GR_CLR_BUF    : label        is "X35Y56";
        attribute RLOC        of PREV_ST_CLK_BUF    : label        is "X63Y52";
        attribute RLOC        of STACK_WR_BUF       : label        is "X14Y63";
        attribute RLOC        of SP_INC_BUF         : label        is "X3Y52";
        attribute RLOC        of SP_DEC_BUF         : label        is "X3Y63";
        attribute RLOC        of SP_RESET_BUF       : label        is "X0Y52";
        attribute RLOC        of OUT_CLK_BUF        : label        is "X33Y63";
        attribute RLOC        of OUT_CLR_BUF        : label        is "X33Y56";

        attribute RLOC        of GR_FAULT_FLAG_BUF  : label        is "X59Y60";
        attribute RLOC        of STACK_OV_FLAG_BUF  : label        is "X59Y61";
        attribute RLOC        of START_BUF          : label        is "X32Y63";
        attribute RLOC        of RET_BUF            : label        is "X34Y63";
        attribute RLOC        of N_STACK_RD_BUF     : label        is "X28Y63";
        attribute RLOC        of STACK_OPAD_EN_BUF  : label        is "X62Y52";

-- Implementation ---------------------------------------------------------------

begin

-- Nodes and components instantiation -------------------------------

        ONG_0_I : INIT_OP_NODE port map(I     => ON_I(0),
                                        CLK   => CLK,
                                        RESET => RESET,
                                        O     => ON_O(0));

        ONG     : for J in 14 downto 1 generate
        begin
        I       : OP_NODE      port map(I     => ON_I(J),
                                        CLK   => CLK,
                                        RESET => RESET,
                                        O     => ON_O(J));
        end generate;

        CNG     : for J in 5 downto 0 generate
        begin
        I       : COND_NODE    port map(I     => CN_I(J),
                                        COND  => CN_COND(J),
                                        O_0   => CN_O_0(J),
                                        O_1   => CN_O_1(J));
```

```
        end generate;

        LG      : for J in 5 downto 0 generate
        begin
        I       : SR_LATCH      port map(S      => L_I_S(J),
                                         R      => L_I_R(J),
                                         Q      => L_O_Q(J),
                                         NQ     => L_O_NQ(J));
        end generate;

-- Conditions ---------------------------------------------------

        GR_OK_BUF       : BUF           port map(I      => GR_OK,
                                                 O      => GR_OK_BUF_O);

        FINISH_BUF      : BUF           port map(I      => FINISH,
                                                 O      => FINISH_BUF_O);

        CALL_BUF        : BUF           port map(I      => CALL,
                                                 O      => CALL_BUF_O);

        SP_LIMIT_BUF: BUF               port map(I      => SP_LIMIT,
                                                 O      => SP_LIMIT_BUF_O);

        CN_COND(0)      <= GR_OK_BUF_O;
        CN_COND(1)      <= GR_OK_BUF_O;
        CN_COND(2)      <= FINISH_BUF_O;
        CN_COND(3)      <= CALL_BUF_O;
        CN_COND(4)      <= SP_LIMIT_BUF_O;
        CN_COND(5)      <= SP_LIMIT_BUF_O;

-- Connections between nodes ------------------------------------

        ON_I(0)         <= ON_O(13);
        ON_I(1)         <= ON_O(11);
        ON_I(2)         <= ON_O(1);
        ON_I(3)         <= CN_O_0(0);

        ON_4_OR :       ORN     generic map(N  => 2)
                                port map(I(0)  => ON_O(3),
                                         I(1)  => CN_O_0(1),
                                         O     => ON_I(4));

        ON_5_OR :       ORN     generic map(N  => 2)
                                port map(I(0)  => CN_O_1(0),
                                         I(1)  => CN_O_1(1),
                                         O     => ON_I(5));

        ON_6_OR :       ORN     generic map(N  => 2)
                                port map(I(0)  => ON_O(5),
                                         I(1)  => ON_O(7),
                                         O     => ON_I(6));

        ON_I(7)         <= CN_O_0(3);
        ON_I(8)         <= CN_O_1(3);
        ON_I(9)         <= ON_O(8);
        ON_I(10)        <= ON_O(9);
        ON_I(11)        <= CN_O_0(4);
        ON_I(12)        <= CN_O_1(4);
        ON_I(13)        <= ON_O(12);
        ON_I(14)        <= CN_O_1(5);

        CN_0_OR :       ORN     generic map(N  => 3)
                                port map(I(0)  => ON_O(0),
                                         I(1)  => ON_O(2),
                                         I(2)  => CN_O_0(5),
                                         O     => CN_I(0));

        CN_I(1)         <= ON_O(4);
        CN_I(2)         <= ON_O(6);
        CN_I(3)         <= CN_O_0(2);
        CN_I(4)         <= CN_O_1(2);
        CN_I(5)         <= ON_O(10);
```

```
        L_I_S(0)        <= ON_O(3);
        L_I_S(1)        <= ON_O(14);

        L_S_2_OR:       ORN     generic map(N  => 2)
                                port map(I(0)  => ON_O(0),
                                         I(1)  => ON_O(8),
                                         O     => L_I_S(2));

        L_I_S(3)        <= ON_O(11);
        L_I_S(4)        <= ON_O(11);
        L_I_S(5)        <= ON_O(8);

        L_R_0_OR:       ORN     generic map(N  => 2)
                                port map(I(0)  => ON_O(0),
                                         I(1)  => ON_O(5),
                                         O     => L_I_R(0));

        L_I_R(1)        <= ON_O(0);
        L_I_R(2)        <= ON_O(6);

        L_R_3_OR:       ORN     generic map(N  => 2)
                                port map(I(0)  => ON_O(0),
                                         I(1)  => ON_O(6),
                                         O     => L_I_R(3));

        L_R_4_OR:       ORN     generic map(N  => 2)
                                port map(I(0)  => ON_O(0),
                                         I(1)  => ON_O(2),
                                         O     => L_I_R(4));

        L_R_5_OR:       ORN     generic map(N  => 2)
                                port map(I(0)  => ON_O(0),
                                         I(1)  => ON_O(10),
                                         O     => L_I_R(5));

-- Normal outputs ------------------------------------------------

        GR_FAULT_INT_BUF        : BUF   port map(I     => ON_O(3),
                                                 O     => GR_FAULT_INT);

        STACK_OV_INT_BUF        : BUF   port map(I     => ON_O(14),
                                                 O     => STACK_OV_INT);

        GR_CLK_BUF              : BUF   port map(I     => ON_O(6),
                                                 O     => GR_CLK);

        GR_CLR_OR               : ORN   generic map(N  => 4)
                                        port map(I(0)  => ON_O(0),
                                                 I(1)  => ON_O(2),
                                                 I(2)  => ON_O(4),
                                                 I(3)  => ON_O(10),
                                                 O     => GR_CLR_OR_O);

        GR_CLR_BUF              : BUF   port map(I     => GR_CLR_OR_O,
                                                 O     => GR_CLR);

        CURR_GR_CLK_OR          : ORN   generic map(N  => 2)
                                        port map(I(0)  => ON_O(1),
                                                 I(1)  => ON_O(9),
                                                 O     => CURR_GR_CLK_OR_O);

        CURR_GR_CLK_BUF         : BUF   port map(I     => CURR_GR_CLK_OR_O,
                                                 O     => CURR_GR_CLK);

        CURR_GR_CLR_BUF         : BUF   port map(I     => ON_O(0),
                                                 O     => CURR_GR_CLR);

        PREV_ST_CLK_BUF         : BUF   port map(I     => ON_O(1),
                                                 O     => PREV_ST_CLK);

        STACK_WR_BUF            : BUF   port map(I     => ON_O(8),
                                                 O     => STACK_WR);

        SP_INC_BUF              : BUF   port map(I     => ON_O(10),
```

```
                                        O     => SP_INC);

        SP_DEC_BUF              : BUF   port map(I     => ON_O(11),
                                        O     => SP_DEC);

        SP_RESET_BUF            : BUF   port map(I     => ON_O(0),
                                        O     => SP_RESET);

        OUT_CLK_OR              : ORN   generic map(N  => 4)
                                        port map(I(0)  => ON_O(7),
                                                 I(1)  => ON_O(8),
                                                 I(2)  => ON_O(11),
                                                 I(3)  => ON_O(12),
                                                 O     => OUT_CLK_OR_O);

        OUT_CLK_BUF             : BUF   port map(I     => OUT_CLK_OR_O,
                                        O     => OUT_CLK);

        OUT_CLR_OR              : ORN   generic map(N  => 3)
                                        port map(I(0)  => ON_O(0),
                                                 I(1)  => ON_O(2),
                                                 I(2)  => ON_O(10),
                                                 O     => OUT_CLR_OR_O);

        OUT_CLR_BUF             : BUF   port map(I     => OUT_CLR_OR_O,
                                        O     => OUT_CLR);

-- Latched outputs ------------------------------------------------

        GR_FAULT_FLAG_BUF       : BUF   port map(I     => L_O_Q(0),
                                        O     => GR_FAULT_FLAG);

        STACK_OV_FLAG_BUF       : BUF   port map(I     => L_O_Q(1),
                                        O     => STACK_OV_FLAG);

        START_BUF               : BUF   port map(I     => L_O_Q(2),
                                        O     => START);

        RET_BUF                 : BUF   port map(I     => L_O_Q(3),
                                        O     => RET);

        N_STACK_RD_BUF          : BUF   port map(I     => L_O_NQ(4),
                                        O     => N_STACK_RD);

        STACK_OPAD_EN_BUF       : BUF   port map(I     => L_O_Q(5),
                                        O     => STACK_OPAD_EN);

end STRUCT;
```

Irq_Block.vhd

```
--------------------------------------------------------------------------------
--
-- IRQ block with mask and pooling registers
--
--------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library PRIMS;
use PRIMS.XC6000_COMPONENTS.ALL;

library XC6200LIB;
use XC6200LIB.REG_PACK.PREGN;
use XC6200LIB.REG_PACK.REGN;


entity IRQ_BLOCK is

        port(CLK              : in    STD_LOGIC;
             GR_FAULT_INT     : in    STD_LOGIC;
             STACK_OV_INT     : in    STD_LOGIC;
             GR_FAULT_FLAG    : in    STD_LOGIC;
             STACK_OV_FLAG    : in    STD_LOGIC);

-- Entity attributes -----------------------------------------------------------

        attribute OVERLAP     of IRQ_BLOCK   : entity       is "YES";
        attribute RTDEFER     of IRQ_BLOCK   : entity       is "";

end IRQ_BLOCK;


architecture STRUCT of IRQ_BLOCK is

-- Components declaration ------------------------------------------------------

        component PREGN is
                generic(N      : integer := 8;
                        OS     : integer := 1;
                        IV     : integer := 0);

                port(CLK       : in    STD_LOGIC;
                     Q         : out   STD_LOGIC_VECTOR((N-1) downto 0));
        end component;

        component REGN is
                generic(N      : integer := 8;
                        OS     : integer := 1);

                port(D         : in    STD_LOGIC_VECTOR((N-1) downto 0);
                     CLK       : in    STD_LOGIC;
                     CLR       : in    STD_LOGIC;
                     Q         : out   STD_LOGIC_VECTOR((N-1) downto 0));
        end component;

-- Entity internal signals -----------------------------------------------------

        signal MASK_REG_Q, AND_O      : STD_LOGIC_VECTOR(1 downto 0);

        signal GND_O, NOR_O, IRQ_OB_O : STD_LOGIC;

-- Architecture attributes -----------------------------------------------------

        attribute RLOC        of POOL_REG    : label        is "X60Y60";
        attribute RLOC        of MASK_REG    : label        is "X60Y62";

        attribute RLOC        of AND_0       : label        is "X61Y62";
        attribute RLOC        of AND_1       : label        is "X61Y63";
```

```
        attribute RLOC         of NOR_0       : label        is "X62Y63";

        attribute LOC          of IRQ_OPAD    : label        is "P120";

-- Implementation -----------------------------------------------------------

begin

        GND             : GND          port map(GROUND=> GND_O);

        POOL_REG        : REGN         generic map(N  => 2,
                                                   OS  => 1)

                                       port map(D(1)  => STACK_OV_FLAG,
                                                D(0)  => GR_FAULT_FLAG,
                                                CLK   => CLK,
                                                CLR   => GND_O,
                                                Q     => open);

        MASK_REG        : PREGN        generic map(N  => 2,
                                                   OS => 1,
                                                   IV => 0)

                                       port map(CLK   => CLK,
                                                Q     => MASK_REG_Q);

        AND_0           : AND2         port map(I0    => GR_FAULT_INT,
                                                I1    => MASK_REG_Q(0),
                                                O     => AND_O(0));

        AND_1           : AND2         port map(I0    => STACK_OV_INT,
                                                I1    => MASK_REG_Q(1),
                                                O     => AND_O(1));

        NOR_0           : NOR2         port map(I0    => AND_O(0),
                                                I1    => AND_O(1),
                                                O     => NOR_O);

        IRQ_OB          : OBUF         port map(I     => NOR_O,
                                                O     => IRQ_OB_O);

        IRQ_OPAD        : OPAD         port map(OPAD  => IRQ_OB_O);

end STRUCT;
```

VCU.vhd

```
--------------------------------------------------------------------------------
--
-- Virtual control unit template
--
--------------------------------------------------------------------------------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library PRIMS;
use PRIMS.XC6000_COMPONENTS.ALL;

library WORK;
use WORK.BLOCKS_IO;
use WORK.IRQ_BLOCK;
use WORK.SYNC;


entity VCU is

        generic(N_IN_OUT       : integer := 28;
                N_G_LINES      : integer := 6;
                N_ST_LINES     : integer := 2;
                N_SP_CELLS     : integer := 2);

-- Entity attributes -----------------------------------------------------------

        attribute RTMAX        of VCU  : entity       is "X0Y0";

end VCU;


architecture STRUCT of VCU is

-- Components declaration ------------------------------------------------------

        component BLOCKS_IO is
                generic(N_IN_OUT       : integer := 16;
                        N_G_LINES      : integer := 6;
                        N_ST_LINES     : integer := 2;
                        N_SP_CELLS     : integer := 2);

                port(CLK1              : in    STD_LOGIC;
                     CLK2              : in    STD_LOGIC;
                     START             : in    STD_LOGIC;
                     FINISH            : out   STD_LOGIC;
                     CALL              : out   STD_LOGIC;
                     RET               : in    STD_LOGIC;
                     GR_OK             : out   STD_LOGIC;
                     SP_LIMIT          : out   STD_LOGIC;
                     STACK_WR          : in    STD_LOGIC;
                     N_STACK_RD        : in    STD_LOGIC;
                     STACK_OPAD_EN     : in    STD_LOGIC;
                     SP_INC            : in    STD_LOGIC;
                     SP_DEC            : in    STD_LOGIC;
                     SP_RESET          : in    STD_LOGIC;
                     CURR_GR_CLK       : in    STD_LOGIC;
                     CURR_GR_CLR       : in    STD_LOGIC;
                     PREV_ST_CLK       : in    STD_LOGIC;
                     IN_CLK            : in    STD_LOGIC;
                     OUT_CLK           : in    STD_LOGIC;
                     OUT_CLR           : in    STD_LOGIC;
                     GR_CLK            : in    STD_LOGIC;
                     GR_CLR            : in    STD_LOGIC);
        end component;

-- Entity internal signals -----------------------------------------------------

        signal GCLK_IP_O, GCLK_IB_O           : STD_LOGIC;
```

```
        signal CLK_REG_Q, CLK1, CLK2          : STD_LOGIC;

        signal GCLR_IP_O, GCLR_IB_O, GCLR     : STD_LOGIC;
        signal GR_CLR, G_GR_CLR               : STD_LOGIC;
        signal GR_CLK, G_GR_CLK               : STD_LOGIC;

        signal GR_OK, SP_LIMIT                : STD_LOGIC;
        signal GR_FAULT_INT, STACK_OV_INT     : STD_LOGIC;
        signal CURR_GR_CLK, CURR_GR_CLR       : STD_LOGIC;
        signal PREV_ST_CLK                    : STD_LOGIC;
        signal STACK_WR, N_STACK_RD           : STD_LOGIC;
        signal STACK_OPAD_EN                  : STD_LOGIC;
        signal SP_INC, SP_DEC, SP_RESET       : STD_LOGIC;
        signal IN_CLK, OUT_CLK, OUT_CLR       : STD_LOGIC;
        signal GR_FAULT_FLAG, STACK_OV_FLAG   : STD_LOGIC;

        signal START, FINISH, CALL, RET       : STD_LOGIC;

-- Architecture attributes ----------------------------------------------------

        attribute GLOBAL      of GCLK_GB      : label  is "GCLK";
        attribute GLOBAL      of GCLR_GB      : label  is "GCLR";
        attribute GLOBAL      of GR_CLK_GB    : label  is "G1";
        attribute GLOBAL      of GR_CLR_GB    : label  is "G2";

        attribute RLOC        of CLK_REG      : label  is "X13Y1";

        attribute RLOC        of SYNC         : label  is "X0Y0";

        attribute RLOC        of IRQ          : label  is "X0Y0";

        attribute RLOC        of CELLS        : label  is "X0Y0";

-- Implementation -------------------------------------------------------------

begin

-- Clock components -----------------------------------------------

        GCLK_IP         : IPAD          port map(IPAD  => GCLK_IP_O);

        GCLK_IB         : IBUF          port map(I     => GCLK_IP_O,
                                                 O     => CLK1);

        IN_CLK  <= CLK1;

        CLK_REG         : RPFD          port map(C     => CLK1,
                                                 Q     => CLK_REG_Q);

        DBCLK_GB        : BUFGP         port map(I     => CLK_REG_Q,
                                                 O     => CLK2);

-- Clear components -----------------------------------------------

        GCLR_IP         : IPAD          port map(IPAD  => GCLR_IP_O);

        GCLR_IB         : IBUF          port map(I     => GCLR_IP_O,
                                                 O     => GCLR_IB_O);

        GCLR_GB         : BUFGP         port map(I     => GCLR_IB_O,
                                                 O     => GCLR);

-- Global 1 signal components for graph clock ---------------------

        GR_CLK_GB       : BUFGP         port map(I     => GR_CLK,
                                                 O     => G_GR_CLK);

-- Global 2 signal components for graph clear ---------------------

        GR_CLR_GB       : BUFGP         port map(I     => GR_CLR,
                                                 O     => G_GR_CLR);

-- Synchorization circuit -----------------------------------------
```

```
        SYNC    : SYNC          port map(CLK           => CLK2,
                                         RESET         => GCLR,
                                         CALL          => CALL,
                                         FINISH        => FINISH,
                                         GR_OK         => GR_OK,
                                         SP_LIMIT      => SP_LIMIT,
                                         GR_FAULT_INT  => GR_FAULT_INT,
                                         STACK_OV_INT  => STACK_OV_INT,
                                         GR_CLK        => GR_CLK,
                                         GR_CLR        => GR_CLR,
                                         CURR_GR_CLK   => CURR_GR_CLK,
                                         CURR_GR_CLR   => CURR_GR_CLR,
                                         PREV_ST_CLK   => PREV_ST_CLK,
                                         STACK_WR      => STACK_WR,
                                         N_STACK_RD    => N_STACK_RD,
                                         STACK_OPAD_EN => STACK_OPAD_EN,
                                         SP_INC        => SP_INC,
                                         SP_DEC        => SP_DEC,
                                         SP_RESET      => SP_RESET,
                                         OUT_CLK       => OUT_CLK,
                                         OUT_CLR       => OUT_CLR,
                                         GR_FAULT_FLAG => GR_FAULT_FLAG,
                                         STACK_OV_FLAG => STACK_OV_FLAG,
                                         START         => START,
                                         RET           => RET);

-- IRQ Activation circuit ------------------------------------------

        IRQ     : IRQ_BLOCK     port map(CLK           => CLK1,
                                         GR_FAULT_INT  => GR_FAULT_INT,
                                         STACK_OV_INT  => STACK_OV_INT,
                                         GR_FAULT_FLAG => GR_FAULT_FLAG,
                                         STACK_OV_FLAG => STACK_OV_FLAG);

-- Implementation cells --------------------------------------------

        CELLS   : BLOCKS_IO     generic map(N_IN_OUT   => N_IN_OUT,
                                            N_G_LINES  => N_G_LINES,
                                            N_ST_LINES => N_ST_LINES,
                                            N_SP_CELLS => N_SP_CELLS)

                                port map(CLK1          => CLK1,
                                         CLK2          => CLK2,
                                         START         => START,
                                         FINISH        => FINISH,
                                         CALL          => CALL,
                                         RET           => RET,
                                         GR_OK         => GR_OK,
                                         SP_LIMIT      => SP_LIMIT,
                                         STACK_WR      => STACK_WR,
                                         N_STACK_RD    => N_STACK_RD,
                                         STACK_OPAD_EN => STACK_OPAD_EN,
                                         SP_INC        => SP_INC,
                                         SP_DEC        => SP_DEC,
                                         SP_RESET      => SP_RESET,
                                         CURR_GR_CLK   => CURR_GR_CLK,
                                         CURR_GR_CLR   => CURR_GR_CLR,
                                         PREV_ST_CLK   => PREV_ST_CLK,
                                         IN_CLK        => IN_CLK,
                                         OUT_CLK       => OUT_CLK,
                                         OUT_CLR       => OUT_CLR,
                                         GR_CLK        => G_GR_CLK,
                                         GR_CLR        => G_GR_CLR);

end STRUCT;
```

Anexo V – Código Fonte do Controlador da Placa de Desenvolvimento FireFly™

Sumário

Neste anexo são apresentadas as listagens do código fonte do controlador (*device driver*) da placa de desenvolvimento *Firefly*™ utilizada neste trabalho para implementação e teste de circuitos numa FPGA XC6216 da Xilinx.

Tal como foi referido no capítulo 7, o módulo desenvolvido destina-se a ser utilizado nos sistemas operativos Microsoft Windows 98 e 2000.

O seu desenvolvimento foi baseado no modelo WDM (*Windows Driver Model*), o qual permite tirar partido das características *plug-n-play* existentes quer nos sistemas operativos acima referidos, quer na placa *FireFly*™.

Além disso, disponibiliza também as seguintes funcionalidades:

- Transferência de dados entre as memórias da placa e do computador hospedeiro;
- Mapeamento da memória da placa no espaço de endereçamento de um processo a executar no computador hospedeiro;
- Processamento de interrupções a partir de um processo de utilizador.

De notar que o termo memória utilizado acima refere-se tanto à SRAM existente na placa como à memória de configuração da FPGA XC6200.

Uma discussão aprofundada deste módulo está fora do âmbito desta dissertação, pelo que a quem estiver interessado recomenda-se a leitura de alguns livros dedicados a este assunto e publicados recentemente, dos quais se destacam [OneFol99, Cant99].

PciBoardIoCtrl.h

```
/*****************************************************************************

    PciBoardIoCtrl.h:  PCI board device I/O controls definition file

    Version 1.0       4/6/99

    Copyright 1999 University of Aveiro - Electronics and Telecom. Department
    All rights reserved.
    Developed by Arnaldo Oliveira - arnaldo@ua.pt

*****************************************************************************/

#ifndef __PCIBOARDIOCTRL_H
#define __PCIBOARDIOCTRL_H

////////////////////////////////////////////
/* Includes                               */

#ifdef DRIVER
       #include "devioctl.h"
#else
       #pragma warning(disable : 4201)
       #include "winioctl.h"
#endif

#ifndef CTL_CODE
  #pragma message("CTL_CODE undefined. Include winioctl.h or devioctl.h before this file")
#endif

////////////////////////////////////////////
/* Definitions                            */

#define FILE_DEVICE_PCIBOARD                0x0000C000

#define PCIBOARD_CTRLCODE_BASE              0x800

////////////////////////////////////////////

#define IOCTL_PCIBOARD_GETINFO        (ULONG)CTL_CODE(FILE_DEVICE_PCIBOARD,              \
                                              PCIBOARD_CTRLCODE_BASE + 0x000,            \
                                              METHOD_BUFFERED,                           \
                                              FILE_ANY_ACCESS)

#define IOCTL_PCIBOARD_GETPORTBASE    (ULONG)CTL_CODE(FILE_DEVICE_PCIBOARD,              \
                                              PCIBOARD_CTRLCODE_BASE + 0x010,            \
                                              METHOD_BUFFERED,                           \
                                              FILE_READ_ACCESS | FILE_WRITE_ACCESS)

#define IOCTL_PCIBOARD_MAPMEMORY      (ULONG)CTL_CODE(FILE_DEVICE_PCIBOARD,              \
                                              PCIBOARD_CTRLCODE_BASE + 0x011,            \
                                              METHOD_BUFFERED,                           \
                                              FILE_READ_ACCESS | FILE_WRITE_ACCESS)

#define IOCTL_PCIBOARD_UNMAPMEMORY    (ULONG)CTL_CODE(FILE_DEVICE_PCIBOARD,              \
                                              PCIBOARD_CTRLCODE_BASE + 0x012,            \
                                              METHOD_BUFFERED,                           \
                                              FILE_READ_ACCESS | FILE_WRITE_ACCESS)

#define IOCTL_PCIBOARD_ENABLEIRQ      (ULONG)CTL_CODE(FILE_DEVICE_PCIBOARD,              \
                                              PCIBOARD_CTRLCODE_BASE + 0x01A,            \
                                              METHOD_BUFFERED,                           \
                                              FILE_READ_ACCESS | FILE_WRITE_ACCESS)

#define IOCTL_PCIBOARD_DISABLEIRQ     (ULONG)CTL_CODE(FILE_DEVICE_PCIBOARD,              \
                                              PCIBOARD_CTRLCODE_BASE + 0x01B,            \
                                              METHOD_BUFFERED,                           \
                                              FILE_READ_ACCESS | FILE_WRITE_ACCESS)

#define IOCTL_PCIBOARD_GETIRQSTATUS   (ULONG)CTL_CODE(FILE_DEVICE_PCIBOARD,              \
                                              PCIBOARD_CTRLCODE_BASE + 0x01C,            \
```

```
                                        METHOD_BUFFERED,                        \
                                        FILE_READ_ACCESS | FILE_WRITE_ACCESS)

#define IOCTL_PCIBOARD_WRITEPORT      (ULONG)(ULONG)CTL_CODE(FILE_DEVICE_PCIBOARD,  \
                                        PCIBOARD_CTRLCODE_BASE + 0x020,         \
                                        METHOD_BUFFERED,                        \
                                        FILE_WRITE_ACCESS)

#define IOCTL_PCIBOARD_READPORT       (ULONG)CTL_CODE(FILE_DEVICE_PCIBOARD,         \
                                        PCIBOARD_CTRLCODE_BASE + 0x021,         \
                                        METHOD_BUFFERED,                        \
                                        FILE_READ_ACCESS)

#define IOCTL_PCIBOARD_WRITEMEMORY    (ULONG)CTL_CODE(FILE_DEVICE_PCIBOARD,         \
                                        PCIBOARD_CTRLCODE_BASE + 0x022,         \
                                        METHOD_BUFFERED,                        \
                                        FILE_WRITE_ACCESS)

#define IOCTL_PCIBOARD_READMEMORY     (ULONG)CTL_CODE(FILE_DEVICE_PCIBOARD,         \
                                        PCIBOARD_CTRLCODE_BASE + 0x023,         \
                                        METHOD_BUFFERED,                        \
                                        FILE_READ_ACCESS)

#define IOCTL_PCIBOARD_WRITEBUFFER    (ULONG)CTL_CODE(FILE_DEVICE_PCIBOARD,         \
                                        PCIBOARD_CTRLCODE_BASE + 0x024,         \
                                        METHOD_BUFFERED,                        \
                                        FILE_WRITE_ACCESS)

#define IOCTL_PCIBOARD_READBUFFER     (ULONG)CTL_CODE(FILE_DEVICE_PCIBOARD,         \
                                        PCIBOARD_CTRLCODE_BASE + 0x025,         \
                                        METHOD_BUFFERED,                        \
                                        FILE_READ_ACCESS)

#endif // __PCIBOARDIOCTRL_H
```

BoardIoTypes.h

```
/*****************************************************************************

    BoardIoTypes.h:           Board I/O types definition file

    Version 1.0               4/6/99

    Copyright 1999 University of Aveiro - Electronics and Telecom. Department
    All rights reserved.
    Developed by Arnaldo Oliveira - arnaldo@ua.pt

*****************************************************************************/

#ifndef __BOARDIOTYPES_H
#define __BOARDIOTYPES_H

////////////////////////////////////////////
/* Definitions                            */

#define NAME_STR_LENGTH       30
#define VERSION_STR_LENGTH    20
#define COPYRIGHT_STR_LENGTH  40

////////////////////////////////////////////

typedef struct SDriverInfo
{
       CHAR name[NAME_STR_LENGTH];
       CHAR version[VERSION_STR_LENGTH];
       CHAR copyright[COPYRIGHT_STR_LENGTH];
}TDriverInfo;

////////////////////////////////////////////

typedef ULONG TPortBase;

////////////////////////////////////////////

typedef PVOID TMemoryBase;

////////////////////////////////////////////

typedef struct SEnableIrqIn
{
       ULONG mask;
       HANDLE usrNotifyEvent;
}TEnableIrqIn;

////////////////////////////////////////////

typedef ULONG TIrqStatus;

////////////////////////////////////////////

typedef struct SWriteIn
{
       ULONG offset;
       ULONG data;
}TWriteIn;

////////////////////////////////////////////

typedef ULONG TReadIn;

typedef ULONG TReadOut;

////////////////////////////////////////////

typedef struct SWriteBufferIn
{
       ULONG offset;
```

```
        ULONG count;
        ULONG data[1];
}TWriteBufferIn;

//////////////////////////////////////////////

typedef struct SReadBufferIn
{
        ULONG offset;
        ULONG count;
}TReadBufferIn;

typedef ULONG TReadBufferOut[1];

#endif // __BOARDIOTYPES_H
```

FireflyDrv.h

```
/*****************************************************************************

    FireflyDrv.h:             Shared user mode and kernel mode header file
                              for the FireFly PCI board WDM device driver.

    Version 1.0               4/6/99

    Copyright 1999 University of Aveiro - Electronics and Telecom. Department
    All rights reserved.
    Developed by Arnaldo Oliveira - arnaldo@ua.pt

*****************************************************************************/

#ifndef __FIREFLYDRV_H
#define __FIREFLYDRV_H

//////////////////////////////////////////////
/* Includes                                 */

#include "BoardIoTypes.h"
#include "PciBoardIoCtrl.h"

//////////////////////////////////////////////
/* Definitions                              */

#ifndef FAR
#define FAR
#endif

//////////////////////////////////////////////

// {13246240-2261-11d3-B812-004F5600CE3C}
DEFINE_GUID(GUID_FIREFLY, 0x13246240, 0x2261, 0x11d3, 0xb8, 0x12, 0x0, 0x4f, 0x56, 0x0,
0xce, 0x3c);

//////////////////////////////////////////////

#define FF_IRQ_STATUS_REG     0x30
#define FF_IRQ_ACK_REG        0x30
#define FF_IRQ_MASK_REG       0x34

#define FF_IRQ_STATUS_MASK    0x0007
#define FF_IRQ_ACK_VALUE      0x0007
#define FF_IRQ_DISABLE_VALUE  0x0000

#endif // __FIREFLYDRV_H
```

FireflyMain.h

```
/*****************************************************************************

    FireflyMain.h:            main header file
                              for the FireFly PCI board WDM device driver.

    Version 1.0               4/6/99

    Copyright 1999 University of Aveiro - Electronics and Telecom. Department
    All rights reserved.
    Developed by Arnaldo Oliveira - arnaldo@ua.pt

*****************************************************************************/

#ifndef __FIREFLYMAIN_H
#define __FIREFLYMAIN_H

////////////////////////////////////////////
/* Includes                                */

#ifdef __cplusplus
extern "C"
{
#endif // __cplusplus

#include <wdm.h>

#ifdef __cplusplus
}
#endif // __cplusplus

#include "FireflyDrv.h"

////////////////////////////////////////////
/* Definitions                             */

#define CODE_SEG_INIT         code_seg("init")
#define CODE_SEG_PAGED        code_seg("page")
#define CODE_SEG_LOCKED       code_seg()

#define DATA_SEG_INIT         data_seg("init")
#define DATA_SEG_PAGED        data_seg("page")
#define DATA_SEG_LOCKED       data_seg()

////////////////////////////////////////////

#define ARRAY_SIZE(a)         (sizeof(a) / sizeof((a)[0]))

////////////////////////////////////////////

#define ALIGNED_ACCESS(address, dataSize)    ((address & (dataSize - 1)) == 0)

////////////////////////////////////////////

typedef struct tagDEVICE_EXTENSION
{
        PDEVICE_OBJECT        pFuncDevObj;
        PDEVICE_OBJECT        pLowerDevObj;
        PDEVICE_OBJECT        pPhysDevObj;

        UNICODE_STRING        interfaceName;

        BOOLEAN               started;
        BOOLEAN               enabled;
        BOOLEAN               removing;
        KEVENT                removeEvent;
        LONG                  usageCount;
        KSPIN_LOCK            spinLock;

        PVOID                 pMemoryBase;
        ULONG                 memoryLength;
```

```
        PMDL                    pMdl;
        PVOID                   pUsrMappedAddr;
        PFILE_OBJECT            pMappedFileObj;

        PVOID                   pPortBase;
        ULONG                   portLength;
        BOOLEAN                 mappedPort;

        PKINTERRUPT             pIrqObj;
        ULONG                   irqLevel;
        ULONG                   irqStatus;
        HANDLE                  usrNotifyEvent;
        PFILE_OBJECT            pEventFileObj;
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;

////////////////////////////////////////////
/* Variables exports                      */

extern TDriverInfo driverInfo;

////////////////////////////////////////////
/* Functions prototypes                   */

VOID DriverUnload(PDRIVER_OBJECT pDrvObj);

NTSTATUS AddDevice(PDRIVER_OBJECT pDrvObj, PDEVICE_OBJECT pPhysDevObj);

NTSTATUS DispatchCreate(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp);

NTSTATUS DispatchClose(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp);

NTSTATUS DispatchCleanup(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp);

VOID CleanupDeviceQueue(PKDEVICE_QUEUE pDevQueue, PFILE_OBJECT pFileObj);

NTSTATUS CompleteIrp(PIRP pIrp,  NTSTATUS status,  ULONG info);

NTSTATUS ForwardIrpAndWait(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp);

NTSTATUS OnCompleteIrp(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp, PKEVENT pEvent);

VOID OnCancelIrp(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp);

BOOLEAN EnableDevice(PDEVICE_EXTENSION pDevExt);

BOOLEAN DisableDevice(PDEVICE_EXTENSION pDevExt);

BOOLEAN LockDevice(PDEVICE_EXTENSION pDevExt);

BOOLEAN LockDevice(PDEVICE_OBJECT pFuncDevObj);

VOID UnlockDevice(PDEVICE_EXTENSION pDevExt);

VOID UnlockDevice(PDEVICE_OBJECT pFuncDevObj);

BOOLEAN IrqServRoutine(PKINTERRUPT pIrqObj, PDEVICE_EXTENSION pDevExt);

VOID DpcForIrqServRoutine(PKDPC pDpc, PDEVICE_OBJECT pFuncDevObj, PIRP pIrp,
                          PDEVICE_EXTENSION pDevExt);

#endif // __FIREFLYMAIN_H
```

FireflyMain.cpp

```
/*****************************************************************************

    FireflyMain.cpp:   main implementation file
                       for the FireFly PCI board WDM device driver.

    Version 1.0        4/6/99

    Copyright 1999 University of Aveiro - Electronics and Telecom. Department
    All rights reserved.
    Developed by Arnaldo Oliveira - arnaldo@ua.pt

*****************************************************************************/

#define INITGUID

//////////////////////////////////////////
/* Includes                             */
#include "FireflyMain.h"
#include "FireflyDevCtrl.h"
#include "FireflyIo.h"
#include "FireflyPnp.h"
#include "FireflyPower.h"
#include "FireflyRw.h"
#include "FireflySysCtrl.h"

//////////////////////////////////////////
/* Variables declaration                */

TDriverInfo driverInfo =
{
        "Firefly WDM Device Driver",
        "Version 1.0",
        "Copyright 1999 - Arnaldo Oliveira"
};

//////////////////////////////////////////
/* Functions implementation             */

#pragma CODE_SEG_INIT

extern "C" NTSTATUS DriverEntry(PDRIVER_OBJECT pDrvObj, PUNICODE_STRING pRegistryPath)
{
        pDrvObj->DriverUnload = DriverUnload;
        pDrvObj->DriverStartIo = DriverStartIo;

        pDrvObj->DriverExtension->AddDevice = AddDevice;

        pDrvObj->MajorFunction[IRP_MJ_CREATE] = DispatchCreate;
        pDrvObj->MajorFunction[IRP_MJ_CLOSE] = DispatchClose;

        pDrvObj->MajorFunction[IRP_MJ_CLEANUP] = DispatchCleanup;

        pDrvObj->MajorFunction[IRP_MJ_READ] = DispatchRead;
        pDrvObj->MajorFunction[IRP_MJ_WRITE] = DispatchWrite;

        pDrvObj->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DispatchDevIoCtrl;
        pDrvObj->MajorFunction[IRP_MJ_SYSTEM_CONTROL] = DispatchSysCtrl;

        pDrvObj->MajorFunction[IRP_MJ_PNP] = DispatchPnp;
        pDrvObj->MajorFunction[IRP_MJ_POWER] = DispatchPower;

        return STATUS_SUCCESS;
}

//////////////////////////////////////////

#pragma CODE_SEG_PAGED

VOID DriverUnload(PDRIVER_OBJECT pDrvObj)
{
```

```
	PAGED_CODE();
}

//////////////////////////////////////////

#pragma CODE_SEG_PAGED

NTSTATUS AddDevice(PDRIVER_OBJECT pDrvObj, PDEVICE_OBJECT pPhysDevObj)
{
	PAGED_CODE();

	PDEVICE_OBJECT pFuncDevObj;

	NTSTATUS status = IoCreateDevice(pDrvObj, sizeof(DEVICE_EXTENSION), NULL,
					FILE_DEVICE_PCIBOARD, 0, FALSE, &pFuncDevObj);
	if (!NT_SUCCESS(status))
	{
		return status;
	}

	PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)pFuncDevObj->DeviceExtension;
	pDevExt->pFuncDevObj = pFuncDevObj;
	pDevExt->pPhysDevObj = pPhysDevObj;

	pDevExt->started = FALSE;
	pDevExt->enabled = FALSE;
	pDevExt->removing = FALSE;
	pDevExt->usageCount = 0;

	KeInitializeEvent(&pDevExt->removeEvent, NotificationEvent, FALSE);
	KeInitializeSpinLock(&pDevExt->spinLock);

	pDevExt->pMdl = NULL;
	pDevExt->pUsrMappedAddr = NULL;
	pDevExt->pMappedFileObj = NULL;

	pDevExt->irqStatus = 0;
	pDevExt->usrNotifyEvent = NULL;
	pDevExt->pEventFileObj = NULL;

	status = IoRegisterDeviceInterface(pPhysDevObj, &GUID_FIREFLY, NULL,
					&pDevExt->interfaceName);
	if (!NT_SUCCESS(status))
	{
		IoDeleteDevice(pFuncDevObj);
		return status;
	}
	IoSetDeviceInterfaceState(&pDevExt->interfaceName, TRUE);

	pDevExt->pLowerDevObj = IoAttachDeviceToDeviceStack(pFuncDevObj, pPhysDevObj);

	IoInitializeDpcRequest(pFuncDevObj, DpcForIrqServRoutine);

	pFuncDevObj->Flags |= DO_POWER_PAGABLE;
	pFuncDevObj->Flags |= DO_BUFFERED_IO;
	pFuncDevObj->Flags &= ~DO_DEVICE_INITIALIZING;

	return STATUS_SUCCESS;
}

//////////////////////////////////////////

#pragma CODE_SEG_PAGED

NTSTATUS DispatchCreate(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp)
{
	PAGED_CODE();

	return CompleteIrp(pIrp, STATUS_SUCCESS, 0);
}

//////////////////////////////////////////

#pragma CODE_SEG_PAGED
```

```
NTSTATUS DispatchClose(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp)
{
	PAGED_CODE();

	return DispatchCleanup(pFuncDevObj, pIrp);
}

//////////////////////////////////////////

#pragma CODE_SEG_PAGED

NTSTATUS DispatchCleanup(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp)
{
	PAGED_CODE();

	PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)pFuncDevObj->DeviceExtension;
	PIO_STACK_LOCATION pStackLoc = IoGetCurrentIrpStackLocation(pIrp);

	CleanupDeviceQueue(&pFuncDevObj->DeviceQueue, pStackLoc->FileObject);
	UnmapMemory(pDevExt, pStackLoc->FileObject);
	DisableIrq(pDevExt, pStackLoc->FileObject);
	return CompleteIrp(pIrp, STATUS_SUCCESS, 0);
}

//////////////////////////////////////////

#pragma CODE_SEG_PAGED

VOID CleanupDeviceQueue(PKDEVICE_QUEUE pDevQueue, PFILE_OBJECT pFileObj)
{
	PAGED_CODE();

	KIRQL oldIrql;
	PIRP pCancelIrp;
	PIRP pRequeueIrp;
	PIO_STACK_LOCATION pStackLoc;
	LIST_ENTRY cancelList;
	LIST_ENTRY requeueList;
	PLIST_ENTRY pListHead;
	PKDEVICE_QUEUE_ENTRY pQueueEntry;

	InitializeListHead(&cancelList);
	InitializeListHead(&requeueList);
	IoAcquireCancelSpinLock(&oldIrql);

	if (IsListEmpty(&pDevQueue->DeviceListHead))
	{
		IoReleaseCancelSpinLock(oldIrql);
		return;
	}

	while ((pQueueEntry = KeRemoveDeviceQueue(pDevQueue)) != NULL)
	{
		pCancelIrp = CONTAINING_RECORD(pQueueEntry, IRP,
					Tail.Overlay.DeviceQueueEntry);
		pStackLoc = IoGetCurrentIrpStackLocation(pCancelIrp);

		if (pStackLoc->FileObject == pFileObj)
		{
			pCancelIrp->Cancel = TRUE;
			pCancelIrp->CancelIrql = oldIrql;
			pCancelIrp->CancelRoutine = NULL;
			InsertTailList(&cancelList, &pCancelIrp->Tail.Overlay.ListEntry);
		}
		else
			InsertTailList(&requeueList, &pCancelIrp->Tail.Overlay.ListEntry);
	}

	while (!IsListEmpty(&requeueList))
	{
		pListHead = RemoveHeadList(&requeueList);
		pRequeueIrp = CONTAINING_RECORD(pListHead, IRP, Tail.Overlay.ListEntry);
```

```
                if (!KeInsertDeviceQueue(pDevQueue,
                                        &pRequeueIrp->Tail.Overlay.DeviceQueueEntry))
                        KeInsertDeviceQueue(pDevQueue,
                                            &pRequeueIrp->Tail.Overlay.DeviceQueueEntry);
        }

        IoReleaseCancelSpinLock(oldIrql);

        while (!IsListEmpty(&cancelList))
        {
                pListHead = RemoveHeadList(&cancelList);
                pCancelIrp = CONTAINING_RECORD(pListHead, IRP, Tail.Overlay.ListEntry);
                CompleteIrp(pCancelIrp, STATUS_CANCELLED, 0);
        }
}

//////////////////////////////////////////////

#pragma CODE_SEG_LOCKED

NTSTATUS CompleteIrp(PIRP pIrp,  NTSTATUS status,  ULONG info)
{
        pIrp->IoStatus.Status = status;
        pIrp->IoStatus.Information = info;
        IoCompleteRequest(pIrp, IO_NO_INCREMENT);
        return status;
}

//////////////////////////////////////////////

#pragma CODE_SEG_PAGED

NTSTATUS ForwardIrpAndWait(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp)
{
        PAGED_CODE();

        KEVENT event;
        KeInitializeEvent(&event, NotificationEvent, FALSE);

        IoCopyCurrentIrpStackLocationToNext(pIrp);
        IoSetCompletionRoutine(pIrp, (PIO_COMPLETION_ROUTINE)OnCompleteIrp,
                               (PVOID)&event, TRUE, TRUE, TRUE);

        PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)pFuncDevObj->DeviceExtension;
        NTSTATUS status = IoCallDriver(pDevExt->pLowerDevObj, pIrp);
        if (status = STATUS_PENDING)
        {
                KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, NULL);
                status = pIrp->IoStatus.Status;
        }

        return status;
}

//////////////////////////////////////////////

#pragma CODE_SEG_LOCKED

NTSTATUS OnCompleteIrp(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp, PKEVENT pEvent)
{
        KeSetEvent(pEvent, 0, FALSE);
        return STATUS_MORE_PROCESSING_REQUIRED;
}

//////////////////////////////////////////////

#pragma CODE_SEG_LOCKED

VOID OnCancelIrp(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp)
{
        if (pFuncDevObj->CurrentIrp == pIrp)
        {
                IoReleaseCancelSpinLock(pIrp->CancelIrql);
                CompleteIrp(pIrp, STATUS_CANCELLED, 0);
```

```
                IoStartNextPacket(pFuncDevObj, TRUE);
        }
        else
        {
                KeRemoveEntryDeviceQueue(&pFuncDevObj->DeviceQueue,
                                         &pIrp->Tail.Overlay.DeviceQueueEntry);
                IoReleaseCancelSpinLock(pIrp->CancelIrql);
                CompleteIrp(pIrp, STATUS_CANCELLED, 0);
        }
        UnlockDevice(pFuncDevObj);
}

//////////////////////////////////////////////

#pragma CODE_SEG_LOCKED

BOOLEAN EnableDevice(PDEVICE_EXTENSION pDevExt)
{
        pDevExt->enabled = TRUE;
        return TRUE;
}

//////////////////////////////////////////////

#pragma CODE_SEG_LOCKED

BOOLEAN DisableDevice(PDEVICE_EXTENSION pDevExt)
{
        pDevExt->enabled = FALSE;
        DisableIrq(pDevExt, pDevExt->pEventFileObj);
        return TRUE;
}

//////////////////////////////////////////////

#pragma CODE_SEG_LOCKED

BOOLEAN LockDevice(PDEVICE_EXTENSION pDevExt)
{
        LONG devUsageCount = InterlockedIncrement(&pDevExt->usageCount);
        ASSERT((devUsageCount > 0) || (pDevExt->removing));

        if (pDevExt->removing)
        {
                if (InterlockedDecrement(&pDevExt->usageCount) == -1)
                        KeSetEvent(&pDevExt->removeEvent, 0, FALSE);
                return FALSE;
        }

        return TRUE;
}

//////////////////////////////////////////////

#pragma CODE_SEG_LOCKED

BOOLEAN LockDevice(PDEVICE_OBJECT pFuncDevObj)
{
        return LockDevice((PDEVICE_EXTENSION)pFuncDevObj->DeviceExtension);
}

//////////////////////////////////////////////

#pragma CODE_SEG_LOCKED

VOID UnlockDevice(PDEVICE_EXTENSION pDevExt)
{
        LONG devUsageCount = InterlockedDecrement(&pDevExt->usageCount);
        ASSERT(devUsageCount >= -1);

        if (devUsageCount == -1)
        {
                ASSERT(pDevExt->removing);
                KeSetEvent(&pDevExt->removeEvent, 0, FALSE);
```

```
        }
}

//////////////////////////////////////////////

#pragma CODE_SEG_LOCKED

VOID UnlockDevice(PDEVICE_OBJECT pFuncDevObj)
{
        UnlockDevice((PDEVICE_EXTENSION)pFuncDevObj->DeviceExtension);
}

//////////////////////////////////////////////

#pragma CODE_SEG_LOCKED

BOOLEAN IrqServRoutine(PKINTERRUPT pIrqObj, PDEVICE_EXTENSION pDevExt)
{
        pDevExt->irqStatus = READ_PORT_ULONG((PULONG)((PUCHAR)pDevExt->pPortBase +

FF_IRQ_STATUS_REG));

        if (pDevExt->irqStatus & FF_IRQ_STATUS_MASK)
        {
                WRITE_PORT_ULONG((PULONG)((PUCHAR)pDevExt->pPortBase + FF_IRQ_ACK_REG),
                                 FF_IRQ_ACK_VALUE);
                IoRequestDpc(pDevExt->pFuncDevObj, NULL, pDevExt);
                return TRUE;
        }
        else
                return FALSE;
}

//////////////////////////////////////////////

#pragma CODE_SEG_LOCKED

VOID DpcForIrqServRoutine(PKDPC pDpc, PDEVICE_OBJECT pFuncDevObj, PIRP pIrp,
                          PDEVICE_EXTENSION pDevExt)
{
        if (pDevExt->pEventFileObj)
        {
                ASSERT(pDevExt->usrNotifyEvent);
                KeSetEvent((PKEVENT)pDevExt->usrNotifyEvent, 0, FALSE);
        }
}
```

FireflyDevCtrl.h

```
/*****************************************************************************

    FireflyDevCtrl.h:  Device I/O controls and functions header file
                       for the FireFly PCI    board WDM device driver.

    Version 1.0        4/6/99

    Copyright 1999 University of Aveiro - Electronics and Telecom. Department
    All rights reserved.
    Developed by Arnaldo Oliveira - arnaldo@ua.pt

*****************************************************************************/

#ifndef __FIREFLYDEVCTRL_H
#define __FIREFLYDEVCTRL_H

////////////////////////////////////////////
/* Functions prototypes                    */

NTSTATUS DispatchDevIoCtrl(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp);

NTSTATUS MapMemory(PDEVICE_EXTENSION pDevExt, PFILE_OBJECT pFileObj,
                   PVOID* pUsrMappedAddr);

NTSTATUS UnmapMemory(PDEVICE_EXTENSION pDevExt, PFILE_OBJECT pFileObj);

NTSTATUS EnableIrq(PDEVICE_EXTENSION pDevExt, PFILE_OBJECT pFileObj, ULONG irqMask,
                   HANDLE usrNotifyEvent);

NTSTATUS DisableIrq(PDEVICE_EXTENSION pDevExt, PFILE_OBJECT pFileObj);

#endif // __FIREFLYDEVCTRL_H
```

FireflyDevCtrl.cpp

```
/*****************************************************************************

    FireflyDevCtrl.cpp:   Device I/O controls and functions implementation file
                          for the FireFly PCI board WDM device driver.

    Version 1.0           4/6/99

    Copyright 1999 University of Aveiro - Electronics and Telecom. Department
    All rights reserved.
    Developed by Arnaldo Oliveira - arnaldo@ua.pt

*****************************************************************************/

////////////////////////////////////////////
/* Includes                               */

#include "FireflyMain.h"
#include "FireflyDevCtrl.h"

////////////////////////////////////////////
/* Functions implementation               */

#pragma CODE_SEG_PAGED

NTSTATUS DispatchDevIoCtrl(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp)
{
        PAGED_CODE();

        if (!LockDevice(pFuncDevObj))
                return CompleteIrp(pIrp, STATUS_DELETE_PENDING, 0);

        PIO_STACK_LOCATION pStackLoc = IoGetCurrentIrpStackLocation(pIrp);
        ULONG ioCtrlCode = pStackLoc->Parameters.DeviceIoControl.IoControlCode;
        ULONG inLength = pStackLoc->Parameters.DeviceIoControl.InputBufferLength;
        ULONG expInLength;
        ULONG outLength = pStackLoc->Parameters.DeviceIoControl.OutputBufferLength;
        ULONG expOutLength;
        PFILE_OBJECT pFileObj = pStackLoc->FileObject;

        PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)pFuncDevObj->DeviceExtension;

        NTSTATUS status = STATUS_PENDING;
        ULONG info = 0;

        switch (ioCtrlCode)
        {
        case IOCTL_PCIBOARD_GETINFO:
        {
                expOutLength = sizeof(TDriverInfo);
                if (expOutLength < outLength)
                        status = STATUS_INVALID_PARAMETER;
                else
                {
                        TDriverInfo* pDriverInfo =
                                (TDriverInfo*)pIrp->AssociatedIrp.SystemBuffer;

                        RtlCopyMemory(pDriverInfo, &driverInfo, sizeof(TDriverInfo));
                        status = STATUS_SUCCESS;
                        info = sizeof(TDriverInfo);
                }
                break;
        }

        case IOCTL_PCIBOARD_GETPORTBASE:
        {
                expOutLength = sizeof(TPortBase);
                if (expOutLength < outLength)
                        status = STATUS_INVALID_PARAMETER;
                else
                {
```

```
                TPortBase* pPortBase = (TPortBase*)pIrp->AssociatedIrp.SystemBuffer;

                *pPortBase = (ULONG)pDevExt->pPortBase;
                status = STATUS_SUCCESS;
                info = sizeof(TPortBase);
        }
        break;
}

case IOCTL_PCIBOARD_MAPMEMORY:
{
        expOutLength = sizeof(TMemoryBase);
        if (expOutLength < outLength)
                status = STATUS_INVALID_PARAMETER;
        else
        {
                TMemoryBase* pMemoryBase =
                        (TMemoryBase*)pIrp->AssociatedIrp.SystemBuffer;

                status = MapMemory(pDevExt, pFileObj, pMemoryBase);
                if (NT_SUCCESS(status))
                        info = sizeof(TMemoryBase);
        }
        break;
}

case IOCTL_PCIBOARD_UNMAPMEMORY:
{
        status = UnmapMemory(pDevExt, pFileObj);
        break;
}

case IOCTL_PCIBOARD_ENABLEIRQ:
{
        TEnableIrqIn* pEnableIrqIn =
                (TEnableIrqIn*)pIrp->AssociatedIrp.SystemBuffer;

        expInLength = sizeof(TEnableIrqIn);
        if (expInLength < inLength)
                status = STATUS_INVALID_PARAMETER;
        else
                status = EnableIrq(pDevExt, pFileObj, pEnableIrqIn->mask,
                                pEnableIrqIn->usrNotifyEvent);
        break;
}

case IOCTL_PCIBOARD_DISABLEIRQ:
{
        status = DisableIrq(pDevExt, pFileObj);
        break;
}

case IOCTL_PCIBOARD_GETIRQSTATUS:
{
        expInLength = sizeof(TIrqStatus);
        if (expInLength < inLength)
                status = STATUS_INVALID_PARAMETER;
        else
        {
                TIrqStatus* pIrqStatus =
                        (TIrqStatus*)pIrp->AssociatedIrp.SystemBuffer;

                *pIrqStatus = pDevExt->irqStatus;
                pDevExt->irqStatus = 0;
                status = STATUS_SUCCESS;
                info = sizeof(TIrqStatus);
        }
        break;
}

case IOCTL_PCIBOARD_WRITEPORT:
{
        expInLength = sizeof(TWriteIn);
        if (expInLength < inLength)
```

```
                status = STATUS_INVALID_PARAMETER;
        else
        {
                TWriteIn* pWriteIn = (TWriteIn*)pIrp->AssociatedIrp.SystemBuffer;

                 if ((pWriteIn->offset + sizeof(pWriteIn->data) >=
                      pDevExt->portLength) ||
                     (!ALIGNED_ACCESS((ULONG)pDevExt->pPortBase + pWriteIn->offset,
                                      sizeof(pWriteIn->data))))
                        status = STATUS_ACCESS_VIOLATION;
        }
        break;
}

case IOCTL_PCIBOARD_READPORT:
{
        expInLength = sizeof(TReadIn);
        expOutLength = sizeof(TReadOut);
        if ((expInLength < inLength) ||
                (expOutLength < outLength))
                status = STATUS_INVALID_PARAMETER;
        else
        {
                TReadIn* pReadIn = (TReadIn*)pIrp->AssociatedIrp.SystemBuffer;

                if ((*pReadIn + sizeof(TReadOut) >= pDevExt->portLength) ||
                        (!ALIGNED_ACCESS((ULONG)pDevExt->pPortBase + (*pReadIn),
                                         sizeof(TReadOut))))
                        status = STATUS_ACCESS_VIOLATION;
        }
        break;
}

case IOCTL_PCIBOARD_WRITEMEMORY:
{
        expInLength = sizeof(TWriteIn);
        if (expInLength < inLength)
                status = STATUS_INVALID_PARAMETER;
        else
        {
                TWriteIn* pWriteIn = (TWriteIn*)pIrp->AssociatedIrp.SystemBuffer;

                 if ((pWriteIn->offset + sizeof(pWriteIn->data) >=
                     pDevExt->memoryLength) ||
                     (!ALIGNED_ACCESS((ULONG)pDevExt->pMemoryBase + pWriteIn->offset,
                                      sizeof(pWriteIn->data))))
                        status = STATUS_ACCESS_VIOLATION;
        }
        break;
}

case IOCTL_PCIBOARD_READMEMORY:
{
        expInLength = sizeof(TReadIn);
        expOutLength = sizeof(TReadOut);
        if ((expInLength < inLength) ||
                (expOutLength < outLength))
                status = STATUS_INVALID_PARAMETER;
        else
        {
                TReadIn* pReadIn = (TReadIn*)pIrp->AssociatedIrp.SystemBuffer;

                if ((*pReadIn + sizeof(TReadOut) >= pDevExt->memoryLength) ||
                     (!ALIGNED_ACCESS((ULONG)pDevExt->pMemoryBase + (*pReadIn),
                                      sizeof(TReadOut))))
                        status = STATUS_ACCESS_VIOLATION;
        }
        break;
}

case IOCTL_PCIBOARD_WRITEBUFFER:
{
        expInLength = sizeof(TWriteBufferIn);
        if (expInLength < inLength)
```

```
                        status = STATUS_INVALID_PARAMETER;
                }
                else
                {
                        TWriteBufferIn* pWriteBufferIn =
                                (TWriteBufferIn*)pIrp->AssociatedIrp.SystemBuffer;

                        if (pWriteBufferIn->count == 0)
                                status = STATUS_SUCCESS;
                        else
                        {
                                expInLength = sizeof(TWriteBufferIn) +
                                                ((pWriteBufferIn->count - 1) *
                                                  sizeof(pWriteBufferIn->data[0]));
                                if (expInLength < inLength)
                                        status = STATUS_INVALID_PARAMETER;
                                else if ((pWriteBufferIn->offset + (pWriteBufferIn->count *
                                         sizeof(pWriteBufferIn->data[0])) >=
                                                pDevExt->memoryLength) ||
                                                (!ALIGNED_ACCESS((ULONG)pDevExt->pMemoryBase
                                                + pWriteBufferIn->offset,
                                                sizeof(pWriteBufferIn->data[0]))))
                                        status = STATUS_ACCESS_VIOLATION;
                        }
                }
                break;
        }

        case IOCTL_PCIBOARD_READBUFFER:
        {
                expInLength = sizeof(TReadBufferIn);
                if (expInLength < inLength)
                        status = STATUS_INVALID_PARAMETER;
                else
                {
                        TReadBufferIn* pReadBufferIn =
                                (TReadBufferIn*)pIrp->AssociatedIrp.SystemBuffer;

                        if (pReadBufferIn->count == 0)
                                status = STATUS_SUCCESS;
                        else
                        {
                                expOutLength = pReadBufferIn->count * sizeof(TReadBufferOut);
                                if (expOutLength < outLength)
                                        status = STATUS_INVALID_PARAMETER;
                                else if ((pReadBufferIn->offset + (pReadBufferIn->count *
                                         sizeof(TReadBufferOut)) >= pDevExt->memoryLength) ||
                                                (!ALIGNED_ACCESS((ULONG)pDevExt->pMemoryBase +
                                                 pReadBufferIn->offset,
                                                 sizeof(TReadBufferOut))))
                                        status = STATUS_ACCESS_VIOLATION;
                        }
                }
                break;
        }

        default:
                status = STATUS_INVALID_DEVICE_REQUEST;
                break;
        }

        if (status == STATUS_PENDING)
        {
                IoMarkIrpPending(pIrp);
                IoStartPacket(pFuncDevObj, pIrp, NULL, OnCancelIrp);
        }
        else
        {
                CompleteIrp(pIrp, status, info);
                UnlockDevice(pFuncDevObj);
        }

        return status;
}
```

```
//////////////////////////////////////////////

#pragma CODE_SEG_PAGED

NTSTATUS MapMemory(PDEVICE_EXTENSION pDevExt, PFILE_OBJECT pFileObj,
                                   PVOID* pUsrMappedAddr)
{
        PAGED_CODE();

        KIRQL oldIrql;

        KeAcquireSpinLock(&pDevExt->spinLock, &oldIrql);
        if (!pDevExt->pMappedFileObj)
                pDevExt->pMappedFileObj = pFileObj;
        KeReleaseSpinLock(&pDevExt->spinLock, oldIrql);
        if (pDevExt->pMappedFileObj != pFileObj)
                return STATUS_ACCESS_DENIED;

        ASSERT(!pDevExt->pUsrMappedAddr);
        ASSERT(!pDevExt->pMdl);

        pDevExt->pMdl = IoAllocateMdl(pDevExt->pMemoryBase, pDevExt->memoryLength, FALSE,
                                      FALSE, NULL);
        if (!pDevExt->pMdl)
        {
                pDevExt->pMappedFileObj = NULL;
                return STATUS_INSUFFICIENT_RESOURCES;
        }

        MmBuildMdlForNonPagedPool(pDevExt->pMdl);

        *pUsrMappedAddr = MmMapLockedPages(pDevExt->pMdl, UserMode);
        pDevExt->pUsrMappedAddr = *pUsrMappedAddr;

        return STATUS_SUCCESS;
}

//////////////////////////////////////////////

#pragma CODE_SEG_PAGED

NTSTATUS UnmapMemory(PDEVICE_EXTENSION pDevExt, PFILE_OBJECT pFileObj)
{
        PAGED_CODE();

        if ((pDevExt->pMappedFileObj) && (pDevExt->pMappedFileObj == pFileObj))
        {
                ASSERT(pDevExt->pUsrMappedAddr);
                ASSERT(pDevExt->pMdl);

                MmUnmapLockedPages(pDevExt->pUsrMappedAddr, pDevExt->pMdl);
                IoFreeMdl(pDevExt->pMdl);

                pDevExt->pMdl = NULL;
                pDevExt->pUsrMappedAddr = NULL;
                pDevExt->pMappedFileObj = NULL;
                return STATUS_SUCCESS;
        }
        else
                return STATUS_ACCESS_DENIED;
}

//////////////////////////////////////////////

#pragma CODE_SEG_LOCKED

NTSTATUS EnableIrq(PDEVICE_EXTENSION pDevExt, PFILE_OBJECT pFileObj, ULONG irqMask,
                   HANDLE usrNotifyEvent)
{
        NTSTATUS status;

        if (usrNotifyEvent != NULL)
        {
                KIRQL oldIrql;
```

```
                KeAcquireSpinLock(&pDevExt->spinLock, &oldIrql);
                if (!pDevExt->pEventFileObj)
                        pDevExt->pEventFileObj = pFileObj;
                KeReleaseSpinLock(&pDevExt->spinLock, oldIrql);
                if (pDevExt->pEventFileObj != pFileObj)
                        return STATUS_ACCESS_DENIED;

                ASSERT(!pDevExt->usrNotifyEvent);

                status = ObReferenceObjectByHandle(usrNotifyEvent, SYNCHRONIZE, NULL,
                                                   KernelMode, &pDevExt->usrNotifyEvent,
                                                   NULL);
                if (!NT_SUCCESS(status))
                {
                        pDevExt->usrNotifyEvent = NULL;
                        pDevExt->pEventFileObj = NULL;
                        return status;
                }

                WRITE_PORT_ULONG((PULONG)((PUCHAR)pDevExt->pPortBase + FF_IRQ_MASK_REG),
                                 irqMask);

                return STATUS_SUCCESS;
        }
        else
                return STATUS_INVALID_PARAMETER;
}

//////////////////////////////////////////////

#pragma CODE_SEG_LOCKED

NTSTATUS DisableIrq(PDEVICE_EXTENSION pDevExt, PFILE_OBJECT pFileObj)
{
        NTSTATUS status;

        if ((pDevExt->pEventFileObj) && (pDevExt->pEventFileObj == pFileObj))
        {
                ASSERT(pDevExt->usrNotifyEvent);

                WRITE_PORT_ULONG((PULONG)((PUCHAR)pDevExt->pPortBase + FF_IRQ_MASK_REG),
                                 FF_IRQ_DISABLE_VALUE);

                ObDereferenceObject(pDevExt->usrNotifyEvent);
                pDevExt->usrNotifyEvent = NULL;
                pDevExt->pEventFileObj = NULL;
                return STATUS_SUCCESS;
        }
        else
                return STATUS_ACCESS_DENIED;
}
```

FireflyRW.h

```
/*****************************************************************************

    FireflyRw.h:              Read/Write functions header file
                              for the FireFly PCI    board WDM device driver.

    Version 1.0               4/6/99

    Copyright 1999 University of Aveiro - Electronics and Telecom. Department
    All rights reserved.
    Developed by Arnaldo Oliveira - arnaldo@ua.pt

*****************************************************************************/

#ifndef __FIREFLYRW_H
#define __FIREFLYRW_H

////////////////////////////////////////////
/* Functions prototypes                    */

NTSTATUS DispatchWrite(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp);

NTSTATUS DispatchRead(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp);

#endif // __FIREFLYRW_H
```

FireflyRW.cpp

```
/*****************************************************************************

    FireflyRw.cpp:            Read/Write functions implementation file
                              for the FireFly PCI    board WDM device driver.

    Version 1.0               4/6/99

    Copyright 1999 University of Aveiro - Electronics and Telecom. Department
    All rights reserved.
    Developed by Arnaldo Oliveira - arnaldo@ua.pt

*****************************************************************************/

////////////////////////////////////////////
/* Includes                                */

#include "FireflyMain.h"
#include "FireflyRw.h"

////////////////////////////////////////////
/* Functions implementation                */

#pragma CODE_SEG_PAGED

NTSTATUS DispatchWrite(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp)
{
        PAGED_CODE();

        PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)pFuncDevObj->DeviceExtension;

        if (!LockDevice(pDevExt))
                return CompleteIrp(pIrp, STATUS_DELETE_PENDING, 0);

        PIO_STACK_LOCATION pStackLoc = IoGetCurrentIrpStackLocation(pIrp);
        if (pStackLoc->Parameters.Write.Length == 0)
        {
```

```
                UnlockDevice(pDevExt);
                return CompleteIrp(pIrp, STATUS_SUCCESS, 0);
        }
        if (pStackLoc->Parameters.Write.Length > pDevExt->memoryLength)
        {
                UnlockDevice(pDevExt);
                return CompleteIrp(pIrp, STATUS_ACCESS_VIOLATION, 0);
        }

        IoMarkIrpPending(pIrp);
        IoStartPacket(pFuncDevObj, pIrp, NULL, OnCancelIrp);
        return STATUS_PENDING;
}

//////////////////////////////////////////

#pragma CODE_SEG_PAGED

NTSTATUS DispatchRead(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp)
{
        PAGED_CODE();

        PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)pFuncDevObj->DeviceExtension;

        if (!LockDevice(pDevExt))
                return CompleteIrp(pIrp, STATUS_DELETE_PENDING, 0);

        PIO_STACK_LOCATION pStackLoc = IoGetCurrentIrpStackLocation(pIrp);
        if (pStackLoc->Parameters.Read.Length == 0)
        {
                UnlockDevice(pDevExt);
                return CompleteIrp(pIrp, STATUS_SUCCESS, 0);
        }
        if (pStackLoc->Parameters.Read.Length > pDevExt->memoryLength)
        {
                UnlockDevice(pDevExt);
                return CompleteIrp(pIrp, STATUS_ACCESS_VIOLATION, 0);
        }

        IoMarkIrpPending(pIrp);
        IoStartPacket(pFuncDevObj, pIrp, NULL, OnCancelIrp);
        return STATUS_PENDING;
}
```

FireflyIo.h

```
/*****************************************************************************

    FireflyIo.h:              I/O functions header file
                              for the FireFly PCI    board WDM device driver.

    Version 1.0               4/6/99

    Copyright 1999 University of Aveiro - Electronics and Telecom. Department
    All rights reserved.
    Developed by Arnaldo Oliveira - arnaldo@ua.pt

*****************************************************************************/

#ifndef __FIREFLYIO_H
#define __FIREFLYIO_H

////////////////////////////////////////////
/* Functions prototypes                   */

VOID DriverStartIo(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp);

#endif // __FIREFLYIO_H
```

FireflyIo.cpp

```
/*****************************************************************************

    FireflyIo.cpp:            I/O functions implementation file
                              for the FireFly PCI board WDM device driver.

    Version 1.0               4/6/99

    Copyright 1999 University of Aveiro - Electronics and Telecom. Department
    All rights reserved.
    Developed by Arnaldo Oliveira - arnaldo@ua.pt

*****************************************************************************/

////////////////////////////////////////////
/* Includes                               */

#include "FireflyMain.h"
#include "FireflyIo.h"

////////////////////////////////////////////
/* Functions implementation               */

#pragma CODE_SEG_LOCKED

VOID DriverStartIo(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp)
{
        KIRQL oldIrql;

        IoAcquireCancelSpinLock(&oldIrql);
        if ((pIrp != pFuncDevObj->CurrentIrp) || (pIrp->Cancel))
        {
                IoReleaseCancelSpinLock(oldIrql);
                return;
        }
        else
        {
                ASSERT(pIrp == pFuncDevObj->CurrentIrp);
                IoSetCancelRoutine(pIrp, NULL);
```

```
        IoReleaseCancelSpinLock(oldIrql);
}

PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)pFuncDevObj->DeviceExtension;
PIO_STACK_LOCATION pStackLoc = IoGetCurrentIrpStackLocation(pIrp);
NTSTATUS status = STATUS_SUCCESS;
ULONG info = 0;

switch(pStackLoc->MajorFunction)
{
        case IRP_MJ_WRITE:
        {
                PUCHAR pWriteBuffer = (PUCHAR)pIrp->AssociatedIrp.SystemBuffer;
                ULONG writeLength = pStackLoc->Parameters.Write.Length;

                WRITE_REGISTER_BUFFER_UCHAR((PUCHAR)pDevExt->pMemoryBase,
                                            pWriteBuffer,
                                            writeLength);
                info = writeLength;
                break;
        }
        case IRP_MJ_READ:
        {
                PUCHAR pReadBuffer = (PUCHAR)pIrp->AssociatedIrp.SystemBuffer;
                ULONG readLength = pStackLoc->Parameters.Read.Length;

                READ_REGISTER_BUFFER_UCHAR((PUCHAR)pDevExt->pMemoryBase, pReadBuffer,
                                           readLength);
                info = readLength;
                break;
        }
        case IRP_MJ_DEVICE_CONTROL:
        {
                ULONG ioCtrlCode =
                        pStackLoc->Parameters.DeviceIoControl.IoControlCode;
                ULONG inLength =
                        pStackLoc->Parameters.DeviceIoControl.InputBufferLength;
                ULONG outLength =
                        pStackLoc->Parameters.DeviceIoControl.OutputBufferLength;

                switch(ioCtrlCode)
                {
                        case IOCTL_PCIBOARD_WRITEPORT:
                        {
                                TWriteIn* pWriteIn =
                                        (TWriteIn*)pIrp->AssociatedIrp.SystemBuffer;

                                WRITE_PORT_ULONG((PULONG)((PUCHAR)pDevExt->pPortBase +
                                                          pWriteIn->offset),
                                                 pWriteIn->data);
                                break;
                        }
                        case IOCTL_PCIBOARD_READPORT:
                        {
                                TReadIn* pReadIn =
                                        (TReadIn*)pIrp->AssociatedIrp.SystemBuffer;
                                TReadOut* pReadOut =
                                        (TReadOut*)pIrp->AssociatedIrp.SystemBuffer;

                                *pReadOut =
                                        READ_PORT_ULONG((PULONG)
                                                        (PUCHAR)pDevExt->pPortBase +
                                                        (*pReadIn)));
                                info = sizeof(TReadOut);
                                break;
                        }
                        case IOCTL_PCIBOARD_WRITEMEMORY:
                        {
                                TWriteIn* pWriteIn =
                                        (TWriteIn*)pIrp->AssociatedIrp.SystemBuffer;

                                WRITE_REGISTER_ULONG((PULONG)
                                                     (PUCHAR)pDevExt->pMemoryBase +
                                                     pWriteIn->offset),
```

```
                                                pWriteIn->data);
                                        break;
                                }
                                case IOCTL_PCIBOARD_READMEMORY:
                                {
                                        TReadIn* pReadIn =
                                                (TReadIn*)pIrp->AssociatedIrp.SystemBuffer;
                                        TReadOut* pReadOut =
                                                (TReadOut*)pIrp->AssociatedIrp.SystemBuffer;

                                        *pReadOut =
                                                READ_REGISTER_ULONG((PULONG)((PUCHAR)
                                                                pDevExt->pMemoryBase +
                                                                (*pReadIn)));
                                        info = sizeof(TReadOut);
                                        break;
                                }
                                case IOCTL_PCIBOARD_WRITEBUFFER:
                                {
                                        TWriteBufferIn* pWriteBufferIn =
                                        (TWriteBufferIn*)pIrp->AssociatedIrp.SystemBuffer;

                                        WRITE_REGISTER_BUFFER_ULONG((PULONG)
                                              ((PUCHAR)pDevExt->pMemoryBase +
                                              pWriteBufferIn->offset),
                                              pWriteBufferIn->data, pWriteBufferIn->count);
                                        break;
                                }
                                case IOCTL_PCIBOARD_READBUFFER:
                                {
                                        TReadBufferIn* pReadBufferIn =
                                        (TReadBufferIn*)pIrp->AssociatedIrp.SystemBuffer;
                                        TReadBufferOut* pReadBufferOut =
                                        (TReadBufferOut*)pIrp->AssociatedIrp.SystemBuffer;
                                        ULONG readLength = pReadBufferIn->count;

                                        READ_REGISTER_BUFFER_ULONG((PULONG)
                                             ((PUCHAR)pDevExt->pMemoryBase +
                                             pReadBufferIn->offset),
                                             *pReadBufferOut, readLength);
                                        info = readLength * sizeof(TReadBufferOut);
                                        break;
                                }
                                default:
                                {
                                        status = STATUS_NOT_SUPPORTED;
                                        break;
                                }
                        }
                        break;
                }
                default:
                {
                        status = STATUS_NOT_SUPPORTED;
                        break;
                }
        }

        CompleteIrp(pIrp, status, info);
        IoStartNextPacket(pFuncDevObj, TRUE);
        UnlockDevice(pDevExt);
}
```

FireflyPnp.h

```
/*****************************************************************************

    FireflyPnp.h:             Plug and play functions header file
                              for the FireFly PCI board WDM device driver.

    Version 1.0               4/6/99

    Copyright 1999 University of Aveiro - Electronics and Telecom. Department
    All rights reserved.
    Developed by Arnaldo Oliveira - arnaldo@ua.pt

*****************************************************************************/

#ifndef __FIREFLYPNP_H
#define __FIREFLYPNP_H

////////////////////////////////////////////
/* Functions prototypes                    */

NTSTATUS DispatchPnp(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp);

NTSTATUS PnpDefaultMinorHandler(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp);

NTSTATUS PnpStartDeviceMinorHandler(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp);

NTSTATUS PnpStopDeviceMinorHandler(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp);

NTSTATUS PnpRemoveDeviceMinorHandler(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp);

NTSTATUS StartDevice(PDEVICE_OBJECT pFuncDevObj, PCM_PARTIAL_RESOURCE_LIST pRawList,

PCM_PARTIAL_RESOURCE_LIST pTransList);

VOID StopDevice(PDEVICE_OBJECT pFuncDevObj);

VOID RemoveDevice(PDEVICE_OBJECT pFuncDevObj);

#endif // __FIREFLYPNP_H
```

FireflyPnp.cpp

```
/*****************************************************************************

    FireflyPnp.cpp:           Plug and play functions implementation file
                              for the FireFly PCI board WDM device driver.

    Version 1.0               4/6/99

    Copyright 1999 University of Aveiro - Electronics and Telecom. Department
    All rights reserved.
    Developed by Arnaldo Oliveira - arnaldo@ua.pt

*****************************************************************************/

////////////////////////////////////////////
/* Includes                                */

#include "FireflyMain.h"
#include "FireflyPnp.h"

////////////////////////////////////////////
/* Functions implementation                */

#pragma CODE_SEG_PAGED

NTSTATUS DispatchPnp(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp)
{
        PAGED_CODE();

        if (!LockDevice(pFuncDevObj))
                return CompleteIrp(pIrp, STATUS_DELETE_PENDING, 0);

        PIO_STACK_LOCATION pStackLoc = IoGetCurrentIrpStackLocation(pIrp);
        ASSERT(pStackLoc->MajorFunction == IRP_MJ_PNP);

        static NTSTATUS (*PnpMinorHandlers[])(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp) =
        {
                PnpStartDeviceMinorHandler,   // IRP_MN_START_DEVICE
                PnpDefaultMinorHandler,       // IRP_MN_QUERY_REMOVE_DEVICE
                PnpRemoveDeviceMinorHandler,  // IRP_MN_REMOVE_DEVICE
                PnpDefaultMinorHandler,       // IRP_MN_CANCEL_REMOVE_DEVICE
                PnpStopDeviceMinorHandler,    // IRP_MN_STOP_DEVICE
                PnpDefaultMinorHandler,       // IRP_MN_QUERY_STOP_DEVICE
                PnpDefaultMinorHandler,       // IRP_MN_CANCEL_STOP_DEVICE
                PnpDefaultMinorHandler,       // IRP_MN_QUERY_DEVICE_RELATIONS
                PnpDefaultMinorHandler,       // IRP_MN_QUERY_INTERFACE
                PnpDefaultMinorHandler,       // IRP_MN_QUERY_CAPABILITIES
                PnpDefaultMinorHandler,       // IRP_MN_QUERY_RESOURCES
                PnpDefaultMinorHandler,       // IRP_MN_QUERY_RESOURCE_REQUIREMENTS
                PnpDefaultMinorHandler,       // IRP_MN_QUERY_DEVICE_TEXT
                PnpDefaultMinorHandler,       // IRP_MN_FILTER_RESOURCE_REQUIREMENTS
                PnpDefaultMinorHandler,       //
                PnpDefaultMinorHandler,       // IRP_MN_READ_CONFIG
                PnpDefaultMinorHandler,       // IRP_MN_WRITE_CONFIG
                PnpDefaultMinorHandler,       // IRP_MN_EJECT
                PnpDefaultMinorHandler,       // IRP_MN_SET_LOCK
                PnpDefaultMinorHandler,       // IRP_MN_QUERY_ID
                PnpDefaultMinorHandler,       // IRP_MN_QUERY_PNP_DEVICE_STATE
                PnpDefaultMinorHandler,       // IRP_MN_QUERY_BUS_INFORMATION
                PnpDefaultMinorHandler,       // IRP_MN_DEVICE_USAGE_NOTIFICATION
                PnpDefaultMinorHandler,       // IRP_MN_SURPRISE_REMOVAL
        };

        ULONG minorFunc = pStackLoc->MinorFunction;
        NTSTATUS status;
        if (minorFunc >= ARRAY_SIZE(PnpMinorHandlers))
        {
                status = PnpDefaultMinorHandler(pFuncDevObj, pIrp);
        }
        else
        {
```

```
                status = (*PnpMinorHandlers[minorFunc])(pFuncDevObj, pIrp);
        }

        if (minorFunc != IRP_MN_REMOVE_DEVICE)
                UnlockDevice(pFuncDevObj);

        return status;
}

//////////////////////////////////////////

#pragma CODE_SEG_PAGED

NTSTATUS PnpDefaultMinorHandler(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp)
{
        PAGED_CODE();

        IoSkipCurrentIrpStackLocation(pIrp);
        PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)pFuncDevObj->DeviceExtension;
        return IoCallDriver(pDevExt->pLowerDevObj, pIrp);
}

//////////////////////////////////////////

#pragma CODE_SEG_PAGED

NTSTATUS PnpStartDeviceMinorHandler(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp)
{
        PAGED_CODE();

        NTSTATUS status = ForwardIrpAndWait(pFuncDevObj, pIrp);
        if (!NT_SUCCESS(status))
                return CompleteIrp(pIrp, status, pIrp->IoStatus.Information);

        PIO_STACK_LOCATION pStackLoc = IoGetCurrentIrpStackLocation(pIrp);

        PCM_PARTIAL_RESOURCE_LIST pRawList =
                pStackLoc->Parameters.StartDevice.AllocatedResources ?
                &pStackLoc->Parameters.StartDevice.AllocatedResources->
                                                        List[0].PartialResourceList :
                NULL;

        PCM_PARTIAL_RESOURCE_LIST pTransList =
                pStackLoc->Parameters.StartDevice.AllocatedResourcesTranslated ?
                &pStackLoc->Parameters.StartDevice.AllocatedResourcesTranslated->
                                                        List[0].PartialResourceList :
                NULL;

        status = StartDevice(pFuncDevObj, pRawList, pTransList);

        return CompleteIrp(pIrp, status, 0);
}

//////////////////////////////////////////

#pragma CODE_SEG_PAGED

NTSTATUS PnpStopDeviceMinorHandler(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp)
{
        PAGED_CODE();

        NTSTATUS status = PnpDefaultMinorHandler(pFuncDevObj, pIrp);
        StopDevice(pFuncDevObj);
        return status;
}

//////////////////////////////////////////

#pragma CODE_SEG_PAGED

NTSTATUS PnpRemoveDeviceMinorHandler(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp)
{
        PAGED_CODE();
```

```
        PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)pFuncDevObj->DeviceExtension;
        pDevExt->removing = TRUE;
        UnlockDevice(pDevExt);
        UnlockDevice(pDevExt);
        KeWaitForSingleObject(&pDevExt->removeEvent, Executive, KernelMode, FALSE, NULL);

        StopDevice(pFuncDevObj);
        NTSTATUS status = PnpDefaultMinorHandler(pFuncDevObj, pIrp);

        RemoveDevice(pFuncDevObj);

        return status;
}

//////////////////////////////////////////////

#pragma CODE_SEG_PAGED

NTSTATUS StartDevice(PDEVICE_OBJECT pFuncDevObj, PCM_PARTIAL_RESOURCE_LIST pRawList,

PCM_PARTIAL_RESOURCE_LIST pTransList)
{
        PAGED_CODE();

        NTSTATUS status;
        PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)pFuncDevObj->DeviceExtension;

        ASSERT(!pDevExt->started);

        if (!pTransList)
                return STATUS_DEVICE_CONFIGURATION_ERROR;

        PCM_PARTIAL_RESOURCE_DESCRIPTOR pTransDesc = pTransList->PartialDescriptors;
        ULONG descCount = pTransList->Count;

        BOOLEAN hasMemory = FALSE;
        PHYSICAL_ADDRESS memoryBase;
        ULONG memoryLength;

        BOOLEAN hasPort = FALSE;
        PHYSICAL_ADDRESS portBase;
        ULONG portLength;
        BOOLEAN mappedPort = FALSE;

        BOOLEAN hasIrq = FALSE;
        ULONG irqVector;
        KIRQL irqLevel;
        KINTERRUPT_MODE irqMode;
        KAFFINITY irqAffinity;

        for (ULONG i = 0; i < descCount; ++i, ++pTransDesc)
        {
                switch (pTransDesc->Type)
                {
                        case CmResourceTypeMemory:
                        {
                                hasMemory = TRUE;
                                memoryBase = pTransDesc->u.Memory.Start;
                                memoryLength = pTransDesc->u.Memory.Length;
                                break;
                        }
                        case CmResourceTypePort:
                        {
                                hasPort = TRUE;
                                portBase = pTransDesc->u.Port.Start;
                                portLength = pTransDesc->u.Port.Length;
                                mappedPort = ((pTransDesc->Flags & CM_RESOURCE_PORT_IO)==0);
                                break;
                        }
                        case CmResourceTypeInterrupt:
                        {
                                hasIrq = TRUE;
                                irqVector = pTransDesc->u.Interrupt.Vector;
                                irqLevel = (KIRQL)pTransDesc->u.Interrupt.Level;
```

```
                                irqMode = (pTransDesc->Flags ==
                                           CM_RESOURCE_INTERRUPT_LATCHED) ?
                                          Latched : LevelSensitive;
                                irqAffinity = pTransDesc->u.Interrupt.Affinity;
                                break;
                        }
                        case CmResourceTypeDma:
                        {
                                break;
                        }
                        default:
                        {
                                return STATUS_DEVICE_CONFIGURATION_ERROR;
                        }
                }
        }

        if ((!hasMemory) || (!hasPort) || (!hasIrq))
                return STATUS_DEVICE_CONFIGURATION_ERROR;

        pDevExt->pMemoryBase = (PVOID)MmMapIoSpace(memoryBase, memoryLength, MmNonCached);
        if (!pDevExt->pMemoryBase)
                return STATUS_NO_MEMORY;
        pDevExt->memoryLength = memoryLength;

        if (mappedPort)
        {
                pDevExt->pPortBase = (PVOID)MmMapIoSpace(portBase, portLength, MmNonCached);
                if (!pDevExt->pPortBase)
                {
                        MmUnmapIoSpace(pDevExt->pMemoryBase, memoryLength);
                        return STATUS_NO_MEMORY;
                }
        }
        else
                pDevExt->pPortBase = (PVOID)portBase.LowPart;
        pDevExt->portLength = portLength;
        pDevExt->mappedPort = mappedPort;
        pDevExt->irqLevel = irqLevel;

        DisableDevice(pDevExt);

        status = IoConnectInterrupt(&pDevExt->pIrqObj, (PKSERVICE_ROUTINE)IrqServRoutine,
                                    (PVOID)pDevExt, NULL, irqVector, irqLevel, irqLevel,
                                    irqMode, FALSE, irqAffinity, FALSE);
        if (!NT_SUCCESS(status))
        {
                MmUnmapIoSpace(pDevExt->pMemoryBase, pDevExt->memoryLength);
                if (mappedPort)
                        MmUnmapIoSpace(pDevExt->pPortBase, pDevExt->portLength);
                return status;
        }

        if (!KeSynchronizeExecution(pDevExt->pIrqObj, (PKSYNCHRONIZE_ROUTINE)EnableDevice,
                                    (PVOID)pDevExt))
        {
                IoDisconnectInterrupt(pDevExt->pIrqObj);
                MmUnmapIoSpace(pDevExt->pMemoryBase, pDevExt->memoryLength);
                if (mappedPort)
                        MmUnmapIoSpace(pDevExt->pPortBase, pDevExt->portLength);
                return STATUS_INVALID_DEVICE_STATE;
        }

        pDevExt->started = TRUE;

        return STATUS_SUCCESS;
}

//////////////////////////////////////////

#pragma CODE_SEG_PAGED

VOID StopDevice(PDEVICE_OBJECT pFuncDevObj)
{
```

```
        PAGED_CODE();

        PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)pFuncDevObj->DeviceExtension;

        if (!pDevExt->started)
                return;
        pDevExt->started = FALSE;

        ASSERT(pDevExt->pIrqObj);
        KeSynchronizeExecution(pDevExt->pIrqObj, (PKSYNCHRONIZE_ROUTINE)DisableDevice,
                               (PVOID) pDevExt);
        IoDisconnectInterrupt(pDevExt->pIrqObj);
        pDevExt->pIrqObj = NULL;

        MmUnmapIoSpace((PVOID)pDevExt->pMemoryBase, pDevExt->memoryLength);
        if (pDevExt->mappedPort)
                MmUnmapIoSpace((PVOID)pDevExt->pPortBase, pDevExt->portLength);
}

//////////////////////////////////////////////

#pragma CODE_SEG_PAGED

VOID RemoveDevice(PDEVICE_OBJECT pFuncDevObj)
{
        PAGED_CODE();

        PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)pFuncDevObj->DeviceExtension;
        NTSTATUS status;

        IoSetDeviceInterfaceState(&pDevExt->interfaceName, FALSE);
        if (pDevExt->interfaceName.Buffer)
                ExFreePool((PVOID)pDevExt->interfaceName.Buffer);

        if (pDevExt->pLowerDevObj)
                IoDetachDevice(pDevExt->pLowerDevObj);

        IoDeleteDevice(pFuncDevObj);
}
```

FireflyPower.h

```
/*****************************************************************************

    FireflyPower.h:           Power management functions header file
                              for the FireFly PCI board WDM device driver.

    Version 1.0               4/6/99

    Copyright 1999 University of Aveiro - Electronics and Telecom. Department
    All rights reserved.
    Developed by Arnaldo Oliveira - arnaldo@ua.pt

*****************************************************************************/

#ifndef __FIREFLYPOWER_H
#define __FIREFLYPOWER_H

////////////////////////////////////////////
/* Functions prototypes                   */

NTSTATUS DispatchPower(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp);

#endif // __FIREFLYPOWER_H
```

FireflyPower.cpp

```
/*****************************************************************************

    FireflyPower.cpp:  Power management functions implementation file
                       for the FireFly PCI board WDM device driver.

    Version 1.0        4/6/99

    Copyright 1999 University of Aveiro - Electronics and Telecom. Department
    All rights reserved.
    Developed by Arnaldo Oliveira - arnaldo@ua.pt

*****************************************************************************/

////////////////////////////////////////////
/* Includes                               */

#include "FireflyMain.h"
#include "FireflyPower.h"

////////////////////////////////////////////
/* Functions implementation               */

#pragma CODE_SEG_PAGED

NTSTATUS DispatchPower(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp)
{
        PAGED_CODE();

        PoStartNextPowerIrp(pIrp);
        IoSkipCurrentIrpStackLocation(pIrp);
        PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)pFuncDevObj->DeviceExtension;
        return PoCallDriver(pDevExt->pLowerDevObj, pIrp);
}
```

FireflySysCtrl.h

```
/*****************************************************************************

     FireflySysCtrl.h:        System control (WMI) functions header file
                              for the FireFly PCI board WDM device driver.

     Version 1.0              4/6/99

     Copyright 1999 University of Aveiro - Electronics and Telecom. Department
     All rights reserved.
     Developed by Arnaldo Oliveira - arnaldo@ua.pt

*****************************************************************************/

#ifndef __FIREFLYSYSCTRL_H
#define __FIREFLYSYSCTRL_H

////////////////////////////////////////////
/* Functions prototypes                   */

NTSTATUS DispatchSysCtrl(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp);

#endif // __FIREFLYSYSCTRL_H
```

FireflySysCtrl.cpp

```
/*****************************************************************************

     FireflySysCtrl.cpp:      System control (WMI) functions implementation file
                              for the FireFly PCI board WDM device driver.

     Version 1.0              4/6/99

     Copyright 1999 University of Aveiro - Electronics and Telecom. Department
     All rights reserved.
     Developed by Arnaldo Oliveira - arnaldo@ua.pt

*****************************************************************************/

////////////////////////////////////////////
/* Includes                               */

#include "FireflyMain.h"
#include "FireflySysCtrl.h"

////////////////////////////////////////////
/* Functions implementation               */

#pragma CODE_SEG_PAGED

NTSTATUS DispatchSysCtrl(PDEVICE_OBJECT pFuncDevObj, PIRP pIrp)
{
        PAGED_CODE();

        IoSkipCurrentIrpStackLocation(pIrp);
        PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)pFuncDevObj->DeviceExtension;
        return IoCallDriver(pDevExt->pLowerDevObj, pIrp);
}
```

Referências

[AdaBam99] Alexandro M. S. Adário, Sergio Bampi, "Reconfigurable Computing: Viable Applications and Trends", actas da conferência VLSI'99 - *VLSI Systems on a Chip*, Kluwer Academic Publishers, pp. 583-594, Lisboa, Portugal, Dezembro 1999.

[AdaRoeBam99] Alexandro M. S. Adário, Eduardo L. Roehe, Sergio Bampi, "Dinamically Reconfigurable Architecture for Image Processor Applications", actas da conferência DAC'99 – *36th Design Automation Conference*, pp. 623-628, New Orleans, Junho 1999.

[AleHan75] Igor Aleksander, F. Keith Hanna, "Automata Theory: An Engineering Approach", Crane, Russak & Company, Inc., 1975.

[Altera99] Altera Corporation, "Device Data Book", 1999.

[AshDevNew92] Pranav Ashar, Srinivas Devadas, A. Richard Newton, "Sequential Logic Synthesis", Kluwer Academic Publishers, 1992.

[Ashenden96] Peter J. Ashenden, "The Designer's Guide to VHDL", Morgan Kaufmann, 1996.

[Baranov74] Samary Baranov, "Synthesis of Microprogrammed Automata", Energy Publishing Company, 1974 (em Russo).

[Baranov94] Samary Baranov, "Logic Synthesis for Control Automata", Kluwer Academic Publishers, 1994.

[Baranov98] Samary Baranov, "Minimization of Algorithmic State Machines", actas do simpósio SBCCI'98 – *XI Brazilian Symposium on Integrated Circuit Design*, pp. 176-179, Búzios, Rio de Janeiro, Brasil, Outubro 1998.

[BelHut98] Peter Bellows, Brad Hutchings, "JHDL - An HDL for Reconfigurable Systems", actas do simpósio FCCM'98 – *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 175-184, Abril 1998.

[BooRumJac99] Grady Booch, James Rumbaugh, Ivar Jacobson, "The Unified Modeling Language User Guide", Addison Wesley Longman, Inc., 1999.

[Booth67] Taylor L. Booth "Sequential Machines and Automata Theory", John Wiley & Sons, Inc., 1967.

[BraHacMcMSan84] R. Brayton, G. Hachtel, C. McMullen, A. Sangiovanni-Vicentelli, "Logic Minimization Algorithms for VLSI Synthesis", Kluwer Academic Publishers, Boston, MA, 1984.

[BraRudSanWan87] R. Brayton, R. Rudell, A. Sangiovanni-Vicentelli, A. Wang, "Mis: Multiple-level interactive logic optimization systems", IEEE Transactions on Computer-Aided Design, Vol. CAD-6, pp. 1062-1081, Novembro 1987.

[Brebner96] G. Brebner, "A Virtual Hardware Operating System for the Xilinx XC6200", actas do workshop FPL'96 – *6th International Workshop on Field-Programmable Logic and Applications*, pp. 327-336, Darmstadt, Alemanha, Setembro 1996.

[BroRos96] Stephen Brown, Jonathan Rose, "FPGA and CPLD Architectures: A Tutorial", IEEE Design & Test of Computers, Vol. 13, Nº2, pp. 42-57, 1996.

[CamWol91] Raul Camposano, Wayne Wolf (Editores), "High-Level VLSI Synthesis", Kluwer Academic Publishers, 1991.

[Cant99] Chris Cant, "Writing Windows WDM Device Drivers", R&D Books, 1999.

[ChaMou94] Pak K. Chan, Samiha Mourad, "Digital Design using Programmable Gate Arrays", Prentice Hall, Inc., 1994.

[Clare73] Christopher R. Clare, "Designing Logic Systems Using State Machines", McGraw-Hill, Inc., 1973.

[DruHar89] Doron Drusinsky, David Harel, "Using Statecharts for Hardware Description and Synthesis", IEEE Transactions on Computer-Aided Design, Vol. 8, Nº7, pp. 798-807, Julho 1989.

[DuHacLinNew91] Xuejun Du, Gary Hachtel, Bill Lin, A. Richard Newton, "MUSE: A Multilevel Symbolic Encoding Algorithm for State Assignment", IEEE Transactions on Computer-Aided Design, Vol. 10, Nº1, pp. 28-38, Janeiro 1991.

[EldHut94] James G. Eldridge, Brad L. Hutchings, "RRANN: The Run-Time Reconfigurable Artificial Neural Network", IEEE Custom Integrated Circuit Conference, 1994.

[ErcLanMor99] Milós Ercegovac, Tomás Lang, Jaime H. Moreno, "Introduction to Digital Systems", John Wiley & Sons, Inc., 1999.

[FCCM99] Actas do simpósio FCCM'99 – *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, California, Abril 1999.

[FPGA99] Actas do simpósio FPGA'99 – *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Monterey, California, Fevereiro 1999.

[FPL98] Actas do workshop FPL'98 – *8th International Workshop on Field-Programmable Logic and Applications*, Tallin – Estónia, Agosto 1998.

[FPL99] Actas do workshop FPL'99 – *9th International Workshop on Field-Programmable Logic and Applications*, Glasgow – Reino Unido, Agosto 1999.

[GajDutWuLin91] D. D. Gajski, N. D. Dutt, C. H. Wu, Y. L. Lin, "High-Level Synthesis: Introduction to Chip and System Design" , Kluwer Academic Publishers, 1991.

[GajDutWuLin92] Daniel D. Gajski, Nikil D. Dutt, Allen Wu, Steve Lin, "High-Level Synthesis: Introduction to Chip and System Design", Kluwer Academic Publishers, 1992.

[GajRam94] Daniel D. Gajski, Loganath Ramachandran, "Introduction to High-Level Syntesis", IEEE Design & Test of Computers, Vol. 11, Nº4, pp. 44-54, 1994.

[Gajski97] Daniel D. Gajski, "Principles of Digital Design", Prentice Hall, Inc., 1997.

[GajVahNarGon94] Daniel D. Gajski, Frank Vahid, Sanjiv Narayan, Jie Gong, "Specification and Design of Embedded Systems", Prentice Hall, Inc., 1994.

[GraNel95] Paul Graham, Brent Nelson, "A Hardware Genetic Algorithm for the Traveling Salesman Problem on Splash2", actas do workshop FPL'95 – *5th International Workshop on Field-Programmable Logic and Applications*, pp. 352-361, Oxford, Reino Unido, Agosto 1995.

[Green86] David Green, "Modern Logic Design", Addison-Wesley Publishing Company, 1986.

[GupLia97] Rajesh K. Gupta, Stan Y. Liao, "Using a Programming Language for Digital System Design", IEEE Design & Test of Computers, Vol. 14, Nº2, pp. 72-80, 1997.

[GupMic93] Rajesh K. Gupta, Giovanni De Micheli, "Hardware-Software Cosynthesis for Digital Systems", IEEE Design & Test of Computers, Vol. 10, Nº3, pp. 29-41, 1993.

[Harel87] David Harel, "Statecharts: A Visual Formalism for Complex Systems", Science of Computer Programming, Nº8, pp. 231-271, 1987.

[Hauck98] S. Hauck, "Configuration Compression for the Xilinx XC6200 FPGA", actas do simpósio FCCM'98 – *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 138-146, Napa Valley, California, Abril 1998.

[IEEE94] IEEE, "IEEE standard VHDL language: reference manual", The Institute of Electrical and Electronics Engineers, 1994.

[IsmJer95] Tarek Ben Ismail, Ahmed Amine Jerraya, "Synthesis Steps and Design Models for Codesign", IEEE Computer, Vol. 28, Nº2, pp. 44-52, Fevereiro 1995.

[Katz94] Randy H. Katz, "Contemporary Logic Design", The Benjamin/Cummings Publishing Company, Inc., 1994.

[Kohavi70] Zvi Kohavi, "Switching and Finite Automata Theory", McGraw-Hill, Inc., 1970.

[KurBagAthMuñ00] Fadi J. Kurdahi, Nader Bagherzadeh, Peter Athanas, José L. Muñoz, "Guest Editors' Introduction: Configurable Computing", IEEE Design & Test of Computers, Vol. 17, Nº1, pp17-19, 2000.

[LauSkl99] Nuno Lau, Valery Sklyarov, "Dynamically Reconfigurable Implementation of Control Circuits", actas da conferência VLSI'99 - *VLSI Systems on a Chip*, Kluwer Academic Publishers, pp. 137-148, Lisboa, Portugal, Dezembro 1999.

[LiHau99] Zhiyuan Li, Scott Hauck, "Don't Care Discovery for FPGA Configuration Compression", actas do simpósio FPGA'99 – *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 91-98, Monterey, California, Fevereiro 1999.

[Lipman98] Jim Lipman, "Chip verification: a formal affair?", EDN, Janeiro 1998 (http://www.ednmag.com/reg/1998/010198/01df_02.htm).

[LudSloSin99] Stefan Ludwig, Robert Slous, Satnam Singh, "Implementing PhotoShop™ Filters in Virtex™", actas do workshop FPL'99 – *9th International Workshop on Field-Programmable Logic and Applications*, pp. 233-242, Glasgow, Reino Unido, Agosto 1999.

[MacPatSin99] Donald MacVicar, John W. Patterson, Satnam Singh, "Rendering PostScript™ Fonts on FPGAs", actas do workshop FPL'99 – *9th International Workshop on Field-Programmable Logic and Applications*, pp. 223-232, Glasgow, Reino Unido, Agosto 1999.

[MalFraAlvAmo98] L. Maltar, F. M. G. França, V. C. Alves, C. L. Amorim, "Reconfigurable Hardware for Tomographic Processing", actas do simpósio SBCCI'98 – *XI Brazilian Symposium on Integrated Circuit Design*, Búzios, Rio de Janeiro, Brasil, Outubro 1998.

[Maxfield96] Clive Maxfield, "Field Programmable Devices", EDN, Vol. 3, Nº21, pp. 201-206, Outubro 1996.

[Mayer97] John Mayer, "Reconfigurable Computing Redefines Design Flexibility", Computer Design, pp. 49-52, Fevereiro 1997.

[McFKow90] Michael C. McFarland, Thaddeus. J. Kowalski, "Incorporating Bottom-Up Design into Hardware Synthesis", IEEE Transactions on Computer-Aided Design, Vol. 9, Nº9, pp. 938-950, Setembro 1990.

[McFParCam90] Michael C. McFarland, Alice C. Parker, Raul Camposano, "The High-Level Synthesis of Digital Systems", Proceedings of the IEEE, Vol. 78, Nº2, pp. 301-318, Fevereiro 1990.

[McGLys99] Gordon McGregor, Patrick Lysaght, "Self Controlling Dynamic Reconfiguration: A Case Study", actas do workshop FPL'99 – *9th International Workshop on Field-Programmable Logic and Applications*, pp. 144-154, Glasgow, Reino Unido, Agosto 1999.

[Melo00] Andreia Barbosa de Melo, "Especificação, Optimização e Teste de Algoritmos de Controlo Hierárquicos", Dissertação de Mestrado em Engenharia Electrónica e de Telecomunicações, Universidade de Aveiro, Janeiro 2000.

[MelSarTysYemZve98] O. Melnikov, V. Sarvanov, R. Tyshkevich, V. Yemelichev, I. Zverovich, "Exercices in Graph Theory", Kluwer Academic Publishers, 1998.

[MicGup97] Giovanni De Micheli, Rajesh K. Gupta, "Hardware/Software Co-Design", Proceedings of the IEEE, Vol. 85, Nº3, pp. 349-365, Março 1997.

[Micheli94] Giovanni De Micheli, "Synthesis and Optimization of Digital Circuits", McGraw-Hill, Inc., 1994.

[MicLauDuz92] Petra Michel, Ulrich Lauhter, Peter Duzy (Editores), "The Synthesis Approach to Digital System Design", Kluwer Academic Publishers, 1992.

[Milne94] George Milne, "Formal specification and verification of digital systems", McGraw-Hill, Inc., 1994.

[Murata89] Tadao Murata, "Petri Nets: Properties, Analysis and Applications", Proceedings of the IEEE, Vol. 77, Nº4, pp. 541-590, Abril 1989.

[Nec98] NEC, "NEC Develops Dynamically Reconfigurable Logic Engine", *Press Release*, Fevereiro 1998 (http://www.nec.co.jp/english/today/newsrel/9902/1502.html).

[NelNagCarIrw95] Victor P. Nelson, H. Troy Nagle, Bill D. Carroll, J. David Irwin, "Digital Logic Circuit Analysis & Design", Prentice Hall, Inc., 1995.

[NisGuc97] S. Nisbert, S. Guccione, "The XC6200 Development System", actas do workshop FPL'97 – *7th International Workshop on Field-Programmable Logic and Applications*, pp. 61-68, Londres, Reino Unido, Setembro 1997.

[OliLauSkl98] Arnaldo Oliveira, Nuno Lau, Valery Sklyarov, "Synthesis of VHDL Code from the Hierarchical Specification of Control Circuits for Dynamically Reconfigurable FPGAs", actas da conferência VIUF'98 – *VHDL International User Forum*, Orlando, EUA, Outubro 1998.

[OliMelSkl99] A. Oliveira, A. Melo, V. Sklyarov, "Specification, Implementation and Testing of HFSMs in Dynamically Reconfigurable FPGAs", actas do workshop FPL'99 - *9th International Workshop on Field Programmable Logic and Applications*, pp. 313-322, Glasgow, Reino Unido, Agosto 1999.

[OliSkl99] Arnaldo Oliveira, Valery Sklyarov, "Especificação, Projecto e Implementação de Circuitos de Controlo Virtuais", Electrónica e Telecomunicações, Vol. 2, Nº4, pp. 487-495, Janeiro 1999.

[OneFol99] Walter Oney, Forrest Foltz, "Programming the Microsoft Windows Driver Model", Microsoft Press, 1999.

[Pam96] Jean E. Vuillemin, Patrice Bertin, Didier Roncin, Mark Shand, Hervé H. Touati, Philippe Boucard, "Programmable Active Memories: Reconfigurable Systems Come of Age", IEEE Transactions on VLSI Systems, Vol. 4, Nº1, pp. 56-69, Março 1996.

[Peterson81] J. L. Peterson, "Petri Net Theory and the Modeling of Systems", Prentice Hall, Englewood Cliffs, New Jersey, 1981.

[Reisig92] W. Reisig, "A Primer in Petri Net Design", Springer-Verlag, New York, 1992.

[RobLys99] David Robinson, Patrick Lysaght, "Modelling and Synthesis of Configuration Controllers for Dynamically Reconfigurable Logic Systems Using the DCS CAD Framework", actas do workshop FPL'99 – *9th International Workshop on Field-Programmable Logic and Applications*, pp. 41-50, Glasgow, Reino Unido, Agosto 1999.

[Rocha99] António Adrego da Rocha, "Synthesis and Simulation of Reprogrammable Control Units from Hierarchical Specifications", Tese de Doutoramento em Engenharia Electrotécnica e Informática, Universidade de Aveiro, Julho 1999.

[RocSkl97a] António Adrego da Rocha, Valery Sklyarov, "VHDL Modeling of Hierarchical Finite State Machines", actas do 5º workshop BELSIGN, Dresden, Abril 1997.

[RocSkl97b] António Adrego da Rocha, Valery Sklyarov, "Simulação em VHDL de Máquinas de Estados Finitas Hierárquicas", Electrónica e Telecomunicações, Vol. 2, Nº1, pp. 83-94, Setembro 1997.

[RocSklFer97] António Adrego da Rocha, Valery Sklyarov, António Ferrari, "Hierarchical Description and Design of Control Circuits Based on Reconfigurable and Reprogrammable Elements", actas do workshop IWLAS'97 – *International Workshop on Logic and Architectural Synthesis*, pp. 73-82, Grenoble, Dezembro 1997.

[SalSma97] Zoran Salsic, Asim Smailagic, "Digital Systems Design and Prototyping using Field Programmable Logic", Kluwer Academic Publishers, 1997.

[ShiLukChe98] N. Shirazi, W. Luk, P. Y. K. Cheung, "Run-Time Management of Dynamically Reconfigurable Designs", actas do workshop FPL'98 – *8th International Workshop on Field-Programmable Logic and Applications*, pp. 58-68, Tallin, Estónia, Agosto 1998.

[SidMeiPra99a] Reetinder P. S. Sidhu, Alessandro Mei, Viktor K. Prasanna, "String Matching on Multicontext FPGAs using Self-Reconfiguration", actas do simpósio FPGA'99 – *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 217-226, Monterey, California, Fevereiro 1999.

[SidMeiPra99b] Reetinder P. S. Sidhu, Alessandro Mei, Viktor K. Prasanna, "Genetic Programming Using Self-Reconfigurable FPGAs", actas do workshop FPL'99 – *9th International Workshop on Field-Programmable Logic and Applications*, pp. 301-312, Glasgow, Reino Unido, Agosto 1999.

[SinBel94] Satnam Singh, Pierre Bellec, "Virtual Hardware for Graphics Applications using FPGAs", actas do simpósio FCCM'94 – *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, California, Abril 1994.

[SinHogMcA96] Satnam Singh, Jonathan Hogg, Derek McAuley, "Expressing Dynamic Reconfiguration by Partial Evaluation", actas do simpósio FCCM'96 – *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 188-194, Napa Valley, California, Abril 1996.

[SinLeeLuKurBagFil98] Hartej Singh, Ming-Hau Lee, Guangming Lu, Fadi J. Kurdahi, Nader Bagherzadeh, Eliseu M. C. Filho, "MorphoSys: A Reconfigurable Architecture for Multimedia Applications", actas do simpósio SBCCI'98 – *XI Brazilian Symposium on Integrated Circuit Design*, pp. 134-139, Búzios, Rio de Janeiro, Brasil, Outubro 1998.

[SinPatBurDal97] Satnam Singh, John Patterson, Jim Burns, Michael Dales, "PostScript™ Rendering with Virtual Hardware", actas do workshop FPL'97 – *7th International Workshop on Field-Programmable Logic and Applications*, Londres, Reino Unido, Setembro 1997.

[Sis92] Ellen M. Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan, Robert K. Brayton, Alberto Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis", Memorando nº UCB/ERL M92/41, Electronics Research Laboratory, University of California, Berkeley, CA 94720, Maio 1992.

[Skahill96] Kevin Skahill, "VHDL for Programmable Logic", Addison-Wesley, Publishing Inc., 1996.

[SklFer98] Valery Sklyarov, António Ferrari, "Synthesis of Control Devices Described by Hierarchical Graph-Schemes", actas da conferência ACAC'98 – *3rd Australian Computer Architecture Conference*, pp. 181-191, Perth, Fevereiro 1998.

[SklRoc96a] Valery Sklyarov, António Adrego da Rocha, "Síntese de Unidades de Controlo Descritas por Grafos de um Esquema Hierárquicos", Electrónica e Telecomunicações, Vol. 1, Nº6, pp. 577-588, Setembro 1996.

[SklRoc96b] Valery Sklyarov, António Adrego da Rocha, "Synthesis of Control Units Described by Hierarchical Graph-Schemes", actas do 4º workshop BELSIGN, Santander, Novembro 1996.

[Sklyarov00] Valery Sklyarov, "Synthesis and Implementation of RAM-based Finite State Machines in FPGAs", actas do workshop FPL'00 – *10th International Workshop on Field-Programmable Logic and Applications*, Villach, Austria, Agosto 2000.

[Sklyarov84] Valery Sklyarov, "Hierarchical Graph-Schemes", Latvian Academy of Science, Nº2, pp. 82-87, Riga, 1984 (em Russo).

[Sklyarov87] Valery Sklyarov, "Parallel Graph-Schemes and Finite State Machines Synthesis", Latvian Academy of Science, Automatics and Computers, Nº5, pp. 68-76, Riga, 1987 (em Russo).

[Sklyarov99] Valery Sklyarov, "Hierarchical Finite-State Machines and Their Use for Digital Control", IEEE Transactions on VLSI Systems, Vol. 7, Nº2, pp. 222-228, Junho 1999.

[Statemate90] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, Mark B. Trakhtenbrot, "Statemate: A Working Environment for the Development of Complex Reactive Systems", IEEE Transactions on Software Engineering, Vol. 16, Nº4, pp. 403-414, Abril 1990.

[StaWol97] Jørgen StaunStrup, Wayne Wolf (Editores), "Hardware/Software Co-Design: Principles and Practice", Kluwer Academic Publishers, 1997.

[Stroustrup95] B. Stroustrup, "The C++ Programming Language", Segunda Edição, Addison-Wesley Publishing Company, 1995.

[ThoAdaSch93] Donald E. Thomas, Jay K. Adams, Herman Schmit, "A Model and Methodology for Hardware-Software Codesign", IEEE Design & Test of Computers, Vol. 10, Nº3, pp. 6-15, 1993.

[ThoMoo96] Donald E. Thomas, Philip R. Moorby, "The Verilog hardware description language", Kluwer Academic Publishers, 1996.

[Trimberger94] Stephen M. Trimberger (Editor), "Field Programmable Gate Array Technology", Kluwer Academic Publishers, 1994.

[VLSI99] Luís Miguel Silveira, Srinivas Devadas, Ricardo Reis (Editores), actas da conferência VLSI'99 – *VLSI: Systems on a Chip*, Kluwer Academic Publishers, 1999.

[WalCha95] Robert A. Walker, Samit Chaudhury, "Introduction to the Scheduling Problem", IEEE Design & Test of Computers, Vol. 12, Nº2, pp. 60-69, 1995.

[WinPro80] David Winkel, Franklin Prosser, "The Art of Digital Design: An Introduction to Top-Down Design", Prentice Hall, Inc., 1980.

[WirHut95] Michael J. Wirthlin, Brad L. Hutchings, "DISC: The Dynamic Instruction Set Computer", actas do simpósio FCCM'95 – *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 99-107, Napa Valley, California, Abril 1995.

[Wolf94] Wayne H. Wolf, "Hardware-Software Co-Design of Embedded Systems", Proceedings of the IEEE, Vol. 82, Nº7, pp. 967-989, Julho 1994.

[WooTraHer98] Roger Woods, David Trainor, Jean-Paul Heron, "Applying an XC6200 to Real-Time Image Processing", IEEE Design & Test of Computers, Vol. 15, Nº 1, pp. 30-38, 1998.

[Xilinx97a] Xilinx Corporation, "The Programmable Locic Data Book", 1997.

[Xilinx97b] Xilinx Corporation, "Velab: VHDL Elaborator for XC6200", 1997 (http://www.xilinx.com/apps/velabrel.htm).

[Xilinx97c] Xilinx Corporation, "Series 6000 User Guide", 1997.

[Xilinx99] Xilinx Corporation, "The Programmable Logic Data Book", 1999.

[ZhaPap96] Wei Zhao, Christos A. Papachristou, "Synthesis of Reusable DSP Cores Based on Multiple Behavior", IEEE/ACM International Conference on Computer Aided Design, IEEE/ACM Disgest of Technical Papers, pp. 103-108, San Jose, California, Novembro 1996.

Lista de Acrónimos

ASIC	Application Specific Integrated Circuit
ASM	Algorithmic State Machine
BJT	Bipolar Junction Transistor
CAD	Computer Aided Design
CISC	Complex Instruction Set Computer
CMOS	Complementary Metal Oxide Semiconductor
CPLD	Complex Programmable Logic Device
DLL	Dynamic Link Library
DMEES	Distância Mínima entre os Estados de Entrada e de Saída
DSP	Digital Signal Processor
DTM	Distância Total Mínima
EDIF	Electronic Data Interchange Format
EECMOS	Electrically Erasable Complementary Metal Oxide Semiconductor
EEPLD	Electrically Erasable Programmable Logic Device
EEPROM	Electrically Erasable Programmable Read Only Memory
EPROM	Erasable Programmable Read Only Memory
ER	Elemento Reprogramável
FPD	Field Programmable Device
FPGA	Field Programmable Gate Array
FPLD	Field Programmable Logic Device
FSM	Finite State Machine
FSMD	Finite State Machine with Datapath
GAL	Gate Array Logic
GFSM	Generalized Finite State Machine
GS	Graph Scheme
HaPFSM	Hierarchical and Parallel Finite State Machine
HDL	Hardware Description Language
HFSM	Hierarchical Finite State Machine
HGS	Hierarchical Graph Scheme
HPFSM	Hierarchical Parallel Finite State Machine

IP	Intellectual Property
IRP	Input/output Request Packet
LSI	Large-Scale Integration
LUT	Lookup Table
MFC	Microsoft Foundation Classes
MPGA	Mask Programmable Gate Array
MPLD	Mask Programmable Logic Device
MSI	Medium-Scale Integration
OTP	One-time Programmable
PAL	Programmable Array Logic
PCI	Peripheral Component Interconnect
PFSM	Parallel Finite State Machine
PHFSM	Parallel Hierarchical Finite State Machine
PHGS	Parallel Hierarchical Graph Scheme
PLA	Programmable Logic Array
PLD	Programmable Logic Device
PROM	Programmable Read Only Memory
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
SoC	Systems on a Chip
SPLD	Simple Programmable Logic Device
SRAM	Static Random Access Memory
SSI	Small-Scale Integration
STD	State Transition Diagram
STT	State Transition Table
UML	Unified Modeling Language
UVCMOS	Ultraviolet Complementary Metal Oxide Semiconductor
VFSM	Virtual Finite State Machine
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
VLSI	Very-Large Scale Integration
WDM	Windows Driver Model