FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Suporte em Linux para reconfiguração dinâmica de hardware

Bruno Miguel da Silva Monteiro

Tese submetida no âmbito do

Mestrado Integrado em Engenharia Electrotécnica e de Computadores

Major de Telecomunicações

Orientador: Prof. João Canas Ferreira

Fevereiro de 2009

U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

MIEEC - MESTRADO INTEGRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES 2008/2009

A Dissertação intitulada

"SUPORTE EM LINUX PARA RECONFIGURAÇÃO DINÂMICA DE HARDWARE"

foi aprovada em provas realizadas em 27/Fevereiro/2009

o júri

Presidente Professor Doutor Pedro Henrique Henriques Guedes de Oliveira
Professor Catedrático da Faculdade de Engenharia da Universidade do Porto

Professor Doutor Arnaldo Silva Rodrigues de Oliveira
Professor Auxiliar Convidado da Universidade de Aveiro

Professor Doutor João Paulo de Castro Canas Ferreira
Professor Auxiliar da Faculdade de Engenharia da Universidade do Porto

Bruno Miguel da Silva Monteiro

O autor declara que a presente dissertação (ou relatório de projecto) é da sua exclusiva autoria e foi escrita sem qualquer apoio externo não explicitamente autorizado. Os resultados, ideias, parágrafos, ou outros extractos tomados de ou inspirados em trabalhos de outros autores, e demais referências bibliográficas usadas, são correctamente citados.

Autor - BRUNO MIGUEL DA SILVA MONTEIRO

Faculdade de Engenharia da Universidade do Porto

Resumo

A cada nova geração de FPGAs verifica-se uma evolução tanto da capacidade como da densidade lógica. A inclusão de CPUs embutidos e circuitos dedicados permite a implementação de sistema reconfiguráveis mais complexos e com melhores desempenhos. Os computadores reconfiguráveis normalmente tiram partido do CPU embutido para o controlo das operações, utilizando os blocos reconfiguráveis para implementar circuitos dedicados no processamento de dados. Algumas FPGAs são capazes de se reconfigurarem sem interromperem o seu funcionamento, permitindo a utilização de módulos diferentes ao longo do tempo, optimizados para tarefas especificas.

Contudo a utilização da computação reconfigurável é dificultada pela complexidade do processo e pela falta de ferramentas de desenvolvimento. Assim, é necessário dotar os projectistas de uma forma *standard* de controlar os recursos reconfiguráveis, utilizando um sistema operativo como o GNU/Linux. Este projecto pretende expandir o núcleo GNU/Linux (2.6.27) com métodos para controlar a reconfiguração das FPGAs e aceder aos módulos instanciados. Através de um controlador de dispositivo, os programadores dispõem de uma forma fácil para utilizar os recursos das áreas reconfiguráveis, utilizando o fluxo normal de projecto a que estão habituados.

A unidade básica de *hardware* reconfigurável, para o sistema, é o DRM (*Dynamic Reconfigurable Module*), ou seja, um módulo em *hardware* que utiliza os recursos das áreas reconfiguráveis para implementar funções lógicas optimizadas para uma tarefa. A troca entre DRMs é realizada pelo sistema operativo sempre que as aplicações o requisitem, garantindo a disponibilidade de recursos.

O desenvolvimento do trabalho esteve orientado para o estudo dos mecanismos básicos necessários. Foi desenvolvido um sistema base, incluindo as configurações do *hardware* e do *software*, que permite que o núcleo GNU/Linux adicione, remova, comunique com e troque DRMs. Esta tese detalha a interface do SO com o *hardware* e as diferentes formas de comunicação entre o núcleo e o utilizador. Foi implementada um solução como prova de conceito, servindo de base a futuros desenvolvimentos.

Abstract

Successive generations of FPGAs have continuously improved logic density and capacity. Recent generations include embedded CPUs and other dedicated blocks enabling the implementation of very complex high-performance reconfigurable embedded systems. One of the main approaches to reconfigurable computing employs embedded CPU for control operations and exploits the reconfigurable fabric for the implementation of dedicated support circuits for the computation-intensive stages of data processing. The ability of some FPGAs to reconfigure themselves while maintaining normal operation (dynamic partial reconfiguration) allows the use of mutually exclusive dedicated hardware modules at different time periods during application execution.

However, the use of reconfigurable computing is hampered by the complexity of the process and the lack of development support tools. It is therefore important to provide application developers with a standard way of managing the dynamically reconfigurable resources when using traditional operating systems like GNU/Linux. The current project aims to extend a current GNU/Linux Kernel (2.6.27) to manage the dynamic reconfiguration of hardware and the access to the dynamically installed hardware modules. With the new device driver developed in this work, GNU/Linux application developers have an easy way to exploit the resources of the reconfigurable areas. This will enable the use of the normal application development flow in the context of reconfigurable computing.

The basic hardware unit to the system is a Dynamic Reconfigurable Module (DRM), i.e. hardware modules that implement logical functions in the reconfigurable area. The exchange between DRMs is performed by the operating system at the time the applications request it, ensuring the availability of resources.

We developed a base system, including hardware and software configurations. The implemented expansion allows an application to add, remove, communicate and switch between DRMs. This thesis details the interface with the hardware layer and communication between the kernel and the user space, through sysfs and a char device. A solution was implemented as proof of concept, providing the basis for future developments.

Agradecimentos

Para a realização desta tese houve várias pessoas que contribuíram positivamente a quem gostaria de agradecer o seu contributo.

A primeiro lugar gostaria de agradecer ao meu orientador, o Prof. João Canas Ferreira, pelo apoio demonstrado e pela confiança depositada.

Gostaria também de agradecer aos engenheiros Miguel Silva e Eurico Damásio, investigadores da FEUP na área dos circuitos reconfiguráveis, pela ajuda e apoio disponibilizado. Uma palavra de apreço também aos meus colegas de laboratório, Bruno Dantas e Francisco Basadre.

Por fim, um agradecimento muito especial aos meus pais e irmãos pelo encorajamento e paciência tidas durante a realização deste trabalho.

Bruno Monteiro

"If we knew what it was we were doing,
it would not be called research, would it?"

Albert Einstein

Conteúdo

Lista de Figuras

Lista de Tabelas

Abreviaturas e Símbolos

ASIC	*Application Specific Integrated Circuit*
BRAM	*Block Random-Access Memory*
CLB	*Configurable Logic Block*
CPLD	*Complex Programmable Logic Device*
DRM	*Dynamic Reconfigurable Module*
E/S	Entrada/Saida
FPGA	*Field Programmable Gate Arrays*
I2C	*Inter-Integrated Circuit*
ICAP	*Internal Configuration Access Port*
GPIO	*General Purpose Input/Output*
LUT	*Look-Up Table*
MPR	*Module-Based Partial Reconfiguration*
OPB	*On-Chip Peripheral Bus*
PAL	*Programmable Array Logic*
PLB	*Processor Local Bus*
PLD	*Programmable Logic Device*
PRA	*Partial Reconfigurable Area*
RAM	*Random-Access Memory*
SO	Sistema Operativo
SPI	*Serial Peripheral Interface*
SRAM	*Static Random Access Memory*
USB	*Universal Serial Bus*

Capítulo 1

Introdução

O interesse em dispositivos reconfiguráveis, especialmente em FPGAs (*Field Programmable Gate Arrays*), tem-se acentuado ao longo da última década. Inicialmente eram muito úteis no desenvolvimento de protótipos, devido à sua facilidade de configuração. A capacidade das FPGAs foi melhorada combinando os blocos lógicos tradicionais com microprocessadores embutidos e alguns periféricos, formando uma plataforma FPGA. Actualmente, as plataformas FPGA constituem um sistema eficiente, mesmo como solução final. Assim, as FPGAs são um bom compromisso entre o desempenho de um circuito dedicado e a sua flexibilidade.

Um dos últimos desenvolvimentos está relacionado com a capacidade de reconfiguração parcial dinâmica, que possibilita a alteração do dispositivo sem que isso implique uma paragem do funcionamento dos circuitos já configurados. Esta liberdade permite criar sistemas que apenas utilizem as zonas de *hardware* necessárias, diminuindo o consumo, ou modifiquem uma região do *hardware* para desempenhar uma determinada tarefa, aumentando o desempenho e diminuindo a área física necessária.

Actualmente, os projectos que utilizam a reconfiguração dinâmica baseiam-se no conhecimento total da estrutura do *hardware*. Os projectistas utilizam funções especificas para interagir com cada módulo do *hardware*, controlando o estado global dos recurso com métodos próprios de reconfiguração. No entanto, o aumento dos recursos das FPGAs e da complexidade dos projectos inviabiliza esta abordagem linear.

A flexibilidade dos recursos reconfiguráveis assemelha-os cada vez mais ao *software*, sendo natural a utilização de metodologias de projecto semelhantes. A utilização de um Sistema Operativo (SO), capaz de gerir as múltiplas aplicações e os vários módulos reconfiguráveis, permite um desenvolvimento mais rápido e flexível. O controlo do *hardware* é realizado pelo SO, resolvendo conflitos de acessos e permitindo aos programadores desenvolverem aplicações da forma tradicional, sem necessitarem de conhecer detalhadamente

o *hardware*.

Os sistemas embutidos estão constantemente em evolução, sofrendo alterações significativas na sua estrutura de base. Com a utilização de uma camada de abstracção é possível adaptar as aplicações desenvolvidas mais rapidamente a diferentes arquitecturas, minimixando o impacto das alterações na base do sistema.

1.1 Objectivos

O objectivo principal deste trabalho foi desenvolver um controlador de dispositivo que permita ao núcleo GNU/Linux (versão 2.6.27) aceder e controlar módulos reconfiguráveis dinamicamente. Assim, desenvolveram-se os mecanismos necessários por forma a que o núcleo GNU/Linux seja capaz de:

- configurar a(s) área(s) dinâmica(s) com um módulo previamente sintetizado;
- alterar a configuração da área dinâmica para diferentes módulos;
- controlar o acesso das aplicações à(s) área(s) dinâmica(s);
- controlar o acesso das aplicações aos módulos reconfiguráveis;
- aceder à memória de configuração das FPGAs.

O trabalho foi desenvolvido com vista a disponibilizar uma interface simples e intuitiva que auxilie o desenvolvimento de novas aplicações que utilizem a reconfiguração dinâmica.

1.2 Estrutura da tese

Para além deste capítulo introdutório, esta tese está organizada em mais quatro capítulos. No capítulo 2 são apresentados os sistemas reconfiguráveis dinamicamente, bem como as ferramentas que suportam a reconfiguração parcial. São analisados os métodos de gestão dos sistemas reconfiguráveis, principalmente os baseados num sistema operativo. No capítulo 3 é apresentado o modelo de funcionamento e a interface do controlador de dispositivo. O capítulo 4 descreve a implementação realizada, apresentando os resultados obtidos. No capítulo 5 são apresentas as conclusões do trabalho realizado e apontadas possíveis direcções para melhoria da usabilidade dos sistemas reconfiguráveis.

Capítulo 2

Enquadramento

Os circuitos configuráveis, normalmente designados por PLD (*Programmable Logic Device*), permitem que o utilizador altere a sua configuração de forma a implementar diferentes funções. Apesar de não apresentarem o mesmo desempenho do que um ASIC (*Application Specific Integrated Circuit*), estes dispositivos possibilitam um desenvolvimento mais flexível. A capacidade de definir, ou redefinir, a sua configuração lógica em qualquer momento torna o desenvolvimento mais rápido e económico, evitando-se o longo ciclo de fabrico dos ASICs e o custo associado a cada iteração até ao produto final.

A família de PLDs é ainda bastante extensa incluindo circuitos como FPGAS (*Field Programmable Gate Arrays*), CPLDs (*Complex Programmable Logic Devices*) ou PALs (*Programmable Array Logic*). De entre eles, os mais interessantes para este projecto são as FPGAs pois possuem maior densidade lógica e melhor desempenho.

2.1 Introdução

As FPGAs são circuitos integrados constituídos por blocos lógicos reprogramáveis. Cada bloco pode desempenhar uma função lógica diferente, interligando-se de diversas formas com os restante blocos. A sua estrutura será um pouco mais detalhada na secção 2.2.

Estes dispositivos são comercializados desde os anos 80, evoluindo constantemente a sua capacidade e densidade lógica. A sua utilização torna-se natural em vários domínios como processamento digital de sinal, processamento de áudio e imagem, bioinformática e criptografia [1]. São geralmente sistemas que tiram partido da realização de múltiplas tarefas em simultâneo, optimizando o consumo e espaço necessários.

O consumo de potência em tarefas implementadas numa FPGA é, normalmente, mais baixo do que quando realizadas num processador uma vez que, tipicamente, operam a

frequências mais baixas e não necessitam de recursos adicionais para processar operações individuais (*fetching*, *decoding*, ...) [1]. Há ainda a possibilidade de diminuir o consumo de energia através da aproximação de blocos funcionais ou melhoria das ligações de sinal de relógio, responsáveis por uma parte significativa de consumo nas arquitecturas tradicionais [2].

Os erros de projecto ou falhas permanentes, devido a processos de fabrico, têm um impacto importantíssimo nos sistemas actuais. Com capacidade de reconfiguração das FPGAs é possível minimizar os riscos, prolongando o tempo de vida dos dispositivos, ao evitar as áreas danificadas, tal como é exemplificado na figura 2.1.

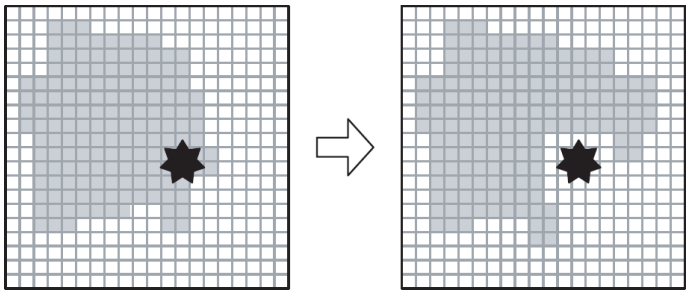


Figura 2.1: Correcção de falhas (preto) em FPGAs através da através da reconfiguração dos blocos afectados (cinzento) para outras áreas (segundo [1]).

As potencialidades das FPGAs foram largamente incrementadas com a inclusão de microprocessadores embutidos no *chip* da FPGA, blocos de memória e periféricos na mesma placa, formando uma plataforma FPGA. As plataformas FPGA são aplicadas em vários sistemas embutidos, desempenhando funções com elevadas exigência de desempenho ou consumo como comunicações sem fios, encriptação ou processamento digital de vídeo/áudio, permitindo corrigir falhas ou actualizar protocolos posteriormente. O *hardware* reconfigurável é utilizado para processar os blocos funcionais mais "pesados", recolhendo-se apenas os dados finais, tal como é ilustrado na figura 2.2.

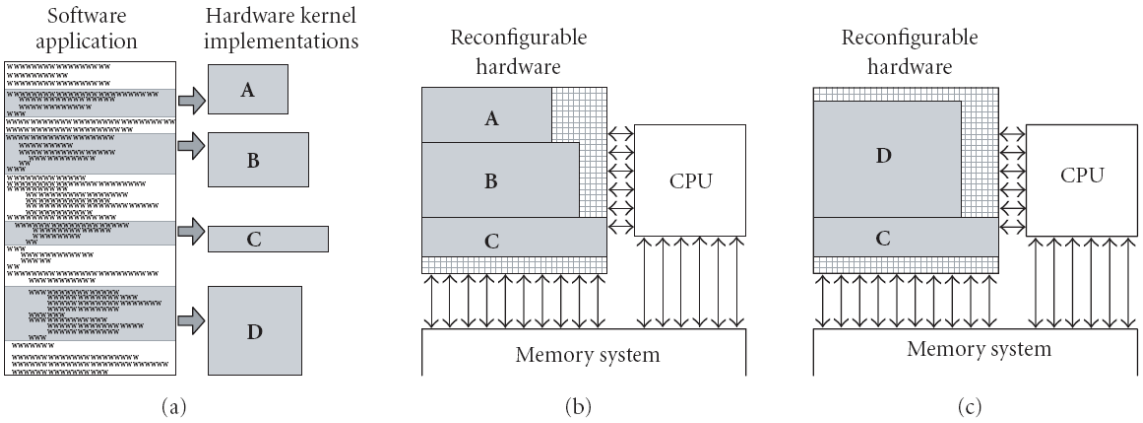


Figura 2.2: Implementação de algoritmos(a) em sistemas combinados CPU + FPGA(b), utilizando a reconfiguração dinâmica para optimizar os recursos(c)(retirado de [1]).

É possível encontrar esta tecnologia em diversos domínios como nas telecomunicações, medicina, redes, encriptação, aerospacial, robótica, automóvel e multimédia. A sua utilização nestes domínios deve-se à flexibilidade em alterar protocolos, à capacidade de processar dados em paralelo ou funções repetitivas e à relação área/funcionalidades/consumo [1].

As plataformas FPGAs permitiram a reformulação de alguns paradigmas da computação como a computação reconfigurável [3]. A computação reconfigurável é realizada por máquinas em que os dados e o controlo podem variar dinamicamente, contrastando com o fluxo de dados único de um processador.

A arquitectura de um computador reconfigurável não é única [4]. A figura 2.3 demonstra as várias possibilidades. Os primeiros quatro tipos têm em comum a presença de um processador de controlo, isolado fisicamente da área reconfigurável, diferindo na forma como se liga à estrutura reconfigurável.

Na figura 2.3(a) a estrutura reconfigurável é ligada através dos barramentos de E/S do sistema. Sendo a mais comum e fácil de implementar peca pela reduzida largura de banda. As figuras 2.3(b) e 2.3(c) têm melhor largura de banda ao incluírem a estrutura reconfigurável numa zona de acessos rápidos, como a da memória. O sistema da figura 2.3(d) incorpora a estrutura reconfigurável no caminho de dados do processador, permitindo o acesso a dados locais como os registos mas não tira partido dos ganhos de uma arquitectura paralela.

A figura 2.3(e) representa uma nova estrutura em que, ao contrário das restantes, o processador é embutido na estrutura dinâmica. Este processador pode ser um *Soft-Core* [5], implementado utilizando os recursos lógicos da FPGA, ou estar embutido na estrutura da FPGA, como é o caso do PowerPC utilizado na VirtexII PRO [6].

2.2 Arquitectura das FPGAs

A arquitectura mais comum das FPGAs, normalmente designada por *island-style*, implementa uma matriz de blocos lógicos, interligados por um rede configurável, e rodeada por blocos de E/S, cujas configurações são controladas por um conjunto de células de memória, conforme ilustrado na figura 2.4. A célula lógica é normalmente designada por CLB (*Configurable Logic Block*), e pode ser implementada de formas diferentes como se descreve a seguir. A rede de ligações entre CLBs é constituida por ligações locais, que ligam CLBs vizinhos, e por ligações globais, que podem ligar CLBs em lados opostos. Os blocos E/S situam-se na periferia do dispositivo, permitindo a ligação do circuito ao exterior.

As FPGAs são normalmente divididas de acordo com a granulosidade da sua arquitectura, não existindo um fronteira bem delimitada. Nas arquitecturas de granulosidade fina

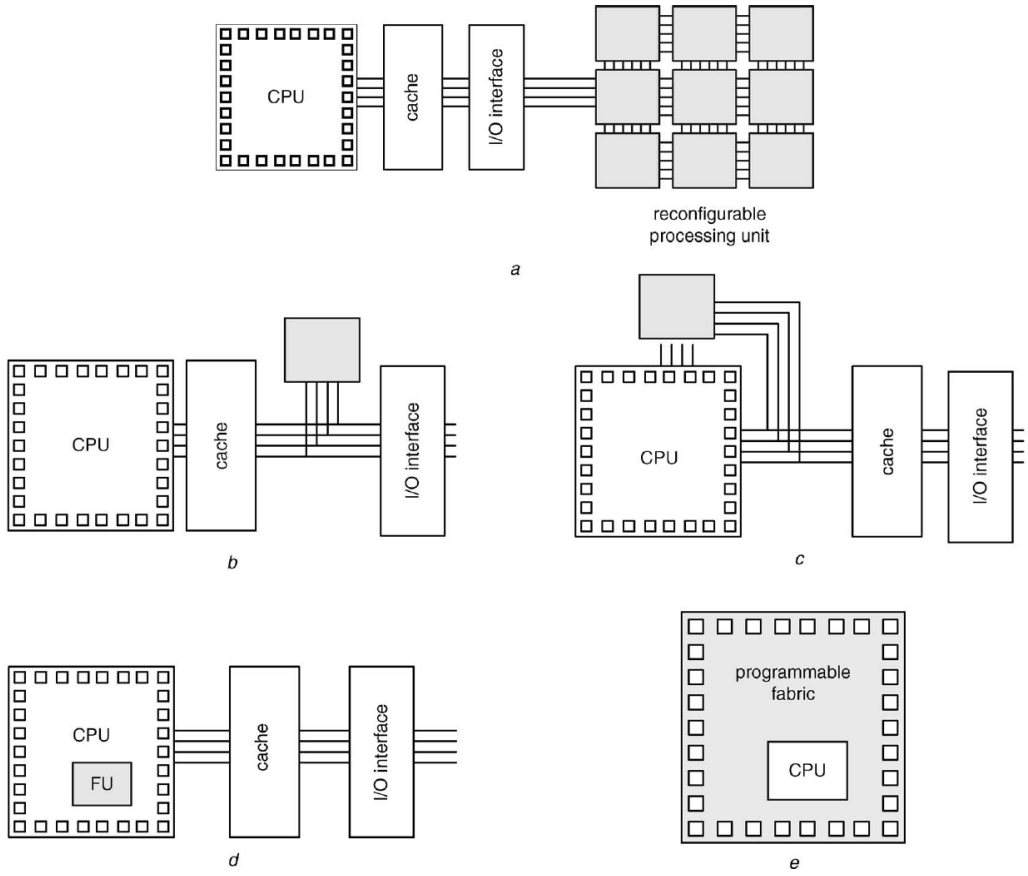


Figura 2.3: Classificação da arquitectura de um computador reconfigurável (segundo de [4]).

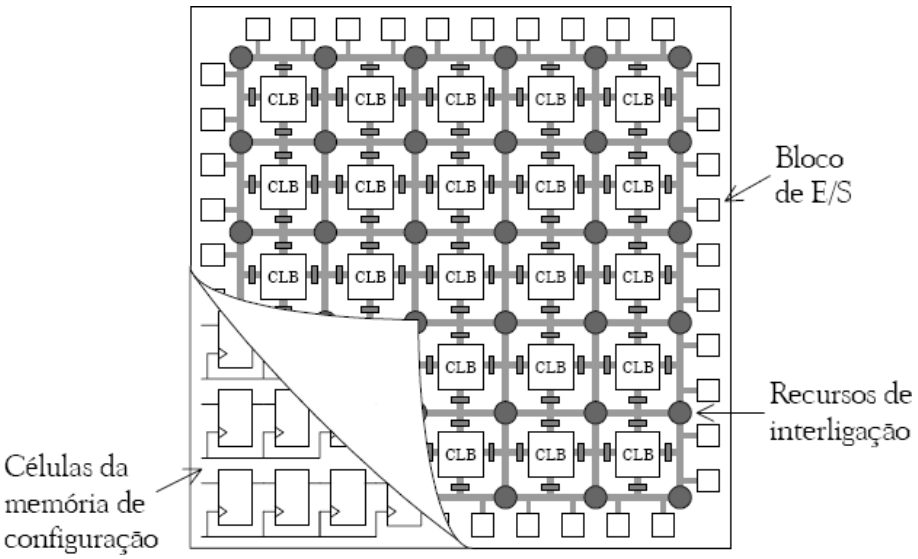


Figura 2.4: Arquitectura Geral de uma FPGA (retirado de [7]).

os CLBs são mais simples, contendo um número reduzido de portas lógicas e multiplexadores e, raramente, um *flip-flop*. Noutras arquitecturas, de granulosidade mais grossa, os CLBs têm LUTs (*Look-Up Table*), para a implementação lógica, e mais *flip flops*, para guardar o estado. Esta é a estrutura mais utilizada pois permite implementar funcionalidades mais complexas aproveitando melhor a rede de ligações. É comum os CLBs utilizarem uma rede interna de multiplexadores para ligarem as suas entradas e saídas às LUTs e *flip-flops*. Configurando correctamente as ligações internas e externas dos CLBs é possível implementar qualquer circuito lógico.

A estrutura da rede de ligações é diversa, procurando-se atingir o ponto óptimo entre as ligações locais e globais. O objectivo é minimizar os atrasos sem afectar a sua flexibilidade.

Com a estrutura descrita é possível implementar qualquer circuito combinacional ou sequencial, podendo ser necessário uma elevada área. Assim, para aumentar o desempenho das FPGAs incluíram-se mais blocos para além dos CLBs como multiplicadores, blocos RAM (*Random-Access Memory*) e processadores.

A limitada capacidade de reter dados de um CLB foi colmatada com a inclusão blocos de RAM estática. Os blocos de memória, distribuídos pela estrutura da FPGA, podem ser utilizados com *buffers*, guardando os dados intermédios ou diminuindo a frequência de comunicação com o exterior.

A inclusão de um ou mais processadores [8], como o PowerPC [9], potencia os algoritmos com um controlo de fluxo das operações mais complexo. Apesar de ser difícil de optimizar a utilização deste recurso, é possível utilizar as ferramentas normais de computação como compiladores [10] e sistemas operativos [11, 12, 13].

As FPGAs diferem, também, na forma como são configuradas. A tecnologia anti-fusível apresenta um baixo consumo no entanto só pode ser programada uma única vez. É utilizada principalmente em aplicações espaciais devido à sua resistência a radiações [14].

As memórias Flash são uma solução que apresenta uma baixo consumo e permite preservar a configuração, mesmo ficando sem energia. No entanto o número de vezes que pode ser reconfigurada é limitado.

As SRAMs(*Static Random Access Memory*) são a solução mais comum devido à sua velocidade e infinita capacidade de reconfiguração. No entanto têm maior consumo energético e tamanho quando comparadas com as duas soluções anteriormente apresentadas. É comum encontrar-se uma memória não volátil associada à FPGA, garantindo a correcta configuração da FPGA após uma falha energética.

As FPGAs podem, assim, apresentar arquitecturas diversas [7] de acordo com a função a desempenhar. A plataforma de desenvolvimento utilizada foi a Virtex II Pro [6], descrita a seguir.

2.2.1 Virtex II Pro

As Virtex II Pro [15] são uma família de FPGAs da Xilinx baseada na tecnologia SRAM. Tal como se pode observar na figura 2.5, apresenta uma estrutura praticamente simétrica. As colunas de CLBs são separadas, com espaçamentos regulares, por uma coluna de BRAMs, existindo um processador embutido do lado esquerdo.

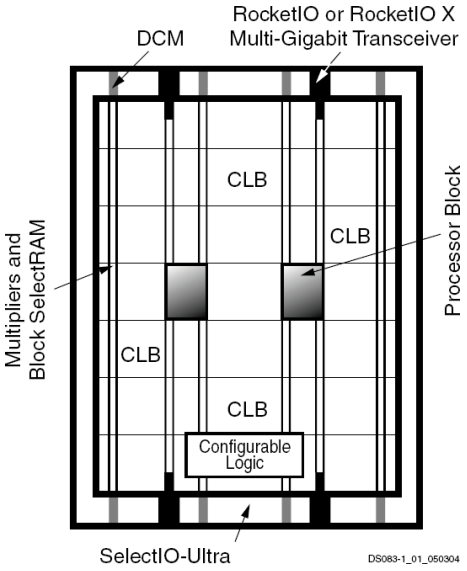
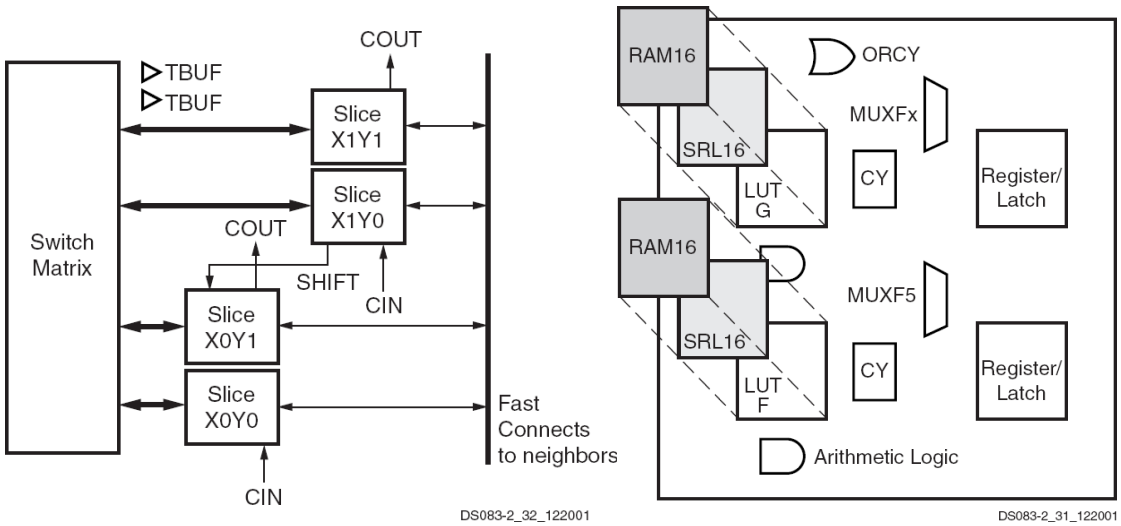


Figura 2.5: Arquitectura Geral de uma VirtexII (retirado de [15]).

Cada CLB é constituído por quatro células divididas em duas colunas, em que cada coluna tem a sua própria lógica de transporte como se pode verificar na figura 2.6 (a). Na 2.6 (b) pode observar-se mais detalhadamente a lógica presente em cada *Slice*.



(a) (b)

Figura 2.6: Estrutura de um CLB (a) com a visão detalhada de uma *Slice* (b) (segundo [15]).

A memória de configuração pode ser representada por uma matriz rectangular de *bits*. A matriz está organizada em vectores verticais de um *bit* de largura e comprimento proporcional ao número de linhas, sendo esta a estrutura mínima que se pode configurar [16]. Os vectores são agrupados em colunas cujo tamanho varia de acordo com a tabela 2.1. A figura 2.7 demonstra a atribuição das diferentes colunas da memória de configuraçãoaos recursos configuráveis.

Tabela 2.1: Tipos de colunas de configuração (baseado em [17]).

Tipo de coluna de configuração	**Número de vectores da coluna de configuração**	**Número de colunas de configuração por componentes**
Central	8	1
CLB	48	Nº de colunas de CLBs
IOB	54	2
Interligações dos blocos de memória RAM	27	2
Conteúdo dos blocos de memória RAM	64	2

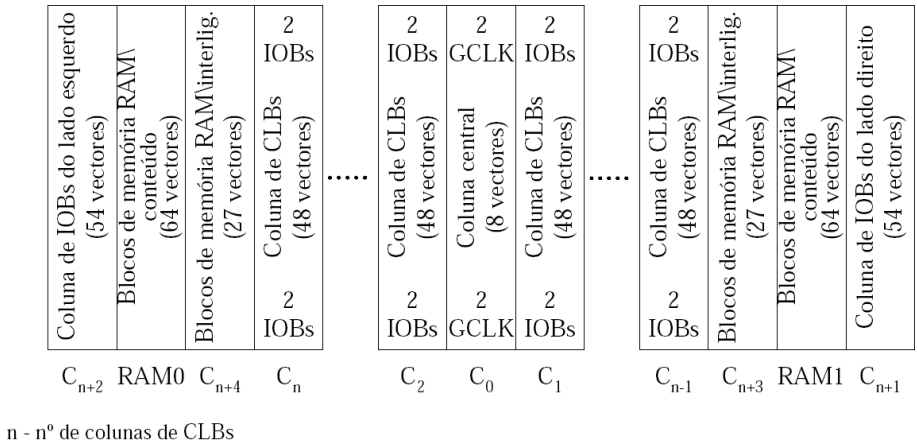


Figura 2.7: Organização da memória de configuração (retirado de [17]).

A alteração da memória de configuração é realizada recorrendo a ficheiros *Bitstream*. Estes ficheiros contêm a configuração de um ou mais vectores da memória de configuração, podendo ter associados alguns parâmetros como o endereço a que se refere o vector. Os *bitstreams* são específicos para cada modelo de FPGA.

2.3 Reconfiguração dinâmica de FPGAs

Os sistemas reconfiguráveis permitem que as aplicações possam recorrer a diferentes configurações do *hardware*, optimizadas para cada tarefa que pretendem realizar. Desta forma é possível acelerar o processamento ou utilizar uma área menor para executar uma

determinada tarefa que utilize as partes, sequencialmente, de uma área maior. A reconfiguração dinâmica é, assim, a alteração da configuração durante a execução de tarefas.

A reconfiguração demora tempo, na ordem das centenas de milissegundos para uma reconfiguração completa [15], pelo que esta técnica impõem algumas limitações. Os processos de configuração e utilização do *hardware* que está a ser alterado são mutuamente exclusivos tornando o tempo gasto na reconfiguração em tempo "perdido", do ponto de vista das operações. Têm sido propostos diversos métodos de forma a optimizar o tempo gasto em cada reconfiguração [18, 19].

2.3.1 Arquitectura de configuração

As FPGAs com capacidade de reconfiguração, tecnologias SRAM e Flash, apresentam diferentes processos de reconfiguração [3]. Este pode ser realizado em série ou através de estruturas endereçáveis de dimensão pré-definida.

Na reconfiguração em série, de contexto único, o vector de configuração é sequencial começando na posição zero. Apesar da sua rigidez requer um número reduzido de pinos.

Os dispositivo multi-contexto permitem guardar várias configurações em simultâneo. Cada configuração é guardada em paralelo, sendo apenas uma escolhida através de um multiplexador. Desta forma é possível, num único ciclo de relógio, trocar completamente a configuração. No entanto requer mais área que o normal, para armazenar cada configuração.

Nas FPGAs mais recentes, a memória de configuração é endereçável ao vector. Assim, é possível alterar apenas uma coluna da configuração sem afectar as restantes como se pode ver a seguir.

2.3.2 Reconfiguração parcial

A reconfiguração parcial é a capacidade de alterar apenas uma porção da configuração da FPGA. A porção mínima de área reconfigurável depende da estrutura interna da FPGA, sendo independente das restantes áreas.

A divisão da totalidade da memória em vectores endereçáveis permite maior flexibilidade no desenvolvimento de módulos. A área lógica pode ser partilhada por funções concorrentes, sendo configurada para a que está activa. Assim, a área reconfigurável ganha uma nova dimensão, o tempo. A figura 2.8 demonstra a possibilidade de escalonamento de funções em que cada uma têm o tempo e espaço necessários.

O tempo de reconfiguração parcial está na ordem dos milissegundos [7], dependendo da dimensão do módulo. Um escalonamento eficiente dos módulos pode minimizar o impacto deste tempo aproveitando o espaço livre para carregar o módulo seguinte. Desta forma o processamento não é interrompido pelo processo de configuração. Naturalmente que o aumento de desempenho está dependente da área livre.

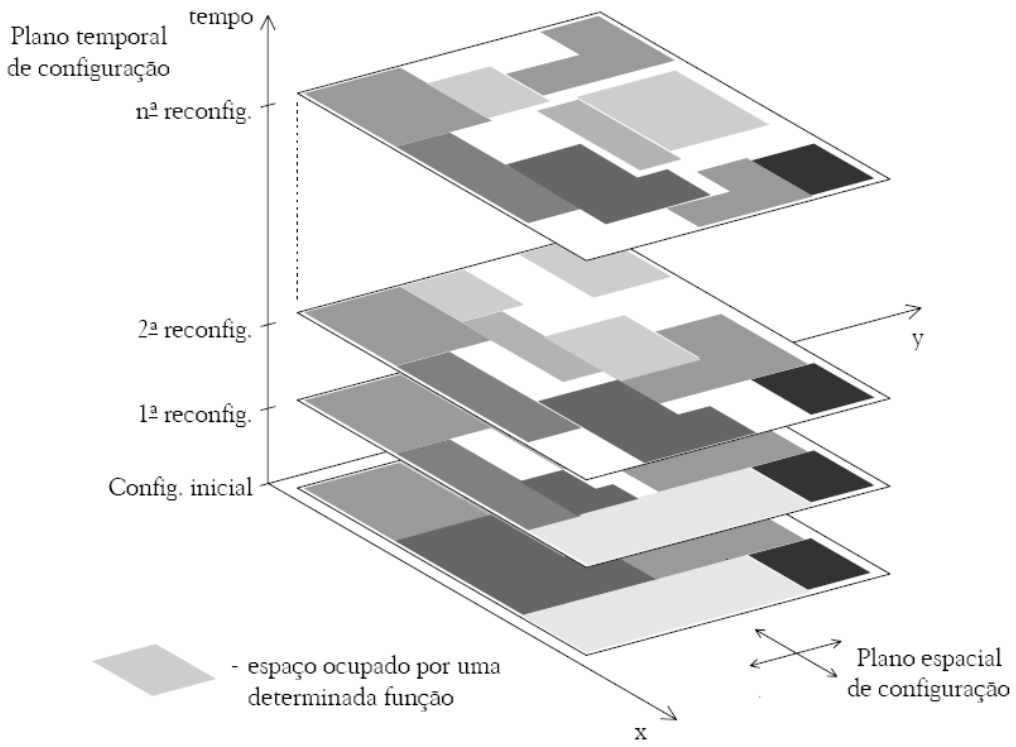


Figura 2.8: Visão tridimensional do espaço de configuração (segundo [7]).

O posicionamento e dimensão dos módulos é importantíssimo para um projecto eficiente. O número de colunas que são reconfiguradas é directamente proporcional à dimensão horizontal do módulo, sendo necessário reconfigurar toda a coluna mesmo que se mude apenas uma pequena parte. Espera-se, por isso, que um projectista tente tirar o máximo partido de um módulo verticalmente. Este problema foi minimizado nas Virtex5 [20] em que a cada coluna é dividida em segmentos endereçáveis.

2.3.3 Ferramentas de suporte

O *bitstream* de cada módulo é, normalmente, gerado para uma determinada posição que inclui um conjunto conhecido de recursos, contendo no cabeçalho o endereço a que se refere cada coluna. Existem projectos [21, 22] que flexibilizam o desenvolvimento durante a fase de compilação permitindo reposicionar o *bitstream* noutra região, tirando partido da simetria entre colunas.

2.3.3.1 Xilinx

As ferramentas da Xilinx suportam a geração de *bitstreams* parciais. Utiliza-se o *ISE Foundation*, com umas alterações [23], para gerar as *netlists* da área estática e de cada um

dos módulos reconfiguráveis dinamicamente. Cada área reconfigurável tem a interface definida, tendo os módulos reconfiguráveis de seguir o mesmo padrão.

Com o *PlanAhead Design Analysis Tool* obtêm-se os *bitstreams* correspondentes a cada uma das *netlists*. Assim, no final do processo estão disponíveis as configurações para área total da FPGA e um *bitstream* para módulo reconfigurável. A localização dos módulos é fixa, contendo internamente a sua localização dentro da FPGA.

O fluxo de projecto é apresentado na anexo B.

2.3.3.2 BitLinker

O BitLinker é uma ferramenta, desenvolvida por Miguel Silva [21, 24] que agrega vários módulos de *hardware*, já compilados, num novo *bitstream* desenvolvido para a reconfiguração parcial dinâmica. A aplicação é semelhante a um ligador (*linker*) uma vez que agrega componentes pré-compilados num novo módulo binário.

O CPU comunica com a área dinâmica através de um módulo ligado a barramentos bidirecionais (entrada/saída). A interface pode ser de 64 ou 32 *bits*, o PLB (*Processor Local Bus*) e o OPB *On-Chip Peripheral Bus* respectivamente. Este módulo é normalmente designado por *Dock* uma vez que define a interface de ligação entre os módulos reconfiguráveis e o resto do sistema.

A comunicação entre a *Dock* e os módulos ou entre módulos diferentes é realizada através de CLBs pré-definidos. Esta abordagem é justificada pois são muitas as configurações possíveis, tal como é exemplificado na figura 2.9, e é necessário manter pontos fixos de referência, designados por *Bus Macros*.

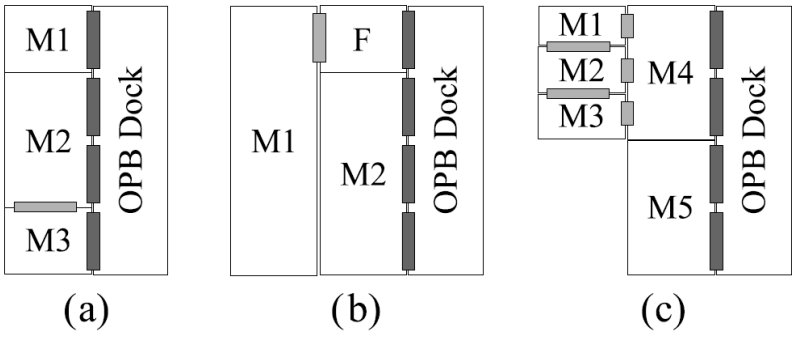


Figura 2.9: Diferentes configurações da área dinâmica com diferentes componentes (retirado de [24]).

O BitLinker retira a informação do *bitstream* total, sendo necessário projectar os novos módulos num projecto completo. Outros cuidados são necessários como por exemplo, garantir que o módulo pode ser recolocado e ocupa uma área inferior à da área reconfigurável. A ferramenta é capaz de retirar vários módulos do mesmo projecto, adaptá-los à estrutura da área dinâmica, executar testes de compatibilidade e corrigir pequenos erros

na ligação entre módulos. No final, produz um ficheiro com o *bitstream* parcial de cada módulo. O seu funcionamento é apresentado na figura 2.10.

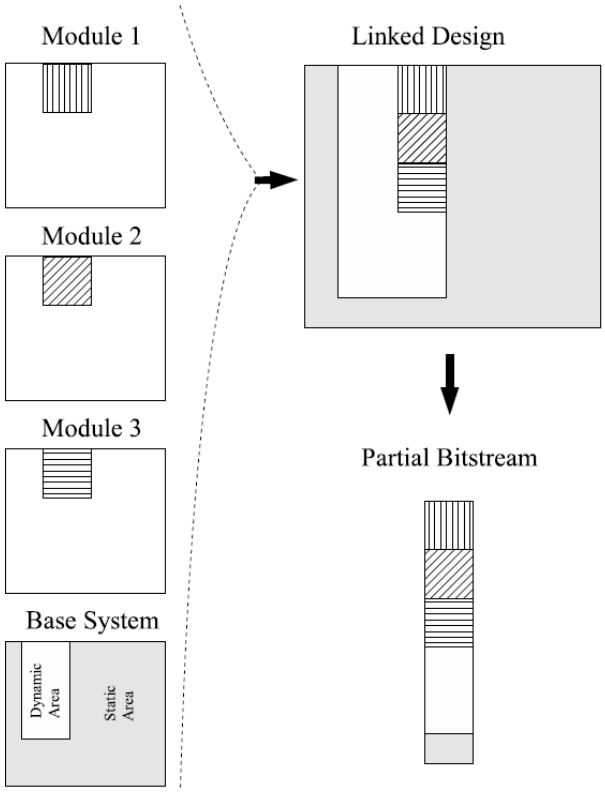


Figura 2.10: Funcionamento conceptual do BitLinker (segundo [21]).

A ferramenta possui, também, a capacidade de colocar automaticamente os módulos na área dinâmica. Primeiro configura os módulos junto da *Dock*, de seguida os módulos a este ligados e assim sucessivamente para cada nível. A atribuição de uma determinada área é realizada por níveis, permitindo maior flexibilidade na troca de módulos semelhantes, e o mais próximo possível dos pontos fixos de ligação entre diferentes níveis.

A aplicação já suporta a reconfiguração parcial dinâmica através de ficheiros MPR (*Module-Based Partial Reconfiguration*) e de uma biblioteca de funções, escrita em C. Os ficheiros estão organizados em duas secções, o cabeçalho e configuração. O cabeçalho contém informações relativas ao tipo de ficheiro (*bitstream* completo, mínimo ou para regiões livres), número de colunas ocupadas, tamanho do *bitstream* parcial e o formato das entradas e saídas do módulo.

2.4 Gestão de sistemas reconfiguráveis dinamicamente

Os sistemas reconfiguráveis têm cada vez mais recursos distintos. Por exemplo, uma plataforma FPGA para além dos recursos internos (FPGA, CPU, BRAM, ...) contêm tam-

bém memórias RAM externas, portas RS232, portas *ethernet*, interfaces USB e SATA, ..., enfim, são um verdadeiro computador. A gestão de uma área reconfigurável dinamicamente pode ser aproximada à gestão dos processos por um Sistema Operativo, ou seja, gerir os recursos que cada aplicação pode usufruir.

No entanto, apesar crescimento investimento, ainda não existem ferramentas que auxiliem o desenvolvimento e utilização de aplicações. O actual processo de desenvolvimento é bastante distinto do tradicional, aumentando as dificuldades de transição das equipas de desenvolvimento para esta nova tecnologia. Assim, torna-se necessário desenvolver ferramentas que suportem a utilização de módulos reconfiguráveis em sistemas em funcionamento.

2.4.1 Sistema operativo

A optimização de recursos estáticos já é complicada. Mais um grau de liberdade torna o controlo bastante complexo.

A utilização de um Sistema Operativo (SO) para controlar os recursos surge como uma opção natural permitindo o desenvolvimento de aplicações com um nível de abstracção superior. Apesar de algumas perdas no desempenho, os aumentos potenciais quer da flexibilidade quer da velocidade de projecto são uma mais valia, representado assim um bom compromisso.

Com a utilização de um SO é possível projectar novas aplicações utilizando linguagens já conhecidas, no desenvolvimento de *software*, e interligar *software* e *hardware* de uma forma simples e produtiva, optimizando-se os recursos existentes.

O SO deve suportar os processos de comunicação, como *ethernet* ou RS232, gerir a memória e controlar os periféricos. Desta forma, os projectistas podem focar a sua atenção na funcionalidade que pretendem implementar uma vez que o SO já reconhece os dispositivos base do sistema. Esta capacidade permite ainda aplicar técnicas de depuração mais evoluídas, baseadas em métodos já testados.

A utilização de métodos conhecidos facilita, ainda, a adaptação aos novos sistemas e arquitecturas permitindo uma mais rápida adaptação de novos utilizadores. Desta forma é mais simples adaptar aplicações já desenvolvidas para outros sistemas, benificiando do modelo de abstração disponilizado pelo SO.

Esta camada de abstracção permite ainda o aumento da portabilidade das novas aplicações. Como estas utilizam os métodos disponibilizados pelo SO, se houver uma alteração no *hardware* não é necessário reescrever o código da aplicação, bastando alterar o controlador desse módulo se for necessário. Desta forma minimiza-se a dependência de um modelo específico de FPGA quer durante o desenvolvimento quer depois na produção.

GNU/Linux

O GNU/Linux [25] é o SO que mais plataformas suporta. Os PowerPCs embutido nas FPGAs da Xilinx não são excepção. Estes são suportados nas versões 2.4, através da arquitectura *ppc*, e 2.6, através da arquitectura *powerpc*. A versão actualmente recomendada é a 2.6 e é a que tem maior desenvolvimento, enquanto que na 2.4 apenas se fazem correcções de *bugs*.

A utilização do GNU/Linux permite a utilizações de ferramentas "comuns" como compiladores ou depuradores, permitindo que novos projectistas se adaptem rapidamente ao novo sistema de desenvolvimento. Por exemplo, é possível executar um terminal, via porta série, na FPGA controlando os vários processos em execução.

A Xilinx iniciou um projecto [11] que visa testar e desenvolver código, de suporte às suas plataformas, para o GNU/Linux. Estão em fase de testes controladores para periféricos GPIO (*General Purpose Input/Output*), I2C (*Inter-Integrated Circuit*), SPI (*Serial Peripheral Interface*) e USB (*Universal Serial Bus*). Neste projecto podemos encontrar alguma documentação sobre a execução do núcleo GNU/Linux em FPGAs.

A reconfiguração parcial dinâmica é suportada através do controlador de dispositivo XILINX HWICAP . Este controlador permite o acesso à porta ICAP (*Internal Configuration Access Port*) e à reconfiguração de qualquer área da FPGA, através do *IP-Core* OPB HWICAP [26], não sendo realizado qualquer controlo e validação de dados. Espera-se que o utilizador seja competente e saiba o que faz.

Outros Sistemas Operativos

Existem alguns projectos académicos com o objectivo de desenvolver Sistemas Operativos específicos para *hardware* reconfigurável.

O projecto ReConfigME [27, 12] implementa um SO, dividido em camadas, com suporte para reconfiguração dinâmica. Impõem restrições nos formatos e dimensões dos programas e módulos desenvolvidos, encontrando-se ainda em fase de desenvolvimento.

O BORPH [28, 13] (Berkeley Operating system for ReProgrammable Hardware) é uma adaptação do Linux para FPGAs permitindo executar aplicações tradicionais ou implementadas directamente em *hardware*. Este projecto introduz a utilização de processos em *hardware*, estabelecendo a comunicação de uma forma similar aos processos tradicionais. Desta forma, é possível executar os programas já desenhados e projectar novos utilizando mecanismos já conhecidos de *stdin* e *stdout*.

2.5 Projectos semelhantes

Existem alguns projectos que visam adaptar o GNU/Linux, ou uma variante, à reconfiguração dinâmica dotando-o de métodos de gestão da área reconfigurável.

O projecto FPGA *support system* (FSS) [29] utilizou o *e*COS de modo a controlar um sistema, constituído por um processador ARM ligada a uma FPGA, capaz de de executar as tarefas básicas de reconfiguração. O grande foco do projecto foram as tarefas em Tempo Real.

O sistema Egret [30] foi utilizado [31] na reconfiguração dinâmica utilizando o μClinux [32]. Este SO é uma adaptação do Kernel GNU/Linux [25] para sistemas embutidos com recursos limitados, como o caso das plataformas FPGA, suportando as aplicações típicas do GNU/Linux como linha de comandos, bibliotecas em C ou chamadas ao sistema Unix.

Rana *et. al.* [33] desenvolveram um sistema reconfigurável dinamicamente com base no GNU/LINUX. Consideram que o sistema tem uma FPGA central, de controlo, e permitem ligar uma ou mais FPGAs que implementam vários módulos reconfiguráveis.

2.6 Resumo

Ao longo deste capítulo apresentou-se uma breve descrição dos sistemas reconfiguráveis, com relevo para os reconfiguráveis dinâmica e parcialmente. Abordou-se a complexidade destes sistemas e a necessidade de ferramentas de projecto, descrevendo-se os projectos realizados na área. Procurou-se descrever o essencial sobre estas matérias de modo a facilitar a compreensão do trabalho realizado.

Capítulo 3

Especificação do sistema

No capítulo anterior analisou-se a evolução da complexidade das FPGAs. Estes sistemas podem vir a revolucionar as actuais formas de computação ao permitirem o desenvolvimento de soluções com o desempenho do *hardware* e a flexibilidade do *software*. No entanto, tal como em todas as novas tecnologias, ainda não existem ferramentas de projecto que tornem o processo fácil.

Existem algumas ferramentas capazes de produzir módulo reconfiguráveis dinamicamente, como as apresentadas na secção 2.3.3. Neste capítulo descreve-se a utilização de um nova ferramenta para o passo seguinte, a utilização dos módulos reconfiguráveis dinamicamente. A camada de abstracção proposta pretende tornar o fluxo de projecto mais rápido e flexível. É necessário, contudo, alguma cautela relativamente aos recursos consumidos de modo a que não ultrapassem os benefícios da sua utilização.

3.1 Pré-requisitos

O modelo do sistema assume o cumprimento de alguns pré-requisitos:

- o sistema contém uma FPGA reconfigurável dinamicamente através de uma porta interna de configuração - ICAP;
- as áreas reconfiguráveis estão acessíveis através de interfaces físicas pré-definidas, compatíveis com a interface utilizada, a DOCKI32O32, definida na secção 4.1.1;
- os *bitstreams* parciais de cada módulo reconfigurável, implementam as funcionalidades pretendidas e têm um interface compatível com a DOCKI32O32;
- o SO está convenientemente configurado e são passadas informações correctas sobre a configuração do *hardware*;

- as aplicações conhecem os nomes dos módulos que pretendem utilizar.

3.2 Análise de casos de utilização

A grande vantagem dos recursos reconfiguráveis é a sua flexibilidade, no entanto torna-se mais complexo desenvolver o controlo para um sistema com tantos graus de liberdade. Assim, para melhor compreensão do potencial das aplicações, e dos problemas envolvidos, são analisados três casos de utilização genéricos. Cada caso de utilização tem como base o caso anterior, acrescentando uma nova capacidade.

3.2.1 Utilização de um módulo reconfigurável

O caso de utilização mais simples é implementação de um único DRM (*Dynamic Reconfigurable Module*) numa única PRA (*Partial Reconfigurable Area*).

A utilização de uma interface física previamente definida, como a DOCKI32O32 descrita na secção 4.1.1, define o canal de comunicação sendo apenas necessário formatar adequadamente a mensagem. Quer a aplicação quer o módulo devem conhecer antecipadamente a dimensão dos vectores de dados, ou utilizar comandos de inicio e fim de trama, evitando assim que o sistema esteja à espera de valores inexistentes ou inválidos.

A partilha do DRM é facilmente resolvida utilizando os mecanismos próprios do GNU/Linux, considerando o DRM como um ficheiro. Este ficheiro só pode ser controlado por um único processo, que pode no entanto utilizar o DRM e libertá-lo para outro processo. Nesta situação pode-se utilizar os registos da área reconfigurável para passar informação entre os processos.

Um controlo mais refinado da área reconfigurável permite um melhor aproveitamento dos recursos e o aumento do potencial do sistema. Por exemplo, se os parâmetros internos do DRM estiverem localizados numa única *frame* é interessante re-reconfigurar (R^2) a PRA para os alterar. Desta forma utiliza-se um estrutura mais flexível e económica, o *software*, para o controlo do estado do *hardware* evitando a abordagem tradicional através de barramentos. Apesar desta aplicação não estar limitada a uma *frame*, necessita de um posicionamento inteligente dos recursos.

3.2.2 Utilização de vários módulos reconfiguráveis

O caso de utilização mais comum é a partilha de uma área reconfigurável por mais do que um DRM, existindo diferentes programas a tentar aceder a diferentes configurações dos recursos.

Com a utilização de vário DRMs, o sistema tem de controlar não só quem é que tem acesso aos recursos reconfiguráveis como também a ordem de utilização. Tal como é feito nos processos, existem várias formas de resolver este problema, devendo ser escolhida a

que melhor se adequando ao objectivo. A troca de DRMs tem um custo temporal associado bastante superior ao da troca entre processos pelo que é necessário alguma cautela, de modo a que cada aplicação tenha uma "janela de utilização" adequada.

A troca entre DRMs implica que o sistema guarde o *bitstream* correspondente a cada DRM, permitindo uma troca transparente para a aplicação. O sistema deve guardar o *bitstream* de uma forma eficiente e que minimize o tempo de reconfiguração.

Neste modelo deve-se considerar a possibilidade de o DRM "ter memória", ou seja, o seu resultado depende do estado interno dos registos. Assim, é possível guardar a configuração instantânea de cada DRM, através de técnicas de *readback*, permitindo preservar o estado do DRM entre diferentes acessos. Este processo pode ser optimizado através da minimização dos registos guardados, permitindo ao sistema manipular uma quantidade de dados menor.

3.2.3 Utilização de várias áreas reconfiguráveis

O sistema pode conter mais do que uma zona reconfigurável. Nesse caso é necessário alterar o processo de escalonamento de modo a que contemple todas as áreas reconfiguráveis e os seus recursos. As áreas reconfiguráveis não necessitam de ser homogéneas, devendo o escalonamento dos DRMs ter em conta factores como a frequência de utilização e os recursos utilizados pelo DRM.

Neste caso de utilização, o sistema deve ser capaz de reposicionar os *bitstreams* de cada módulo ou cada DRM disponibilizar um *bitstream* para cada área. A última opção é a mais simples de implementar, no entanto é a que impõem mais restrições, como por exemplo na preservação do estado dos registos.

3.2.4 Segurança do sistema

Os principais focos de possíveis vulnerabilidades no sistema são o acesso indevido à configuração de um DRM e o acesso à configuração da FPGA. O primeiro problema é um caso típico de gestão de acessos, que pode ser resolvido utilizando o sistema de permissões e algum cuidado na programação. O segundo problema implica que os sistema verifique que cada acesso à porta ICAP, validando a zona que pretende configurar.

3.3 Modelo de abstracção

O objectivo principal deste projecto é disponibilizar um nível de abstracção para o *hardware* reconfigurável. A abstracção desenvolvida permite o desenvolvimento de novas aplicações suportadas numa camada modular e expansível.

O modelo considera que cada módulo reconfigurável, designado por DRM, é um periférico virtual, definindo uma interface para cada operação permitida, como é detalhado na

secção 3.4. A figura 3.1 demonstra o modelo geral do controlador em que as aplicações dispõem dos métodos habituais de leitura e escrita nos dispositivos.

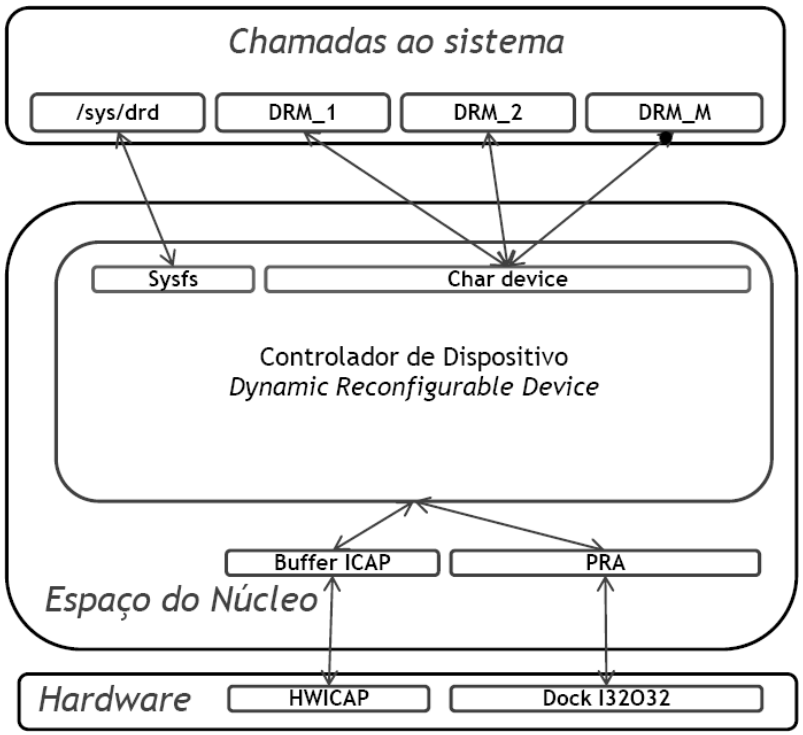


Figura 3.1: Modelo de atracção do sistema.

Cada DRM funciona de forma isolada, do ponto de vista estrutural, contendo as suas próprias informações. Ao ser instanciado, cada módulo é definido com um nome, que o identifica no resto do sistema, e um nodo, como é definido na secção 3.4.2 . O *bitstream* pode ser alterado em qualquer altura pelo processo-dono. Após a utilização o DRM pode ser removido ou libertado, permitindo que seja utilizado por outros processos.

Os parâmetros de utilização de cada DRM são configuráveis, podendo-se definir as dimensões dos vectores de leitura e escrita para cada operação ou as suas configurações internas.

O DRD (*Dynamic Reconfigurable Driver*) tem um papel transversal ao controlar todos os acessos à porta de reconfiguração e às áreas reconfiguráveis. Cada área reconfigurável é representada por uma PRA, que contém os dados do seu estado e configuração. Durante a operação de um DRM, a área reconfigurável está bloqueada, sendo imediatamente libertada no final. O ICAP funciona por *bitstreams*, estando disponível após concluir uma reconfiguração.

3.4 Interface com o sistema

Cada DRM é representado por um ficheiro na pasta */dev*, normalmente designado por *char device*. É através deste ficheiro que se comunica e controla o módulo reconfigurável, tal como se define na secção 3.4.2.

O DRD é controlado através de uma interface própria que utiliza o sistema de ficheiros *sysfs* [34]. Este sistema de ficheiros,que reside em memória virtual, é criado durante o arranque do sistema não guardando o histórico de sessões anteriores. Neste sistema cada método e definido por um ficheiro, tal como se encontra explicado na secção 3.4.1.

Alteração da interface durante o desenvolvimento

Inicialmente o controlador foi desenvolvido utilizando apenas a interface com o *sysfs*. Esta abordagem permite agregar no mesmo "sítio" a informação e o controlo do sistema. Desta forma o seu funcionamento é mais simples e intuitivo. Além disso, as funcionalidades estão disponíveis através de utilização de qualquer biblioteca que permita navegar por pastas, ler e escrever em ficheiros. A utilização mais simples em sistemas de desenvolvimento poderia ser feita em BASH, a linha de comandos do GNU/Linux, utilizando comandos como *cp*, *echo*, *cat* ou *ls*.

Apesar da simplicidade permitida, o *sysfs* não dispõe dos mecanismos de controlo de acessos necessários ao sistema. Assim, utiliza-se o *sysfs* para disponibilizar as informações do sistema e adicionar novos módulos e um *char device* para aceder a cada DRM.

3.4.1 Interface através do *sysfs*

O modelo de funcionamento através do *sysfs* disponibiliza um ficheiro virtual para cada operação. A árvore de pastas e ficheiros pretende ser simples e intuitiva, facilitando o desenvolvimento de novas aplicações. Nas secções seguintes descreve-se detalhadamente a interface, indicando exemplos de funcionamento utilizando a consola.

3.4.1.1 Raiz do sistema: */sysfs/drd/*

A raiz do sistema é a pasta */sysfs/drd/* que é criada na inicialização do *driver*. Nesta pasta são cridas três directorias, uma para a porta ICAP, outra para agrupar as áreas reconfiguráveis e outra para agrupar o DRMs. São disponibilizados quatro ficheiros que implementam as funcionalidade descritas na tabela 3.1.

O ficheiro *active* permite fazer uma consulta rápida pelos DRMs activos, indicando também a área onde estão configurados. O estado de uma PRA pode ser *free*, *busy* ou *lock* dependendo se não tem DRM, se está um DRM configurado inactivo ou se tem um DRM a processar informação, respectivamente, como exemplificado a seguir.

Tabela 3.1: Funções disponíveis em */sys/drd*.

Nome	Funcionalidade	Acesso	Exemplo
add	Adiciona um novo DRM vazio	E	*$ echo "DRM_NAME" > add*
remove	Remove um DRM	E	*$ echo "DRM_NAME" > remove*
active	Devolve o nome dos DRMs activos	L	*$ cat active*
state	Devolve o estado das PRAs	L	*$ cat state*

```
$ cat state
PRA dock0 is free.
```

A criação de um novo DRM inicializa uma estrutura na pasta */sys/drd/drm* com o nome do módulo. Durante o processo de inicialização são atribuídos um *major device number* (MAJOR) e um *minor device number* (MINOR), necessários ao *char device* de acordo com a secção 3.4.2.

3.4.1.2 Estado do ICAP: */sysfs/drd/icap*

Em */sysfs/drd/icap* pode-se verificar o estado interno da porta ICAP, através dos ficheiros definidos na tabela 3.2.

Tabela 3.2: Funções disponíveis em */sys/drd/icap*.

Nome	Funcionalidade	Acesso	Exemplo
reset	Reset	E	*$ echo 1 > reset*
status_register	Devolve o valor do *status_register*	L	*$ cat status_register*

3.4.1.3 Áreas Reconfiguráveis: */sysfs/drd/pra*

Cada área reconfigurável é definida na árvore do dispositivo, como é demonstrado na secção A.1.1. É criada uma pasta para cada uma, estando os seus atributos disponíveis para consulta de acordo com a tabela 3.3.

Tabela 3.3: Funções disponíveis em */sys/drd/praX*.

Nome	Funcionalidade	Acesso	Exemplo
size	Devolve a dimensão da PRA	L	*$ cat size*
state	Devolve o estado das PRAs	L	*$ cat state*

A dimensão da PRA está no formato *SLICE_XxminYymax:SLICE_XxmaxYmin*, tal como é definido no ficheiro de restrições (**.ucf*). O estado segue o formato descrito na secção anterior, acrescentado o nome do DRM se for o caso.

3.4.1.4 DRM - Módulos reconfiguráveis: */sysfs/drd/drm*

Cada DRM tem a sua própria pasta, agregando os métodos respectivos, com o nome do módulo. A interface expõem as funções definidas pela tabela 3.4:

Tabela 3.4: Funções disponíveis em */sys/drd/drm/drm_name*.

Nome	Funcionalidade	Acesso	Exemplo
state	Retorna o estado do módulo	L	*$ cat state*
chardevice	Retorna o nome e os números do *char device*	L	*$ cat chardevice*

3.4.2 Interface através do *char device*

Após a inicialização do módulo é necessário criar um *char device* na pasta */dev* com o MAJOR e MINOR definidos na interface em 3.4. Para o núcleo GNU/Linux estes números definem perfeitamente o nodo, não podendo existir números duplicados. O controlador utiliza rotinas dinâmicas de modo a utilizar números livres, não sendo por isso possível definir na imagem do úcleo os nodos correspondentes.

Estão definidos dois modos de acesso ao *driver*. O primeiro, que é definido por omissão, permite definir o *bitstream* do DRM. O segundo modo permite comunicar com DOCKI32O32, estando o *bitstream* correspondente configurado na FPGA. Através da função *ioctl()* é possível alterar o modo de funcionamento.

A tabela 3.5 indica as funcionalidades disponíveis através da função *ioctl()*, referenciando as constante definidas no ficheiro *drd_ioctl.h*. De acordo com a constante utilizada pode-se definir um valor através parâmetro *arg* da função *ioctl()* ou consultar o valor de configuração. O ficheiro *drd_ioctl.h* define, ainda, as constantes utilizadas para definir o estado e o modo de funcionamento do DRM.

Tabela 3.5: Funções disponíveis através da função *ioctl()*.

Valor	Funcionalidade	Erros
DRD_IOCRESET	Faz um *reset* ao DRM	
DRD_IOCSMODE	Define o modo de funcionamento	*EINVAL*
DRD_IOCQCLEAR	Apaga o *bitstream*	*EEXIST*
DRD_IOCSWRITESIZE	Define a dimensão do vector de escrita	
DRD_IOCSREADSIZE	Define a dimensão do vector de leitura	
DRD_IOCSSTATE	Define o estado do DRM	*EINVAL*
DRD_IOCGSTATE	Devolve o estado do DRM	*EINVAL*

As dimensões dos vectores de leitura ou escrita representam o número de valores de 32 *bits* que serão lidos ou escritos. A comunicação com os DRMs é sempre de 32 *bits* pelo que tanto as aplicações como os módulos devem seguir este padrão.

Os erros possíveis encontram-se definidos pela biblioteca standard GNU/Linux *errno.h*. Procurou-se utilizar valores adequados à função pretendida de modo a simplificar o desenvolvimento de aplicações.

3.5 Resumo

Neste capítulo apresentou-se o modelo de funcionamento do controlador de dispositivo, indicando-se alguns exemplos. Cada DRM é representado por um nodo na pasta */dev* e por uma pasta dentro da directoria */sys/drd/drm*, onde se pode consultar o seu estado.

No capítulo 4 descreve-se a implementação das funcionalidades disponíveis, definindo-se os métodos utilizados e justificando-se as opções tomadas no projecto.

Capítulo 4

Implementação e resultados

O desenvolvimento do *device driver* não correu como previsto. Neste momento, em que a tese é escrita, estão a funcionar os mecanismos de comunicação com o ICAP e com a OPB Dock. Foram implementados, também, as funções do controlo dos *bitstreams*, ainda que num estado experimental.

Este capítulo pretende detalhar os conhecimentos adquiridos ao longo do projecto, possibilitando assim a sua evolução em trabalhos futuros. Serão apresentados os aspectos práticos da implementação detalhando as funções e bibliotecas utilizadas

4.1 Sistema de desenvolvimento

4.1.1 Descrição do *hardware*

A configuração do *hardware* utilizado segue o modelo indicado por Miguel Silva em [35]. O sistema foi implementado numa placa com uma FPGA Xilinx XC2VP7 [15], que inclui um processador PowerPC 405 embutido, e 32 MB de memória estática (SRAM). O sistema foi criado utilizando o Xilinx EDK, que contém os módulos necessários (IP Cores) para configurar a base, tal com é descrito no anexo A.

O sistema está representado na figura 4.1. O processador comunica com o resto do sistema através do barramento de 64 *bits* PLB (*Processor Local Bus*). A memória interna da FPGA e a memória SRAM externa são acedidas através de controladores próprios ligados ao PLB. O OPB (*On-Chip Peripheral Bus)* liga os restantes periféricos ao PLB mas utilizando um barramento de 32 *bits* de modo a poupar os recursos.

A comunicação com o exterior é realizada utilizando de uma porta séria RS232 (*UART Controler*), que após o arranque do GNU/Linux permite o controlo e monitorização do sistema através de uma linha de comandos.

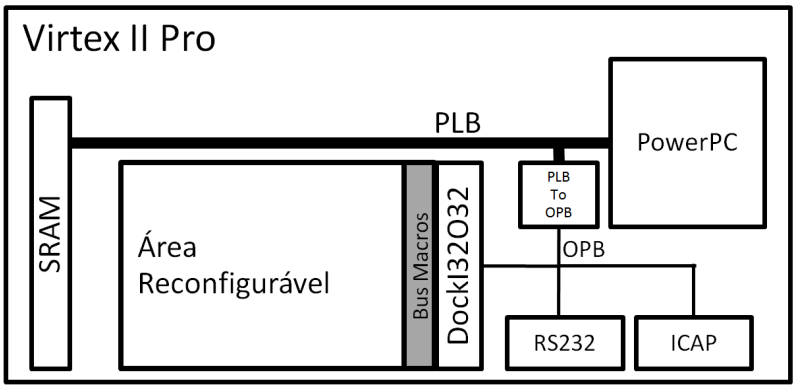


Figura 4.1: Arquitectura geral do *hardware* do sistema de desenvolvimento (baseado em [35]).

Controlador OPB HWICAP

O acesso à configuração interna da FPGA é realizado através do controlador da porta ICAP, o OPB HWICAP [26]. O *reset* e a consulta do registo de estado são efectuados através de endereços próprios.

As operações com o controlador são de 32 *bits* (uma *word*) através um *buffer* interno de 512 *words*, quer de dados quer de comandos. O controlador está, normalmente, num estado dessincronizado à espera da palavra de sincronismo para processar as tramas de informação. Cada trama é constituída por um comando seguido dos dados correspondentes [16]. Um *bitstream* bem construído, como descrito no anexo B, contém toda a informação necessária.

Área reconfigurável

O acesso aos módulos reconfiguráveis é realizado através de uma interface de 32 *bits* ligada ao OPB, normalmente designada por *OPB Dock*. Esta ligação pode ser implementada de diferentes formas, permitindo incorporar funcionalidades especificas. A sua funcionalidade base é permitir a comunicação com os módulos reconfiguráveis através das *Bus Macro*, de modo a garantir que ligações ocorrem sempre na mesma localização.

O modelo utilizado, a DOCKI32O32, é a sua implementação mais simples, ou seja, define um barramento com um canal de 32 bits em cada sentido e 8 bits de controlo. O modelo de utilização prevê a implementação de dois endereços de configuração, um para a PRA outro para o DRM, que não foram implementados fisicamente apesar de o *driver* os suportar.

4.1.2 Descrição do *software*

O sistema arranca a versão 2.6.27 do núcleo GNU/Linux disponibilizado pela Xilinx em [11]. A imagem do núcleo foi aumentada com um sistema de ficheiros base, que

inclui a versão 1.12.0 do *BusyBox* [36], alguns ficheiros de configuração essenciais e as bibliotecas necessárias. O anexo A detalha este processo definindo passo a passo a construção de um sistema de ficheiros.

O *BusyBox* implementa um conjunto de programas essenciais a qualquer sistema UNIX mas optimizados para sistemas embutidos. Foram adicionados os programas básicos para obter um sistema funcional como uma linha de comandos, um editor de texto ou as aplicações para navegar pelas ficheiros e pastas.

4.2 Adicionar o controlador à árvore do núcleo

Os controladores nos sistemas GNU/Linux são compilados juntamente com núcleo. Os controladores podem ser compilados embutidos no núcleo ou em forma de módulo, independentes. Para adicionar o controlador à árvore do núcleo é necessário editar os ficheiros *Makefile* e *Kconfig*. O primeiro contém a informação a passar ao compilador enquanto o segundo permite configurar o núcleo através de menus de selecção.

Neste projecto, o código foi desenvolvido na directoria *drivers/drd*, incluindo as bibliotecas e ficheiros de compilação.

O menu de configuração do núcleo está acessível no formato de linha de comandos (*make config*) ou com interface gráfico (*make xconfig*). Para incluir novas opções no processo editou-se o ficheiro *Kconfig* na pasta *drivers* e adicionou-se um novo menu, definido no ficheiro *drivers/drd/Kconfig* tal como demonstram os exemplos seguintes:

```
source "drivers/drd/Kconfig"
```

```
menu "Dynamic Reconfigurable Device"

config DRD
        depends on XILINX_VIRTEX_II_PRO
        tristate "Dynamic Reconfigurable Device"
        help
                This option enables support for Dynamic Reconfigurable Device.
endmenu
```

O menu permite configurar variáveis que são lidas pelas *Makefiles*, neste caso a variável *DRD*. As variáveis definidas como *tristate* permitem que esse controlador seja compilado embutido no núcleo ou como módulo. O processo permite definir dependências através do parâmetro *depends on*, evitando possíveis erros de configuração, como neste caso em que o driver é especifico para a Virtex II Pro. É normal utilizar este método de verificação para garantir a coerência dentro do núcleo, evitando verificações repetitivas durante a execução.

O exemplo seguinte demonstra o caso em que o código do controlador está na pasta *drivers/drd*, incluído através da *Makefile* da pasta *drivers*. A segunda *Makefile* indica que o controlador *sysdrd* é constituído por vários executáveis.

```
obj-$(CONFIG_DRD)       += drd/
```

```
obj-$(CONFIG_DRD)       += sysdrd.o
sysdrd-y := drd.o buffer_icap.o
```

Os controladores configurados como módulos podem ser compilados isoladamente como executáveis independentes (**.ko*) e incluídos no sistema de ficheiros definido em A.2.2.

Utilizando módulos independentes é possível analisar o sistema durante as várias fases. Desta forma pode-se verificar os recursos consumidos na inicialização do módulo ou garantir que os recursos são totalmente libertados na sua remoção, seguindo as boas práticas da programação.

4.3 Análise do funcionamento do controlador de dipositivo

O código de um controlador de dipositivo só é executado em resposta a uma determinada chamada ao sistema. No entanto, a sua estrutura global está bem definida em três momentos: inicialização, funcionamento e remoção.

A inicialização ocorre quando o módulo é carregado no núcleo. Nesta fase deve-se inicializar as variáveis e reservar os recursos necessários como os espaços em memória ou o acesso aos endereços dos barramentos. Se alguma etapa do processo não for bem sucedida deve-se libertar os recursos pedidos. Na remoção libertam-se os recursos pedidos, deixando o sistema limpo. As secções 4.3.1, 4.3.2 e 4.3.3 detalham as funções e métodos utilizados.

A comunicação com os dispositivos é realizada através dos métodos descritos na secção 4.3.4. Os métodos de comunicação entre o núcleo e o utilizador são explicados nas secções 4.3.5, 4.3.6 e 4.3.7.

4.3.1 Estrutura base

A estrutura de um controlador não é rígida, variando sempre de acordo com as funcionalidades pretendidas. No entanto existem métodos essenciais como a sua inicialização ou remoção. Os cabeçalhos *init.h* e *module.h* providenciam esses métodos (*module_init* e *module_exit*).

O código de demonstração seguinte contém as funções de inicialização e remoção do controlador. Durante a inicialização regista-se os controladores para interagir com o

OPB HWICAP e com as áreas reconfiguráveis, como será detalhado a seguir, e cria-se o sistema de ficheiros e pastas necessário, descrito na secção 3.4.

```
#include <linux/init.h>
#include <linux/module.h>
```

```
static int __init drd_module_init(void)
{       ...
        of_register_platform_driver(&drd_icap_of_driver);
        of_unregister_platform_driver(&drd_icap_of_driver);
        ...             }
module_init(drd_module_init);
```

```
static void __exit drd_module_cleanup(void)
{       ...
        of_unregister_platform_driver(&pra_of_driver);
        of_unregister_platform_driver(&icap_of_driver);
        ...             }

module_exit(drd_module_cleanup);
```

Os módulos utilizam identificadores próprios, como os exemplificados a seguir, que embora não tenham uma função especifica são considerados boas práticas [37] ao indicarem um conjunto mínimo e essencial de informação.

```
MODULE_AUTHOR("Bruno␣Monteiro");
MODULE_DESCRIPTION("Dynamic␣Reconfigurable␣Device␣For␣Virtex␣II␣PRO");
MODULE_LICENSE("GPL");
```

OpenFirmware

A tendência actual dos controladores de dispositivos é serem escritos com recursos a funções *of* (*OpenFirmware* [38]). Os cabeçalhos *of_device.h* e *of_platform.h* disponibilizam os métodos para obter informação dos dispositivos, no entanto o controlador deve seguir uma estrutura fixa como a demonstrada a seguir.

```
#include <linux/of_device.h>
#include <linux/of_platform.h>

static int __devinit drd_icap_of_probe
        (struct of_device *op, const struct of_device_id *match)
{       ...             }
```

```
static int __devexit drd_icap_of_remove(struct of_device *op)
{       ...     }

static const struct of_device_id __devinitconst drd_icap_of_match[] = {
        { .compatible = "xlnx,opb-hwicap-1.00.b",},
        {},
};
MODULE_DEVICE_TABLE(of, drd_icap_of_match);
```

```
static struct of_platform_driver drd_icap_of_driver = {
        .owner = THIS_MODULE,
        .name = "drd_icap",
        .match_table = drd_icap_of_match,
        .probe = drd_icap_of_probe,
        .remove = __devexit_p(drd_icap_of_remove),
        .driver = {
                .name = "drd_icap",
        },
};
```

Quando um *of_driver* é registado, o sistema verifica a árvore de dispositivos à procura de uma, ou mais, correspondência com a *match_table*. Esta define o dispositivo para o qual este controlador foi escrito. Neste caso, o parâmetro de compatibilidade funciona como um identificador inequívoco. As funções de inicialização e remoção (*probe* e *remove*) funcionam da mesma forma que as do módulo ao garantirem que o *driver* detém os recursos necessários e está num estado perfeitamente definido.

4.3.2 Ligação com a árvore de dispositivo

A árvore do dispositivo [39] contém as informações do *hardware* do sistema. É através dela que se definem os barramentos e as ligações existentes, do ponto de vista do processador. A árvore do dispositivo é, tal como o nome o sugere, uma definição hierárquica dos dispositivos do sistema e das suas ligações. A sua construção é explicada no anexo A.1.1.

```
dockO32I32_0: docko32i32@40500000 {
        compatible = "xlnx,docko32i32-2.00.a";
        reg = < 0x40500000 0x10000 >;
        xlnx,family = "virtex2p";
        drd,slice_size = "SLICE_X8Y79:SLICE_X33Y22";
        drd,ramb_size = "RAMB16_X1Y8:RAMB16_X2Y1";
        } ;
```

```
String slices = of_get_property(op->node, "drd,slice_size", NULL);
```

Pela descrição da *dockO32I32*, a interface com a área reconfigurável, pode-se verificar a versão utilizada, através do parâmetro *compatible*, ou os endereços do barramento que lhe estão atribuídos, através do parâmetro *reg*. Este está no formato "< endereço_inicial tamanho >".

A localização da área reconfigurável é definida pelos recursos que incorpora, normalmente definidos através de *slices* e *brams*. Estas caracteristicas são passadas para o núcleo através dos parâmetros *drd,slice_size* e *drd,ramb_size* para posteriormente permitir a validação dos *bitstreams*. As consultas dos parâmetros podem ser realizadas individualmente como no exemplo, tendo em atenção o formato do campo pretendido.

4.3.3 Alocação de recursos

A alocação de recursos consiste em reservar espaço na memória virtual para uma determinada função. As páginas da memória estão organizadas em sectores de 4096 *bytes* e podem ser reservadas através da função *(u8 *) get_zeroed_page(GFP_KERNEL)*.

Os endereços dos periféricos não estão acessíveis directamente, uma vez que se referem ao espaço da memória virtual. Assim, é necessário remape-á-los para endereços disponíveis através da função *ioremap()*. A seguir apresenta-se um exemplo prático em que se consulta árvore do dispositivo, reserva-se a memória e posteriormente remapei-a-se para endereços acessíveis. O restante acesso ao dispositivo ocorre através do *base_address* como se pode verificar na secção 4.3.4.

```
struct of_device *op
struct resource res;
void __iomem *base_address;

of_address_to_resource(op->node, 0, &res);
request_mem_region(res->start, res->end - res->start + 1);
base_address = ioremap_nocache(res->start, res->end - res->start + 1);
```

Os recurso requisitados devem ser libertados quando já não forem necessrios. Embora os dispositivos do projecto sejam especificos, deve-se preservar quer a flexibilidade do código desenvolvido quer a estabilidade do sistema para futuras aplicações. O exemplo seguinte indica as funções para libertar os recurso do exemplo anterior.

```
iounmap(base_address);
release_mem_region(base_address, res->end - res->start + 1);
```

4.3.4 Interface com os barramentos

O acesso ao barramento OPB, onde estão ligados quer a DOCKI32O32 quer o OPB HWICAP, é realizado através das funções *in_be32* e *out_be32* disponibilizadas pelo cabeçalho *io.h*.

A função de leitura retorna o valor lido no endereço seleccionado enquanto que a função de escrita não retorna qualquer valor, escrevendo o valor de *data* no barramento. A troca de dados ocorre com inteiros sem sinal de 32 *bits* (*u32*). O endereço base foi analisado na secção anterior enquanto que o *offset* depende do dispositivo, que define uma funcionalidade para cada endereço que lhe está atribuido. Normalmente cada dispositivo tem uma gama de endereços superior à que necessita. Os endereços utilizados pelo OPB HWICAP estão definido em [26].

```
#include <linux/io.h>

(u32) in_be32(base_address + offset);
(void) out_be32(base_address + offset, data);
```

Por norma os *offsets* são fixos e estão definidos como constantes no código, adicionando-se os respectivos comentários para facilitar a análise.

4.3.5 Interface com o *sysfs*

A interface com o *sysfs* é realizada através das funções definidos pelos cabeçalhos *kobject.h* e *sysfs.h*. Os excertos de código a seguir apresentados exemplificam a interacção com este sistema de ficheiro. Foram omitidos vários pormenores de implementação de modo a manter a explicação simples e directa.

```
#include <linux/kobject.h>
#include <linux/sysfs.h>
```

No *sysfs* cada ficheiro é representado por um *attribute*. Para cada *attribute* pode ser definido uma função de leitura e outra de escrita, dependendo das permissões dadas.

As permissões são definidas no formato *standard* dos sistemas UNIX, ou seja, através de quatro dígitos em formato octal. Cada *bit* do octal define, com o valor lógico '1', permissões para ler, escrever ou executar. As permissões para o dono, o grupo ou restantes utilizadores são definidas por um octal cada uma. Neste exemplo, são apresentado dois atributos, um com permissões de esctrita ('0222') e outro só com permissões de leitura ('0444').

```
static ssize_t drd_remove_drm_store(struct drd *drdev,
        struct drd_attribute *attr, const char *buf, size_t count)
{       ...             }

static struct drd_attribute drd_remove_drm_attribute =
        __ATTR(remove, 0222, NULL, drd_remove_drm_store);

static ssize_t drd_list_pra_status_show(struct drd *drdev,
        struct drd_attribute *attr, char *buf)
{       ...             }

static struct drd_attribute drd_list_pra_status_attribute =
        __ATTR(state, 0444, drd_list_pra_status_show, NULL);
```

A estrutura base para o *sysfs* é o *kobject* que pode agregar mais do que um *attribute*, funcionando aproximadamente como uma directoria. Para agrupar mais do um *kobjet* utiliza-se um *kset*, que se assemelha a uma sub-pasta.

```
struct drd {
        struct kobject kobj;
        struct kset *drd_modules;
        struct kset *drd_pra;
        ...
}; #define to_drd(x) container_of(x, struct drd, kobj)
```

```
static struct attribute *drd_module_group_attrs[] = {
        &state_attribute.attr,
        &bitstream_attribute.attr,
        NULL,   /* need to NULL terminate the list of attributes */
};

static struct kobj_type drd_module_ktype = {
        .sysfs_ops = &drd_module_sysfs_ops,
        .release = drd_module_release,
        .default_attrs = drd_module_group_attrs,
};
```

Neste projecto existe um *kobject* principal (*/sys/drd*) que ramifica, através de dois *ksets*, para os DRMs e as PRA. Agrega, também, os *attribute* para gerir os DRMs, como definido na secção 3.4.1.1, e um *kobj* para as operação com o OPB HWICAP. Cada *kobj* está associado a uma estrutura que contém os dados necessários à implementação do modelo proposto em 3.4. Cada estrutura é independente e identificada pelo seu *kobject*, por exemplo, cada DRM tem uma estrutura que contém dados como o seu *bitstream* ou o estado actual desse módulo.

O exemplo a seguir demonstra as funções utilizadas durante a inicilização para criar a pasta */sys/drd* e as subpastas para agregarem as áreas e os módulos reconfiguráveis.

```
kobject_init_and_add(&drd->kobj, &drd_ktype, NULL, "drd");
drd->pra_list = kset_create_and_add("pra", NULL, &drd->kobj);
drd->drm_list = kset_create_and_add("drm", NULL, &drd->kobj);
```

4.3.6 Interface com o *char device*

Após a inicialização do DRM, o núcleo cria um nodo que o representa. Este nodo é definido por uma MAJOR e um MINOR. O MAJOR representa o controlador enquanto que o MINOR identifica o DRM para o controldador. O processo de atribuição deste valores é automático, estando o resultado disponível para leitura através do ficheiro *chardevice* na pasta do DRM. Para criar um *char device* na pasta */dev* pode-se utilizar o comando: *$ mknod -c /dev/drm%d MAJOR MINOR*, em que o inteiro é definido pelo MINOR. Como alternativa o comando *$mdev -s* pesquisa os *sysfs* e cria todos os nodos necessários, e os desnecessários também mas tal não "pesa" no sistema.

O *driver* desenvolvido define as operações realizadas pelas funções de acesso ao ficheiro do *char device*, tal como demonstra a estrutura representada a seguir.

```
static struct file_operations drm_fops = {
        .owner = THIS_MODULE,
        .write = drm_write,
        .read = drm_read,
        .open = drm_open,
        .release = drm_close,
        .ioctl = drm_ioctl,             };
```

Abrir e fechar o *char device*

Ao abrir e ao fechar o ficheiro são executadas as rotinas normais para garantir que o ficheiro apenas é utilizado por um processo. Desta forma garante-se que a partilha do DRM, se acontecer, é porque processo o permitiu ao partilhar o descritor do ficheiro e espera-se que o programador tome as devidas precauções para evitar conflitos.

Salienta-se o facto de no momento de abertura se passar o DRM para a estrutura do ficheiro, permitindo que seja utilizado posteriormente pelas funções de leitura e escrita.

```
static int drm_open(struct inode *inode, struct file *file)
{...
        struct drm *drvdata;
        drvdata = container_of(inode->i_cdev, struct drm, cdev);
        file->private_data = drvdata;
        ...}
```

```
static int drm_close(struct inode *inode, struct file *file)
{       ...             }
```

Função de Leitura e Escrita

Após a validação das permissões pela função de abertura, as funções de leitura e escrita implementam as funcionalidades pretendidas. O DRM pode estar em dois estados possíveis, o de configuração do *bitstream* e o de acesso à *DOCKI32O32*. O controlador verifica, em cada acesso, que a PRA atribuída ao DRM é válida. Caso não o seja, o controlador procura uma PRA livre, como se exemplifica na secção 4.3.9.

```
static ssize_t drm_write(struct file *file, const char __user *buf,
        size_t count, loff_t *ppos)
{
        struct drm *module = file->private_data;
        u32 *kbuf = (u32 *) __get_free_page(GFP_KERNEL);

        copy_from_user(kbuf, buf, count)
        if (module->mode == DRM_ICAP_MODE){
                ...
                status = add_to_bitstream (module->bits, kbuf, count);
        }else{
                ...
                module->dock = get_free_pra (drd);
                status = (drm_write_buffer (module->dock, kbuf, status))<<2;
        }       ...
        return status;
}
```

O exemplo anterior, da função de escrita, e o seguinte, da função de leitura, demonstram alguns dos passos executados pelo controlador, omitindo-se os processos de verificação e validação de dados.

```
static ssize_t drm_read(struct file *file, char __user *buf,
        size_t count, loff_t *ppos)
{
        struct drm *module = file->private_data;
        u32 *kbuf = (u32 *) __get_free_page(GFP_KERNEL);
        ...
        module->dock = get_free_pra (drd);
        status = (drm_read_buffer (module->dock, kbuf, module->read_size))<<2;
        copy_to_user(buf, kbuf, status);
        ...
        return status;
}
```

Os espaços de memória do núcleo e do utilizador são diferentes e não devem ser misturados, dando origem a falhas de segurança ou comportamentos imprevistos. Para os evitar, os dados são copiados de uma zona para outra através das funções *copy_from_user()* e *copy_to_user()*.

A troca de dados entre as duas regiões é contabilizada em *bytes* enquanto que o acesso aos dispositivos em *words*. Assim, o controlador converte o valores, procedendo a ajustes se necessário e mantendo a integridade dos dados. Um exemplo disso é a função que guarda o *bitstream* na secção 4.3.8.

4.3.7 Interface através da função *ioctl()*

A função *ioctl()* permite executar funções variadas, suprimindo as limitações do acesso sequencial das funções de leitura e escrita. Esta função recorre a um comando que pode ser acompanhado de um argumento complementar. Os valores de retorno são utilizados para devolver o valor pedido pelos comandos de leitura.

O valor dos comandos é definido através de macros disponibilizadas pelo cabeçalho *ioctl.h*. Estas macros permitem definir o tipo de comando como sendo escrita, leitura, leitura e escrita ou sem parâmetros, consoante o caso que mais se adequar. A base destes números é uma constante normamente definida pela MAJOR do controlador, no entanto como este é dinâmico escolheu-se um valor.

```
#define MAGIC_NUM 100
#define DRD_IOCRESET            _IO(MAGIC_NUM, 0)
#define DRD_IOCSMODE            _IOW(MAGIC_NUM, 1, int)
#define DRD_IOCQCLEAR           _IOR(MAGIC_NUM, 2, int)
...
```

A implementação da função *ioctl()* é um simples *case* que chama a função correspondente a cada comando, como se pode verificar pelo exemplo seguinte. Para evitar erros os comandos são devidamente validados.

```
static int drm_ioctl(struct inode *inode, struct file *file,
        unsigned int cmd, unsigned long arg)
{
        struct drm *module = file->private_data;
        if (_IOC_TYPE (cmd) != MAGIC_NUM)
                return -ENOTTY;
        switch (cmd) {
                case DRD_IOCRESET: ...  break;
                case DRD_IOCSMODE: ...  break;  ....
                default:
                        return -ENOTTY;
        }       ...
}
```

4.3.8 Armazenar um *bitstream*

Os sectores de memória estão organizados em blocos de 4096 *bytes* (PAGE_SIZE), dentro do núcleo. No entanto os *bitstreams* de teste ocupam mais do que este limite, e não é viável utilizar ficheiros no espaço do núcleo. Assim, criou-se um sistema de listas ligadas para guardar o *bitstream*, permitindo diminuir o tempo de transição entre DRMs.

Os dados do *bitstream* são guardados de forma sequencial, preenchendo-se os *bytes* livres da última *word* com zeros para não interferir com a reconfiguração da FPGA

4.3.9 Pesquisa por um DRM ou PRA

O núcleo GNU/Linux têm um sistema próprio de listas ligadas, definidas pelo cabeçalho *stat.h*. Cada *Kset* utilizado no *sysfs* inclui uma desta listas, que é utilizada pelo controlador para fazer pesquisas.

```
struct pra * pra_tmp;

list_for_each_entry(pra_tmp, &drdev->pra_list->list, kobj.entry){
        if (strcmp(pra_tmp->state, PRA_STATE_FREE) == 0) return pra_tmp;
}
list_for_each_entry(pra_tmp, &drdev->pra_list->list, kobj.entry){
        if (strcmp(pra_tmp->state, PRA_STATE_BUSY) == 0){
                set_drm_free (pra_tmp->drm);
                return pra_tmp;
        }
}
```

O exemplo anterior demontra uma pesquisa por uma PRA livre. Primeiro verifica-se se existe um região livre e só depois se liberta uma PRA que não esteja a ser utilizada naquele momento. Por motivos temporais já expostos, esta implementação não contempla a verificação do *bitstream*.

4.4 Resultado experimentais

Para observar o comportamento do sistema foram desenvolvidos dois módulos reconfiguráveis simples, ou seja, enquanto um incrementava o valor escrito o outro decrementava. Com estes módulos foi possível testar diferentes *bitstreams*, observando a integridade do sistema. Cada *bitstream* do módulo reconfigurável tem um dimensão 81610 *Bytes*, correspondendo à configuração da PRA com uma largura de 26 *slices*.

O desenvolvimento dividiu-se em duas fases, como explicado na secção 3.4. A seguir descrevem-se os resultados obtidos nas diferentes fases. Durante o desenvolvimento foram realizados testes funcionais, por questões de observabilidade. Agora, no final de

cada fase, mediu-se o desempenho do sistema através do processo descrito nas próximas secções.

4.4.1 Resultados utilizando apenas o *sysfs*

Os testes realizados nesta fase, através do programa *time*, consistiam em medir o tempo gasto pelo sistema para executar uma determinada aplicação. Como é um programa executado através da BASH não é muito optimizado, no entanto, serve os propósitos principais que foram obter uma estimativa da ordem de grandeza do tempo necessário para as diversas fases.

O teste foi dividido em quatro fases. Na primeira verificou-se o tempo gasto pelas funções de inicialização do *driver*. A segunda consistiu em passar o *bitstream* do módulo reconfigurável parcialmente para o *Kernel Space*. A activação do DRM permite medir o tempo gasto pela reconfiguração da área dinâmica através do OPB HWICAP. A última medição permite verificar o tempo gasto para escrever um inteiro e ler outro. As funções utilizadas são mostradas a seguir.

```
BITSTREAM=/home/base_rec_area_inc_partial.bit
DRDMODULE=/sys/drd/drm/add

# Inicializar o driver
time modprobe sysdrd

# Inicializar um DRM
time cp ${BITSTREAM} ${DRDMODULE}/bitstream

# Activar um DRM
time echo 1 > ${DRDMODULE}/state

# Utilizacao unica
time sh /home/time_io.sh
```

Resultados obtidos

Os resultados dos testes, descritos anteriormente, estão demonstrados na tabela 4.1. Os valores tabelados estão em segundos. O processo foi repetido duas vezes, no entanto os resultados foram idênticos.

4.4.2 Testes finais

A versão actual do controlador implementado utiliza um *char device* para controlar e comunicar com o DRM. Pode-se consultar as informações sobre o sistema e inicializar

Tabela 4.1: Resultado experimentais utilizando o *sysfs*

Teste	**Tempo**		
	Total (*s*)	**Utilizador (*s*)**	**Sistema (*s*)**
Inicializar o *driver*	0.43	0.02	0.39
Inicializar um DRM	0.05	0.01	0.03
Activar um DRM	0.06	0.01	0.04
Utilização única	0.05	0.02	0.02

um DRM através da interface definida *sysfs*. Os testes realizados permitem obter um valor aproximado do tempo gasto pelas operações envolvidas.

Método de medição

O testes realizados estão divididos em duas partes. Na primeira parte utilizou-se o comando *time*, como nos testes descritos anteriormente. Esta parte consistia em medir o tempo consumido para inicializar o controlador e um DRM, atribuindo-lhe um MAJOR e um MINOR. A criação do nodo na pasta */dev* não é automática, utilizando-se por isso o comando *mdev -s* que pesquisa a estrutura do *sysfs* e cria todos os nodos referenciados. Esta solução não é óptima uma vez que cria vários nodos inúteis que, no entanto, não interferem com o desempenho do sistema.

Para a segunda parte do teste desenvolveu-se código específico para efectuar as medições, do qual se retirou o exemplo seguinte. Foram efectuadas medições nas várias operações possíveis com o DRM. A primeira consistiu em medir o tempo gasto para passar os *bitstreams* de testes do espaço do utilizador para o espaço do núcleo. Se seguida verificou-se o tempo consumido para configurar a PRA. Os restantes testes incidiram sobre as operações de leitura e escrita com o módulo reconfigurável, obtendo-se resultados para operações com apenas um inteiro e operações utilizando um vector de 1024 inteiros, que corresponde a uma página da memória.

```
clock_t init, end, diff;
struct tms tms_init, tms_end;
loop = 1000;
...
```

```
...
init = times(&tms_init);
for (i=0; i < loop; i++){
        load_bitstream (bits, file_device);
}
end = times(&tms_end);
print_time (loop, init, end, tms_init, tms_end);
...
```

Como as operações realizadas são muito rápidas, as medições efectuadas consideram um ciclo de 1000 operações calculando-se posteriomente a média por operação, como se pode verificar pelo exemplo anterior. Desta forma os resultados reflectem principalmente o tempo usado pela função de teste, excluindo o tempo gasto pelas próprias medições e pela devolução dos resultados para o utilizador.

Resultados obtidos

Como descrito anteriormente, as medições finais foram divididas em duas fases. A tabela 4.2 regista os resultados da fase de inicialização enquanto os resultado da fase operacional estão indicados na tabela 4.3, indicados em milissegundos.

Tabela 4.2: Resultado finais de inicialização

Teste	**Tempo**		
	Total (*s*)	**Utilizador (*s*)**	**Sistema (*s*)**
Inicializar o controlador	0.31	0.02	0.27
Criar um DRM	0.02	0.00	0.00
Criar nodos	3.56	0.76	2.79

Tabela 4.3: Resultado finais de operação

Teste	**Tempo**		
	Total (*ms*)	**Utilizador (*ms*)**	**Sistema (*ms*)**
Carregar um *bitstream*	7.43	0.33	7.11
Reconfigurar a PRA	5.91	0.01	5.90
Escrita de um valor	0.05	0.00	0.05
Leitura de um valor	0.05	0.00	0.05
Escrita de um vector (1024 valores)	0.26	0.00	0.25
Leitura de um vector (1024 valores)	0.33	0.01	0.34

4.4.3 Análise dos resultados

O tempo gasto pela inicialização do controlador é bastante superior ao necessário uma vez que se imprime na consola várias informações para o utilizador, através da função *printk*. Assim, é normal que demore mais tempo que as restantes que, propositadamente para a medição, não imprimem nada para a consola.

Os resultados obtidos utilizando apenas os métodos do *sysfs* são bastante elevados. A utilização de *scripts* através da BASH sempre foi conhecida pela flexibilidade e pelo custo associado. O programa *time* executa dois processos, um para executar a tarefa pretendida e outro para medir o tempo. Assim, o tempo de execução será superior ao real. Além disso apresenta uma resolução baixa que pode interferir com as medições realizadas. Assim, os resultados finais representam melhor o desempenho do sistema.

O tempo gasto pela aplicação *mdev* é bastante elevado quando comparado com os restantes. Como referido anteriormente não é a solução ideal, sendo aconselhado que as aplicações futuras futuras criem o *char device*, através da função *mknod()* após a consulta do MAJOR e MINOR na interface própria, definida na tabela 3.4.

Nos resultado finais, como esperado, verifica-se que a utilização de um programa em C é bastante mais rápida, reduzindo o tempo necessário em uma ordem de grandeza ou em várias no caso da comunicação com o DRM. A quase total utilização do tempo por parte do sistema era previsível, atendendo à natureza das operações realizadas.

O tempo necessário para passar um *bitstream* para o espaço do núcleo e o tempo gasto para reconfigurar a PRA são aproximados. O processamento necessário é semelhante para transferir a mesma informação, aproximadamente 80 KB do *bitstream*. No entanto, contrariamente ao esperado, a comunicação com o OPB-HWICAP utiliza menos tempo do que replicar informação na memória.

O tempo de escrita e de leitura é idêntico, como seria de esperar. Verifica-se que a utilização de vectores é vantajosa face à utilização de um único valor, não sendo o tempo gasto proporcional ao tamanho do vector. Para os valores utilizados verifica-se que o tempo médio utilizado por cada valor do vector é aproximadamente de 0,2 μs, bastante menor do que utilizando apenas um vector unitário. A dimensão do vector foi propositadamente escolhida para ocupar exactamente uma página da memória, o tamanho máximo do vector de comunicação possível entre o utilizador e o controlador. Esperando-se por isso uma maior optimização das funções realizadas pela núcleo para vectores com esta dimensão. Para vectores múltiplos deste, o tempo gasto em média por cada valor deve ser semelhante.

Os resultados obtidos são aceitáveis. Considerando a utilização do sistema para processar imagens capturadas a 25 FPS (*frames* por segundo), que tem 40 *ms* para processar cada uma, a reconfiguração da PRA ocupa apenas 6 *ms*, aproximadamente 15% do tempo da *frame*. Assim, seria viável a sua aplicação num sistema de monitorização que adapta-se a imagem a cada segundo, não sendo perceptível para o olho humano a breve interrupção realizada.

4.5 Funcionalidades por implementar

Como indicado no início do capítulo, o sistema não está completo. O trabalho desenvolvido focou-se em perceber os mecanismos subjacentes ao tema e na validação dos mesmos. Assim, foram desenvolvidas as funcionalidades mínimas, que permitissem validar o modelo proposto.

Existem, por isso, algumas funcionalidades que podem ser implementadas/melhoradas numa utilização futura.

Apesar do *bitstream* conter a localização, essa informação não é validada. Por enquanto deixa-se essa responsabilidade aos utilizadores mas para obter um sistema robusto seria necessário validar o *bitstream* do DRM logo após a sua passagem para o espaço do núcleo. A validação pode ser realizada separando o *bitstream* em *frames* e comparando o cabeçalho de cada uma com a informação da árvore do dispositivo.

Como o sistema ainda não é capaz de validar um *bitstream* não foi possível implementar a R^2, reconfiguração pontual de registos da FPGA. Seria necessário manipular o *bitstream*, conjugando-o com o trabalho desenvolvido por Miguel Silva [21, 24], para que tal fosse possível.

A estrutura do controlador prevê mais do que uma área reconfigurável, no entanto como não foram implementadas funções que permitam validar o posicionamento do *bitstream* esta funcionalidade não foi implementada. O sistema não consegue garantir o posicionamento do *bitsream* na PRA certa, sendo necessário a intervenção do utilizador.

Não foram desenvolvidas as técnicas de *readback*, essenciais para preservar o estado da configuração entre trocas de DRM com memória. A função *hwicap_get_configuration*, da biblioteca *buffer_icap.h*, devolve o conteúdo de uma *frame*, previamente seleccionada através de um comando de escrita.

4.6 Código desenvolvido

Em anexo a esta tese está um CD que contém os ficheiros código desenvolvido, quer do controlador quer dos módulos reconfiguráveis. Adicionaram-se também os ficheiros de configuração do compilador cruzado, do núcleo GNU/Linux, do *BusyBox* e do sistema de ficheiros, permitindo desta forma reproduzir o trabalho realizado.

Para auxiliar desenvolvimentos futuros foram incluidos no CD os *scripts* desenvolvidos para executar tarefas rotineiras de compilação. O CD contém também os *bitstreams* e a imagem do núcleo utilizados para configurar a FPGA.

4.7 Resumo

Neste capítulo descreveu-se os sistema de desenvolvimento utilizado, analisando os resultados obtidos. Foi analisada a interacção com o sistema, descrevendo-se os métodos utilizados para a comunicação e organização dos dispositivos. Analisou-se o sistema do ponto de visto do desenvolvimento de modo a explicar o trabalho realizado, indicando possíveis implementações para o que ainda não está funcional.

Capítulo 5

Conclusões e trabalho futuro

5.1 Conclusão

O objectivo principal deste trabalho, expresso na secção 1.1, foi desenvolver métodos que simplifiquem a utilização da reconfiguração dinâmica das FPGA. Esse objectivo não foi atingido completamente, ficando uma parte do desenvolvimentos por concluir como se verificou na secção 4.5.

A reconfiguração dinâmica das FPGAs é ainda uma tecnologia pouco conhecida e utilizada, por conseguinte falta documentação adequada. Apesar de existir esta encontra-se dispersa e não está vocacionada para iniciantes no tema. Descobrir documentação adequada ao objectivo deste trabalho foi uma das principais dificuldades encontradas uma vez que ou eram *papers* superficiais, sem detalhes de implementação, ou manuais demasiado técnicos.

A outra grande dificuldade sentida foi o comportamento experimental das ferramentas que lidam com a reconfiguração dinâmica. O processo não é muito expedito, sendo bastante rígido na sequência de passos. Por exemplo, no PlanAhead seleccionar uma janela fora de ordem pode fazer com que aplicação termine. Existe ainda um longo caminho para melhorar a usabilidade destas ferramentas.

Apesar das dificuldades sentidas, o esforço despendido foi bastante enriquecedor. Foram ultrapassadas várias barreiras conceptuais e compreendidos vários paradigmas dos sistemas embutidos, em que não havia experiência. Foi desenvolvido um sistema de raiz que inclui tarefas como a utilização de uma cadeia de compiladores cruzada, a compilação do núcleo GNU/Linux e dos seu módulos ou a inclusão de programas e ficheiros de configuração essenciais. Foram compreendidos e aplicados os métodos básicos de interacção entre o SO e o *hardware*. Estes métodos estão bem explícitos no código desenvolvido, permitindo a sua utilização em trabalhos futuros.

O trabalho realizado aproxima a utilização dos sistemas reconfiguráveis dinamicamente, colocando o ponto de partida de novos projectos mais próximos da sua meta. A

utilização de um SO para controlar o Sistema tem uma grande vantagem, com um custo reduzido. Os resultados dos tempos de acesso obtidos são bastante razoáveis. Assim ao potenciar a redução do tempo de projecto e permitir a redução da sua complexidade, a abordagem proposta nesta tese contribui certamente para o desenvolvimento de melhores sistemas dedicados.

5.2 Trabalho futuro

No decurso do trabalho identificaram-se vários temas que podem ser objecto de um estudo mais detalhado, no sentido de melhorar e aprofundar as soluções utilizadas, aumentando o desempenho do sistema.

A interface com a área dinâmica é bastante simples, não utilizando mecanismos de sincronismo e validação de dados. Seria interessante desenvolver-se uma nova interface capaz de garantir a integridade de dados entre as aplicações e os módulo reconfiguráveis ou, possivelmente, guardar dados intermédios entre reconfigurações. Esta interface está no caminho crítico, podendo evoluir para algo mais complexo que diminua a dependência do controlo constante por parte do SO.

Nem todos os registos da área reconfigurável são essenciais para conservar o estado de um DRM, assim seria útil o desenvolvimento de técnicas que permitissem a troca de DRMs. Esta troca, realizada de uma forma transparente para os processos, permite um escalonamento das aplicações mais adaptado às necessidades especificas dos sistemas, como os de tempo real. A reconfiguração de uma área menor permite, ainda, uma melhoria do desempenho.

Outro objecto de estudo é a manipulação do *bitstream* pelo SO. Através da adaptação do trabalho desenvolvido em [24], o controlo da posição dos diferentes módulos dentro da área dinâmica adicionaria um novo nível de liberdade ao sistema, permitindo por exemplo a utilização de diferentes sequências de módulos.

A combinação das duas sugestões anteriores,ou seja, a manipulação da configuração de cada módulo após a sua utilização tem um potencial enorme. A possibilidade de alterar os parâmetros internos do módulo sem uma estrutura extra para o efeito permite diminuir os recursos necessários enquanto que a verificação do *bitstream* permite a implementação que técnicas de depuração ou detecção de falhas durante a execução.

O desempenho global do sistema pode ser melhorado aumentando o paralelismo da aplicações. Como se pode verificar em [18] a utilização de técnicas de DMA (*Direct Memory Access*) aumentaria a taxa de transmissão de dados entre os módulos reconfiguráveis e a memória, libertando o CPU para outras tarefas. O núcleo GNU/Linux já tem alguns controladores que utilizam DMA, no entanto não foram analisados.

Anexo A

Configuração do sistema de desenvolvimento

Neste anexo descreve-se o processo de configuração até o núcleo GNU/Linux estar a correr na FPGA. O processo está dividido em duas fases. Na primeira configura-se o *hardware* enviando-se o *bitstream* para a FPGA. Na segunda parte gera-se uma imagem do núcleo, compatível com a arquitectura, e enviando-a para a memória SDRAM

A.1 Configuração do *Hardware*

A placa de desenvolvimento utilizada, *Memec Virtex-II Pro P7-fg456*, contém uma FPGA com um PowerPC 405 embutido, 32 MB de SDRAM, uma porta série, quatro *leds* e um *lcd* de 16 dígitos. A configuração do sistema é realizada através do Xilinx EDK[1] que contém os IP Cores[2] para os vários componentes da placa.

O EDK possui um assistente de configuração rápido, no entanto não possui o ficheiro de configuração para a placa utilizada sendo necessário descarregá-lo[3], que só funciona até à versão 9.1. O ficheiro descarregado deve ser descompactado para a pasta onde o EDK está instalado.

Utilizando o assistente de inicialização criou-se um projecto para a placa *Memec Virtex-II Pro P7-fg456 Development Board with P160 Comm Module Rev 4*. O configuração inclui um PowerPC a funcionar a 300 MHz, e os barramentos a 100 MHz. Os periféricos foram configurados de acordo com a tabela A.1

Posteriormente adicionou-se os IP Cores da *OPB FPGA Internal Configuration Acess Port* (*v. 1.10.a*) e da *dockO32I32* [4]. Os periféricos forma ligados ao CPU através da barramento OPB e os endereços configurados manualmente com tamanho 64K.

[1] Xilinx Embedded Development Kit.
[2] Módulos de *hardware* previamente sintetizados.
[3] http://www.em.avnet.com/xbd
[4] disponibilizada por Eurico Damásio

Tabela A.1: Parâmetros de configuração dos periféricos.

Nome	**Controlador**	**Opções**
RS232	OPB_UART16550	*Configure as UART 16550*, *Use interrupt*
LEDs_4Bit	OPB_GPIO	
SDRAM_8Mx32	PLB_SDRAM	*Use high speed pipeline stage*
plb_bram_if_cntlr_1	PLB BRAM IF CNTLR	*Memory size: 16KB*

A.1.1 Árvore do Dispositivo - *Device Tree*

As configurações do *hardware* são passadas para o Kernel GNU/Linux através de uma árvore do dispositivo (*device tree* [39] - (*.dts)). Estes ficheiros contém o *hardware* estruturado por barramentos e endereços, permitindo passar parâmetros adicionais.

O Xilinx EDK permite gerar *device trees* automaticamente. Descarregaram-se[5] os ficheiros necessários e descompactou-se para a raiz do projecto EDK. Acrescentou-se ao ficheiro *bsp/device-tree/data/device-tree_v2_1_0.tcl* o código seguinte na linha 1109, de modo a suportar a memória SDRAM:

```
"plb_sdram" {
                    # Handle bankless memories.
                    lappend tree [memory $slave "" ""]
                    set memory_count [expr $memory_count + 1]
                }
```

Posteriormente seguiu-se o guia disponível em [11]. Ou seja, editou-se em "*Software > Software Platform Settings > OS and Libraries*" a *console device* para *RS232* e *bootargs* para *console=ttyS0 root=/dev/ram*. Gerou-se a árvore do dispositivo em "*Software > Generate Libraries and BSPs*" que fica disponível em *ppc405_0/libsrc/device-tree/xilinx.dts*.

A.2 Configuração do *Software*

Nesta secção descrevem-se os passos necessários à configuração do Kernel GNU/Linux. O sinal $ no início da linha indica um comando, enquanto que a notação ${x} indica que se refere à variável x. A utilização de variáveis revela-se bastante útil, principalmente nos *scripts* para as operações muitas vezes repetidas.

A.2.1 Compilador Cruzado

Um compilador cruzado permite compilar *software* numa plataforma para outra diferente, como por exemplo gerar um executável num PC x86_64 que será executado num

[5] http://git.xilinx.com/cgi-bin/gitweb.cgi?p=device-tree.git;a=snapshot;h=HEAD

PowerPC. Para tal utilizou-se o CroosTool-NG, disponível em [10], para obter um ambiente de desenvolvimento compatível com o PowerPC.

Como são ferramentas em desenvolvimento, mesmo o CroosTool-NG utilizado foi uma versão do SVN, é normal que ocorram incompatibilidades entre diferentes versões dos programas. A tabela A.2 indica as versões utilizadas.

Tabela A.2: Parâmetros de configuração do CrossTool-NG [10].

Parâmetro	**Valor**
Target Architecture	powerpc
Type	Cross
Linux Kernel Version	2.6.27
binutils version	2.18
gcc version	4.3.2
C library	glibc
glibc version	2.7

```
$ wget http://ymorin.is-a-geek.org/download/crosstool-ng/crosstool-ng-1.3.2.tar
    .bz2
$ tar -xvf crosstool-ng-1.3.2.tar.bz2
$ ./crosstool-ng-1.3.2/configure --prefix=${DEVEL\_DIR/croostool}
$ make ./crosstool-ng-1.3.2/
$ make install ./crosstool-ng-1.3.2/
$ cd ${DEVEL\_DIR/croostool
$./bin/ct-ng menuconfig
$./bin/ct-ng build
```

Sequência de comandos para instalar o CrossTool-NG.

A.2.2 Sistema de Ficheiros

Foi construído um sistema de ficheiros mínimo[6], uma vez que não existem muitos recursos e a velocidade de comunicação com a FPGA é lenta, que é incluído juntamente com a imagem do Kernel GNU/Linux. O sistema é constituído por um conjunto de directorias, nodos, bibliotecas, ficheiros de configuração e programas. A raiz do sistema é designada por ${RAMDISK}.

```
$ mkdir -p ${RAMDISK}/{bin,dev,etc,home,lib,proc,sbin,usr,var,tmp,sys}
$ mkdir -pv ${RAMDISK}/var/{lock,log,run,spool}
$ mkdir -pv ${RAMDISK}/var/{opt,cache,lib/{misc,locate},local}
```

Comandos para criar a estrutura de pastas.

[6]Baseado no trabalho de Eurico Damásio

```
$ sudo mknod -m 0666 ${RAMDISK}/dev/ttyS0 c 4 64
$ sudo mknod -m 0666 ${RAMDISK}/dev/null  c 1 3
$ sudo mknod -m 0666 ${RAMDISK}/dev/ram   b 1 1
$ sudo mknod -m 0666 ${RAMDISK}/dev/console c 5 1
```

Cria os nodos essenciais ao Sistema.

As bibliotecas essenciais (**.so*) foram copiadas da directoria do CrossTool-NG para a pasta *$RAMDISK/lib*.

Os ficheiros de configuração essenciais foram criados na pasta */etc*. O ficheiro *fstab* contém a informação das partições, como o *sysfs*. No ficheiro *inittab* estão os programas que devem ser executados nas diferentes fases de vida do sistema, como por exemplo no arranque. O ficheiro *init.d* contém os comandos de inicialização do sistema, como montar partições e criar nodos.

O *BusyBox* implementa um conjunto de programas essenciais a qualquer sistema UNIX mas optimizados para sistema embutidos. Foram adicionados os programas para obter um sistema funcional como uma linha de comandos, um editor de texto, as aplicações para navegar pelas ficheiros e pastas e funcionalidades para manipular os módulos do Kernel ou os processos.

A.2.3 Configurar e Compilar o Kernel

Utilizou-se a versão do núcleo GNU/Linux mantida pela Xilinx[7]. A placa utilizada não é suportada oficialmente, no entanto tem um comportamento semelhante à Virtex4 uma vez que possuem o mesmo processador (PowerPC 405). Assim, substitui-se a árvore de dispositivos existente no kernel (*arch/powerpc/boot/dts/virtex405-ml405.dts*) pela gerada no EDK (*xilinx.dts*).

```
$ cp arch/powerpc/configs/40x/virtex4_defconfig .config
```

Copia as configurações da Xilinx Virtex 4.

Editou-se o ficheiro de configuração, através do comando *make ARCH=powerpc xconfig*, fazendo as alterações indicadas na tabela A.3. Removeu-se os controladores referentes a interfaces com a rede uma vez que não foram utilizados.

Tabela A.3: Alterações ao ficheiro de configuração da Virtex 4.

Variável	Conteúdo
Initramfs	${RAMDISK}
Initial Kernel command string	*"console=ttyS0,9600 root=/dev/ram rw"*

[7] http://git.xilinx.com/cgi-bin/gitweb.cgi

Posteriormente compilou-se o núcleo com o comando demonstrado a seguir, ficando a imagem final em *arch/powerpc/boot/simpleImage.virtex405-ml405.elf*.

```
$ make ARCH=powerpc CROSS_COMPILE=${CROOSS\_TOOL\_PREFIX} zImage
```

Comando para compilar o núcleo GNU/Linux.

Anexo B

Como criar módulos reconfiguráveis

Neste anexo descreve-se, passo a passo, o processo para gerar módulos reconfiguráveis dinamicamente utilizando as ferramentas da Xilinx. Pretende-se enumerar os passos realizados durante este projecto de modo a que poder facilmente repetir-se o processo. Assume-se que o leitor já tenho os conhecimentos necessários no fluxo de projecto convencional.

B.1 *Software* utilizado

As ferramentas da Xilinx não têm suporte, de origem, para a reconfiguração parcial. Necessitam por isso de ser alteradas com um suplemento adiciona, as PR (*Partial Reconfiguration*) TOOLS[1]. Estas ferramentas alteram os parâmetros de síntese pelo que só devem ser utilizadas em projectos reconfiguráveis dinâmicamente.

Apenas a versão completa das aplicações funciona pelo que é necessário a licença dos seguintes *softwares*:

- ISE 9.2i SP4;
- PlanAhead 10.1.

Atendendo ao teor experimental das ferramentas, é necessário seguir escrupulosamente a ordem do processo de instalação.

B.2 Síntese Lógica

Utilizando uma linguagem de descrição do *hardware*, como o *vhdl*, descreve-se o circuito que se pretende implementar. Posteriormente no ISE gera-se a *netlist* do circuito.

O Sistema Base contém instanciado todos os módulo estáticos e um módulo reconfigurável, para cada área reconfigurável. Os módulos reconfiguráveis, para cada área, têm

[1] http://www.xilinx.com/support/prealounge/protected/index.htm

a mesma interface de modo a manter a integridade do circuito entre reconfigurações. Todas as ligações entre a área estática e uma área dinâmica são efectuadas através de uma *Bus Macro*. No *top level* só podem existir módulos e ligações, não sendo permitido a utilização de recursos lógicos neste nível.

As *Bus Macro* impõem às ferramentas de síntese uma localização fixa, desta forma a comunicação com área reconfigurável ocorre utilizando os mesmos recursos independentemente do módulo utilizado. As *Bus Macro* podem ser descarregadas juntamente com as PR TOOLS, sendo especificas para cada FPGA. Existem *Bus Macro* síncronas e assíncronas devendo-se escolher a mais adequada ao projecto, no entanto é necessário utilizar o sentido correcto da ligação, ou seja da esquerda para a direita ou da direita para a esquerda.

Os módulos reconfiguráveis devem ser sintetizados em projectos diferentes, mas utilizando o mesmo *top level*. As funcionalidades lógicas devem ser implementadas em sub-módulos.

B.2.1 Posicionamento da área reconfigurável e da *Bus Macro*

A área reconfigurável deve ser definida manualmente e garantindo que têm recurso suficientes para implementar qualquer módulo. Assim, através de um processo iterativo deve-se verificar cada módulo, começando pelos maiores. A área reconfigurável deve aproveitar o máximo de recursos verticalmente uma vez que a reconfiguração é feita por coluna.

O posicionamento dos módulos é definido através do PACE (ISE > User Constraints > Create Area Constraints), que produz/altera um ficheiro de restrições (**.ucf*). Para melhor estimar a necessidade de recursos os parâmetros de síntese devem ser alterados temporariamente. Em especial o parâmetro *Keep Hierarchy* para *Soft*, tal como demonstra a figura B.1, de modo a manter os recursos de cada módulo agrupados.

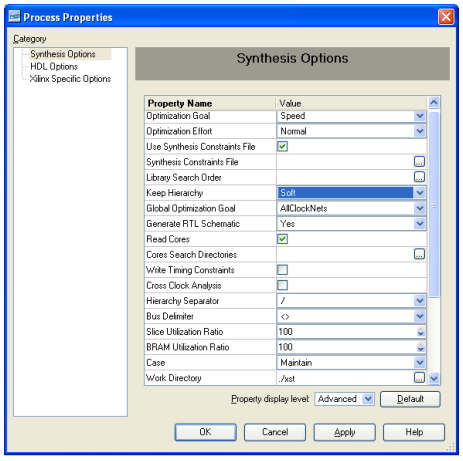


Figura B.1: Janela de selecção dos parâmetros de síntese.

As *Bus Macro* são definidas no ficheiro de restrições através do parâmetro *LOC*, como por exemplo *INST "bmti0" loc "SLICE_X32Y44";*. As *Bus Macro* devem ser definidas em *Slices* de coordenadas, X e Y, pares.

Na *netlist* final, o módulo reconfigurável deve estar definido como uma *black box*. O módulo reconfigurável tem de ser declarado (*MODE = RECONFIG*) como tal no ficheiro de restrições, permitindo sintetizar sem erros.

```
#PRA
AREA_GROUP "AG_base_rec_area" RANGE = SLICE_X8Y79:SLICE_X33Y22 ;
AREA_GROUP "AG_base_rec_area" RANGE = RAMB16_X1Y8:RAMB16_X2Y1 ;
INST "base_rec_area" AREA_GROUP = "AG_base_rec_area" ;
AREA_GROUP "AG_base_rec_area" MODE = RECONFIG ;

#SA
AREA_GROUP "AG_static" RANGE = SLICE_X0Y79:SLICE_X6Y0,SLICE_X34Y79:SLICE_X67Y0;
AREA_GROUP "AG_static" RANGE = RAMB16_X3Y8:RAMB16_X5Y1,RAMB16_X0Y9:RAMB16_X0Y0,
    RAMB16_X5Y0:RAMB16_X5Y0;
INST "system_i/dcm_0" AREA_GROUP = "AG_static" ;
INST "system_i/docko32i32_0" AREA_GROUP = "AG_static" ;
INST "system_i/leds_4bit" AREA_GROUP = "AG_static" ;
INST "system_i/opb" AREA_GROUP = "AG_static" ;
INST "system_i/opb_hwicap_0" AREA_GROUP = "AG_static" ;
INST "system_i/opb_intc_0" AREA_GROUP = "AG_static" ;
INST "system_i/plb" AREA_GROUP = "AG_static" ;
INST "system_i/plb2opb" AREA_GROUP = "AG_static" ;
INST "system_i/plb_bram_if_cntlr_1" AREA_GROUP = "AG_static" ;
INST "system_i/plb_bram_if_cntlr_1_bram" AREA_GROUP = "AG_static" ;
INST "system_i/reset_block" AREA_GROUP = "AG_static" ;
INST "system_i/rs232" AREA_GROUP = "AG_static" ;
INST "system_i/sdram_8mx32" AREA_GROUP = "AG_static" ;

#BUSMACROS
INST "bmtce0" loc = "SLICE_X32Y40";
INST "bmti0" loc = "SLICE_X32Y44";
INST "bmti1" loc = "SLICE_X32Y48";
INST "bmti2" loc = "SLICE_X32Y52";
INST "bmti3" loc = "SLICE_X32Y56";
INST "bmto0" loc = "SLICE_X32Y60";
INST "bmto1" loc = "SLICE_X32Y64";
INST "bmto2" loc = "SLICE_X32Y68";
INST "bmto3" loc = "SLICE_X32Y72";
```

Declaração da restrições relativas à reconfiguração dinâmica(*system.ucf*).

Os módulos estáticos foram declarados dentro da mesma área, permitindo que o algoritmo de síntese optimize as ligações entre eles. O módulo *plb_bram_if_cntlr_1_bram* não é reconhecido no PACE, sendo necessário declará-lo manualmente. Salienta-se o

facto de a área reconfigurável agregar vários blocos de *slices* separados, devido à localização física do PowerPC e das E/S. Neste caso as ligações com a SDRAM ocorrem do lado esquerdo, o *hard-core* está localizado no lado direito e a porta ICAP no canto inferior direito.

B.3 Síntese Física

No PlanAhead cria-se um projecto novo, importando a *netlist*(**.ngc*) e o ficheiro de restrições(**.ucf*) do Sistema Base. As *Bus Macro* e os módulos estáticos devem ser sintetizados na mesma pasta, adicionada ao projecto no momento em que se define a *netlist* do *top_level*.

Depois seguem-se os seguintes passos:

- Define-se o projecto com reconfigurável parcialmente (File > Set PR Project).
- Define-se a área reconfigurável com reconfigurável (*Set Reconfigurable*), no menu representado na figura B.2.

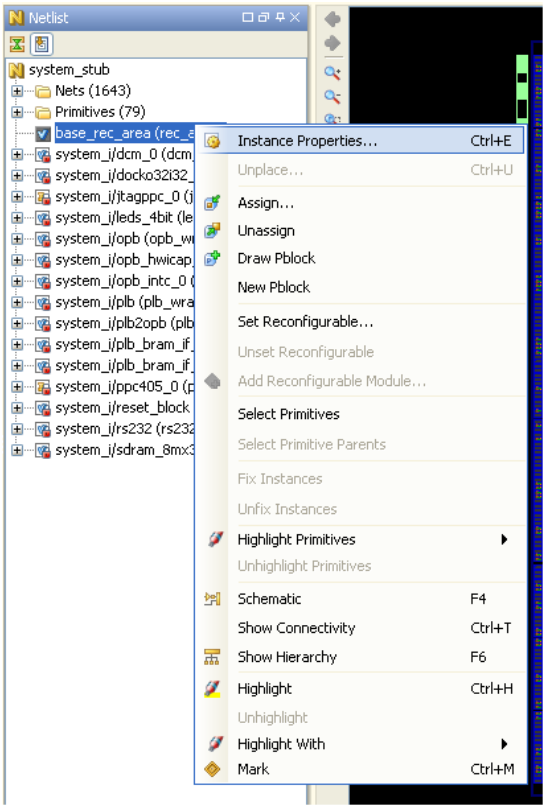


Figura B.2: Definição do módulo como reconfigurável.

- Executar o comando *Launch Runs* do menu ilustrado na figura B.3. Primeiro a área estática e depois os restante módulos.

- Gerar os *bitstreams* finais em *Run PR Assemble*.

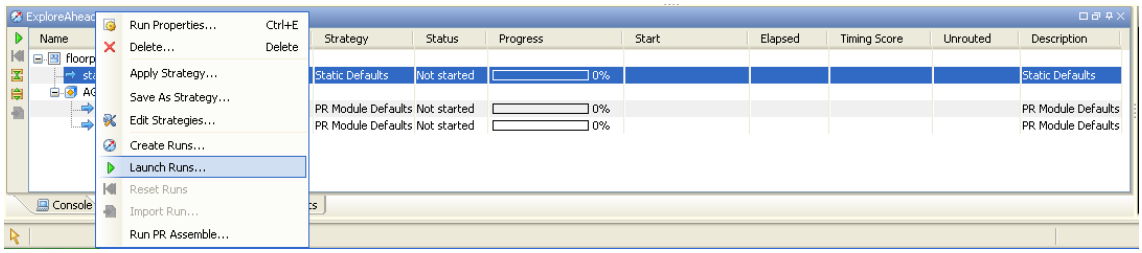


Figura B.3: Menu do *ExploreAhead Runs*.

Os *Bitstreams* finais (**.bit*) dos módulos reconfiguráveis e da área estática encontram-se em *project_dir/project_name.runs/floorplan_1/merge*.

Referências

[1] P. Garcia, K. Compton, M. Schulte, E. Blem, e W. Fu. An overview of reconfigurable hardware in embedded systems. *EURASIP Journal on Embedded Systems*, 2006(ID 56320):19, 2006.

[2] V. George, , H. Zhang, e J. Rabaey. The design of a low energy FPGA. Em *Proceedings. 1999 International Symposium on Low Power Electronics and Design*, páginas 188–193, 1999.

[3] S. Hauck e A. DeHon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Elsevier B.V, Novembro 2007.

[4] T . Todman, G. Constantinides, S. Wilton, P. Cheung, W. Luk, e O. Mencer. Reconfigurable computing: Architectures and design methods. volume 152, páginas 193–205, Março 2005.

[5] J. G. Tong, I. D. L. Anderson, e M. A. S. Khalid. Soft-core processors for embedded systems. *Microelectronics, 2006. ICM '06. International Conference on*, páginas 170–173, Dezembro 2006.

[6] Xilinx. *Virtex-II Pro and Virtex-II Pro X FPGA User Guide*. Xilinx, 4.2 edição, Novembro 2007.

[7] M. G. O. Gericota. *Metodologias de teste para FPGAs (Field Programmable Gate Arrays) integradas em sistemas reconfiguráveis*. Tese de doutoramento, Faculdade de Engenharia da Universidade do Porto, Porto, Portugal, Abril 2003.

[8] B. H. Fletcher. FPGA embedded processors: Revealing true system performance. San Francisco, 2005. Embedded Systems Conference.

[9] IBM. *PowerPC 405 CPU Core Product Overview*. IBM, Setembro 2006.

[10] Y. E. Morin. Crosstool-ng, Setembro 2008. http://ymorin.is-a-geek.org /dokuwiki/-projects/ crosstool.

[11] John Linn. Xilinx open source Linux wiki, Setembro 2008.

[12] G. B. Wigley, D. A. Kearney, e D. Warren. Introducing ReConfigME: An operating system for reconfigurable computing. Em *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, páginas 687–697, London, UK, 2002. Springer-Verlag.

[13] H. Kwok-Hay So e R. W. Brodersen. Improving usability of FPGA-based reconfigurable computers through operating system support. Em *Proceedings of the 16th International Conference on Field Programmable Logic and Applications*, 2006.

[14] K. O'Neill. Antifuse FPGA technology: Best option for satellite applications. *COTS Journal*, Dezembro 2003.

[15] Xilinx. *Virtex-II Pro / Virtex-II Pro X Complete Data Sheet*. Xilinx, Novembro 2007.

[16] Xilinx. *Virtex FPGA Series Configuration and Readback*. Xilinx, 2.8 edição, Novembro 2005.

[17] M.L. Silva. Ferramentas de apoio ao teste concorrente para FPGAs com reconfiguração parcial dinâmica. Tese de mestrado, Faculdade de Economia da Universidade do Porto, 2004.

[18] M.L Silva e J.C. Ferreira. Exploiting dynamic reconfiguration of platform FPGAs: implementation issues. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, página 8, Abril 2006.

[19] F. Farshadjam, M. Dehghan, M. Fathy, e M. Ahmadi. A new compression based approach for reconfiguration overhead reduction in Virtex based RTR systems. *Computers & Electrical Engineering*, 32(4):322 – 347, 2006.

[20] Xilinx. *Virtex 5 FPGA User Guide*. Xilinx, 4.5 edição, Janeiro 2009.

[21] M.L Silva e J.C. Ferreira. Generation of hardware modules for run-time reconfigurable hybrid CPU/FPGA systems. Em *XX Conference on Design of Circuits and Integrated Systems (DCIS'2005)*, Lisboa, Portugal, Novembro 2005.

[22] E. L. Horta, J. W. Lockwood, D. E. Taylor, e D. Parlour. Dynamic hardware plugins in an FPGA with partial run-time reconfiguration. Em *DAC '02: Proceedings of the 39th conference on Design automation*, páginas 343–348, New York, NY, USA, 2002. ACM.

[23] Inc Xilinx. Partial reconfiguration early access software tools, 2008.

[24] M.L Silva e J.C. Ferreira. Support for partial run-time reconfiguration of platform FPGAs. *J. Syst. Archit.*, 52(12):709–726, 2006.

[25] L. Torvalds. The Linux kernel archives, Setembro 2008. http://www.kernel.org/.

[26] Inc. Xilinx. *OPB HWICAP*. Xilinx, Inc., 1.4 edição, Março 2004.

[27] G. B. Wigley. *An Operating System for Reconfigurable Computing*. Tese de doutoramento, University of South Australia's School of Computer and Information Science, Adelaide, Australia, Janeiro 2005.

[28] H. Kwok-Hay So e R. W. Brodersen. *BORPH: An Operating System for FPGA-Based Reconfigurable Computers*. Tese de doutoramento, EECS Department, University of California, Berkeley, Julho 2007.

[29] Martyn Edwards e Peter Green. Run-time support for dynamically reconfigurable computing systems. *Journal of Systems Architecture*, 49(4-6):267–281, 2003.

[30] N. Bergmann, J. Williams, e P. Waldeck. Egret: A flexible platform for real-time reconfigurable system-on-chip. Em *The International Conference onThe International Conference on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, Nevada, USA, Junho 2003.

[31] J. Williams e N. Bergmann. Embedded Linux as a platform for dynamically self-reconfiguring systems-on-chip. 2005.

[32] uClinux. uclinux: Embedded Linux/microcontroller project, 2007. http://www.uclinux.org.

[33] V. Rana, M. Santambrogio, D.Sciuto, B. Kettelhoit, M. Koester, M. Porrmann, e U. Ruckert. Partial dynamic reconfiguration in a Multi-FPGA clustered architecture based on Linux. Em *Parallel and Distributed Processing Symposium*, páginas 1–8. IEEE International, Março 2007.

[34] P. Mochel. The sysfs filesystem. Em *Proceedings of Annual Linux Symposium*, Ottawa, Canada, 2005. kernel.org.

[35] M.L Silva e J.C. Ferreira. Run-time reconfiguration support for FPGAs with embedded CPUs: the hardware layer. Em *12th Reconfigurable Architectures Workshop (RAW 2005)*, Denver, Colorado, USA, Abril 2005.

[36] D. Vlasenko. Busybox: The swiss army knife of embedded Linux, Setembro 2008. http://www.busybox.net/.

[37] J. Corbet, A. Rubini, e G. Kroah-Hartman. *Linux Device Drivers*. O'Reilly, 3ª edição, Fevereiro 2005.

[38] Sun Microsystems. The Open Firmware home page, Maio 2005. http://playground.sun.com/1275/home.html.

[39] G. Likely e J. Boyer. A symphony of flavours: Using the device tree to describe embedded hardware. Em *Proceedings of Linux Symposium*, Ottawa, Ontario, Canada, Julho 2008. Linux Symposium.