FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

FEUP

Integração de unidades de memória flash em sistemas embutidos sobre plataformas FPGA

Francisco Nuno Alves Orge Basadre

Tese submetida no âmbito do

Mestrado Integrado em Engenharia Electrotécnica e de Computadores

Major de Telecomunicações

Orientador: José Carlos Alves (Professor Doutor)

Janeiro de 2009

Resumo

Devido à cada vez maior necessidade dos sistemas embutidos (MP3s, telemoveis, PDAs, etc) necessitarem de unidades de armazenamento adicionais desenvolveu-se um sistema de acesso a unidades de memória *flash* a partir de sistemas digitais embutidos, implementados sobre plataformas FPGA. O sistema em si consiste numa interface de comunicação com a unidade de armazenamento e de um *soft-processor* embutido, encarregue de fazer dinamicamente a gestão e organização dos dados na unidade. Considera-se, por isso, um sistema com uma configuração mista de *hardware* e *software*. Uma solução inovadora que trás vantagens aos sistemas actualmente existentes, que são puramente em *hardware* ou *software*.

Abstract

Due to the increasing need of the Embedded systems for additional data storage it has been developed a memory flash access system, it was developed in a digital built-in system, implemented on FPGA platform. The system consists in a storage unit communication interface and a soft processor embedded in FGPA, that is in charge of dynamically manage and organize the storage unit data. It is considered, therefore, a system with a mixed configuration of hardware and software. Is an innovator solution that brings advantages to the actual systems, that are purely in hardware or software.

Agradecimentos

Durante as extensas horas de trabalho e estudo usadas para realizar a Dissertação obtive vários tipos de apoios tanto morais como de aconselhamento técnico, quero portanto deixar aqui o agradecimento a todas essas pessoas que me apoiaram.

Um agradecimento especial para os "companheiros"de tese Bruno Dantas, Bruno Monteiro e também para o Prof. Dr José Carlos Alves por todo o apoio e orientação prestado.

Aproveito também para agradecer a família e amigos que me apoiaram, acompanharam e me incentivaram a ir mais longe durante a realização da dita Dissertação, mas, como também ao longo de todos os anos de estudo na Faculdade de Engenharia da Universidade do Porto, pois com o resolver da Dissertação finalizarei essa etapa e não queria deixar de agradecer.

Francisco Nuno Alves Orge Basadre

"Life is like riding a bicycle.
To keep your balance you must keep moving"

Albert Einstein

Conteúdo

Lista de Figuras

Lista de Tabelas

Abreviaturas e Símbolos

MMC	MultiMediaCard
SD	Secure Digital
MBR	Master Boot Record
FPGA	Field-programmable gate array
PDA	Personal digital assistant
FAT	File Allocation Table
JFFS	Journalling Flash File System
YAFFS	Yet Another Flash File System
SPI	Serial Peripheral Interface
ECC	Error Correction Code
CRC	Cyclic redundancy check
CLK	clock
CS	chip select
DI	data in
DO	data out
LGPL	Lesser General Public License
FIFO	First in first out
IO	Input/Output
UART	universal asynchronous receiver/transmitter
LBA	Logic block address
EOF	End of file

Capítulo 1

Introdução

Este projecto foi desenvolvido no âmbito da disciplina de Dissertação da Faculdade de Engenharia da Universidade do Porto do curso Mestrado Integrado em Engenharia Electrotécnica e Computadores. A Dissertação tem como tema "Integração de unidades de memória *flash* em sistemas embutidos sobre plataformas FPGA".

Os cartões de memória MMC e SD, são pequenos dispositivos de memória *flash* de baixo custo e baixo consumo energético, desenhados especificamente para aplicações de armazenamento. São especialmente vantajosos em equipamentos móveis alimentados a bateria e que necessitem de uma capacidade média de armazenamento.

Devido ao seu pequeno tamanho, vão ao encontro das necessidades dos fabricantes, designadamente, de telemóveis, PDAs, máquinas digitais, leitores multimédia, para além de outros, que necessitam de suportes adicionais de memória e que não impliquem um aumento do volume dos terminais. Na figura 1.1 pode-se ver um estudo de mercado que indica um claro aumento do número de revendas a nível mundial.

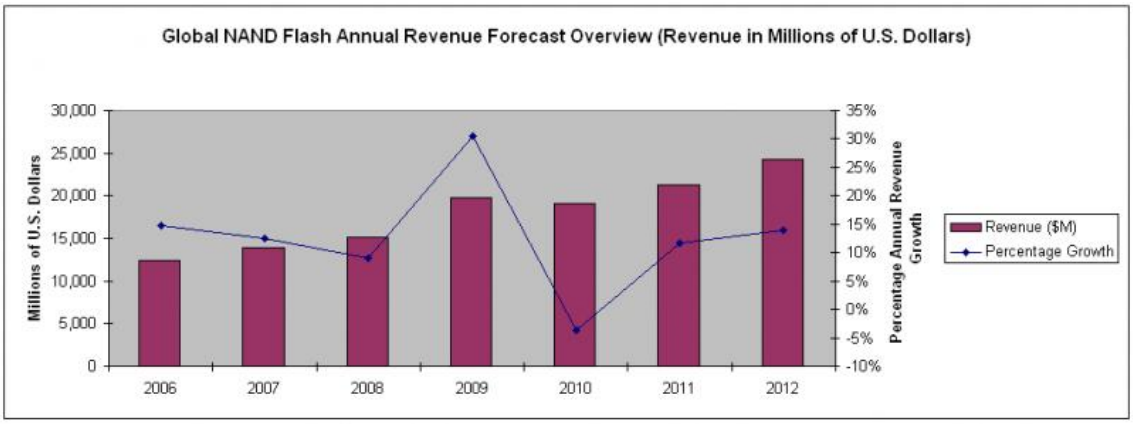

Figura 1.1: Evolução mundial das vendas de memórias *flash* [1]

É de notar que o preços das memórias *flash* aproxima-se cada vez mais do zero e ao mesmo tempo, nas FPGAs, a densidade de *gates* dobra a cada 18 meses e com custos relativamente baixos para os padrões industriais. Como consequência, temos que, cada vez mais engenheiros optam por incorporar *soft-processors* nos seus projectos, ou até eliminam por completo o processador e *software* projectando o sistema completamente em *hardware* [6].

Tudo indica que todo o mercado relacionado com as tecnologias de memórias *flash* e das FPGA continuará a crescer, assim como a exigir um continuo investimento em inovação e desenvolvimento.

1.1 Motivação e objectivos

Visa-se desenvolver e implementar uma biblioteca de módulos de *hardware* digital capazes de suportar o acesso a unidades de memória *flash* de baixo custo (cartões de memória), a partir de sistemas digitais embutidos, implementados sobre plataformas FPGA. A interface a desenvolver deverá permitir a gestão dinâmica de um sistema de ficheiros mantido na memória *flash*, e que seja uma alternativa viável aos sistemas já existentes. Para realizar a dissertação terão que ser desenvolvidas duas camadas lógicas:

Uma camada física que implementará as funções básicas do dispositivo como a escrita, leitura e apagar blocos. O acesso ao dispositivo (cartão de memória MMC/SD) será feito através de um barramento SPI.

Para optimizar as funções básicas do dispositivo, aproveitando os recursos fornecidos pela FPGA, utilizada neste projecto, põe-se a hipótese de existir uma memória intermédia, que fará de *buffer* na troca de dados entre camadas [7]. Este circuito intermédio terá que ser implementado em *hardware*, ou seja, na FPGA, devido à capacidade de armazenamento oferecida pela FPGA e à velocidade que é necessária para a troca de dados entre o processador e o cartão de memória.

Uma segunda camada será desenvolvida/implementada, designada a "camada do sistema de ficheiros". Requisito para este projecto, é a portabilidade do sistema de ficheiros para uso nos sistemas operativos mais utilizados (Windows, Linux, MacOS) nos PCs actuais. Pretende-se criar um sistema de ficheiros com a dita portabilidade. Sabe-se que até ao momento, não existe nenhum sistema de ficheiros dedicado a este tipo de dispositivos de armazenamento.

Este projecto pode ser efectuado em três configurações possíveis: implementar a interface em *hardware* ocupando espaço na FPGA, em *software*, gastando recursos do microprocessador, e, por último, um misto das duas hipóteses anteriores. Uma comparação de desempenho entre sistemas em *hardware* e *software* pode ser vista na figura 1.2 com uma clara vantagem dos sistemas em *hardware*.

O projecto que se pretende desenvolver irá ser com base na ultima hipótese e é implementado num misto de *hardware* e *software*. Para cumprir esse objectivo a camada física será feita em *hardware* e a camada do sistema de ficheiros em *software* recorrendo-se para isso a um microprocessador/microcontrolador. Esta solução ainda não existe no mercado e será sem dúvida

Software version (processor at 100MHz)	Hardware version (FPGA at 24MHz)
252 ms	6.9 ms

Software versus hardware performance (1000 blocks of text data)

Figura 1.2: Comparação entre Hardware e Software [2]

de interesse desenvolve-la. À partida, das três será a mais optimizada numa relação de taxa de transferência de dados por espaço ocupado.

No final será de comparar a solução criada com outras já existentes, mas elaboradas puramente em software e/ou hardware.

1.2 Estrutura do Documento

Este documento segue uma estrutura que se relaciona com o planeamento temporal, do desenvolvimento do projecto e consiste em 7 capítulos, do modo que se segue:

Primeiro capitulo Contém a definição do tema de Dissertação, assim como a sua introdução. Também nele se refere a área de abrangência do projecto e os objectivos que se podem concretizar.

Segundo capitulo Trata-se do estudo do estado da arte. Neste estudo procura-se encontrar um ponto de partida para o projecto com base em projectos e/ou artigos relacionados com o tema.

Terceiro capitulo Define-se um arquitectura lógica, casos de utilização do projecto e anda se faz uma especificação de requisitos de alto nível.

Quarto capitulo Faz-se referência à forma como o projecto foi implementado. Seguindo uma lógica relacionada com as fases do desenvolvimento do projecto, neste capitulo, explica-se a implementação da camada de baixo nível, o mesmo é dizer camada física de acesso ao dispositivo (cartão MMC ou SD).

Quinto capitulo Encontra-se no seguimento do anterior. Aqui explica-se a implementação da camada do sistema de ficheiros.

Sexto capitulo Apresenta-se o produto final, o resultado do projecto.

Sétimo capitulo Elaboram-se raciocínios à volta dos resultados obtidos e apresenta-se possíveis melhorias ao sistema.

Capítulo 2

Estado da Arte

Neste capitulo vai-se fazer um estudo do estado da arte de todas as tecnologias relacionadas com o projecto, que se pretende desenvolver e implementar. O objectivo final é de criar um ponto de partida, com base no que já está desenvolvido e a partir desse ponto dar continuidade, no sentido de cumprir o tema o proposto.

Começou-se o estudo pela base do tema proposto, ou seja, pelos cartões MMC e SD e todo o seu funcionamento e tentou-se a partir dai "subir", ao nível lógico, até ao utilizador final.

2.1 Cartões de memória MMC/SD

Os cartões SD foram desenvolvidos de forma a que sejam compatíveis com os cartões MMC. Então, os sistemas de acesso a um SD podem também, sem grandes alterações, aceder a um cartão MMC.

Um MMC pesa menos de 2 gramas e possui a dimensão de $32 \times 24 \times 1.5$mm, igual à dos cartões SD, e tem uma capacidade de armazenamento que pode variar entre os 16 e os 512MB, sendo no caso dos SD muito maior e podendo ir até aos 32GB, tendo uma relação preço por capacidade muito menor [3] [5].

Toda a informação relativa ao dispositivo e à sua configuração está gravada no próprio cartão (frequência máxima, identificação do cartão, etc.). A interface de comunicação é simples de adaptar ao uso de um microprocessador/microcontrolador e ainda existe um modo alternativo de comunicação através de um protocolo SPI.

O cartão MMC/SD tem um controlador interno que gere a interface, ECC (*Error Correction Code*), possíveis faltas do dispositivo, gestão da energia e relógio. No nosso caso vamos utilizar o modo de comunicação SPI em que é necessário enviar comandos para inicializar o cartão.

Todos os comandos retornam uma mensagem de resposta de acordo com a especificação SPI dos cartões de memória MMC/SD.

Algumas acções têm que ser executadas antes de se começar a trabalhar com o MMC ou SD. Temos assim que:

- Quando o cartão for alimentado, é necessário enviar o comando *Reset Card*
- Para inicializar o cartão, enviar o comando *Initialise Card*
- Depois de inicializado, é necessário obter a identificação do cartão (16 bytes)
- O cartão está pronto para ler e escrever dados.

2.1.1 Descrição do funcionamento de um cartão de memória MMC/SD

A tensão de operação dos cartões MMC/SD está indicada no registo OCR que deve ser lido para confirmar se a tensão que está a ser aplicada é a correcta. A tensão deverá estar entre os 2.7 e os 3.6 Volts. A corrente poderá chegar aos 10 miliamperes, pelo que o o circuito a desenvolver deverá ser capaz de fornecer, no mínimo, esta corrente [5] [3].

A memória está organizada em blocos de 512 byte, chamados sectores. Cada bloco pode ser escrito e apagado individualmente. Existem três modos de escrita /leitura:

- *Stream mode*
- *Single Block Mode*
- *Multiple Block Mode*

2.1.1.1 Stream mode

Neste modo, o Circuito pode escrever/ler continuamente informação. O endereço tem que ser especificado quando dá o comando de escrever/ler. A operação acaba quando for enviado o comando de parar. Neste modo, não existe verificação da integridade da informação.

Para ler, pode ser enviado um qualquer endereço desde que seja válido, mas já no modo de leitura, o endereço tem que estar alinhado com o inicio de um sector e o tamanho da informação lida tem que ser múltiplo do tamanho do sector.

2.1.1.2 Single Block Mode

Neste modo, o Circuito lê um bloco de um tamanho específico. O bloco está protegido por um código CRC de 16 bit que é gerado pela unidade que envia e verificado pela que recebe.

O tamanho do bloco lido está limitado pelo tamanho do sector do dispositivo. Cada bloco tem de estar contido num sector.

Na escrita, o bloco que se pretende escrever tem que ter um tamanho igual ao do sector e o endereço apontar para o inicio do um dos sectores.

2.1.1.3 Multiple Block Mode

Este modo é igual ao *single block mode* mas com a diferença de que agora se escreve e se lê múltiplos blocos de informação que são armazenados/lidos de endereços seguidos da memória, a partir do endereço inicial.

Esta operação termina quando for enviado o comando de parar. O alinhamento e restrições do tamanho dos blocos na escrita/leitura são iguais aos usados no *single block mode*.

A unidade mais pequena que se pode apagar num MMC é um sector. Para aumentar a velocidade desta operação, múltiplos sectores podem ser pagados ao mesmo tempo. A operação de apagar está dividida em duas fases:

- Escolha dos sectores que se pretende apagar
- Apagar os sectores escolhidos

2.1.1.4 Escolha dos sectores que se pretende apagar

Envia-se um comando com o endereço inicial, seguido de outro com o endereço final. Depois do intervalo, de sectores individuais ou grupos, que se pretende apagar estar escolhido, envia-se o comando para apagar.

2.1.1.5 Apagar os sectores escolhidos

Os grupos ou sectores individuais escolhidos são apagados de forma sequencial, não ao mesmo tempo.

Existem mais comandos, que estão documentados nas respectivas especificações.

2.2 Cartões MMC/SD em modo SPI

Os cartões de memória MMC/SD permitem duas configurações possíveis para o barramento de dados. A configuração standard MMC/SD que usa um barramento de 2/4/8 bits, e uma alternativa que usa o barramento em modo SPI. Neste projecto vai ser utilizado o modo SPI. E é neste tipo de tecnologia que o estudo se irá concentrar.

2.2.1 Topologia no modo SPI

O modo SPI é uma alternativa ao modo normal que está definido para ser usado nos cartões MMC/SD. A comunicação através de uma interface SPI é bastante mais simples quando comparada ao modo normal de comunicação. Os cartões podem ser ligados através de uma interface genérica SPI (2.1). O SD tem o *SPI mode 0* definido no seu modo SPI, já o MMC, o modo SPI, é definido através de registos síncronos com o a transição do sinal SCLK, mas que funciona de modo equivalente ao do SD.

Pino	Nome	Tipo	Descrição SPI
1	CS	Entrada	Activa o chip
2	DataIn	Entrada	Entrada de dados no cartão
3	VSS1	Alimentação	Massa
5	CLOCK	Entrada	Relógio
6	VSS2	Alimentação	Massa
7	DataOut	Saída	Saída de dados do cartão

Tabela 2.1: Topologia spi

A interface SPI de um MMC é uma interface genérica equivalente às disponíveis no mercado, e tem quatro sinais:

- CS: Sinal que activa o *chip*.
- CLK: Entrada do sinal de relógio.
- DataIn: Sinal de dados do circuito de interface para o cartão.
- DataOut: Sinal de dados do cartão para o circuito de interface.

A interface SPI do cartão MMC/SD com e exterior é implementada de forma a que a transferência se inicie com o sinal CS, o número de bits transferidos seja múltiplo de oito e a leitura escrita dos bits seja alinhada com o sinal de *Clock*.

2.2.2 Envio de comandos e resposta

No modo SPI, o sentido da transferência de dados é fixo e é um modo de comunicação orientado ao byte em que os dados são transferidos em série. O comando do processador é uma trama com comprimento fixo de seis bytes. Quando um comando é transmitido para o cartão, é sempre enviada uma resposta. A transmissão é sincronizada por um sinal de relógio gerado pelo circuito de interface. Devido ao sinal de relógio ser gerado pelo circuito, este depois de enviar um comando, tem que ficar a espera de uma resposta válida. O tempo de resposta válido (Ncr, consultar B) é de 0 a 8 bytes para o SD e 1 a 8 bytes para o MMC. O sinal CS tem que estar a "0"durante a transferência.

O código CRC é opcional no modo SPI, mas a trama tem que ter o tamanho certo, logo, mesmo que não se use o código CRC, terão que se colocar os bytes do CRC na trama mesmo que sejam aleatórios. Durante a leitura o sinal DI tem que estar a "1".

Na figura 2.1 pode-se visualizar os estados lógicos das linhas e respectivos tempos de acesso.

2.2.3 Lista de comandos

A especificação SPI define apenas a camada de ligação lógica e não o protocolo de comunicação.

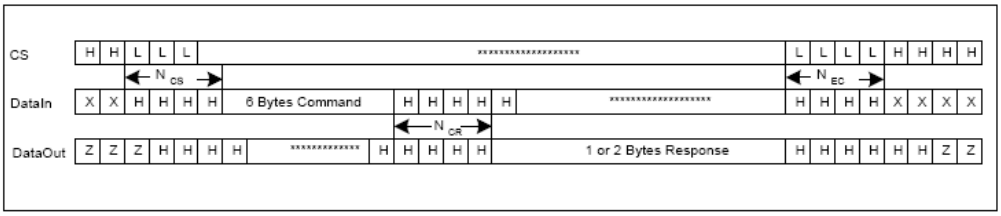

Figura 2.1: Envio e resposta de comandos [3]

Os cartões MMC/SD têm o protocolo de comunicação especifico em que cada comando é expresso através de uma abreviação do género *GO_IDLE_STATE* ou CMD <n>, em que o <n> é o numero do índex do comando, que pode ser de valores no intervalo de 0 a 63. Em anexo é apresentada a lista de comandos utilizados pelos cartões MMC e SD que poderá ser consultada [5] [3].

2.2.4 Tipos de resposta do cartão

Existem três tipos de resposta R1,R2 e R3 e que dependem do comando enviado. A resposta R1 é a mais usada, responde praticamente a todos os comandos. A trama da resposta R1 é mostrada na figura 2.2, o valor 0x00 quer dizer que houve sucesso na acção do comando enviado. Se ocorrer algum erro, o bit respectivo da indicação desse erro fica a "1". A resposta R3 só é usada para o comando CMD58 e verifica-se uma resposta de 40 bits. Os primeiros 8 são a resposta R1 seguidos do conteúdo do OCR.

Alguns comandos demoram mais tempo que o Ncr (ver B.1) segue-se então uma resposta R1b do cartão. É uma resposta R1 seguida de uma *busy flag*. Nesse estado o DO fica em nível baixo ("0") enquanto o cartão faz o processamento. O processador deverá esperar pelo sinal 0xFF que indica que o processo acabou.

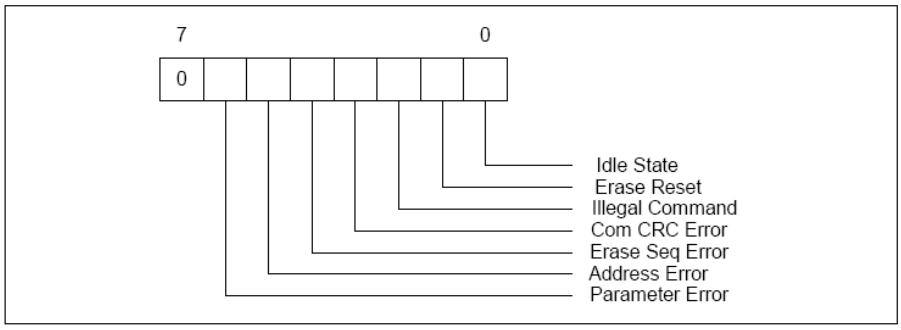

Figura 2.2: Resposta do tipo R1 [3]

2.2.5 Inicialização do cartão em modo SPI

Quando se aplica tensão no cartão este entra por *default* em modo de transferência MMC/SD e por isso terá que enviar uma série de comandos de forma a que entre em modo SPI.

2.2.5.1 Inserção do cartão

Depois de ser aplicada tensão ao cartão (aproximadamente 3.3v), tem que se esperar 1ms pelo menos. Nesse sentido, coloca-se o DI e o CS a "1"e aplica-se mais de 74 períodos de relógio no SCLK, sendo que depois o cartão fica a espera de comandos de inicialização.

2.2.5.2 *Reset*

Envia-se o CMD0 com o CS a "0"para fazer *reset* ao cartão. O cartão amostra o sinal CS quando recebe o comando CMD0. Se o sinal CS estiver a "0"o cartão entra no modo SPI, desde que o comando CMD0 seja enviado quando está a inicializar, e na trama o espaço do código CRC tem que conter bits válidos. Quando o cartão entra em modo SPI, o CRC deixa de ser verificado e então pode-se colocar bits aleatórios na trama no lugar do CRC, com a excepção dos comandos CMD0 e CMD8, que tem que ter o código CRC válido. Quando o CMD0 for aceite, o cartão entra no estado *idle* e responde como o R1 em *In_Idle_State* bit (0x01). O código CRC pode ser activado usando o CMD59.

2.2.5.3 Inicialização do cartão

Se o cartão está no modo *idle*, só aceita os comandos CMD0, CMD1 e CMD58 e todos os demais comandos são rejeitados. Quando recebe o CMD1, inicializa. Durante a inicialização o micro deve enviar continuamente o comando CMD1 e verificar a resposta. Quando o cartão tiver inicializado com sucesso, a resposta do cartão R1 será (0x00). Este processo leva várias centenas de milisegundos, dependendo do tamanho cartão e que quanto maior tanto mais tempo demora, facto esse que se deverá ter em consideração, para determinar o valor do *timeout*. Depois da inicialização pode ser enviado um comando genérico, por exemplo de leitura/escrita. O OCR e CSD podem ser lidos para verificar as características do cartão. O valor inicial do tamanho do bloco é maior que 512, podendo ser atribuído outro valor nesta altura, usando o comando CMD16.

Para que o SD e o MMC sejam compatíveis, usa-se o ACMD41 em vez do CMD1 para iniciar o SDC do cartão. Primeiro envia-se o ACMD41 e se for rejeitado, envia-se o CMD1.

2.2.6 Transferência de dados

Numa transferência de dados, um ou mais blocos são enviados/recebidos antes de ser recebida qualquer resposta.

2.2.6.1 Envio e resposta de pacote de dados

Um bloco de dados consiste numa trama com a seguinte sequência *Token, Data Block* e CRC. Há três tipos de data *token* que estão indicados na figura.

2.2.6.2 Leitura de um único bloco

O argumento especifica a localização do endereço do primeiro byte donde se pretende começar a leitura. A camada superior é que esta responsável por indicar o endereço de um byte alinhado com o do sector. Depois do CMD17 ser aceite, a operação de leitura começa e o cartão responde com um bloco de dados. Quando o micro recebe o bloco de dados, têm que ser enviados dois bytes de CRC mesmo que não sejam precisos. O bloco tem um tamanho por *default* de 512 bytes, mas pode ser mudado com o comando CMD16. Se ocorrer algum erro, o cartão responde com um *error token* que vem no pacote de dados.

Todo o processo pode ser visualizado nas figuras 2.5 e 2.3. No caso de erro na figura 2.4.

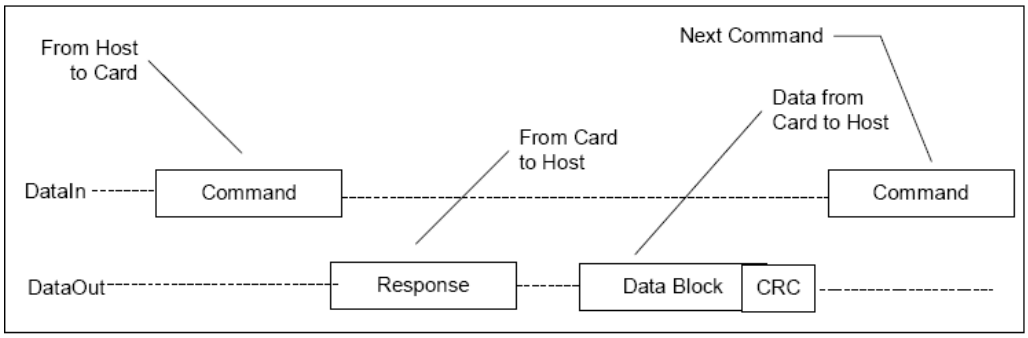

Figura 2.3: Leitura de um bloco de dados [3]

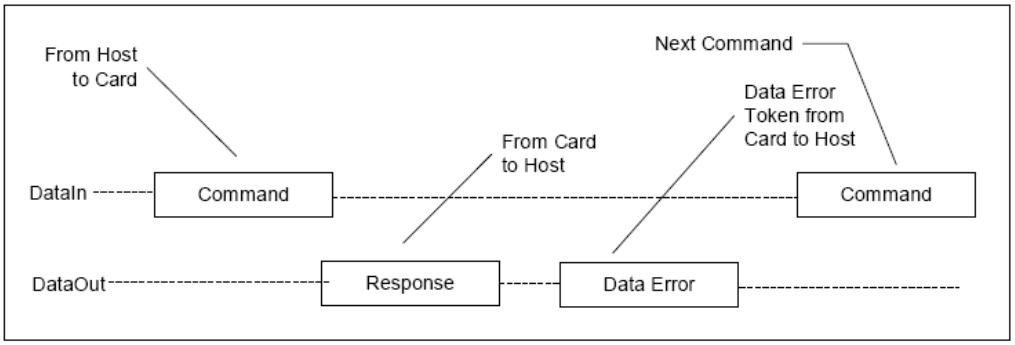

Figura 2.4: Leitura de um bloco de dados com erro [3]

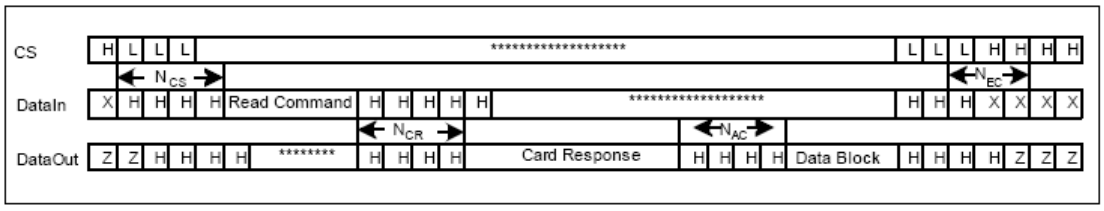

Figura 2.5: Esquema temporal da leitura de um bloco [3]

2.2.6.3 Escrita de um único bloco

Quando um comando de leitura é aceite, o micro envia um pacote de dados para o cartão, depois de esperar um período equivalente a um byte. O pacote tem que ter o mesmo formato que o pacote no caso de leitura. Depois do pacote ter sido enviado, o cartão envia a resposta imediatamente a seguir, mas antes disso, no final da recepção do pacote de dados, o cartão envia uma *busy* flag, que indica que ainda está a processar o pacote recebido. A maior parte dos cartões grava um bloco de valor fixo, 512 bytes.

O processo de escrita pode ser visualizado na figura 2.6 e 2.7.

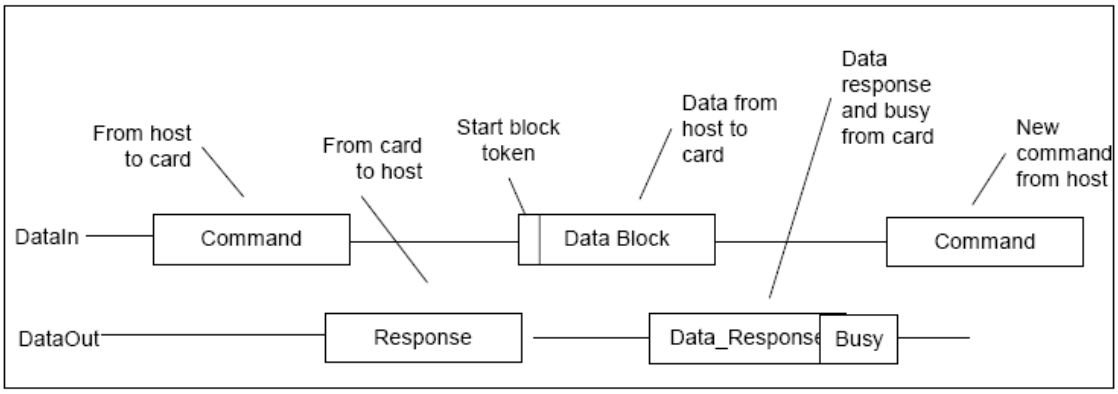

Figura 2.6: Escrita de um bloco de dados [3]

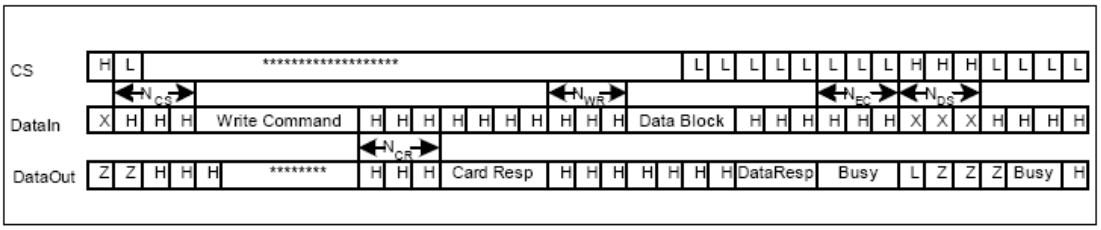

Figura 2.7: Esquema temporal da escrita de um bloco [3]

2.3 Sistema de ficheiros

Antes de se poder aceder a dispositivos de armazenamento (HDs, disquetes, cartões SD/MMC, etc.) é necessária uma preparação prévia, uma formatação física. Este processo consiste em dividir a unidade de armazenamento em blocos. A formatação física já vem de fábrica, mas pode ser alterada se o utilizador quiser dividir o disco em partições. Em seguida deve fazer-se uma formatação lógica que nada mais é do que "instalar" o sistema de ficheiros no dispositivo de armazenamento.

O sistema de ficheiros é uma estrutura que indica como os ficheiros devem ser guardados nas unidades de armazenamento, como o espaço é utilizado e que permite gerir como as partes de um ficheiro podem ficar "espalhada" no dispositivo de armazenamento.

Até ao momento não existem sistemas de ficheiros criados especificamente para cartões de memória MMC/SD. Analisando os sistemas de ficheiros existentes, indicam-se os que têm características consideradas mais relevantes para o uso neste tipo de dispositivos e a respectiva referencia donde foi retirada a informação sobre o sistema de ficheiros:

- FAT16/32 [8]
- JFFS2 [4]
- YAFFS [9]

O sistema de ficheiros JFFS2 e o YAFFS são *Open Source* e até ao momento suportam a maioria das memórias NAND, e que foram desenvolvidos especificamente para o uso neste tipo de memórias embutidas. O YAFFS é mais rápido e consome menos energia que o JFFS2, mas o JFFS2 disponibiliza a possibilidade de compressão e descompressão o que pode ser bastante útil em memórias FLASH de pequena capacidade. O FAT foi desenvolvido para discos rígidos, mas pode ser usado neste tipo de dispositivos e tem a vantagem de ser portável para quase todos os sistemas operativos.

Tanto o JFFS2 como o YAFFS disponibilizam gestão de blocos com defeito, nível de uso, correcção de erros e este tipo de características permite o seu uso a nível industrial, por exemplo.

Os cartões de memória MMC/SD tem um limite de escritas por sector, que vai de cem mil a um milhão. Ora se considerarmos o pior caso (cem mil) em que o utilizador faz vinte e quatro escritas por dia no mesmo sector do cartão ao longo de todos os dias do ano, demoraria cerca de trinca e quatro anos a ocorrer desgaste nesse sector. Como neste tipo de dispositivos normalmente se tem um tempo de vida relativamente curto, considera-se que mesmo em casos de uso exagerado do cartão, não há necessidade de efectuar controlo do nível de uso.

Existem alguns requisitos para o sistema de ficheiros:

- Portabilidade - pretende-se que o sistema de ficheiros adoptado permita a leitura do cartão noutros sistemas (Windows, Linux, etc.).
- Velocidade de acesso - O sistema de ficheiros tem que ter velocidade de acesso para que sejam acedidos ficheiros do tipo *log* de dados, pelo que não será necessário o acesso a vários ficheiros ao mesmo tempo.
- Espaço físico ocupado no sistema - Devido ao sistema ser implementado num sistema embutido com espaço físico limitado, o sistema de ficheiros terá que ocupar o menor possível quer em hardware quer em software. Terá que ser encontrado um equilíbrio performance vs espaço físico.

2.3.1 Master boot record

A *Master boot record* contém informações acerca do tipo da partições, alguns dados sobre as mesmas (se são partições activas, ou seja, de arranque, tipo de sistema de ficheiros, posição no

disco, tamanho, etc.) e indica o sistema esquema de partições existente. Podem existir quatro partições (primárias) mais as partições estendidas. Dentro das partições estendidas podem-se criar até 63 sub-partições (lógicas) cuja Tabela de Partições se encontra descrita algures dentro de toda a partição estendida.

À partida, para todos os sistemas operativos, a *Master boot record* está localizada no primeiro sector do dispositivo de armazenamento. É a primeira informação que o sistema operativo procura quando acede a um dispositivo. Como MBR contem informações ácerca da tabela de partições, se por algum acaso acontecer alguma coisa aos 512 bytes da MBR, toda a informação no disco é perdida.

Em [8] pode ser consultada uma tabela que descreve os valores da MBR.

2.3.2 FAT16

Este sistema de ficheiros foi introduzido com o MS-DOS em 1981 sendo portanto um sistema já bastante antigo. Foi desenhado originalmente para se poder transferir e manipular ficheiros em disquetes. Ao longo dos anos teve algumas actualizações, onde, por exemplo se introduziu capacidade de gravar ficheiros com nomes maiores que o original 8.3 caracteres. Uma das grandes vantagens deste sistema de ficheiros é que é compatível com uma grande quantidade de sistemas operativos, incluindo Windows 95/98/Me, OS/2, Linux, e algumas versões do UNIX.

O sistema FAT (ou FAT16) consegue trabalhar com 65536 *clusters*. Esse número é obtido elevando o número 2 a 16 (daí a terminologia FAT16). Mas, na verdade, o sistema FAT16 usa apenas 65525 *lusters* por disco (ou partição). É importante frisar que o tamanho do *cluster* deve obedecer também a uma potência de 2: 2 KB, 4 KB, 8 KB, 16 KB e 32 KB, ou seja, não é possível ter *cluster* de 5 KB, 7 KB, etc. O tamanho dos *clusters* no sistema FAT também é uma potência de 2. O limite máximo de tamanho para uma partição em FAT16, é de 2 GB (correspondente a 2 elevado a 16).

Devido a esta grande limitação que consiste em ter um número máximo e fixo de *clusters* por partição, este sistema de ficheiros não consegue acompanhar a evolução tecnológica, e dai que à medida que o espaço dos discos rígidos vai aumentando, o tamanho de cada *cluster* também aumenta. Numa partição de 2GB, cada *cluster* é de 32 kilobytes, o que significa que o ficheiro mais pequeno, no disco, irá ocupar 32KB de espaço.

Ainda existem mais algumas limitações neste sistema de ficheiros, como por exemplo: um número máximo de entradas que podem ser escritas na raíz (512 ficheiros e/ou pastas); não suporta compressão; encriptação ou segurança avançada baseada em listas de acesso.

2.3.2.1 Informação Estrutural

A estrutura básica do sistema de ficheiros FAT está divida em quatro secções.

- Região reservada (Boot da FAT16)
- *File Allocation Table*

- Directório da raíz
- Região de dados

2.3.2.2 Volume ID FAT16

Em anexo pode-se consultar os valores contidos no volume ID do sistema de ficheiros FAT16.

2.3.2.3 FAT - File Allocation Table

A estrutura FAT contém uma lista ligada dos ficheiros do sistema de ficheiros. Qualquer que seja o ficheiro, num directório ou subdirectório, tem que ter um apontador para o *cluster* onde se inicia esse ficheiro. A FAT pode indicar o próximo *cluster* do ficheiro ou indicar o fim de ficheiro, ver a tabela 2.2. Cada *cluster* tem um número associado a FAT.

Valor	Descrição
00h	*Cluster* livre
01h - 02h	Reservado
03h - FFEFh	Número do próximo *cluster*
F7h	Um ou mais *clusters* danificados
F8h - FFh	Fim de ficheiro

Tabela 2.2: FAT

Estrutura dos directórios Cada entrada numa lista de directórios tem um comprimento de 32 bytes. Só há um directório com uma localização fixa, o directório da raíz (root). O tamanho total do directório da raíz está definido na *boot* da FAT.

Os subdirectórios são criados limpando toda a informação contida num *cluster*. Depois do *cluster* não ter informação reserva-se esse *cluster* e grava-se as tabelas com a informação acerca dos ficheiros e outros subdirectórios que possam existir nesse directório. Em cada directório são criadas duas entradas padrão, uma para o próprio directório "."e outra para o directório "pai"deste "..".

As tabelas de directórios têm um comprimento de 32 bytes e com o formato indicado pela tabela 2.3.

No campo atributos coloca-se o tipo de ficheiro, da forma indicada na tabela 2.4.

2.3.3 FAT32

O sistema de ficheiros FAT32 apareceu originalmente com o Windows 95 *Service Pack 2* e é na realidade uma extensão do sistema de ficheiros FAT6, mas estão disponíveis em número muito maior de *clusters* por partição.

O tamanho máximo da partição em FAT32 é de 2 TB. Mas se fizermos as contas notamos que 2 elevado a 32 é equivalente a 128 TB. Pode parecer confuso, mas o número máximo de *clusters*

Offset	Tamanho	Descrição
00h	8 bytes	Nome do ficheiro
08h	3 bytes	Extensão do ficheiro
0Bh	1 bytes	Atributos
0Ch	1 bytes	Reservado para o windows NT
0Dh	1 bytes	Milisegundos
0Eh	2 bytes	Tempo
10h	2 bytes	Data
12h	2 bytes	Ultima vez acedido
14h	2 bytes	Reservado para FAT32
16h	2 bytes	Tempo da ultima escrita
18h	2 bytes	Data da ultima escrita
1Ah	2 bytes	Cluster onde começa o ficheiro/directoria
1Ch	4 bytes	Comprimento do ficheiro/directoria

Tabela 2.3: Entradas dum directório

no caso do FAT32 não é de 2 elevado a 32. Apesar de seu endereçamento ser de 32 bits, na verdade são usados apenas 28 bits. Com isso, a quantidade máxima de *clusters* seria 2 elevado a 28, que corresponde a 8 TB. Mas segundo a Microsoft, o número máximo de sectores que um disco pode ter é de 2 elevado a 32. Como cada sector tem 512 bytes, o tamanho máximo de um disco no FAT32 acaba por ser de 2 TB.

O FAT32 aumenta a performance global de utilização do disco quando comparamos com o sistema FAT16, e embora traga várias melhorias ao sistema FAT16 também partilha das suas limitações.

O sistema de ficheiros FAT32 estruturalmente é semelhante ao FAT16, com a grande diferença que usa apontadores de 32bits na tabela FAT, podendo assim endereçar um maior valor de *clusters*.

Devido a semelhança com o FAT16, não será descrita a sua estrutura pois seria redundante.

2.3.4 JFFS2

O JFFS2 é um sistema de ficheiros do tipo *log-structured* e foi desenhado especificamente para o uso em sistemas embutidos que usam memórias *flash*. Ao contrário de outros sistemas de ficheiros mais antigos, que foram desenhados para o uso em discos rígidos e que precisam de uma estrutura intermédia para acederem as memórias flash o JFFS2, é colocado directamente na memória *flash*.

Inicialmente estava previsto que fosse portável, em particular, para os sistemas operativos embutidos eCos e Red Hat's. Embora esta portabilidade fosse pretendida, de momento, apenas pode ser usado com o *kernel* do Linux da série 2.4.

Enquanto o JFFS usava apenas um tipo de nó, o JFFS2 é mais flexível, permitindo que sejam definidos novos tipos de nós mantendo a mesma compatibilidade.

Todos os nós começam com um *header* 2.8 que é comum a todos, que contém o comprimento total dos nós, o tipo de nó e um *cyclic redundancy checksum*.

Offset	Descrição
8	Reservado
7	Reservado
6	Nome do ficheiro
5	Modificado
4	Directório
3	Volume
2	Sistema
1	Escondido
0	Leitura apenas

Tabela 2.4: Atributos de um ficheiro

A estrutura é composta por um valor único que identifica a estrutura do nó e o seu significado, uma máscara que indica o comportamento que um *kernel* que não suporta o nó deve ter. Até ao momento existem três tipos de nós que definem e implementam o sistema de ficheiros JFFS2:

- JFFS2_NODETYPE_INODE
- JFFS2_NODETYPE_DIRENT
- JFFS2_NODETYPE_CLEANMARKER

2.3.4.1 Operação

Cada nó físico no dispositivo é representado por uma pequena estrutura jffs2 *raw node ref*, que contém o *offset* da lista, a sua dimensão total e dois apontadores para outros nós. Um para o próximo bloco físico que irá ser apagado e o outro para o próximo nó na lista de nós.

Para e efectuar a *garbage collection* é necessário encontrar uma referência para um certo tipo de nó. Em vez de se terminar a lista ligada com um apontador nulo, o último nó contém um apontador para uma *struct* jffs2 *inode cache*. Esta estrutura indica que a lista ligada chegou ao fim.

2.3.4.2 *Mounting*

A montagem de sistema de ficheiros JFFS2 envolve 4 operações:

- Leitura de toda memória, verificação do CRC em todos os nós e alocação de todas as referências dos nós. Criação de uma *hash table* com base nessas referências;
- Mapeamento de todos os dados ligados aos nós. Apagar todos os nós obsoletos;
- Apagar todos os nós que não estão ligados e sempre que um directório é apagado, é reiniciada uma nova pesquisa;
- Libertar toda a informação temporária que foi colocada em cache

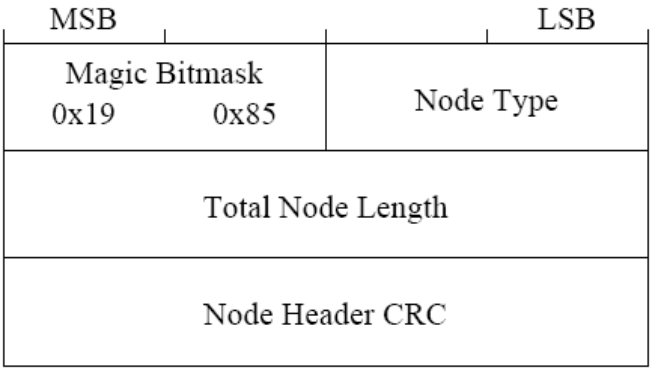

Figura 2.8: Cabeçalho de um nó em jffs2 [4]

2.3.5 YAFFS

YAFFS é a sigla de "yet another flash file system"e foi desenhado especificamente para o uso em dispositivos *flash* NAND. Foi então, desenhado para explorar as características das memórias NAND, maximizando a performance deste tipo de dispositivos. O sistema de ficheiros usa técnicas de *journaling* e verificação ECC de forma a compensar as falhas que ocorrem nas memórias do tipo *flash* NAND.

Este sistema de ficheiros em vez de ser desenvolvido directamente no *kernel* do Linux, foi desenvolvido num programa que pode ser executado como uma aplicação. Esta aplicação está dividida em quatro módulos:

- yaffs guts: Algoritmos do sistema de ficheiros, escritos em C.
- yaffs fs.c: Interface com camada de VFS do Linux.
- nand interface: Camada física.
- Portability functions: Funções de diversos tipos para o uso genérico do sistema de ficheiros.

Este sistema de ficheiros, devido à sua estrutura, permite flexibilidade para o teste e desenvolvimento. O sistema de ficheiros tem alguma portabilidade, por exemplo, para o Windows CE e Linux.

2.3.5.1 Resultados da análise do YAFFS

Depois da análise da informação encontrada verifica-se que é um sistema de ficheiros robusto, com uma boa performance. A leitura e a escrita são bastante rápidas e o *garbage collection* não causa uma latência elevada. Tem alguma portabilidade e embora seja desenhado para as memórias NAND "puras"tem características interessantes que podem ser usadas nos cartões MMC/SD.

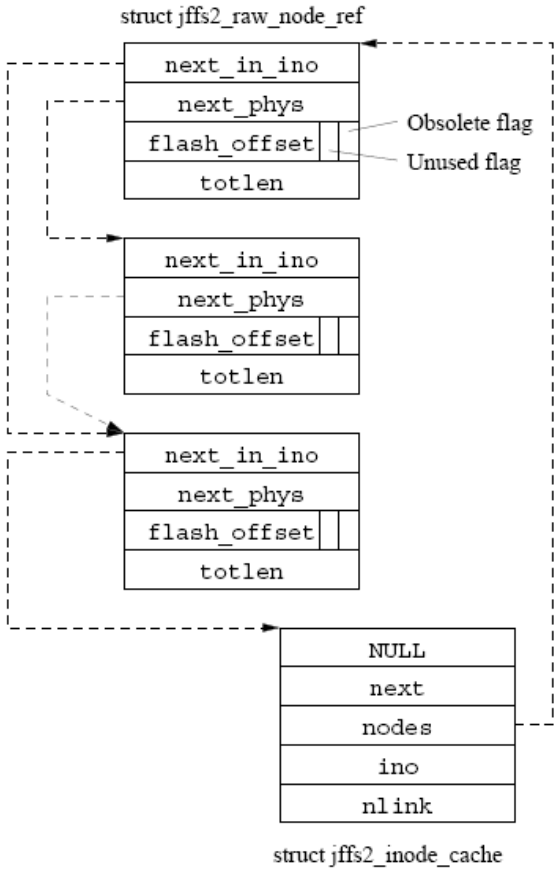

Figura 2.9: Estrutura do jffs2 [4]

2.4 Microntrolador e Microprocessador

Para efectuar a camada do sistema de ficheiros é necessário o uso de um Microntrolador ou Microprocessador. A escolha irá ser feita de forma a que o sistema final cumpra a especificação prevista.

Como o sistema vai ser desenvolvido numa placa de desenvolvimento com a FPGA Spartan 3, a ideia passa por procurar um Microntrolador ou Microprocessador que se possa embutir na FPGA, os chamados *soft processor*. Um *soft-processor* é uma propriedade intelectual (IP) implementada usando uma linguagem de descrição de hardware (HDL).

Os maiores benefícios desta abordagem incluem a facilidade de mudança de configuração do sistema de forma a dar maior prioridade a preço ou performance, menor tempo de desenvolvimento, fácil integração com os circuitos da FPGA e a inexistência do risco da obsolescência.

Este tipo de sistemas permitem com alguma facilidade adicionar ou retirar periféricos e ajustar o desempenho do microprocessador através do dimensionamento e implementação de circuitos auxiliares, de barramentos e com optimizações do núcleo IP disponibilizadas ao longo do tempo.

Convém também referir desvantagens deste tipo de processadores em relação aos convencionais nomeadamente à sua ainda pequena utilização a nível mundial, embora esteja em franco

crescimento. Consequentemente, ainda existe uma pequena gama de ferramentas e fornecedores relacionadas com os *soft-processors*.

2.4.1 PicoBlaze

Para aplicações mais simples, ou que sejam extremamente especializadas, a Xilinx oferece o PicoBlaze [10].

O PicoBlaze é um microcontrolador de 8 bits de arquitectura RISC Core compacto. Sendo totalmente *free*, possui uma boa relação custo-benefício. Está optimizado para ser utilizado em famílias de FPGAs, tais como a: ProTM, SpartanTM, VirtexTM-II e Virtex II.

Ao contrário do MicroBlaze, este microprocessador é fornecido como um arquivo em VHDL ou Verilog de forma a ser usado em FPGAs. Desta forma ele fica, de certa forma, imune a obsolescência de produtos, uma vez que o código fonte é aberto.

O PicoBlaze utiliza uma BRAM na FPGA e ocupa 96 *slices* é possível armazenar até 1024 instruções de programa, que são carregadas automaticamente durante a configuração do FPGA. É capaz de executar no máximo 44 a 100 milhão de instruções por segundo (MIPS), numa Spartan 3, um número aceitável dadas as características deste microcontrolador.

O núcleo do PicoBlaze é totalmente incorporado na FPGA e não necessita de recursos externos sendo extremamente flexível. As suas funcionalidades básicas são facilmente ampliadas através dos pinos de entrada e saída, permitindo a adaptação de circuitos externos para ampliar as suas funcionalidades. O PicoBlaze contém uma grande número de I/O que são flexíveis e de custo mais baixo em comparação a controladores do tipo *off-the-shelf*.

Integrar o PicoBlaze numa FPGA reduz o espaço de ocupação em silício e custo de desenvolvimento. Os programas das aplicações são escritos em assembler e são compilados usando o Xilinx KCPSM3 assembler.

Devido a maior dificuldade de programação usando a linguagem assembly efectuou-se uma pesquisa afim de encontrar um compilador de C para o PicoBlaze. Verificou-se que não existem compiladores oficiais da Xilinx devido a:

> "Whilst on the subject of compilers it is worth considering if they are really suitable for PicoBlaze anyway. Please understand that I am not against having a compiler but we should keep in mind that it does only support a program space of 1024 instructions and has a scratch pad memory of only 64-bytes."
>
> Ken Chapman
> Senior Staff Engineer, Applications Specialist, Xilinx UK

No entanto como pretendido, encontrou-se um compilador desenvolvido por Francesco Poderico [1] a nível a amador.

[1] http://www.asm.ro/fpga/ Acesso em Janeiro/2009

As principais características do PicoBlaze são:

- 16 registradores de 8 bits para uso geral
- memória de programa de 1024 * 18 bits automaticamente carregada durante a configuração do dispositivo
- ULA de 8 bits com *flags* de *carry* e zero
- 256 portas de entrada mais 256 portas de saída
- RAM interna de 64 bytes
- performance previsível, sempre 2 ciclos de *clock* por instrução

2.4.1.1 Software

Para desenvolver código para o PicoBlaze foram encontradas apenas duas soluções:

- Em instruções assembly [10]
- Compilador de C [2]

KCPSM3 Assembler O assemblador do PicoBlaze vem incluindo no *kit* do PicoBlaze. É um simples executável DOS com mais dois ficheiros a acompanhar ROM_form.v e ROM_form.coe.

Para criar um programa utiliza-se um editor de texto para escrever o código e grava-se o ficheiro com a extensão .psm, numa janela DOS executa-se o assemblador sendo gerado um modulo, escrito Verilog, que define a BRAM do PicoBlaze.

Sempre que um erro no código é encontrado, o ficheiro Verilog não é gerado e o assemblador indica o erro por forma a poder ser corrigido.

A lista de instruções (*instruction set*) usada para criar o código assembly para o PicoBlaze pode ser encontrada no manual [10].

Compilador C Foi encontrado um compilador C desenvolvido por Francesco Poderico. A compilação em C trás diversas vantagens a programação, como por exemplo:

- É bastante mais fácil programar em C do que em assembly
- C é portável
- Fazer debug em assembly pode ser bastante complexo

[2] http://www.asm.ro/fpga/ Acesso em Janeiro/2009

A grande desvantagem do compilador C é que gera um código que pode não ser tão curto como se fosse escrito em assembly, trazendo desvantagens ao nível de desempenho do próprio microcontrolador.

Este compilador não faz qualquer optimização do código gerado, podendo tornar-se, por isso, uma grande desvantagem e pondo mesmo em risco a sua utilização.

Para compilar o código executa-se uma aplicação em ambiente DOS, sendo bastante fácil a sua utilização.

2.4.2 Microblaze

O Microblaze é comercializado pela Xilinx [11] como parte do software EDK. Este *software* é na realidade resultado de um conjunto de várias aplicações já desenvolvidas pela Xilinx e que oferecem suporte ao desenvolvimento de projectos baseados em lógica programável, sejam eles projectos que envolvam o uso de *soft-processors* ou não.

A ocupação de lógica do MicroBalze é de aproximadamente 800 a 2600 look-up tables. Existem alguns sistemas operativos já portados para o MicroBlaze, entre os quais, o Nucleus da Mentor Graphics, o ThreadX, o uC/OS-II da Micrium, uClinux da LinuxWorks, entre outros.

O MicroBalze é um processador RISC (Reduced Instruction Set Computing) de 32 bits, com arquitectura do tipo HARVARD. Este dispositivo é evidentemente optimizado para utilização nas FPGAs da Xilinx. A organização básica deste microprocessador consiste em 32 registos de uso geral, uma ULA (unidade lógica aritmética) e dois níveis de interrupções.

Algumas características do Microblaze:

- Profundidade do pipeline
- lógica de debug de hardware
- Memória cache de instrução
- Memória cache de dados
- Suporte à captura de excepções de hardware
- Unidade de ponto flutuante
- Divisor em hardware

2.5 Opencores

OpenCores [3] é um portal web criado por Damjan Lampret em Outubro de 1999. O seu principal objectivo é desenvolver e publicar módulos de hardware que estejam sob a Lesser General Public License (LGPL) para software. O código fonte está completamente disponível e aberto para o uso publico.

[3] www.opencores.com Acesso em Janeiro/2009

Na base de dados do portal foram encontrados muitos projectos já testados e estáveis, nomeadamente na secção "Communication controller" podem-se encontrar projectos relacionados com interfaces de comunicação com cartões MMC/SD. Partes do código desses projectos poderá ser utilizado, integrado e até modificado de forma a cumprir os objectivos da Dissertação.

2.6 Sistema de ficheiros em FPGAs

Em 2006 foi realizado na Faculdade de Engenharia da Universidade do Porto um projecto com objectivos semelhantes embora com abordagens completamente diferentes [12].

Este projecto em vez de ter sido desenvolvido num misto de software e hardware foi completamente desenvolvido em hardware.

O objectivo do projecto foi o de implementar um sistema que permitisse a leitura e escrita de dados em ficheiros armazenados num cartão de memória do tipo SD e com o sistema de ficheiros FAT16. No momento da conclusão do relatório tinham sido atingidos os seguintes objectivos:

- Leitura de sectores de 512 bytes do cartão
- Execução dos comandos de leitura e deslocamento do ficheiro através de simulação

O sistema de ficheiros foi parcialmente implementado, pois, devido a sua elevada complexidade, não se conseguiu sintetizar, embora tenha sido testado em ferramentas de simulação com sucesso.

2.7 Projectos em software semelhantes

Foram encontradas algumas bibliotecas *open source* [4], de projectos para sistemas embutidos com cartões MMC/SD, que envolvem o gravação de ficheiros num sistema de ficheiros FAT. Todos estes projectos referem-se apenas camada do sistema de ficheiros, mas existem vários exemplos de projectos na internet em que são utilizados. Os projectos exemplos encontrados em que se usou estes projectos eram para microcontroladores convencionais, não tendo sido encontrada nenhuma implementação para soft processors.

2.7.1 EFSL (Embedded Filesystem Library)

Este projecto corre num PC (GNU/Linux), TMS C6000 DSP TI e no ATmega from Atmel. A perfomance deste sistema aumenta com o aumento da capacidade da memória disponível. A implementação do sistema de ficheiros necessita de 1.5KB de RAM e por isso se torna ideal para sistemas embutidos com pouca memória.

[4] http://sourceforge.net/projects/efsl Acesso em Janeiro/2009; http://elm-chan.org/fsw/ff/00index_e.html Acesso em Janeiro/2009

2.7.2 FF/TFF (FatFile and TinyFatFile)

Este projecto já foi testado num microcontrolador AVR (ATmega64, 8bit RISC), H8/300H (Renesas HD64F3694, 16bit CISC), PIC (PIC24FJ64GA002, 16bit RIS-C), TLCS (Toshiba TMP-86FM29, 8bit CISC), V850ES (NEC uPD70F3716, 32bit RISC) num MMC/SD, CF e ATA HDD.

Foram encontrados vários projectos a usarem esta implementação do sistema de ficheiros. Nos exemplos vistos, verificou-se que foram utilizados drivers, desenvolvidos especificamente para o microcontrolador utilizado nesse projecto, que implementaram a camada de baixo nível.

Capítulo 3

Especificação do Sistema a implementar

Os cartões MMC e SD são dispositivos amovíveis com alguma capacidade de armazenamento de dados, que utilizam um protocolo de comunicação bem definido [3] [5]. A interface especificada é possível de implementar usando somente lógica reconfigurável de uma FPGA. Ambos os tipos de cartões de memoria SD standard e MMC têm protocolos compatíveis, podendo-se criar um único sistema de acesso.

Existem alguns projectos na base de dados da OpenCores [1] e em [12] que poderão servir de exemplo, reutilização e integração de código no desenvolvimento e implementação do projecto ao nível da camada de acesso aos cartões MMC/SD.

Terá que se desenvolver um sistema de ficheiros inovador, no sentido em que terá que se adaptar aos requisitos especifico deste projecto. Dos sistemas estudados o que mais se adequa e será a base do sistema a desenvolver é o FAT 16, pois está bem documentado, consegue-se aceder a um ficheiro num curto "espaço" de tempo, é um sistema de ficheiro simples e não ocupará um espaço (memória de programa) exagerado no sistema, por último é portável para qualquer sistema operativo.

Uma comparação entre o Picoblaze e o Microblaze, no âmbito das características procuradas para este projectos, verifica-se que o Picoblaze é o mais adequado. Foi elaborada uma tabela 3.1 simples mas que mostra algumas diferenças de relevo entre as duas possibilidades.

Característica	Picoblaze	Microblaze
Área ocupada	++	-
Frequência de relógio	+/-	++
Custo	++	–
Ferramentas existentes	-	++
Resultado	++	+

Tabela 3.1: Picoblaze vs Microblaze

[1] www.opencores.com Acesso em Janeiro/2009

3.1 Arquitectura do sistema

Pretende-se adoptar uma arquitectura lógica, em camadas, de modo a que o sistema tenha características do tipo:

- Transparência
- Uma complexidade distribuída
- Adaptável e escalável
- Que continue funcional, mesmo com a evolução rápida das tecnologias subjacentes
- Robustez
- Estabilidade

O sistema será desenvolvido numa configuração mista de *hardware* e *software*. A camada física ficará encarregue de executar todo o protocolo de comunicação com o cartão. A camada do sistema de ficheiros será desenvolvida quase na sua totalidade em software, podendo parte ser feita em hardware, por forma a libertar o microcontrolador de algumas tarefas.

À partida, nesta configuração, o sistema ficará mais rápido, mas o nível de ocupação da FPGA aumenta.

Existirão então duas camadas, que terão que ser desenvolvidas e implementadas, neste sistema de interface, entre o micro e o cartão de memória:

- Camada do sistema de ficheiros
- Camada física

A estrutura de camadas implementada pode ser vista na figura 3.1.

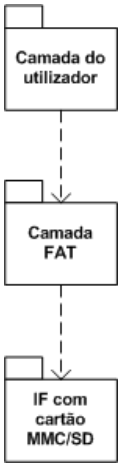

Figura 3.1: Camadas do Sistema

Existe ainda uma terceira camada a Aplicação que embora não faça parte do produto final deste projecto, terá ser desenvolvida/implementada por motivos de testes das duas camadas inferiores.

3.2 Casos de uso

Na figura 3.2 exemplificam-se os possíveis os casos de utilização.

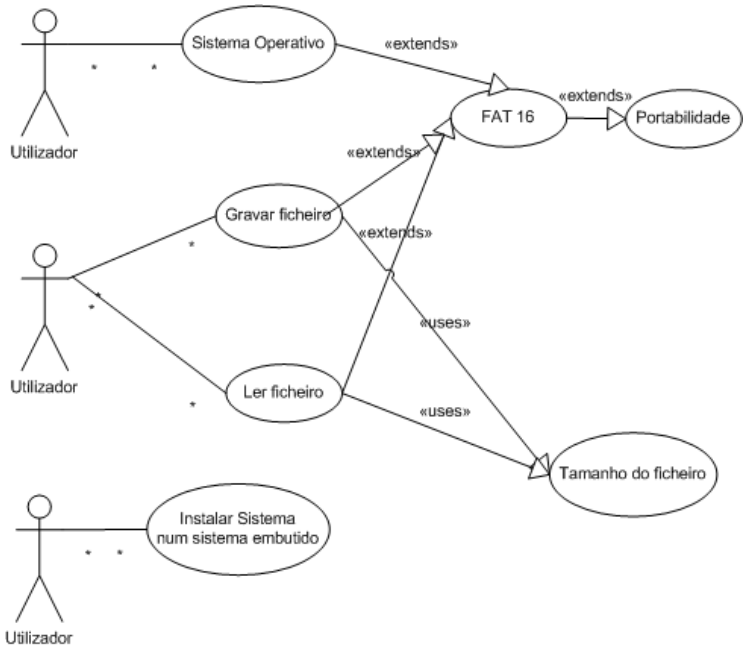

Figura 3.2: Casos de utilização

Capítulo 4

Circuito de interface com cartões MMC/SD

Como a placa de desenvolvimento da FPGA Spartan 3 externamente não tem uma interface para cartões de memoria, teve que se começar por implementar uma. Posteriormente desenvolveu-se as componentes de hardware embebidas na própria FPGA.

Neste capitulo explica-se as opções tomadas e as funções dos componentes de hardware desenvolvidos para comunicar com o cartão.

Utilizou-se a base de dados da OpenCores[1] para reutilizar, adaptar e integrar código no projecto. Desta forma permitiu acelerar a execução do projecto.

4.1 Circuito exterior de interface do cartão com a FPGA

O circuito externo A tem que cumprir tanto a especificação e topologia da placa de desenvolvimento como dos cartões MMC/SD. Os cartões utilizam uma interface SPI e a placa de desenvolvimento tem 3 conectores de expansão disponíveis para se poder utilizar. Cada um com trinta e sete pinos de I/O mais um de GND, um de 5v e um de 3,3v. O cartão requer que os sinais de que tanto a alimentação como os sinais de dados sejam de 3,3v que é o fornecido pela placa estando então este requisito satisfeito.

A interligação entre os pinos foi implementada com *pull-up* e *pull-down*, de forma a que não ocorram oscilações indesejadas no circuito.Como o DI e DO estão normalmente com o valor lógico "1", devem ser *pull-up*. De acordo com a especificação dos MMC/SD, é recomendada uma resistência de 50k a 100kOhm para criar os registo *pull-up*. O sinal de relógio não é mencionado na especificação porque é gerado pelo microcontrolador. Quando há possibilidade de oscilar deve ser colocado no estado lógico "0".

[1] www.opencores.com Acesso em Janeiro/2009

Relativamente a situação de *Hot insertion/removal* é preciso ter algumas considerações em conta. Se a fonte de alimentação estiver ligada directamente ao socket do cartão no momento de contacto entre o cartão e o socket, ocorre uma queda de tensão, devido ao condensador. Com a inserção de uma bobina é possível melhorar o circuito, diminuindo a queda de tensão.

4.2 Arquitectura

A camada de acesso e interface ao leitor de cartões foi quase na totalidade desenvolvida e implementada recorrendo a FPGA Spartan 3. Nesta secção apresentam-se os módulos que integram o subsistema de acesso e interface aos cartões MMC e SD. A arquitectura do sistema desenvolvido pode ser visto na figura 4.1.

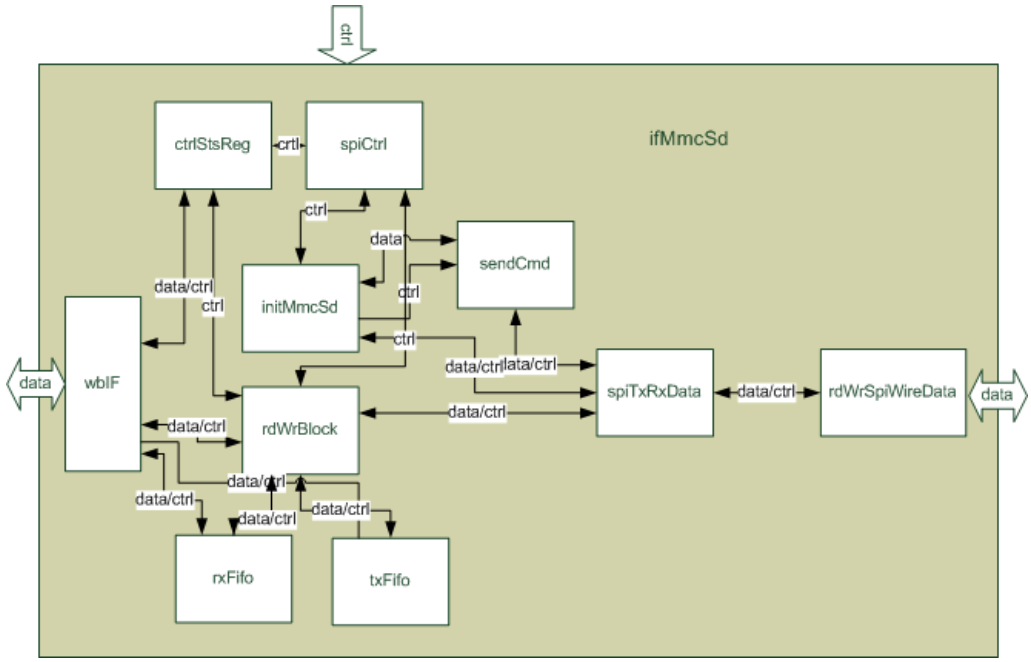

Figura 4.1: Arquitectura lógica da Interface MMC/SD

4.2.1 Módulo *spiMasterWishBone*

Este módulo faz a interface entre este sistema e o exterior. Pretende-se que a interface seja desenvolvida por forma a que a camada de baixo nível seja independente e transparente para a camada superior, tornado este subsistema escalável. Assim implementou-se este módulo de forma a que a comunicação com o exterior cumpri-se a especificação (*standard*) Wishbone [13].

Na tabela 4.1 pode-se ver a interface implementada.

Este módulo internamente utiliza o endereço(barramento "address") para desmultiplexar os dados E/S para o módulo pretendido:

- CTRL_STS_REG_BASE

Pino	E/S	Tamanho	Descrição
clk	Entrada	1	Sinal de relógio
address	Entrada	8	Barramento de endereços
dataIn	Entrada	8	Barramento de dados de entrada
dataOut	Saída	8	Barramento de dados de saída
writeEn	Entrada	1	Bit de comando usado para indicar uma escrita ou leitura
strobe_i	Entrada	1	Bit de comando que sincroniza a leitura ou escrita de dados
ack_o	Saída	1	Bit de comando que responde ao sinal de strobe indicando a conclusão da operação
ctrlStsRegSel	Saída	1	Bit que controla o acesso acesso aos registos internos do circuito
rxFifoSel	1	Saída	Bit que controla o acesso da fifo de leitura
txFifoSel	1	Entrada	Bit que controla o acesso da fifo de escrita
dataFromCtrlStsReg	8	Entrada	Barramento de dados entrada do modulo de registos internos do circuito
dataFromRxFifo	8	Entrada	Barramento de dados entrada do modulo da fifo de leitura
dataFromRxFifo	8	Entrada	Barramento de dados entrada do modulo da fifo de escrita

Tabela 4.1: spiMasterWishbone

- RX_FIFO_BASE
- TX_FIFO_BASE

Outra função deste módulo é a de gerar o sinal de *ack* quando o *dataIn* é lido e quando a resposta de dados está disponível, em *dataOut*.

4.2.2 Módulo *ctrlStsRegBI*

Este módulo tem duas funções principais:

1. Faz a actualização dos registos utilizados neste subsistema (SPIInterface), através das indicações recebidos do exterior e interior ao próprio sistema. Para actualizar um registo a partir do exterior terá que se verificar o valor do registo a aceder no barramento *address*, o *writeEn* com o valor "1". A escrita é síncrona com o sinal de relógio e o *strobe_i* tem que ter o calor "1", já a leitura é assíncrona.

2. Outra função deste módulo é realizar o sincronismo entre sinais de relógio. O subsistema (SPIInterface) para operar utiliza a frequência de relógio *spiSysClk* de 50MHz, de forma a fazer uso da frequência máxima permitida pelos cartões MMC/SD. Acontece que os sinais provenientes do barramento exterior podem operar a uma frequência superior e então, nesta

situação, terá que se realizar um sincronismo de todos os sinais exteriores para a frequência interna deste sistema. Indica-se por exemplo a seguinte situação, para que melhor se perceba:

É recebido o sinal exterior de *reset*, quando o sinal de *clk* é gerado a uma frequência de 400MHz e o de *spiSysClk* está a ser gerado a 50MHz, o sinal de *reset* tem a duração de 1 ciclo de relógio ($1/400M$). Se não realizarmos uma sincronização, existe uma forte probabilidade de o sinal de *clk* ocorrer entre duas transições do *spiSysClk* e então o *reset* não será efectuado no sistema.

Para realizar a sincronização assumiu-se que as frequências cumprem o seguinte requisito $clk < 5 * spiSysClk$ e implementou-se um *shift resgister* de seis andares.

Quando ocorre um sinal exterior de *reset* coloca-se todos os andares do *shift resgister* com o valor "1"e esses valores vão então sendo deslocados e colocando um "0"a cada ciclo de *clk*. Entretanto é sempre colocado no *rst* global do subsistema o último valor desse *shift register*, ou seja, vai existir um "1"durante 6 ciclos de *clk* e desde que $clk < 5 * spiSysClk$ o subsistema vai sempre amostrar o *reset* gerado.

Usou-se esta técnica de sincronismo para os todos os sinais que precisem de ser sincronizados entre os duas frequências de relógio:

- *Reset* “rst”
- Sinal que inicia a operação “spiTransCtrl”
- Sinal de actividade “spiTransSts”

Na tabela 4.2 pode-se ver a interface implementada.

4.2.3 Módulo *spiCtrl*

Módulo que faz o controlo das operações (leitura, escrita e inicialização) que estão a ser executadas no sistema, para que não ocorram duas operações em simultâneo. Não pode acontecer, por exemplo, uma leitura enquanto ainda está a ser efectuada uma escrita. Este módulo também controla a actualização do registo *spiTransCtrlS* (busy ou not_busy), registo global do estado do sistema.

O módulo é implementado através de uma máquina de estados simples, que espera por um pedido para inicializar uma operação. Quando recebe um pedido, responde com um sinal de inicio de operação, se o estado do sistema assim o permitir ao mesmo tempo que coloca o sistema em *busy*. Enquanto a operação decorre não responde a outros pedidos e espera que a operação actual finalize, quando receber indicação de que a operação finalizou volta ao estado de espera e aceita outros pedidos.

A interface implementada pode ser vista na tabela 4.3.

Pino	E/S	Tamanho	Descrição
clk	Entrada	1	Sinal de relógio
rstFromWire	Entrada	8	Sinal de reset
dataIn	Entrada	8	Barramento de dados de entrada
dataOut	Saída	8	Barramento de dados de saída
address	Entrada	8	Barramento usado para saber a que registo se pretende aceder
writeEn	Entrada	1	Bit de comando usado para indicar uma escrita ou leitura
strobe_i	Entrada	1	Bit de comando que sincroniza a leitura ou escrita de dados
spiSysClk	Entrada	1	Sinal de relógio usado para o barramento SPI
spiTransType	Saida	2	Registo que indica o tipo de operação que vai ou está a ser executado
spiTransCtrl	Saida	1	Registo que dá ordem para iniciar a operação
spiTransStatus	Entrada	1	Registo que indica se o circuito está ou não ocupado numa operação
ctrlStsRegSel	Entrada	1	Bit que controla o acesso acesso aos registos internos do circuito
rstSyncToBusClkOut	Saida	1	Sinal de controlo que indica reset aos módulos deste circuito
rstSyncToSpiClkOut	Saida	1	Sinal de controlo, sincronizado com o clk do SPI, que indica reset aos módulos deste circuito

Tabela 4.2: ctrlStsRegBI

Pino	E/S	Tamanho	Descrição
clk	Entrada	1	Sinal de relógio
readWriteSD-BlockRdy	Entrada	1	Sinal que indica o fim da transmissão do bloco de dados pelo barramento SPI
rst	Entrada	1	Sinal de controlo que reinicia o módulo
SDInitRdy	Entrada	1	Indica que inicialização do MMC/SD pode ser executada pelo módulo de inicialização
spiTransCtrl	Entrada	1	Sinal que pede a inicialização da operação
spiTransType	Entrada	2	Registo que indica qual a operação a executar
readWriteSD-BlockReq	Saída	2	Registo indica ao módulo de leitura/escrita para começar a operação
SDInitReq	Entrada	1	Sinal que indica ao bloco de de inicialização para começar a operação
spiCD_n	Saída	1	Sinal activa o chip select do barramento SPI
spiTransSts	Saída	1	Indica se uma operação está a decorrer

Tabela 4.3: spiCtrl

4.2.4 Módulo *initSD*

Este módulo executa o protocolo de inicialização dos cartões de memória MMC/SD, como definido nas especificações [3] [5]. Destaca-se algumas operações que passam despercebidas na

especificação, mas que têm que ser executadas correctamente, para que o cartão arranque correctamente.

Pode-se visualizar no fluxograma 4.2 todo o processo de inicialização executado por este módulo em 4.4 encontra-se uma descrição da interface utilizada.

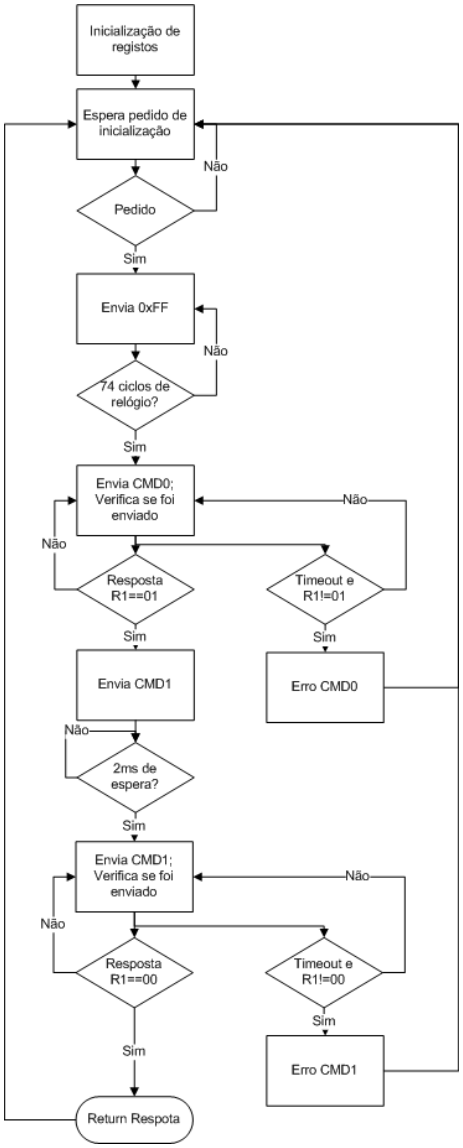

Figura 4.2: Processo de inicialização

Na tabela 4.4 pode-se ver a interface implementada.

4.2.5 Módulo *readWriteSDBlock*

Este módulo executa o protocolo de escrita e de leitura de um cartão MMC ou SD, como definido na especificação. É implementado através de uma maquina de estados, que se divide em duas funções:

- Escrita 4.4

Pino	E/S	Tamanho	Descrição
checkSumByte	Saída	8	CRC enviado para o barramento SPI
clk	Entrada	1	Sinal de relógio
cmdByte	Saída	8	Comando enviado para o SPI segundo o protocolo SPI
dataByte1	Saída	8	Primeiro byte de dados enviado para o SPI segundo o protocolo SPI
dataByte2	Saída	8	Segundo byte de dados enviado para o SPI segundo o protocolo SPI
dataByte3	Saída	8	Terceiro byte de dados enviado para o SPI segundo o protocolo SPI
dataByte4	Saída	8	Quarto byte de dados enviado para o SPI segundo o protocolo SPI
initError	Saída	2	Registo que indica a ocorrência de um erro durante a inicialização do cartão
respByte	Entrada	8	Registo onde fica guardada a resposta do cartão
respTout	Entrada	1	Sinal que indica a ocorrência de Time Out
rst	Entrada	1	Sinal de controlo que reinicializa o módulo
rxDataRdy	Entrada	1	Sinal que indica que foi lido um byte do barramento SPI
rxDataRdyClr	Saída	1	Sinal que indica que o barramento está livre para efectuar leituras
SDInitRdy	Saída	1	Indica que o modulo está disponível para começar a inicialização do cartão
SDInitReq	Entrada	1	Sinal que indica para começar o a inicialização do cartão
sendCmdRdy	Entrada	1	Sinal que indica que se pode enviar um comando para o barramento SPI
sendCmdReq	Saída	1	Pedido para enviar comando para o barramento SPI
spiClkDelayIn	Entrada	8	O cartão MMC e SD necessitam de ser inicializados com relógio SPI de 400KHz este valor permite ajustar esse valor em relação ao relógio de sistema
spiClkDelayOut	Saída	8	Valor que indica o período de relógio SPI de forma a ter 400Khz
spiCS_n	Saída	1	Sinal que activa o pino Chip Select do cartão
txDataEmpty	Entrada	1	Sinal que indica que se pode enviar um byte para o barramento SPI
txDataFull	Entrada	1	Sinal que indica que não se pode enviar um byte para o barramento SPI
txDataOut	Saída	8	Registo com o byte a enviar para o barramento SPI
txDataWen	Saída	1	Pedido para enviar byte para SPI

Tabela 4.4: initSD

- Leitura 4.3

A máquina de estados começa num estado de espera por um pedido de leitura ou escrita de um bloco de dados. Quando recebe um desses pedidos executa o protocolo de comunicação definido na especificação. No fim da operação volta a estado de espera.

Descrição da interface utilizada neste bloco: 4.5

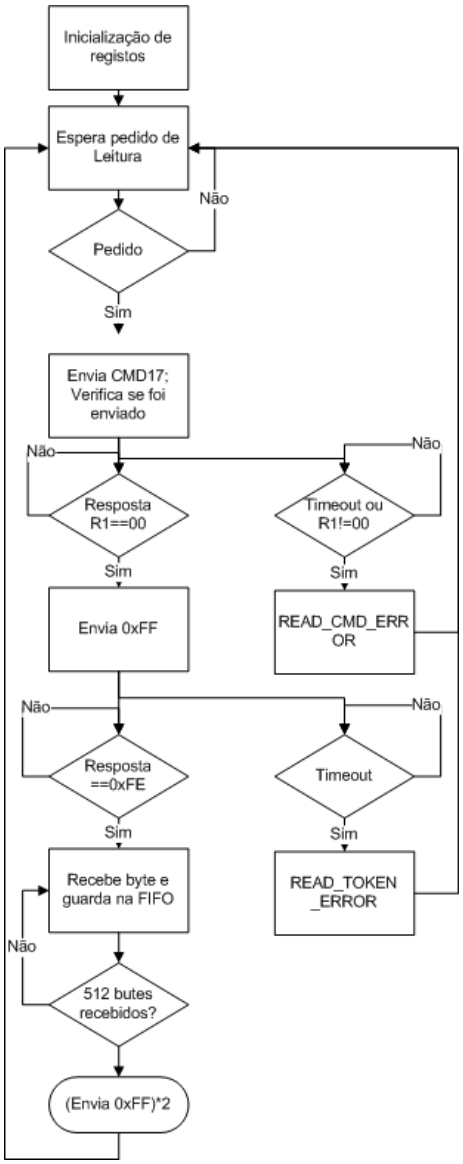

Figura 4.3: Leitura de um bloco

4.2.6 Módulo *sendCmd*

É necessário enviar para o cartão comandos num formato especifico (ler 2.2 ou consultar [3] [5]), então no sistema terá que ser criado um circuito que faça formatação do código binário de forma a que se cumpra esse formato.

Analisando o subsistema verifica-se que mais que um módulo de *hardware* tem que enviar comandos para o cartão no formato especificado. Nesta situação pode-se optar por duas soluções, ou seja, criar um circuito de envio de comandos em cada módulo, ou então criar um circuito global que possa ser usado por todos os módulos. Optou-se, por criar um único módulo global responsável pela formatação dos comandos, parâmetros e CRC. Consegue-se com esta solução diminuir a área ocupada pelo sistema global, para além de o simplificar.

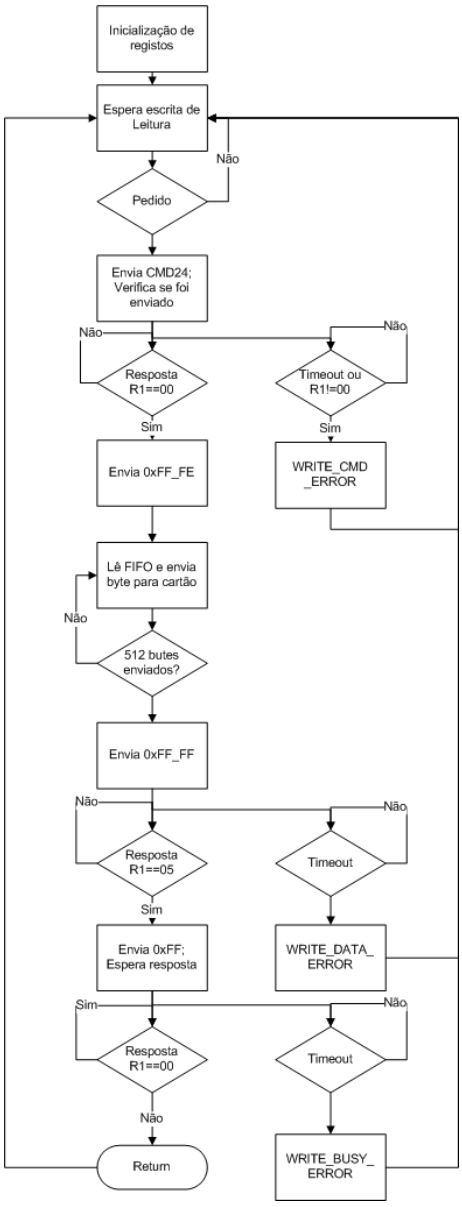

Figura 4.4: Escrita de um bloco

Os módulos que necessitam enviar uma mensagem de comandos para o cartão, colocam a mensagem no respectivo barramento deste módulo. A mensagem é processada, colocada no formato e enviada para o cartão de acordo com as especificações [3] [5].

Devido a mais de um módulo deste circuito ter que aceder a este circuito, é necessário multiplexar os barramentos de entrada de forma a que não ocorram choques entre sinais. Teve-se que implementar um multiplexador dentro do próprio módulo para resolver a situação.

Descrição da interface utilizada neste bloco: 4.6

Pino	E/S	Tamanho	Descrição
blockAddr	Entrada	32	Endereço onde se pretende ler ou escrever um bloco
checkSumByte	Saída	8	CRC enviado para o barramento SPI
clk	Entrada	1	Sinal de relógio
cmdByte	Saída	8	Comando enviado para o SPI segundo o protocolo SPI
dataByte1	Saída	8	Primeiro byte de dados enviado para o SPI segundo o protocolo SPI
dataByte2	Saída	8	Segundo byte de dados enviado para o SPI segundo o protocolo SPI
dataByte3	Saída	8	Terceiro byte de dados enviado para o SPI segundo o protocolo SPI
dataByte4	Saída	8	Quarto byte de dados enviado para o SPI segundo o protocolo SPI
readError	Saída	8	Registo que indica a ocorrência de um erro na leitura de um bloco
readWriteSDBlockRdy	Saída	1	Sinal que indica o fim da transmissão do bloco de dados pelo barramento SPI
readWriteSDBlockReq	Entrada	1	Registo indica para começar a operação de leitura/escrita
respByte	Entrada	8	Registo onde fica guardada a resposta do cartão
respTout	Entrada	1	Sinal que indica a ocorrência de Time Out
rst	Entrada	1	Sinal de controlo que reinicializa o módulo
rxDataIn	Entrada	8	Byte que se pretende enviar pelo barramento SPI
rxDataRdy	Entrada	1	Sinal que indica que foi lido um byte do barramento SPI
rxDataRdyClr	Saída	1	Indicar que o barramento está disponível para fazer leituras
rxFifoData	Saída	8	Dados da FIFO de leitura
rxFifoWen	Saída	1	Sinal que activa a FIFO de leitura
sendCmdRdy	Entrada	1	Sinal que indica que se pode enviar um comando para o barramento SPI
sendCmdReq	Saída	1	Pedido para enviar comando para o barramento SPI
spiCS_n	Saída	1	Sinal que activa o pino Chip Select do cartão
txDataEmpty	Entrada	1	Sinal que indica que se pode enviar um byte para o barramento SPI
txDataFull	Entrada	1	Sinal que indica que não se pode enviar um byte para o barramento SPI
txDataOut	Saída	8	Registo com o byte a enviar para o barramento SPI
txDataWen	Saída	1	Pedido para enviar byte para SPI
txFifoData	Saída	8	Dados da FIFO de escritaa
txFifoRen	Saída	1	Sinal que activa a FIFO de escrita
writeError	Saída	2	Registo que indica a ocorrência de um erro na escrita do bloco

Tabela 4.5: readWriteSDBlock

4.2.7 Módulo *spiTxRxData*

A comunicação com o cartão é feita através do protocolo SPI, sendo o módulo *readWriteSPI-WireData* responsável pela realização física desse protocolo. Devido aos múltiplos módulos de

Pino	E/S	Tamanho	Descrição
checkSumByte_1	Entrada	8	Byte CRC, do barramento 1
checkSumByte_2	Entrada	8	Byte CRC, do barramento 2
clk	Entrada	1	Sinal de relógio
cmdByte_1	Entrada	8	Byte que indica o comando, do barramento 1
cmdByte_2	Entrada	8	Byte que indica o comando, do barramento 2
dataByte1_1	Entrada	8	Primeiro byte de atributos do comando, do barramento 1
dataByte1_2	Entrada	8	Primeiro byte de atributos do comando, do barramento 2
dataByte2_1	Entrada	8	Segundo byte de atributos do comando, do barramento 1
dataByte2_2	Entrada	8	Segundo byte de atributos do comando, do barramento 2
dataByte3_1	Entrada	8	Terceiro byte de atributos do comando, do barramento 1
dataByte3_2	Entrada	8	Terceiro byte de atributos do comando, do barramento 2
dataByte4_1	Entrada	8	Quarto byte de atributos do comando, do barramento 1
dataByte4_2	Entrada	8	Quarto byte de atributos do comando, do barramento 2
respByte	Saída	8	Byte de resposta recebido do cartão
respTout	Saída	1	Ocorrencia de Time Out
rst	Entrada	1	Sinal de reset
rxDataIn	Entrada	8	Bytes lidos do cartão
rxDataRdy	Entrada	1	Sinal que indica que foi lido um byte do barramento SPI
rxDataRdyClr	Saída	1	Indicar que o barramento está disponível para fazer leituras
sendCmdRdy	Saída	1	Sinal que indica que o comando foi enviado
sendCmdReq1	Entrada	1	Sinal que indica que um comando está no barramento 1 para ser enviado
sendCmdReq2	Entrada	1	Sinal que indica que um comando está no barramento 2 para ser enviado
txDataEmpty	Entrada	1	O barramento SPI está disponível para enviar dados
txDataFull	Entrada	1	O barramento SPI está ocupado
txDataOut	Saída	8	Barramento de dados para enviar para o cartão
txDataWen	Saída	1	Byte que indica que se pretende enviar um byte pelo barramento de dados

Tabela 4.6: sendCmd

hardware que acedem à interface SPI foi necessário criar este módulo. A sua função é de multiplexar os barramentos de dados de entrada e saída do módulo *readWriteSPIWireData* de forma a que não ocorram choques de dados e o circuito seja sintetizável.

Na tabela 4.7 pode-se ver a interface implementada.

Pino	E/S	Tamanho	Descrição
clk	Entrada	1	Sinal de relógio
rst	Entrada	1	Sinal de controlo que reinicializa o módulo
tx1DataIn	Entrada	8	Byte de entrada proveniente do barramento 1
tx2DataIn	Entrada	8	Byte de entrada proveniente do barramento 2
tx3DataIn	Entrada	8	Byte de entrada proveniente do barramento 3
tx4DataIn	Entrada	8	Byte de entrada proveniente do barramento 4
tx1DataWEn	Entrada	1	Sinal que activa o barramento de entrada 1
tx2DataWEn	Entrada	1	Sinal que activa o barramento de entrada 2
tx3DataWEn	Entrada	1	Sinal que activa o barramento de entrada 3
tx4DataWEn	Entrada	1	Sinal que activa o barramento de entrada 4
txDataOut	Saída	8	Byte de saída
txDataFull	Saída	1	Sinal que indica que o barramento de saída está a ser utilizado
txDataFullClr	Entrada	1	Sinal de entrada que indica fim de transmissão de byte
rx1DataRdyClr	Entrada	1	Sinal que indica o fim de leitura de dados do barramento 1
rx2DataRdyClr	Entrada	1	Sinal que indica o fim de leitura de dados do barramento 2
rx3DataRdyClr	Entrada	1	Sinal que indica o fim de leitura de dados do barramento 3
rx4DataRdyClr	Entrada	1	Sinal que indica o fim de leitura de dados do barramento 4
rxDataIn	Entrada	8	Byte lido do barramento SPI
rxDataOut	Saída	8	Byte lido do barramento SPI
rxDataRdy	Saída	1	Indica se o módulo está a efectuar leituras
rxDataRdySet	Entrada	1	Sinal que coloca o módulo em modo de leitura

Tabela 4.7: spiTxRxData

4.2.8 Módulo *readWriteSPIWireData*

Este módulo é responsável pela execução do protocolo de comunicação SPI no sistema.Todos os dados enviados e recebidos para o cartão são formatados e enviados a partir deste módulo.

O módulo recebe bytes do sistema e multiplexa-os temporalmente no fio *spiDataOut* e vice-versa para os dados que recebe por SPI do fio *spiDataIn*. Outra função importante que é executada neste módulo é a geração de sinal de relógio *spiClkOut* usado para realizar o sincronismo com o cartão.

Descrição da interface utilizada neste bloco: 4.8

4.2.9 *Buffers* de leitura e escrita

De forma a que a escrita de dados possa ser executada à taxa máxima de transferência criaram-se dois *buffers*. Estes *buffers* são uma memória intermédia entre a leitura e escrita de dados para o cartão. Um dos *buffers* permite a aplicação da camada superior guardar os dados que pretende enviar para o cartão, o outro, o de leitura, guarda o sector lido do cartão. Em cada leitura/escrita tem que ser recebido/enviado um sector, por isso, implementaram-se *buffers* de 512 bytes.

As operações de leitura e escrita só são efectuadas quando é dada a ordem pela camada superior. Se forem lidos ou escritos 512 bytes o endereço volta a posição inicial, podendo fazer uma

Pino	E/S	Tamanho	Descrição
clk	Entrada	1	Sinal de relógio
clkDelay	Entrada	8	Valor usado para calcular o período do relógio SPI
rst	Entrada	1	Sinal de controlo que reinicializa o módulo
rxDataOut	Saída	8	Byte recebido pelo barramento SPI
rxDataRdySet	Saída	1	Sinal que indica que um byte foi lido
spiClkOut	Saída	1	Sinal de clock SPI
spiDataIn	Entrada	1	Sinal por onde se faz a recepção de dados do SPI
spiDataOut	Saída	1	Sinal por onde se faz a emissão de dados do SPI
txDataEmpty	Saída	1	Indica que o módulo está disponível para enviar um byte para o SPI
txDataFull	Entrada	1	Sinal que coloca o módulo em modo de envio de byte ou bytes para SPI
txDataFullClr	Saída	1	Sinal que indica ao módulo o fim de envio de bytes para o SPI
txDataIn	Entrada	8	Byte que se pretende enviar pelo barramento SPI

Tabela 4.8: readWriteSPIData

nova leitura ou rescrever os dados a partir dessa posição antes de se enviar os dados para o cartão ou pedir para ler outro sector.

Cada *buffer* é do tipo FIFO, tanto para receber bytes como para enviar, e são implementados em duas BRAM, de 512 bytes, da FPGA optimizando-se assim o espaço ocupado na FPGA. O endereçamento dos dados é sequencial(FIFO) e é efectuado no próprio módulo de hardware.

Em vez de se usar uma FIFO existe a alternativa de endereçar a posição do buffer em que se pretende escrever. Está alternativa não seria difícil implementar e até chegou mesmo a ser feita, mas posteriormente foi alterada para o tipo FIFO.

À partida, o sistema global será usado para a escrita de ficheiros, que por norma são muito maiores que um sector (512 bytes). Portanto, numa escrita para a memória, os bytes deverão ser enviados de forma sequencial, não havendo necessidade de endereçar bytes individualmente. A pensar neste aspecto e sabendo que a camada superior será implementada recorrendo a um PicoBlaze que tem recursos bastante limitados, implementou-se um *buffer* do tipo FIFO de forma a optimizar ao máximo tanto o espaço ocupado como a velocidade de leituras e escritas para o cartão.

Sendo assim, os dados são enviados de forma sequencial e o endereçamento é feito em *hardware*, não sendo necessário a cada byte indicar o endereço, libertando o PicoBlaze dessa tarefa e ao mesmo tempo optimiza-se o sistema para a escrita e leitura de ficheiros.

Estes buffers foram desenvolvidos com duplo sincronismo de relógio, quer isto dizer que se pode escrever e ao mesmo tempo se fazer uma leitura. Os dois relógios não têm que ser iguais. Quando pretendemos escrever um sector para o cartão, usa-se o relógio do *bus* para escrever no *buffer* e o relógio do sistema para ler o *buffer*. Passando-se ao contrário no caso de leitura de um sector do cartão.

Descrição da interface utilizada neste bloco: 4.9

Pino	E/S	Tamanho	Descrição
busClk	Entrada	1	Sinal de relógio usado para escrever os dados no buffer
spiSysClk	Entrada	1	Sinal de relógio usado para ler os dados no buffer
rstSyncToBusClk	Entrada	1	reset sincronizado com o relógio do bus, circuito do sistema de ficheiros
rstSyncToSpiClk	Entrada	1	reset sincronizado com o relógio do SPI
fifoREn	Entrada	1	Sinal que indica a operação que se pretende efectuar no buffer, leitura ou escrita
fifoEmpty	Saída	1	Sinal que indica que FIFO está vazia
busAddress	Entrada	3	Este buffer é accionado pelo circuito exterior através do seu endereço
busWriteEn	Entrada	1	Sinal do circuito exterior que indica a operação que se pretende efectuar no buffer, leitura ou escrita
busStrobe_i	Entrada	1	Sinal de sincronismo, que indica, que o barramento de dados tem dados válidos
busFifoSelect	Entrada	1	Este sinal fica activo quando o circuito exterior endereça este buffer
busDataIn	Entrada	8	Barramentos de dados de entrada
busDataOut	Saída	8	Barramentos de dados de saída
fifoDataOut	Saída	8	Barramento de dados que são enviados para o cartão

Tabela 4.9: txFifo

4.3 Subsistema - Módulo *ifMmcSd*

O módulo *ifMmcSd* reúne os módulos apresentados anteriormente num só módulo. Resultando num subsistema com as seguintes características:

- Suporte de cartões MMC e SD
- Utiliza um protocolo SPI
- FIFOs de 512 bytes para leitura e escrita
- Interface Wishbone de 8 bits
- Relógios de separados para o Bus e SPI
- Transferência de dados a 25MBbits/s, taxa máxima dos cartões

4.3.1 Descrição funcional

Para realizar operações de inicialização, escrita e leitura num cartão tem que se realizar alguns procedimentos.

4.3.1.1 Inicialização

Antes de se poder realizar operações de leitura ou escrita no cartão tem que realizar um procedimento de inicialização. Este procedimento é executado de forma automática, pelo *hardware* com a inserção do cartão na ranhura. Não tendo a aplicação de se "preocupar" com este procedimento.

4.3.1.2 Escrita de um bloco

Para que se escreva um bloco é necessários executar um procedimento. Começa-se o procedimento por escrever 512 bytes no buffer e depois defini-se um conjunto de registos no circuito de interface que são acedidos por endereçamento.

O procedimento de escrita de um bloco está descrito em 4.5

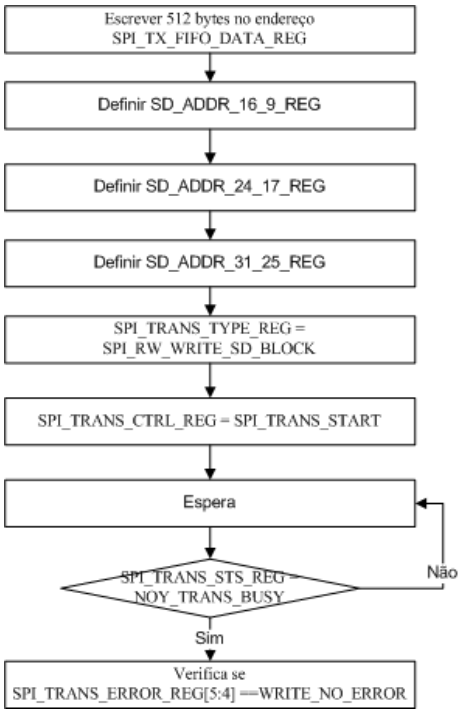

Figura 4.5: Procedimento de escrita de um bloco

4.3.1.3 Leitura de um bloco

Para se ler um bloco do cartão MMC ou SD é necessários executar um procedimento. Esse procedimento passa por definir um conjunto de registos no circuito de interface e os registos são acedidos pelo endereço. No final executa-se a leitura dos 512bytes do buffer.

Pode-se verificar o procedimento em 4.6.

4.3.2 Registos

O sistema disponibiliza um conjunto de registos que podem ser acedidos a partir do exterior. Com esses registos a aplicação exterior pode controlar o funcionamento do subsistema (*ifMmcSd*.

Os registos são acedidos por endereçamento e o valor é lido/escrito pelos barramentos de I/O em conjunto com os sinais de controlo(*we,strobe_i,etc*). Todos os registos 4.10 têm 8 bits que podem ser acedidos a partir do exterior quer para leitura ou escrita.

Cada registo pode ter um conjunto de valores especificados em 4.11, 4.12, 4.13 e 4.14.

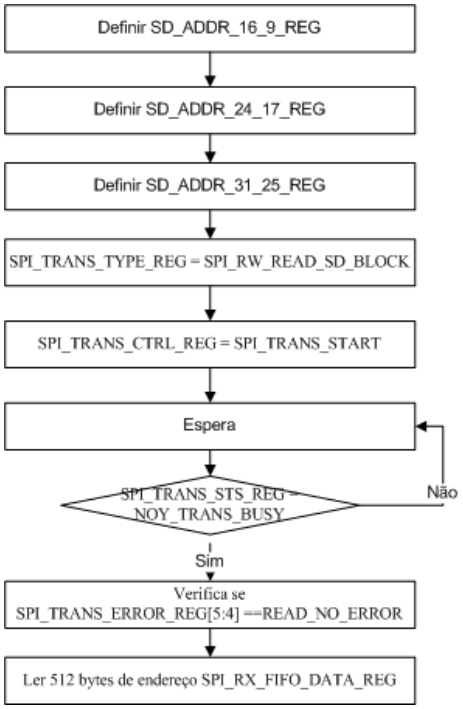

Figura 4.6: Procedimento de leitura de um bloco

Endereço	Valor
0x1	SPI_MASTER_CONTROL_REG
0x2	TRANS_TYPE_REG
0x3	TRANS_CTRL_REG
0x4	TRANS_STS_REG
0x5	TRANS_ERROR_REG
0x7	SD_ADDR_16_9_REG
0x8	SD_ADDR_24_17_REG
0x9	SD_ADDR_31_25_REG
0x10	RX_FIFO_DATA_REG
0x14	RX_FIFO_CONTROL_REG
0x20	TX_FIFO_DATA_REG
0x24	TX_FIFO_CONTROL_REG

Tabela 4.10: Registos da I/F SPI

Posição do bit	Nome	Descrição	Valor pre-definido	R/W
0	RST	= 1 reinicia todos os registos e lógica do sistema	0	W

Tabela 4.11: SPI_MASTER_CONTROL_REG

Posição do bit	Nome	Descrição	Valor pre-definido	R/W
[1 : 0]	TRANS_TYPE	Define o tipo de operação; 1=INIT_SD 2=RW_READ_SD_BLOCK 3=RW_WRITE_SD_BLOCK	0	R/W
0	TRANS_START	1=Inicia a transmissão	0	W
0	TRANS_BUSY	1=Comunicação ocupada	0	R

Tabela 4.12: TRANS_REG

Posição do bit	Nome	Descrição	Valor pre-definido	R/W
[5 : 4]	SD_WRITE_ERROR	0=WRITE_NO_ERROR 1=WRITE_CMD_ERROR 2=WRITE_DATA_ERROR 3=WRITE_BUSY_ERROR	0	R
[3 : 2]	SD_READ_ERROR	0=READ_NO_ERROR 1=READ_CMD_ERROR 2=READ_TOKEN_ERROR	0	R
[1 : 0]	SD_INIT_ERROR	0=INIT_NO_ERROR 1=INIT_CMD0_ERROR 2=INIT_CMD1_ERROR	0	R

Tabela 4.13: TRANS_ERROR_REG

Posição do bit	Nome	Descrição	Valor pre-definido	R/W
[7 : 0]	SD_ADDR_16_9_ADDR[7:0]	LBA1 do bloco que se pretende aceder	00	R/W
[7 : 0]	SD_ADDR_24_17_ADDR[7:0]	LBA2 do bloco que se pretende aceder	00	R/W
[7 : 0]	SD_ADDR_31_25_ADDR[7:0]	LBA3 do bloco que se pretende aceder	00	R/W

Tabela 4.14: LBA_REG

Posição do bit	Nome	Descrição	Valor pre-definido	R/W
[7 : 0]	RX_FIFO_DATA	Leitura de dados do buffer RX	0	R
[7 : 0]	TX_FIFO_DATA	Escrita de dados do buffer TX	0	W
[7 : 0]	RX_FIFO_CONTROL_REG	Apaga todos os valores do buffer	0	w
[7 : 0]	TX_FIFO_CONTROL_REG	Apaga todos os valores do buffer	0	w

Tabela 4.15: FIFO_REG

Capítulo 5

Implementação do sistema de ficheiros FAT

A camada que implementa o sistema de ficheiros tem como função principal gerir o modo como a informação é organizada na unidade de armazenamento, sendo todo o protocolo de transferência com a unidade de armazenamento completamente transparente.

O sistema de ficheiros desenvolvido é um sistema bastante compacto, baseado no FAT 16. Para o implementar recorreu-se ao *soft-processor* PicoBlaze e a alguns módulos de hardware adicionais.

O PicoBlaze é o núcleo desta camada que processa toda a informação. Os módulos adicionais foram desenvolvidos para libertar o PicoBlaze de algumas tarefas, que são executadas em paralelo, optimizando-se o desempenho do Sistema global, diminuindo o tamanho do programa, antecipando assim a possibilidade do PicoBlaze ficar sem espaço em memória de programa.

A comunicação com a camada superior é feita através de comunicação em série e tem um protocolo especifico. Para comunicar com a camada inferior é usado o protocolo Wishbone, que pode ser consultado em [13].

Durante o desenvolvimento desta camada foi necessário ter uma especial atenção em relação as limitações do PicoBlaze, nomeadamente, a ocupação da memória.

5.1 Arquitectura física

Em 5.1 apresenta-se o diagrama de blocos, simplificado, da arquitectura física implementada.

5.2 Hardware

Nesta secção faz-se uma explicação do funcionamento dos módulos implementados.

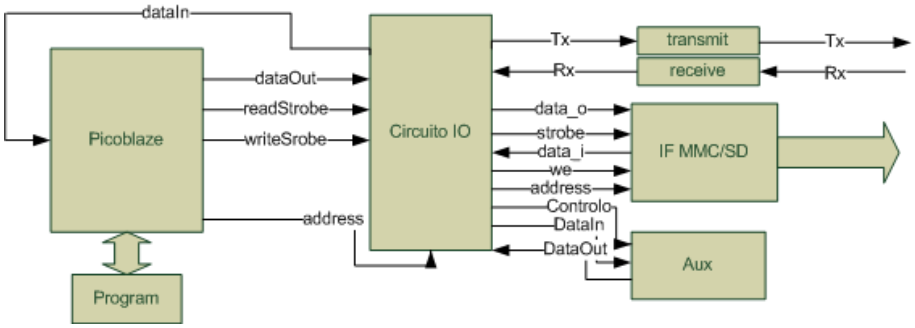

Figura 5.1: Arquitectura física do sistema FAT

5.2.1 Circuito multiplexador das portas I/O - *picoMuxIO*

O PicoBlaze para comunicar com o exterior tem um conjunto de barramentos e sinais de controlo:

- Um barramento de entrada de 8 bits
- Um barramento de saída de 8 bits
- Um barramento de endereçamento
- Sinais de controlo

Com apenas este conjunto de IOs o PicoBlaze é um microcontrolador bastante limitado e não se adequava ao nosso circuito, pois este subsistema necessita de comunicar com várias interfaces:

- Comunicação em série, dois barramentos de entrada e saída de dados
- Circuito de interface com o cartão, dois barramentos de entrada e saída de dados
- Barramentos de controlo, associados a comunicação em série e I/F do cartão

É necessário que o microcontrolador comunique com vários dispositivos através dos respectivos barramentos e gere os sinais de controlo necessários a essas comunicações. Para fazer a comunicação utilizou-se o barramento de endereços do PicoBlaze para codificar uma determinada operação e criou-se um circuito em *hardware* (picoMuxIO). O picoMuxIO descodifica essa operação e faz a intercomunicação com o resto do sistema, de acordo com a operação pretendida.

5.2.2 Circuito de comunicação com a aplicação

Para comunicar com a aplicação desenvolveu-se um subsistema que implementa o protocolo de comunicação standard UART. Seguiu-se por este tipo de comunicação por permitir:

- Comunicar com um PC
- Comunicar facilmente com outros sistemas embutidos

- Não diminuir a taxa de transferência de dados

Este sistema embutido para além de poder comunicar com um vulgar PC, permite que facilmente se utiliza a aplicação *Hyper Terminal* para realizar operações de teste e debug.

Este forma de comunicação permite comunicar com outros dispositivos que também tenham uma interface UART. É uma forma de comunicação bastante simples de se implementar e que permite taxas de transferência relativamente elevadas, ocupa uma área bastante reduzida e o protocolo de comunicação pode ser facilmente desenvolvido.

Quando efectuamos uma comunicação UART com o PC as taxas de comunicação estão de acordo com a especificação RS-232 (9800 bytes/s...128 kbytes/s...), mas já entre circuitos na própria FPGA pode ir até 10MBytes/s, situando-se esta taxa acima da capacidade de envio/recepção prevista para o PicoBlaze.

O circuito físico do sistema de comunicação UART (transmissor e receptor) é constituindo pelos seguintes módulos de hardware:

- UART_TX
- UART_RX
- Gerador de clock

Características de comunicação:

- 1 start bit
- 8 data bits
- Sem paridade
- 1 stop bit

Este sistema de comunicação ocupa uma área bastante reduzida da FPGA, ver 5.1.

Módulo	Área/Slices
UART_TX	18
UART_RX	22

Tabela 5.1: UART Área ocupada

5.2.2.1 Módulo *UART_TX*

Este módulo tem a função de enviar dados por comunicação em série que recebe por um barramento de 8 bits em paralelo. Internamente é constituído por dois módulos:

- embededUART_TX

- buffer_TX

O módulo *embededUART* realiza a comunicação, ou seja, recebe os dados do buffer e transmite-os no protocolo UART. O buffer_TX é um buffer 16 bytes que armazena os bytes recebidos e que os vai enviado para o embededUART_TX à medida que este vai pedindo. Na tabela 5.2 poderá ser vista a interface do módulo implementado.

Pino	E/S	Tamanho	Descrição
data_in	Entrada	8	Entrada de dados de 8 bits em paralelo
write_buffer	Entrada	1	Quando este sinal estiver a "1"indica que na próxima transição positiva do sinal de relógio o data_in deve ser amostrado
reset_buffer	Entrada	1	Sinal que elimina todos os dados contidos no buffer
en_16_x_baud	Entrada	1	Sinal de sincronismo com a taxa de transmissão desejada
serial_out	Saída	1	Dados de saída da comunicação em série
buffer_full	Saída	1	Sinal indicador que o buffer está cheio todos os dados de entrada são ignorados neste estado
buffer_half_full	Saída	1	Sinal indicador que o buffer está meio cheio
clk	Entrada	1	Sinal de relógio

Tabela 5.2: UART_TX

5.2.2.2 Módulo *UART_RX*

Este módulo tem um barramento de dados de saída de 8 bits, sinais de controlo e um fio de entrada por onde são recebidos os dados em série. Internamente é constituído por dois módulos:

- embededUART_RX
- buffer_RX

O módulo embededUART faz comunicação, ou seja, recebe os dados da aplicação em série pelo fio RX e vai enchendo o buffer. O buffer_RX é um buffer 16 bytes que armazena os bytes recebidos da aplicação e os vai enviado para o PicoBlaze à medida que este vai pedindo. Na tabela 5.3 poderá ser vista a interface do módulo implementado para que melhor se perceba a função do módulo.

5.2.3 Gerador de Clock

Os módulos de transmissão UART precisam de estar sincronizados com a taxa de transmissão. Para fazer esse sincronismo teve que se criar um módulo que implemente um sinal de *enable* à taxa de transmissão pretendida. O sincronismo é feito a partir do momento que se recebe o *start bit*.

Pino	E/S	Tamanho	Descrição
serial_in	Entrada	1	Dados lidos em série
data_out	Saída	8	Barramento de saída de 8 bits em paralelo
read_buffer	Entrada	1	Sinal que indica que os dados foram lidos do data_out o próximo byte deverá estar disponível no próximo ciclo de relógio
reset_buffer	Entrada	1	Sinal que elimina todos os dados contidos no buffer
en_16_x_baud	Entrada	1	Sinal de sincronismo com a taxa de transmissão desejada
buffer_data_present	Saída	1	Sinal de indica que o buffer tem pelo menos um byte
buffer_full	Saída	1	Sinal que indica que o buffer esta cheio, todos os dados recebidos por serial_in serão ignorados
buffer_half_full	Saída	1	Sinal que indica que o buffer está meio cheio
clk	Entrada	1	Sinal de relógio

Tabela 5.3: UART_RX

$$clock\_division = \frac{clk\_rate}{BAUD * 16}$$

O sinal de referência gerado tem a duração de 1 ciclo de *clock* e uma taxa de 16 vezes a taxa da comunicação por série desejada.

5.2.4 Circuito Auxiliar

Devido ao PicoBlaze ter atingido o limite de memória de programa, optou-se pela criação de um circuito exterior auxiliar para executar algumas rotinas. Desta forma passaram-se para o exterior rotinas de elevada complexidade que aumentavam em muito o tamanho do programa (ocupando memória) e que iriam atrasar o processamento dos dados. Para além disso, com esta simplificação, faz-se uso de características em que o *hardware* é superior ao *software*, resultando um sistema mais optimizado.

Para o PicoBlaze aceder a este circuito usou-se o mesmo método que anteriormente. Usou-se o picoMuxIO 5.2.1 para endereçar o circuito auxiliar e através do barramentos de dados faz-se a troca de comandos e informações.

O circuito auxiliar implementado tem a função de auxiliar a rotina de pesquisa de entradas na *root* da partição. É necessário criar um método que pesquise a *root* e que verifique se um determinado ficheiro já existe.

Esta rotina é implementada da seguinte forma:

O programa começa por introduzir num *shift register*, de 11 andares de 8 bits, o nome do ficheiro. Escolheu-se um *shif register* porque facilmente pode gravar o nome do ficheiro e comparar com os dados que vai recebendo pelo barramento de entrada, em série.

De seguida o PicoBlaze envia para o circuito auxiliar os nomes dos ficheiros contidos na *root*, o circuito recebe no barramento de entrada de 8 bits cada um dos bytes em série correspondente ao nome do ficheiro. Esses bytes são comparados à medida que são recebidos e o *shift register* vai "rodando" a medida que compara, com o byte recebido. No fim de cada 11 bytes o *shift register* volta ao primeiro byte e o PicoBlaze pode enviar o primeiro byte do nome seguinte.

O resultado de igualdade de cada uma das 11 comparações em série é guardado num registo de 1 bit. Este registo é uma saída do circuito e pode ser acedido pelo PicoBlaze, para verificar se existe igualdade entre o nome recebido e o guardado no *shift register*.

5.3 Software

O programa desenvolvido para o PicoBlaze segue o fluxograma indicado em 5.2.

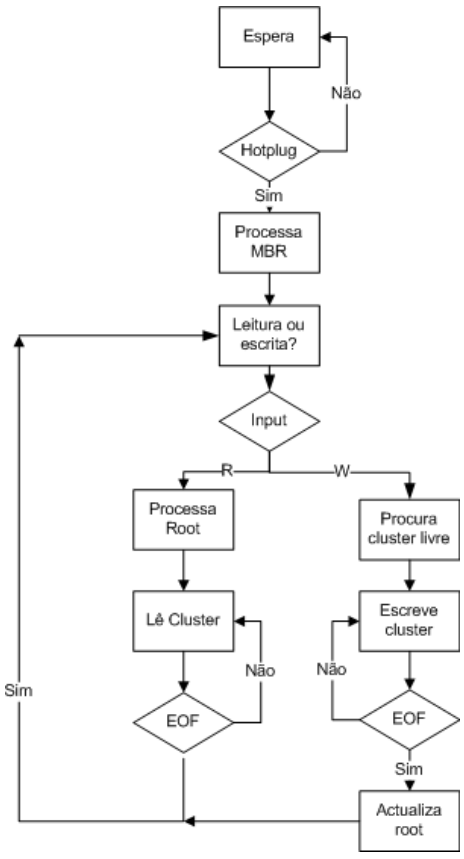

Figura 5.2: Fluxograma do FAT

5.3.1 Espera

Neste estado o PicoBlaze está em *standby* à espera do sinal externo de detecção de cartão. O sinal de detecção do cartão é activado quando um cartão é inserido na ranhura do *socket*. A inserção do cartão desbloqueia o PicoBlaze iniciando o programa que processa o sistema de ficheiros.

5.3.2 Processa MBR

Neste estado o programa chama a rotina *process_master_boot_record*, que começa por ler o sector 0 do cartão. De seguida verifica se o sector lido é a MBR ou o Volume ID do cartão. Se estivermos na situação de MBR lê o apontador para o Volume ID da primeira partição existente no cartão.

Depois de ler o Volume ID, o programa guarda em registos internos todas as informações imprescindíveis 5.4 sobre o volume.

Nome	offset
Sector_per_cluster	0x0d
Number_reserved_sectors	0x0e
Number_of _FAT	0x10
Sectors_per_FAT	0x16

Tabela 5.4: Dados processados, Volume ID

Com os dados lidos, pode-se resolver as seguintes fórmulas matemáticas:

$$fat\_begin\_lba = Partition\_LBA\_Begin + Number\_of\_Reserved\_Sectors$$

$$cluster\_begin\_lba = Partition\_LBA\_Begin + Number\_of\_Reserved\_Sectors$$

$$+(Number\_of\_FATs * Sectors\_Per\_FAT)$$

$$\Leftrightarrow cluster\_begin\_lba = fat\_begin\_lba + (Number\_of\_FATs * Sectors\_Per\_FAT)$$

fat_begin_lba Este valor indica o endereço da tabela FAT no cartão MMC ou SD. A tabela FAT contem os apontadores dos *clusters*, os apontadores servem para fazer o seguimento dos ficheiros na estrutura do Sistema de Ficheiros. Também indica quais os *clusters* livres na partição, de forma a se poder gravar um novo ficheiro.

cluster_begin_lba Valor que indica onde começa o espaço de dados; este espaço está dividido em *clusters*.

No fluxograma 5.3 pode-se ver as etapas que são executadas pela função que processa a MBR e Volume ID.

Na figura 5.4 mostrasse um exemplo de um sector de 512 bytes correspondente ao Volume ID dum cartão. Nesse mesma figura indica-se os valores necessários para o processamento do volume ID e o seu respectivo *offset*.

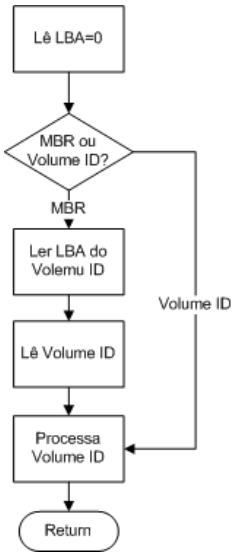

Figura 5.3: Fluxograma processamento MBR e Volume ID

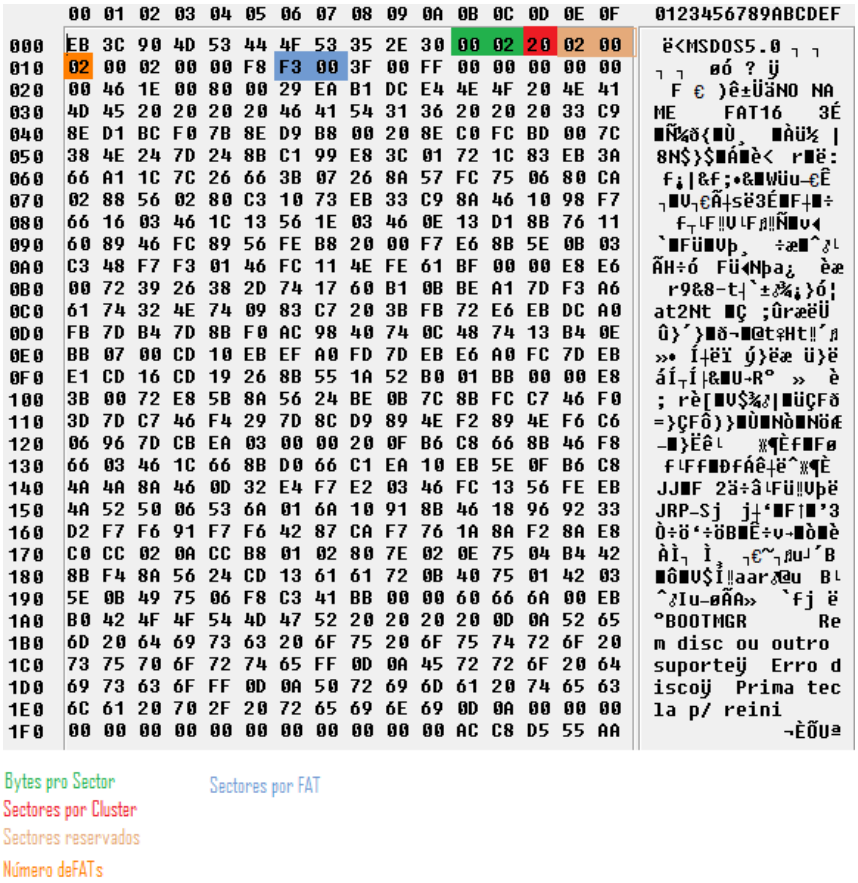

Figura 5.4: FAT 16 Volume ID dados necessários

5.3.3 Leitura ou escrita?

Este estado divide-se em dois sub-estados. O primeiro estado é um ciclo, em que o programa espera que a aplicação indique se pretende fazer uma leitura ou escrita no cartão. Depois da aplicação ter indicado a operação que pretende fazer entra no segundo estado. Neste, o programa pergunta a aplicação o nome do ficheiro que pretende gravar ou ler. À medida que o nome é recebido por comunicação UART, vai sendo gravado na memória interna do PicoBlaze.

5.3.4 Processa *Root*

Neste estado o PicoBlaze lê a *root* 5.5 da partição e faz uma pesquisa pelo nome do ficheiro indicado anteriormente (presente na memória) pela aplicação; se não encontrar envia mensagem de erro para aplicação; se o nome do ficheiro é encontrado, lê o valor do apontador para o primeiro *cluster* do ficheiro.

Na figura 5.5 indicasse um exemplo de uma entrada de um ficheiro na *root* de uma partição.

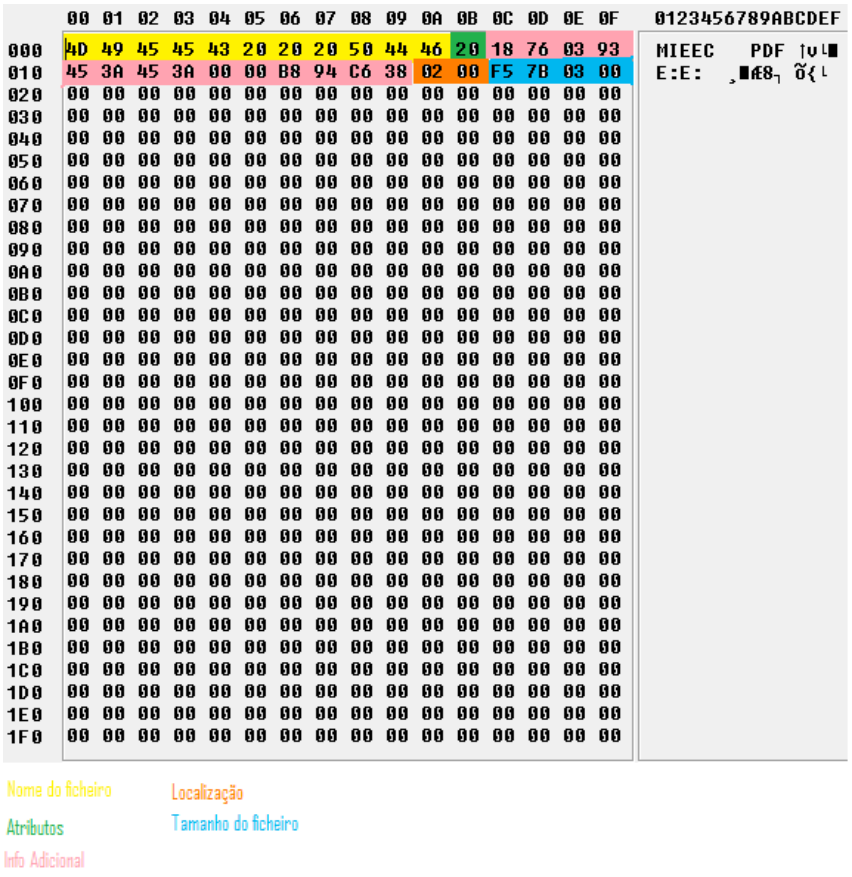

Figura 5.5: Entrada de um ficheiro na *root*

5.3.5 Lê *cluster*

Um *cluster* é constituído por vários sectores, se considerarmos por exemplo, que para um determinado sistema de ficheiros um *cluster* são 32 sectores. Então, é necessário criar-se um ciclo que leia 32 sectores sequencialmente, que corresponde a um *cluster*. No fim da leitura de cada *cluster* é necessário verificar na tabela FAT se no ficheiro existem mais *clusters*. Caso existam, lê na FAT o apontador para o próximo *cluster* e recomeça o ciclo de leitura, caso seja o último *cluster* finaliza a leitura do ficheiro.

Na figura 5.6 pode-se ver os processos que a função de leitura executa para ler um *cluster*.

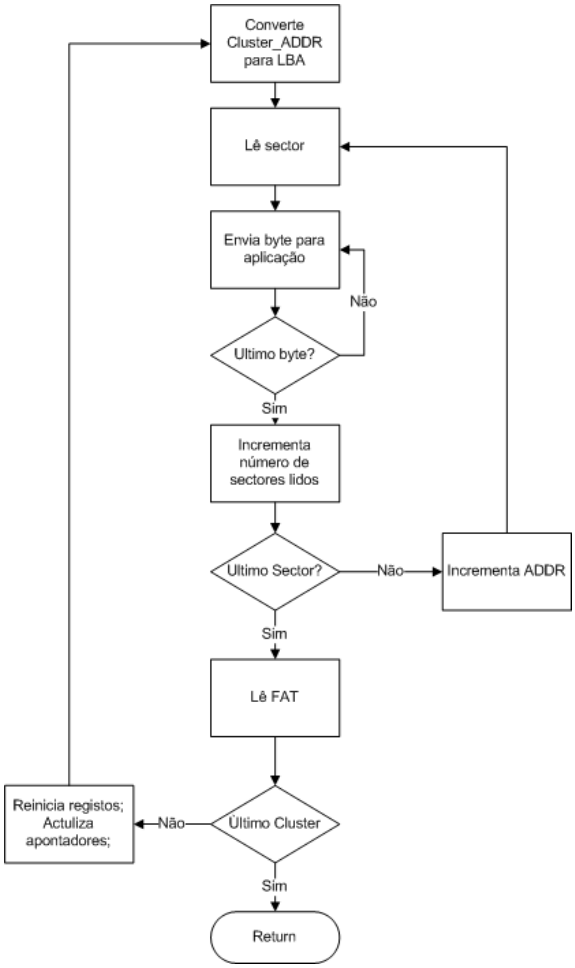

Figura 5.6: Leitura de ficheiro

5.3.6 Procura *cluster* livre

Nesta rotina o programa faz uma pesquisa a tabela FAT (ver 5.7) da partição e procura pelo primeiro *cluster* livre. Quando um *cluster* livre é encontrado retorna o valor do apontador para esse *cluster*.

A rotina foi implementada recorrendo a ciclo que incrementa o apontador da tabela FAT até encontrar o valor 0x0000. Como a tabela FAT é constituída por apontadores de 16 bits têm que ser lidos 2 bytes de cada vez.

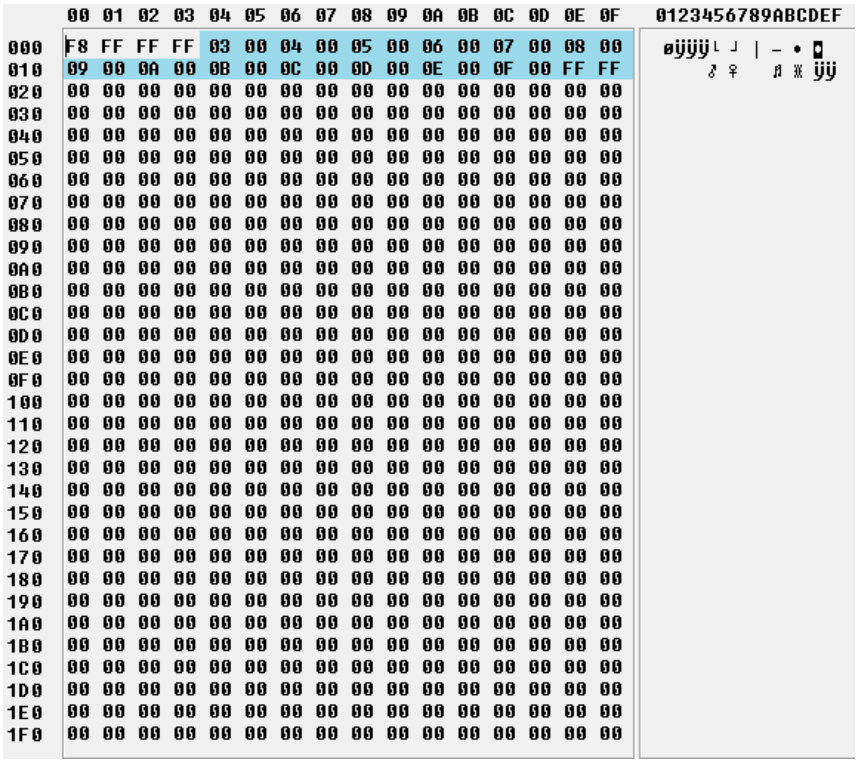

Figura 5.7: FAT 16 estrutura da tabela FAT

5.3.7 Escreve *cluster*

Esta rotina escreve a informação (ficheiro) recebida da aplicação, em formato “raw”, e gravar no cartão no formato FAT 16.

A rotina (ver 5.8) começa por converter o apontador do *cluster* indicado num endereço (LBA), e então, entra num ciclo de leitura de dados da aplicação e escrita no cartão. Esse ciclo consiste em receber os bytes da aplicação e em escreve-los no *buffer* do cartão, a cada sector (512 bytes) que recebe dá ordem para escreve no cartão e incrementa o endereço para a posição seguinte até preencher um *cluster*.

No fim da escrita de cada *cluster* inicia-se a pesquisa por um *cluster* livre na FAT 5.7. Quando um é encontrado actualiza-se a tabela FAT colocando o apontador para o *cluster* livre na posição do *cluster* que foi preenchido. Neste ponto o ciclo recomeça voltando-se a gravar um novo *cluster*.

A rotina finaliza quando recebe o byte EOF da aplicação. Nesse momento pára a recepção de dados da aplicação, preenche o resto do sector com "0s", grava no cartão e retorna.

Figura 5.8: Escreve cluster

5.3.8 Actualiza *Root*

Depois de finalizar a escrita de dados o programa tem que actualizar a *root* 5.5. Para isso lê o *cluster* da *root* e procura por uma entrada livre indicada pelo byte 0xE5 ou pelo fim de entradas na *root* (byte 0x00).

De seguida introduz a entrada do ficheiro acabado de gravar no sistema de ficheiros. A entrada só contem as informações relevantes e as outras são colocadas com valores pré-definidos.

As informações escritas são: O nome do ficheiro que foi anteriormente enviado pela aplicação e gravado na memória interna do PicoBlaze; o valor para o primeiro *cluster* do ficheiro que também está gravado na memoria interna; e o valor do tamanho do ficheiro. Relativamente a todas as outras informações (data e hora da criação, data e hora da ultima modificação) não faz sentido escrever o valor correcto porque neste sistema não haveria forma de obter essas informações ou aumentava bastante a sua complexidade. Por isso optou-se por colocar valores pré-definidos.

Capítulo 6

Sistema Final

O sistema final foi implementado e desenvolvido usando o Spartan 3 Starter Kit, desenvolveu-se um circuito exterior de interface com o cartão e usou-se um PC, que fez interface com a Spartan 3 através da porta de série (RS-232).

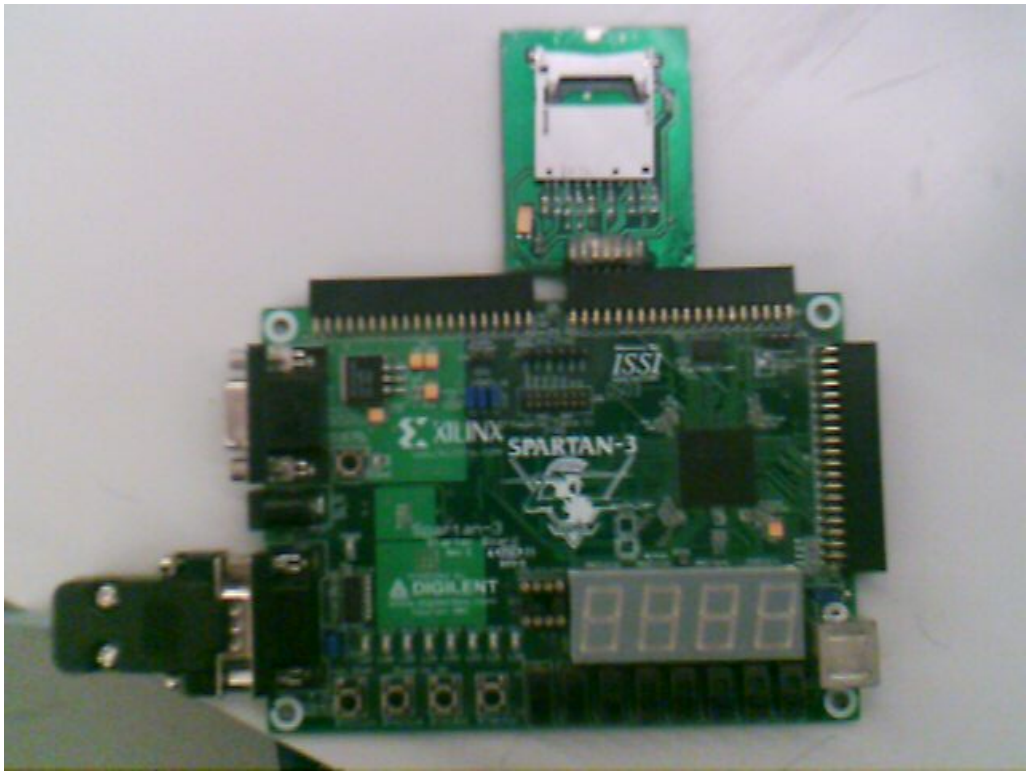

Figura 6.1: kit desenvolvimento

6.1 Funcionalidades implementadas

O sistema oferece as seguintes funcionalidades:

- Comunicação com a aplicação por UART
  - Barramento de comunicação com apenas dois fios: TX e RX
  - 1 *start bit*, 8 *data bits*, 1 *stop bit*
  - *Buffer* de 16 bytes na escrita
  - *Buffer* de 16 bytes na leitura
- Protocolo de comunicação simples
- Suporta cartões de memória MMC e SD
- Leitura e escrita de dados em FAT16
  - Criação de ficheiros na raíz da partição da primeira partição
  - Dados escritos e lidos sob a forma de um ficheiro
  - Suporte para *clusters* de vários tamanhos, de acordo com a formatação do S.O.
  - Indicação do tamanho do ficheiro gravado
- Portabilidade de leitura e escrita na maior parte dos Sistemas Operativos
- Área ocupada de reduzidas dimensões
- Taxa de transmissão de aproximadamente 4MB/s
- Suporta Hot Plug do cartão
  - Detecção do cartão inserido
  - Inicialização automática do cartão

6.2 Descrição funcional

A comunicação com a aplicação é feita por UART usando-se dois fios de comunicação "RX" e "TX". A comunicação é feita em modo assíncrono em que os dois sistemas terão que ter os relógios sincronizados, com um erro máximo de 0.5%. Se for utilizado para comunicação o *standard* RS-232 ter-se-á que usar as taxas convencionais neste tipo de transmissão. Já se utilizarmos UART entre *hardware* embutido as taxas de transferência de dados podem ir até 10MB/s.

A taxa de transmissão do sistema pode ser definida no módulo gerador de *clock* 5.2.3 e tem um valor fixo definido no momento em que se efectua a síntese do sistema.

O protocolo de comunicação com a aplicação é bastante simples. Foi desenvolvido de forma a que o processo que a aplicação está a executar não gaste em excesso os recursos do microprocessador. Deste forma todo o processo de armazenar dados na unidade de armazenamento *flash* é bastante rápido, eficiente e optimizado.

Indica-se em 6.2 o processo de leitura de dados *raw* armazenados no cartão,

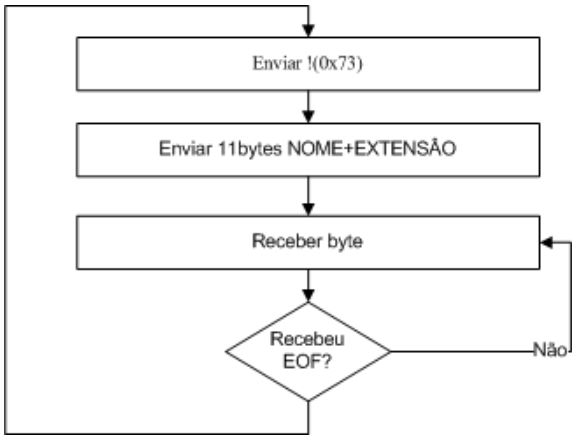

Figura 6.2: Processo de leitura de dados

e em 6.3 indica-se o processo de escrita de dados na unidade de armazenamento *flash*.

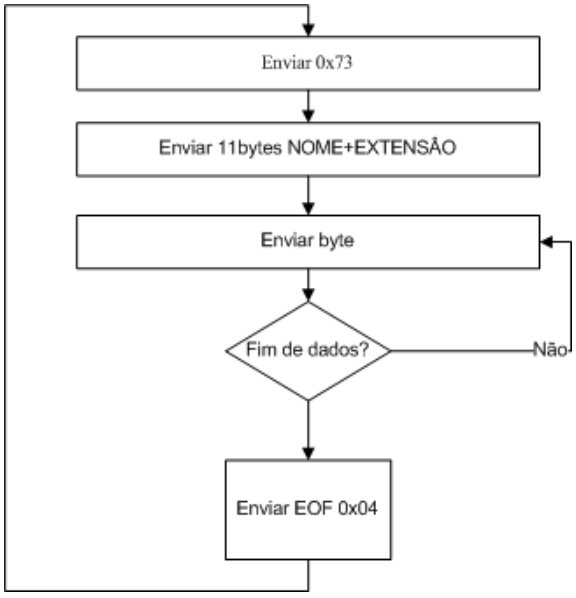

Figura 6.3: Processo para a escrita de dados

6.3 Sumário da utilização de recursos

Selected Device: 3s200ft256-5

Numberof Slices: 674 out of 1920 35%

Number of Slice Flip Flops: 764 out of 3840 19%
Number of 4 input LUTs: 1255 out of 3840 32%
    Number used as logic: 1146
    Number used as Shift registers: 41
    Number used as RAMs: 68
Number of IOs: 159
Number of bonded IOBs: 105 out of 173 60%
Number of BRAMs: 3 out of 12 25%
Number of GCLKs: 3 out of 8 37%
Number of DCMs: 1 out of 4 25%

Capítulo 7

Conclusões e trabalho futuro

Temos que a tecnologia de lógica reconfigurável (FPGA) é cada vez mais utilizada. E para isso notaram-se os contributos de alguns factores, tais como, um preço de compra cada vez mais baixo e competitivo, a existência de cada vez mais ferramentas e com melhor qualidade. Nos cartões de memória *flash* também se verifica um crescimento acentuado, muito devido, tal como nas FPGAs, aos seus preços cada vez mais baixos e a cada vez maiores capacidades de armazenamento. Sendo que a relação custo/capacidade de armazenamento está a tender para zero.

O Sistema foi desenvolvido com base no que os fabricantes e consumidores de sistemas embutidos actualmente procuram, mas deixando espaço para a natural evolução das tecnologias.

A nível de *hardware* o Sistema foi desenvolvido numa tecnologia de lógica reconfigurável permitindo assim que seja possível uma continua actualização, substituição e acrescento de módulos de hardware.

O programa desenvolvido para o PicoBlaze é dependente da tecnologia e de alguma dificuldade/rigidez para se poder fazer actualizações. No entanto teve-se uma especial atenção na concepção do código relativo ao protocolo de comunicação com a aplicação, por forma a que um programador possa facilmente alterar o protocolo usado e assim se permite alguma flexibilidade na integração deste sistema noutro.

Os testes indicam que ao nível da camada física o sistema implementado, usando um relógio de 50MHz, permite fazer transferências de dados próximas da taxa máxima dos cartões (25MB/s). Já a camada do sistema de ficheiros é mais lenta ao nível do processamento de dados, sendo a limitadora da taxa de transferência. O PicoBlaze necessita de fazer cerca de 10 a 15 instruções por cada leitura ou escrita e usando-se uma frequência de relógio de 90MHz dá 45MIPS o que resulta numa taxa de transferência aproximada de 4MB/s.

Esta taxa de transferência é suficiente na maior parte dos sistemas embutidos. Refira-se e note-se que o uso do PicoBlaze fez com que o Sistema global ocupe uma área muito pequena, permitindo uma grande flexibilidade na integração deste sistema num outro e uma diminuição dos custos de implementação, pois é gratuito.

A implementação do sistema de leitura e escrita de cartões de memória MMC/SD numa FPGA mostrou-se perfeitamente viável, utilizando-se cerca de 35% da capacidade de lógica da Spartan 3 que tem 200k *gates*. Todos os módulos de *hardware* utilizados estão documentados e podem ser facilmente substituídos, no futuro, por módulos mais optimizados ou com funções adicionais que possam ser úteis à aplicação final.

Por último, pode-se realizar um estudo comparativo das três configurações possíveis para este sistema *hardware*, *software* e misto de *hardware/software*.

Totalmente em *hardware* Verifica-se que a configuração totalmente em *hardware* é a que tem maior taxa de processamento de dados, dá a possibilidade de usar técnicas de paralelismo e *pipelining* que optimizam o sistema significativamente. Mas tem a desvantagem de ocupar uma área relativamente elevada, não sendo possível sequer usar uma Spartan 3.

Totalmente em *software* É a solução mais lenta das três em processamento de dados, onde o processador necessita processar varias instruções para executar as rotinas sendo por isso pouco eficiente. Para aumentar a taxa de transferência ter-se-ia que usar um processador com capacidade para um relógio de elevada frequência, por forma a aumentar a capacidade de processamento. Para além disso esta solução gasta recursos ao processador, pois, tipicamente neste tipo de sistemas, o processador tem outras funções para executar. A área ocupada nesta configuração é constante e depende do processador usado pelo que necessário se torna apenas uma memória de tamanha adequado ao *software* usado.

Misto de *Hardware/Software* Um sistema misto de *hardware/software* é o mais equilibrado numa relação área desempenho. Nesta situação tirasse partido das melhores características que cada configuração oferece. As rotinas que necessitem de elevada capacidade de processamento criam-se em módulos de *hardware*, sendo assim executadas num "mini" sistema embutido dedicado a execução da rotina e para alem disso é executado em paralelo. Os módulos que não necessitem de grande velocidade mas que poderiam ocupar alguma área em *hardware*, são executados em software ocupando espaço em memória.

Nome	Hardware	Hardware e Software	Software
Recursos físicos utilizados	-	+	+/-
Taxas de transferência	++	+/-	-
Custos	-	+/-	+
Resultado final	+/-	+	+/-

Tabela 7.1: Comparação de configurações de sistema

Concluindo, foi desenvolvido um sistema viável, funcional, uma solução de fácil integração, que pode ser usado num sistema embutido que necessite de armazenamento adicional de dados. Pode ser usado em vários tipos de aplicações desde as mais simples que necessitam apenas de

guardar dados de sensores, até às aplicações mais complexas em que apenas funciona como auxiliar do CPU. É suficientemente genérico para ser integrado num grande número de sistemas embutidos a ao mesmo tempo suficientemente optimizado para o uso em aplicações dedicadas.

7.1 Trabalhos futuros

Indica-se de seguida algumas melhorias a fazer ao sistema final, mas ter-se-á que ter em consideração que a memória do PicoBlaze está quase esgotada e para que se possa implementar as melhorias tem-se primeiro que libertar algum espaço:

1. Diminuir a área ocupada em *hardware*
2. Optimizar o desempenho do *hardware*
3. Diminuir e optimizar o *software*
4. Criar uma memória intermédia só para a tabela FAT. Devido ao número de vezes que a tabela é lida quando se escreve ou lê um ficheiro, faz todo o sentido ler o sector em uso da tabela uma vez só e gravá-lo numa BRAM. A tabela é acedida a partir dessa memória não tendo o sistema que estar constantemente a ler o sector da tabela FAT, assim acede de imediato a essa memória tornando o processo mais rápido.
5. Devido a complexidade de fazer a contagem de bytes em *software*, o valor gravado nas informações do ficheiro relativamente ao seu tamanho é arredondado ao número de sectores utilizados multiplicado por 512 (tamanho do sector). Uma melhoria que se pode fazer é desenvolver em *hardware* um contador externo ao microcontrolador.
6. A rotina que acede ao circuito externo do comparação do nome do ficheiro não está totalmente implementada, deixa-se como trabalho futuro.

Anexo A

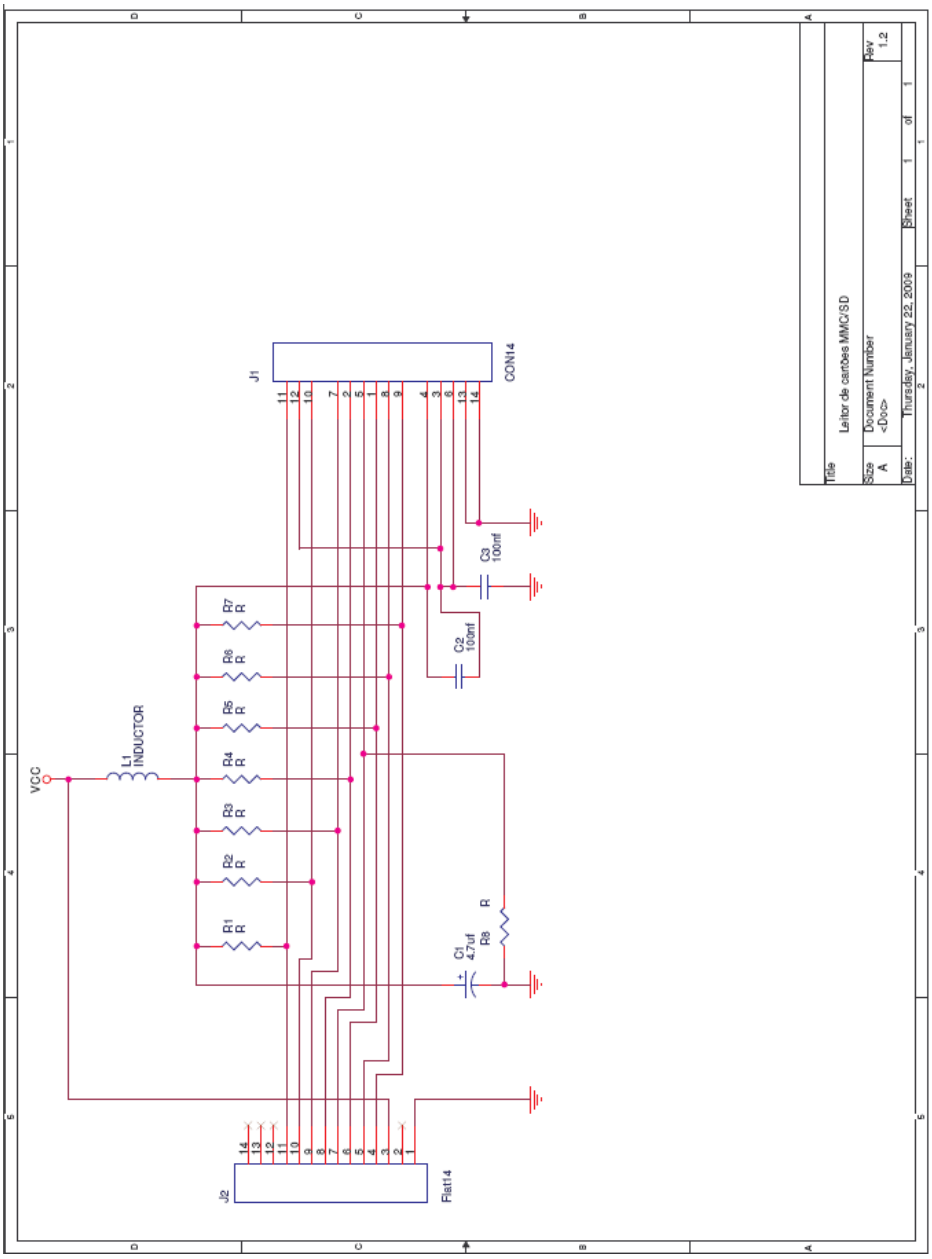

Figura A.1: Esquematico do circuito de leitura de cartões

Anexo B

	Min	Max	Unit
NCS	0	-	8 ciclos de relógio
NCR	1	8	8 ciclos de relógio
NRC	1	-	8 ciclos de relógio
NAC	1	10 * (TAAC + NSAC)	8 ciclos de relógio
NWR	1	-	8 ciclos de relógio
NEC	0	-	8 ciclos de relógio
NDS	0	-	8 ciclos de relógio

Tabela B.1: Constantes de tempo [5] [3]

Anexo C

Comandos	Argumento	Resp.	Data	Abreviação	Descrição
CMD0	Nenhum	R1	Não	GO_IDLE_ STATE	Reset do software
CMD1	Nenhum	R1	Não	SEND_OP_ COND	Iniciar o processo de inicialização
ACMD41(*1)	*2	R1	Não	APP_SEND_ OP_COND	Iniciar o processo de inicialização, para os cartões SD
CMD8	*3	R7	Não	SEND_IF_ COND	Verificar a tensão, apenas para cartões SD v2
CMD9	Nenhum	R1	Sim	SEND_CSD	Ler o registo CSD
CMD10	Nenhum	R1	Sim	SEND_CID	Ler o registo CID
CMD12	Nenhum	R1b	Não	STOP_ TRANS- MISSION	Parar a leitura de dados
CMD16	tamBloco[31:0]	R1	Não	SET_ BLOCKLEN	Mudar o tamanho do bloco a ler/escrever
CMD17	Endereço[31:0]	R1	Sim	READ_SINGLE_ BLOCK	Ler um único bloco
CMD18	Endereço[31:0]	R1	Sim	READ_ MULTIPLE_ BLOCK	Ler múltiplos blocos
CMD23	Número de blocos[15:0]	R1	Não	SET_BLOCK_ COUNT	Definir o número de blocos a transferir na próxima leitura/escrita de múltiplos blocos, apenas para MMC
ACMD23(*1)	Número de blocos[15:0]	R1	Não	SET_WR_ BLOCK_ ERASE_ COUNT	Definir o número de blocos a pré-apagar no próximo comando de escrita, apenas para cartões SD
CMD24	Endereço[31:0]	R1	Sim	WRITE_BLOCK	Escrever um bloco
CMD25	Endereço[31:0]	R1	Sim	WRITE_ MULTIPLE_ BLOCK	Escrever múltiplos blocos
CMD55(*1)	Nenhum	R1	Não	APP_CMD	Comando especifico da aplicação
CMD58	Nenhum	R3	Não	READ_OCR	Ler OCR

Tabela C.1: Lista dos principais comandos do modo SPI [5] [3]

Referências

[1] "Weakening Consumer Spending Impacts NAND-type Flash Memory Market". *Cellular News*, Abril 2004.

[2] David Pellerin e Milan Saini. "FPGAs Provide Acceleration for Software Algorithms". *FPGA and structured ASIC*, 2004.

[3] SD Association. "SD Specifications", Setembro 2006. Simplified Specification Physical Layer Part 1.

[4] David Woodhouse. "The Journalling Flash File System". *Ottawa Linux Symposium*, páginas 1 – 12, 2001.

[5] SanDisk. *"MultiMediaCard Product Manual"*, Abril 2000. Suporta a v1.4 da especificação MMC.

[6] Michael Barr. "Programmable Logic: What's it to Ya?". *Embedded Systems Programming*, páginas 75 – 84, Junho 1999.

[7] Andrew S. Tanenbaum. *"Structured Computer Organization"*. Prentice Hall, 1999.

[8] Microsoft Corporation. "FAT32 Series Datasheet", Dezembro 2000.

[9] Charles Manning. "YAFFS spec", Dezembro 2001.

[10] Xilinx. *"PicoBlaze 8-bit Embedded Microcontroller User Guide"*, ug129 (v1.1.2) edição, Junho 2008.

[11] Xilinx. *"MicroBlaze Processor Reference Guide"*, ug081 (v9.0) edição, Janeiro 2008.

[12] Tiago Oliveira Ribeiro. "Sistema de ficheiros em FPGAs". Faculdade de Engenharia da Universidade do Porto, 2006.

[13] Opencores. *"WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores"*.