Universidade de Aveiro
2009/2010

Departamento de Electrónica, Telecomunicações e Informática

Carlos David Alexandre Serra

"ANÁLISE E IMPLEMENTAÇÃO DE ORDENAÇÃO DE DADOS EM FPGA"

Universidade de Aveiro
2010

Departamento de Electrónica, Telecomunicações
e Informática

Carlos David Alexandre Serra

ANÁLISE E IMPLEMENTAÇÃO DE ORDENAÇÃO DE DADOS EM FPGA

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do Grau de Mestre em Electrónica e Telecomunicações, realizada sob a orientação científica do Professor Doutor Valeri Skliarov, Professor Catedrático do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro e Investigador do Instituto de Engenharia Electrónica e Telemática de Aveiro (IEETA), co-orientação científica da Professora Doutora Iouliia Skliarova, Professora Auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro e investigadora do Instituto de Engenharia Electrónica e Telemática de Aveiro (IEETA).

agradecimentos / acknowledgments

Antes de mais gostaria de agradecer aos meus orientadores, o Prof. Doutor Valeri Skliarov e Prof.ª Doutora Iouliia Skliarova, por terem sido excelentes orientadores, por toda a ajuda, motivação e disponibilidade que sempre tiveram comigo ao longo da elaboração desta tese.

Agradeço também aos meus colegas de trabalho, particularmente João Lima e Luís Figueiredo pela ajuda prestada. Foi juntamente com o Luís que a parte inicial do projecto foi desenvolvida.

Por fim, agradeço aos meus pais e ao meu irmão, a toda a minha família, aos meus amigos e a todos os que ao longo do meu percurso académico me ajudaram a chegar até aqui.

o júri / the jury

presidente	**Prof. Dr. António Manuel de Brito Ferrari Almeida** Professor Catedrático da Universidade de Aveiro
orientador	**Prof. Dr. Valeri Skliarov** Professor Catedrático da Universidade de Aveiro
co-orientadora	**Prof.ª Dra. Iouliia Skliarova** Professora Auxiliar da Universidade de Aveiro
arguente	**Prof. Dr. Hélio Mendes de Sousa Mendonça** Professor Auxiliar da Faculdade de Engenharia da Universidade do Porto

palavras-chave FPGA, árvores binárias, máquinas de estados finitos

resumo

Desde os primórdios da computação que os algoritmos de ordenação têm sido investigados. Estes podem ser baseados em diferentes tipos de estruturas de dados. A sua implementação num dado sistema permite um acesso mais eficaz aos dados armazenados em memória.

O aumento da capacidade de processamento da FPGA (Field Programmable Gate Arrays) torna possível a implementação de algoritmos de ordenação que actuem sobre listas de dados de tamanho razoável.

Nesta tese foi desenvolvida uma aplicação de *software*, assim como um circuito a ser implementado em FPGA, que permitem realizar a transferência de 1024 dados do sistema computacional de uso geral para a FPGA, através de ligação USB. Os dados enviados possuem 16 bits, com gama de valores entre 0 e 65535 e são criados pela aplicação de *software* desenvolvida. Os dados são ordenados na FPGA e no sistema computacional de uso geral usando a estrutura de dados árvore binária. Posteriormente visualizam-se estes valores e o tempo necessário para os ordenar, tanto no monitor, ligado à placa com FPGA, como na consola do computador de uso geral. No final, é comparado o tempo necessário para ordenar nos dois sistemas. A FPGA utilizada foi a Spartan-3E, da Xilinx®.

keywords FPGA, binary trees, finite-state machines

abstract

Sorting algorithms have been investigated since the beginning of computing era. Their implementation in a system optimizes the process of data access. These algorithms may be based on different kinds of data structures.

The increase of the processing capacity of FPGA (Field Programmable Gate Arrays) allows for the implementation of sorting algorithms that act upon data lists of considerable size.

In this thesis, a software application and a circuit to be implemented in an FPGA were developed, which allow for the transfer of 1024 data values from the general purpose computer system to the FPGA, via USB interface. The values sent possess 16 bits, ranging from 0 to 65535 and are created by the developed software application. The data were sorted using the binary tree data structure both in software and in the FPGA. The sorted results were presented on a VGA monitor screen connected to the FPGA board and in the console output of the developed application. The respective sorting time, calculated both in software and in the FPGA, were compared and analyzed. An FPGA of Spartan-3E family of Xilinx was used as a hardware platform.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Sumário

Neste capítulo apresentar-se-á um pequeno enquadramento do mundo das FPGAs (Field Programmable Gate Arrays). Além disso explica-se porque foi importante trabalhar neste projecto.

1.1 Enquadramento

Nos últimos anos, a tecnologia utilizada na construção de circuitos integrados evoluiu imenso, permitindo construir ICs (*integrated circuits*) cada vez mais pequenos e complexos, dando eventualmente origem aos *System-on-Chip* (SoC), onde vários componentes distintos, como microprocessadores, memórias RAM (Random Access Memory) e ROM (Read-Only Memory), componentes I/O (In/Out), etc., são montados no mesmo *chip*. Este crescimento, aliado à existência de linguagens de descrição de *hardware* (HDL – Hardware Description Language), ofereceu aos projectistas uma maior liberdade e rapidez na criação de novos circuitos digitais.

Um dos resultados na criação de ICs cada vez mais pequenos foi o aparecimento da FPGA, que permitia aos projectistas criar e testar um circuito digital no próprio *hardware,* uma vez que as FPGAs podem ser configuradas com novos projectos sempre que o projectista assim o deseje. Estas características, aliadas ao seu baixo custo, criaram o conceito de projectista no local, onde os circuitos são desenhados à medida das necessidades do local onde se encontra, abrindo caminho a um maior crescimento no uso de sistemas reconfiguráveis. As FPGAs, tal como todos os restantes sistemas digitais, evoluíram de forma bastante rápida, sendo neste momento já possível encontrar FPGAs com capacidade de processamento bastante elevadas.

Os algoritmos de ordenação são um dos tópicos de estudo mais antigos em sistemas de computação. O propósito destes algoritmos é a disposição de dados segundo uma determinada ordem, de modo a optimizar o processo de busca e acesso. Uma das soluções investigadas, a estrutura de dados em árvore binária, pode também ser implementada em FPGA, de maneira a tornar a inserção, ordenação e extracção de dados uma tarefa mais rápida e eficaz.

1.2 Motivação

Tal como foi dito anteriormente, as FPGAs actuais possuem já uma boa capacidade de processamento, tornando a implementação de algoritmos de ordenação nestes sistemas uma opção cada vez mais viável. Este tipo de algoritmos tem sido estudado ao longo dos anos. Nesta tese focou-se a atenção nas árvores binárias, desenvolvendo-se um algoritmo baseado neste tipo de estrutura, de maneira a obter-se uma forma eficaz de armazenamento, ordenação e extracção de dados.

Uma vez que se prevê também uma maior necessidade de interacção das FPGAs com o exterior, seja através de computadores, monitores, teclados e mesmo com outras placas FPGA (de forma a aumentar as potencialidades que estes dispositivos apresentam), desenvolveu-se uma aplicação de *software* e uma descrição de circuito que permitem efectuar a troca de dados entre um sistema computacional de uso geral e a FPGA através do porto USB (Universal Serial Bus), estando essa comunicação apta a corresponder aos requisitos propostos inicialmente, isto é, transmissão de 1024 valores de dados, sendo estes inteiros sem sinal, possuindo 16 bits (gama de valores entre 0 e 65535). A FPGA utilizada foi uma Xilinx Spartan-3E-500 FG320. Ao tentar cumprir estes objectivos, pretende-se assim criar conhecimento acerca dos componentes e uso da FPGA. A figura seguinte mostra os sistemas utilizados e o percurso dos dados:

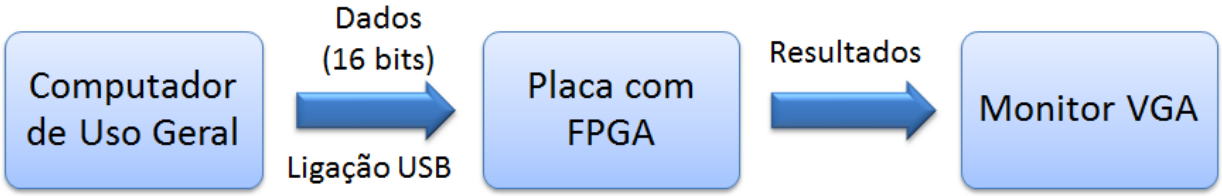


Fig. 1.1 – Sistemas utilizados e percurso dos dados no trabalho.

1.3 Objectivos

Para a realização deste trabalho foram delineados os seguintes objectivos:

- Desenvolver uma aplicação de *software* que permita a troca de dados por USB entre computador e FPGA.
- Desenvolver um circuito que suporte a comunicação USB do lado da FPGA.
- Gerar uma lista aleatória no computador possuindo 1024 valores, em que cada um pode variar entre 0 e 65535 (valores de 16 bits).

- Transferir essa lista para a FPGA.

- Ordenar essa lista, tanto no computador de uso geral como na FPGA, usando um algoritmo de árvore binária. A FPGA apresenta os resultados no monitor, o computador na janela de comandos.

- Medir tempos de ordenação no computador e na FPGA e tirar conclusões.

1.4 Estrutura da tese

Esta dissertação encontra-se dividida em sete capítulos, de maneira a explicar as diferentes partes que foram estudadas e trabalhadas ao longo da tese.

- **Capítulo 2 – FPGA e HDLs** – Neste capítulo o foco é dado à FPGA e às características das linguagens de descrição de *hardware*. É feita uma pequena revisão da evolução dos dispositivos lógicos programáveis. São discutidos a arquitectura da FPGA usada e os componentes que a compõem, assim como as suas funções e as principais empresas que as fabricam.

- **Capítulo 3 – Ferramentas de Apoio** – Aqui são descritas as ferramentas utilizadas na realização do trabalho. Para além disso é apresentada a noção de IP *Core*, sendo descrito o tipo *Memory Generator*, já que foi o mais utilizado nesta tese. São também descritos os ficheiros de configuração BIT e as suas fases de criação, bem como os ficheiros de restrições UCF.

- **Capítulo 4 – Árvores Binárias** – Neste capítulo é introduzida a noção de árvore binária. São também explicadas as suas características, vários tipos e propriedades. Para realçar a aplicação do algoritmo de ordenação baseado em árvore binária em diversas áreas práticas, são apresentados vários projectos publicados que utilizaram esta estrutura de dados de maneira a tornar o seu sistema mais rápido.

- **Capítulo 5 – Estrutura do Projecto e Ligação Computador-FPGA –** São apresentadas as descrições de circuito utilizadas no projecto, a ligação Computador-FPGA implementada através de USB e os sinais que regulam essa comunicação. É também descrito o programa que gera a lista de valores aleatórios.

- **Capítulo 6 – Máquinas de Estados Finitos –** É aqui introduzida a noção de máquinas de estados finitos (FSM), sendo descritas as FSM utilizadas neste projecto.

- **Capítulo 7 – Resultados e Conclusões –** Aqui são apresentadas as conclusões acerca do sucesso dos objectivos propostos para esta tese, fazendo uma análise aos resultados. É também apresentada uma proposta para trabalho futuro.

- **Anexo A –** É apresentado em código VHDL a FSM que serviu para fazer a ordenação da lista gerada pelo sistema computacional de uso geral na FPGA, utilizando o algoritmo de árvore binária.

Capítulo 2

FPGA e HDLs

Sumário

Neste capítulo apresenta-se uma breve história de como surgiram as FPGAs, tal como alguns dos seus antepassados. Será feita uma análise dos componentes e da arquitectura da FPGA usada nesta tese.

Descrevem-se também algumas características das linguagens de descrição de *hardware*.

2.1 Introdução

As FPGAs são circuitos integrados reconfiguráveis. A sua popularidade está a aumentar, devido ao seu relativo baixo custo e também ao facto de poderem ser reconfiguradas pelo utilizador o número de vezes desejado. Esta característica torna possível optimizar descrições de circuitos, sejam fornecidos pela fábrica ou por outras fontes, para o fim a que se destinam (daí o nome *Field Programmable* – programáveis no campo). Nesta tese foram utilizadas especificações disponíveis no *site* da Digilent Inc. [1], que mais tarde foram adaptadas às necessidades existentes.

Numa FPGA é possível descrever praticamente qualquer circuito digital lógico. Isso pode ser feito desenhando um esquemático do circuito lógico ou através de uma linguagem de descrição de *hardware*. Nesta tese foi utilizada a VHDL. Depois do circuito lógico estar descrito, ele é sintetizado no computador, utilizando uma ferramenta própria para esse efeito (*ISE* WebPACK da Xilinx®). Posteriormente é criado um ficheiro binário que pode ser usado para configurar a FPGA. O circuito pretendido fica assim implementado na FPGA.

Na FPGA utilizada neste trabalho o ficheiro binário que descreve o funcionamento do circuito pode ser carregado para a sua PROM (Programmable Read-Only Memory) ou para a RAM, sendo que neste último caso a FPGA necessita ser reconfigurada sempre que se desligue a energia.

2.2 Uma breve história da evolução dos sistemas lógicos programáveis

Antes de existirem dispositivos em que fosse possível reconfigurar circuitos digitais lógicos era necessário utilizar *chips* de circuitos digitais integrados, com apenas algumas portas (estas portas poderiam ser AND, OR, NOT, etc.), sendo que para criar um circuito lógico relativamente complexo era necessário juntar um grande número (algumas dezenas) destes *chips* numa placa, interligando-os fisicamente, o que tornava dispendioso montar e modificar um circuito deste tipo.

Com a introdução das PALs (Programmable Array Logic) o problema da complexidade foi atenuado, pois permitiu a construção de circuitos com peças que eram menores, mais baratas e mais rápidas. Permitiu também que fosse possível programar o *chip* de maneira a representar o circuito digital pretendido, sem ser necessário modificar fisicamente as ligações entre as portas. A PAL (ver Fig. 2.1) é constituída por duas matrizes, uma de portas AND (programável) e outra de portas OR (fixas). Embora sendo *field-programmable*, usando sistemas electrónicos especiais, só podiam ser programadas uma vez.

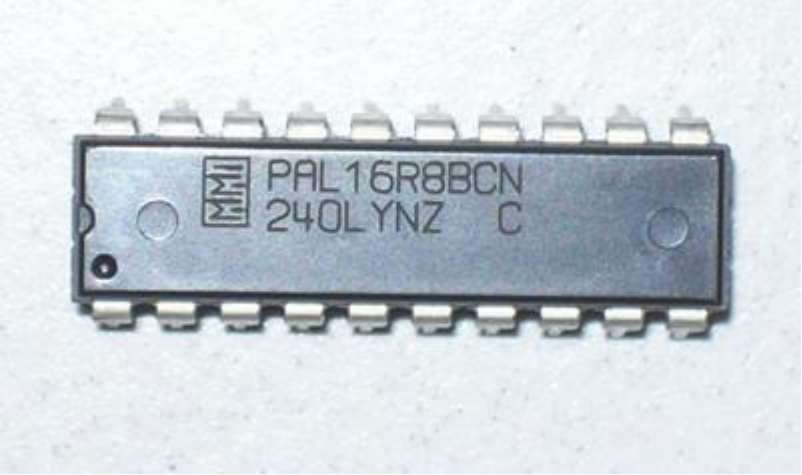


Fig. 2.1 – Imagem da MMI PAL 16R8. [2]

Um avanço nos PLDs (Programmable Logic Device) foi a GAL (Generic Array Logic, ver fig. 2.2), que tinha as mesmas vantagens da PAL, mas era também possível ser reprogramada, pois usava tecnologia EEPROM (Electrically Erasable Programmable Read-Only Memory). No entanto, não resolvia o facto de os *chips* conterem relativamente poucos recursos lógicos (equivalentes a algumas centenas de portas lógicas).

Fig. 2.2 – Duas GALs da Lattice Semiconductor. [3]

Sensivelmente ao mesmo tempo apareceram no mercado dois tipos de PLDs que permitiam a sua reprogramação e possuíam um grande número de portas: os CPLDs (Complex Programmable Logic Device) e as FPGAs (Field-Programmable Gate Array). Estes dois dispositivos são bastante semelhantes, mas possuem algumas características diferentes:

- Um CPLD é basicamente um conjunto de PLDs ligados entre si, dentro de um único dispositivo. Pode ser programado a dois níveis, ao nível dos PLDs e ao nível da sua matriz de interligação. O número de PLDs é relativamente baixo, utilizando uma arquitectura que é designada por "*coarse-grain*" (grãos grossos, ver fig. 2.3). Tal como uma GAL, o CPLD pode ser reprogramado e a sua execução é iniciada assim que receba energia.

- Uma FPGA é constituída por um grande número de pequenos blocos de lógica reconfigurável ("*fine-grain*" ou grão fino, ver fig. 2.3), separados por *switches*, que são programados para interligar os blocos, de maneira a construir o circuito desejado.

Geralmente, as FPGAs são melhores em situações onde o circuito digital a implementar é mais complexo, enquanto as CPLDs são melhores para circuitos mais simples.

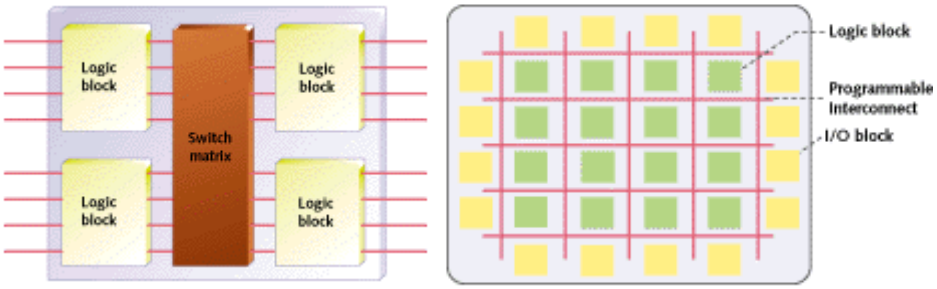


Fig. 2.3 – Arquitectura interna de uma CPLD (à esquerda) e FPGA (à direita). [4]

Embora existam várias companhias que produzem FPGAs, apenas duas delas se destacam no mercado [5]:

- Xilinx, Inc. [6] – Inventou as FPGAs em 1985. Monopolizou completamente o mercado até meados dos anos noventa, quando começaram a surgir as primeiras empresas concorrentes. Actualmente é a empresa líder, possuindo acima de 50% do mercado de FPGAs.

- Altera Corporation [7] – Rival de longa data da Xilinx, é a segunda companhia de FPGAs mais influente no mercado. Em conjunto com a Xilinx, possuem cerca de 80% de cota no mercado [8].

- Outras – Várias companhias têm tentado impor-se no mercado. Entre elas, destacam-se a Lattice Semiconductor Corporation [9], Actel Corporation [10], SiliconBlue Technologies Corporation [11] e a QuickLogic Corporation [12].

De modo a cobrir um maior espectro no mercado as empresas oferecem diferentes tipos de FPGAs, tentando ir de encontro às posses e necessidades dos seus clientes alvo. Assim, para clientes com maiores recursos e em que as suas aplicações são mais exigentes, podem ser adquiridas FPGAs mais avançadas (como as FPGAs da família Virtex-6 da Xilinx), enquanto que para orçamentos mais modestos existem FPGAs mais fracas (como as Spartan). As FPGAs das famílias Virtex possuem um maior número de recursos especiais, são mais rápidas e possuem maior quantidade de blocos lógicos configuráveis que as das famílias Spartan, sendo assim mais dispendiosas.

2.3 Blocos da FPGA

Enquanto um CPLD usa blocos PLD grandes, a FPGA usa pequenas células lógicas de forma a obter qualquer função lógica. Essas células são compostas por LUTs (Look-

Up Table), *flip-flops* D e *multiplexers*. O número de cada um destes componentes numa *slice* (bloco de lógica configurável) varia conforme a FPGA utilizada.

Outra funcionalidade que se pode atribuir a uma *slice* é a salvaguarda de dados (através do *flip-flop*). O número de entradas da célula lógica varia entre as três e as dez e as saídas entre uma e duas, dependendo da função pretendida pelo projectista e do tipo de FPGA em questão.

Através da combinação de várias *slices* da FPGA pode-se descrever um circuito lógico bastante complexo. A interligação das *slices* é feita através de ligações programáveis, que controlam a direcção dos dados entre uma *slice* e a outra.

Para além dos blocos combinatórios, que permitem descrever funções lógicas, as FPGAs possuem também outros recursos, tais como blocos RAM (BRAM) e multiplicadores embutidos. Embora os blocos combinatórios possam ser usados para guardar informação, para um grande volume de dados essa opção torna-se impraticável, existindo assim a alternativa de guardar dados usando o BRAM. Da mesma maneira, embora seja possível calcular o produto entre dois operandos utilizando blocos de lógica configurável (CLBs), através de um circuito lógico que calcule o produto, a existência de multiplicadores dedicados torna essa operação muito mais eficiente.

Nos dias de hoje, dizer se uma FPGA tem mais recursos do que outra não é tão fácil como antigamente, pois quando as FPGAs foram criadas não possuíam recursos especiais, pelo que o seu tamanho era determinado pelo número equivalente de portas. Para além de ser preciso ter em conta os recursos especiais, o facto de que estes só melhoram o desempenho de algumas aplicações torna a comparação entre FPGAs ainda mais difícil.

Resumindo, a arquitectura das FPGAs é constituída por:

- Blocos de Lógica Configurável (CLBs) ou *slices* – São blocos que permitem implementar qualquer função lógica, assim como servir de memória distribuída. Cada um contém Look-Up Tables (LUT).

- Blocos de Entrada/Saída (IOBs) – Regulam o sentido da informação entre os dispositivos lógicos da FPGA com os pinos de encapsulamento.

- Bloco RAM – Cada FPGA da família Spartan-3E possui duas colunas de blocos RAM dentro de um anel IOB (Input/Output Blocks), à excepção da XC3S100E, que possui apenas uma. O uso destes blocos permite efectuar a salvaguarda de um maior número de dados do que seria possível utilizando apenas os CLBs.

- Blocos multiplicadores – Na família Spartan-3E, os seus operandos possuem 18 bits. A existência destes blocos torna a multiplicação mais rápida e eficaz.

- Digital Clock Managers (DCMs) – São blocos especiais que permitem a multiplicação e divisão do sinal relógio. Basicamente, ajudam o projectista a escolher uma frequência de relógio que mais lhe convém. Nesta tese foi usado apenas um bloco DCM, não para alterar a frequência, mas para estabilizar o sinal de relógio de 50 MHz.

2.4 Placa de desenvolvimento

A placa de desenvolvimento utilizada neste trabalho foi a Nexys-2, da Digilent Adept Inc. (ver fig. 2.4).

Fig. 2.4 – Placa com FPGA Nexys-2, desenvolvida pela Digilent Adept Inc.[13]

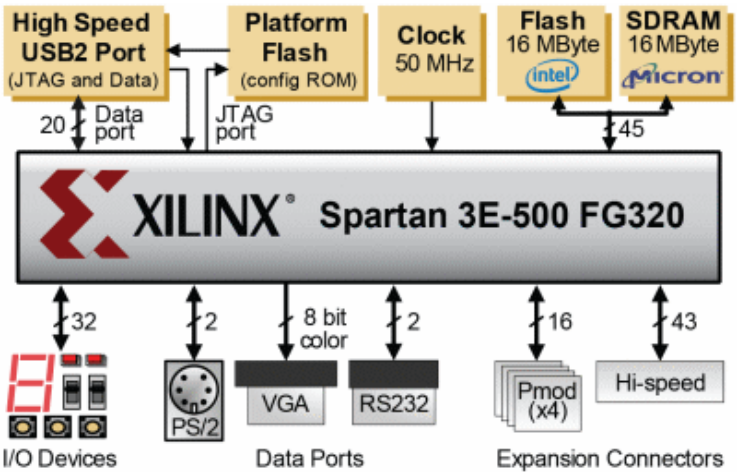


Fig. 2.5 - Diagrama de blocos da placa Nexys-2. [13]

A placa Nexys-2 possui as seguintes características [13] (ver fig. 2.5):

- FPGA Xilinx Spartan 3E-500 FG320;
- Porto USB 2.0, que permite configuração e transferência de dados para a FPGA;
- Alimentação através de bateria, transformador ou por USB;
- Memórias SDRAM da Micron e *flash* da Intel, ambas com capacidade de 16MB;
- Memória ROM *flash* para a configuração da FPGA;
- Oscilador de 50MHz;
- Conectores PS/2, VGA (8 bits de cor) e RS232;
- Inclui 8 LEDs, 4 *displays* de 7 segmentos, 4 botões e 8 interruptores.

2.5 Linguagens de descrição de *hardware*

De maneira a desenhar os circuitos lógicos pretendidos podem ser usados esquemáticos desenhados em editores. Estes editores podem ser obtidos através do fornecedor da FPGA. Em alternativa aos esquemáticos pode utilizar-se um tipo de linguagem especialmente criada para descrever a estrutura e comportamento de circuitos, como as linguagens de descrição de *hardware* (HDL).

O facto de as HDLs incluírem noções explícitas de tempo e permitirem paralelismo, isto é, a execução de processos de forma simultânea, torna este tipo de linguagens distintas das linguagens de programação. Na linguagem C, por exemplo, o paralelismo não é possível, pois cada instrução é executada sequencialmente. No entanto, como a descrição de um circuito em HDL é um processo bastante moroso, existem esforços para desenvolver conversores de C para HDL [14] [15].

As HDLs permitem não só descrever um circuito lógico tendo em vista a sua aplicação num dispositivo de *hardware*, mas também a simulação desse mesmo circuito (e consequente depuração), o que é bastante importante, pois o seu teste em *hardware* gasta muito mais tempo ao projectista.

Para simular o código pretendido, é utilizado um programa onde o projectista cria um ficheiro que descreve os sinais de entrada e inclui o bloco a simular, chamado *testbench*. Depois de executar a simulação, o simulador apresenta as formas de onda

dos sinais que interessa testar, permitindo ver de forma rápida a zona do tempo pretendido, as transições dos sinais, etc. O projectista pode ver rapidamente se existe algum erro e, caso isso aconteça, onde ele se encontra.

Experimentar o código directamente em *hardware* apresenta algumas desvantagens em relação à sua simulação. A síntese ocupa muito mais tempo, e, uma vez implementado o circuito, não é possível ter uma visão das formas de onda dos seus sinais. No entanto, oferece uma visão do que acontece na realidade, já que mesmo que o código não apresente quaisquer problemas na simulação, não é de todo garantido que funcione de igual modo num sistema físico.

As linguagens de descrição de *hardware* mais utilizadas são a Verilog e a VHDL. Tanto uma linguagem como a outra são relativamente difíceis de aprender, embora duas semanas sejam o suficiente para se ficar com os conceitos básicos. No entanto, para se dominar qualquer uma delas, é necessário entre seis meses a um ano de aprendizagem. Nesta tese foi utilizada a VHDL.

2.6 Conclusões

Neste capítulo apresentaram-se alguns dispositivos reconfiguráveis que precederam a FPGA. Descreveu-se a FPGA utilizada na tese, assim como algumas particularidades das linguagens de descrição de *hardware*.

Capítulo 3

Ferramentas de Apoio

Sumário

Neste capítulo são apresentadas as ferramentas que foram utilizadas na construção do projecto, sendo também dada uma introdução básica relativamente ao seu uso. É apresentado o conceito de IP *core*, descrevendo-se aquele que foi mais utilizado no projecto, o *Memory Generator*.

3.1 Ferramentas de apoio

O desenvolvimento de um projecto é suportado através de várias ferramentas de apoio. O mercado oferece diferentes escolhas no que toca a ferramentas de simulação e de síntese, no entanto nem todas são de uso livre. A Xilinx oferece ferramentas de projecto livres tanto para Windows como para Linux (ISE WebPack, uma versão reduzida do ISE completo), enquanto a Altera só oferece para Windows (Quartus II Web Edition, que é a versão reduzida do Quartus II; a versão para Linux está disponível apenas no pacote inteiro). Embora sejam versões reduzidas, são uma óptima escolha, tendo em conta aquilo que conseguem fazer. Embora estejam limitadas a dispositivos de pequeno e médio tamanho, nos dias de hoje esses dispositivos possuem já bastantes capacidades. O preço que custaria adquirir a versão completa em ambos os casos seria acima dos $2000, sendo a licença válida apenas para 12 meses [5]. Nesta tese foi utilizado o ISE WebPack.

É ainda mais fácil encontrar simuladores livres na internet do que ferramentas de projecto. Neste trabalho foi utilizado o simulador da Mentor Graphics, o ModelSim.

3.2 Criação de um projecto usando o ISE WebPack

Em seguida irão ser descritos os passos básicos para a criação de um novo projecto utilizando o ISE WebPack.

Passo 1 – Clicar em *New Project*, na janela *File* (ver fig. 3.1).

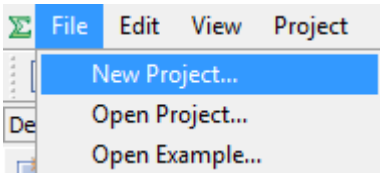


Fig. 3.1 – Iniciar um novo projecto no ISE.

Passo 2 – Na janela seguinte é possível escolher o nome do projecto, a localização onde irá ser guardado, assim como a sua descrição (opcional) (ver fig. 3.2).

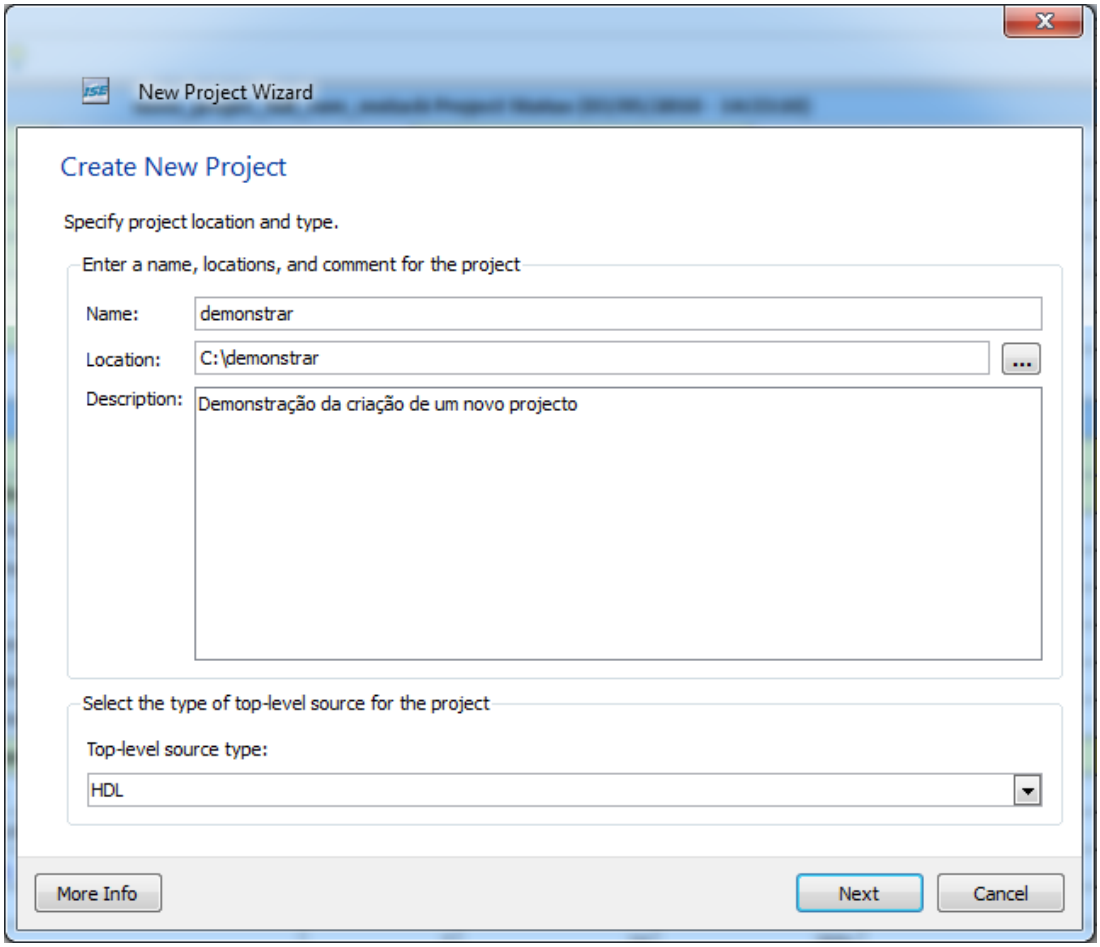


Fig. 3.2 – Janela de Criação de um Novo Projecto.

Passo 3 – Nesta janela é necessário escolher as propriedades do dispositivo. De maneira a seleccionar a FPGA onde se pretende implementar o projecto escolhe-se a família (ex: Spartan-3E), o número do dispositivo (ex: XC3S500E, XC3S1200E...), o seu pacote (ex: FG320) e a sua velocidade (ex: -4). Estas características podem ser vistas no *chip* FPGA, tal como a figura 3.3 exemplifica:

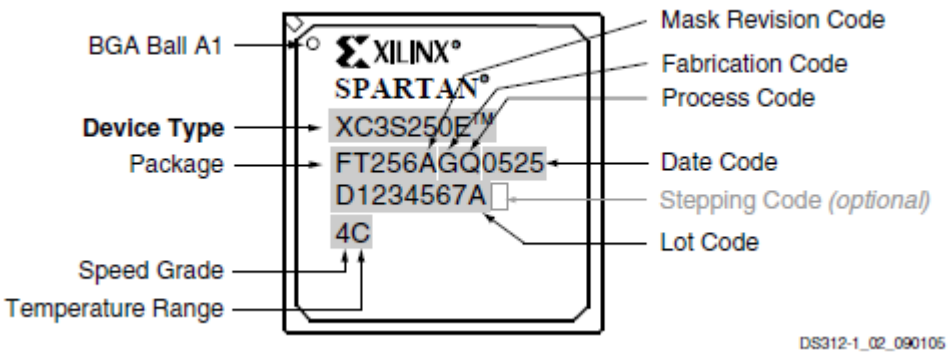


Fig. 3.3 – Características do modelo de uma FPGA. [16].

- Em *Device Type* encontra-se o modelo da Spartan. Nesta tese foi utilizado principalmente o modelo XC3S500E. Os números que se encontram entre o S e o E referem-se ao número de portas, sendo que o *chip* da imagem tem 250 000 portas de sistema.

- O modelo (*package*) pode ser visto na segunda linha. Na FPGA XC3S500E, a sua *package* é FG320.

- *Speed Grade* (índice de velocidade) é um termo relativo, variando de família para família e de marca para marca. Pode simbolizar o tempo que demora um LUT ou o tempo que demoram as passagens de 1 para 0 e de 0 para 1. Na família Spartan, quanto mais elevado for o número, mais rápida será a FPGA (-5 significa *High Performance*, -4 *Standard Performance*). A FPGA utilizada nesta tese possui um *Speed Grade* de -4.

Em *Synthesis Tools* (ver Fig. 3.4) é possível escolher a ferramenta de síntese. Para a versão livre do ISE, está apenas disponível a ferramenta XTS. As ferramentas *Synplify* [17], *Synplify Pro* e *Precision* [18] oferecem uma melhor optimização do circuito descrito, mas têm de se adquirir separadamente.

Em *Simulator* (ver Fig. 3.4) é possível escolher qual o simulador a utilizar, caso o projectista deseje simular código. O ISE já traz o seu próprio simulador (Isim), no entanto também é possível usar outros, como por exemplo o ModelSim.

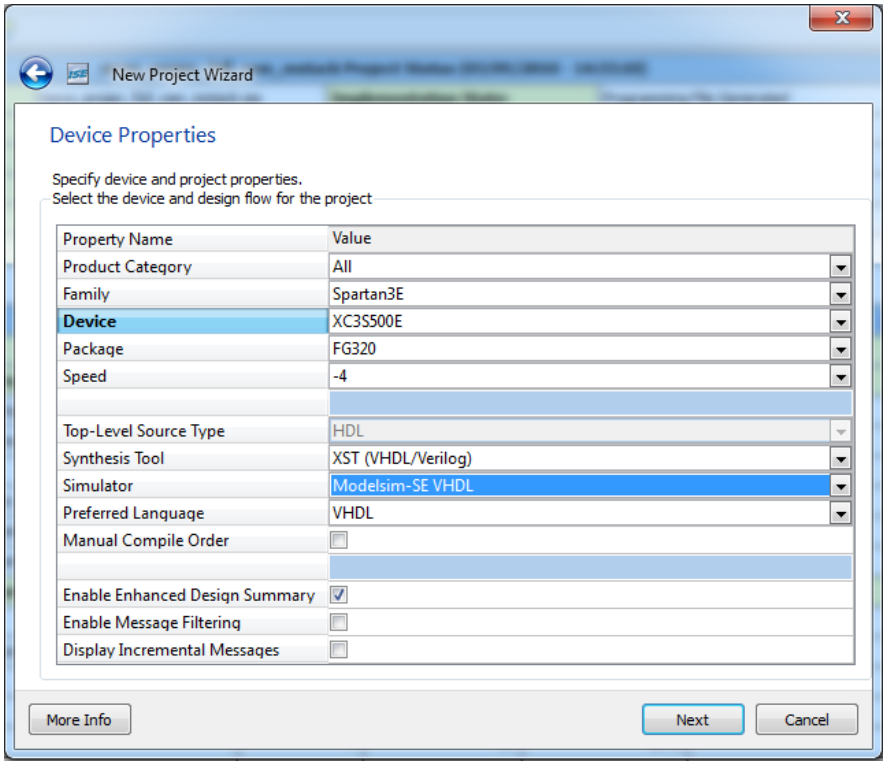


Fig. 3.4 – Definição das propriedades do dispositivo.

Em seguida, o ISE WebPack oferece a possibilidade de criar um *source file* ou adicionar ficheiros existentes ao projecto. Na janela final é possível ver um resumo de todas as especificações escolhidas (ver fig. 3.5).

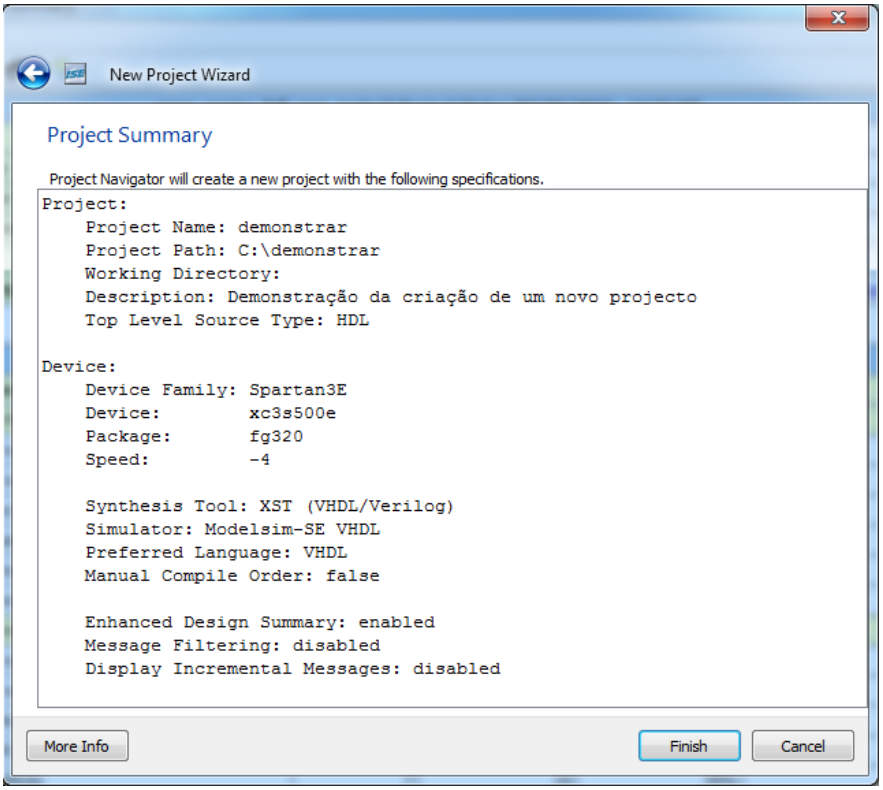


Fig. 3.5 – Resumo final das características do projecto a ser criado.

Depois de o projecto ter sido criado deve especificar-se a forma como o relógio irá ser inicializado na FPGA. Para isso, clica-se em *Generate Programming File* seguido de *Process -> Process Properties -> Startup Options*. Em *FPGA Start-Up Clock* (ver fig. 3.6) é possível escolher entre:

- CCLK – A sequência de início é sincronizada com o relógio de configuração da FPGA (CCLK). Este relógio é gerado internamente caso a FPGA seja a *Master*. Caso seja a *Slave* o CCLK é obtido através de um sinal de entrada. Esta é a opção predefinida e deve ser usada caso o ficheiro de configuração seja carregado na PROM.

- User Clock – Opção raramente utilizada, apenas escolhida quando se pretende que a sequência de relógio seja sincronizada com um sinal definido pelo utilizador.

- JTAG Clock – Esta deve ser a opção utilizada quando se carrega o ficheiro de configuração para a FPGA usando ligação USB. Sincroniza a sequência de início com o JTAG Test Clock (JTC).

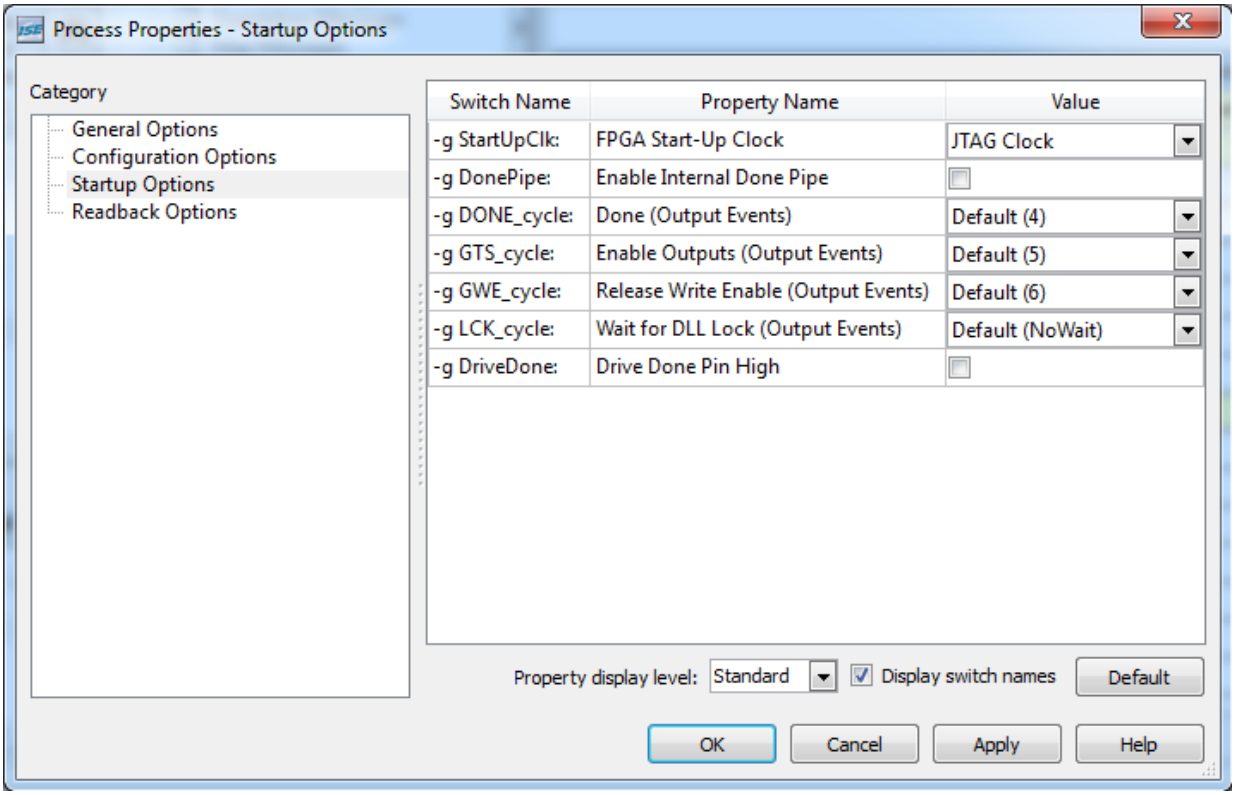


Fig. 3.6 – Janela *Startup Options*, onde se pode escolher o relógio de inicialização.

3.3 Ambiente gráfico do ISE WebPack

O ISE WebPack possui um ambiente gráfico onde é possível desenhar os circuitos lógicos, gerir os ficheiros, verificar a correcção sintáctica e sintetizar o projecto. Ao abrir o ISE, ou logo a seguir à criação de um novo projecto, é possível ver o *Design Summary*, que mostra o resumo de todo o projecto, recursos ocupados e outras informações relativas ao desempenho do projecto na FPGA.

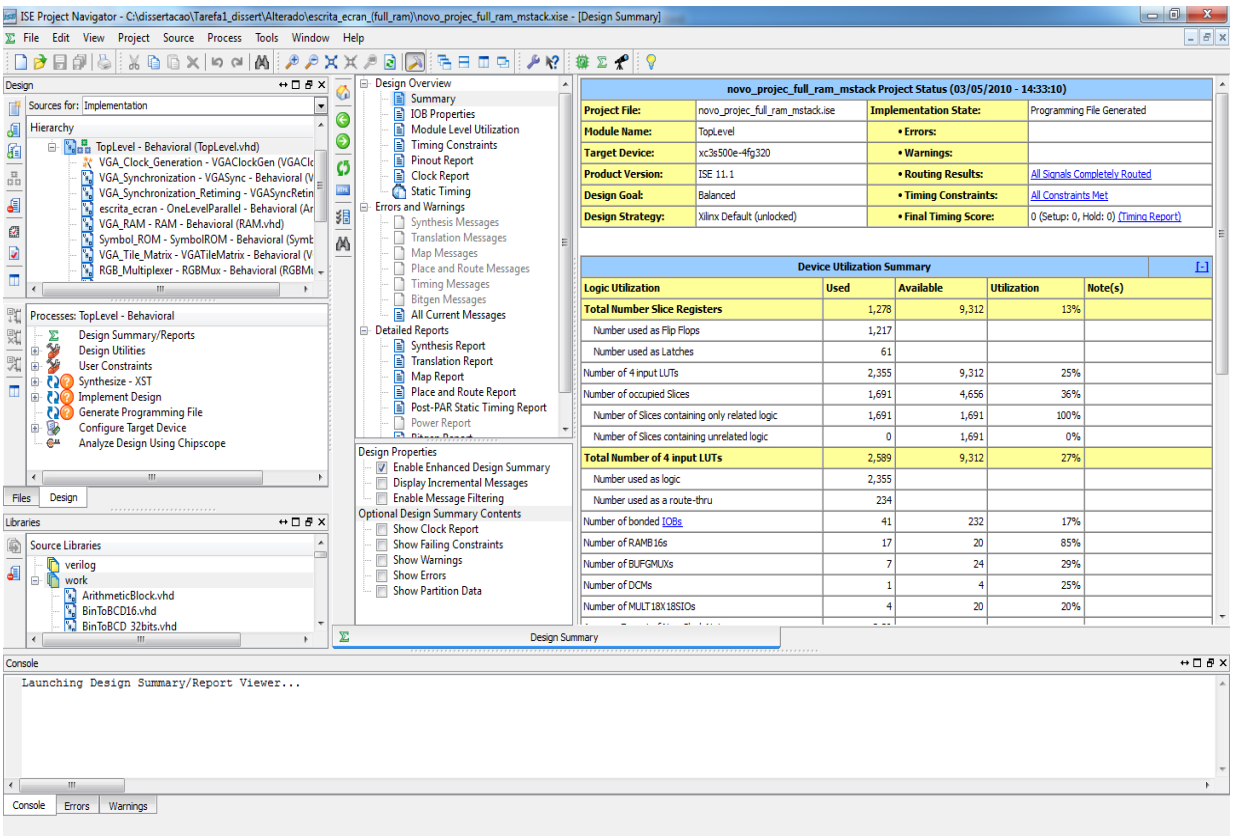


Fig. 3.7 – Imagem do ambiente gráfico do ISE WebPack.

O ambiente gráfico apresenta 5 janelas (ver fig. 3.7):

- ***Hierarchy*** – É possível ver a organização hierárquica dos ficheiros que compõem o projecto, com os de nível superior a aparecer deslocados mais à esquerda e os de nível mais baixo deslocados à direita. Pode-se também escolher se o projecto irá ser implementado ou simulado.

- ***Processes*** – Seleccionando um determinado ficheiro de projecto, esta janela mostra os processos possíveis. Para um processo de baixo nível não há muitas opções disponíveis. Seleccionando o ficheiro *top-level* de um projecto, esta janela mostra as opções de síntese, implementação e geração do ficheiro de configuração, entre outras.

- ***Libraries*** – É possível ver todos os ficheiros relacionados com o projecto, não só do tipo VHD, mas também UCF, bibliotecas, IP *Cores*, etc.

- ***Janela Principal*** – Pode ver-se o *Design Summary*, assim como os ficheiros do projecto (VHD, UCF, etc.).

- ***Console*** – Nesta janela o ISE avisa o projectista sempre que haja erros, avisos ou qualquer outro desenvolvimento que seja importante tomar conhecimento. O projectista pode também dar ordens ao ISE através de comandos nesta janela.

3.4 Etapas da criação de um ficheiro de configuração no ISE

Depois de se iniciar um projecto, é necessário criar ficheiros contendo código HDL que permitem descrever um determinado circuito lógico. Posteriormente, o projectista pode, através do ISE WebPack, criar um ficheiro que contenha a descrição desse circuito. O ficheiro criado é do tipo BIT e de maneira a criá-lo o ISE passa por várias fases (ver fig. 3.8), que podem ser acompanhadas por simulações [19] [20] [21] [22]:

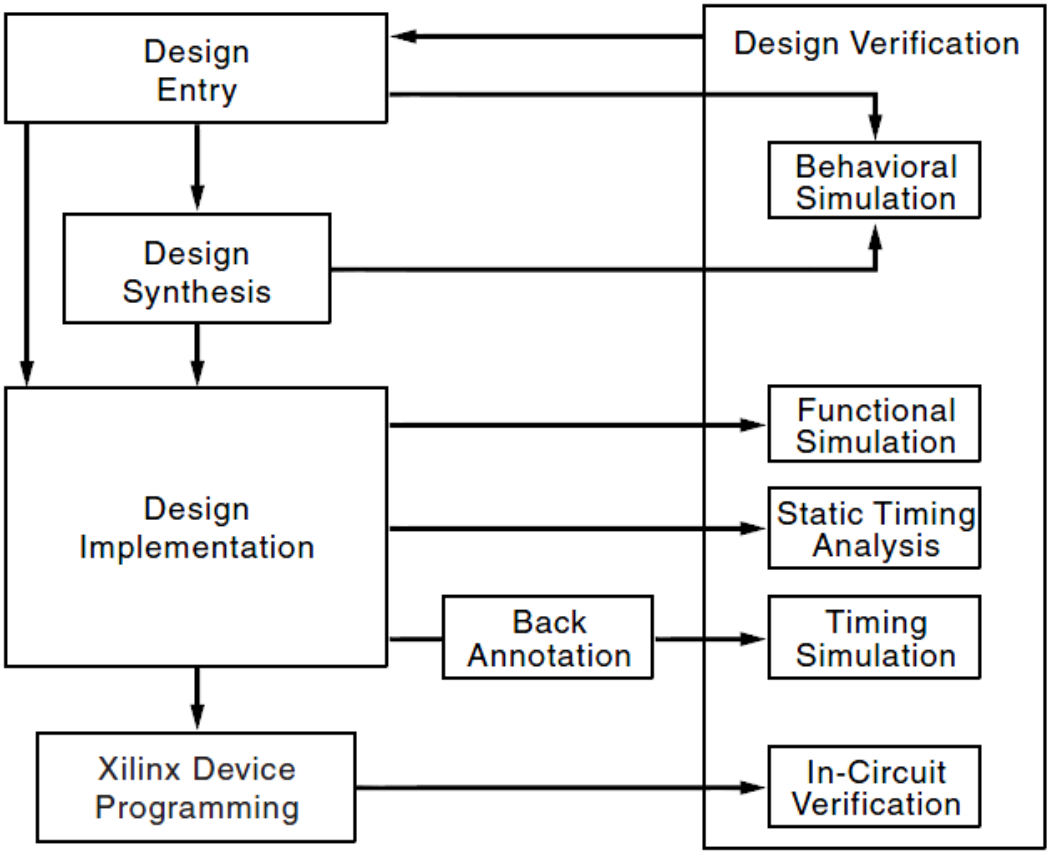


Fig. 3.8 – Fases no desenvolvimento de um projecto na ferramenta ISE.

Design Entry – O circuito lógico do projecto é descrito através de linguagens de descrição de *hardware*, IP *cores*, ficheiros UCF ou esquemáticos.

Design Synthesis – Durante a síntese é verificada a sintaxe na descrição do circuito. A forma como irá ser efectuada depende da ferramenta de síntese escolhida. Tal como já foi dito, a ferramenta que vem com o ISE WebPack é a XST. Caso haja erros, o ISE avisa o projectista através da consola e aborta a síntese. Caso contrário, o ISE cria um ficheiro designado por *netlist* que irá ser utilizado na etapa seguinte.

Design Implementation – Esta fase consiste em três subfases: *Translate*, *Map* e *Place and Route*. Estes passos são específicos para a Xilinx, uma vez que na Altera *Translate* e *Map* são combinados num único passo. O ficheiro UCF é adicionado aqui para indicar os portos da FPGA a utilizar (ver fig. 3.9). Em *Translate* o resultado da *Design Synthesis* é combinado num único ficheiro, num *Native Generic Database* (NGD). Na subfase *Map*, o ISE tenta colocar o circuito descrito nos recursos disponíveis da FPGA, tais como CLBs e IOBs. Esta etapa produz um ficheiro *Native Circuit Description* (NCD), o qual contém informação acerca de atrasos de *switching*, mas não de propagação. Em *Place and Route* são determinadas as ligações entre os componentes físicos. Este é o passo mais moroso de toda a implementação. No final é calculado o atraso total que existe nos circuitos implementados, o que irá influenciar a frequência máxima de relógio permitida. O ficheiro NCD é actualizado, sendo utilizado na fase seguinte para criar o ficheiro de configuração BIT.

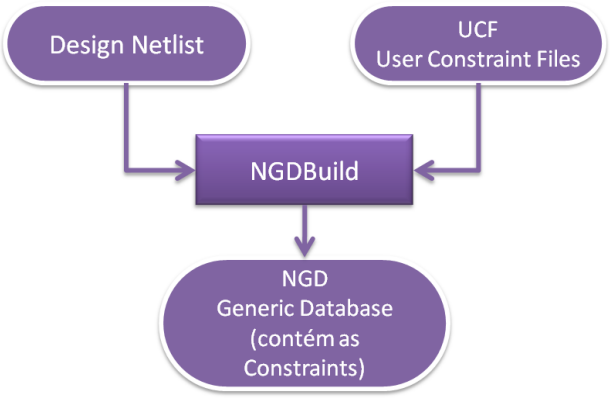


Fig. 3.9 – Criação de um ficheiro NGD contendo as restrições. [23].

Xilinx Device Programming – Nesta fase é gerado o ficheiro BIT. Através de cabos de ligação, por exemplo através de cabo JTAG, o ficheiro pode ser carregado para a FPGA.

Ao longo de todas estas fases o projectista pode simular o circuito a nível comportamental (*behavioral simulation*), funcional (*functional simulation*) e temporal (*timing simulation*) de maneira a certificar-se da integridade do código. Qualquer erro detectado aborta o processo de criação do ficheiro BIT.

3.5 Ficheiros UCF

Para fazer a ligação entre o circuito descrito e os blocos de entrada/saída da FPGA é usado um ficheiro de texto em que são especificadas quais as restrições de tempo e de localização que irão ser utilizadas. As restrições temporais indicam quais os caminhos que são mais importantes, sendo indicado à ferramenta de implementação que a sua colocação deve ser o mais próximo possível. As restrições de localização servem para controlar o mapeamento de elementos lógicos e indicar a que portos se referem os sinais de entrada e de saída do circuito. Por exemplo, caso se utilizem os LEDs, botões ou os *displays* da placa, é preciso indicar no ficheiro UCF a sua localização. Apenas os pinos que se pretende utilizar devem estar nesse ficheiro. Caso se possua um "*Master* UCF" com todos os códigos das ligações aos pinos, deve-se comentar (utilizando o carácter # no início da linha a comentar) as ligações que não irão ser utilizadas.

O nome dos sinais dos pinos são *case sensitive* e cada frase é terminada por um ponto e vírgula (;). No entanto, as restrições reservadas da Xilinx, como LOC, HIGH, LOW, não são *case sensitive*. O nome do pino a especificar tem de ser igual ao seu nome nos ficheiros VHD do projecto.

```
NET "mclk" TNM_NET = "Clock50In";
TIMESPEC "TS_Clock50In" = PERIOD "mclk" 50 MHz HIGH 50 %;
#------------------- Clock ---------------------------------------------------#
NET "Clock50In"     LOC = "B8";
#------------------- Buttons -------------------------------------------------#
NET "Button<3>"     LOC = "H13";
NET "Button<2>"     LOC = "E18";
NET "Button<1>"     LOC = "D18";
NET "UserReset"     LOC = "B18";  # Button 0
##------------------- VGA Monitor -------------------------------------------#
NET "VGARed<2>"     LOC = "R8";
NET "VGARed<1>"     LOC = "T8";
NET "VGARed<0>"     LOC = "R9";
NET "VGAGreen<2>"   LOC = "P6";
NET "VGAGreen<1>"   LOC = "P8";
NET "VGAGreen<0>"   LOC = "N8";
NET "VGABlue<1>"    LOC = "U4";
NET "VGABlue<0>"    LOC = "U5";
NET "HSync"         LOC = "T4";  # Horizontal Synchronization
NET "VSync"         LOC = "U3";  # Vertical Synchronzation
##-----------------------------------------------------------------------------
NET "astb"  LOC = "V14"  ;
NET "dstb"  LOC = "U14"  ;
NET "pdb<0>" LOC= "R14"; # Bank = 2 , Pin name = IO_L24N_2/A20 , Type = DUAL , Sch name = U-FD0
NET "pdb<1>" LOC= "R13"; # Bank = 2 , Pin name = IO_L22N_2/A22 , Type = DUAL , Sch name = U-FD1
NET "pdb<2>" LOC= "P13"; # Bank = 2 , Pin name = IO_L22P_2/A23 , Type = DUAL , Sch name = U-FD2
NET "pdb<3>" LOC= "T12"; # Bank = 2 , Pin name = IO_L20P_2 , Type = I/O , Sch name = U-FD3
NET "pdb<4>" LOC= "N11"; # Bank = 2 , Pin name = IO_L18N_2 , Type = I/O , Sch name = U-FD4
NET "pdb<5>" LOC= "R11"; # Bank = 2 , Pin name = IO , Type = I/O , Sch name = U-FD5
NET "pdb<6>" LOC= "P10"; # Bank = 2 , Pin name = IO_L15N_2/D1/GCLK3 , Type = DUAL/GCLK , Sch name = U-FD6
NET "pdb<7>" LOC= "R10"; # Bank = 2 , Pin name = IO_L15P_2/D2/GCLK2 , Type = DUAL/GCLK , Sch name = U-FD7
NET "pwait"  LOC = "N9"  ;
NET "pwr"  LOC = "V16"  ;
```

Fig. 3.10 – Ficheiro UCF utilizado no projecto.

Na figura anterior pode observar-se a interligação entre os sinais do projecto e os pinos da FPGA. As primeiras linhas são referentes ao sinal de relógio e ao seu período. Seguem-se os códigos dos pinos ligados aos botões da placa. Depois são especificados

os pinos relacionados com a interacção com o monitor. Finalmente são referenciadas as restrições relacionadas com o porto USB.

3.6 IP *Cores*

Os núcleos de propriedade intelectual (IP *cores*) são blocos de código que desempenham uma determinada função. Quem usufrui do seu uso é geralmente alguém que não o projectista. Da mesma maneira que em programação existem bibliotecas, em descrição de *hardware* existem IP *cores*. Idealmente, um IP *core* deve ser bastante portátil, isto é, conseguir ser reutilizado quando se move de um sistema para outro ou quando o seu ambiente é alterado. A ferramenta ISE WebPack da Xilinx oferece vários tipos de IP *cores* e apresenta-os tendo em conta não só a sua função, como também a sua área de aplicação (ver Tabela 1), por exemplo: Automóvel e Industrial, Elementos de Memória, Processamento Digital de Sinal, Processamento de Imagem e Vídeo, entre outras. Neste projecto foi usado principalmente o *Block Memory Generator*, um IP *core* do tipo Elemento de Memória, de maneira a fazer uso de blocos de memória embutidos na FPGA.

Plataforma alvo	Tipos de IP	IP *Cores*
Plataformas Base	Blocos de Construção	➢ Memórias e FIFOs ➢ Operadores Aritméticos ➢ Operadores de Vírgula Flutuante
	Depuração e Procura de Erros	➢ ChipScopeTM Pro Integrated Chip ➢ Analisador Lógico Integrado ➢ Input/Output Virtual
	Recursos da Arquitectura da FPGA	➢ Clocking Wizard ➢ Memory Interface Generator (MIG) ➢ RocketIO™ Multi-Gigabit Transceivers (MGTs) ➢ System Monitor Wizard
	Conectividade	➢ Interfaces de barramento de dados tais como o PCI TM e PCI-X™ ➢ Interfaces de ligação em rede, tais como Ethernet, SPI-4.2, RapidIO, CAN e PCI EXPRESS®
		➢ DDS, FIR, FFT, etc.

Domínios Específicos	Funções DSP	➢ Forward Error Correction, tais como Codificador e Descodificador de Reed-Solomon, Descodificador de Viterbi, etc.
	Processamento de Imagens e Vídeo	➢ Conversores Cor-Espaço ➢ Matriz de Conversão de *Cores*, Correcção de Gama, Escalador de Vídeo, etc.
Mercados Específicos	Automóvel e Industrial	➢ CAN, Ethernet AVB, etc.
	Telecomunicações por fios	➢ Ten Gigabit Ethernet MAC, Tri-mode Ethernet MAC, etc.
	Telecomunicações sem fios	➢ Encoder/Decoder de Canais LTE, Pesquisador 3GPP, etc. ➢ CPRI, OBSAI e Serial Rapid IO, etc.

Tabela 1 – Vários tipos de IP *cores*, fornecidos no ISE WebPack [24].

O *Block Memory Generator* permite inicializar e especificar o número de blocos RAM embutidos na FPGA que se pretende utilizar. É possível juntar vários blocos de forma a criar uma única RAM com o tamanho necessário.

Seguidamente são mostrados os passos de maneira a implementar um BRAM usando o *Memory Generator*.

Com o ficheiro *Top-Level* seleccionado, ir a *Project* => *New Source* (ver fig. 3.11). Em alternativa, escolher *New Source* clicando no botão direito no ficheiro *Top Level*.

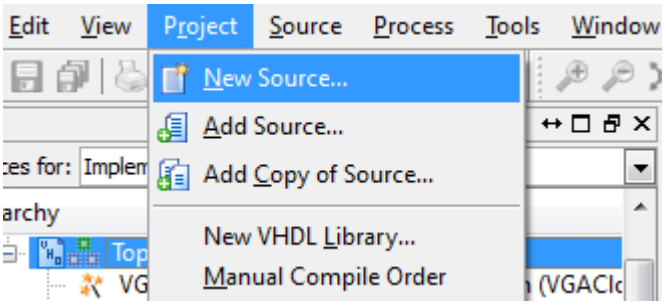


Fig. 3.11 – Primeiro passo para usar o *Memory Generator*.

Na janela seguinte selecciona-se IP (CORE Generator & Architecture Wizard) e dá-se um nome ao bloco de memória a gerar (ver fig. 3.12).

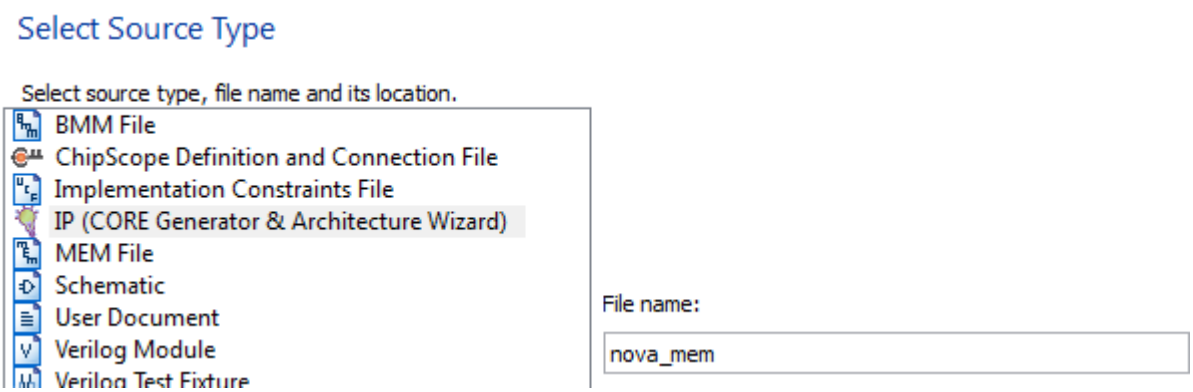


Fig. 3.12 – Escolher IP *Core*.

De seguida, escolhe-se o IP *Core* que se pretende utilizar. Tal como foi dito anteriormente, o ISE WebPack vem com vários IP *Cores*, para diferentes fins. O *Memory Generator* pode ser encontrado em *Basic Elements* -> *Memory Elements* ou em *Memories & Storage Elements* -> RAMs & ROMs. Em alternativa pode escrever-se o nome na barra de procura (ver fig. 3.13).

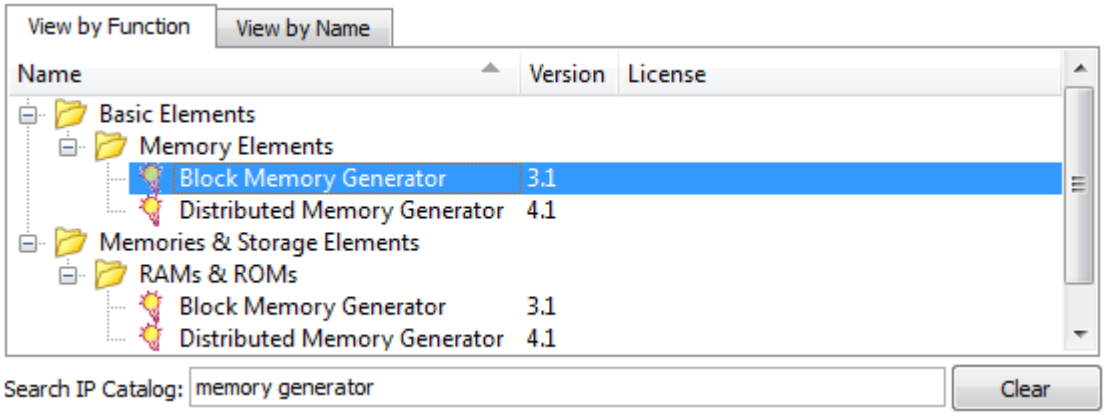


Fig. 3.13 – Escolha do IP *core Block Memory Generator*.

Depois de se escolher *Block Memory Generator* aparece um resumo do IP *core* escolhido. O ISE irá de seguida inicializar o *Memory Generator*. Na primeira janela de diálogo (ver fig. 3.14) é possível escolher os seguintes tipos de blocos RAM e ROM:

- *Single Port RAM*
- *Simple Dual Port RAM*
- *True Dual Port RAM*
- *Single Port ROM*
- *Dual Port ROM*

Embora fisicamente os blocos na FPGA sejam *dual-port*, é possível simular blocos com apenas um porto (*Single-Port*).

Cada bloco tem duas entradas completamente independentes, entrada A e entrada B, sendo que cada uma possui a sua própria frequência de funcionamento, porto de habilitação e de escrita, barramento de dados e de endereços. Estes dois portos acedem a um espaço partilhado na memória. Usar os dois portos (*Dual Port*) permite ler e escrever na memória ao mesmo tempo, tornando assim o circuito mais rápido.

A diferença entre *Simple* e *True Dual Port* é que em *True Dual Port* é possível escrever e ler nos dois portos (A e B). Em *Simple Port* o porto A serve apenas para escrever e o porto B apenas para ler. A utilização de apenas um tipo de entrada (*Single Port*) poupa recursos à FPGA.

É também possível escolher qual o tipo de optimização no que toca à área do bloco RAM. O *Memory Generator* oferece três algoritmos:

- *Minimum Area*
- *Low Power*
- *Fixed primitives*

Seleccionando o primeiro é permitido escolher o algoritmo que optimiza a área ocupada pelo bloco RAM. Nesta tese, esta foi a opção utilizada em todas as configurações de blocos RAM.

A opção *Low Power* dá ênfase à poupança de energia, minimizando o número de primitivas activas numa operação de leitura ou de escrita.

Em certos casos é necessário utilizar um tipo de primitiva fixo. A opção *Fixed Primitives* permite exactamente isso.

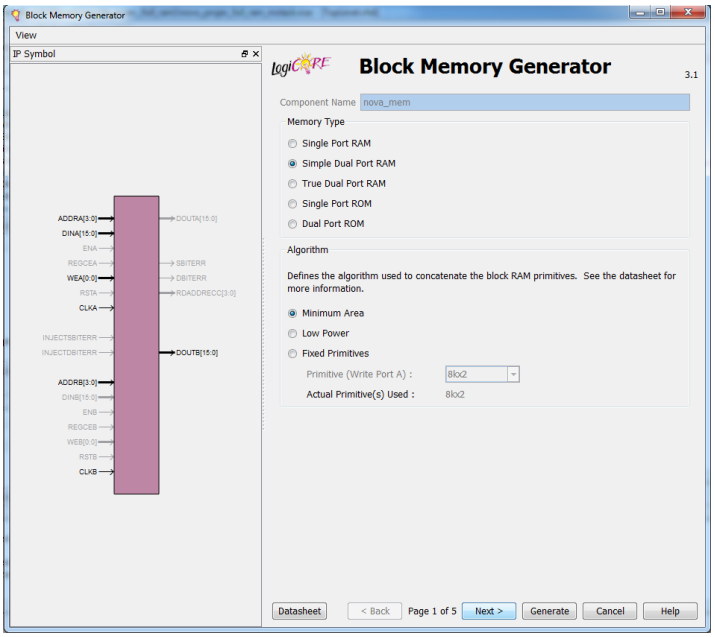


Fig. 3.14 – Primeira janela de diálogo do *Memory Generator*.

Na janela de diálogo seguinte (ver fig. 3.15) é possível especificar o tamanho do bloco RAM (o número máximo de registos e o número de bits em cada registo) e o modo de operação. Pode-se ainda escolher a existência de uma entrada de habilitação da memória (ENA), que permite ao projectista decidir quando o BRAM estará activo. Caso a existência dessa entrada não seja seleccionada, o bloco RAM estará sempre activo.

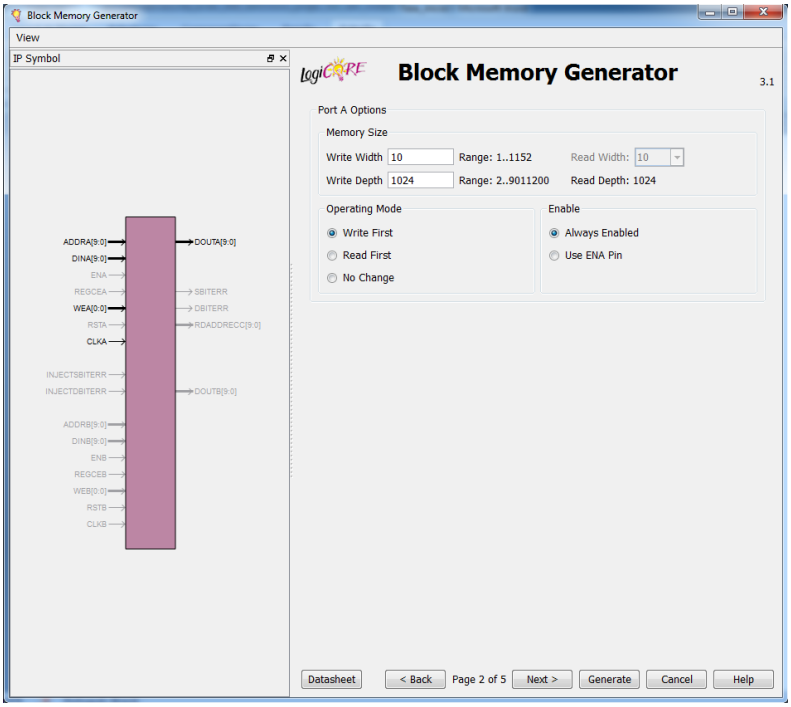


Fig. 3.15 – Segunda janela de diálogo do *Memory Generator*.

Os modos de operação permitem especificar a maneira como os registos dos dados de saída do bloco RAM serão actualizados sempre que ocorra uma operação de escrita. Existem três modos a escolher: *Write First Mode*, *Read First Mode* e *No Change Mode*.

- ***Write First Mode*** – Neste modo, durante uma operação de escrita, o registo dos dados de saída possui o valor do registo dos dados de entrada (ver fig. 3.16).

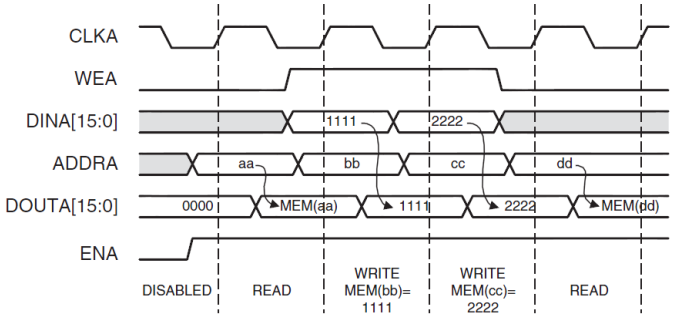


Fig. 3.16 – Exemplo de *Write First Mode*.

- ***Read First Mode*** – Durante a operação de escrita, o registo dos dados de saída contém os dados armazenados em memória no endereço especificado (ver fig. 3.17).

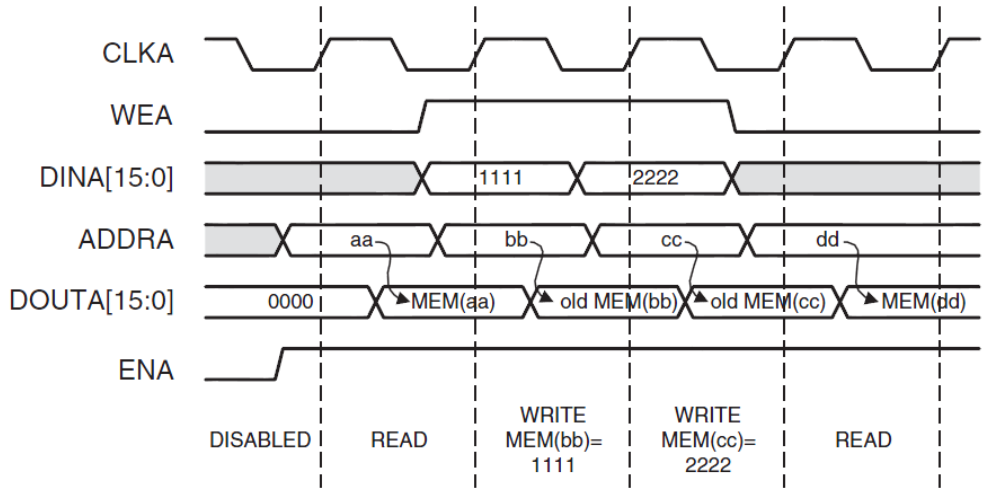


Fig. 3.17 - Exemplo de *Read First Mode*.

- ***No Change Mode*** – O registo dos dados de saída permanecem inalterados durante uma operação de escrita (ver fig. 3.18).

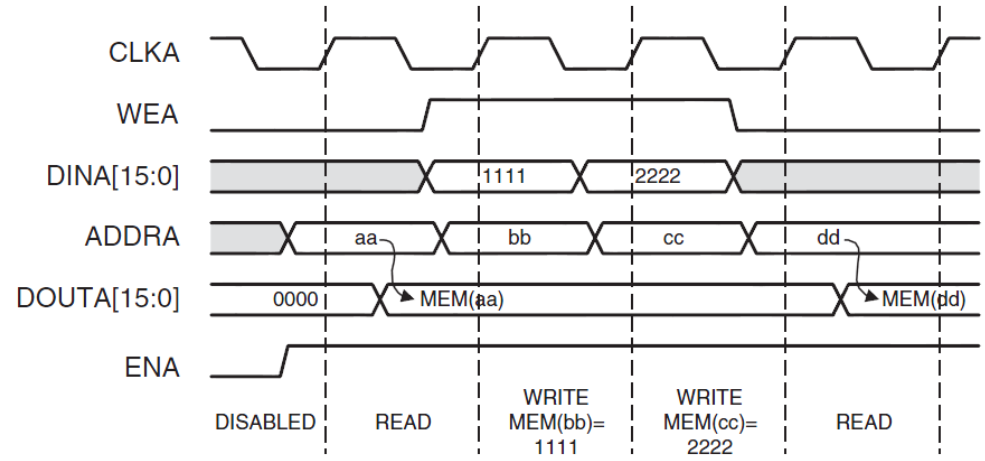


Fig. 3.18 - Exemplo de *No Change Mode*.

Na terceira janela de diálogo do *Memory Generator* (ver fig. 3.19) é possível, entre outras coisas, inicializar a memória com valores predefinidos. Para isso basta criar primeiro um ficheiro de inicialização COE e, de seguida, seleccionar a opção carregar ficheiro de inicialização (*Load Init File*), indicando o caminho até ao ficheiro. Caso não se escolha nenhum ficheiro, a memória é criada com registos nulos.

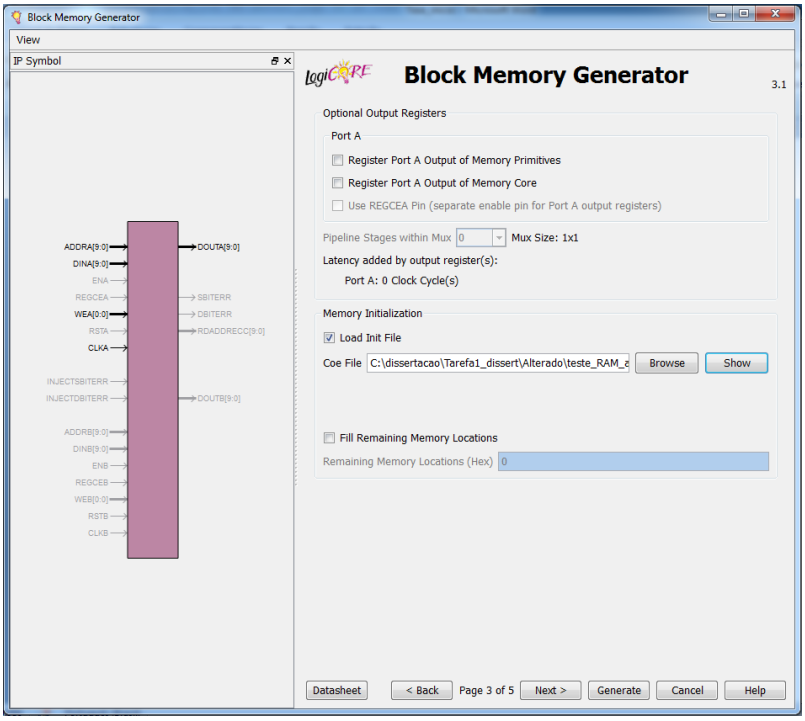


Fig. 3.19 – Terceira janela de diálogo do *Memory Generator*.

O ficheiro de inicialização permite descrever os valores que irão ser carregados para a memória BRAM. A primeira linha define a base com que os valores são guardados (base binária = 2, decimal = 10, hexadecimal = 16) e a segunda define os próprios valores. Para se inicializar uma memória BRAM com os valores de 3000 a 3009 nas posições de 0 a 9 pode usar-se o seguinte ficheiro, criado num editor de texto, como o *Notepad*:

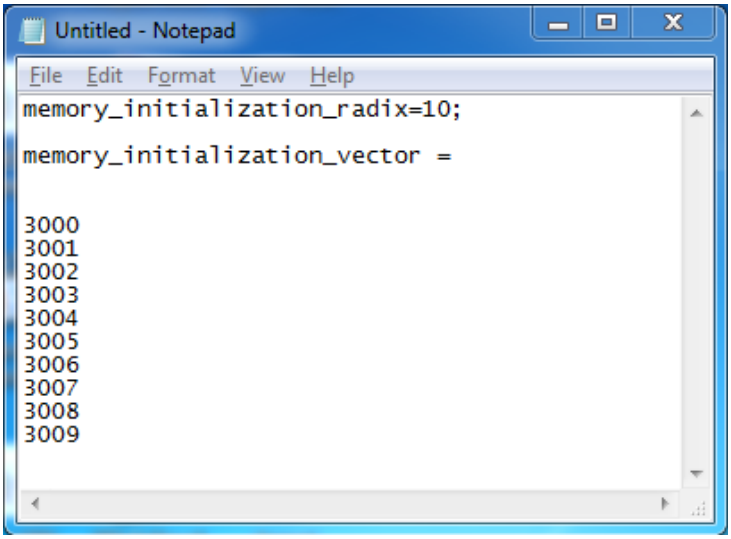


Fig. 3.20 – Exemplo de um ficheiro COE.

Depois de escolher o ficheiro COE, pode-se confirmar que os valores a serem inicializados no bloco RAM são os correctos, clicando em *Show*. O *Memory Generator* irá apresentar uma janela do seguinte tipo, onde mostra os valores em cada registo:

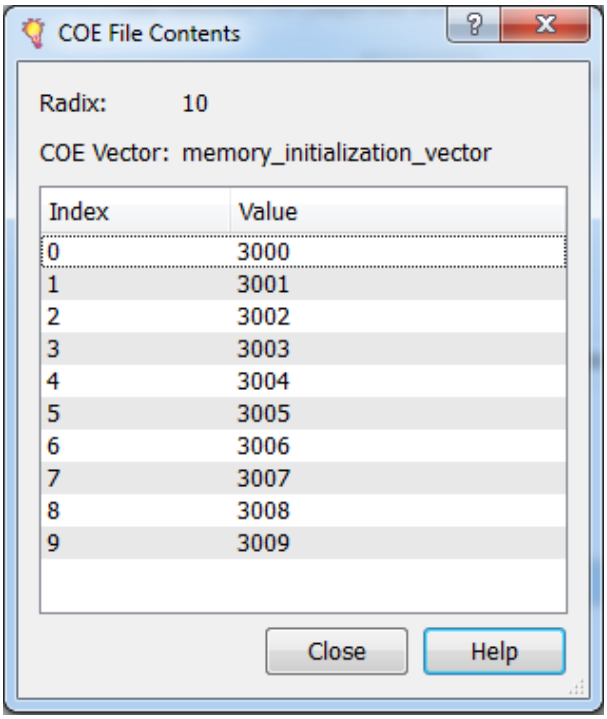


Fig. 3.21 – Visualização dos valores inicializados no BRAM.

Os passos descritos anteriormente são os mais importantes na definição de um novo bloco RAM usando o *Memory Generator*. Clicando em *Generate*, o IP *core* vai ser gerado. Uma vez feito isso basta adicioná-lo ao projecto.

A imagem seguinte mostra a disponibilidade de blocos RAM dos diferentes tipos de FPGA da família Spartan-3E da Xilinx:

Dispositivo	Número de colunas RAM	Blocos RAM por coluna	Número de Blocos RAM totais	Totalidade de bits de RAM	Totalidade de Kbits de RAM
XC3S100E	1	4	4	73728	72
XC3S250E	2	6	12	221184	216
XC3S500E	2	10	20	368640	360
XC3S1200E	2	14	28	516096	504
XC3S1600E	2	18	36	663552	648

Tabela 2 – Disponibilidade de Blocos RAM em FPGAs da família Spartan-3E [25].

3.7 ModelSim

Como já foi dito anteriormente, a maior parte do tempo gasto pelo projectista na implementação de um circuito lógico é na detecção e correcção de erros. Nesse caso, ter um bom simulador e estar consciente das suas capacidades é bastante importante.

Nesta tese, o simulador que foi utilizado sempre que houve necessidade de estudar melhor os sinais presentes num circuito foi o ModelSim. Este simulador pode ser utilizado independentemente ou pode ser iniciado através do ISE. Tal como o ISE, o ModelSim permite montar vários ficheiros utilizando diferentes linguagens de HDL.

O ModelSim possui um ambiente gráfico, o qual pode ser visto na figura 3.22.

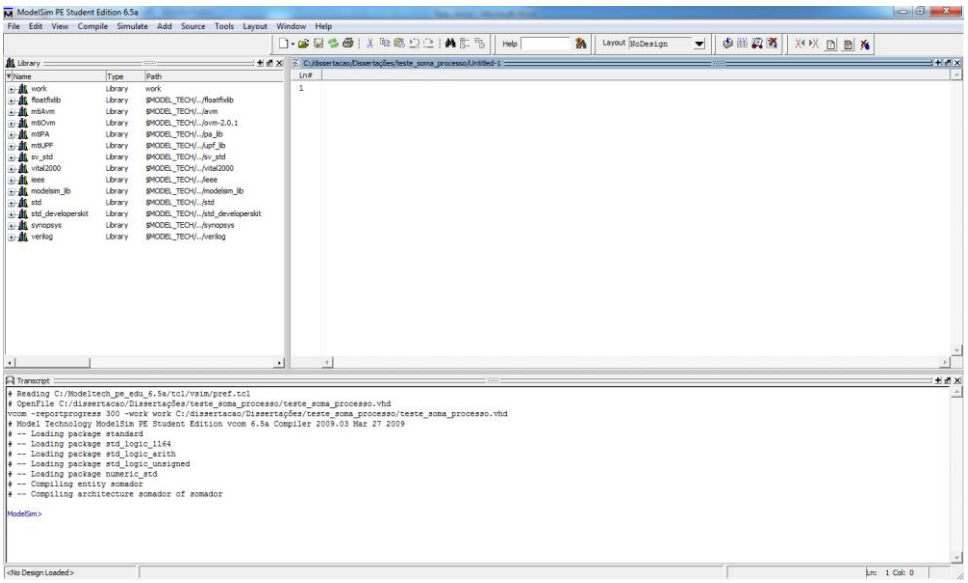


Fig. 3.22 – Imagem do ambiente gráfico do ModelSim.

Do lado esquerdo encontra-se a janela que mostra as bibliotecas ou, caso o utilizador escolha, as unidades de simulação. Fazendo duplo clique na entidade que se pretende simular, na janela bibliotecas, o simulador passa automaticamente para a janela de simulação, onde é possível escolher os sinais que se querem observar.

Na janela central o projectista irá escrever o seu código ou abrir ficheiros já existentes. No caso de uma simulação, esta janela mostra o gráfico das formas de onda dos sinais.

Na zona inferior encontra-se a janela *transcript*, que funciona como uma consola, alertando o projectista para eventos que necessitem de atenção, como erros de síntese. É também aqui que o projectista inicia processos, através da introdução de comandos.

De seguida, são descritos os passos de maneira a simular-se um circuito simples.

Passo 1 – Começa-se por se criar um novo ficheiro, clicando em botão *File*, na barra de ferramentas e escolhe-se *New* -> *Source* -> *VHDL* (ver fig. 3.23).

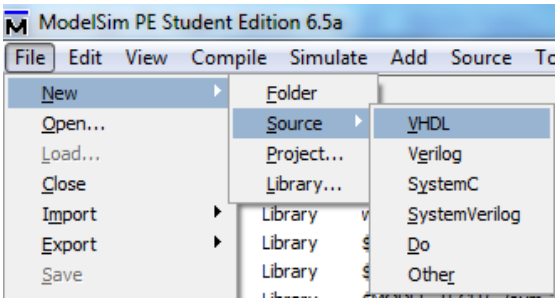


Fig. 3.23 – Escolha de uma nova *source* VHDL.

Passo 2 – Depois de se ter especificado o circuito em VHDL, é necessário compilá-lo. Em *Compile -> Compile*, selecciona-se o(s) ficheiro(s) desejado(s) e clica-se de novo em *Compile*. De seguida cria-se um ficheiro test-bench ou então um ficheiro DO, de maneira a inicializar os sinais de entrada. A simulação inicia-se clicando no ficheiro na janela *Libraries*. O ModelSim muda automaticamente para a janela de simulação, onde se seleccionam os sinais a serem visualizados (ver fig. 3.24).

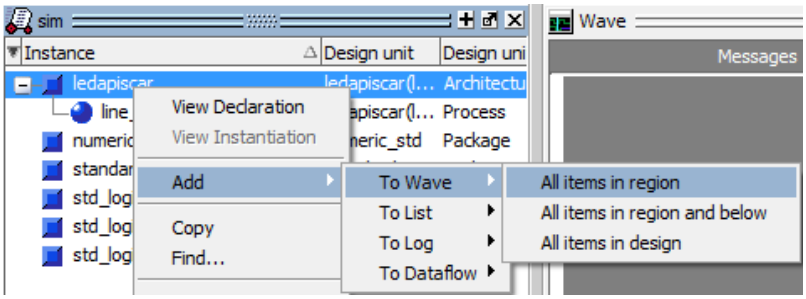


Fig. 3.24 – Escolha de sinais a mostrar no gráfico de simulação. Neste caso, serão escolhidos todos os sinais.

Passo 3 – Ao executar o ficheiro DO é possível observar a evolução dos sinais (ver fig. 3.25). Na janela propriedades de grelha, acedida clicando no botão direito na linha temporal e escolhendo *Grid & Timeline Properties*, é possível escolher a unidade temporal que mais se adequa (desde fs, passando por ns, µs até aos minutos e horas). O simulador permite mostrar os sinais desde o início da simulação até ao ponto especificado de paragem e escolher a base dos valores mostrados (binária, decimal, hexadecimal, etc.).

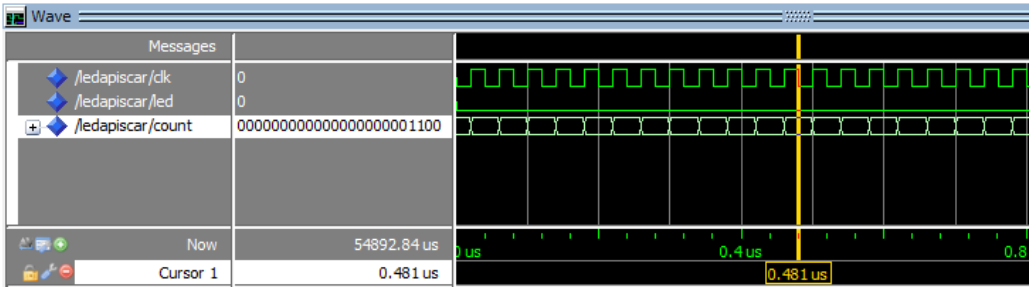


Fig. 3.25 – Janela de simulação do ModelSim.

Para criar um ficheiro DO, clica-se em *File* -> *New* -> *Source* -> *Do*. Para que o ficheiro possa ser reutilizável é aconselhável que se comece pela instrução *restart*. De seguida, atribuem-se valores aos sinais de entrada, utilizando o comando *force*, seguido do nome do sinal, do seu valor e do momento temporal em que esse valor será inicializado. Caso se deseje adicionar outros valores que o sinal venha a ter durante a simulação, basta separar o primeiro do seguinte por uma vírgula e escrever de novo o valor e tempo em que ocorre. Por fim, indica-se o tempo de simulação, com o comando *run* (ver fig. 3.26).

```
restart
force clk 0 0, 1 20 ns -r 40 ns
run 100000 ns
```

Fig. 3.26 – Exemplo de um ficheiro DO. Irá inicializar o sinal *clk* com um período de 40 ns.

Para finalizar, basta correr a simulação e verificar se os sinais estão de acordo com o que o projectista espera.

Em alternativa ao uso do ficheiro DO, pode utilizar-se um ficheiro *test-bench*, que inclui o circuito a simular e define os sinais de entrada (ver fig. 3.27).

```
Ln#
 1  library IEEE;
 2  use IEEE.STD_LOGIC_1164.ALL;
 3  use IEEE.STD_LOGIC_ARITH.ALL;
 4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
 5
 6  entity tb_LEDapiscar is
 7         port (led : buffer std_logic);
 8  end tb_LEDapiscar;
 9
10  architecture tb_LEDapiscar of tb_LEDapiscar is
11
12  signal clk         : std_logic := '0';
13
14  component LEDapiscar is
15      Port (   clk: in std_logic;
16           led : out std_logic );
17  end component;
18
19  begin
20
21  hello_world_instant : LEDapiscar
22          port map (clk, led);
23
24  process
25     begin
26     wait for 20 ns; clk  <= not clk;
27  end process;
28
29  end architecture tb_LEDapiscar;
```

Fig. 3.27 – Exemplo de uma descrição do tipo *test-bench*.

3.8 Digilent Adept

De maneira a carregar o ficheiro BIT para a FPGA é necessário um cabo de ligação USB e *drivers* para controlar o fluxo de dados. A aplicação inicialmente utilizada e posteriormente modificada, a *Digilent Adept* (ver fig. 3.28), permite carregar ficheiros de configuração, assim como escrever e ler bytes de dados para registos especificados, assumindo que na FPGA se encontra um circuito preparado para os receber e guardar. A aplicação permite ainda testar a FPGA para verificar a sua integridade.

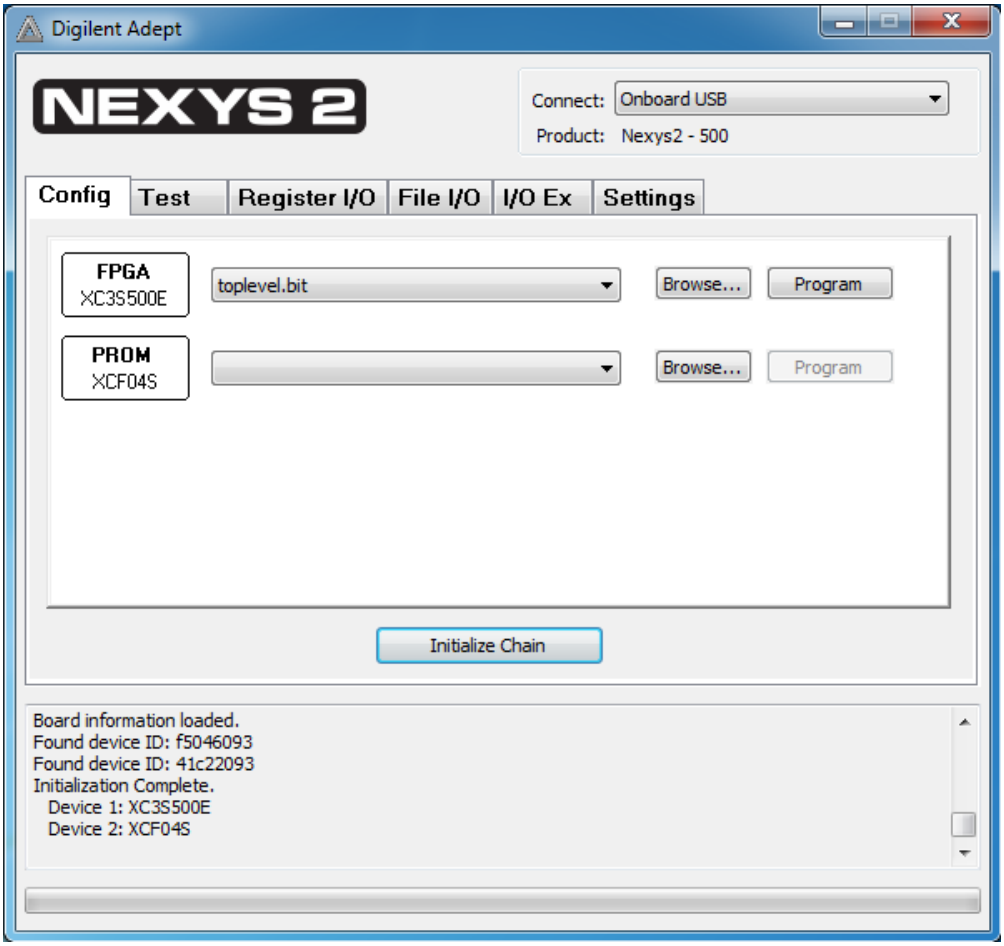


Fig. 3.28 – Aspecto gráfico do programa *Digilent Adept*, utilizado para carregar ficheiros BIT para a FPGA.

Ao inicializar o *Digilent Adept*, é necessário primeiro conectá-lo à placa. Se ela já se encontrar ligada ao computador de uso geral basta clicar em *Connect* e escolher *Onboard USB*. Na linha abaixo pode-se visualizar o nome da placa detectada (no exemplo da figura 3.28, verifica-se que aparece Nexys2 – 500). Na parte inferior da interface gráfica também surgem indicações acerca da nova conectividade, mostrando os novos dispositivos encontrados, sendo o da FPGA do exemplo da figura 3.28 o Device 1: XC3S500E. Depois de feita a inicialização basta ir a *Browse*, escolher o ficheiro BIT que se pretende carregar para a FPGA e clicar em *Program*. O circuito lógico fica assim descrito na FPGA e a sua execução é iniciada.

Embora no início se tenha utilizado a aplicação original, foi necessário modificá-la, de maneira a cumprir as especificações pretendidas (transmissão do sistema computacional de uso geral para a FPGA de 1024 valores, cada um com 16 bits). A aplicação original apenas permitia o envio de dados até 255 valores, cada um com 8 bits. A *Digilent Adept* reteve a função de carregar o ficheiro BIT para a FPGA. A aplicação alterada corre através da consola de comandos.

3.9 Conclusões

Neste capítulo foram descritas as ferramentas de apoio utilizadas durante a criação e desenvolvimento do trabalho, as etapas da criação de um ficheiro de configuração no ISE WebPack e foi apresentada a noção de IP *Core*.

Capítulo 4

Árvores Binárias

Sumário

Neste capítulo é apresentada a noção de estruturas de dados, é feita uma introdução à árvore binária e a algumas das suas características, tipos, propriedades e métodos de travessia.

4.1 Estruturas de dados

De maneira a existir um acesso mais eficaz à informação guardada em memória, não basta ter em conta apenas aspectos tecnológicos, como a velocidade de acesso, mas também a forma como os dados estão armazenados. Tal como numa biblioteca os livros estão ordenados segundo um determinado parâmetro de forma a tornar o seu acesso mais rápido (por ordem alfabética, por género, por autor, etc.), também os dados guardados em componentes electrónicos devem ter uma certa ordem ou estrutura.

Ao longo dos tempos foram desenvolvidos diferentes tipos de estruturas de dados, tais como vectores ou *arrays*, filas, pilhas, árvores, listas ligadas, etc. Certas estruturas são mais eficazes para umas aplicações que outras. Nesta tese foi utilizada a árvore binária (ver fig. 4.1).

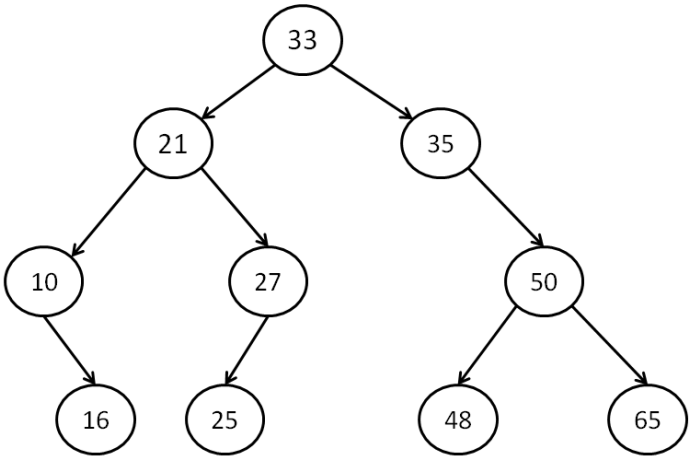


Fig. 4.1 – Exemplo de uma árvore binária.

4.2 Árvores binárias

As árvores binárias constituem a estrutura de dados ideal para implementar bases de dados, pois, uma vez ordenadas, são bastante eficazes na pesquisa, guarda e extracção de dados. É um tipo de estrutura bastante poderoso em sistemas onde a velocidade de decisão é vital. Existe por isso uma grande aplicação destas em sistemas computacionais [26].

As árvores binárias são constituídas pela raiz (o único nó que não possui nó pai) e nós filhos. Cada nó pode conter no máximo dois nós filhos e apenas um nó pai [27] [28]. Os nós terminais de uma árvore (os que não possuem nós filhos) chamam-se nós folhas.

Ao introduzir um valor numa árvore vazia, esse valor torna-se na raiz. Os valores seguintes a inserir vão-se tornar no nó filho esquerdo ou direito, consoante sejam menores ou maiores que a raiz, respectivamente. Caso a raiz já possua um nó filho no lugar que se pretende ocupar, o valor a inserir é comparado com esse nó filho. Caso esse nó também possua um nó filho no espaço que se pretende ocupar, o processo de investigação continua até que se encontre uma posição desocupada, sendo aí inserido. Estes passos, representados na figura seguinte, são repetidos até se formar a árvore.

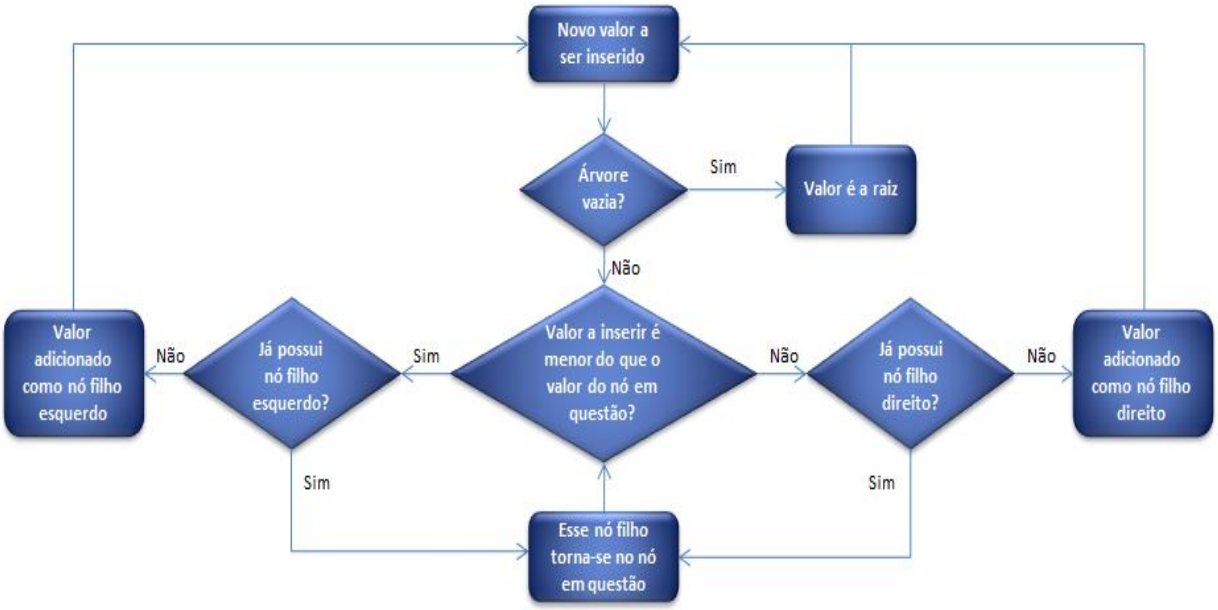


Fig. 4.2 – Diagrama da formação de uma árvore binária.

Duas das noções relativas aos nós e à árvore em si são a altura e a profundidade [29].

- Altura do nó – É a distância de um determinado nó ao nó folha do seu ramo.
- Altura da árvore – É a altura da raiz da árvore.

- Profundidade do nó – É a distância de um determinado nó até à raiz.

- Profundidade da árvore – É a profundidade máxima dos seus nós.

Por exemplo, a árvore da figura 4.1 possui altura e profundidade 3. A altura do nó com valor 27 é 1 e a sua profundidade é 2.

4.3 Tipos de árvores binárias

Dependendo da forma como os nós da árvore se encontram ocupados, podem-se distinguir vários tipos de árvores binárias. Estas, por sua vez, podem possuir propriedades que ajudam a calcular o número de nós da árvore:

- Árvore estritamente binária – É uma árvore binária cujos nós possuem dois ou nenhum nó filho. Não existem nós com apenas um filho (ver fig. 4.3).

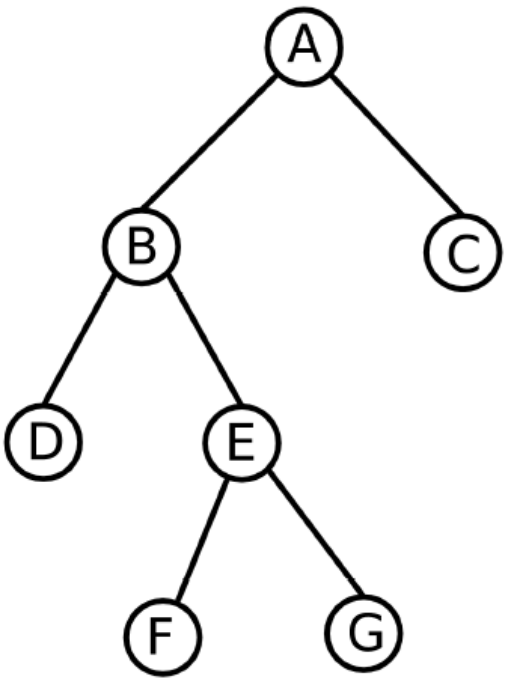


Fig. 4.3 – Árvore estritamente binária. [30]

- Árvore completa – É uma árvore estritamente binária em que os nós folha de cada ramo se encontram no último ou no penúltimo nível (altura 0 ou 1) (ver fig. 4.4).

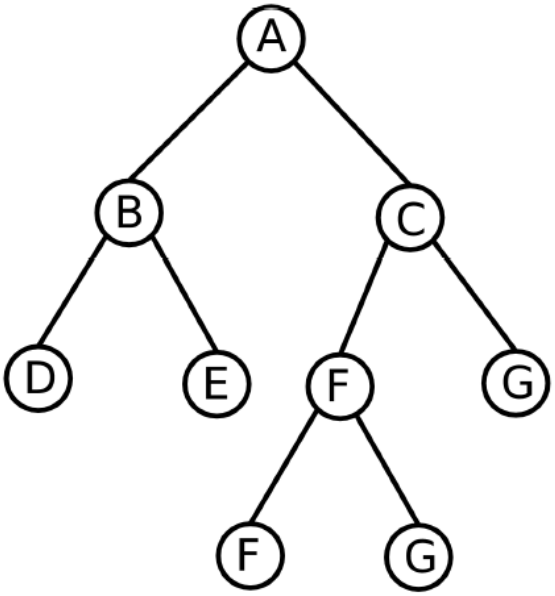


Fig. 4.4 – Árvore binária completa. [30]

- Árvore cheia – É uma árvore estritamente binária onde os nós folha estão todos à mesma profundidade (altura 0) (ver fig. 4.5).

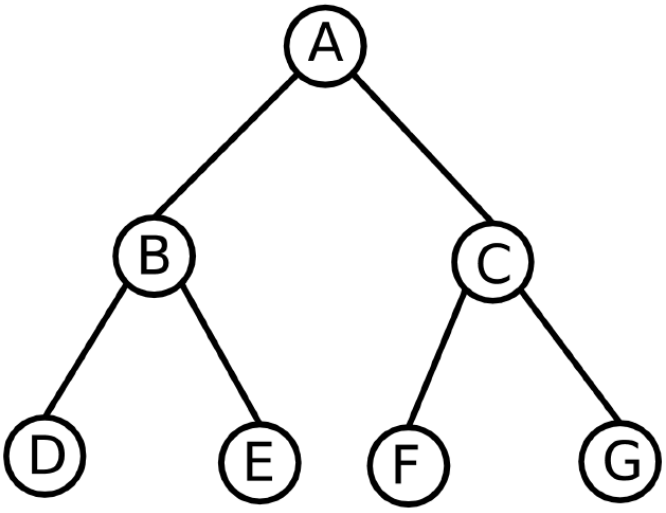


Fig. 4.5 – Árvore estritamente binária, cheia e completa. [30]

4.4 Algumas propriedades de árvores binárias

As árvores binárias cheias são o tipo de árvore em que é mais fácil calcular o número de nós que as compõem. Isto acontece porque ao subir de um nível para o outro o número de nós duplica (raiz ou nível 0 – 1 nó, nível 1 – 2 nós, nível 2 – 4 nós...).

Assim, numa árvore cheia, o número de nós em cada nível é dado por 2^n, sendo *n* o nível (ver fig. 4.6). O número de nós que compõem a árvore é dado por $2^{m+1} - 1$, sendo *m* a profundidade da árvore.

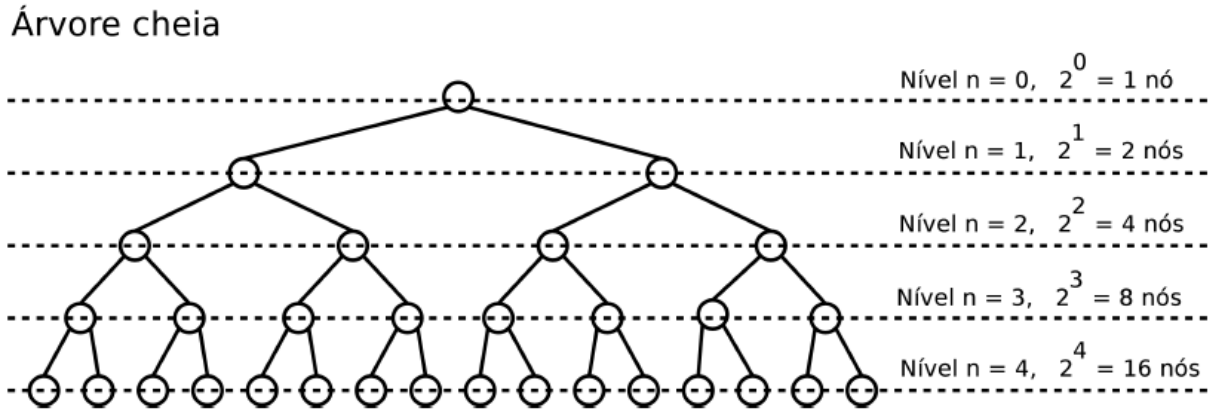


Fig. 4.6 – Árvore binária do tipo cheia. [30]

Numa árvore binária completa a determinação do número de nós é semelhante à árvore binária cheia, excepto no último nível. Uma árvore pode ser cheia e completa. Para todos os níveis excepto o último, o número de nós em cada nível é dado por 2^n, sendo *n* o nível. No último nível, o mínimo de nós que é possível ter são dois e o máximo são 2^p, sendo *p* o último nível (ou a profundidade da árvore). Assim, numa árvore binária completa o número de nós varia entre $2^{k+1} + 1$ (sendo *k* o penúltimo nível) e $2^{p+1} - 1$ (ver fig. 4.7).

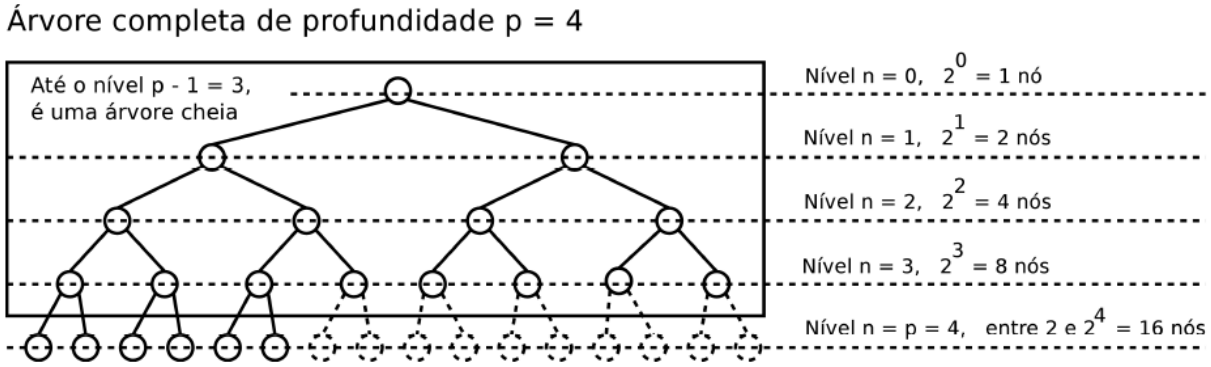


Fig. 4.7 – Árvore binária do tipo completa. [30]

4.5 Travessia da árvore binária

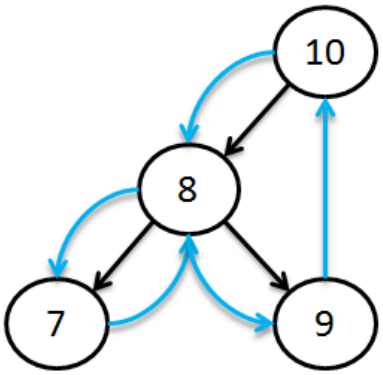
A forma como se percorrem os nós na ordenação de uma árvore binária é semelhante à inserção de nós na árvore, com pequenas diferenças: para inserir cada

nó basta apenas ir percorrendo a árvore de maneira a chegar à posição certa onde o nó deve ser colocado. A travessia é um pouco mais complicada, pois é necessário anotar os nós à medida que se percorre a árvore. Para esse fim utiliza-se uma *stack*, que guarda o endereço de cada nó percorrido. Caso se encontre um nó folha e se pretenda subir na árvore basta ir buscar à *stack* o endereço do último nó guardado. O uso da *stack* no percurso de uma árvore binária é exemplificado na secção 6.2.1.

Do ponto de vista de *software*, a travessia pode ser realizada de maneira iterativa ou recursiva. A maneira iterativa descreve todas as iterações que poderão existir, tornando a sua programação mais extensa e complexa. Usando a maneira recursiva, o programa irá ser mais pequeno e de mais fácil leitura do que a abordagem iterativa. A utilização da forma recursiva é a preferida ao processar árvores binárias em C. Existem numerosos exemplos de travessias de árvores binárias usando a forma recursiva, sendo mesmo frequente dar o exemplo desta estrutura de dados para explicar a abordagem da recursividade (usando a linguagem C).

A forma de percorrer os nós da árvore binária durante a ordenação também pode variar. A forma mais habitual (e a que foi utilizada neste trabalho) é a forma *inorder* [31]. Nesta forma, começa-se por percorrer o ramo esquerdo, tentando primeiro encontrar o nó mais à esquerda. Quando este for encontrado, esse valor é guardado na lista de valores ordenados e verifica-se se possui algum filho direito. Se isso acontecer, volta-se a procurar o valor que está mais à esquerda nesse ramo. Caso contrário, escreve-se na lista de valores ordenados o seu nó pai e verifica-se se este possui algum nó direito. Em caso afirmativo, procura-se mais uma vez o valor mais à esquerda nesse ramo. O processo repete-se até se chegar à raiz. O seu valor guarda-se na lista de valores ordenados e, de seguida, continua-se a percorrer o ramo direito da mesma forma como foi percorrido o ramo esquerdo.

De maneira a exemplificar este tipo de travessia considere-se uma árvore em que foram inseridos os seguintes valores: 10, 8, 7, 9. No percurso *inorder* tenta-se primeiro encontrar o nó mais à esquerda, que neste caso é o 7. O valor desse nó é guardado na lista de valores ordenados. Como este nó não possui filho direito, guarda-se o valor do seu nó pai, que possui valor 8. Seguidamente investiga-se a existência de nó filho direito. Neste exemplo esse nó filho existe (nó de valor 9), logo o seu valor é guardado. Como esse nó não possui quaisquer nós filhos, sobe-se na árvore. O valor do nó em questão, neste caso o nó raiz, é guardado na lista. Como todos os nós foram processados, o percurso acaba neste ponto (ver fig. 4.8).



Lista de valores ordenados
7
8
9
10

Fig. 4.8 – Exemplo de um percurso *inorder*.

Esta travessia, do tipo Esquerda-Raiz-Direita, encontra-se representada em diagrama na figura 4.9.

Fig. 4.9 – Diagrama da ordenação de uma árvore binária, utilizada nesta tese (*inorder*).

Outras formas de travessia, menos utilizadas, são a *preorder* (Raiz-Esquerda-Direita), *postorder* (Esquerda-Direita-Raiz) e *level-order*. Nesta última todos os nós de cada nível são visitados antes de se descer de nível.

4.6 Aplicação da estrutura de dados árvore binária em vários projectos

A estrutura de dados árvore binária é usada em diversas aplicações de maneira a melhorar o desempenho do sistema em que se insere. Existem vários artigos publicados, em que foi utilizada a FPGA, que referem a implementação deste tipo de estrutura de dados de forma a tornar o sistema mais rápido. Seguidamente apresentam-se alguns desses trabalhos, assim como qual o papel que a árvore binária teve neles:

- Scalable High-Throughput SRAM-Based Architecture for IP-Lookup Using FPGA, *por Hoang Le, Weirong Jiang e Viktor K. Prasanna* [32] – Este artigo procura demonstrar uma forma rápida de encontrar endereços IP guardados numa RAM (*IP Lookup*), usando a FPGA. A árvore binária utilizada aqui serviu obviamente para diminuir o tempo que é necessário para extrair o IP pretendido, numa tabela de endereços com 228K prefixos.

- Compact Binary Tree Representation of Logic Function with Enhanced Throughput, *por Padmanabhan Balasubramanian e Cemal Ardil* [33] – Neste artigo, a estrutura em árvore binária foi utilizada para representar funções lógicas, de maneira a obter-se um maior *throughput*. De forma a validar esta experiência, foi utilizada a FPGA. Na árvore binária, cada nó simboliza uma função lógica (AND, OR, etc.).

- FPGA Implementation of High Speed Infrared Image Enhancement, *por M. Chandrashekar, U. Naresh Kumar, K. Sudershan Reddy e K. Nagabhushan Raju* [34] **–** Neste artigo, que trata da implementação em FPGA do realce de imagens infravermelhas (IRI) a alta velocidade, é utilizada uma árvore binária para descrever as funções SMQT (*Successive Mean Quantization Transform*), onde os seus nós representam MQUs (*Mean Quantization Units*).

- Embedded Computation of Maximum-Likelihood Phylogeny Inference Using Platform FPGA, *por Terrence S. T. Mak e K. P. Lam* [35] – É utilizada uma árvore binária alterada, onde a raiz possui três nós em vez de dois. Neste artigo a árvore binária é utilizada como uma maneira rápida e eficaz de aceder aos dados guardados.

- VLSI Design of an Anti-Collision Protocol for RFID Tags, *por S. M. A. Motakabber, Mohd Alauddin Mohd Ali e Nowshad Amin* [36] – Neste artigo são investigados a *Radio Frequency Identification* (RFID) e algoritmos que diminuem o mais possível a probabilidade de colisões. O algoritmo pesquisado de maneira a atingir esse fim foi através de uma árvore binária. O sistema foi testado em FPGA.

- An Fpga Implementation of a Microprogrammable Controller to Perform Lossless Data Compression Based on the Huffman Algorithm, *por Tiago Maritan Ugulino de Araújo, Eduardo Ribas Pinto, José Antônio Gomes de Lima e Leonardo Vidal Batista* [37] – Neste artigo é investigado o algoritmo de Huffman, de forma a realizar compressão de dados sem perdas, usando para isso um microcontrolador implementado em FPGA. São utilizadas árvores binárias em conjunto com o algoritmo de Huffman de maneira a cumprir esse objectivo.

- Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing, *por Herbert Walder, Christoph Steiger e Marco Platzner* [38] – Este artigo investiga a partição da memória livre. Considera-se que ao fazer a partição de um pedaço de memória irá dar origem a dois pedaços de memória. Assim, é utilizada uma árvore binária para representar as novas partições. A estrutura de árvore binária pode também ser utilizada para

encontrar rapidamente a partição de memória com o tamanho ideal. Foi utilizada a FPGA para testar os resultados.

- Performance Optimization of an FPGA-Based Configurable Multiprocessor for Matrix Operations*, por Xiaofang Wang e Sotirios G. Ziavras* [39] – A árvore binária é utilizada neste artigo para optimizar o desempenho de um multiprocessador configurável que realiza operações sobre matrizes em FPGA.

4.7 Conclusões

Neste capítulo foi descrita a estrutura de dados árvore binária, os seus tipos, propriedades e formas de travessia. Foram também apresentados alguns exemplos da sua aplicação em artigos publicados.

Capítulo 5

Estrutura do Projecto e Ligação Computador-FPGA

Sumário

Neste capítulo explica-se o propósito dos ficheiros incluídos neste projecto, de maneira a que um valor a ser recebido na FPGA através da ligação USB seja guardado, ordenado, convertido em BCD e finalmente impresso no monitor. Por fim, descreve-se como foi implementada a ligação Computador-FPGA através de USB.

5.1 Projecto no ISE WebPack

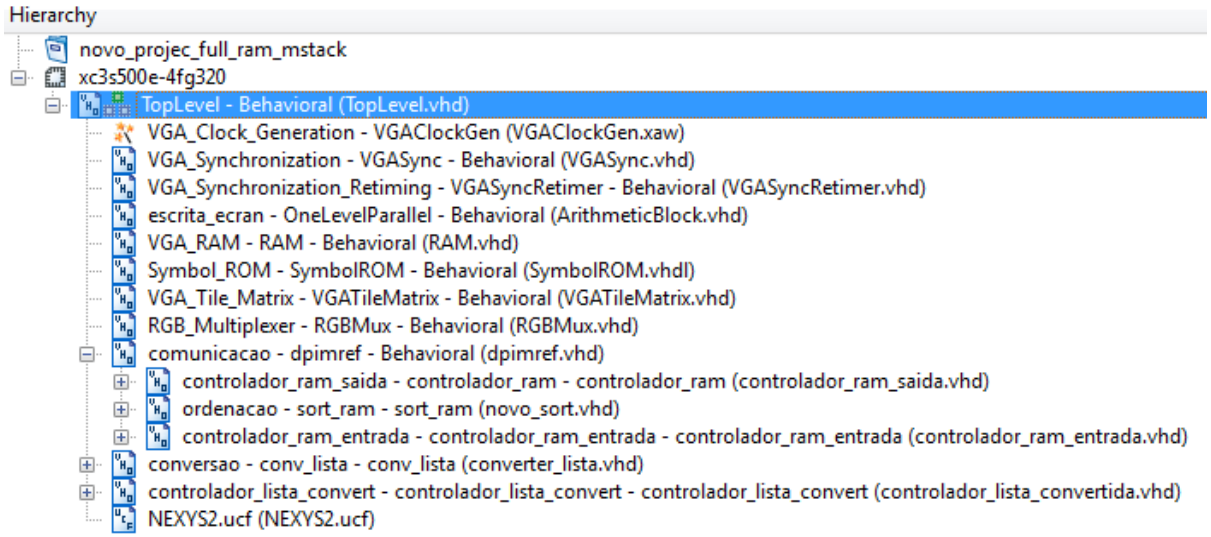


Fig. 5.1 – Ficheiros utilizados no projecto.

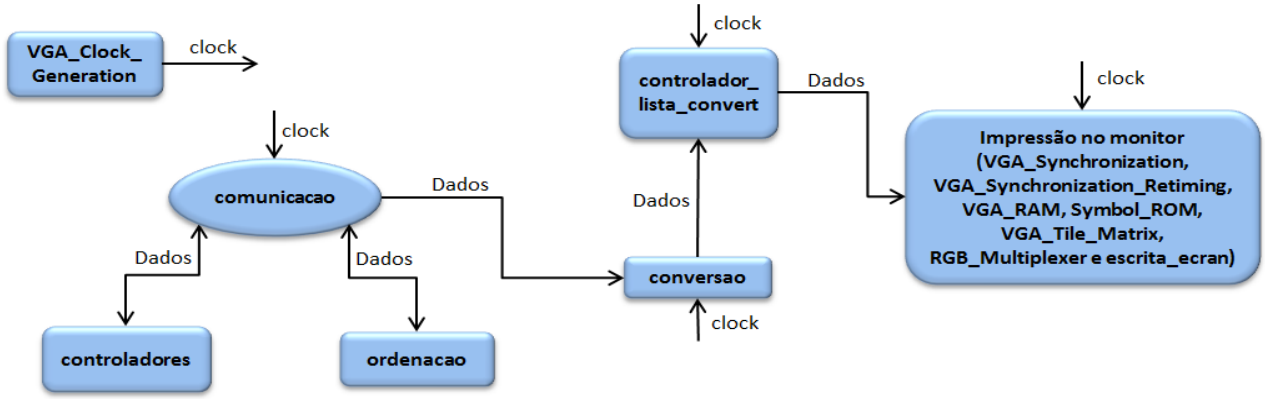


Fig. 5.2 – Diagrama de blocos utilizados no projecto.

O projecto montado em ISE WebPack é composto por vários ficheiros VHDL e IP *cores*, com vários níveis de hierarquia (ver fig. 5.1 e 5.2). De seguida encontram-se as descrições do funcionamento de cada um destes blocos:

- TopLevel – Aqui estão incluídas as instanciações de circuitos de um nível de hierarquia mais baixo. Os portos de entrada e de saída da FPGA, que comunicam com outros componentes da placa, como por exemplo sinais de USB, LEDs, *Switches*, etc., são descritos aqui. São também definidos sinais que vão ser necessários nas interligações dos circuitos descritos. Foram incluídos os seguintes blocos: VGA_Clock_Generation, escrita_ecran, VGA_RAM, VGA_Synchronization, VGA_Synchronization_Retiming, Symbol_ROM, VGA_Tile_Matrix, RGB_Multiplexer, comunicacao, conversao e controlador_lista_convert.

- VGA_Clock_Generation – É um IP *core* do tipo *Single DCM_SP*. Foi utilizado para ajudar a estabilizar a frequência de trabalho. A frequência de relógio foi estabelecida em 50 MHz, que é a mesma da placa com FPGA (ver fig. 5.3).

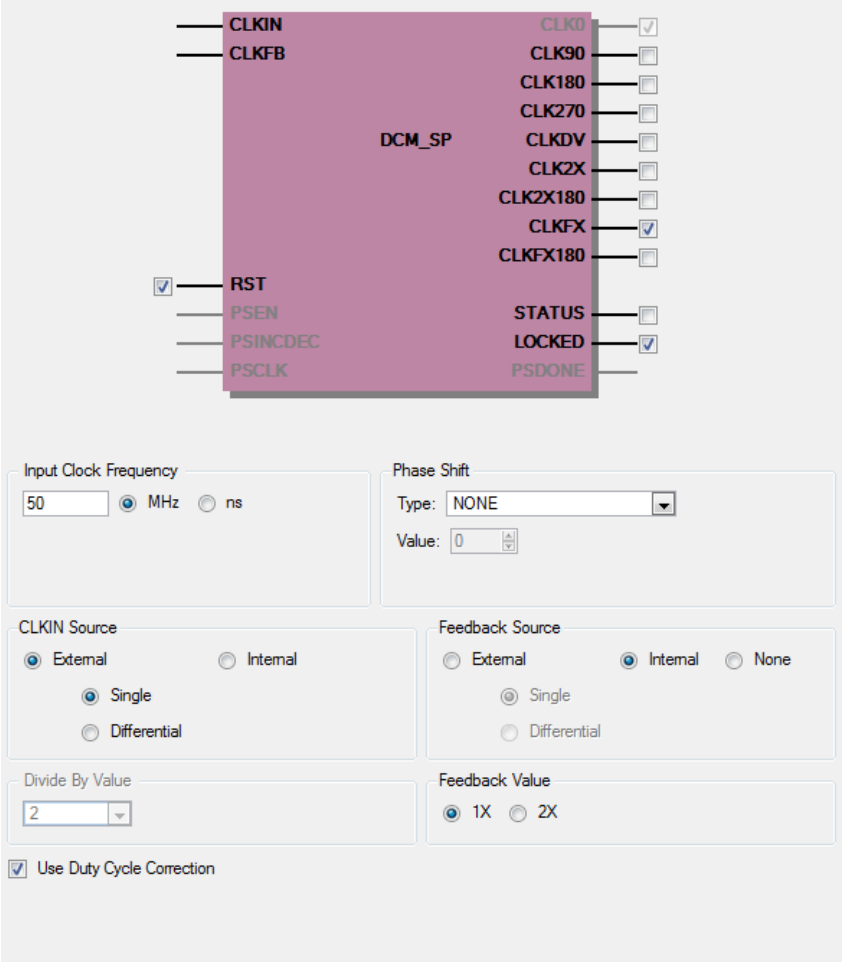


Fig. 5.3 – Imagem da primeira janela de diálogo do IP *Core Clock Generator*.

De seguida são descritos alguns dos portos utilizados pelo DCM_SP:

- CLKIN – É o porto que recebe o sinal de entrada. Escolheu-se como sinal de entrada o relógio da placa com FPGA, com a frequência de 50 MHz.

- RST - É o porto que, uma vez activado, reinicia o funcionamento do bloco.

- CLKFX – Este porto é um sinal de saída do bloco. Contém o sinal de relógio digital com a frequência especificada no bloco DCM. A frequência foi definida em 50 MHz, a mesma da frequência de entrada.

- LOCKED – Este sinal de saída é opcional. É activado quando o DCM atinge o estado LOCK, depois de amostrar milhares de ciclos de relógio. Até que o sinal LOCKED esteja activo, as saídas de relógio do DCM (como o CLKFX) podem conter *gliches* ou outros efeitos indesejados.

❖ VGA_Synchronization, VGA_Synchronization_Retiming, VGA_RAM, Symbol_ROM, VGA_Tile_Matrix e RGB_Multiplexer – São blocos que, em conjunto, mapeiam os pixéis do monitor, divididos em linhas e colunas, preparando o uso de caracteres ASCII a escrever no monitor. Estes blocos foram descritos por um antigo aluno da Universidade de Aveiro, sendo reutilizados neste projecto.

❖ escrita_ecran – Nesta descrição, os valores vão ser lidos da lista ordenada em código BCD e impressos no monitor. O valor do tempo necessário para ordenar é também impresso.

❖ comunicacao – Descrição de *hardware* baseada no dpimref fornecido pela Digilent Inc., de maneira a gerir a recepção e envio de dados entre o computador de uso geral e a FPGA através da ligação USB. Incluíram-se os controladores de vários BRAMs, assim como a descrição do circuito cuja função é a ordenação de dados.

- Controladores – Ajudam a regular a leitura e escrita de dados das listas implementadas em BRAM. Nestes controladores tomam-se medidas para que não haja duas ou mais tentativas de escrita ou de leitura ao mesmo tempo.

- ordenacao – Descrição do bloco que vai ler, um a um, os valores gerados aleatoriamente no sistema computacional de uso geral, que foram transferidos e guardados no BRAM, de maneira a ordená-los através do algoritmo da árvore binária, explicado anteriormente. Foram implementadas em BRAM duas listas intermédias, de forma a construir a árvore binária e a realizar posteriormente o seu percurso.
  No final obtém-se o número de ciclos de relógio necessários para a construção da árvore e ordenação de dados. Através desse valor será possível calcular o tempo despendido na ordenação.

❖ conversao – Possui a responsabilidade de converter a lista de valores ordenados (ainda em código binário) para código BCD. À medida que os números são convertidos guardam-se numa nova lista. De forma a executar a conversão é utilizado um bloco denominado bintoBCD.
É também nesta descrição que é calculado e convertido para BCD o tempo demorado na execução do bloco de ordenação.

❖ controlador_lista_convert – É o controlador que regula as entradas e saídas de dados da lista convertida, em código BCD. Poderia ter sido incluído no bloco conversao, mas como a lista de valores ordenados em BCD também teria de estar disponível no bloco escrita_ecran, foi decidido incluir-se no TopLevel, de maneira a facilitar o seu acesso a ambas as descrições.

❖ NEXYS2.ucf – O ficheiro UCF especifica as restrições que vão ser utilizadas.

5.2 Ligação Computador-Placa por USB

Parte do objectivo da tese foi o desenvolvimento de *software* que gerisse a ligação entre o sistema computacional de uso geral e a FPGA. Como se pretende enviar 1024 números para a FPGA, cada um com 16 bits, alterou-se o programa DPCUTIL, escrito em código C, fornecido pela Digilent®, de maneira a cumprir este requisito. A transmissão de um número consiste em enviar dois valores, os 8 bits dos dados e do registo mais significativos e de seguida os 8 bits menos significativos. A figura 5.4 exemplifica a partição dos valores de endereço e de dados.

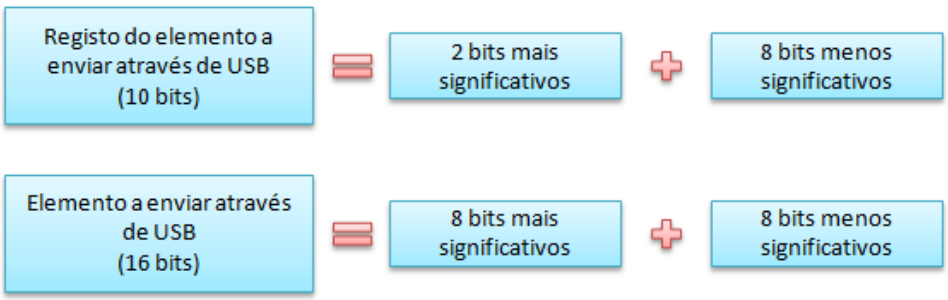


Fig. 5.4 – Partição de registos de endereço e de dados.

A lista é gerada através da função *rand* (ver fig. 5.5). Esta função gera um número aleatório, sendo executada 1024 vezes. Uma vez que a sua semente é alterada sempre que o programa é executado, listas geradas em diferentes execuções do programa vão também ser diferentes. De seguida a lista gerada é transmitida para a FPGA. É aproveitada a função *DoPutReg*, que envia o endereço na variável *szFirstParam* e o valor dos dados em *szSecondParam*. Quando o último valor da lista for enviado, esta é ordenada no computador de uso geral.

```
/* De maneira a tornar a lista mais aleatória, modifica-se
   a seed com usando o valor do relógio */
unsigned int iseed = (unsigned int)time(NULL);
srand (iseed);

/* Gera-se a lista aleatória */

printf("lista = \n");

for (i=0; i<n; i++)
{
  /* os números criados são limitados a valores entre 0 e 65535,
  que é o máximo permitido: */
  numero_aleatorio = rand()/(((double)RAND_MAX + 1) / 65536);
  lista[i]=numero_aleatorio;
  printf("\t%d", lista[i]);
}

printf("\n\n");


  write_conf = FALSE; // desactiva as notificações de envio bem sucedidas

/* parte em que são enviados os dados da lista aleatória para a FPGA */

  for (i=0; i<n2; i++){
  aux_reg = i & mask2;
  aux_reg = aux_reg>>8;

  itoa(aux_reg, szFirstParam, 10); //// registo, 8 bits + significativos

  aux = lista[i] & mask2;
  aux = aux>>8;

  itoa(aux, szSecondParam, 10);       /// dados, 8 bits + significativos
  DoPutReg();

  aux_reg = i & mask1;
  itoa(aux_reg, szFirstParam, 10); //// registo, 8 bits - significativos

  aux = lista[i] & mask1;
  itoa(aux, szSecondParam, 10);       // dados, 8 bits - significativos
  DoPutReg();
  }

  write_conf = TRUE;


/* função de ordenação da lista e sua impressão na janela de comandos */
  ordenar_soft () ;
```

Fig. 5.5 – Parte do código criado em *software*.

Na FPGA, os primeiros valores de endereço e de dados recebidos são guardados em sinais *buffer*. Quando o segundo conjunto de valores é recebido, são devidamente concatenados. Uma vez possuindo o valor do registo e dos seus dados, poderão ser guardados na BRAM.

O programa alterado corre apenas através da consola de comandos e não num ambiente gráfico, como o original corria.

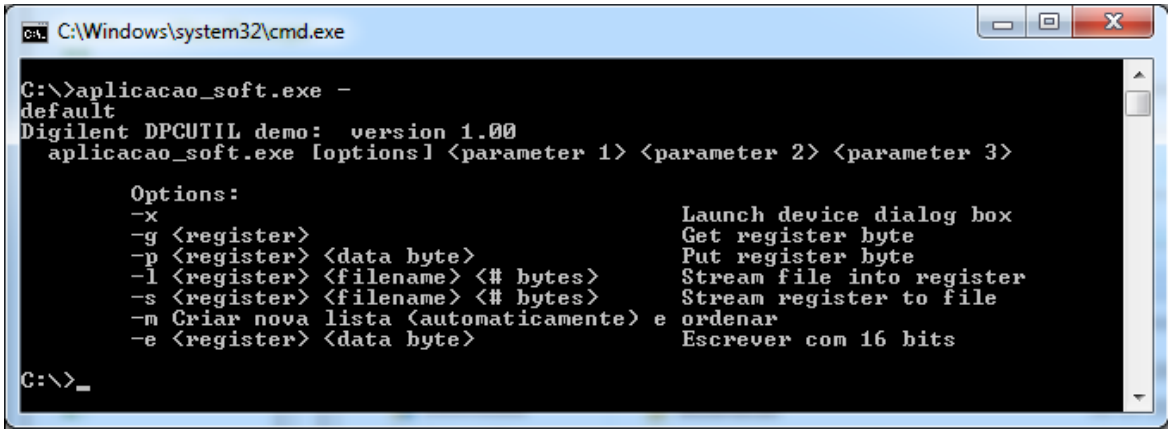


Fig. 5.6 – Programa alterado, baseado no Digilent DPCUTIL.

As instruções que foram adicionadas ao programa original foram as últimas duas opções que aparecem na lista, sendo a mais importante a opção “Criar nova lista e ordenar”.

Ao ser executado pela primeira vez, é necessário fazer a conexão com a placa (ver fig. 5.7). Para isso corre-se a aplicação com a opção –x. Ex: aplicacao_soft.exe –x.

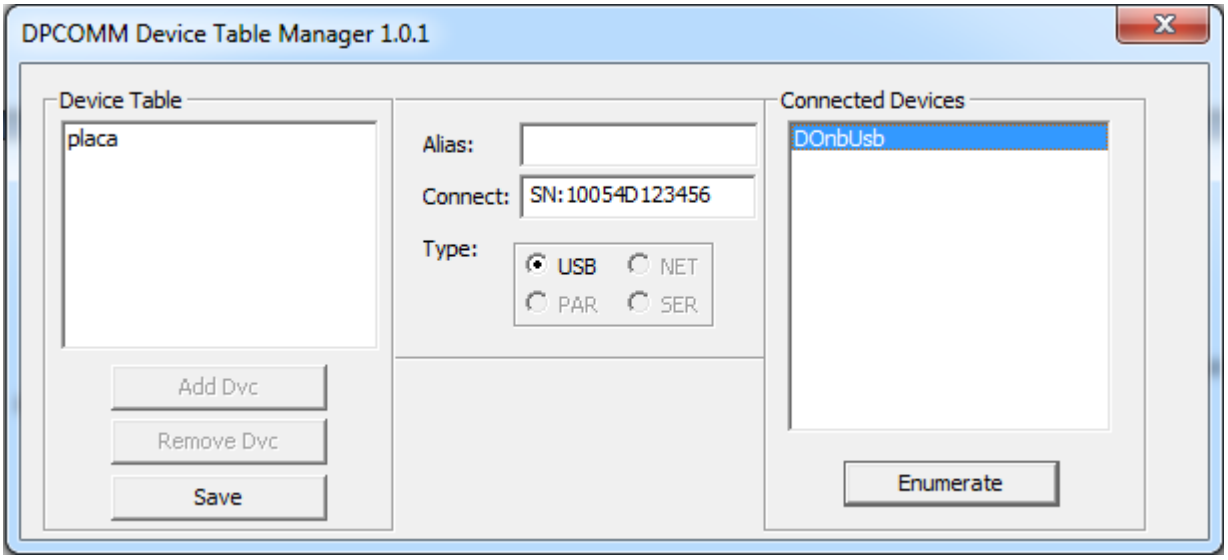


Fig. 5.7 – Janela que permite configurar a ligação com a FPGA.

A imagem seguinte mostra os dois sistemas utilizados neste trabalho, o sistema computacional de uso geral e a FPGA, ligados por cabo USB.

Fig. 5.8 – Ligação do computador pessoal com uma placa FPGA, através de USB.

As sequências de tarefas realizadas nos dois sistemas são representadas no seguinte diagrama:

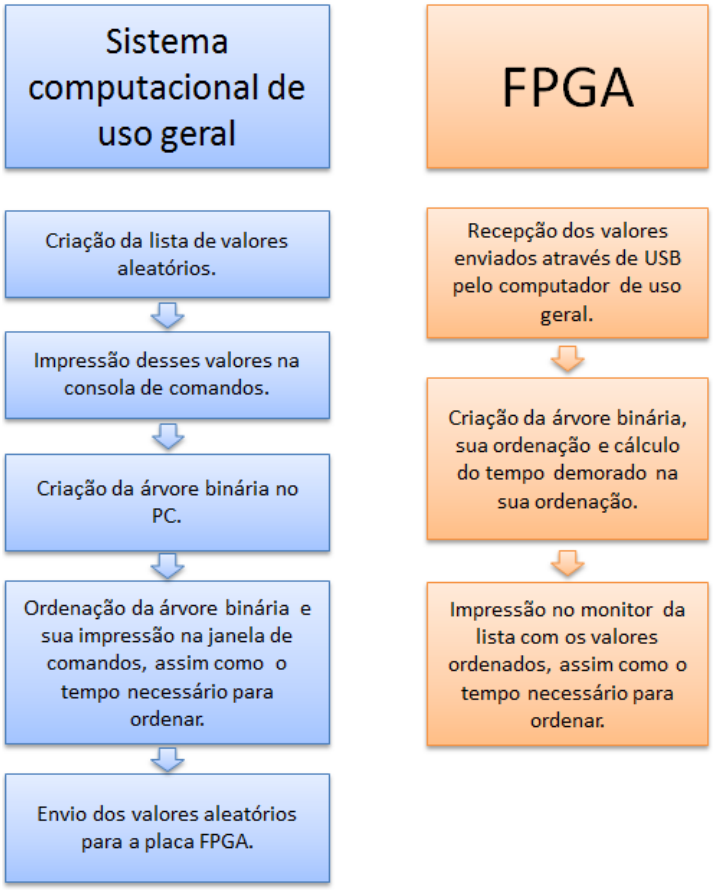


Fig. 5.9 – Funções executadas, tanto no sistema computacional de uso geral como na FPGA.

5.3 Sinais de regulação do fluxo de dados

Para que haja regulação do tráfego de dados entre o computador de uso geral e a FPGA, são usados sinais que permitem definir que tipo de operação se trata, sempre que exista uma troca de dados entre os dois sistemas. O sinal WRITE indica um processo de escrita caso possua valor lógico '0' ou um processo de leitura caso possua valor lógico '1'. O sinal ASTB indica que os dados no *data bus* (barramento de dados) são relativos a um endereço caso possua valor lógico '0'. Com o DSTB a '0', a informação no *data bus* é relativa a dados. Os sinais ASTB e DSTB nunca irão possuir valor lógico '0' simultaneamente. Para sinalizar o final da operação, usa-se o sinal WAIT, que é o único sinal de controlo modificado pelo periférico. Antes de se iniciar qualquer transferência de dados, o sinal WAIT tem de se encontrar a '0'. No início de uma operação, os sinais ASTB e DSTB vão manter-se estáveis até que o sinal WAIT passe a ter valor '1', sinalizando final de transferência. O *host* força assim os sinais ASTB e DSTB a ficarem inactivos (valor '1') e o periférico (FPGA) desactiva o sinal WAIT ('0'), estando neste momento pronto para outro ciclo de transferências. As figuras 5.9 – 5.12 exemplificam os modos possíveis.

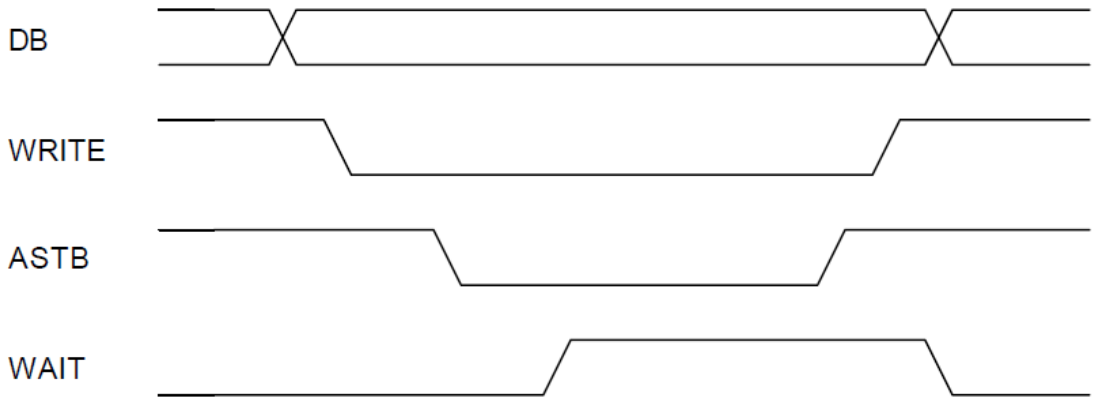


Fig. 5.10 – Address Write. Trata-se de uma operação de escrita, sendo que o *data bus* contém o endereço. [40]

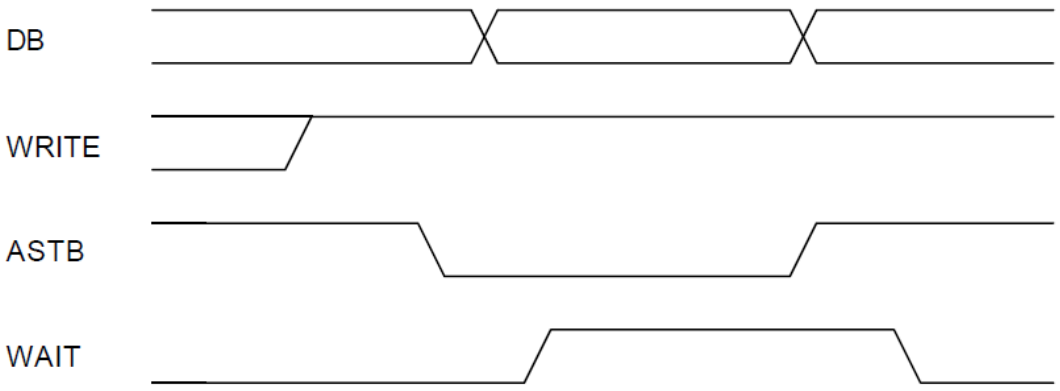


Fig. 5.11 – Address Read. Trata-se de uma operação de leitura, sendo que o *data bus* contém o endereço. [40]

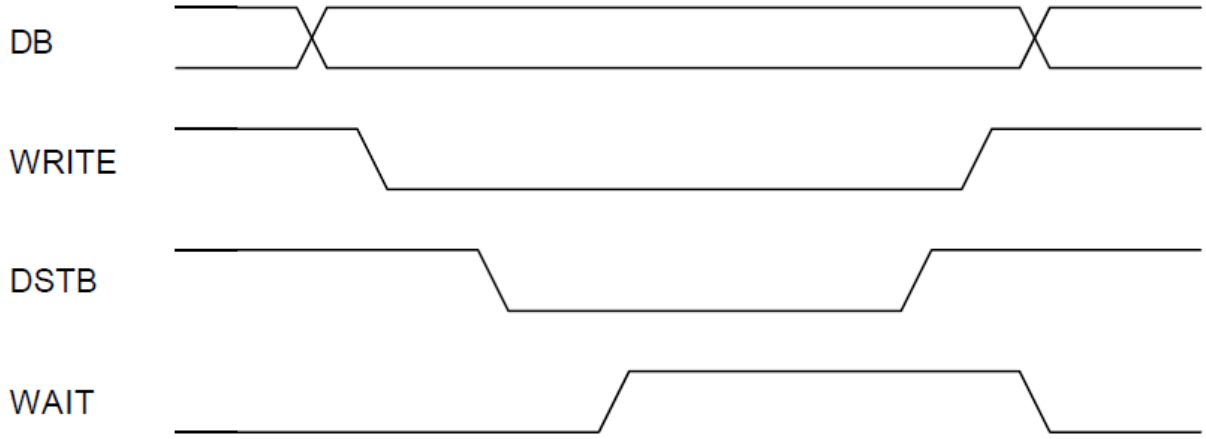


Fig. 5.12 – Data Write. Trata-se de uma operação de escrita, sendo que o *data bus* contém os dados. [40]

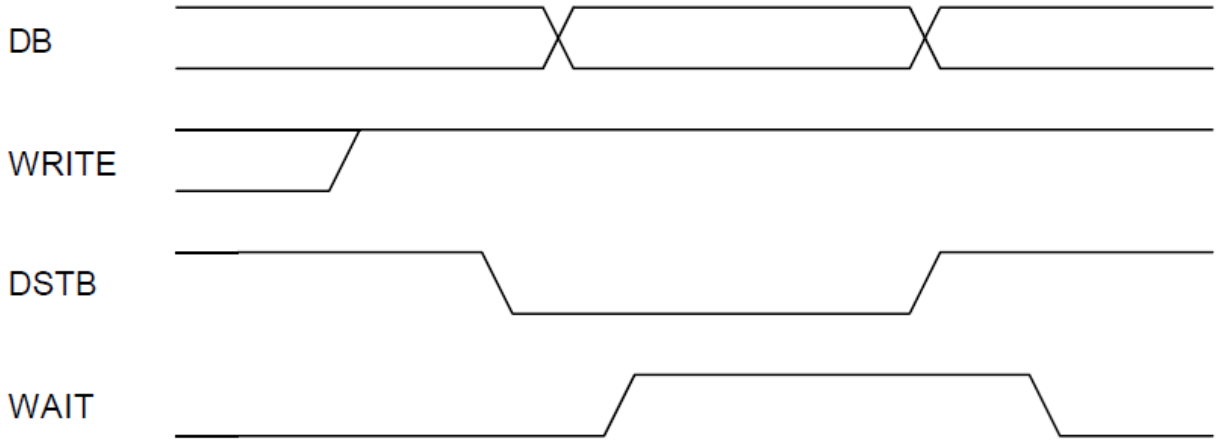


Fig. 5.13 – Data Read - Trata-se de uma operação de leitura, sendo que o *data bus* contém os dados. [40]

5.4 Conclusões

Neste capítulo foram apresentados os blocos criados durante a realização do trabalho, a aplicação desenvolvida para a transferência de dados entre o computador de uso geral e a FPGA. Foram também descritos os sinais de controlo utilizados numa transferência de dados entre os dois sistemas.

Capítulo 6

Máquinas de Estados Finitos

Sumário

Neste capítulo é introduzida a noção de máquinas de estados finitos. São ainda descritas as máquinas de estados finitos utilizadas neste projecto, incluindo a que descreve a construção da árvore binária, a conversão da lista em código binário para código BCD e a impressão de caracteres no monitor.

6.1 Máquinas de estados finitos

Uma máquina de estados finitos (FSM – *Finite-State Machine*) [41] é um modelo comportamental composto por um número finito de estados, transições e acções. Um estado é determinado por ocorrências passadas, reflectindo as mudanças desde o início do sistema até ao momento presente. Uma vez que a FSM contém informação sobre o estado corrente bem como sobre o estado para o qual irá saltar na próxima transição do relógio, esta estrutura possui uma componente de memória. A transição entre estados é feita de acordo com uma função de transição e as saídas são governadas por uma função de saída. Numa FSM as transições de estado dependem do estado actual e de entradas externas. Uma acção é a descrição de uma actividade que deve ser realizada num determinado momento.

Um exemplo clássico para demonstrar a utilidade e funcionamento das máquinas de estado é o caso do loquete. Sendo o estado inicial o de Trancado, a FSM irá mudar de estado para Aberto, caso se rode a chave. Caso não se receba essa entrada, o estado não se altera. No estado Aberto, caso se receba como entrada o fecho do loquete, o estado muda de novo para Trancado. A figura 6.1 exemplifica:

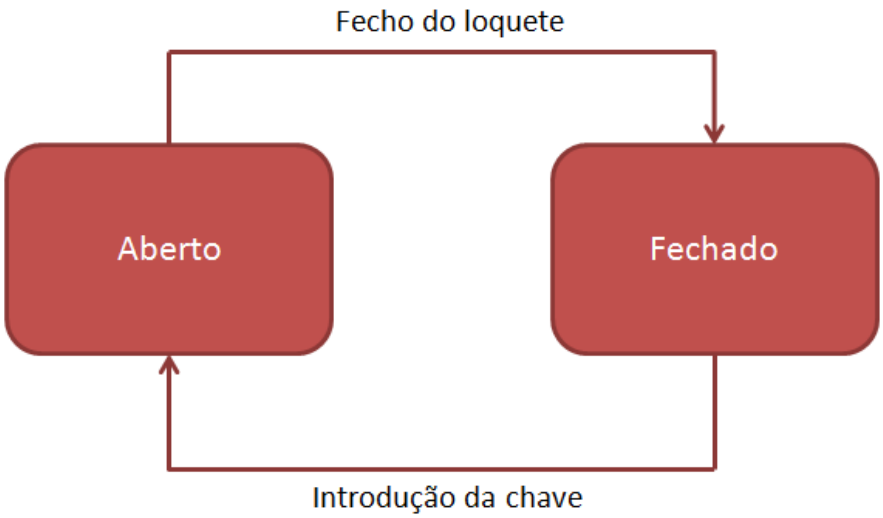


Fig. 6.1 – Exemplo de uma máquina de estados.

Cada estado tem o tempo de duração de um período de relógio, assumindo que existe transição desse estado. Assim, o projectista deve ter o cuidado de construir uma FSM que, na sua execução, irá percorrer um menor número de estados, de maneira a descrever o algoritmo pretendido.

Podem-se distinguir dois tipos de FSM: Máquina de Moore (ver fig. 6.2) e Máquina de Mealy [42] (ver fig. 6.3). A diferença entre as duas reside na forma como a saída do estado é obtida. Na Máquina de Moore, a saída depende apenas do estado em que se encontra. Na Máquina de Mealy, a saída depende do estado corrente assim como da entrada. Geralmente usam-se máquinas mistas.

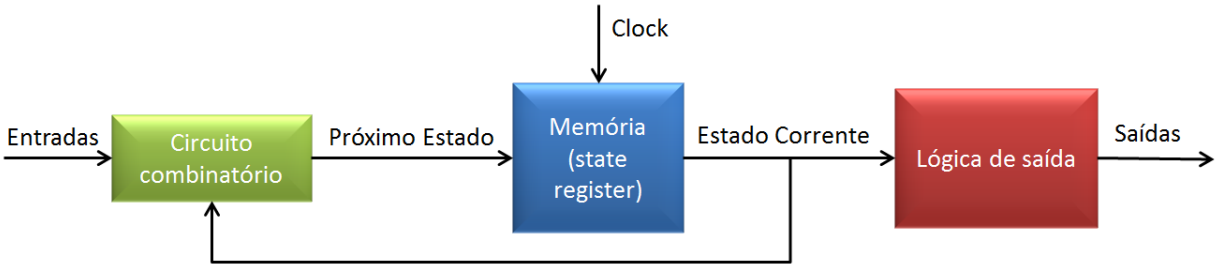


Fig. 6.2 – Máquina de Moore.

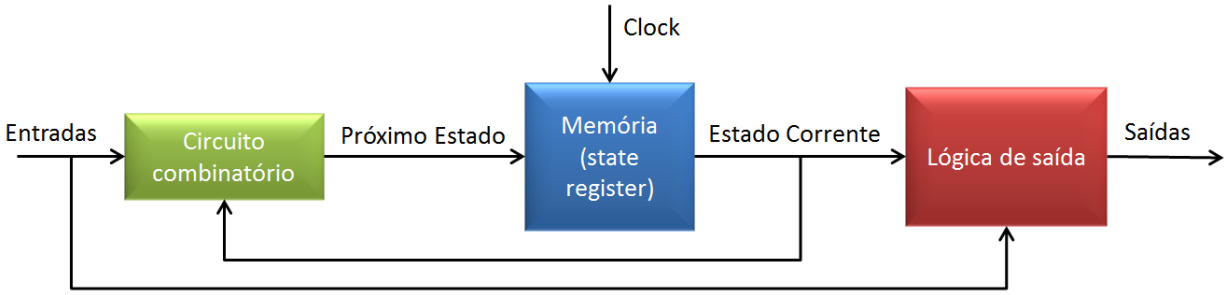


Fig. 6.3 – Máquina de Mealy.

Uma das principais aplicações de uma FSM é actuar como controlador de um sistema digital.

6.2 Máquinas de estados finitos na tese

Nesta tese foram utilizadas várias máquinas de estados finitos: na construção da estrutura de dados em árvore binária, na conversão da lista de código binário para código BCD, na impressão dos valores no monitor e na construção de controladores dos BRAMs. De seguida, ir-se-á descrever em pormenor estas FSMs.

6.2.1 FSM da árvore binária e de ordenação de dados

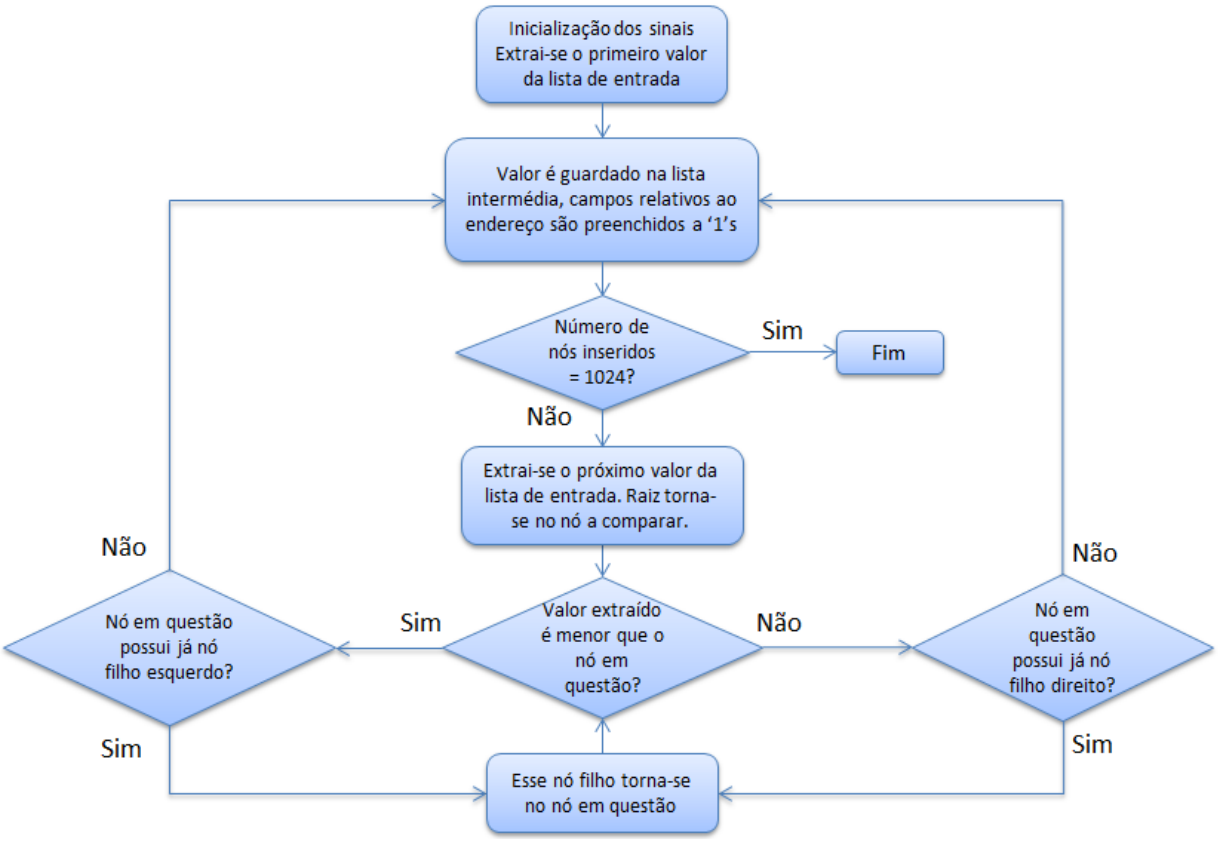


Fig. 6.4 – Diagrama que representa a construção da árvore binária.

A figura 6.4 ilustra o funcionamento da máquina de estados finitos que constrói a árvore binária. O primeiro estado começa por inicializar os sinais, incluindo o sinal que conta o número de ciclos de relógio e extrai-se o primeiro valor da lista de entrada.

Este valor vai ser guardado na primeira posição na lista intermédia (pois é a raiz). No estado seguinte, define-se o endereço do BRAM que guarda a lista de valores recebidos do computador de uso geral de maneira a que seja extraído o valor seguinte. Esse valor é comparado com a raiz, para se saber se possui valor inferior ou superior. Verifica-se também se o valor com o qual se está a comparar já possui nós filhos ou não. Caso não possua, o valor extraído torna-se no seu nó filho. O seu valor é copiado para a lista temporária e os campos destinados aos endereços dos nós filhos (ver fig. 6.5) são preenchidos a '1's (indicando que não possui nós filhos). Caso possua, continua-se a percorrer a árvore até que se encontre uma posição na árvore desocupada.

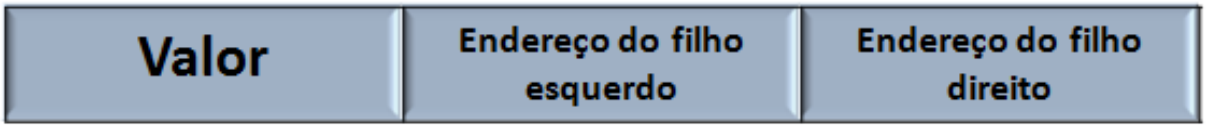


Fig. 6.5 – Campos de um registo da lista temporária.

De seguida, o campo do endereço relativo ao nó filho no registo do nó pai vai ser actualizado (conforme o caso, vai ser actualizado o campo do endereço do nó filho esquerdo ou direito). O valor extraído encontra-se agora inserido na árvore.

Estes estados repetem-se até que todos os valores da lista sejam extraídos da lista de entrada e inseridos na árvore. Nesse momento, a árvore binária estará construída.

Uma vez criada a árvore binária, procede-se à criação da lista com valores ordenados. Começa-se por extrair o primeiro valor da lista intermédia (raiz) e verifica-se se o campo relativo ao endereço do filho esquerdo se encontra preenchido com '1's. Caso não se encontre, o valor desse campo é utilizado para extrair da lista intermédia o nó filho esquerdo. É também utilizada outra lista, chamada *lstack*, com a responsabilidade de registar o endereço dos nós na lista intermédia que vão sendo percorridos, à medida que se atravessa o ramo. Isto permite subir na árvore, depois de se ter investigado a existência de ramos a partir de um nó (ver fig. 6.6).

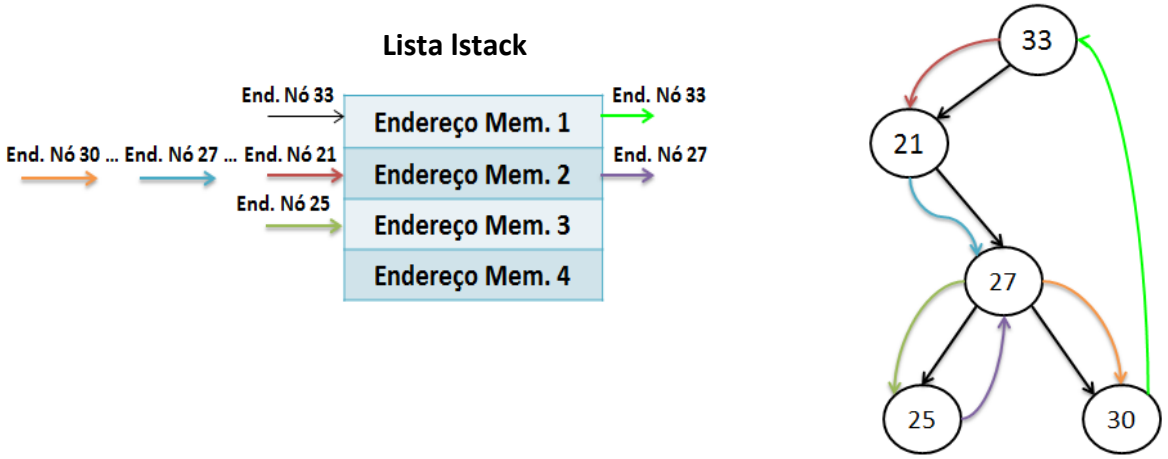


Fig. 6.6 – Exemplo de inserção e extracção dos endereços dos nós de uma árvore utilizando a lista *lstack*.

Estes estados repetem-se até que se encontre um registo cujo campo do endereço do nó filho esquerdo não se encontre preenchido a ‘1’s, sendo que o nó guardado nesse registo é o nó mais à esquerda da árvore. O seu valor é então inserido na primeira posição da lista de valores ordenados (ver fig. 6.7).

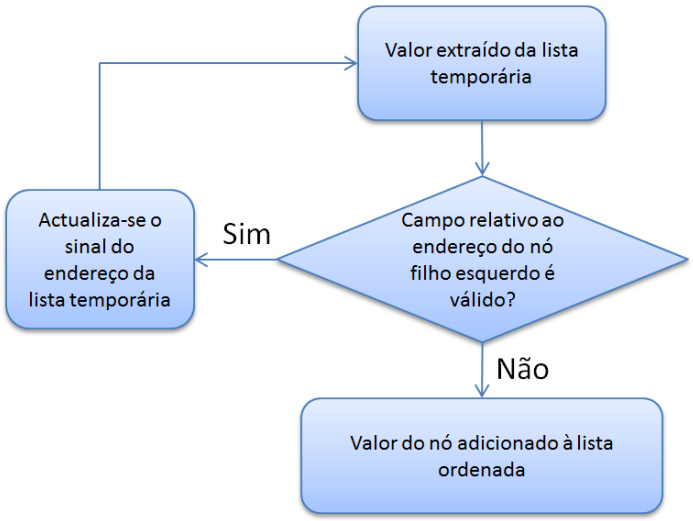


Fig. 6.7 – Diagrama representando a busca da folha mais à esquerda de um ramo.

De seguida, verifica-se se o campo relativo ao endereço do nó filho direito possui um valor válido. Em caso afirmativo, voltam-se a percorrer os estados descritos no parágrafo anterior (procurar o nó mais à esquerda no presente ramo). Caso contrário, extrai-se o registo do nó pai, com a ajuda da lista *lstack*, guarda-se o seu valor na lista ordenada e investiga-se de novo a existência de nó filho direito (ver fig. 6.8).

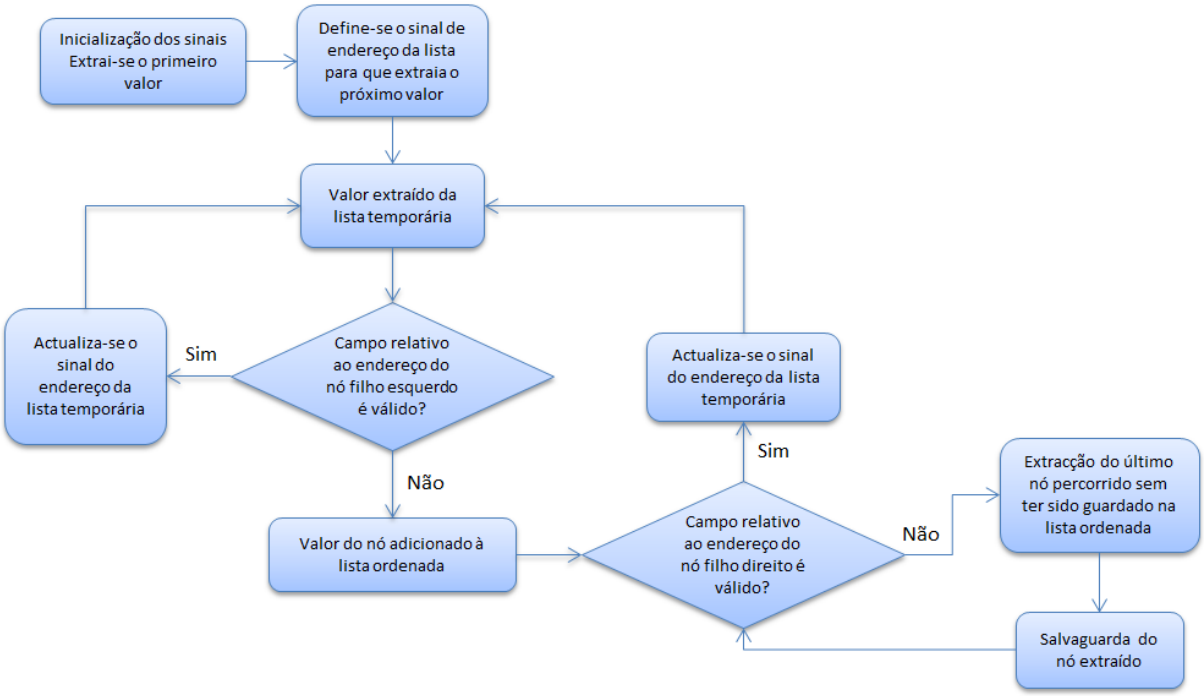


Fig. 6.8 – Diagrama representando a ordenação de uma árvore binária.

Assim que tenham sido ordenados os 1024 valores, a FSM chega ao seu estado final. Neste momento, o sinal contendo o número de ciclos de relógio tomados na execução da FSM é copiado para um sinal de saída, que mais tarde irá ser convertido para código BCD e impresso no monitor.

6.2.2 FSM de conversão de valores binários para BCD

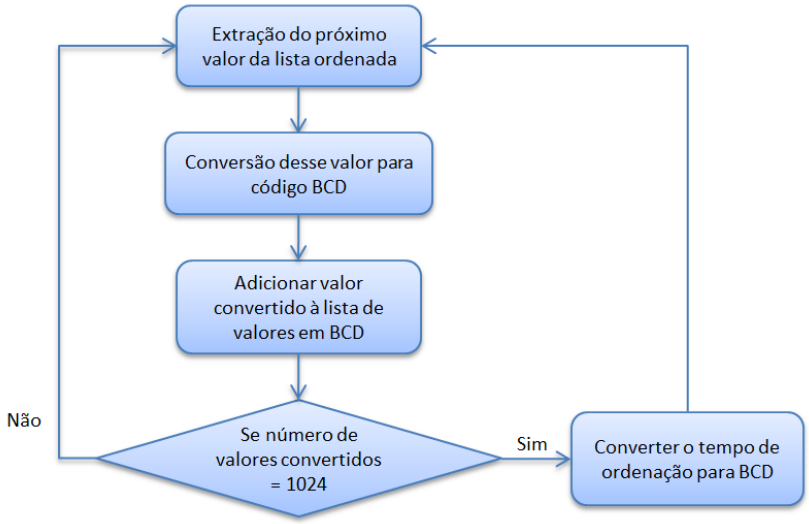


Fig. 6.9 - Diagrama representando o processo de conversão dos valores em código binário para código BCD.

A figura 6.9 mostra a FSM cujo propósito é converter a lista de valores ordenados que se encontra em código binário, criada anteriormente no bloco de ordenação, para outra em código BCD.

No primeiro estado são inicializados todos os sinais. De seguida passa-se para o próximo estado, onde se vai extrair o valor guardado na lista, no registo especificado. Quando esse valor estiver disponível é copiado para o sinal de entrada do conversor bintoBCD e o seu sinal de *reset* fica a '0', o que indica que vai iniciar a conversão. No próximo estado verifica-se se o conversor terminou a conversão e, em caso afirmativo, copia-se o valor convertido para a lista BCD no registo especificado. Caso contrário, a FSM irá permanecer nesse estado. Este ciclo vai ser repetido até que todos os valores estejam convertidos e guardados.

De seguida é convertido o valor do tempo de execução que foi necessário para ordenar os valores da árvore binária. A frequência de trabalho do circuito de ordenação é de 50 MHz, logo o seu período é de 20 ns. Multiplicando o número de ciclos por 20 obtém-se o valor do tempo em ns. O resultado da multiplicação é copiado para a entrada do circuito responsável pela conversão de código binário para BCD, é activado e passa ao estado seguinte. Quando a conversão estiver concluída, o valor do

tempo é guardado num registo que será usado para escrever no monitor. A máquina de estados volta então ao seu estado inicial. Caso haja alterações na lista de valores ordenados (em código binário), esta será rapidamente convertida e impressa no monitor.

6.2.3 FSM de escrita no monitor

O propósito desta FSM é a extracção de valores da lista ordenada em BCD e a sua impressão no monitor (ver fig. 6.10).

No estado inicial os sinais são inicializados. Como o monitor não possui espaço para mostrar 1024 valores numa única página, decidiu-se que numa página seriam escritos os primeiros 512 e na segunda os restantes. Assim, no estado seguinte, caso o segundo botão a contar da direita da placa com FPGA seja pressionado, serão escritos os últimos valores, caso contrário escreve os primeiros valores. O sinal do endereço da lista em BCD começará em 512 ou 0 dependendo do botão estar pressionado ou não. Depois de se definir o sinal do endereço da lista, passa-se ao estado seguinte, onde o seu valor guardado estará disponível. No próximo estado vão ser atribuídos valores aos sinais coluna e ascii. O sinal coluna indica a coluna de um determinado carácter e o sinal ascii o carácter a escrever. De maneira a tornar a máquina reutilizável, para que não se descreva directamente todas as colunas e caracteres, optou-se por utilizar um código que durante a sua execução actualizaria automaticamente todos os sinais relativos à escrita no monitor. Assim, relativamente ao sinal coluna, escreveu-se:

```
column_local <= i*6 - (1+j);
```

Quanto ao sinal onde se define o algarismo a escrever no monitor, ASCII_local, também foi usada uma maneira que permite que seja reutilizável.

```
ASCII_local <= "0011" & a_escrever(3+4*j downto 4*j);
```

Inevitavelmente, a linha acabará por ficar cheia. Isso acontecerá quando i for igual a 16. Nesse caso, o sinal que define a linha a escrever, line_local, é incrementada, i = 1 e j = 0. O processo volta a ser repetido até que os 512 números sejam escritos, sendo então impresso o tempo necessário para a ordenação na FPGA. Quando todos os caracteres estiverem impressos, a máquina de estados volta ao início, de maneira a actualizar possíveis mudanças na lista ou mesmo no estado do botão.

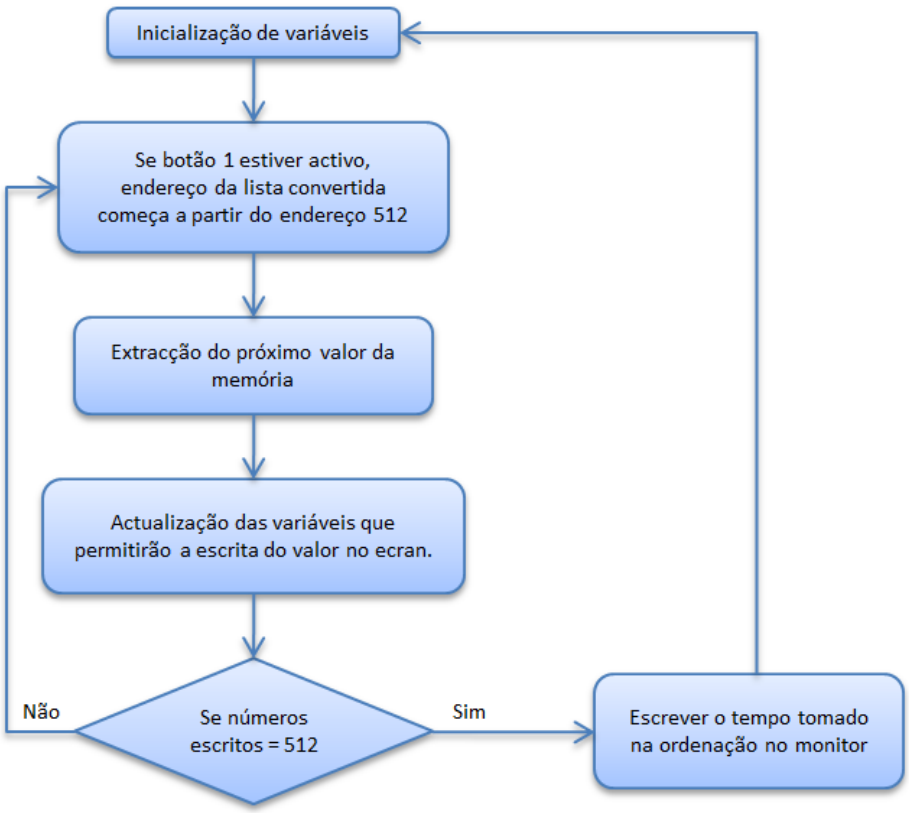


Fig. 6.10 - Diagrama representando o processo de impressão de valores no monitor.

6.3 Conclusões

Neste capítulo apresentou-se a noção de máquinas de estados finitos. Foram descritas as FSMs utilizadas nos blocos de construção da árvore binária e ordenação de dados, conversão de dados de código binário para código BCD e para impressão de dados no monitor.

Capítulo 7

Resultados e Conclusões

Sumário

Neste último capítulo apresentam-se os resultados obtidos neste trabalho, analisa-se a quantidade de recursos ocupados na FPGA e retiram-se as principais conclusões. É também apresentada uma proposta para trabalho futuro.

7.1 Resultados e conclusões

Depois da conclusão do desenvolvimento do *software* no sistema computacional de uso geral e da descrição do circuito lógico para a FPGA, foram efectuados testes em que se verificou o seu correcto funcionamento, podendo assim concluir-se que os objectivos inicialmente propostos foram cumpridos. A criação de uma lista aleatória pelo computador pessoal, a sua transferência para a FPGA, a sua ordenação através da estrutura de dados em árvore binária e a sua exposição para o utilizador ver, quer no computador de uso geral através da janela de comandos (ver fig. 7.1), quer na FPGA através do monitor VGA (ver fig. 7.2), ocorreu nos dois sistemas com sucesso.

```
55692   55698   55730   55804   55820   55842   56074   56154   56160   56186
56190   56236   56260   56294   56454   56648   56754   56770   56876   56946
57024   57036   57116   57188   57228   57248   57422   57502   57512   57586
57600   57646   57678   57732   57756   57806   57896   57928   57966   58364
58458   58482   58546   58718   58818   58834   58864   58878   58928   58940
58968   59028   59050   59114   59202   59206   59266   59374   59558   59594
59604   59628   59658   59740   59784   59820   59916   60084   60102   60222
60280   60300   60312   60396   60498   60506   60514   60694   60724   60762
60774   60782   60788   60822   60836   60838   60852   61000   61118   61248
61286   61288   61312   61400   61536   61596   61600   61626   61660   61666
61668   61694   61706   61758   61794   61832   61844   61856   61936   61958
62048   62094   62120   62236   62236   62266   62346   62372   62400   62402
62508   62512   62568   62588   62624   62682   62760   62790   62820   62968
63014   63214   63308   63364   63458   63480   63570   63612   63616   63784
63824   63998   64138   64206   64268   64328   64348   64414   64438   64442
64488   64490   64512   64576   64578   64648   64932   65090   65096   65172
65214   65338   65368   65410   65508
Tempo necessario para ordenar: 604736 ns.
```

Fig. 7.1 – Imagem da consola com os valores criados e ordenados pelo computador pessoal.

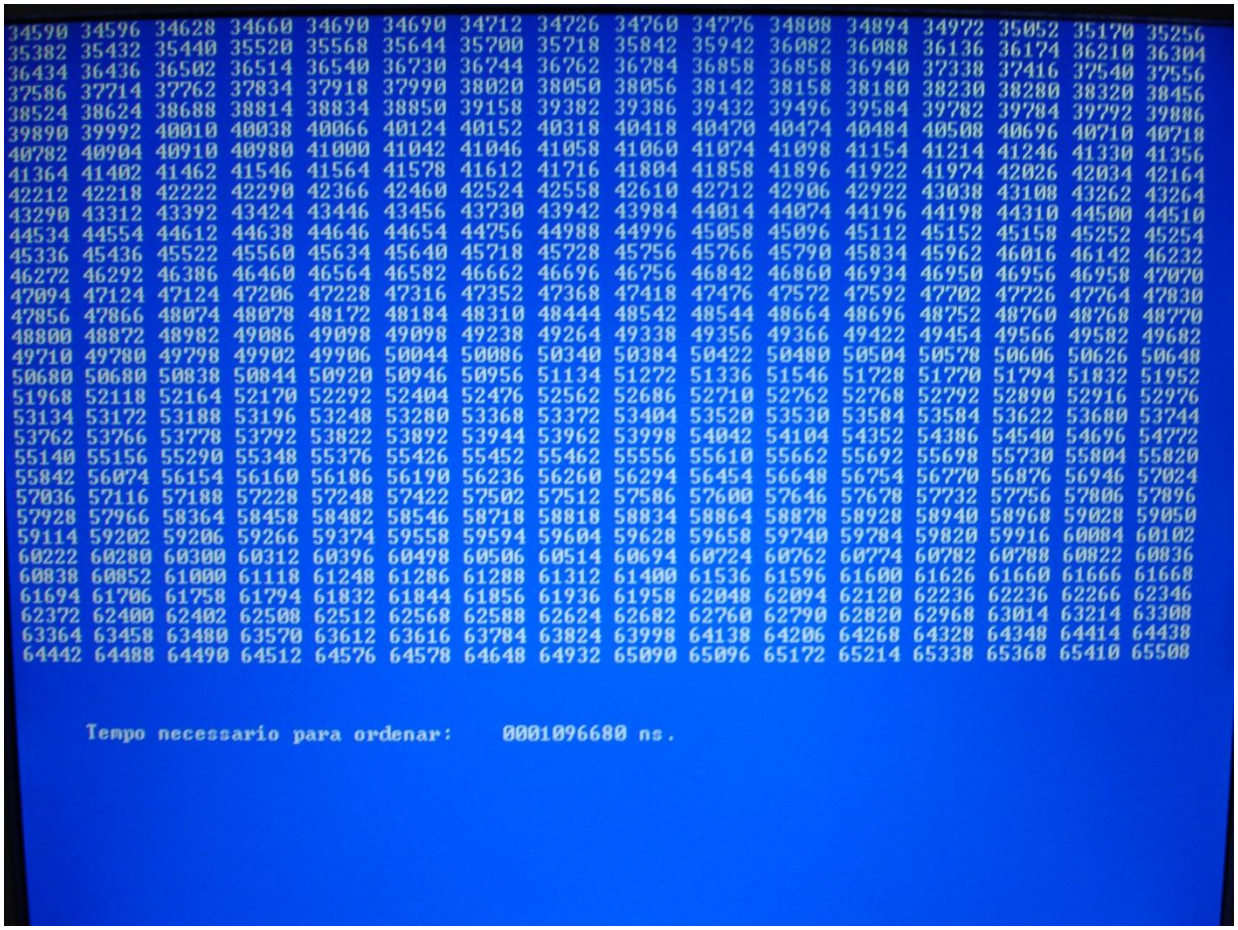


Fig. 7.2 – Imagem do monitor VGA com os valores ordenados pela FPGA (últimos 512 valores).

Para além de se observar o rácio entre o tempo necessário para ordenar a lista com 1024 valores no computador pessoal e na FPGA também se analisou o rácio na ordenação de listas com menor número de valores (ver Tabela 3). Isto permite verificar se o rácio se mantém em listas de menor dimensão.

Número de valores ordenados	Tempo necessário para ordenar no computador pessoal Intel (R) Core (TM) 2 Duo CPU a 2.53 GHz (ns)	Tempo necessário para ordenar na FPGA a 50 MHz (ns)	Rácio entre o valor do tempo despendido na ordenação no computador pessoal e na FPGA
200	98868	167940	0.5887
	94520	150846	0.6266
	97096	164967	0.5886
400	220022	410040	0.5366
	212374	405967	0.5231
	225312	419856	0.5366

600	350617	658740	0.5323
	374565	654239	0.5725
	368541	664031	0.5550
800	491910	911980	0.5394
	500479	927099	0.5398
	490061	914632	0.5358
1024	654817	1152485	0.5682
	668105	1168300	0.5719
	655581	1059760	0.6186

Tabela 3 – Resultados obtidos acerca do tempo de execução da ordenação nos dois sistemas.

Cada valor do tempo necessário para ordenar dados no sistema computacional de uso geral obtido na tabela acima é o resultado da média de 10000 medições, pois o Windows usa *multithreading*. Assim, o valor do tempo obtido em apenas uma operação de ordenação não é o mais próximo da realidade. Pelo contrário, na FPGA o tempo necessário para ordenar a mesma lista é sempre igual.

Verifica-se que o tempo necessário para a ordenação de dados através do *software* no sistema computacional de uso geral é um pouco acima de metade do necessário na FPGA, independentemente do tamanho da lista, sendo a média do rácio de todas as medidas apresentadas de 0.5622. O resultado desejado seria uma situação onde o tempo necessário para ordenar na FPGA fosse igual ou menor ao tomado pelo computador pessoal. É de notar, no entanto, que a frequência de relógio neste sistema é cerca de 50 vezes mais elevada do que a frequência de trabalho utilizada na FPGA. Caso se tivesse utilizado 100 MHz como frequência de trabalho na FPGA, o tempo de cada ciclo de relógio seria reduzido para metade, o que tornaria o tempo necessário para ordenar na FPGA semelhante ao do computador de uso geral.

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Total Number Slice Registers	1,180	9,312	12%	
Number used as Flip Flops	1,119			
Number used as Latches	61			
Number of 4 input LUTs	2,291	9,312	24%	
Number of occupied Slices	1,613	4,656	34%	
Number of Slices containing only related logic	1,613	1,613	100%	
Number of Slices containing unrelated logic	0	1,613	0%	
Total Number of 4 input LUTs	2,459	9,312	26%	
Number used as logic	2,291			
Number used as a route-thru	168			
Number of bonded IOBs	39	232	16%	
Number of RAMB16s	9	20	45%	
Number of BUFGMUXs	7	24	29%	
Number of DCMs	1	4	25%	
Number of MULT18X18SIOs	4	20	20%	
Average Fanout of Non-Clock Nets	3.39			

Fig. 7.3 – Recursos ocupados pelo projecto numa FPGA XC3S500E, referente à utilização de listas com 200 ou 400 valores.

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Total Number Slice Registers	1,242	9,312	13%	
Number used as Flip Flops	1,181			
Number used as Latches	61			
Number of 4 input LUTs	2,301	9,312	24%	
Number of occupied Slices	1,660	4,656	35%	
Number of Slices containing only related logic	1,660	1,660	100%	
Number of Slices containing unrelated logic	0	1,660	0%	
Total Number of 4 input LUTs	2,526	9,312	27%	
Number used as logic	2,301			
Number used as a route-thru	225			
Number of bonded IOBs	39	232	16%	
Number of RAMB16s	11	20	55%	
Number of BUFGMUXs	7	24	29%	
Number of DCMs	1	4	25%	
Number of MULT18X18SIOs	4	20	20%	
Average Fanout of Non-Clock Nets	3.31			

Fig. 7.4 – Recursos ocupados pelo projecto utilizando listas com 600, 800 ou 1024 valores, numa FPGA XC3S500E (Xilinx Spartan-3E).

Como se pode ver nas figuras 7.3 e 7.4, o recurso da FPGA mais utilizado foi o Bloco RAM, independentemente do tamanho da lista. Isto é devido ao tamanho considerável das listas implementadas em BRAM que guardam numerosos dados, usadas na recepção, ordenação e conversão dos dados. No entanto, no que toca aos restantes recursos utilizados (DCMs, multiplicadores, *slice registers*, etc.) não é requerida uma grande utilização da parte destes.

De notar também que para guardar listas com 200 e 400 valores o número de BRAMs necessários é igual. Isto deve-se ao facto de os blocos RAM ocupados para guardar listas de 200 valores possuírem tamanho suficiente para guardar listas de 400 valores. A mesma situação acontece para 600, 800 e 1024 valores.

Com o seu baixo custo a FPGA apresenta-se, assim, como uma alternativa viável ao uso de computadores de uso geral na implementação deste tipo de algoritmo, especialmente em sistemas dedicados.

7.2 Trabalho Futuro

Apesar de não ser uma novidade [43], o desenvolvimento de algoritmos de ordenação utilizando a estrutura de dados árvore binária é um campo que continua a ser alvo de investigação [44]. Neste projecto foi implementado em FPGA um algoritmo iterativo, no entanto a escolha de algoritmos recursivos pode tornar certas implementações mais rápidas [45], embora na maior parte dos casos acabe por ocupar também mais recursos [46]. Para trabalho futuro sugere-se o desenvolvimento de um algoritmo de árvore binária recursivo, utilizando máquinas de estados finitos hierárquicas paralelas [47] (PHFSM – Parallel Hierarchical Finite State Machines) ou

através de máquinas de estados finitos hierárquicas recursivas [48] (RHFSM – Recursive Hierarchical Finite State Machines), pois permitiria observar o tempo de execução e recursos ocupados, de maneira a escolher-se o melhor tipo de implementação.

Anexo A

Código da FSM que constrói a árvore binária e ordena os dados

A seguir mostra-se a máquina de estados (FSM) que foi usada para criar a árvore binária e consequente ordenação:

```
process (clk_in)
begin
if falling_edge (clk_in) then
next_state <= current_state;
next_module <= current_module;
inc <= '0';
dec <= '0';
activar_escrita_saida <= '0';

if terminado = '0' then
       ciclos <= ciclos +1 ;
end if;

case current_module is
  when z0 =>
       case current_state is
         when a0 =>
               ciclos <= (others => '0');
               ler_ram_entrada <= '0';
               mem_addr_sort <= (others => '0');
               mem_addr_sort_signal <= (others => '0');
               ram_ender_in <= (others => '0');
               ram_ender_in_1 <= (others => '0');
               ram_ender_out <= (others => '0');
               reg_a_comparar(RAM_data+2*RAM_address_size-1    downto    2*RAM_address_size)    <=
       douta_sort;
               reg_a_comparar(2*RAM_address_size-1 downto 0) <= (others => '1');

               lstack_ender_escrever <= (others => '0');
               lstack_ender_ler <= (others => '0');

               lstack_dados_write <= (others => '0');

               lstack_pointer_wr <= (others => '0');
               lstack_pointer_read <= (others => '0');

               atraso <= 0;
               ram_dados_in(RAM_data+2*RAM_address_size-1    downto    2*RAM_address_size)    <=
       douta_sort;
               ram_dados_in(2*RAM_address_size-1 downto 0) <= (others => '1');

               terminado <= '0';
               done <= '0';

               next_state <= a1;

       when a1 =>
               ram_ender_in <= ram_ender_in_1 +1;
               mem_addr_sort_signal <= mem_addr_sort_signal +1;
               next_state <= a1_1;

       when a1_1 =>
               ram_ender_in_1 <= ram_ender_in;
               ler_ram_entrada <= '1';
               mem_addr_sort <= mem_addr_sort_signal;
               next_state <= a2;
```

```
when a2 =>
	if lido_entrada = '1' then
		ler_ram_entrada <= '0';
		AddMe <= douta_sort;
		reg_a_comparar <= ram_dados_out;
		next_state <= a3;
		else next_state <= a2;
	end if;

when a3 =>
	if	AddMe	<	reg_a_comparar(RAM_data+2*RAM_address_size-1	downto
	2*RAM_address_size) then
		if	reg_a_comparar(2*RAM_address_size-1	downto	RAM_address_size)	=
		"11111111111" then
			ram_dados_in(RAM_data+2*RAM_address_size-1			downto
		2*RAM_address_size) <= AddMe;
			ram_dados_in(2*RAM_address_size-1 downto 0) <= (others => '1');
			next_state <= a3_1;

		else
			ram_ender_out	<=	reg_a_comparar(2*RAM_address_size-1	downto
		RAM_address_size);
			reg_a_comparar <= ram_dados_out;
			next_state <= a3;
		end if;

	elsif reg_a_comparar(RAM_address_size-1 downto 0) = "11111111111" then
		ram_dados_in(RAM_data+2*RAM_address_size-1 downto 2*RAM_address_size) <=
	AddMe;
		ram_dados_in(2*RAM_address_size-1 downto 0) <= (others => '1');
		next_state <= a3_2;

	else
		ram_ender_out <= reg_a_comparar(RAM_address_size-1 downto 0);
		reg_a_comparar <= ram_dados_out;
		next_state <= a3;
	end if;

when a3_1 =>
	ram_ender_in <= ram_ender_out;
	ram_dados_in	<=	ram_dados_out(RAM_data+2*RAM_address_size-1	downto
2*RAM_address_size) & ram_ender_in_1 & ram_dados_out(RAM_address_size-1 downto 0);
	next_state <= a4;

when a3_2 =>
	ram_ender_in <= ram_ender_out;
	ram_dados_in	<=	ram_dados_out(RAM_data+2*RAM_address_size-1	downto
RAM_address_size) & ram_ender_in_1;
	next_state <= a4;

when a4 =>
	if ram_ender_in_1 = ROM_words-1
	then
		next_state <= a5;

	else
		next_state <= a1;
	end if;

	ram_ender_out <= (others => '0');

when a5 =>
	if ender_ram_saida = ROM_words then
		next_state <= a9;
	end if;

	if atraso = 2 then
		reg_a_comparar <= ram_dados_out;
		atraso <= 0;
		next_state <= a6;
	else atraso <= atraso +1;
	end if;

when a6 =>
	if reg_a_comparar(2*RAM_address_size-1 downto RAM_address_size) /= "11111111111"
	then
		lstack_ender_escrever <= lstack_pointer_wr+1;
```

```
                        lstack_pointer_wr <= lstack_pointer_wr +1;
                        lstack_ender_ler <= lstack_ender_escrever;
                        lstack_pointer_read <= lstack_pointer_read+1;
                        lstack_dados_write    <=    reg_a_comparar(2*RAM_address_size-1    downto
                RAM_address_size);
                        ram_ender_out      <=      reg_a_comparar(2*RAM_address_size-1      downto
                RAM_address_size);
                        next_state <= a5;

                elsif reg_a_comparar(RAM_address_size-1 downto 0) /= "11111111111" then
                        lstack_pointer_read <= lstack_pointer_read+1;
                        lstack_ender_escrever <= lstack_pointer_wr;
                        lstack_dados_write <= reg_a_comparar(RAM_address_size-1 downto 0);
                        ram_ender_out <= reg_a_comparar(RAM_address_size-1 downto 0);
                        activar_escrita_saida <= '1';
                        ender_sort <= ender_ram_saida;
                        escrever_dados_saida    <=    reg_a_comparar(RAM_data+2*RAM_address_size-1
                downto 2*RAM_address_size);
                        next_state <= a5;


                else --- escrever para a ram de saida
                        activar_escrita_saida <= '1';
                        ender_sort <= ender_ram_saida;
                                        escrever_dados_saida                                                    <=
                reg_a_comparar(RAM_data+2*RAM_address_size-1 downto 2*RAM_address_size);

                        lstack_pointer_read <= lstack_ender_ler;
                        lstack_pointer_wr <= lstack_ender_ler;
                        ram_ender_out <= lstack_dados_read;
                        next_state <= a7;
                end if;

        when a7 =>
                if ender_ram_saida = ROM_words then
                        next_state <= a9;
                end if;

                if atraso = 2 then
                        ---     escrever para a ram de saida
                        activar_escrita_saida <= '1';
                        ender_sort <= ender_ram_saida;
                        escrever_dados_saida    <=    ram_dados_out(RAM_data+2*RAM_address_size-1
                downto 2*RAM_address_size);

                        reg_a_comparar <= ram_dados_out;

                        ram_ender_out <= lstack_dados_read;
                        atraso <= 0;
                        next_state <= ver_apenas_direita;

                        else atraso <= atraso +1;
                end if;
--
--
        when ver_apenas_direita =>

                if ender_ram_saida = ROM_words then
                        next_state <= a9;

                elsif ram_dados_out(RAM_address_size-1 downto 0) /= "11111111111" then
                        if lstack_ender_ler > 0 then
                                lstack_ender_ler <= lstack_pointer_read-1;
                        end if;

                        lstack_pointer_read <= lstack_pointer_read+1;

                        lstack_ender_escrever <= lstack_pointer_wr;
                        lstack_dados_write <= reg_a_comparar(RAM_address_size-1 downto 0);
                        ram_ender_out <= reg_a_comparar(RAM_address_size-1 downto 0);
                        next_state <= a5;

                else
                        --- escrever para a ram de saida
                        next_state <= a7;

                        if lstack_ender_ler > 0 then
                                lstack_ender_ler <= lstack_pointer_read-1;
```

```
                        lstack_pointer_read <= lstack_pointer_read-1;
                        lstack_pointer_wr <= lstack_ender_ler-1;
                        next_state <= a8;
                end if;

                ram_ender_out <= lstack_dados_read;

        end if;

    when a8 =>
        if atraso = 2 then
                atraso <= 0;
                ram_ender_out <= lstack_dados_read;
                next_state <= a7;

        else atraso <= atraso +1;
        end if;

    when a9 =>
        done <= '1';
        terminado <= '1';
        cycles <= ciclos;

    when others => null;

    end case;
when others => null;
end case;

end if;
end process;


process (clk_in,rst)
begin

    if rst = '1' then
        ender_ram_saida <= (others => '0');

    elsif falling_edge (clk_in) then

        if activar_escrita_saida = '1' then
        ender_ram_saida  <= ender_ram_saida +1 ;
        end if;

    end if;

end process;
```

Anexo B

Nomenclatura

BRAM – Bloco RAM

CI – Circuito Integrado

CLB – Configurable Logic Block

CPLD – Complex Programmable Logic Device

DCM – Digital Clock Manager

EEPROM – Electrically Erasable Programmable Read Only

FPGA – Field Programmable Gate Array

FSM – Finite State Machine

GAL – Generic Logic Array

HDL – Hardware Description Language

IOB – Input/Output Block

IP – Intelectual Property

ISE – Integrated Synthesis Environment

LUT – Lookup Table

PAL – Programmable Array Logic

PLD – Programmable Logic Device

PROM – Programmable ROM

RAM – Random Access Memory

ROM – Read Only Memory

UCF – User Constraint File

USB – Universal Serial Bus

VHDL – VHSIC Hardware Description Language

Bibliografia

[1] http://www.digilentinc.com/Products/Detail.cfm?Prod=ADEPT.

[2] http://www.sapteka.net/IntroductionToPLD.htm.

[3] http://www.absoluteastronomy.com/topics/Programmable_logic_device.

[4] http://www.netrino.com/Embedded-Systems/How-To/Programmable-Logic.

[5] http://www.fpga4fun.com.

[6] www.xilinx.com.

[7] www.altera.com.

[8] http://seekingalpha.com/article/85478-altera-and-xilinx-report-the-battle-continues.

[9] www.latticesemi.com.

[10] www.actel.com.

[11] www.siliconbluetech.com.

[12] www.quicklogic.com.

[13] http://www.digilentinc.com/Products/Detail.cfm?Prod=NEXYS2.

[14] http://c-to-verilog.com.

[15] http://www.impulseaccelerated.com.

[16] Spartan-3E FPGA Family: Data Sheet, Xilinx®, 2009
Available at:
http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf.

[17]
http://www.synopsys.com/Tools/SLD/AlgorithmicSynthesis/Documents/syn_dsp_ds.pdf.

[18]
http://www.mentor.com/products/fpga/synthesis/precision_rtl/upload/PrecisionFamilyAug2007.pdf.

[19] Prof. Dr. J. Reichardt, *Tools and Steps in the VHDL based ISE/ModelSim Design Flow*, Hamburg University of Applied Sciences, 2003.
Available at:
http://users.etech.fh-hamburg.de/users/Reichardt/ISE_design_method.pdf.

[20]
http://63.241.181.135/itp/xilinx10/isehelp/ise_c_implement_fpga_design.htm.

[21] http://www.csit-sun.pub.ro/research/fpga/fpga_design/soft.html.

[22] http://www.ece.ualberta.ca/~cmpe480/lab/labs/lab0/lab0.pdf.

[23] http://www.xilinx.com/itp/xilinx4/data/docs/cgd/entry7.html.

[24]
http://www.xilinx.com/tools/feature/csi/coregen.htm?height=700&width=600.

[25] Using Block RAM in Spartan-3 Generation FPGAs, Xilinx®, March 1, 2005.
Available at:
http://www.xilinx.com/support/documentation/application_notes/xapp463.pdf.

[26]
http://www.al.urcamp.tche.br/infocamp/edicoes/marc06/arvores_binarias.pdf.

[27] http://ctp.di.fct.unl.pt/~alopes/aed/Ch5-Arvores.pdf.

[28] http://www.ime.usp.br/~pf/algoritmos/aulas/bint.html.

[29]
http://cs.wellesley.edu/~cs230/spring07/lectures/lec17_trees/lec17_trees.pdf.

[30] http://d.yimg.com/kq/groups/23650627/855905455/name/arvores-binarias.pdf.

[31]
http://www.jcmiras.net/computers_and_internet/binary_tree_traversals/index.html.

[32] Hoang Le, Weirong Jiang, Viktor K. Prasanna, *Scalable high-throughput sram-based architecture for ip-lookup using fpga*, in International Conference on Field Programmable Logic and Applications, 2008.

Available at: http://halcyon.usc.edu/~pk/prasannawebsite/papers/hle_fpl08.pdf.

[33] Padmanabhan Balasubramanian, Cemal Ardil, *Compact Binary Tree Representation of Logic Function with Enhanced Throughput*, International Journal of Computer, Information, and Systems Science, and Engineering, 2007.
Available at: http://www.waset.org/journals/ijcisse/v1/v1-2-12.pdf.

[34] M. Chandrashekar, U. Naresh Kumar, K. Sudershan Reddy and K. Nagabhushan Raju, *FPGA Implementation of High Speed Infrared Image Enhancement,* International Journal of Electronic Engineering Research, ISSN 0975 – 6450 Volume 1 Number 3, pp. 279–285, © Research India Publications, 2009.
Available at: http://www.ripublication.com/ijeerv1/ijeerv1n3_11.pdf.

[35] Terrence S. T. Mak and K. P. Lam, *Embedded Computation of Maximum-Likelihood Phylogeny Inference Using Platform FPGA*, Proceedings of the 2004 IEEE Computational Systems Bioinformatics Conference, pp. 512-514, 2004.
Available at:
http://gatekeeper.dec.com/pub/toomany/068_Mak_Terrence_Embedded_Phylogeny.pdf.

[36] S. M. A. Motakabber, Mohd Alauddin Mohd Ali, Nowshad Amin, *VLSI Design of an Anti-Collision Protocol for RFID Tags*, European Journal of Scientific Research, ISSN 1450-216X Vol.28 No.4 (2009), pp.559-565, © EuroJournals Publishing, Inc., 2009.
Available at: http://www.eurojournals.com/ejsr_28_4_07.pdf.

[37] Tiago Maritan Ugulino de Araújo, Eduardo Ribas Pinto, José Antônio Gomes de Lima and Leonardo Vidal Batista, *An Fpga Implementation of a microprogrammable Controller to Perform Lossless Data Compression Based on the Huffman Algorithm*, Workshop Iberchip, 2007, Lima. Anais do XIII Workshop Iberchip, 2007. v. 1. pp. 100-103.
Available at: http://www.iberchip.org/iberchip2007/articulos/5/a/paper/3--tiagomaritan-HuffmanFPGA_Iberchip2007-Final.pdf.

[38] Herbert Walder, Christoph Steiger, Marco Platzner, *Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing*, Proceedings of the 17th International Symposium on Parallel and Distributed Processing, 2003.
Available at: http://typo3.cs.uni-paderborn.de/fileadmin/Informatik/AG-Platzner/publications/walder03_raw/walder03_raw.pdf.

[39] Xiaofang Wang, Sotirios G. Ziavras, *Performance Optimization of an FPGA-Based Configurable Multiprocessor for Matrix Operations*, In Proceedings of IEEE International Conference on Field Programmable Technology, 2003.
Available at: http://web.njit.edu/~ziavras/FPT03.pdf.

[40] Digilent Parallel Interface Model Reference Manual, Digilent Inc.
Available at:
http://www.digilentinc.com/Data/Products/ADEPT/DpimRef%20programmers%20manual.pdf.

[41] Pong P. Chu, *FPGA Prototyping by VHDL Examples* - Xilinx Spartan 3 Version. John Wiley & Sons, Inc., 2008.

[42] V. Sklyarov, *Reconfigurable models of finite state machines and their implementation in FPGAs,* Journal of Systems Architecture, 47, 2002, pp. 1043-1064.
Available at:
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.129.9198&rep=rep1&type=pdf.

[43] I. Skliarova, *Implementation of Recursive Search Algorithms in Reconfigurable Hardware*, Proceedings of the 4th Winter International Symposium on Information and Communication Technologies - WISICT'05, Cape Town, South Africa, January 2005, pp. 142-147.
Available at:
http://www.ieeta.pt/~iouliia/Papers/2005/WISICT_Skliarova.pdf.

[44] D. Mihhailov, V. Sklyarov, I. Skliarova, A. Sudnitson, *Hardware Implementation of Recursive Algorithms*, actas de 53rd IEEE International Midwest Symposium on Circuits and Systems, Seattle, Washington, EUA, Agosto de 2010.

[45] V. Sklyarov, I. Skliarova, B. Pimentel, *Fpga-Based Implementation And Comparison Of Recursive And Iterative Algorithms*, Proceedings of the 15th International Conference on Field-Programmable Logic and Applications - FPL'2005, Finland, August 2005, pp. 235-240.
Available at:
http://www.ieeta.pt/~iouliia/Papers/2005/FPL_2005.pdf.

[46] V. Sklyarov, I. Skliarova, *Recursive and Iterative Algorithms for N-ary Search Problems*, International Federation for Information Processing, vol. 218, 2nd IFIP Symposium on Professional Practice in Artificial Intelligence - AISPP'2006, ed. J. Debenham, 19th IFIP World Computer Congress - WCC'2006, Santiago de Chile, Chile, August 2006, pp. 81-90.
Available at:
http://www.ieeta.pt/~skl/research/Papers/ChileRecursive.pdf.

[47] V. Sklyarov, I. Skliarova, *Design and Implementation of Parallel Hierarchical Finite State Machines*, Proceedings of the 2nd International Conference on

Communications and Electronics – HUT-ICCE'2008, Hoi An, Vietnam, June 2008, pp. 33-38.
Available at:
http://www.ieeta.pt/~iouliia/Papers/2008/PHFSM.pdf.

[48] V.Sklyarov, *FPGA-based implementation of recursive algorithms. Microprocessors and Microsystems*, Special Issue on FPGAs: Applications and Designs, 2004, Vol 28/5-6 pp 197-211.
Available at:
http://www.ieeta.pt/~skl/research/Papers/Recursive2004.pdf.