Universidade de Évora
Escola de Ciências e Tecnologia

Mestrado em Engenharia Informática

Dissertação

Simulador para Arquitectura MIPS32

David João Domingues Rodrigues Maia

Orientador
Miguel José Simões Barão

Março de 2013

Mestrado em Engenharia Informática

Dissertação

Simulador para Arquitectura MIPS32

David João Domingues Rodrigues Maia

Orientador
Miguel José Simões Barão

Quero dedicar este modesto trabalho aos meus três avós, que já partiram para um lugar melhor. A José João Pedro da Conceição Belezas, uma vez que nasceu com ele a minha vontade de estudar engenharia e a José da Costa Maia e António Augusto Nunes Domingues que sempre foram uma inspiração em tempos difíceis.

Sumário

A virtualização de sistemas é cada vez mais utilizada no mundo informático. O seu emprego acarreta inúmeras vantagens, sendo que, em alguns casos, permite atingir melhor desempenho relativamente a uma máquina nativa.

Esta tese propõe um modelo de implementação de um simulador da arquitectura MIPS32 utilizando a linguagem de programação C, sendo as aplicações de teste desenvolvidas utilizando a linguagem assembly MIPS. É objectivo recriar os primeiros passos no processo de virtualização de sistemas, assim como possibilitar a instalação de um minissistema operativo, baseado na família linux, no simulador. Para tal, será necessário reproduzir o comportamento de vários dispositivos físicos, tais como o disco rígido, interface de rede, TLB, cache, rato, teclado e monitor. Embora estes sejam dispositivos desejáveis, apenas o processador e a memória RAM são componentes fulcrais ao funcionamento do simulador.

De forma a respeitar os requisitos mínimos da arquitectura serão implementados todos os mecanismos necessários, nomeadamente, coprocessadores, modos de operação, registos genéricos e registos do coprocessador central, unidade de gestão de memória, mecanismo de tradução de endereços, sistema de excepções e sistema de interrupções.

Simulator for the MIPS32 Architecture

Abstract

Virtualization systems are increasingly used in the computer world. Their use brings numerous advantages and, in some cases, allows to achieve better performance compared to a native machine.

This thesis proposes an implementation model of a simulator for MIPS32 architecture using the C programming language and the test applications developed using the MIPS assembly language. The aim is to recreate the first steps in the process of virtualization systems, as well as to enable the installation of a Linux-based mini operating system in the simulator. This will need to reproduce the behavior of several physical devices such as hard disk, network interface, memory management unit including a translation lookaside buffer (TLB), cache, mouse, keyboard, monitor. Although these devices are desirable, only the processor and main memory RAM are key components to the operation of the simulator.

In order to meet the minimum requirements of the architecture, all the necessary mechanisms will be implemented including coprocessors, operating modes, generic registers and records of the central coprocessor, memory management unit, address translation mechanism and the exception and interruption systems.

Prefácio

Este documento contém a dissertação intitulada “Simulador para a Arquitectura MIPS32”, entregue em Outubro de 2012 no âmbito do Mestrado em Engenharia Informática do autor David João Domingues Rodrigues Maia[1], na Universidade de Évora, em Portugal. O autor é Licenciado em Engenharia Informática, também pela Universidade de Évora. Actualmente é administrador de sistemas de informação na empresa Deloitte, BPAS - AMS, em Portugal.

O orientador deste trabalho é o Professor Doutor Miguel José Simões Barão[2], Professor auxiliar no Departamento de Informática da Universidade de Évora.

[1] m5847@alunos.uevora.pt
[2] mjsb@uevora.pt

Agradecimentos

Com o desenvolvimento do trabalho final de Mestrado culmina uma importante fase na vida de qualquer estudante. Desta forma, gostaria de dedicar umas palavras de apreço a um conjunto de pessoas, que directa ou indirectamente, ajudaram no desenvolvimento do presente trabalho.

Aos meus pais e irmão, um grande agradecimento pelo amor incondicional, assim como pelo enorme esforço e sacríficos feitos ao longo desta fase, sem os quais nada disto seria possível. Muito obrigado Pai, Mãe e João!

Quero agradecer ao meu orientador, Professor Miguel Barão, pelo incessante apoio, compreensão e claramente pela disponibilidade demonstrada ao longo do desenvolvimento de todo o trabalho. Muito obrigado professor Miguel Barão!

Gostaria de agradecer especialmente à minha namorada Renata Carmona, pelo constante apoio e dedicação no desenvolvimento da dissertação, sem o qual não teria sido possível a entrega a tempo. Um muito obrigado e um beijo especial!

Aos meus colegas e amigos, Daniel Quina, Alexandre Cabo, Laura Simões, Nelson Dias e Hugo Teodoro, gostaria de agradecer pelo incentivo, conselhos e pela troca de ideias, muito importantes na resolução de problemas, assim como pelo suporte no desenvolvimento prático. Obrigado a todos!

Notação e símbolos

$\|\|$	Concatenação de bits numa palavra de 32 bits.
$x_{y..z}$	Selecção dos bits desde y até z da palavra de 32 bits X.
0bn	Valor da constante n em base 2. (Binário)
0xn	Valor da constante n em base 16. (Hexadecimal)
OR	Operação Lógica OR
AND	Operação Lógica AND
XOR	Operação Lógica XOR
NOR	Operação Lógica NOR
USEG	Espaço de memória virtual mapeada destinado ao modo de operação *user*.
KSEG0	Espaço de memória virtual não mapeada com utilização de *cache* destinado ao modo de operação *kernel*. Este segmento aponta para os primeiros 512 Bytes na memória RAM.
KSEG1	Espaço de memória virtual não mapeada sem utilização de *cache* destinado ao modo de operação *kernel*.
KSSEG	Espaço de memória virtual mapeado destinado ao modo de operação *superuser* e *kernel*.
KSEG3	Espaço de memória virtual mapeada destinado ao modo de operação *kernel*.
CP0..3	Coprocessador 0 ... 3
SEL	Variável utilizada pelo processador para identificar o sub registo associado ao registo interno do CP0.

Acrónimos

ALU Arithmetic Logic Unit

CISC Complex Instruction Set Computer

CPU Central Processing Unit

ELF Executable Linkable Format

FMT Fixed Map Translation

FPU Floating Point Unit

GPR General Purpose Register

GPT Global Page Table

IF Instruction Fetch

IPC Inter Process Communication

ISA Instruction Set Architecture

LSB Least Significant Byte

MEM Memory

MIPS Microprocessor without Interlocked Pipeline Stages

MMU Memory Management Unit

MSB Most Significant Byte

PC Program Counter

PFN Physical Frame Number

PRA Privileged Resource Architecture

PTB Page Table Base

PTE Page Table Entry

PTL Page Table Limit

RAM Random Access Memory

RISC Reduced Instruction Set Computer

ROM Read Only Memory

RD Read Registers

TLB Translation Lookaside Buffer

VPN Virtual Page Number

VMM Virtual Machine Monitor

WB Write Back

Conteúdo

Sumário i

Abstract iii

Prefácio v

Agradecimentos vii

Notação e símbolos ix

Acrónimos xii

1 Introdução 1

1.1 Enquadramento e Motivação . . . 1

1.2 Objectivos . . . 3

2 Estado da Arte e Conceitos Envolvidos 5

2.1 Virtualização de Sistemas . . . 5

2.2 Técnicas de tradução e compilação de instruções . . . 13

2.3 Comunicação entre processos (IPC) . . . 16

2.3.1 Named e Unnamed Pipes . . . 16

2.4 Executable and Linkable Format (ELF) . . . 18

2.4.1 Organização de ficheiros ELF . . . 18

2.4.2 Interpretação de ficheiros ELF 22

3 Sistema Proposto 25

3.1 Apresentação 25

3.1.1 O que se pretende 26

3.1.2 Metodologias 28

3.2 Arquitectura 30

3.2.1 Módulos do Sistema 31

3.2.2 Plataforma de desenvolvimento 33

4 Módulos 35

4.1 Processador 36

4.1.1 Modos de Execução 39

4.1.2 Endianness 42

4.1.3 Interpretação de código máquina 45

4.1.4 Pipelines 52

4.1.5 Coprocessadores 57

4.1.6 Registos Genéricos 57

4.1.7 Registos do Coprocessador Central 59

4.1.8 Mecanismo de Excepções 72

4.2 Unidade de Gestão de Memória (MMU) 92

4.2.1 Memória Virtual 92

4.2.2 Paginação 95

4.2.3 Simulação em Software 100

4.3 Mecanismo de Tradução de Endereços (TLB) 104

4.3.1 Interface de comunicação MMU/TLB 105

4.3.2 Instruções de controlo MMU/TLB 109

4.3.3 TLB Flush vs ASID's 110

4.3.4 Excepções da TLB 112

4.3.5 Simulação em Software 116

4.4 Memórias RAM e ROM 121

4.4.1 Simulação em Software 123

4.5 Simulador UE 127
4.5.1 Carregamento e arranque do sistema 127
4.5.2 Execução 133
4.5.3 Término de Execução 134

5 Utilização do Sistema 137
5.1 Interface e funcionamento 138
5.2 Demonstrações 142
5.2.1 Cálculo do factorial em modo e zona de *Kernel* 142
5.2.2 Cálculo da sequência de Fibonacci em modo e zona de *User* 146
5.2.3 Casos de erro e mecanismos MIPS 151
5.2.4 Cálculo de números primos em C (GCC) 157

6 Conclusões 161
6.1 Objectivos alcançados 161
6.2 Limitações 163
6.3 Comparação com trabalhos relacionados 164
6.4 Trabalho Futuro 168

Bibliografia 172

A Compilação de código binário - Makefile 175

B Código Assembler MIPS - Demonstração 1 177

C Código Assembler MIPS - Demonstração 2 181

D Código Assembler MIPS - Demonstração 3 185

E Código Assembler MIPS - Demonstração 4 193

Lista de Figuras

2.1 Hypervisor tipo 1. 6

2.2 Hypervisor tipo 2. 7

2.3 Camadas da emulação de sistemas. 8

2.4 Camadas de virtualização de sistemas completa ou nativa. 9

2.5 Camadas em sistemas paravirtualizados. 11

2.6 Arquitectura Intel VT-x/ VT-i. 12

2.7 Estrutura de compilação para a linguagem Java. 14

2.8 Arquitectura do emulador QEMU. 15

2.9 Comunicação com Pipelines Unix. 17

2.10 Formato de um ficheiro binário ELF. 19

3.1 Camadas de abstração. 26

3.2 Módulos do SimuladorUE. 31

4.1 Máquina de estados da execução no processador. 38

4.2 Tipos de dados no SimuladorUE. 43

4.3 Little Endian. 44

4.4 Big Endian. 44

4.5 Pipeline de instrução única com profundidade 1. 54

4.6 Pipeline paralelo com profundidade 5. 54

4.7 Super pipeline com profundidade 5. 55

4.8 Pipeline Superescalar. 55
4.9 STACK para modo de execução Kernel em sistemas Linux. 77
4.10 Diagrama de actividade para as excepções Reset, Soft Reset e NMI. . . . 89
4.11 Diagrama de actividade para as excepções diferentes de Reset, Soft Reset e NMI. 91
4.12 Estrutura da memória virtual para arquitecturas de 32 bits. 93
4.13 Mecanismo de tradução em sistemas com paginação. 97
4.14 Mecanismo de paginação linear. 98
4.15 Mecanismo de paginação multinível. 99
4.16 Escalonamento de processos com ASID. 111
4.17 Diagrama de sequência para a tradução de endereços virtuais em endereços físicos. 119
4.18 Disposição da memória RAM. 123
4.19 Introdução de um novo bloco na RAM. 125
4.20 Disposição dos canais no simulador. 130

5.1 Desenvolvimento e compilação de programas para o SimuladorUE. 139
5.2 Terminal VMM do simuladorUE. 140
5.3 Funções do SimuladorUE. 141
5.4 Execução do exemplo 1. 144
5.5 Validação dos resultados. 145
5.6 Execução do exemplo2. 148
5.7 Registos genéricos. 149
5.8 Registos internos do coprocessador zero. 150
5.9 Excepção Machine Check. 152
5.10 Excepção Address Error. 153
5.11 Excepção TLBL (TLB Refill). 154
5.12 Excepção Coprocessor Unusable. 155
5.13 Excepção System Call. 156
5.14 Registos genéricos. 159
5.15 Execução do exemplo 4. 160

6.1 Tradução binária dinãmica dos simuladores QEMU e SimuladorUE. . . . 166

Lista de Tabelas

2.1 Tabela com tipos de architectura definidos para o formato ELF. 20

4.1 Codificação para a instrução ADDI. 45
4.2 Codificação para a instrução ADD. 45
4.3 Codificação para a instrução MFC0. 46
4.4 Codificação para a instrução TLBR. 46
4.5 Tabela de codificação OPCODE. 47
4.6 Tabela de codificação SPECIAL. 47
4.7 Tabela de codificação SPECIAL2. 48
4.8 Tabela de codificação RGIMM. 48
4.9 Tabela de codificação SPECIAL3. 48
4.10 Tabela de codificação COP0 (rs). 49
4.11 Tabela de codificação COP0. 49
4.12 Representação interna das instruções do conjunto *opcode*. 51
4.13 Tabela com registos genéricos do coprocessador zero. 57
4.14 Tabela com registos internos do coprocessador zero. 60
4.15 Registo interno UserLocal. 60
4.16 Registo interno HWREna. 61
4.17 Registo interno BadVAddr. 61
4.18 Registo interno Count. 61
4.19 Registo interno Compare. 62

4.20 Registo interno Status. 62
4.21 Registo interno Cause. 64
4.22 Tabela de códigos por execpão para a arquitectura MIPS32. 66
4.23 Registo interno EPC. 66
4.24 Registo interno PRId. 67
4.25 Registo interno EBase. 68
4.26 Registo interno Config. 68
4.27 Tabela de atributos do registo Config0. 68
4.28 Registo interno Config1. 69
4.29 Tabela de atributos do registo Config1. 69
4.30 Registo interno Config2. 69
4.31 Registo interno ErrorEPC. 70
4.32 Quadro de excepções síncronas e assíncronas com respectivas prioridades. (1 maior prioridade, 5 menor prioridade) 73
4.33 Cálculo do endereço base para os pontos de entrada. 73
4.34 Quadro com deslocamentos utilizados no cálculo de pontos de entrada. . . 74
4.35 Cálculo do endereço base para os pontos de entrada. 75
4.36 Estados modificados pela excepção Reset. 79
4.37 Estados modificados pela excepção Soft Reset. 80
4.38 Estados modificados pela excepção NMI. 81
4.39 Estados modificados pela excepção Machine Check. 81
4.40 Estados modificados pela excepção Address Error. 82
4.41 Estados modificados pela excepção TLB Refill. 82
4.42 Estados modificados pela excepção TLB Execute-Inhibit. 83
4.43 Estados modificados pela excepção TLB Read-Inhibit. 83
4.44 Estados modificados pela excepção TLB Invalid. 83
4.45 Estados modificados pela excepção TLB Modified. 84
4.46 Estados modificados pela excepção Cache. 84
4.47 Estados modificados pela excepção Bus Error. 85
4.48 Estados modificados pela excepção Integer Overflow. 85
4.49 Estados modificados pela excepção Trap. 85

4.50 Estados modificados pela excepção System Call. 85
4.51 Estados modificados pela excepção Break Point. 86
4.52 Estados modificados pela excepção Reserved Instruction. 86
4.53 Estados modificados pela excepção Coprocessor Unusable. 86
4.54 Estados modificados pela excepção Floating Point. 87
4.55 Estados modificados pela excepção do Coprocessor 2. 87
4.56 Estados modificados pela excepção Watch. 87
4.57 Estados modificados pela excepção Interrupt. 88
4.58 Zonas de endereçamento na memória virtual. 94
4.59 Zonas de endereçamento em função do modo de operação do processador. 95
4.60 Registo interno Index. 105
4.61 Registo interno Random. 105
4.62 Registo interno EntryLo0 e EntryLo1. 106
4.63 Especificação dos bits do registo *EntryLo0 e 1*. 106
4.64 Registo interno Context. 107
4.65 Registo interno Pagemask. 108
4.66 Codificação do tamanho das páginas. 108
4.67 Registo interno Wired. 108
4.68 Registo interno EntryHi. 109
4.69 Exemplo de mapeamento de páginas virtuais para físicas. 110
4.70 Exemplo de mapeamentos com páginas físicas partilhadas. 112
4.71 Registos do coprocessador zero salvaguardados quando ocorre a excepção TLB Refill. 114
4.72 Pontos de entrada para a excepção TLB Refill. 114
4.73 Registos do coprocessador zero salvaguardados quando ocorre a excepção TLB Modified. 115
4.74 Registos do coprocessador zero salvaguardados quando ocorre a excepção TLB Invalid. 116
4.75 Tabela de geração de endereços físicos em função do tamanho de página. . 120
4.76 Tabela de mapeamento de endereços fisícos para dispositivos. 131
4.77 Codificação para a instrução END. 135

Capítulo 1

Introdução

Com o primeiro capítulo pretende-se fazer uma introdução sobre o âmbito do trabalho e o contexto onde este se insere na temática do processo de simulação de sistemas de informação. O enquadramento e a motivação para o trabalho são apresentados na secção 1.1 e os objectivos definidos para esta dissertação são enumerados na secção 1.2.

1.1 Enquadramento e Motivação

Em 1981, John Hennessy na Universidade de Standford, começou a trabalhar no que se tornou no primeiro processador MIPS. O conceito inicial foi de aumentar drasticamente o desempenho do processador utilizando pipelines de instruções, uma técnica já conhecida na altura mas difícil de implementar.

Naquela época, as arquitecturas seguiam o desenho da arquitectura CISC (*Complex Instruction Set Computer*), um desenho de instruções complexas cuja execução completa demorava vários ciclos de relógio, o que implicava que o processador perdesse muito tempo à espera que uma instrução completasse a sua execução para poder iniciar a execução da instrução seguinte. Se uma instrução necessitasse de aceder à memória, o processador não fazia nada senão esperar que o acesso à memória terminasse para a acabar a instrução, deixando subaproveitados os recursos de hardware existentes no processador, que poderiam entretanto ser reaproveitados para outras tarefas.

Da necessidade de aumentar o desempenho do processador nasceu a arquitectura RISC (*Reduced Instruction Set Computer*) que utiliza um conjunto de instruções mais simples e uniformes. As instruções perderam riqueza na quantidade e complexidade de operações possíveis de fazer em cada instrução, sendo que as mesmas demoram aproximadamente 1 ciclo do relógio do processador. Com a utilização dos pipelines os processadores tiveram um aumento visível do seu desempenho uma vez que reutilizavam os tempos mortos do processador.

Actualmente existem várias arquitecturas de computadores, sendo as mais conhecidas: Intel x86, x64 ou x86-64, PowerPc, SPARC, MIPS, Alpha e ARM, entre muitas outras. Estas arquitecturas baseiam-se nas antecessoras RISC e CISC, das quais evoluíram e adicionaram novas extensões mantendo a sua base inalterada.

Chama-se a atenção para o facto de que as arquitecturas existentes poderem não funcionar no mesmo ramo, isto é, algumas arquitecturas acima referidas são utilizadas nos conhecidos computadores de mesa, também designados como “desktops”, com sistemas operativos complexos, em microcontroladores ou em sistemas embebidos que podem ou não utilizar sistemas operativos. As arquitecturas Intel x86, x64, PowerPc e SPARC são conhecidas por serem utilizadas em computadores de mesa, cujos famosos sistemas operativos, Windows, Unix, Linux, Macintosh, Solaris, AIX, entre outros, assentam. Por outro lado, na área dos microcontroladores e sistemas embebidos, as arquitecturas líder são ARM, MIPS, Atmel AVR, entre outras, as quais permitem a programação de sistemas mais simples de forma a correrem directamente sobre o *hardware*.

Nos casos em que o *hardware* é bastante limitado, é comum não haver sistemas operativos ou aplicações complexas para gestão dos recursos, mas sim aplicações relativamente simples “flashadas” na memória que correm muitas vezes inteiramente em modo privilegiado, também conhecido por modo *kernel*, como abordado mais à frente. Seguem-se exemplos dos casos acima citados, nomeadamente: as calculadoras, videojogos como a Playstation 2, periféricos, routers, microcontroladores, protótipos, etc. Além dos referidos, existe outra área, actualmente emergente no mercado, designadamente, dispositivos móveis como os telemóveis, que utilizam sistemas operativos embutidos, tais como o Android da Google, Windows Mobile, Windows CE ou o Mobile Linux.

Com a evolução dos tempos nasceu a necessidade de simular o funcionamento de uma arquitectura dentro de outra, conhecida como virtualização de sistemas. Esta técnica consistia na criação de aplicações que simulavam o comportamento de dispositivos físicos, possibilitando a execução de código binário gerado para uma arquitectura específica onde o sistema nativo é apelidado de “anfitrião” e o sistema simulado é conhecido por “hóspede”.

A simples simulação do *hardware* por si só não possibilita a instalação de sistemas operativos complexos numa máquina virtual, como se verá mais à frente no presente trabalho. Para tal, além da simulação do *hardware*, é necessário implementar vários mecanismos de cada arquitectura, com desenhos e implementações específicas de cada uma, tais como o mecanismo de excepções ou o mecanismo de interrupções, utilizados para possibilitar a comunicação entre o processador e os periféricos ligados físicamente, como o monitor, rato ou teclado.

1.2 Objectivos

A presente dissertação tem como objectivo a criação de um simulador para a arquitectura MIPS de 32 bits (MIPS32), que possibilite a execução de código binário, desenvolvido especificamente para a referida arquitectura, como se de um *hardware* real se tratasse, tais como programas simples ou sistemas operativos complexos, como o Linux.

O simulador deverá implementar todas as instruções disponibilizadas na ISA MIPS32 (*Instruction Set Architecture*) revisão 2. Para que a aplicação possa correr programas complexos, será necessário a simulação de alguns componentes físicos necessários à sua execução, tais como: um processador genérico MIPS, memória RAM, memória ROM, unidade de gestão de memória MMU (*Memory Management Unit*), mecanismo de tradução de endereços TLB (*Translation Lookaside Buffer*) e memória permanente como o disco rígido.

Todos estes componentes serão implementados de uma forma genérica, sendo objectivo a criação de um sistema modularizado com um nível de compatibilidade com a arquitectura MIPS que permita a comutação dos módulos existentes por outros com implementações específicas. Relativamente à gestão da memória, física e virtual, não será objectivo a demonstração da eficiência ou desempenho de algoritmos de gestão de memória, como o LRU (*Least Recently Used*), FIFO (*First In First Out*), LIFO (*Last In First Out*) ou aleatório (*Random*), mas sim a demonstração dos procedimentos e a forma como são executados no sistema, tal como o *Page Fault* ou o *TLB Miss* que é uma das excepções que mais vezes ocorre num sistema operativo real.

Como objectivo seguinte será a explanação e o desenvolvimento de um *kernel* minimalista que dê suporte a alguns mecanismos e funcionalidades, mormente, a gestão de processos e memória, tanto física como virtual. A gestão de processos e seus respectivos contextos serão executados numa escala reduzida, uma vez que esta depende unicamente do refinamento do sistema operativo em si e não do simulador ou da arquitectura em questão.

Capítulo 2

Estado da Arte e Conceitos Envolvidos

De forma a contextualizar o presente trabalho, é importante introduzir noções e conceitos para uma melhor compreensão do mesmo.

Na secção 2.1 abordaremos o estado da arte actual na área da virtualização de sistemas, nomeadamente os vários tipos de virtualização e simulação existentes. Serão igualmente abordadas algumas técnicas de optimização utilizadas no processo de simulação em 2.2.

A título introdutório, veremos na secção 2.3 os conceitos IPC *(Interprocess Comunication)* e a representação e extracção de informação de ficheiros binários ELF em 2.4, dois conceitos importantes no desenvolvimento da presente tese.

2.1 Virtualização de Sistemas

A virtualização de sistemas encontra-se actualmente em forte expansão. Esta consiste na criação de um ambiente virtual, permitindo a utilização de vários sistemas operativos e aplicações compilados para uma arquitectura diferente da arquitectura nativa.
O seu emprego acarreta inúmeras vantagens para as empresas não só a nível económico, mas em termos de flexibilidade, escalabilidade e variedade de ambientes possíveis de criar, visto que existe actualmente uma necessidade de diminuir o desperdício de recursos, e

esta constitui uma alternativa viável.

Através da virtualização de sistemas é possível criar, utilizar e descartar ambientes de desenvolvimento, testes ou produção muito rapidamente sem a necessidade de aquisição e manutenção de *hardware* novo. É igualmente possível criar uma grande variedade de ambientes e cenários, que os programadores podem utilizar para melhorar o seu trabalho. O seu funcionamento consiste na criação de uma camada *middleware*, que emulará um conjunto de *hardware* de forma a suportar a execução de sistemas operativos, independentemente da arquitectura nativa.

Uma solução de virtualização é composta, essencialmente, por dois actores: o anfitrião (*host*) e o hóspede (*guest*). Podemos entender o anfitrião como sendo o sistema operativo que é executado nativamente por uma máquina física e o hóspede, por sua vez, é o sistema virtualizado que é executado dentro do anfitrião.

Associado à técnica de virtualização encontra-se o conceito de *hypervisor* ou *Virtual Machine Monitor* (VMM), que consiste numa aplicação que permite a execução concorrente de várias máquinas virtuais no mesmo sistema, sendo a mesma compilada para uma arquitectura em particular. O *hypervisor* é responsável pela criação, isolamento e preservação do estado das máquinas virtuais, assim como da gestão dos acessos aos recursos do sistema. Existem no entanto dois tipos de VMM, podendo estes ser:

Tipo 1 Os *hypervisors* do tipo um são geralmente utilizados em servidores e assentam directamente sobre o *hardware* físico, sem utilização de um sistema operativo. Desta forma, toda a gestão e manutenção dos recursos físicos é feito exclusivamente pelo *hypervisor*. Na imagem abaixo podemos ver as camadas existentes entre os sistemas virtualizados, hóspedes e o *hardware* físico.

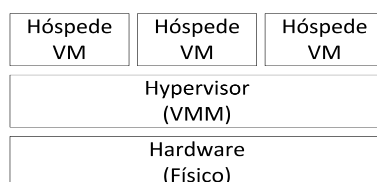


Figura 2.1: Hypervisor tipo 1.

Um exemplo de *hypervisors* modernos que utilizem este modelo são: Oracle VM Server for SPARC, Citrix XenServer, KVM, VMware ESX/ ESXi ou Microsoft Hyper-V.

Tipo 2 Os *hypervisors* do tipo dois, em oposição aos do tipo um, consistem numa aplicação que corre sobre o sistema operativo e não no *hardware* físico. Como podemos ver na figura abaixo, nestes casos os sistemas virtualizados correm na terceira camada de *software* do sistema.

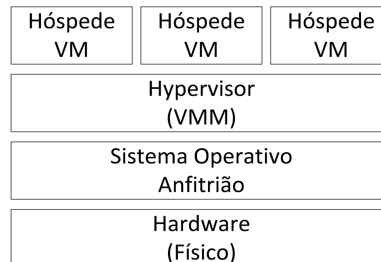


Figura 2.2: Hypervisor tipo 2.

Tanto o VMWare como Oracle Virtual Box, são dois exemplos actuais de *hypervisors* do tipo dois, correndo sobre sistemas operativos como o Windows ou Linux.

No processo de criação de máquinas virtuais, são empregues técnicas como a virtualização por *software* ou virtualização por *hardware*, de forma a conseguirmos executar um sistema operativo hóspede.
No primeiro caso, é feita a simulação do comportamento dos periféricos e dos mecanismos associados, podendo em alguns casos permitir o acesso directo aos periféricos físicos, como veremos em seguida. No segundo caso, o *hardware* físico dispõe de mecanismos que permitem o auxílio e aceleração de instruções associadas à virtualização de sistemas.
A virtualização assistida por *software* categoriza-se em três subgrupos de virtualização, nomeadamente:

Emulação

A emulação é uma técnica utilizada pela virtualização, cujo objectivo é possibilitar a comunicação entre dois sistemas originalmente distintos e incompatíveis. Esta incompatibilidade pode ser do tipo *hardware* para *software* ou *software* para *software*.

Assim, um emulador implementa todas as instruções realizadas pela máquina real em um ambiente abstracto de *software*, possibilitando a execução de aplicações destinadas a uma plataforma específica sem modificações. Um exemplo deste comportamento é a possibilidade de correr aplicações desenvolvidas para a arquitectura MIPS numa máquina com a arquitectura x86.

No quadro abaixo, podemos ver a representação das camadas necessárias no processo de emulação de sistemas:

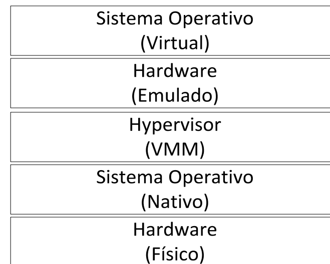


Figura 2.3: Camadas da emulação de sistemas.

O processo de criação de um emulador é bastante complexo, uma vez que necessita simular todas as instruções do processador, assim como suas características internas, como chips ou circuitos integrados do *hardware* que o constitui.
É igualmente necessário simular todos os mecanismos associados aos periféricos como: a implementação do conjunto instruções do processador (ISA), memória principal RAM, unidade de gestão de memória (MMU), mecanismo de tradução de endereços, mecanismos de excepções e interrupções do processador, interface de rede, dispositivos de memória permanente (como os discos rígidos) e acesso a diversos dispositivos.

Comparativamente aos restantes modelos de virtualização, a técnica de emulação é a mais antiga e possui o desempenho mais baixo, uma vez que a simulação dos componentes é muito pesada e minuciosa. Cada instrução do sistema hóspede é interpretada e optimizada pelo simulador, gerando assim um bloco de código (conjunto de instruções) a ser executado na arquitectura do anfitrião.

A grande desvantagem prende-se ao facto de, para se ter um desempenho satisfatório, o sistema anfitrião deve possuir um desempenho ou capacidade superior ao hóspede. Actualmente existem vários tipos de emuladores de sistemas, entre os quais se destacam os emuladores QEMU, Bochs, PearPC e GXemul.

Virtualização Completa

Através da técnica de virtualização completa, também conhecida como virtualização nativa, é possível executar *software* desenvolvido sem modificações, à semelhança do que acontece na emulação. A grande diferença relativamente à técnica anterior é o facto de a arquitectura do hóspede ser a mesma que a arquitectura do anfitrião.

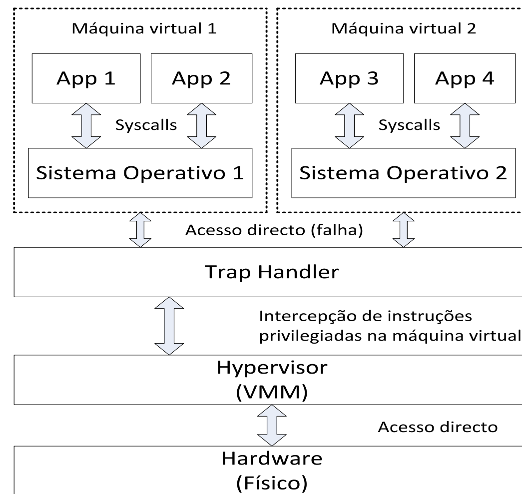


Figura 2.4: Camadas de virtualização de sistemas completa ou nativa.

Nesta solução é necessário simular os componentes físicos da máquina, de forma a permitir a execução de um sistema operativo, sendo geralmente simulados componentes genéricos. Associado à simulação dos componentes físicos, encontra-se a implementação do seguinte conjunto de mecanismos: instruções do processador (ISA), memória principal RAM, unidade de gestão de memória (MMU), mecanismo de tradução de endereços, mecanismos de excepções e interrupções do processador, interface de rede, dispositivos de memória permanente como os discos rígidos e acesso a diversos dispositivos.

Este tipo de virtualização emprega a técnica de tradução binária, no qual o *hypervisor* é responsável por analisar, reorganizar e traduzir blocos de instruções binárias do sistema operativo hóspede. A análise das instruções é feita em tempo de execução (*"on-the-fly"*), a fim de adaptar as instruções geradas à ISA do sistema nativo, nos casos em que não sejam idênticas. É igualmente seu objectivo a optimização das sequências de código enviadas pelo hóspede, com o objectivo de melhorar o desempenho da execução.

É importante referir que o código não privilegiado corre nativamente no *hardware*, sendo o mesmo encaminhado pelo *hypervisor*. Na figura 2.4, podemos ver o comportamento do sistema relativamente à execução de código privilegiado, nomeadamente chamadas ao sistema (*System Calls*).

Todas as instruções privilegiadas são interceptadas pelo *hypervisor*, uma vez que a sua execução directa no sistema hóspede originaria um comportamento imprevisível gerando uma falha tanto no isolamento da máquina virtual, como na segurança do próprio sistema anfitrião. Assim, a categoria de instruções é analisada e tratada pelo *hypervisor*.

Contrariamente à emulação de *hardware*, obtém-se um maior desempenho uma vez que não existe a necessidade de recompilação de código[1], uma vez que a grande maioria das instruções corre nativamente. No entanto, é de notar que a utilização da técnica de tradução binária introduz uma redução no desempenho do *hypervisor* comparativamente com a execução nativa do código binário.

Paravirtualização

A Paravirtualização é uma técnica de virtualização de segunda geração, cujo modo de funcionamento difere das anteriores. Em oposição ao que acontece nos exemplos anteriores, a paravirtualização requer a modificação do código fonte do sistema operativo do hóspede, com o objectivo de substituir instruções não virtualizadas, que comunicam directamente com a camada de virtualização, conhecidas como *hypercalls*.

O *hypervisor* disponibiliza então uma interface de comunicação para operações críticas de *kernel*, como a gestão de memória, controlo sobre o mecanismo de interrupções, entre outros mecanismos, através de uma *Application Programming Interface* (API).

Semelhantemente ao que acontece com a virtualização total, na paravirtualização, o *hypervisor* é responsável pela captura e emulação de algumas operações criticas nas máquinas virtuais, sendo que as restantes correm directamente no *hardware* da máquina de forma controlada pelo *hypervisor*.

Como podemos ver na figura 2.5, cada máquina virtual, em modo não privilegiado dispõe de acesso directo ao *hardware* real. Relativamente a instruções privilegiadas, o *hypervisor* é responsável pela interceptação e execução das mesmas.

Desta forma, a paravirtualização não suporta sistemas operativos não modificados, tais como os sistemas operativos da Microsoft[2], tendo assim, desvantagens de compatibilidade e portabilidade. Relativamente ao desempenho, a paravirtualização possui melhor desempenho que as duas ténicas anteriores, uma vez que o conjunto

---

[1] A recompilação significa que um instrução na arquitectura do sistema hóspede gerará um bloco de instruções para a arquitectura nativa.

[2] Inicialmente a Microsoft não permitia a modificação do código fonte dos seus sistemas operativos, sendo mais tarde permitida a modificação em alguns sistemas operativos, como o Windows XP.

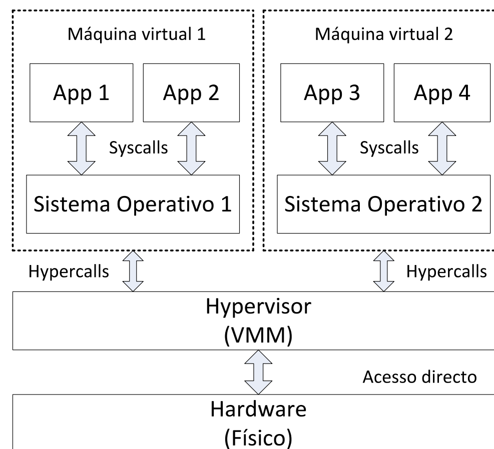


Figura 2.5: Camadas em sistemas paravirtualizados.

de instruções interceptadas pelo o *hypervisor* é menor, assim como a execução e o acesso aos dispositivos físicos é feito nativamente.

Além dos modelos anteriores, existe ainda a virtualização assistida por *hardware*, no qual o processador dispõe de mecanismos físicos para auxiliar a virtualização.

Um exemplo deste comportamento são os processadores Intel e AMD, para a arquitectura x86, que criaram mecanismos nos seus processadores, nomeadamente o Intel VT-x/VT-i e o AMD-v, para suportar e optimizar a virtualização de sistemas. O objectivo principal da utilização desta técnica é o aumento do desempenho da execução das máquinas virtuais, em sistemas que utilizem virtualização total[3] ou nativa.

Vejamos como exemplo o mecanismo utilizado pela Intel VT-x. De forma superficial e segundo [Mat02] e [WGP], esta técnica consiste na criação de dois modos de execução no processador, nomeadamente *root* e *non-root*. O primeiro é destinado à execução do sistema operativo ou *hypervisor*, equivalendo ao funcionamento de um processador tradicional. O segundo modo, *non-root*, é utilizado exclusivamente para as máquinas virtuais.

Na arquitectura x86 existem quatro níveis de privilégio de execução conhecidos como anéis (*rings*) e variando do zero até três, sendo o zero o nível mais privilegiado. Em cada modo de execução do processador, encontram-se implementados os quatro anéis de privilégios permitindo a execução de sistemas operativos no nível zero.

Para gerir a mudança de contexto no processador, é criada uma estrutura conhecida como

[3]Neste tipo de execução os sistemas anfitriões e hóspedes possuem a mesma arquitectura.

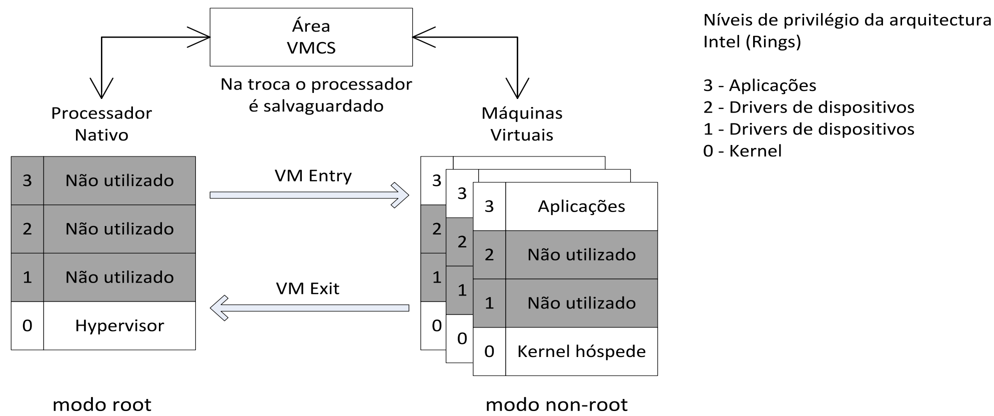


Figura 2.6: Arquitectura Intel VT-x/ VT-i.

Virtual Machine Control Structure (VMCS), que contém essencialmente duas áreas: uma para o sistema operativo hóspede e outra para o *hypervisor*.

Sempre que exista uma transição "VM entry"[4], o estado do processador é salvaguardado na área do *hypervisor* da VMCS, sendo em seguida actualizado o estado do processador com a informação proveniente da área do sistema operativo hóspede.

Quando é executada a transição inversa, "VM exit"[5], o estado do processador do sistema operativo virtual é guardado, sendo em seguida restaurado o estado do processador do sistema operativo nativo (*hypervisor*).

No procedimento "VM entry", é efectuada uma passagem do modo *root* para o modo *non-root*, assim como no procedimento inverso "VM exit", é efectuada uma passagem do modo *non-root* para o modo *root*, como podemos ver na figura 2.6. Desta forma, o sistema operativo virtual pode correr directamente no processador sendo apenas as instruções mais sensíveis, como as interrupções no sistema hóspede, que geram a transição "VM exit" passando a execução para o modo *root*, *hypervisor*, que em seguida executará o procedimento adequado.

[4]VM Entry corresponde à transição do processador nativo (VMM) para o processador virtualizado (sistema hóspede).

[5]VM Exit corresponde à transição do processador virtualizado (sistema hóspede) para o processador nativo (VMM).

Como referido anteriormente, a virtualização total assim como a paravirtualização, requerem que tanto o hóspede como o anfitrião utilizem a mesma arquitectura, uma vez que em ambos os casos os sistemas operativos acedem directamente aos componentes físicos. Um exemplo de sistemas que utilizam este tipo de técnica é o VMWare Workstation ou Server e o VirtualBox. É de notar que em ambos os casos, para a arquitectura x86, estes suportam a utilização de virtualização assistida por *software* em processadores Intel ou AMD, nomeadamente Intel VT-x/VT-i e AMD-v. Já na área da paravirtualização, onde existe a necessidade de modificação do sistema operativo hóspede, existem os seguintes *softwares*: Microsoft Hyper-V, VMWare e Xen.

Por outro lado, temos o processo de emulação de sistemas, no qual não existe a necessidade da arquitectura do hóspede e do anfitrião serem iguais.

2.2 Técnicas de tradução e compilação de instruções

Associado à temática da virtualização de sistemas, encontra-se um conjunto de técnicas utilizadas com o objectivo de aumentar o seu desempenho. Um poderoso exemplo disto é a técnica de tradução binária, no qual muitos *softwares* de virtualização e emulação se baseiam para a conversão de código binário.

Esta conversão consiste na tradução de um conjunto de instruções binárias em outro conjunto de instruções, nomeadamente transformar o código da arquitectura simulada em código da arquitectura nativa do sistema. Existem no entanto dois tipos de tradução binária: tradução binária estática e tradução binária dinâmica.

No primeiro caso, o *software* responsável pela tradução, tentará proceder à conversão total do código binário existente no ficheiro executável para o formato da arquitectura nativa do sistema. Esta operação nem sempre é possível de se efectuar, uma vez que existem limitações na utilização desta técnica, dado que podem existir referências no código cujo valor apenas é conhecido em tempo de execução. Um exemplo deste comportamento são as instruções de saltos incondicionais na arquitectura MIPS, como o `JR $ra`.

Na tradução dinâmica, o funcionamento muda totalmente, uma vez que apenas blocos de código serão traduzidos e mantidos numa memória *cache*. O código binário vai sendo traduzido à medida que é necessário. Em sistemas que utilizem este tipo de técnica, os primeiros momentos de execução demoram mais tempo a executar, uma vez que é necessário traduzir o código, sendo os seus ganhos reflectidos sempre que é necessário executar código que já tenha sido previamente traduzido, e se encontre na memória *cache*. Em sistemas emulados, geralmente, o processo de tradução é bastante simples,

consistindo num ciclo com três operações: leitura, descodificação e execução do código.

A técnica de tradução binária dinâmica é geralmente utilizada em sistemas que dispõe de uma representação intermédia, conhecida também como *bytecode*. Linguagens como o Java ou .NET utilizam actualmente técnicas de compilação e tradução dinâmica, conhecidas como JIT (*Just-In-Time*), com o objectivo de optimizar a execução do código[6].

O código fonte da linguagem Java é compilado, através de um compilador, normalmente `javac`, e gerará um ficheiro ".class" conhecido como *bytecode*, cujo o conteúdo consiste numa representação do código fonte, independente da arquitectura para o qual se destina.

Em seguida, a máquina virtual do java (JVM) no processo de execução, é responsável pela recompilação do código *bytecode* para o código binário da arquitectura do sistema, como podemos ver na figura 2.7.

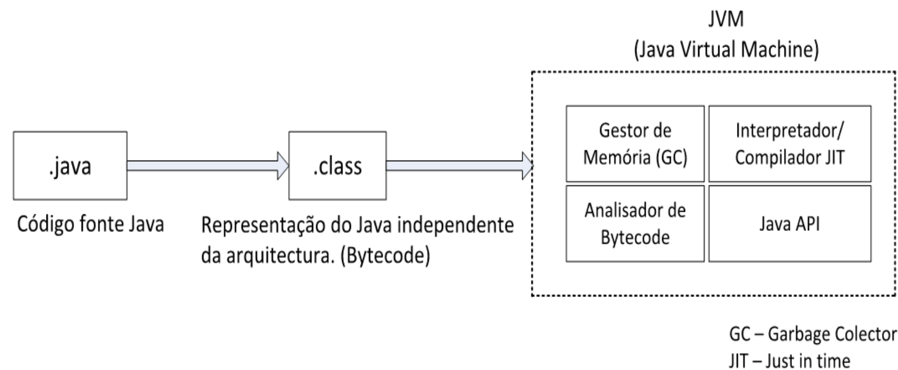


Figura 2.7: Estrutura de compilação para a linguagem Java.

No entanto, compiladores recentes para a linguagem Java dispõem da técnica JIT para optimizarem a execução do código binário nas máquinas virtuais. Este tipo de compiladores, em oposição à compilação estática, dispõe de um processo semelhante ao que acontece na tradução dinâmica. Apenas alguns blocos de código do *bytecode* serão optimizados e compilados em tempo de execução. A escolha dos blocos de código segue uma política de demanda, isto é, apenas os blocos de códigos pedidos serão compilados. É de notar que o arranque dos programas compilados com esta técnica, é mais lento que nos programas compilados estaticamente, no entanto os seus benefícios são reflectidos na execução consecutiva de blocos de código.

[6]Apenas linguagens que tirem partido de *bytecode*, como o Java ou .Net, utilizam compiladores JIT. As restantes linguagens de programação como o C/C++, que não permitem compilação em runtime, dispôem de técnicas semelhantes como o LLVM (*Low Level Virtual Machine*), que não é abordado no presente trabalho.

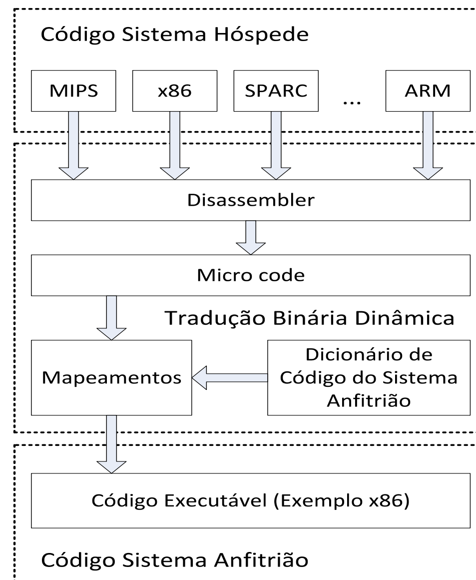


Figura 2.8: Arquitectura do emulador QEMU.

O emulador QEMU, que actualmente permite a virtualização de várias arquitecturas, entre elas a arquitectura MIPS, utiliza a técnica de tradução dinâmica para a virtualização dos sistemas. Esta consiste na criação de uma representação intermédia, conhecida como *Micro-Operations* de forma muito semelhante com o que acontece com o *bytecode* do Java.
Na figura 2.8 observa-se a forma como o emulador QEMU utiliza a tradução binária dinâmica no processo de emulação das arquitecturas.

Como se pode ver o emulador QEMU desmonta e recompila blocos de código binário do sistema hóspede de forma a gerar uma representação intermédia, as *Micro-Operations*. Em seguida, é efectuada a recompilação das *Micro-Operations*, na qual será gerado o código binário para a arquitectura nativa do sistema.

2.3 Comunicação entre processos (IPC)

É comum em sistemas elaborados a criação e utilização de vários processos, dependentes ou independentes entre si, que por sua vez necessitam de comunicar. Num sistema modularizado e distribuído, os vários módulos constituintes do sistema podem funcionar de forma independente, gerando assim uma melhor distribuição de carga e gerando uma maior robustez. Com tais características nasce uma importante questão:

"Como é que os processos irão comunicar entre si?"

A forma mais conhecida de resolver esta questão em ambientes Linux, é através do mecanismo de comunicação entre processos IPC *(InterProcess Communication)*. O IPC consiste num conjunto de técnicas e mecanismos no qual é possível a comunicação entre processos parentes[7], não parentes[8] ou existentes em máquinas diferentes. Do conjunto de técnicas disponibilizadas para IPC sobressaem as seguintes: comunicação por *Network Sockets*, memória partilhada *(Shared Memory)*, *Named Pipes* ou *Unnamed Pipes*, mecanismos de mensagens (Message Queue), pseudoterminais, invocações remotas RPC *(Remote Procedure Calls)*, ficheiros especiais, entre outros.

A escolha do mecanismo mais adequado varia em função das necessidades, podendo esta ser apenas a partilha de informação entre os processos ou *threads*, o aumento do desempenho na transmissão de dados, modularidade, simples conveniência ou a separação de privilégios para aumento da segurança.
Em tais sistemas é necessário que exista um conjunto rigoroso de regras de comunicação e integração de forma a possibilitar a troca de mensagens correctamente.

Em seguida observar-se-á a forma como os processos comunicam entre si utilizando exclusivamente pipes do sistema *Unix*.

2.3.1 Named e Unnamed Pipes

Originalmente os pipes em sistemas *Unix* foram desenhados de forma a permitir o funcionamento encadeado de processos, possibilitando que os mesmos possam comunicar entre si, redireccionando assim o fluxo de informação provenientes dos três principais descritores: `STDIN` (*Standard Input = 0*), `STDOUT` (*Standard Output = 1*) e `STDERR` (*Standard Error = 2*). Actualmente existem dois tipos de *pipes*, nomeadamente, *Named Pipes* e *Unnamed Pipes*.

O seu funcionamento é igual tirando o facto do primeiro criar um dispositivo de co-

[7] Processos gerados através de forks ou exec.
[8] Processos que não possuem qualquer nível de parentesco, funcionam apenas no mesmo sistema.

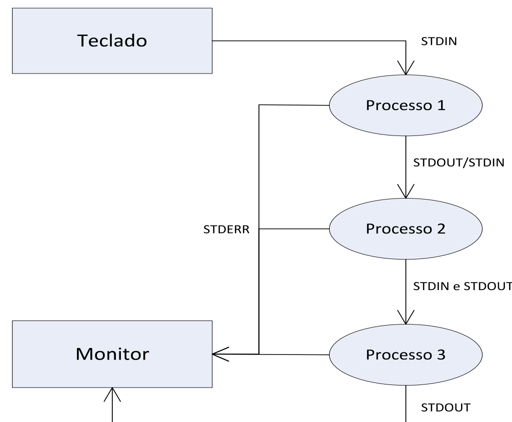


Figura 2.9: Comunicação com Pipelines Unix.

municação de forma permanente no sistema operativo, no qual deverá ser destruído manualmente após a sua utilização. Nos casos *Named Pipe* é necessário criar o *pipe* manualmente através do programa `mkfifo`, antes da execução do processo, de forma a permitir que os processos se possam ligar ao pipe.

No segundo caso, o *pipe* criado existe apenas enquanto o processo se encontrar em execução e será destruído automaticamente quando o mesmo terminar.

Ambas as formas baseiam-se na técnica FIFO (*First In First Out*) para controlar a entrada e saída de informação do canal de comunicação. Ao contrário do que acontece com os pseudoterminais, que criam um canal bidireccional, os *pipes* são canais unidireccionais, sendo necessário a criação de dois *pipes* para que os processos disponham de comunicação bidireccional. Na figura 2.9 podemos ver um esboço do fluxo de comunicação de um conjunto de programas a funcionar utilizando pipelines.

Após a criação de um *pipe* um processo recebe dois descritores, sendo o primeiro o ponto de entrada de informação no canal e o segundo o ponto de saída de informação. Estes descritores não são os descritores *standard* (STDIN, STDOUT e STDERR) criados por defeito para os processos, mas sim identificadores dos canais.

Toda a informação transmitida para o *pipe* é guardada num *buffer*, cujo tamanho e

gestão são controlados pelo sistema operativo. É responsabilidade do sistema operativo garantir que a ordem pela qual a informação é introduzida no *pipe* é exactamente a mesma ordem de saída e que a informação enviada é recebida pelo destinatário.

Para a criação e utilização dos *pipes*, o sistema operativo *Unix* disponibiliza um conjunto de chamadas ao sistema, sendo os mais importantes: *pipe*, *read*, *write* e *close*.

2.4 Executable and Linkable Format (ELF)

Os ficheiros executáveis contêm um conjunto de informação, dados e código binário a ser executado, numa representação específica. Actualmente existem várias representações utilizadas sendo as mais comuns: ELF, COFF, a.out, Mach-O e PE/COFF.

O formato ELF é uma norma que foi originalmente desenvolvida pelos *Unix System Laboratories*.

Ao contrário do que acontece com muitos formatos proprietários, o formato ELF foi desenhado para ser flexível e extensível de forma a poder abranger vários tipos de processadores, arquitecturas ou sistemas operativos diferentes. Como este formato não se prende a nenhum sistema operativo ou processador em particular, foi rapidamente aceite pelos utilizadores de sistemas Unix como um formato *standard* para ficheiros binários, reduzindo-se o número de implementações de interfaces diferentes e diminuindo a necessidade de recodificação ou recompilação de código.

2.4.1 Organização de ficheiros ELF

Os ficheiros binários ELF organizam a informação num conjunto de secções como ilustra a figura 2.10. Oficialmente, existem dois tipos de visões sobre os ficheiros ELF, nomeadamente os ficheiros realocáveis e os ficheiros executáveis. No primeiro caso, estes possuem informação adicional utilizável pelo compilador no processo de "linkagem".

No inicio de cada ficheiro ELF existe um cabeçalho ELF, que possui informação relativa ao ambiente de execução, arquitectura, ponto de entrada na memória virtual e tipo de ficheiro, entre outros aspectos.

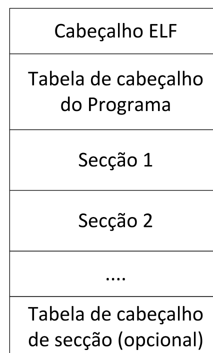


Figura 2.10: Formato de um ficheiro binário ELF.

No bloco de código abaixo (Listing 2.1) pode-se ver a estrutura que representa o cabeçalho ELF.

```
#define EI_NIDENT     16

typedef struct {
        unsigned char   e_ident[EI_NIDENT];
        Elf32_Half      e_type;
        Elf32_Half      e_machine;
        Elf32_Word      e_version;
        Elf32_Addr      e_entry;
        Elf32_Off       e_phoff;
        Elf32_Off       e_shoff;
        Elf32_Word      e_flags;
        Elf32_Half      e_ehsize;
        Elf32_Half      e_phentsize;
        Elf32_Half      e_phnum;
        Elf32_Half      e_shentsize;
        Elf32_Half      e_shnum;
        Elf32_Half      e_shstrndx
}  Elf32_Ehdr;
```

Listing 2.1: Definição da estrutura cabeçalho ELF.

Segue-se uma breve descrição da informação guardada por estas variáveis:

e_ident Esta variável é composta por um *array* de dezasseis bytes independentes da arquitectura, utilizados para interpretar ou descodificar o conteúdo do ficheiro;

e_type Identifica o tipo de ficheiro, podendo este variar entre: *No file type*, *Relocatable file*, *Executable file* ou *Shared Object file*;

e_machine Aqui é guardado o tipo de arquitectura utilizada, variando de acordo com o quadro abaixo:

Nome	**Valor**	**Significado**
EM_NONE	0	No Machine
EM_M32	1	AT&T WE 32100
EM_SPARC	2	SPARC
EM_386	3	intel 80386
EM_68K	4	Motorola 68000
EM_88K	5	Motorola 88000
EM_860	7	Intel 80860
EM_MIPS	8	MIPS RS3000

Tabela 2.1: Tabela com tipos de architectura definidos para o formato ELF.

e_version Identifica a versão do ficheiro;

e_entry Indica o endereço virtual que deve ser utilizado para iniciar a execução do código binário;

e_phoff Guarda o valor do deslocamento da tabela de cabeçalho do programa em bytes. Caso o ficheiro não possua uma tabela de cabeçalho do programa, este campo contém zero;

e_shoff Possui o valor do deslocamento da tabela de cabeçalho das secções em bytes. Caso a tabela não exista, o seu valor é zero;

e_flags Guarda valores a serem utilizados em processadores específicos;

e_ehsize Contém o tamanho do cabeçalho ELF em bytes;

e_phentsize Guarda o tamanho de cada entrada da tabela de cabeçalho do programa em bytes. Todas as entradas possuem um tamanho igual;

e_phnum Guarda o número de entradas na tabela de cabeçalho do programa;

e_shentsize Possui o valor em bytes do tamanho da tabela de cabeçalho das secções;

e_shnum Guarda o número de secções existentes na tabela de cabeçalho das secções;

e_shstrndx Esta variável guarda o índice da tabela de cabeçalho de secção associado ao nome da secção.

Em ficheiros executáveis a tabela *header* do programa é obrigatória uma vez que possui os mapeamentos de secção para segmento que deverão ser carregados para memória. A tabela de cabeçalho de secções é composta por um *array* de estruturas `ELF32_Shdr`, que possui um conjunto de informações relativas a cada secção.

Da mesma forma que a tabela de cabeçalho ELF, a estrutura `ELF32_Shdr` inclui a localização do código binário assim como informações extra do segmento. Segue-se a estrutura que representa a informação de cada secção num ficheiro ELF:

```
typedef struct {
        Elf32_Word      sh_name;
        Elf32_Word      sh_type;
        Elf32_Word      sh_flags;
        Elf32_Addr      sh_addr;
        Elf32_Off       sh_offset;
        Elf32_Word      sh_size;
        Elf32_Word      sh_link;
        Elf32_Word      sh_info;
        Elf32_Word      sh_addralign;
        Elf32_Word      sh_entsize;
}   Elf32_Shdr;
```

Listing 2.2: Definição da estrutura cabeçalho de secção.

Esta estrutura identifica um bloco de dados associado a uma secção e o endereço em que este deve ser carregado em memória, assim como o nome e outros atributos.

sh_name Esta variável guarda o nome da secção;

sh_type Neste campo são definidos o tipo de conteúdo existente na secção. Ver [Sta01] para mais informação;

sh_flags Descreve um conjunto de atributos associados à secção;

sh_addr Guarda o endereço no qual deve ser carregado o primeiro byte da secção;

sh_offset Possui o valor do deslocamento desde o inicio do ficheiro até ao primeiro byte da secção;

sh_size Guarda o tamanho da secção em bytes;

sh_link Guarda uma referência para um índice da tabela de cabeçalho da secção. A sua interpretação depende exclusivamente do tipo de secção;

sh_info Guarda informação extra cuja interpretação depende do tipo de secção;

sh_addralign Indica se a secção deve ser alinhada ou não;

sh_entsize Uma vez que algumas secções possuem uma tabela de símbolos, sendo o tamanho de cada entrada igual, este campo guarda o valor de cada entrada.

Através da estrutura acima citada, é possível aceder a toda a informação relativa a cada secção, assim como a localização de cada atributo dentro do ficheiro ELF.

Para informações detalhadas relativas ao formato ELF ver [Sta01]. Em seguida veremos a forma com o simulador extrai a informação proveniente do ficheiro binário.

2.4.2 Interpretação de ficheiros ELF

O processo de interpretação do ficheiro binário executável, passado como argumento ao simulador, encontra-se dividido em três operações: extrair o cabeçalho ELF, localização da tabela de secções e extracção de conteúdos.

A primeira fase consiste na extracção do cabeçalho ELF localizado no início do ficheiro ELF. No seguinte bloco de código é possivel visualizar a extracção do cabeçalho ELF:

```
Elf32_Ehdr elfEhdr;
Elf32_Shdr *elfShdr;
FILE *targetFile;
char tempBuf[64];
int index, ret = -1;

targetFile = fopen(argv[1], "r");
if(targetFile == NULL){
        printf("File not open!\n");
        exit(-1);
}

if(targetFile){
        /* Ler cabecalho ELF (Endereco zero do ficheiro) */
        fread(&elfEhdr, sizeof(elfEhdr), 1, targetFile);
```

Listing 2.3: Extracção do cabeçalho ELF.

Uma vez executado o bloco de código, o simulador dispõe de acesso aos campos: `e_shoff`, `e_shnum` e `e_shentsize`, que possuem o deslocamento da tabela de cabeçalhos das secções, o número de secções existentes e o tamanho de cada secção, respectivamente.

Assim, através do campo `e_shoff` é possível localizar a tabela de secções, sendo em seguida necessário reservar espaço em memória e modificar o apontador do ficheiro ELF para a localização da tabela de secções como podemos ver no bloco de código abaixo.

```
/* Elf32_Ehdr.e_shnum indica o numero de seccoes existentes
   */
elfShdr = calloc(elfEhdr.e_shnum, sizeof(*elfShdr));
assert(elfShdr);

/* Modificar o apontador do ficheiro para a tabela de
   cabecalho das seccoes */
fseek(targetFile, elfEhdr.e_shoff, SEEK_SET);
fread(elfShdr, sizeof(*elfShdr), elfEhdr.e_shnum, targetFile);
```

Listing 2.4: Modificação do apontador do ficheiro para a tabela de secções.

A última fase consiste em percorrer todas as secções de forma iterativa, uma vez que sabemos o número de entradas na tabela de secções através do `e_shnum`, introduzindo assim o código de cada secção na memória RAM.

Existem no entanto um conjunto de secções especiais, utilizadas especificamente pelo *assembler*, como `.text`, `.bss` ou `.data`. Independentemente do tipo de secção, o simulador processará sem excepções, introduzindo todo o código existente nas secções para a memória do simulador como podemos ver no bloco de código binário abaixo:

```
/*  Percorrer cada seccao */
for(index = 0; (unsigned int) index < elfEhdr.e_shnum; index++){

        /* Se o Elf32_Shdr.sh_addr for diferente de zero, entao a
           seccao deve ser carregada para memoria */
        if(elfShdr[index].sh_addr){

        /* Modificar o apontador do ficheiro para o nome da seccao e
           ler o valor */
        fseek(targetFile, elfShdr[elfEhdr.e_shstrndx].sh_offset +
           elfShdr[index].sh_name, SEEK_SET);

        int j = 0;    //Contador
        int word;     //Instrucao

        fseek(targetFile, elfShdr[index].sh_offset, SEEK_SET);

        for(j=0; j< (elfShdr[index].sh_size)/4; j++){
                //Ler codigo binario da seccao para a memoria RAM
                fread(&word, sizeof(WORD), 1, targetFile);
```

```
                StoreMemory (cpu, sizeof(WORD), ((j)*4)+elfShdr[index
                    ].sh_addr, word);
        } //End FOR de codigo binario
        } //End IF de elfShdr[index].sh_addr
} //End FOR percorrer seccoes

fclose(targetFile);
free(elfShdr);
```

Listing 2.5: Extracção de informação das secções.

Em cada iteração do ciclo, é necessário validar se a secção será carregada para a memória através do valor existente em `elfShdr[index].sh_addr`, uma vez que este identifica o endereço base da secção.

Em seguida é necessário gerar um novo ciclo de forma a percorrer a informação binária da secção, apontada pelo o endereço existente em `elfShdr[index].sh_offset`, introduzindo assim a informação na memória RAM do simulador.

Capítulo 3

Sistema Proposto

Neste capítulo será feita uma descrição geral do sistema proposto assim como as metodologias utilizadas.

A secção 3.1 descreve o que se pretende com o trabalho em 3.1.1 e qual a metodologia a seguir na secção 3.1.2. A arquitectura é exposta na secção 3.2, a apresentação dos módulos do sistema na secção 3.2.1 e a plataforma em que o trabalho foi desenvolvido em 3.2.2.

Os capítulos seguintes expõem detalhadamente os procedimentos adoptados em cada fase do trabalho.

3.1 Apresentação

Cada vez mais a virtualização de sistemas detém um papel fundamental no funcionamento de qualquer empresa. Os seus benefícios são inúmeros sendo que talvez o mais importante seja o económico. A possibilidade de correr sistemas dentro de outros sistemas introduz um nível de flexibilidade elevado para várias áreas como o desenvolvimento de aplicações ou criação de ambientes novos. Assim, a necessidade de aquisição de componentes físicos novos para criação de novos sistemas diminui, assim como a manutenção.

Desta temática surgiu a vontade de criar um simulador para a arquitectura MIPS que não se limite apenas a criar o motor de execução mas sim todos os mecanismos inerentes

à sua arquitectura.

O presente trabalho demonstra a forma como são desenvolvidos os primeiros passos no processo de virtualização de sistemas, assim como seus mecanismos e motor de execução direccionados para a arquitectura MIPS32.

3.1.1 O que se pretende

A virtualização de sistemas começa por simular o comportamento da arquitectura desejada, criando assim uma camada de abstracção no qual o sistema hóspede assentará e poderá executar código nativo da respectiva arquitectura. A figura 3.1 apresenta o conjunto de camadas que existem entre o *hardware* real e o sistema que está a ser virtualizado.

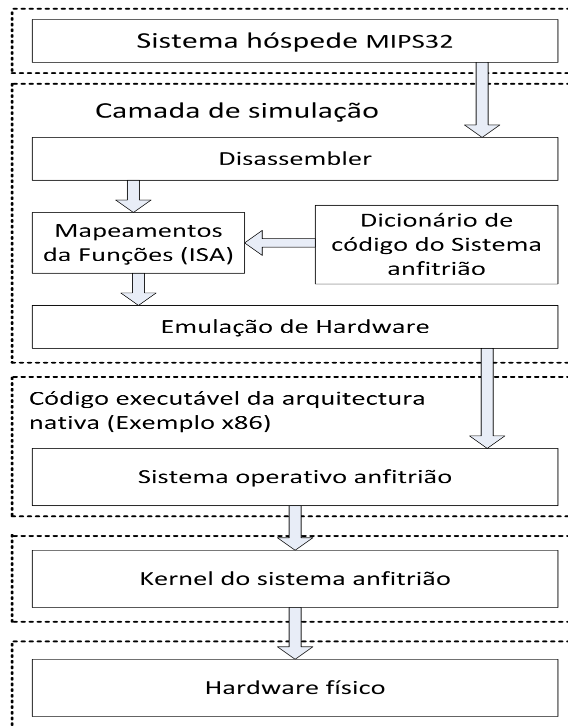


Figura 3.1: Camadas de abstração.

O papel desta camada é optimizar a comunicação entre os dois sistemas operativos, o hóspede e o anfitrião, para que seja apresentado ao sistema virtualizado um ambiente de execução semelhante ao que se obteria com *hardware* físico.

É exactamente isto que acontece em simuladores como o VMWare Player ou vSphere, Oracle VirtualBox, QEMU ou Microsoft Hyper-V. Estes simulam vários componentes genéricos que serão utilizados no simulador, também conhecido por máquina virtual. Estas máquinas, de forma a permitir executar sistemas operativos complexos, necessitam responder a vários requisitos das arquitecturas simuladas, nomeadamente, modos de execução, estruturas do processador, organização e gestão da memória, mecanismos físicos ou virtuais.

O mesmo acontece com a arquitectura MIPS, onde existem vários mecanismos e requisitos mínimos que devem ser cumpridos de forma a se poder afirmar que se têm realmente um simulador.

Este trabalho está directamente relacionado com esta temática, uma vez que o seu objectivo principal é a criação desta camada de *software* na máquina anfitrião, possibilitando a simulação de *hardware* e permitindo que código nativo da arquitectura MIPS possa correr fluentemente.

Assim, o simulador desenvolvido neste trabalho deverá respeitar os seguintes requisitos:

1. Implementação da arquitectura MIPS32 Revisão 2;
2. Implementação dos modos de execução *Kernel* e *User*;
3. Implementação dos coprocessadores de controlo, vírgula flutuante e respectivos registos internos;
4. Implementação dos componentes físicos necessários à execução de um sistema funcional como o processador, memória principal e módulo de ligação entre os vários componentes;
5. Implementação dos mecanismos de memória virtual, nomeadamente de tradução de endereços para a arquitectura MIPS32;
6. Implementação do mecanismo de excepções para a arquitectura MIPS32;
7. Implementação do mecanismo de interrupções para a arquitectura MIPS32;
8. Execução de código binário no formato ELF.

3.1.2 Metodologias

A documentação base utilizada na construção do simulador para a arquitectura MIPS32 é:

1. Referência rápida para a ISA MIPS32, [MT08];
2. Introdução à arquitectura MIPS32, [MT10a];
3. *Instruction Set* da arquitectura MIPS32, [MT10b];
4. *Privileged Resource Architecture* da arquitectura MIPS32, [MT10c];
5. Livro complementar *"See MIPS Run"*, [Swe06];
6. Descrição técnica da *motherboard* Malta MIPS, [MT02];
7. Descrição técnica de um microcontralador que utiliza processador MIPS, [Mic09].

Cronologicamente, o simulador desenvolveu-se nas seguintes fases:

1. Desenho e construção do motor de execução para um processador genérico MIPS32;
2. Construção de um componente que simule a memória RAM para armazenamento de dados em tempo de execução;
3. Desenho e implementação do elo de ligação entre o processador e a memória RAM;
4. Acrescento do mecanismo de excepções no processador;
5. Introdução do mecanismo de tradução de endereços TLB na MMU do processador;
6. Optimização do simulador;
7. Enriquecimento das funcionalidades do processador;
8. Implementação de código binário, simples e elaborado, para testes e demonstrações;
9. Utilização do sistema na sua totalidade.

Em primeiro lugar é necessário construir o motor de execução do processador cuja essência consiste no conjunto de instruções ISA, na unidade de gestão de memória sem tradução de endereços (FMT), nos registos genéricos necessários ao processamento das instruções e nos registos privados do coprocessador de controlo que guardam informações relativas a modos de operação, configurações e estados do processador.

Dado que nesta fase alguns mecanismos ou componentes ainda não existem, é necessário simular determinados "comportamentos" de forma a possibilitar o teste das instruções. Um exemplo destes casos são as instruções de acesso à memória, *load* e *store*, ou a chamada ao sistema *syscall* que acede ao mecanismo de excepções do processador.

O passo seguinte consiste na criação de uma "área" de trabalho para o processador poder armazenar e executar blocos de código sequenciais, mormente a memória RAM. A memória RAM não segue uma especificação já existente, o que não implica que a sua estrutura seja desorganizada. O seu funcionamento deve em todas as situações respeitar o de uma memória RAM real, contextualizada no âmbito do processo de simulação.

Uma vez implementados os dois componentes principais é necessário criar um elo de comunicação entre ambos. Surge assim a necessidade de criar a base na qual todos os componentes se irão ligar, à semelhança do que acontece com uma *motherboard* real. É aqui que todos os componentes serão instanciados, mapeados e interligados entre si de forma a possibilitar o seu funcionamento. É igualmente neste módulo que será feito o controlo externo sobre os programas a correr internamente no simulador. Note-se que uma vez estabelecidos os canais é necessário validar e testar a integração e comunicação dos módulos. Um exemplo destas situações são as instruções acima referidas *Load* e *Store*.

Na quarta fase o simulador está apto para correr instruções aritméticas, lógicas, saltos condicionais e incondicionais, mas não se encontra apto para executar um subconjunto de instruções de controlo e administração do fluxo de execução. Assim, algumas instruções não foram totalmente implementadas uma vez que se encontra em falta o mecanismo de excepções. É nesta fase que este mecanismo é implementado de forma a possibilitar ao processador a utilização das chamadas ao sistema, controlo do mecanismo de tradução de endereços TLB e utilização de *entry points* definidos pela arquitectura, entre outros aspectos.

A gestão de memória é um dos aspectos mais importantes na construção de um sistema operativo uma vez que a memória tem limite e o sistema deve de ser capaz de se organizar internamente de forma a manter a informação protegida e coerente. Para tal, a arquitectura MIPS disponibiliza três mecanismos para auxiliar a gestão e manutenção da memória, sendo eles: *Translation Lookaside Buffer* (TLB), *Fixed Mapping Translation* (FMT) e *Block Address*. Assim nasceu a necessidade de escolher qual o mecanismo mais adequado para se simular e introduzir no simulador. Dado que este deverá correr um pequeno *Kernel* que possibilitará a demonstração do funcionamento e da gestão da memória, optou-se pela implementação e integração na MMU do processador uma TLB. Desta forma será possível também demonstração do funcionamento de um mecanismo de gestão de memória complexo e a sua interacção com o processador.

Na sexta fase decidiu-se que o funcionamento do simulador não poderia ser estático, isto é, a declaração de estruturas e chamada de funções não eram mapeadas, sendo utilizados os nomes específicos de cada implementação. Assim, procedeu-se à optimização do sistema modularizando todos os componentes e mapeando todas as estruturas de dados e funções de forma a permitir a utilização de outros componentes no simulador. Através dos mapeamentos o simulador permitirá que sejam utilizados componentes desenvolvidos por terceiros. Outro aspecto relevante é a possibilidade de parametrização do simulador, permitindo a modificação de estados do processador, estados da memória ou mapeamentos da MMU, assim como a introdução de funcionalidades de interacção entre o utilizador, o simulador e o programa em execução internamente.

No plano de desenvolvimento do simulador, a sétima fase é precisamente a última, uma vez que é nesta fase que se adicionam funcionalidades extra ao processador, claramente desejáveis, sem comprometer as camadas que já se encontram implementadas. Na sua grande maioria, as funcionalidades extra possuem neste estado os requisitos mínimos preenchidos para a sua implementação. Em certas situações pode surgir a necessidade de implementação de instruções destinadas ao uso de um mecanismo concreto ou componente como a *cache*. Pegando como exemplo o mecanismo de interrupções ou o coprocessador de virgula flutuante (CP1), note-se que ambos não são obrigatórios para o funcionamento base do simulador uma vez que o primeiro é utilizado para comunicação com periféricos (opcional) e o segundo para cálculos de vírgula flutuante. No primeiro caso existe uma dependência do mecanismo de excepções o que implica que o mecanismo de interrupções não funcionará sem que o primeiro se encontre a funcionar. Relativamente ao segundo exemplo, o coprocessador um, este apenas tem dependência do coprocessador zero que é o responsável pelo fluxo de execução.

Concluída a implementação do simulador e de todos os seus mecanismos de execução, encontramo-nos na altura indicada para se criar exemplos preparados que demonstrem o funcionamento correcto dos vários componentes e mecanismos da arquitectura MIPS.

Por último temos a fase de utilização do sistema, na qual se poderá desenvolver código binário MIPS e correr no simulador, dispondo de todas as funcionalidades de interacção oferecidas pelo mesmo. Todos os exemplos serão demonstrados em detalhe no capítulo 5 deste trabalho.

3.2 Arquitectura

Em seguida será descrita a arquitectura em que o simulador foi concebido. Para tal será apresentada uma lista de módulos chave que demonstram o funcionamento do simulador e a forma como estes módulos comunicam entre si.

É importante referir que este trabalho foi realizado utilizando ferramentas já disponíveis auxiliadas com outras desenvolvidas inteiramente de raiz. Deste modo, trata-se de um protótipo sobre o qual foram realizados testes, descritos ao longo deste documento, não devendo ser considerado um produto exaustivamente aperfeiçoado.

3.2.1 Módulos do Sistema

A arquitectura do sistema é baseado em vários módulos complementares mas independentes entre si, formando assim um sistema complexo e funcional. Cada módulo tem uma função específica, sendo em alguns casos possível que o seu funcionamento interno varie com o tipo de implementação.

Um exemplo disso é o facto de ser possível parametrizar o sistema de forma a escolher qual a arquitectura que será utilizada no simulador, 32 ou 64 bits, embora apenas a arquitectura de 32 bits se encontre implementada.

A figura 3.2 apresenta um esquema da arquitectura, identificando a forma como os módulos estão relacionados.

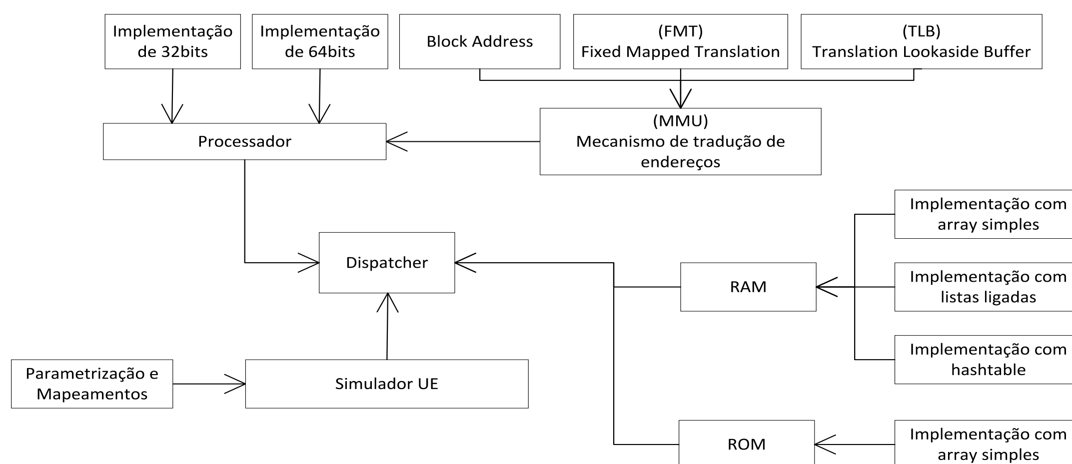


Figura 3.2: Módulos do SimuladorUE.

O módulo principal é o processador, descrito no capítulo 4.1, que implementa os mecanismos de operação, controlo e fluxo de execução da arquitectura MIPS 32 e 64 bits. A unidade de gestão MMU, interna ao processador, recebe pedidos de acesso ao módulo da memória principal pelo processador. É sua responsabilidade, através do mecanismo de tradução activo, proceder à tradução dos endereços virtuais para endereços físicos que serão utilizados para aceder à memória principal, como está desmonstrado nos capítulos 4.2 e 4.3.

Indexado ao funcionamento do módulo *MMU* estão os mecanismos de tradução de endereçõs TLB, FMT e *Block Address*, no qual apenas o primeiro se encontra implementado. Estes são os três formatos de resolução de endereços suportados pela arquitectura MIPS. A TLB é o modelo indicado para sistemas que exijam uma complexidade elevada por parte da gestão de memória, uma vez que permite mapear endereços virtuais nos segmentos USEG, KSEG2 e KSEG3 dinamicamente. O modelo FMT é optimizado para sistemas que exijam uma complexidade baixa de gestão de recursos como os microcontroladores. O terceiro modelo consiste numa tabela com mapeamentos, semelhante ao que acontece com o mecanismo TLB, uma vez que utiliza um subconjunto dos registos do coprocessador central igualmente utilizados pela TLB. Neste mecanismo não existem endereços pré definidos como acontece no mecanismo TLB ou FMT, sendo todos os endereços mapeados manualmente.

A memória do simulador é composta pelas componentes RAM e ROM. A primeira possui três tipos de representações possíveis, sendo o modelo com listas ligadas escolhido para implementação. Esta memória é utilizada pelo simulador como área de trabalho para o processador assim como armazenamento de dados e código binário. Optou-se pela implementação simples de uma memória ROM, com o objectivo de simular o processo de arranque o mais próximo possível da realidade.

O módulo de parametrizações, é composto por um ficheiro *header* de configurações *hardcoded*, que moldará o sistema e permitirá o mapeamento de estruturas e funções no simulador em tempo de compilação. Incialmente pensou-se na utilização de ficheiros parametrizáveis, mas uma vez que o sistema ainda se encontra em desenvolvimento decidiu-se pela parametrização *hardcoded*. Desta forma o simulador criará uma camada de abstração com os componentes utilizados, permitindo que a programação do simulador seja genérica e não contenha detalhes de implementações específicas.

Assim, o simulador é responsável pela instanciação e mapeamento, segundo as parametrizações *hardcoded*, das estruturas internas necessárias ao funcionamento dos dispositivos. É igualmente sua responsabilidade a criação e atribuição dos canais de comunicação entre os periféricos acima citados, assim como o lançamento da execução de cada componente num novo processo. No capítulo 4 será apresentado o modelo e a forma como é

feita a comunicação entre os simulador e os módulos do sistema.

Por último temos o módulo Dispatcher, controlado pelo o simulador e cuja função consiste em controlar os canais de comunicação, assim como mapear os pedidos do processador para os dispositvos ligados na placa mãe. Em *motherboards* reais este módulo é composto por um chip que controla tanto o *Northbridge* como o *Southbridge*. Ambos funcionam como *hubs* onde se ligam vários dispositivos, sendo o primeiro responsável pela comunicação entre o processador, a RAM e placa gráfica. O segundo é responsável por mapear os restantes dispositivos, nomeadamente: PCI, USB, ISA, IDE, ROM ou Legacy.

3.2.2 Plataforma de desenvolvimento

Todo o trabalho foi desenvolvido utilizando o sistema operativo Linux. A produção do código do simulador foi desenvolvida na linguagem ANSI C através do IDE CodeBlocks[1], que por sua vez possui internamente os compiladores GCC, G++ e GDB (Debugger)[2].

A utilização do debuger (gdb) foi uma peça fundamental na construção e depuração dos processos filhos, uma vez que estes são gerados pelo simulador. Através de flags[3] como o *set follow-fork-mode* ou *set detach-on-fork* foi possível modificar o caminho utilizado pelo debuger, de forma a permitir a depuração dos componentes ligados ao simulador.

Para o desenvolvimento de módulos e programas de teste em assembly MIPS, utilizou-se o *Sourcery CodeBench*[4] disponibilizado pela MIPS Inc[5].

Este Workbench disponibiliza um leque de ferramentas que auxiliam o desenvolvimento do código MIPS através da linguagem ANSI C ou assembler MIPS directamente, o que facilita em muito a tarefa de programação.
Com este Workbench é possível, através do processo de compilação cruzada, produzir código binário especificamente para a arquitectura MIPS32.

Do conjunto de ferramentas disponibilizadas, também conhecidas como `binutils`, destacam-se as seguintes:

- Compilador cruzado para a arquitectura MIPS32 (GNU C Compiler) para a linguagem ANSI C (`mips-linux-gnu-gcc`);

[1] Página web para o IDE CodeBlocks - `http://www.codeblocks.org`
[2] Página web para o GDB - `http://www.gnu.org/software/gdb/`
[3] Página oficial GDB - `http://sourceware.org/gdb/onlinedocs/gdb/Forks.html`
[4] Página web para o Workbench MIPS - `http://developer.mips.com`
[5] Página web da MIPS Inc - `http://www.mips.com`

- Compilador para assembler MIPS `as` (`mips-linux-gnu-as`);
- Objdump para reconstrução do código binário (`mips-linux-gnu-objdump`);
- Readelf para obtenção de informação de ficheiros binários ELF (`mips-linux-gnu-readelf`);
- Elfedit para modificação dos cabeçalhos dos mesmos (`mips-linux-gnu-elfedit`);
- Ld para linkagem de ficheiros objectos e bináros (`mips-linux-gnu-ld`);
- Objcopy para manipulação e geração de ficheiros binários (`mips-linux-gnu-objcopy`);
- Debuger para linguagem ANSI C `gdb` (`mips-linux-gnu-gdb`);
- Pré-processador para a linguagem ANSI C `cpp` (`mips-linux-gnu-cpp`).

Embora o compilador GCC disponibilizado para a arquitectura MIPS seja a ferramenta ideal para desenvolvimento de código, mostrou-se demasiado sofisticado para a primeira fase de desenvolvimento e testes no simulador, forçando a utilização do assembler `as` da arquitectura MIPS para a geração de código binário.

De forma a tornar o simulador robusto e válido, em ambiente de desenvolvimento, definiu-se que apenas seriam suportados ficheiros binários que sigam a norma ELF, uma vez que é o formato *standard* para ficheiros executáveis em sistemas operativos da família Linux.

Capítulo 4

Módulos

Este capítulo descreve os vários módulos base constituintes do simulador, nomeadamente o processador na secção 4.1, a unidade de gestão de memória na secção 4.2, mecanismo de tradução de endereços da TLB na secção 4.3, memória RAM e a memória ROM na secção 4.4.

Nas seguintes secções analisa-se em detalhe a estrutura e o funcionamento de um processador genérico para arquitectura MIPS32, abordando a ordenação dos bytes em 4.1.2, a forma como o código objecto é interpretado em 4.1.3 e posteriormente representado na memória do simulador. Serão abordados os casos de utilização dos pipelines em 4.1.4 e o desenho do coprocessador de controlo na secção 4.1.5, também conhecido como coprocessador zero, sendo seus respectivos registos internos e genéricos descritos nas secções 4.1.6 e 4.1.7. Serão igualmente explanados na secção 4.1.1 os modos de execução *User* e *Kernel* do processador e seu funcionamento dentro do simulador. Os mecanismos de excepções e interrupções da arquitectura MIPS32, essenciais para a execução e comunicação do sistema com periféricos, serão abordados detalhadamente na secção 4.1.8.

Uma vez que o processador usufrui do mecanismo de memória virtual, é necessário emular uma memória física que terá os seus endereços mapeados pela memória virtual. Esta organização encontra-se descrita na secção 4.4 deste capítulo. As secções 4.2 e 4.3 serão dedicadas à explanação das estruturas utilizadas para suportar a unidade de gestão de memória (MMU) e a forma como a arquitectura MIPS faz a tradução de endereços com a TLB.

Os capítulos seguintes expõem detalhadamente os procedimentos adoptados em cada fase do trabalho.

4.1 Processador

No processo de construção de um simulador para uma arquitectura, independentemente da sua finalidade, o passo principal é o desenho e construção do motor de execução, mormente, o processador. Este é o mecanismo principal que receberá a informação no seu estado "bruto", e após sua interpretação a executará.

A família MIPS contém várias implementações diferentes para processadores, que podem: variar no tamanho das instruções (32 bits, 64 bits ou codificação de 16 bits para microcontroladores); possuir um ou vários núcleos de execução; permitir a utilização de *threads*; variar no mecanismo de tradução de endereços, entre muitas outras funcionalidades.

Dadas as referidas condições, optou-se pela implementação genérica de um processador MIPS, com apenas um núcleo de execução, onde a ISA será essencialmente a MIPS32 (ver. 2)[1] e o mecanismo de tradução utilizado a TLB. Assim, o primeiro passo na construção do processador é a definição de uma estrutura base que represente o processador.

```
typedef struct CPU_32{

        /* Unidade de Gestao de Memoria
         */
        MMU *mmu;

        /* Canais de comunicacao
         * Tx - Transmissor                    MMU ---> RAM
         * Rx - Receptor                       MMU <--- RAM
         */

        int Tx;
        int Rx;

        /* Registos do Coprocessador 0
         * 32 Registos de 32 bits
         */
        GPR CP0R[CP0_Registos];
```

[1] A ISA MIPS32r2 é a evolução das suas antecessoras (nomeadamente MIPS I, MIPS II, MIPS III e MIPS IV) mantendo actualmente retro compatibilidade em grande parte da arquitectura, tendo algumas instruções sido eliminadas, outras adicionadas, e algumas alterações efectuadas na unidade de controlo.

```
        /* Registos especiais
         *
         * PC - Program Counter, HI, LO
         * Inst (actual)
         */

        GPR HI;
        GPR LO;

        int PC;             //Proxima Instrucao
        int Inst;           //Instrucao Actual
        int PC_Delay;       //Program Counter (Jump's e Branch's)

        int State;          //Estado do CPU
                            //Normal, Delay_Branch ou Idle

        /* Registos do Coprocessador 0
         */
        CP0_Entry *CP0_Reg[32];

} CPU_32;
```

Listing 4.1: Representação do processador genérico MIPS.

É através desta estrutura que são definidas as variáveis necessárias para simular o comportamento de um processador real. No bloco de código 4.1, pode-se observar as definições: da unidade de gestão de memória (MMU) e seus respectivos canais de comunicação, dos registos genéricos, dos registos internos, dos registos especiais HI e LO para operações aritméticas de multiplicação e divisão, do PC (Program Counter) e um registo especial concebido apenas para o processo de simulação, o registo "Inst".

Numa abordagem inicial, foquemo-nos exclusivamente nas variáveis: "State", "Inst", "PC" e "PC_Delay". Relativamente às restantes estruturas mencionadas acima, estas serão abordadas mais à frente neste capítulo, uma vez que não são necessárias para a construção da máquina de estados principal do processador.

O registo Inst contém o endereço da instrução actual no processador enquanto que o registo PC indica qual a instrução seguinte (Inst + 4 bytes). O mesmo se aplica ao registo PC_Delay, cujo objectivo é guardar o endereço para o qual o processador deverá modificar o PC, após ser executada a instrução seguinte a uma instrução de salto. Associado a este comportamento encontra-se o registo State no processador, cujo objectivo consiste em identificar o estado actual em que se encontra o processador a executar instruções.

Só a estrutura em si não permite a execução de código binário, sendo necessário a

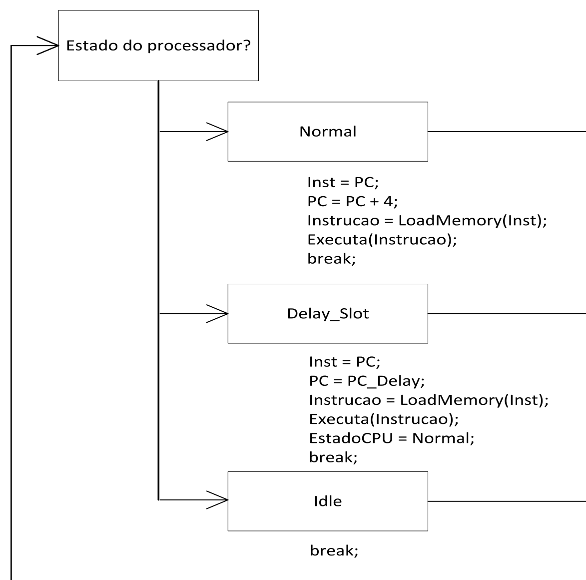


Figura 4.1: Máquina de estados da execução no processador.

definição de uma máquina de estados[2] que fará a gestão de toda a execução de instruções no processador. Desta forma, optou-se pela criação da função:

```
void executa(CPU *cpu);
```

Listing 4.2: Função responsável pela execução do processador.

É através desta função que será efectuado o controlo interno do fluxo de execução[3] no processador. A cada chamada desta função, será gerada uma nova iteração no processador, forçando a execução de uma nova instrução proveniente da RAM assim como a actualização do estado do processador. Como identificado anteriormente, o processador possui internamente três estados no qual pode funcionar: "Normal", "Delay Branch" e "Idle".

O modo Normal é o estado inicial do funcionamento do processador, sendo que todas as instruções são executadas neste estado. Devido a limitações físicas do *hardware* em

[2]Esta máquina de estados é necessária para implementar o comportamento Branch Delay existente nos pipelines (ver secção 4.1.4).

[3]Dado que as fases de execução do processo de pipelining não são simuladas, a sua representação é feita através da função executa.

determinadas situações, nomeadamente a execução de instruções de salto, a instrução seguinte a um salto é executada antes da instrução de salto, levando a uma mudança de estado no processador para "Delay Branch". Esta situação é descrita em detalhe em 4.1.4. O último estado é utilizado pelo *software* controlador do motor de execução, neste caso o simulador, para enviar um sinal ao processador para entrar em estado "Idle"(espera). Normalmente utilizado para iniciar a sequência de encerramento no processador e no simulador. Na figura 4.1 pode-se visualizar a máquina de estados responsável pelos estados internos do processador.

Chama-se no entanto à atenção que a máquina de estados de execução no processador é utilizada meramente para representar o funcionamento dos pipelines (ver 4.1.4) e não os estados internos do processador representados pelo PRA (*Privileged Resource Architecture*) da arquitectura MIPS32.

4.1.1 Modos de Execução

O PRA (*Privileged Resource Architecture*) da arquitectura MIPS disponibiliza quatro modos de execução, nomeadamente, *Kernel*, *User*, *SuperUser* e *Debug*, sendo apenas os dois primeiros obrigatórios para o seu funcionamento. Os modos de *Debug* e *SuperUser* definidos pelo PRA são opcionais, e para o primeiro é necessário a implementação do mecanismo de depuração de *hardware* EJTAG. Este mecanismo adiciona funcionalidades de depuração *debug* ao sistema e é um caso especial de execução pois, assim como o modo *Kernel* também tem acesso total aos recursos do sistema. O modo *SuperUser* é destinado para implementações específicas dos arquitectos de sistemas operativos utilizarem. As flags no registo *Status* do coprocessador zero[4] que controlam os modos de execução do processador são: KSU (modo *Kernel*, *SuperUser*, *User*), EXL (Exception Level), ERL (Error Level) e o registo Debug, caso o mecanismo EJTAG esteja implementado. Segue-se uma pequena descrição dos modos de operação do processador:

- **Modo Kernel**

  Quando o processador se encontra em modo *Kernel*, este tem acesso total às funcionalidades do processador, incluindo acesso total ao espaço de endereçamento na memória virtual, permissão para modificações nos mapeamentos da memória virtual nos mecanismos de tradução de endereços, controlo sobre os registos genéricos, registos internos ou mecanismos de excepções.

  O processador encontra-se em modo Kernel se as seguintes condições forem verdade:

[4] Os registos do coprocessador zero serão abordados na secção 4.1.7.

  - A flag DM no registo *Debug* é 0
  - A flag KSU no registo *Status* é 0b00
  - A flag EXL no registo *Status* é 1
  - A flag ERL no registo *Status* é 1

  Este modo é obrigatório e encontra-se implementado no simulador.

- **Modo User**
  Caso o processador se encontre a correr em modo *User*, este tem acesso[5] aos registos genéricos e ao coprocessador um (vírgula flutuante). Caso os bits $StatusCU_{3..0}$ se encontrem activos, o processador dispõe de acesso aos registos internos do coprocessor cujo bit se encontre activo. Possui também acesso limitado ao espaço de endereçamento na memória virtual variando o seu domínio entre 0x0000.0000 e 0x7FFF.FFFF. O processador encontra-se em modo *User* se as seguintes condições forem verdade:

  - A flag DM no registo *Debug* é 0
  - A flag KSU no registo *Status* é 0b10
  - A flag EXL no registo *Status* é 0

  Este modo é obrigatório e encontra-se implementado no simulador.

- **Modo SuperUser**
  Este modo de execução é opcional e a sua implementação depende exclusivamente dos arquitectos do sistema. O processador encontra-se em modo *Super User* se as seguintes condições forem verdade:

  - A flag DM no registo *Debug* é 0
  - A flag KSU no registo*Status* é 0b01
  - A flag EXL no registo *Status* é 0
  - A flag ERL no registo *Status* é 0

  Este modo é opcional e não se encontra implementado no simulador.

- **Modo Debug**
  Para o processador entrar em modo de execução *Debug* é necessário ter implementado o mecanismo EJTAG e conter a flag DM do registo *Debug* do coprocessador zero a 1. Neste modo de execução o processador contém os mesmos atributos de execução que o modo *Kernel*. Este modo é opcional e não se encontra implementado no simulador.

[5]Apenas em determinadas situações, ver secção 4.1.7 para uma descrição detalhada dos registos internos do coprocessador zero.

Simulação em Software

Os modos acima referidos encontram-se implementados dentro do núcleo de execução do processador, mais propriamente na estrutura que representa os registos internos no ficheiro "Mips32_core.h". O bloco de código seguinte apresenta a estrutura de dados utilizada para representar os registos internos responsáveis pela gestão dos modos de operação do processador.

```
typedef struct CP0_Entry{
        int Numero;         // Numero do Registo
        GPR Sel[8];         // Sel - Registos com Informacao
        char Comp[32];      // Nivel de obrigatoriedade
}CP0_Entry;
```

Listing 4.3: Representação dos registos internos do CP0.

Toda a estrutura assim como os registos internos, existentes e implementados, da arquitectura MIPS32, encontram-se devidamente explanados em 4.1.7, uma vez que nos iremos focar em apenas um registo, *Status*. Segue-se a definição dos registos do coprocessador zero dentro da estrutura do processador:

```
/* Registos do Coprocessador 0
 */
CP0_Entry *CP0_Reg[32];
```

Listing 4.4: Conjunto de registos internos do CP0.

Dado que o modo de operação *debug* não se encontra implementado, o único registo interno responsável pelos modos de operação no processador é o registo *Status*. O controlo é feito através dos bits KSU, EXL e ERL, sendo os primeiros dois bits do KSU utilizados para identificar qual o modo de execução se encontra activo caso os bits EXL e ERL se encontrem desligados. Nos casos em que os bits EXL ou ERL se encontrem activos, os bits KSU são ignorados forçando o processador a entrar em modo *kernel*.

A sua manipulação é feita através de aplicação de máscaras de bits, uma vez que os restantes bits dentro da mesma word devem ser preservados. Desta forma definiu-se um conjunto de máscaras, que juntamente com operações lógicas bitwise permitem a modificação dos bits.

A título de facilitar o acesso aos registos e o *debug* do simulador, optou-se igualmente pela definição de macros para acesso aos valores dos bits específicos.

```
//Status
#define Status          cpu->CP0_Reg[12]->Sel[0]
#define StatusKSU       (Status & (bit_3 | bit_4)) >> 3
```

```
#define StatusERL        (Status & bit_2) >> 2
#define StatusEXL        (Status & bit_1) >> 1
```

Listing 4.5: Macros para o registo interno Status e respectivos bits (*flags*).

Desta forma a manipulação dos bits torna-se bastante simples uma vez que a sua extracção é feita através de macros simplificando o processo de comparação e validação dos mesmos. Por exemplo, para se modificar o modo de operação de *kernel* para *User* é necessário desligar os bits ERL, EXL e introduzir a máscara correspondente nos bits KSU. Para tal basta correr o seguinte código:

```
//Activar padrao 0x10 no KSU
Status = ( Status & zbit_3 ) | ( Status & bit_4);

//Desligar bits EXL e ERL
Status = Status & zbit_2 & zbit_1;
```

Listing 4.6: Exemplo de mudança de modo Kernel para modo User.

4.1.2 Endianness

Inicialmente na arquitectura MIPS, os processadores utilizavam apenas a ordenação de bytes *Little Endian*. Mais tarde, passaram a possibilitar a escolha como é feita a interpretação da informação na memória, nomeadamente, *Little Endian* ou *Big Endian*. Esta diferenciação é de extrema importância para o sistema pois influenciará a forma como o processador irá guardar e interpretar informação proveniente da memória. Alerta-se para o facto que, em sistemas com a mesma arquitectura, se o código compilado e a *endianness* configurada no processador não corresponderem, o sistema não funcionará.

Por defeito, será sempre utilizada a notação *Little Endian* na referência à informação existente no simulador, uma vez que este suporta exclusivamente a ordenação *Little Endian*.

Na figura 4.2 pode-se ver a forma como o simulador representa a informação de 32 e 64 bits. Como se pode ver, os bytes estão ordenados utilizando a ordenação *Little Endian*, desta forma, todos os tipos de dados possuem o byte zero mais à direita. De modo a haver uma maior compatibilidade com os sistemas alvo onde o simulador irá correr, são suportados os seguintes tipos de dados: DWORD, WORD, HWORD e BYTE. Sendo estes representados no ficheiro "TypeDefs.h", na linguagem C, por int64_t, int32_t, int16_t e int8_t respectivamente, como demonstra o bloco de código em 4.7.

Todos os tipos de dados das instruções são extraídos e representados sem sinal (`unsigned`), sendo convertidos posteriormente para uma representação com sinal (`signed`).

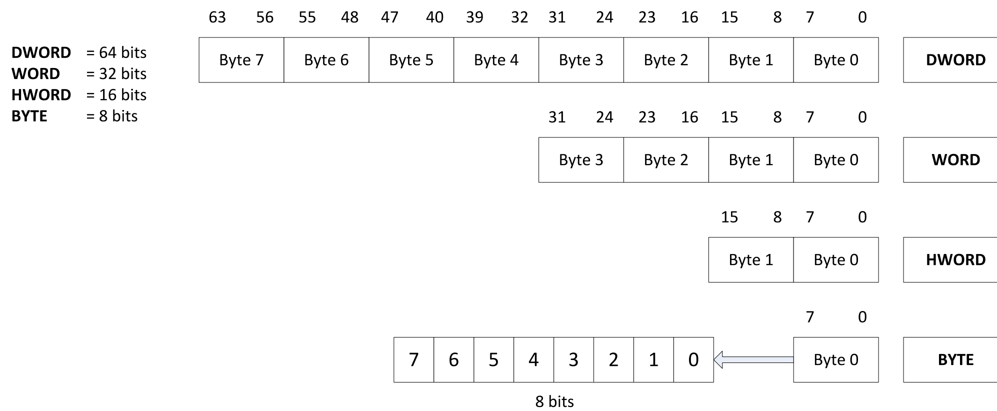


Figura 4.2: Tipos de dados no SimuladorUE.

```
#ifndef TYPES
#define TYPES

/* Tipos de Dados
 */
typedef int64_t DWORD;
typedef int32_t WORD;
typedef int16_t HWORD;
typedef int8_t  BYTE;

#endif
```

Listing 4.7: Tipos de dados.

Segue-se um exemplo para cada tipo de ordenação e respectiva interpretação utilizada pela arquitectura MIPS.

Little Endian

Nos casos em que a ordenação seja *Little Endian*, o primeiro byte é o byte mais à direita na WORD, também conhecido por LSB (*Least Significant Byte*). Imagine-se o número 0x12345678 guardado algures na memória. A sua representação numa arquitectura com ordenação de bytes *Little Endian* encontra-se representada na figura 4.3.

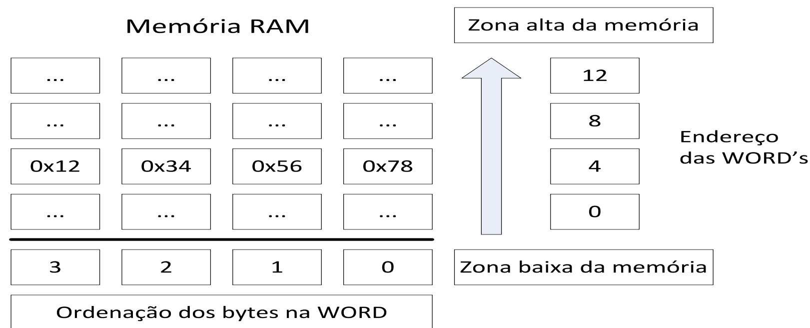


Figura 4.3: Little Endian.

Big Endian

Se o byte zero é o byte mais à esquerda, também conhecido por MSB *(Most Significant Byte)*, então encontramo-nos a utilizar a ordenação de bytes *Big Endian.* Imagine-se o mesmo caso da figura anterior, mas utilizando agora a ordenação de bytes *Big Endian*, ficaremos com a representação apresentada na figura 4.4.

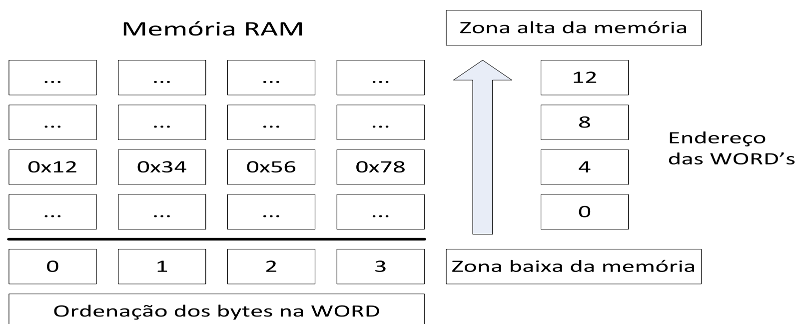


Figura 4.4: Big Endian.

4.1.3 Interpretação de código máquina

De forma a permitir que o processador consiga executar código binário, é necessário em primeiro lugar compreender o seu conteúdo. Para tal, é necessário interpretar a informação proveniente da memória. Desta forma, o processador lê blocos de 32 bits da memória e extrai os bits necessários para entender qual a operação que deve executar. Estes bits são conhecidos como *opcode* e encontram-se nos seis bits mais significativos da WORD lida da memória.

É através de tabelas de codificação de instruções definidas pela arquitectura MIPS32, que é possível identificar a instrução e os seus argumentos para que o processador consiga executar. Na arquitectura MIPS32 (ver.2), a interpretação pode ser feita de duas formas: directamente ou indirectamente. Se os seis bits mais significativos apontarem para uma instrução, como é o caso da instrução ADDI, encontramo-nos no primeiro caso. Se os seis bits mais significativos apontarem para um subconjunto de instruções, como exemplo a instrução ADD, é necessário extrair os seis bits menos significativos de forma a identificar a instrução.

Veja-se o exemplo da instrução ADDI cuja codificação é:

Bits	31 .. 26	25 .. 21	20 .. 16	15 .. 0
Nomenclatura	ADDI	arg. 1	destino	valor imediato
Codificação	001000	rs	rt	valor imediato

Tabela 4.1: Codificação para a instrução ADDI.

Como se pode ver no quadro acima, os seis bits mais significativos identificam a instrução ADDI directamente, sendo os dezasseis bits menos significativos utilizados para identificar o valor imediato passado como terceiro argumento da instrução. Os restantes bits serão utilizados para identificar os registos genéricos no coprocessador central, segundo demonstra a tabela 4.13 na página 57.

Pegue-se agora como exemplo a instrução ADD, cuja codificação é:

Bits	31 .. 26	25 .. 21	20 .. 16	15 .. 11	10 .. 6	5 .. 0
Nomenclatura	SPECIAL	arg. 1	arg. 2	destino	-	ADD
Codificação	000000	rs	rt	rd	00000	100000

Tabela 4.2: Codificação para a instrução ADD.

Em oposição à instrução ADDI, a instrução ADD necessita de duas comparações até que o processador consiga identificar a instrução que irá executar. Assim, é necessário extrair

os seis bits mais significativos e em seguida utilizar os seis bits menos significativos na sub tabela de codificação para encontrar a instrução. Existem no entanto codificações que necessitam de mais comparações até que o processador consiga descodificar a instrução. Um exemplo disso são as instruções do subconjunto COP0, nomeadamente as instruções: MFC0 e MTC0.

Pegue-se nas instruções MFC0 e TLBR, uma vez que ambas pertencem às duas tabelas de codificação do sub conjunto COP0.

Bits	31 .. 26	25 .. 21	20 .. 16	15 .. 11	10 .. 3	2 .. 0
Nomenclatura	COP0	MF	RG	RCP0	-	sel
Codificação	010000	00000	rt	rd	00000000	-

Tabela 4.3: Codificação para a instrução MFC0.

Bits	31 .. 26	25	24 .. 6	5 .. 0
Nomenclatura	COP0	CO	-	TLBR
Codificação	010000	1	000 0000 0000 0000 0000	000001

Tabela 4.4: Codificação para a instrução TLBR.

Nestes casos em particular, a primeira operação no processo de descodificação é a extracção dos seis bits mais significativos de forma a saber se os bits identificam uma instrução ou um sub conjunto. Uma vez identificado o sub conjunto COP0, o passo seguinte é identificar qual será a tabela consultada para identificação da instrução.

Para tal, é necesário interpretar a informação da área RS da instrução, nomeadamente os bits 25 a 21. Na tabela 4.10 pode-se ver a codificação utilizada para o campo RS. Em seguida, compara-se o bit 25 (CO) do campo RS e caso este seja um utiliza-se a segunda tabela de codificação 4.11, no qual o processador utilizará os seis bits menos significativos para identificar a instrução no sub conjunto, neste caso a instrução TLBR. Caso o bit 25 (CO) do campo RS seja zero, então será utilizada a primeira tabela de codificação do sub conjunto COP0, no qual serão comparados os bits 25 a 21 para identificar a instrução como se pode ver na tabela 4.11.

Tabelas de Codificação de Opcodes

Em seguida serão apresentados os quadros com as codificações das instruções e respectivos sub conjuntos, de acordo com a especificação da arquitectura MIPS32 (ver. 2) segundo [MT10b].

OPCODE		**bits 28..26**							
		0	1	2	3	4	5	6	7
bits 31..29		000	001	010	011	100	101	110	111
0	000	SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	010	COP0	COP1	COP2	COP1X	BEQL	BNEL	BLEZL	BGTZL
3	011	*	*	*	*	SPECIAL2	JALX	*	SPECIAL3
4	100	LB	LH	LWL	LW	LBU	LHU	LWR	*
5	101	SB	SH	SWL	SW	*	*	SWR	CACHE
6	110	LL	LWC1	LWC2	PREF	*	LDC1	LDC2	*
7	111	SC	SWC1	SWC2	*	*	SDC1	SDC2	*

Tabela 4.5: Tabela de codificação OPCODE.

SPECIAL		**bits 2..0**							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	SLL	MOVCI	SRL	SRA	SLLV	*	SRLV	SRAV
1	001	JR	JALR	MOVZ	MOVN	SYSCALL	BREAK	*	SYNC
2	010	MFHI	MTHI	MFLO	MTLO	*	*	*	*
3	011	MULT	MULTU	DIV	DIVU	*	*	*	*
4	100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	101	*	*	SLT	SLTU	*	*	*	*
6	110	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	111	*	*	*	*	*	*	*	*

Tabela 4.6: Tabela de codificação SPECIAL.

Todas as codificações representadas por (*) encontram-se reservadas para futuras implementações. Note-se no entanto, que a codificação do sub conjunto COP0 divide-se em duas tabelas, dependendo do estado do RS, nomeadamente uma para os bits 25 a 21 e outra tabela para os bits 5 a 0.

SPECIAL2		**bits 2..0**							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	MADD	MADDU	MUL	*	MSUB	MSUBU	*	*
1	001	*	*	*	*	*	*	*	*
2	010	*	*	*	*	*	*	*	*
3	011	*	*	*	*	*	*	*	*
4	100	CLZ	CLO	*	*	*	*	*	*
5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	*	*	*	*	SDBBP

Tabela 4.7: Tabela de codificação SPECIAL2.

RGIMM		**bits 18..16**							
		0	1	2	3	4	5	6	7
bits 20..19		000	001	010	011	100	101	110	111
0	00	BLTZ	BGEZ	BLTZL	BGEZL	*	*	*	*
1	01	TGEI	TGEIU	TLTI	TLTIU	TEQI	*	TNEI	*
2	10	BLTZAL	BGEZAL	BLTZALL	BGEZALL	*	*	*	*
3	11	*	*	*	*	*	*	*	SYNCI

Tabela 4.8: Tabela de codificação RGIMM.

SPECIAL3		**bits 2..0**							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	EXT	*	*	*	INS	*	*	*
1	001	*	*	*	*	*	*	*	*
2	010	*	*	*	*	*	*	*	*
3	011	*	*	*	*	*	*	*	*
4	100	BSHFL	*	*	*	*	*	*	*
5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	RDHWR	*	*	*	*

Tabela 4.9: Tabela de codificação SPECIAL3.

COP0 (rs)		bits 23..21							
		0	1	2	3	4	5	6	7
bits 25..24		000	001	010	011	100	101	110	111
0	00	MFC0	*	*	*	MTC0	*	*	*
1	01	*	*	RDPGPR	MFMC0	*	*	WRPGPR	*
2	10	Campo CO							
3	11								

Tabela 4.10: Tabela de codificação COP0 (rs).

COP0		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	*	TLBR	TLBWI	*	*	*	TLBWR	*
1	001	TLBP	*	*	*	*	*	*	*
2	010	*	*	*	*	*	*	*	*
3	011	ERET	*	*	*	*	*	*	DERET
4	100	WAIT	*	*	*	*	*	*	*
5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	*	*	*	*	*

Tabela 4.11: Tabela de codificação COP0.

Simulação em Software

O processo de interpretação de instruções, embora não aparente, é de suma importância para o desempenho do simulador, uma vez que para todas as instruções é necessário proceder à sua descodificação, independentemente de a mesma já ter sido executada anteriormente.

Inicialmente pensou-se na utilização de um *swich* para comparar cada instrução em tempo de execução, o que levou a um nível de performance terrivelmente baixo dado que é necessário efectuar muitas comparações para executar uma única instrução. Este modelo foi rapidamente substituído por um novo, cujo principal objectivo é tirar partido de apontadores de funções na linguagem ANSI C. Com este novo formato, é possível que ocorra apenas uma comparação em instruções directas e no máximo três comparações em instruções que possuam até três níveis de codificação, como acontece com as instruções do coprocessador zero MTC0, MFC0, TLBP, TLBR, TLBWI, TLBWR, WAIT, ERET ou DERET.

Para este modelo funcionar, é necessário que existam as tabelas de codificação apresentadas na secção anterior. Segue-se um excerto da implementação da tabelas de codificação utilizando a linguagem ANSI C:

Funções para conjunto opcode	
void SPECIAL(CPU_32 *cpu);	void SPECIAL3(CPU_32 *cpu);
void REGIMM(CPU_32 *cpu);	void LB(CPU_32 *cpu);
void J(CPU_32 *cpu);	void LH(CPU_32 *cpu);
void JAL(CPU_32 *cpu);	void LWL (CPU_32 *cpu);
void BEQ(CPU_32 *cpu);	void LW(CPU_32 *cpu);
void BNE(CPU_32 *cpu);	void LBU(CPU_32 *cpu);
void BLEZ(CPU_32 *cpu);	void LHU(CPU_32 *cpu);
void BGTZ(CPU_32 *cpu);	void LWR(CPU_32 *cpu);
void ADDI(CPU_32 *cpu);	void SB(CPU_32 *cpu);
void ADDIU(CPU_32 *cpu);	void SH(CPU_32 *cpu);
void SLTI(CPU_32 *cpu);	void SWL(CPU_32 *cpu);
void SLTIU(CPU_32 *cpu);	void SW(CPU_32 *cpu);
void ANDI(CPU_32 *cpu);	void SWR(CPU_32 *cpu);
void ORI(CPU_32 *cpu);	void CACHE(CPU_32 *cpu);
void XORI(CPU_32 *cpu);	void LL(CPU_32 *cpu);
void COP2(CPU_32 *cpu);	void RESERVD(CPU_32 *cpu);
void COP1X(CPU_32 *cpu);	void LDC1(CPU_32 *cpu);
void BEQL(CPU_32 *cpu);	void LDC2(CPU_32 *cpu);
void BNEL(CPU_32 *cpu);	void SC(CPU_32 *cpu);

void BLEZL(CPU_32 *cpu);	void SWC1(CPU_32 *cpu);
void BGTZL(CPU_32 *cpu);	void SWC2(CPU_32 *cpu);
void SPECIAL2(CPU_32 *cpu);	void SDC1(CPU_32 *cpu);
void JALX(CPU_32 *cpu);	void SDC2(CPU_32 *cpu);
void SPECIAL2(CPU_32 *cpu);	void SDC1(CPU_32 *cpu);
void JALX(CPU_32 *cpu);	void SDC2(CPU_32 *cpu);

Tabela 4.12: Representação interna das instruções do conjunto *opcode*.

O primeiro passo consiste na declaração de todas as funções que representarão as instruções internamente no motor de execução.

O segundo passo consiste na criação de um array de apontadores para as funções criadas definindo também o tipo de argumento passado, mormente a estrutura que guardará a informação relativa ao processador. É importante referir que todas as funções devem utilizar o mesmo tipo de argumento.

```
static void (*opcodes[64])(CPU_32 *cpu) = {
  SPECIAL, REGIMM,  J,        JAL,      BEQ,      BNE,      BLEZ,     BGTZ,
  ADDI,    ADDIU,   SLTI,     SLTIU,    ANDI,     ORI,      XORI,     LUI,
  COP0,    COP1,    COP2,     COP1X,    BEQL,     BNEL,     BLEZL,    BGTZL,
  RESERVD, RESERVD, RESERVD,  RESERVD,  SPECIAL2, JALX,     RESERVD,  SPECIAL3,
  LB,      LH,      LWL,      LW,       LBU,      LHU,      LWR,      RESERVD,
  SB,      SH,      SWL,      SW,       RESERVD,  RESERVD,  SWR,      CACHE,
  LL,      LWC1,    LWC2,     PREF,     RESERVD,  LDC1,     LDC2,     RESERVD,
  SC,      SWC1,    SWC2,     RESERVD,  RESERVD,  SDC1,     SDC2,     RESERVD
};
```

Listing 4.8: Mapeamento de funções OPCODE em C.

Uma vez declaradas as funções e as tabelas de codificação, o sistema já se encontra pronto para descodificar instruções. Assim, o processo de descodificação é feito através do seguinte código:

```
instrucao = LoadMemory  (cpu, sizeof(WORD), cpu->Inst);
(* opcodes[ bit_31_26(instrucao) ])(cpu);
```

Listing 4.9: Descodificação de instruções em C.

No bloco de código acima pode-se ver a forma como os apontadores para funções funcionam. A função `bit_31_26(instrução)` é responsável por extrair o valor dos bits 31 a 26 no formato `unsigned`, "shiftando" o seu resultado 26 bits para a direita, formando assim um valor inteiro com seis bits. Como se pode observar no quadro 4.5, a tabela dos *opcodes* possui 64 entradas que correspondem exactamente aos 2^6 bits. Este valor será

utilizado em seguida como índice no array de apontadores *opcode*, de forma a criar uma instrução executável na linguagem C, formando assim um apontador para uma função concreta que pode ser executada como por exemplo `ADD(cpu)`.

Nos casos em que seja necessário efectuar mais comparações, o processo é muito semelhante. Os bits extraídos serão escolhidos de acordo com as tabelas de resolução utilizadas, nomeadamente: SPECIAL, SPECIAL2, REGIMM ou SPECIAL3.

4.1.4 Pipelines

A arquitectura MIPS foi concebida para tirar partido da técnica de *pipelining*. Esta é implementada a nível de *hardware* de modo a que se consiga optimizar o *throughput* do processador. Actualmente, a arquitectura MIPS utiliza um *pipeline* de 5 andares, ou seja, todas as instruções estão divididas em 5 fases. Com o pipeline consegue-se tirar maior partido das instruções uma vez que o processador em cada ciclo do relógio executa 5 passos em 5 instruções diferentes.

O objectivo do pipeline é que cada passo da instrução demore no máximo 1 ciclo de relógio do processador para que a cada ciclo do relógio se possa iniciar uma nova instrução. O tempo de cada fase não é fixo podendo este variar. Para efeitos de melhor compreensão serão demonstrados casos de uso do "4 staged pipeline" e do "5 staged pipeline".

5-Staged Pipeline

O "*5 staged pipeline*" consiste em dividir cada instrução em 5 fases, sendo elas:

IF - Instruction Fetch
Carregamento da próxima instrução a ser executada.

RD - Read Registers
Leitura do conteúdo dos registos do processador cujo os números são apontados pelos campos da instrução. Também conhecido por descodificação da instrução e verificação de dependências.

ALU - Arithmetic Logic Unit
Execução de operações aritméticas e lógicas em um ciclo de relógio. As operações de vírgula flutuante necessitam de mais que um ciclo de relógio e são executadas num coprocessador adicional (opcional).

MEM - Memory
Nesta fase, a instrução pode ler e escrever variáveis na memória. Chama-se a atenção que os acessos à memória são operações lentas. Em média, 3 de 4 operações não fazem nada nesta fase, dado que a maior parte das operações são efectuadas sobre os registos genéricos, internos ou da cache.

WB - Write Back
Esta fase serve para guardar os valores obtidos da operação feita no registo de destino.

Em alguns livros o estado IF e o estado RD são considerados um só uma vez que estes ocupam menos de 1 ciclo de relógio.

Tipos de Pipelining

Inicialmente o processador MIPS foi projectado para ser utilizado com *pipelines* de 5 estados. É de notar que a grande maioria dos CPUs mantém este formato, o que não implica que não exista evolução nas técnicas de *pipelining*. Veja-se a família do processador 1074K como exemplo, que possui um pipeline supercalar de 15 estados!
É uma diferença abismal de desempenho. Em seguida demonstraremos 4 categorias de *pipelining* existentes onde se integram uma grande variedade de processadores, onde é possível que varie o número de estados do *pipeline*.

Pipeline de instrução única com profundidade 1 Como se pode ver no *pipeline* exemplificado na figura 4.5, o desempenho não é óptimo uma vez que cada instrução necessita de 5 ciclos do relógio. O *Fetch* da nova instrução é feito quando a última instrução acabar de executar.

Pipeline paralelo Na figura 4.6 temos um pipeline com profundidade 5, no qual se consegue obter um *throughtput* 5 vezes superior ao pipeline da figura 4.5. Em cada ciclo do relógio é possível executar 5 estados diferentes.

Super pipeline Existem outros tipos de pipelines, pensados para melhorar ainda mais o desempenho da máquina. O super pipeline consegue executar duas fases em um ciclo de relógio como se pode ver na figura 4.7.

Superscalar ou Pipeline com Profundidade N Pode-se sempre ir um pouco mais além para aumentar o desempenho como mostra a figura 4.8, no qual se têm um pipeline de 5 fases com 4 vias. Isto é, em cada ciclo o processador pode executar até 4 instruções em simultâneo. Assim, o desempenho aumenta drasticamente, mas aumenta igualmente a probabilidade de dependências e de “stalls” no pipeline.

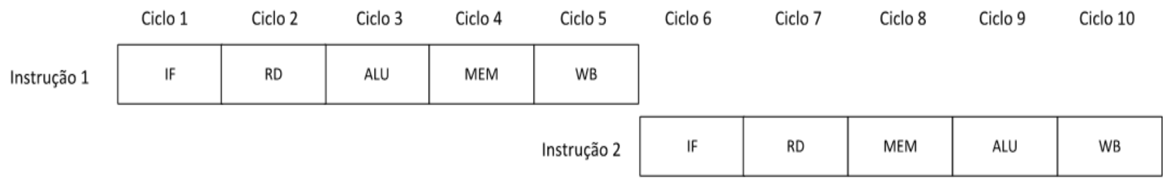


Figura 4.5: Pipeline de instrução única com profundidade 1.

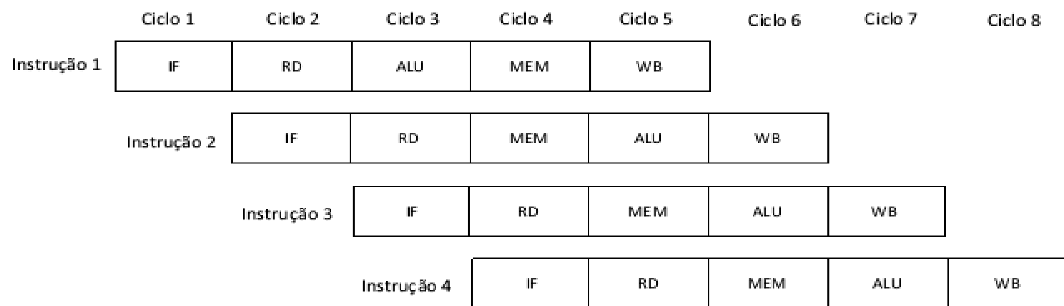


Figura 4.6: Pipeline paralelo com profundidade 5.

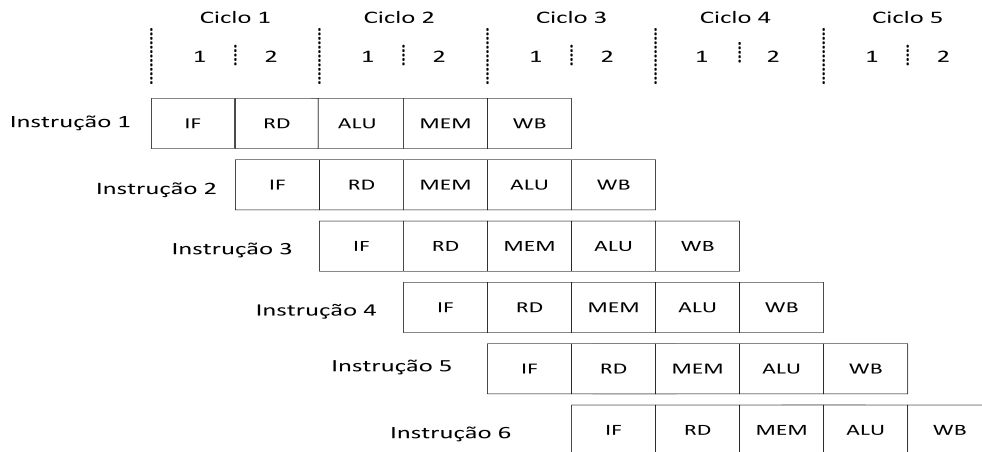


Figura 4.7: Super pipeline com profundidade 5.

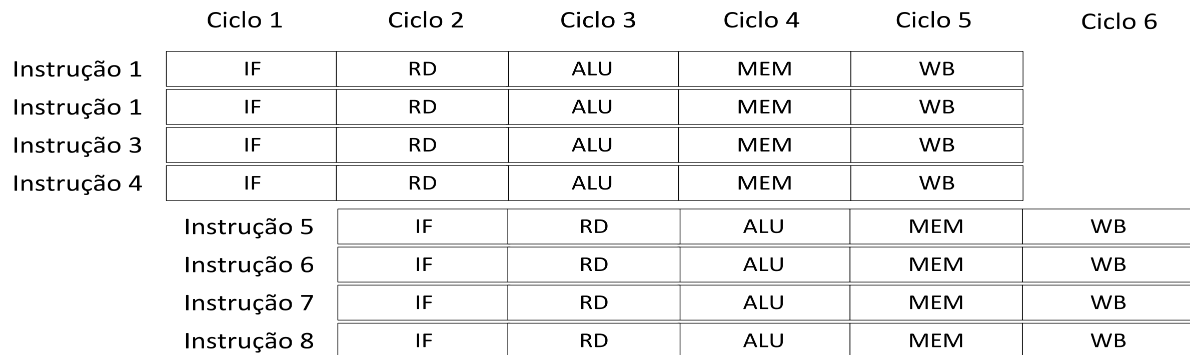


Figura 4.8: Pipeline Superescalar.

Simulação em Software

Uma vez que o *pipelining* é uma técnica de *hardware* concebida para acelerar a velocidade de execução de instruções, no contexto de simulação, a sua utilização é transparente.

O simulador contém uma representação da informação, que será posteriormente convertida e optimizada para código objecto da arquitectura da máquina anfitrião, que por sua vez utilizará técnicas de aceleração em *hardware*. Devido a isso, a divisão das instruções em *software*, não melhora o desempenho de execução do simulador.

Um dos aspectos a ter em consideração relativamente ao *pipelining*, em ambiente de simulação, são os *hazards* gerados quando são executadas instruções de salto ou *branches*, criando assim um estado especial do funcionamento do processador.

A este estado dá-se o nome de "Delay Slot", que ocorre exactamente após a execução de uma instrução de salto, obrigando o processador a executar a instrução seguinte ao salto primeiro que a instrução para onde o salto deveria mudar a execução. Observe-se o seguinte exemplo:

```
.text
          main:
          ....
          j label1
          addi $t0, $t0, 1
          ....

label1:
          add $t0, $t1, $t0
          ....
```

Listing 4.10: Demonstração do comportamento do Delay Branch.

Segundo o exemplo acima, para a arquitectura MIPS de 32 bits, quando a instrução "j label1" estiver no segundo estado, *Read Registers* (RD), a instrução seguinte estará no primeiro estado, *Instruction Fetch* (IF).

Assim, a instrução `addi $t0, $t0, 1` será executada antes da instrução `add $t0, $t1, $t0` uma vez que o processador não tem uma política de remover instruções que já tenham feito "fetch".
No terceiro estado da instrução `j label1` o *Program Counter* (PC) é alterado para o endereço da label "label1" onde a execução continuará.

É bastante comum os geradores de código de linguagens de médio/alto nível fazerem optimizações no código ou apenas introduzirem a instrução `nop`, cuja função é ocupar um

ciclo de relógio no processador, depois de um salto, seja ele condicional ou incondicional, de forma a ultrapassar o problema do "Delay Slot".

4.1.5 Coprocessadores

Os processadores MIPS possuem internamente quatro núcleos de execução conhecidos como coprocessadores. Deste conjunto, apenas o coprocessador zero, apelidado de coprocessador central, é obrigatório para o funcionamento do processador, embora o coprocessador um seja necessário para operações de vírgula flutuante.

Os dois últimos coprocessadores são opcionais, sendo o terceiro livre para implementações específicas do processador e o quarto também dedicado para operações de vírgula flutuante.

4.1.6 Registos Genéricos

Todos os coprocessadores possuem um conjunto de registos genéricos e um conjunto de registos de controlo. Desta forma, o coprocessador central possui internamente 32 registos de 32 bits para utilização genérica. Os programadores possuem ao seu dispor um sub conjunto destes registos para tratadores do *Kernel* do sistema operativo, como exemplo os registos $k0, $k1 ou $at, que é reservado para o assembler em operações de cálculo de endereços. Na tabela 4.13 encontra-se o conjunto de registos genéricos e uma breve discrição.

Nome do Registo	**Número do Registo**	**Descrição**
$0 (zero)	0	Valor sempre zero
$at	1	Reservado ao Assembler
$v0, $v1	2,3	Valores de retorno ou resultados
$a0 ... $a3	4 ... 7	Argumentos de funções
$t0 ... $t7	8 ... 15	Temporários
$s0 ... $s7	16 ... 23	Temporários salvaguardados
$t8, $t9	24 e 25	Temporários que podem ser salvaguardados
$k0 e $k1	26 e 27	Reservado ao Kernel
$gp	28	Global Pointer
$sp	29	Stack Pointer
$fp ou $s8	30	Frame Pointer
$ra	31	Return Address

Tabela 4.13: Tabela com registos genéricos do coprocessador zero.

Além dos 32 registos genéricos o coprocessador zero possui mais 3 registos especiais, respectivamente: *Program Counter* (PC), HI e LO.

O registo PC aponta para a próxima instrução a ser executada e não permite acesso directo por parte do programador. A única forma de modificar o seu valor é a através de instruções de salto definidas na ISA. Os registos HI e LO são utilizados sempre que é executada uma instrução de divisão ou multiplicação. Na multiplicação entre um número de 32 bits com outro número de 32 bits, para valores grandes, obtém-se no máximo um valor de 64 bits, sendo o resultado dividido em dois registos. Caso o resultado seja inferior ao maior número de 32 bits, este é guardado no registo LO, caso contrário será dividido pelos dois registos. Na divisão, o quociente é guardado no registo LO e o resto da divisão guardado no registo HI.

Da mesma forma, o coprocessador um possui 32 registos de 32 bits para executar as suas operações internas. Para implementações de 32 bits os registos são utilizados em pares, dois a dois. Um registo é utilizado para guardar a parte inteira (32 bits) e o registo seguinte para guardar a parte decimal (32 bits)[6].

Simulação em Software

Os registos genéricos encontram-se implementados dentro da estrutura que representa o processador ou coprocessador central.
A sua representação é bastante simples uma vez que estes são compostos por words de 32 bits. Assim optou-se pela criação de um array de words, guardando em cada índice, de acordo com o quadro 4.13, a informação correspondente ao registo. Segue-se a estrutura que representa os registos genéricos:

```
#define CP0_Registos      32
CP0R[CP0_Registos];
```

Listing 4.11: Definição dos registos genéricos na estrutura do processador em C.

O acesso a cada registo é feito sempre através do seu número, utilizado como índice no array de registos. A razão pela qual os números dos registos são utilizados como índice deve-se ao facto de, no processo de interpretação das instruções, os bits extraídos para identificação dos registos genéricos, conhecidos como “rs”, “rd” ou “rt”, possuem sempre 5 bits ($2^5 = 32$ bits).

[6]O formato dos registos genéricos do coprocessador um pode variar com o tipo de implementação.

4.1.7 Registos do Coprocessador Central

Além dos registos genéricos, os coprocessadores possuem internamente um conjunto de registos internos utilizados para descrever e manter configurações, estados dos coprocessadores, assim como: servir de interface entre os mesmos, gerir mecanismos de excepções, gestão da comunicação com periféricos através dos mecanismos de interrupções, entre outros.

Do conjunto de registos internos existentes na arquitectura MIPS32, apenas um subconjunto detém obrigatoriedade de implementação, podendo variar conforme as funcionalidades e mecanismos implementados no processador.

Ao contrário do que acontece com os registos genéricos que possuem um tamanho fixo de 32 bits, os registos internos podem variar o tamanho, sempre em incrementos de 32 bits. Assim os registos internos dispõem de uma variável *Select* que identifica qual o sub registo apontado pelo registo do coprocessador zero. Desta forma, todos os registos internos são compostos pelo número e o SEL.

Segue-se a lista dos registos do coprocessador zero cuja obrigatoriedade de implementação deve ser respeitada.

Registo Nº	**Select**	**Registo**	**Nível de conformidade**	**Estado**
0	0	Index	Obrigatório para TLB MMU	Implementado
1	0	Random	Obrigatório para TLB MMU	Implementado
2	0	EntryLo0	Obrigatório para TLB MMU	Implementado
3	0	EntryLo1	Obrigatório para TLB MMU	Implementado
4	0	Context	Obrigatório para TLB MMU	Implementado
4	1	ContextConfig	Obrigatório para TLB MMU	Não utilizado
4	2	UserLocal	Recomendado para Ver. 2	Implementado
5	0	Pagemask	Obrigatório para TLB MMU	Implementado
5	1	PageGrain	Opcional para Ver. 2	Não utilizado
6	0	Wired	Obrigatório para TLB MMU	Implementado
7	0	HWREna	Obrigatório para Ver. 2	Implementado
8	0	BadVAddr	Obrigatório	Implementado
9	0	Count	Obrigatório	Implementado
10	0	EntryHi	Obrigatório para TLB MMU	Implementado
11	0	Compare	Obrigatório	Implementado
12	0	Status	Obrigatório	Implementado
12	1	IntCtl	Obrigatório para Ver. 2	Não utilizado
12	2	SRSCtl	Obrigatório para Ver. 2	Não utilizado

Registo Nº	Select	Registo	Nível de conformidade	Estado
12	3	SRSMap	Obrigatório para Ver. 2 (SSI)	Não utilizado
13	0	Cause	Obrigatório	Implementado
14	0	EPC	Obrigatório	Implementado
15	0	PRId	Obrigatório	Não utilizado
15	1	EBase	Obrigatório para Ver. 2	Implementado
16	0	Config	Obrigatório	Implementado
16	1	Config1	Obrigatório	Implementado
16	2	Config2	Obrigatório	Implementado
25	0	PerfCnt	Recomendado	Não utilizado
30	0	ErrorEPC	Obrigatório	Implementado

Tabela 4.14: Tabela com registos internos do coprocessador zero.

NOTA: Todos os registos encontram-se implementados e a funcionar correctamente. Os registos com o estado 'Não utilizado' encontram-se igualmente implementados, apenas não são utilizados uma vez que os mecanismos associados não se encontram a funcionar correctamente.

Segue-se a descrição dos registos do coprocessador zero necessários a para implementação da arquitectura MIPS32 revisão 2.

CP0 Registo 4, Select 2 (UserLocal)

O *UserLocal* é um registo que não é interpretado pelo *hardware*, dispondo de permissões de leitura e escrita por *software*. Através deste registo é possível que *software* privilegiado passe informação para *software* não privilegiado através da instrução RDHWR. Para tal ser possível, é necessário que o bit 29 do registo *HWREna* se encontre activo.

31 ... 0
Informação de utilizador

Tabela 4.15: Registo interno UserLocal.

CP0 Registo 7, Select 0 (HWREna)

O registo *HWREna* contém uma máscara que indica quais os registos internos que podem ser lidos através da instrução RDHWR pelo registo *UserLocal*, sempre que o coprocessador zero não se encontre acessível.

A máscara de bits guarda no registo informação de acesso referente aos seguintes conteúdos: CPUNum do registo *EBase*, *SYNCI_Step* relativa à *Cache*, contador do ciclo do relógio do registo *Count*, CCRes para número de ciclos utilizados entre cada actualização do registo *Count*, ULR para acesso ao registo *UserLocal*.

31 30	29 0
Implementado?	Máscara de bits

Tabela 4.16: Registo interno HWREna.

CP0 Registo 8, Select 0 (BadVAddr)

O registo *BadVAddr* possui carácter obrigatório e é utilizado para guardar o endereço virtual da última instrução geradora de uma excepção do tipo: *Address error* (AdEL ou AdES), *TLB Refill*, *TLB Invalid* (TLBL ou TLBS) ou *TLB Modified*.

Sempre que ocorre uma das excepções acima indicadas, o processador é responsável por actualizar o valor do registo *BadVAddr*.

31   0
Endereço virtual que gerou a excepção (BadVAddr)

Tabela 4.17: Registo interno BadVAddr.

CP0 Registo 9, Select 0 (Count)

O registo *Count* actua como um relógio interno, que incrementa o seu valor de forma constante, sendo o seu funcionamento dependente da implementação do processador. Este registo permite a leitura e escrita por *software*, sendo a segunda utilizada para efeitos de diagnóstico, reinicialização ou sincronização de processadores. O mecanismo de *pipeline* no processador utiliza o registo *Count* como relógio para sincronização das instruções.

31   0
Count

Tabela 4.18: Registo interno Count.

CP0 Registo 11, Select 0 (Compare)

O registo *Compare* é utilizado em conjunto com o registo *Count* de forma a criar um relógio e uma interrupção para esse mesmo relógio. Desta forma, o registo *Compare* mantém um valor que assim que for igual ao valor do registo *Count* o processador gerará um pedido de interrupção.

Na revisão 1 o pedido de interrupção do relógio seria gerido em conjunto com a interrupção de *hardware* 5, nos bits *CauseIP*. Na revisão 2 isto já não acontece sendo o pedido de interrupção visto no bit *CauseTI*.

Tabela 4.19: Registo interno Compare.

CP0 Registo 12, Select 0 (Status)

O *Status* é um registo obrigatório com permissões de leitura e escrita, tanto por *hardware* como por *software*, que guarda o modo de operação, assim como estados internos no processador ou tratamento de interrupções.

A sua composição encontra-se dividida em vários blocos, como podemos ver no quadro abaixo, sendo que cada bit ou conjunto de bits representa uma funcionalidade ou estado do processador.

31 28	27	26	25	24	23	22	21	20	19	18	17 16	15 10	9 8	7 6 5	4 3	2	1	0
CU3..0	RP	FR	RE	MX	0	BEV	TS	SR	NMI	ASE	Impl	IM7..2	IM1..0	0	KSU	ERL	EXL	IE

Tabela 4.20: Registo interno Status.

Os 4 bits mais significativos são $CU_{3..0}$ (*Coprocessor Usable*) e servem para identificar se os coprocessadores se encontram acessíveis ou não. Por defeito, independentemente do bit CU0, o coprocessador zero encontra-se sempre acessível se o processador se encontrar a correr em modo *Kernel* ou modo *Debug*.

O bit RP representa a funcionalidade *Reduced Power* e é opcional, sendo a sua implementação em *hardware* dependente do processador. O bit FR (*Floating point register*) tem carácter obrigatório e é utilizado para identificar, caso o processador implemente FPU, o tamanho dos registos internos do coprocessador um. Caso o bit se encontre activo os registos tem tamanho 64bits, caso contrário os registos são guardados em pares de 32 bits. Nos casos em que o processador não implemente FPU este bit deve ser

ignorado.

O bit RE (*Reverse Endianness*) é utilizado para activar/desactivar a funcionalidade de interpretação dos dados na memória. Esta funcionalidade é apenas válida para o modo de *User*, o restantes modos de execução ignoram este bit.

O bit seguinte, MX é opcional e é utilizado para activar as funcionalidades de MDMX e DSP em processadores que implementem ASE.

O bit BEV (*Bootstrap Exception Vector*) é utilizado para modificar os vectores dos pontos de entrada das excepções quando o sistema se encontra na sua inicialização. O seu estado inicial é sempre 1 sendo que, após a inicialização do sistema, o bit se mantém sempre a 0.

O bit TS é utilizado pela TLB sempre que ocorrem múltiplas ocorrências na tabela de mapeamentos. De forma a evitar que o *hardware* fique danificado, a TLB activa este bit gerando em seguida a excepção *Machine Check* para corrigir o erro. A correcção do erro pode ser feita por *software* ou *hardware* sendo a mesma dependente da implementação do processador. Caso a correcção seja feita por *software* o bit TS deve ser limpo antes de se retomar a execução. Alerta-se para o facto que o processador terá um funcionamento indefinido caso o bit seja modificado de 0 para 1 por *software*.

O bit SR indica se o processador se encontra em *Soft Reset* (activo) ou *Non Maskable Interrupt (NMI)* (desligado) quando entra no vector de excepção *Reset, Soft Reset, NMI*, 0xBFC0.0000. Da mesma forma, o bit NMI indica se o processador se encontra em *NMI* (activo) ou *Not NMI* (desligado). Em ambos os casos, os bits são activados por *hardware* e não devem ser modificados por *software* sob pena de funcionamento imprevisível.

O bit ASE é reservado para o MCU ASE, sendo ignorado caso a funcionalidade não se encontre implementada. Os bits 17 e 16 do registo *Status* são reservados para implementações especificas do processador, sendo ignorados caso não sejam utilizados.

Os bits 15 até 10 (IM7 até IM2) são utilizados para controlar as interrupções por *hardware* sendo que, caso o bit se encontre activo, os pedidos de interrupção encontram-se activos, caso contrário encontram-se desligados. Este funcionamento apenas é válido caso o EIC (*External Interrupt Controller*) se encontre desligado. Nos casos em que EIC se encontre activo, estes bits são interpretados de forma diferente, sendo considerados uma IPL (*Interrupt Priority List*). Para informações detalhadas sobre o funcionamento do mecanismo de interrupções ver [MT10c]. Os bits IM1 e IM0 são utilizados para controlar as interrupções por *software* sendo o seu valor ignorado caso o EIC se encontre activo.

Os bits KSU são utilizados para identificar o modo base de operação do processador, sendo os seus valores: 0b00 para modo *Kernel*, 0b01 para modo *Super User*, 0b10 para

modo *User* e 0b11 para modo *Debug*. Para mais informações sobre os modos de operação, ver secção 4.1.1.

O bit ERL (*Error Level*) identifica o estado de erro no processador, sendo o mesmo activado sempre que ocorra um excepção do tipo *Reset, Soft Reset, NMI* ou *Cache error*. Quando este bit se encontra activo, o processador encontra-se a correr em modo *Kernel*, as interrupções são desligadas e a instrução ERET utiliza o registo ErrorEPC em vez do registo EPC para retomar a execução do código binário.

O bit EXL (*Exception Level*) é activado sempre que é gerada uma excepção diferente de *Reset, Soft Reset, NMI* ou *Cache error*. Nestas situações, o processador encontra-se a correr em modo *Kernel*, as interrupções encontram-se desligadas e os registos EPC, CauseBD e SRSCtl não são actualizados.
Nos casos em que ocorra uma excepção e o bit já se encontre activo, o vector da excepção da *TLB Refill* é modificado para o vector *TLB Refill, General Exception*.

Por último temos o bit IE (*Interrupt Enabled*) que funciona como um interruptor principal, que permite ligar e desligar todas as interrupções no sistema. Na revisão 2 surgiram duas novas instruções DI (*Disable Interrupts*) e EI (*Enable Interrupts*), que permitem a modificação deste bit para uso dos programadores.

CP0 Registo 13, Select 0 (Cause)

O registo *Cause* identifica e descrimina a excepção mais recente que ocorreu no processador. É através deste registo que o sistema operativo consegue identificar qual a excepção gerada, os seus atributos e a causa da sua ocorrência.

Na figura seguinte podemos ver a divisão dos bits do registo *Cause*.

31	30	29 28	27	26	25 24	23	22	21	20 18	17 16	15 10	9 8	7	6 2	1 0
DB	TI	CE	DC	PCI	ASE	IV	WP	FDCI	000	ASE	IP9..2	IP1..0	0	ExcCode	0

Tabela 4.21: Registo interno Cause.

O bit BD (*Delay Branch*) indica se a última excepção que ocorreu se encontrava a ser executada em modo normal ou *Delay Branch*.

O bit TI (*Timer Interrupt*) é utilizado pela arquitectura MIPS revisão 2 para validar se existem pedidos de interrupção do relógio por atender.

Os bits CE (*Coprocessor Enabled*) guarda o valor do coprocessador que gerou a excepção *Coprocessor Unusable*.

O bit DC é utilizado para ligar/desligar o mecanismo utilizado pelo registo *Count*, uma vez que a dissipação de energia é notável em aplicações sensíveis ao consumo de energia. O bit PCI é utilizado para activar/desligar a interrupção gerada pelo registo *Performance Counter*.

O bit IV (*Interrupt Vector*) define qual o vector de excepções usado pelo mecanismo de interrupções, 0x180 caso desligado, 0x200 caso se encontre activo.

Os bits $IP_{7..2}$ (*Interrupt Pending*) indicam se existe alguma interrupção de *hardware* pendente. O mesmo acontece com os bits $IP_{1..0}$ que identificam se existem pedidos de interrupção pendentes por *software*. Assim como acontece com o registo *Status*, se o EIC se encontrar implementado e activo, os bits *CauseIP* são interpretados de forma diferente.

Os bits *ExcCode* guardam o código identificativo da última excepção que ocorreu no processador. No quadro seguinte podemos a codificação das excepções existentes na arquitectura MIPS32 Revisão 2.

Decimal	**Hexadecimal**	**Código**	**Descrição**
0	0x00	Int	*Interrupt*
1	0x01	Mod	Excepção *TLB Modified*
2	0x02	TLBL	Excepção *TLB Load*
3	0x03	TLBS	Excepção *TLB Store*
4	0x04	AdEL	Excepção *Address Error on Load*
5	0x05	AdES	Excepção *Address Error on Store*
6	0x06	IBE	*Bus error exception on instruction fetch*
7	0x07	DBE	*Bus error exception on load or store*
8	0x08	Sys	Excepção *Syscall*
9	0x09	Bp	Excepção *Break Point*
10	0x0A	RI	Excepção *Reserved Instruction*
11	0x0B	CpU	Excepção *Coprocessor Unsusable*
12	0x0C	Ov	Excepção *Arithmetic Overflow*
13	0x0D	Tr	Excepção *Trap*
14	0x0E	-	Reservado
15	0x0F	FPE	Excepção *Floating Point*
16	0x10	-	Reservado
17	0x11	-	Reservado
18	0x12	C2E	Excepção reservada para o Coprocessador 2
19	0x13	TLBRI	Excepção *TLB Read-Inhibit*
20	0x14	TLBXI	Excepção *TLB Execution-Inhibit*
21	0x15	-	Reservado

Decimal	Hexadecimal	Código	Descrição
24	0x18	MCheck	Excepção *Machine Check*
25	0x19	Thread	Excepções reservadas para MIPS MT ASE
26	0x1A	DSPDis	Excepção *DSP ASE State Disabled*
27	0x1B	-	Reservado
28	0x1C	-	Reservado
29	0x1D	-	Reservado
30	0x1E	CacheErr	Excepções de *Cache error*
31	0x1F	-	Reservado

Tabela 4.22: Tabela de códigos por execpão para a arquitectura MIPS32.

CP0 Registo 14, Select 0 (EPC)

O *Exception Program Counter (EPC)* é um registo com permissões de leitura e escrita que o processador utilizará, após o tratamento de todas as excepções, para retomar a sua execução.

Nos casos em que o valor do *StatusEXL* se encontre a 1 e surja uma excepção, o processador não modificará o valor do EPC, dado que se encontra a executar código do tratador de uma excepção e será utilizado o vector geral de excepções. Nestes casos, o EPC será igual em ambas as excepções dado que após o seu tratamento, a instrução geradora da excepção será executada novamente e o código do tratador será igualmente executado, agora com a correcção da segunda excepção.

Alerta-se para o facto de as interrupções por *hardware* e *software* serem desligadas assim que o bit *StatusEXL* é activado, não permitindo a modificação do EPC por parte das interrupções.

Para excepções do tipo síncronas ou precisas, o valor do EPC pode variar conforme o valor do bit *CauseBD*. Se este bit se encontrar desligado o valor do EPC aponta para o endereço da instrução geradora da excepção; caso contrário este apontará para o endereço da instrução anterior à instrução que se encontra no *Branch Delay*.

Em excepções assíncronas, caso das interrupções, o registo EPC contém o endereço da instrução para onde o processador deverá retomar a sua execução.

Tabela 4.23: Registo interno EPC.

CP0 Registo 15, Select 0 (PRId)

O registo PRId (*Processor Identification*) é um registo apenas de leitura, escrito por *hardware*, com informação relativa à identificação e nível de revisão do processador.

Para informação detalhada sobre os modelos disponíveis, fabricante ou revisão, ver [MT10c].

31 24	23 16	15 8	7 0
Opções da empresa	ID da empresa	ID do processador	Revisão

Tabela 4.24: Registo interno PRId.

CP0 Registo 15, Select 1 (EBase)

O *EBase* é um registo que surgiu na implementações da revisão 2 da arquitectura MIPS32 com carácter obrigatório. Este guarda a base utilizada no cálculo do endereço de entrada dos tratamentos das excepções. Nos casos em que o bit *StatusBEV* se encontre activo, inicialização do processador, o registo *EBase* contém os bits mais significativos com o valor 0xBFC0.xxxx de forma a manter os endereços base gerados na zona alta do segmento KSEG1, podendo os valores para xxxx variar em função do tipo de excepção.

Nos casos em que o bit *StatusBEV* se encontre desligado, o endereço base é calculado utilizando os bits 31 e 30, sempre com o valor 0b10, forçando o endereço gerado ao domínio do KSEG0 (0x8000.0000), mantendo assim compatibilidade com versões anteriores da arquitectura MIPS32. Sempre que ocorra uma excepção do tipo *cache error* o bit 29 é obrigatoriamente 1 forçando o seu valor ao domínio do KSEG1 (0xA000.0000), segmento não mapeado e sem utilização de *cache*. Para os restantes casos, o *EBase* possui o valor 0b10 || 0x000, gerando o endereço 0x8000.0000. Se o registo *EBase* for modificado com o *StatusBEV* desligado, o processador terá um funcionamento indefinido uma vez que, por defeito e em ambientes com um único processador, os bits $EBase_{29...12}$ encontram-se a zero.

Os 10 bits menos significativos são utilizados para identificar o processador em execução em ambientes com multiprocessadores. Nestes casos, o valor dos bits $EBase_{29...12}$ pode variar, permitindo a existência de vectores de excepções distintos para os vários processadores presentes.

31	30	29 ... 12	11 10	9 ... 0
1	0	Base da Excepção	0 0	Número do CPU

Tabela 4.25: Registo interno EBase.

CP0 Registo 16, Select 0 (Config)

O registo *Config* possui um conjunto de configurações relativas às funcionalidades e mecanismos implementados no processador. A grande maioria dos bits são carregados por *hardware* no arranque no processador sendo os restantes configuráveis através da excepção *Reset*.

31	30 28	27 25	24 16	15	14 13	12 10	9 7	6 4	3	2 0
M	K23	KU	Impl	BE	AT	AR	MT	0	VI	K0

Tabela 4.26: Registo interno Config.

Segue-se a lista de parâmetros configuráveis no processador para o registo *Config0*:

Atributo	**Função**
M	Indica se o registo *Config1* se encontra implementado e configurado.
K23	Dedicado para MMU's com mapeamentos fixos como o FMT.
KU	Atributos de *cache* para MMU com mapeamentos fixos.
Impl	Reservado para implementações do processador.
BE	Big Endian (1) ou Little Endian (0)
AT	Tipo de arquitectura: MIPS32, MIPS64 com e sem restrições de acesso à memória.
AR	Nível de revisão da arquitectura.
MT	Tipo de MMU: TLB, BAT, FMT ou Dual VTLB e FTLB.
VI	*Cache* de instruções virtuais.
K0	Atributos de *cache* para segmento KSEG0.

Tabela 4.27: Tabela de atributos do registo Config0.

CP0 Registo 16, Select 1 (Config1)

O registo *Config1* funciona em conjunto com o registo *Config0* com a diferença que todos os seus campos são escritos por *hardware*, sendo de leitura exclusiva para *software*.

31	30 25	24 22	21 19	18 16	15 13	12 10	9 7	6	5	4	3	2	1	0
M	MMU Size-1	IS	IL	IA	DS	DL	DA	C2	MD	PC	WR	CA	EP	FP

Tabela 4.28: Registo interno Config1.

Segue-se a lista de parâmetros existentes no registo *Config1*:

Atributo	**Função**
M	Indica se o registo *Config2* se encontra implementado e configurado.
MMU Size-1	Número de entradas na TLB -1.
IS	Configurações de *cache*
IL	Configurações de *cache*
IA	Configurações de *cache*
DS	Configurações de *cache*
DL	Configurações de *cache*
DA	Configurações de *cache*
C2	Coprocessador 2 implementado?
MD	Processador contem suporte para MDMX?
PC	Registos *Performance Counter* implementados?
WR	Registos *Watch* implementados?
CA	Compressão de código? (MIPS16e)
EP	Módulo EJTAG implementado?
FP	Unidade FPU *Floating Point Unit* implementada?

Tabela 4.29: Tabela de atributos do registo Config1.

CP0 Registo 16, Select 2 (Config2)

Os registos *Config2* e *Config3* contém as codificações e configurações especificas da *cache* no processador.

O bit mais significativo M indica se o registo *Config3* se encontra implementado no processador, caso contrário este retornará 0. Sendo os restantes bits utilizados para configurações da *cache*.

31	30 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
M	TU	TS	TL	TA	SU	SS	SL	SA

Tabela 4.30: Registo interno Config2.

CP0 Registo 30, Select 0 (ErrorEPC)

O *ErrorEPC* é um registo que permite a escrita, por *software* e *hardware*, e cujo funcionamento é similar ao registo EPC, com a diferença que é utilizado exclusivamente em situações de erro ou excepções do tipo *Reset*, *Soft Reset*, *NMI (Nonmaskable Interrupt)* e *Cache errors*.

Sempre que ocorre uma das excepções acima citadas, o bit *StatusERL* encontra-se activo e o valor do ErrorEPC, lido pela instrução ERET no seu retorno, aponta para o endereço virtual da instrução geradora da excepção ou o endereço anterior da instrução no *Branch Delay Slot*.

Tabela 4.31: Registo interno ErrorEPC.

Os registos directamente relacionados com a utilização e interface com a TLB serão abordados no capítulo 4.3.1.

Simulação em Software

Em oposição ao funcionamento dos registos genéricos, os registos internos possuem uma representação um pouco diferente uma vez que estes são dinâmicos podendo o seu tamanho variar. Pegue-se no seguinte bloco de código utilizado para representar os registos internos, descritos em 4.1.7, na estrutura do processador:

```
/* Registos do Coprocessador 0
 */
CP0_Entry *CP0_Reg[32];

//Estrutura de dados CP0_Entry
typedef struct CP0_Entry{

        //Numero do Registo
        int Numero;

        //Sel - Registos com Informacao
        GPR Sel[8];

        //Nivel de obrigatoriedade
        char Comp[32];
```

```
} CP0_Entry;
```

Listing 4.12: Definição dos registos internos na estrutura do processador em C.

Do conjunto de registos existentes no coprocessador zero, apenas um sub conjunto possui caracter obrigatório de implementação. Esta necessidade varia conforme os mecanismos implementados ou disponíveis pelo *hardware*. Um exemplo deste caso são os registos que controlam o funcionamento do mecanismo TLB, nomeadamente os registos: *Index*, *Random*, *EntryHi*, *EntryLo0*, *EntryLo1* ou *Pagemask*. Cuja obrigatoriedade varia com a presença do módulo no processador, caso contrário estes registos são opcionais.

A necessidade de definir uma estrutura para estes registos nasce do facto de que o seu tamanho pode variar, existindo assim registos com tamanho superior a 32 bits[7]. Por definição, a arquitectura MIPS32 permite que existam registos compostos por um ou mais sub registos de 32 bits. Desta forma existe um número para identificar o registo no coprocessador zero e um campo apelidado de "SEL" para identificar o registo associado a esse número. Existe também um nível de obrigatoriedade associada a cada registo, que de acordo com os mecanismos implementados e a revisão da arquitectura utilizada, actualizará estaticamente na inicialização do processador o seu valor para: "Opcional", "Obrigatório", "Recomendado", "Obrigatório Ver. 2" ou "Obrigatório MMU/TLB".

De modo a simplificar os exemplos, será utilizada a nomenclatura (Num:Sel) para identificar o par: número de registo do coprocessador zero e respectivo sub registo associado. O registo zero do coprocessador é o registo *Index* (0:0), que é utilizado pelo mecanismo TLB, cuja obrigatoriedade de implementação depende do facto de a mesma se encontrar presente e implementada no processador. Este registo é composto por apenas um sub registo.

Em oposição ao registo zero, o registo doze do coprocessador zero é composto por quatro sub registos: *Status* (12:0), *IntCtl* (12:1), *SRSCtl* (12:2) e *SRSMap* (12:3), todos com tamanho 32bits.

Com a estrutura definida no bloco de código anterior (4.12), é possível abranger e implementar todos os registos internos necessários ao funcionamento da arquitectura MIPS32.

[7] O tamanho dos registos varia em incrementos de 32 bits uma vez que cada sub registo possui um tamanho de 32 bits.

4.1.8 Mecanismo de Excepções

Com o decorrer normal da execução, podem surgir situações que obriguem o processador a interromper o seu fluxo de execução. Tais eventos são conhecidos como excepções e podem ser de dois tipos: excepções geradas pela execução de instruções no processador ou por pedidos que não se encontram directamente relacionados com o funcionamento do processador, como por exemplo os pedidos de interrupções dos periféricos.

Desta forma, existem excepções do tipo síncronas ou assíncronas. As excepções síncronas ocorrem exactamente no momento de execução da instrução no processador, sendo associadas à instrução actual no processador. No caso das excepções assíncronas, a excepção não é gerada no momento em que é detectada mas sim posteriormente, quando o processador pode atender o pedido. Nestes casos, a excepção não se encontra directamente associada à instrução executada no processador. São geralmente associados a pedidos de interrupção externas, pedidos de *Reset* ou validação de inconsistências no processador.

As prioridades das excepções não são iguais, variando com a categoria onde se inserem, respectivamente por relevância: Assíncrona Reset, Assíncrona Debug, Assíncrona, Síncrona Debug e Síncrona. No quadro 4.32 podemos ver os tipos de excepções associadas aos modos síncronos e assíncronos.

Sempre que ocorre uma excepção, o processador entra em modo *Kernel*, guarda um conjunto de dados relativos ao seu estado de execução e salta para um endereço pré calculado com o objectivo de executar código binário designado para aquela excepção. O endereço utilizado pelo processador para tratar a excepção é conhecido como ponto de entrada (*EntryPoint*). Este ponto de entrada é calculado utilizando um conjunto de vectores previamente definidos para a arquitectura MIPS32. Desta forma, os endereços são calculados utilizando duas componentes, a base e o seu deslocamento.

Na revisão 2 da arquitectura MIPS32, o cálculo da base pode ser definido por *software* através do registo EBase de forma permitir a customização dos vectores de excepções. O valor do EBase pode ser modificado caso o *StatusBEV* se encontre a zero, caso contrário a operação não é permitida uma vez que no arranque do sistema os vectores de excepções são fixos. No quadro 4.33 pode-se visualizar a forma como é gerado a base para os pontos de entrada.

As excepções *Reset, Soft Reset e NMI* são um caso especial, uma vez que não utilizam deslocamento no cálculo do seu endereço sendo o seu valor sempre, independentemente do estado do *StatusBEV*, 0xBFC00000.

Excepção	**Tipo**	**Prioridade**
Reset	Assíncrona (Reset)	1
Soft Reset	Assíncrona (Reset)	1
Debug Single Step	Síncrona (Debug)	4
Debug Interrupt	Assíncrona (Debug)	2
Imprecise Data Break	Assíncrona (Debug)	2
NMI	Assíncrona	3
Machine Check	Assíncrona	3
Interrupt	Assíncrona	3
Deferred Watch	Assíncrona	3
Debug Instruction Break	Síncrona (Debug)	4
Watch	Síncrona	5
Address Error	Síncrona	5
TLB Refill	Síncrona	5
TLB Invalid	Síncrona	5
TLB Execute	Síncrona	5
Cache Error	Síncrona	5
Bus Error	Síncrona	5
SDBBP	Síncrona (Debug)	4

Tabela 4.32: Quadro de excepções síncronas e assíncronas com respectivas prioridades. (1 maior prioridade, 5 menor prioridade)

<table>
<tr><th rowspan="2">Excepção</th><th colspan="2">StatusBEV</th></tr>
<tr><th>0</th><th>1</th></tr>
<tr><td>Reset, Soft Reset e NMI</td><td colspan="2">0xBFC00000</td></tr>
<tr><td>EJTAG Debug (ProbTrap=0)</td><td colspan="2">0xBFC00480</td></tr>
<tr><td>EJTAG Debug (ProbTrap=1)</td><td colspan="2">0xFF200200</td></tr>
<tr><td>Cache Error</td><td>Para a revisão 1:
0xA0000000
Para a revisão 2:
EBase $_{31..30}$ || 1 || EBase $_{28..12}$
|| 0x000</td><td>0xBFC00200</td></tr>
<tr><td>Restantes
Excepções</td><td>Para a revisão 1:
0x80000000
Para a revisão 2:
EBase $_{31..12}$ || 0x000</td><td>0xBFC00200</td></tr>
<tr><td colspan="3">Note-se que o valor de EBase $_{31..30}$ = 0b10.</td></tr>
</table>

Tabela 4.33: Cálculo do endereço base para os pontos de entrada.

Nos restantes casos, se o *StatusBEV* se encontrar activo, o endereço base utilizado é 0xBFC00 sendo os 12 bits menos significativos utilizados como deslocamento. Se o bit *StatusBEV* se encontrar desactivado, para excepções do tipo *cache*, na revisão 2, o endereço é calculado utilizando a expressão:

```
Endereço Base = EBase 31..30 || 1 || EBase 28..12 || 0x000
```

Os bits 31 e 30 são estáticos (0x10) de forma a manter compatibilidade com a revisão 1, uma vez que o seu endereço base era o 0x80000000. O bit 29 tem de estar obrigatoriamente activo para forçar o endereço ao domínio do segmento KSEG1, que é um segmento não mapeado e sem utilização de *cache*. Relativamente aos bits EBase $_{28..12}$, estes são responsabilidade do sistema operativo de forma a criar os vectores base para cada excepção, sendo os últimos doze bits escritos com zeros.

Para todas as restantes excepções, o cálculo do endereço é igual com a excepção que o bit 29 não se encontra activo, respectivamente:

```
Endereço Base = EBase 31..30 || EBase 29..12 || 0x000
```

Relativamente aos 12 bits menos significativos, utilizados para o deslocamento dos vectores de excepção, estes podem ser dos seguintes tipos.

Excepção	**Vector Deslocamento**
TLB Refill (StatusEXL = 0)	0x000
Cache error	0x100
Genérica	0x180
Interrupção (CauseIV=1)	0x200
Reset, Soft Reset, NMI	Não aplicável

Tabela 4.34: Quadro com deslocamentos utilizados no cálculo de pontos de entrada.

Os vectores de deslocamento são fixos, podendo variar em função de determinados bits no processador. Um exemplo destes casos é a excepção *TLB Refill*, cujo o deslocamento varia em função do *StatusEXL* e endereço base em função do *StatusBEV*.

No quadro 4.35 pode-se ver o conjunto de pontos de entrada definidos para a arquitectura MIPS32 Revisão 2 em função dos bits *StatusEXL*, *StatusBEV*, *CauseIV* e *EJTAG ProbTrap*, assumindo que o valor do registo EBase encontra-se no seu estado inicial.

Excepção	StatusBEV	StatusEXL	CauseIV	EJTAG ProbTrap	Vector de entrada
Reset, Soft Reset e NMI	x	x	x	x	0xBFC00000
EJTAG Debug	x	x	x	0	0xBFC00480
EJTAG Debug	x	x	x	1	0xFF200200
TLB Refill	0	0	x	x	0x80000000
TLB Refill	0	1	x	x	0x80000180
TLB Refill	1	0	x	x	0xBFC00200
TLB Refill	1	1	x	x	0xBFC00380
Cache Error	0	x	x	x	0xA0000100
Cache Error	1	x	x	x	0xBFC00300
Interruption	0	0	0	x	0x80000180
Interruption	0	0	1	x	0x80000200
Interruption	1	0	0	x	0xBFC00380
Interruption	1	0	1	x	0xBFC00400
Outras excepções	0	x	x	x	0x80000180
Outras excepções	1	x	x	x	0xBFC00380
Note-se que o valor "x"indica não interessa e o EBase possui o valor inicial.					

Tabela 4.35: Cálculo do endereço base para os pontos de entrada.

Tratamento de excepções

Como abordado anteriormente, sempre que ocorre uma excepção o processador suspende o fluxo de execução, calcula o endereço que possui o respectivo tratador da excepção e modifica o PC para o endereço calculado. Em seguida, será abordado a metodologia utilizada pela arquitectura MIPS para resolução das excepções geradas.

Segundo [Swe06] o processo de execução de uma excepção é composto por seis fases, sendo elas:

1. Arranque
Assim que o processador salta para o tratador da excepção, pouco ou nenhuma informação relativa ao programa interrompido se encontra resguardada. Desta forma, o primeiro passo é salvaguardar esta informação em memória. Na arquitectura MIPS32 revisão 1 esta tarefa é exclusiva do sistema operativo, sendo que na revisão 2 da arquitectura tenha sido introduzida uma funcionalidade conhecida como *Shadow Set Registers*, totalmente em *hardware*, que guarda cópias dos registos genéricos sempre que ocorre uma interrupção ou excepção, com o objectivo de preservação.

Por convenção, apenas os registos \$k0 e \$k1 podem/ devem ser utilizados nos tratamentos de excepções de baixo nível, sendo utilizados por *software* para copiar os registos

genéricos para a memória de forma a permitir a utilização total dos mesmos registos.

A utilização dos *Shadow Set Registers* é uma poderosa funcionalidade com grande impacto no desempenho total do sistema, uma vez que a ocorrência de excepções e interrupções num sistema é muito elevada. Além deste facto, o resguardo dos registos é feito automaticamente por *hardware*, libertando assim a necessidade de execução de blocos de código estáticos como: "guardar registos genéricos na memória" e "repor registos genéricos da memória", tanto no inicio como no fim do tratamento de cada excepção.

2. Identificação da excepção
O processo de identificação da excepção é feito através dos bits *CauseExcCode*, que podem ser consultados na tabela 4.22. É através destes bits que o sistema operativo consegue identificar qual a excepção que ocorreu, assim como as causas do seu acontecimento e em seguida determinar o tratador adequado.

3. Construção do ambiente de processamento
O desenvolvimento de sistemas operativos, assim como o tratamento de excepções, é feito utilizando linguagens de alto nível, que por sua vez possuem um conjunto de bibliotecas *standard* necessárias para a sua utilização. De forma a possibilitar a utilização destas bibliotecas, é necessário que o sistema operativo reserve espaço na memória como STACK, a fim de preservar registos internos ou genéricos das bibliotecas que possuam permissões de modificação.

Em sistemas Linux [Lov10], cada processo, possui internamente um descritor de processo, composto pela estrutura `thread_info`, que é armazenado em duas páginas consecutivas em memória. Estas páginas possuem 8KBytes (4K + 4K) de tamanho e partilham o mesmo espaço de endereçamento para guardar informação relativa à estrutura do processo, assim como a STACK para o modo de execução *Kernel*, como se pode ver na figura 4.9.

Desta forma, os processos podem correr em modo *Kernel* e em modo *User*, sendo a STACK para o primeiro modo composta exclusivamente por estas duas páginas físicas, e no segundo modo podendo variar com o restante domínio de endereços disponíveis para o processo. É esta STACK no processo interrompido que será utilizada pelo sistema operativo para resguardar os registos genéricos que poderão ser sobrepostos com a utilização de bibliotecas *standard*, independentemente do tipo de excepção: síncrona ou assíncrona.

4. Processamento da excepção
Nesta fase, o tratador da excepção encontra-se pronto para correr o código binário adequado ao tipo de excepção identificada nas fases anteriores, sem restrições de acesso aos registos genéricos do coprocessador zero.

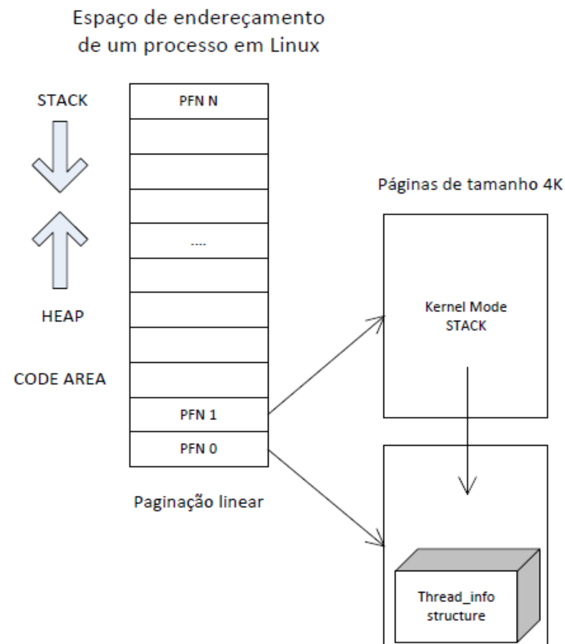


Figura 4.9: STACK para modo de execução Kernel em sistemas Linux.

5. Preparação para retorno
Uma vez que o espaço para cada tratador é limitado, as funções de alto nível são chamadas em sub-rotinas retornando para baixo nível no seu retorno. No fim do tratamento da excepção, é obrigatório repor os valores dos registos genéricos e dos registos de controlo no processador, podendo estes variar em função do tipo de excepção como abordado mais à frente neste capítulo.

6. Retorno da excepção
A última fase no tratamento de excepções é executar o seu retorno, que é feito através da execução da instrução ERET.

Esta instrução é responsável por identificar qual o endereço de retorno, guardado nos registos EPC e ErrorEPC, dependendo do estado do bit *StatusERL*.
Na sua execução, os bits *StatusEXL* e *StatusERL* são desligados em função do registo de retorno utilizado, sendo o modo de operação definido com o valor dos bits *StatusKSU*.
Para a revisão 2 da arquitectura MIPS32, os registos genéricos e internos guardados no *Shadow Set Registers* são repostos por *hardware* automaticamente.

Excepções aninhadas (Nested Exceptions)

O conceito de excepções aninhadas surgiu do facto de ser possível ocorrer uma excepção dentro do código de outra excepção. Esta pode ser gerada de forma síncrona, directamente pela execução de uma instrução, ou assíncrona através de dispositivos externos. Em ambos os casos, este acontecimento geraria caos no sistema uma vez que os registos *Status*, *Cause*, *EPC* e *ErrorEPC* são modificados com a geração de uma excepção.

De forma a ultrapassar este problema, é necessário ter em consideração cuidados adicionais que consistem em resguardar um conjunto de registos, genéricos e internos, na memória. De forma a permitir que uma excepção sobreviva à ocorrência de outra excepção, é necessário guardar os registos de forma estruturada na STACK do processo interrompido. Esta estrutura é conhecida como *Exception Frame* e a cada nova excepção é instanciada uma nova estrutura e guardada na STACK, criando assim um ambiente de execução para a nova excepção. Os recursos existentes na STACK são consumidos e destruídos à medida que as excepções são tratadas, uma vez que o crescimento do aninhamento de excepções descontrolado não é tolerado.

É bastante comum os sistemas operativos atribuírem um nível de prioridade para excepções, permitindo ou não, a geração de novas excepções dentro de excepções em execução. Desta forma, as *frames* alocadas na STACK só crescerão até um nível máximo de prioridades definidas. Um exemplo deste funcionamento é o tratamento de interrupções, no qual podem ser gerados vários tipos excepções na sua execução, nomeadamente *syscalls* uma vez que o sistema é construído utilizando linguagens de alto nível.

Relembro que no caso das interrupções é possível por *software*, ligar e desligar as mesmas, através das instruções DI e EI, introduzindo assim um nível de controlo maior para o sistema operativo.

Funcionamento das Excepções

Como abordado anteriormente neste capítulo, a arquitectura MIPS32 dispõe de um conjunto de excepções que são geradas em situações específicas. Em seguida, serão identificados os tipos de excepções existentes e a suas respectivas descrições assim como o seu funcionamento particular.

1. **Excepção EJTAG Debug**
As excepções deste tipo são geradas quando uma das condições associadas ao mecanismo EJTAG se encontrar activo. Para mais informações, ver a especificação do mecanismo EJTAG em `http://www.mips.com`.

O vector de excepção utilizado é 0xBFC00480, caso o *ProbTrap* bit do registo "EJTAG_Control_Register" se encontrar desligado. Se o bit se encontrar activo, o vector de excepção utilizado é 0xFF200200.

2. Excepção Reset
A excepção *Reset* ocorre quando é introduzido o sinal de "Cold Reset" no processador, levando a uma reinicialização total do mesmo, abortando todas as máquinas de estados internos e repondo os registos do coprocessador zero com o seu valor inicial.

O registo *CauseExcCode* não é actualizado uma vez que não existe nenhum código pré definido para este tipo de excepção. Os seguintes registos do coprocessador zero são modificados com valores pré-definidos:

Estados actualizados	**Bits**	**Valor**
Cause	DC	0
EBase	ExceptionBase	0
Random	-	Tamanho da TLB - 1
Wired	-	0
Config	-	Estado de arranque
Config1	-	Estado de arranque
Config2	-	Estado de arranque
Config3	-	Estado de arranque
Status	RP	0
Status	BEV	1 (Bootstrap)
Status	TS	0
Status	SR	0
Status	NMI	0
Status	ERL	1
Registos Watch	-	0
Performance Counter	IE	0
ErrorEPC	-	PC de reinicialização
PC	-	0xBFC0 0000

Tabela 4.36: Estados modificados pela excepção Reset.

Para informações específicas da operação de *Reset* no processador, ver [MT10c]. O vector de excepção utilizado é o 0xBFC0 0000.

3. Excepção Soft Reset
A excepção *Soft Reset*, da mesma forma que a excepção *Reset*, ocorre quando é enviado por *hardware* um sinal de *Reset* no processador. Este não realiza todo o processo de reinicialização do processador, apenas um sub conjunto das operações. Assim o processador

é reinicializado num estado no qual possa correr instruções da memória não mapeada e sem utilização de *cache*, uma vez que operações de *bus* ou *cache* foram interrompidas, levando a que alguns estados internos no processador possam estar inconsistentes.

A grande diferença entre os dois tipos de *Reset*, é que o "Cold Reset" é utilizado na inicialização do *hardware*, "Power Up", enquanto o *Soft Reset* é utilizado em situações em que o processador fique bloqueado ou não responda, conhecido como "hung". Uma vez que o sinal é introduzido por *hardware*, o CauseExcCode não é actualizado. Os seguintes registos do coprocessador zero são modificados com valores pré-definidos:

Estados actualizados	**Bits**	**Valor**
Status	RP	0
Status	BEV	1 (Bootstrap)
Status	TS	0
Status	SR	1
Status	NMI	0
Status	ERL	1
Registos Watch	-	0
Performance Counter	IE	0
ErrorEPC	-	PC de reinicialização
PC	-	0xBFC0 0000

Tabela 4.37: Estados modificados pela excepção Soft Reset.

Para informações específicas da operação de *Soft Reset* no processador, ver [MT10c]. O vector de excepção utilizado é o 0xBFC0 0000.

4. Excepção Non Maskable Interrupt (NMI)
Este tipo de excepção ocorre sempre que é introduzido o sinal de NMI por *hardware* no processador. O seu funcionamento é muito semelhante ao das interrupções com a diferença que esta não pode ser mascarada, uma vez que a sua natureza obriga que a resposta a um determinado evento seja efectuado sem demora, como erros de *hardware* não recuperáveis (bloqueios internos).

Ao contrário do que acontece com as duas excepções anteriores, esta não tem carácter de reinicialização, sendo as máquinas de estados no processador, *cache* ou memória mantidos sem modificações, à excepção dos seguintes registos:

Estados actualizados	**Bits**	**Valor**
Status	BEV	1 (Bootstrap)
Status	TS	0
Status	SR	0
Status	NMI	1
Status	ERL	1
ErrorEPC	-	PC de reinicialização
PC	-	0xBFC0 0000

Tabela 4.38: Estados modificados pela excepção NMI.

O vector de excepção utilizado é o 0xBFC0 0000.

5. Excepção Machine Check

A excepção *Machine Check* ocorre sempre que o processador detecte inconsistências na sua execução interna por *hardware*. Para processadores que implementem o mecanismo de tradução de endereços TLB, nos casos em que sejam detectadas múltiplas entradas na TLB através da instrução TLBP, o processador gerará a excepção *Machine Check* por *hardware* e activará o bit StatusTS.

Estados actualizados	**Bits**	**Valor**
Cause	ExcCode	MCheck

Tabela 4.39: Estados modificados pela excepção Machine Check.

O vector de excepção utilizado é o *General Exception Vector* com o deslocamento 0x180.

6. Excepção Address Error
Este tipo de excepção ocorre nas seguintes circunstâncias:

1. É efectuada a introdução de uma instrução no processador, fase *instruction fetch*, cujo endereço não se encontre alinhado num limite de word.

2. É executada uma instrução do tipo LOAD ou STORE, com um endereço não-alinhado ao limite de uma word ou half word.

3. Referência a zonas de *Kernel* feitas com o processador em modo de execução *User* ou *Super User*.

4. Referência a zonas de *Super User* feitas com o processador em modo de execução *User*.

Se a operação for um LOAD, o *CauseExcCode* é actualizado para AdEL, caso seja uma operação de STORE o registo é modificado para AdES.

Estados actualizados	**Bits**	**Valor**
Cause	ExcCode	AdEL ou AdES
BadVAddr	-	Endereço que falhou o acesso
Context	VPN2	Imprevisível
EntryHi	VPN2	Imprevisível
EntryLo0	-	Imprevisível
EntryLo1	-	Imprevisível

Tabela 4.40: Estados modificados pela excepção Address Error.

O vector de excepção utilizado é o *General Exception Vector* com o deslocamento 0x180.

7. Excepção TLB Refill

A excepção *TLB Refill* ocorre quando, em processadores que implementem o mecanismo TLB, o endereço virtual pedido pela MMU não se encontra mapeado na TLB.
Sempre que a excepção ocorre, independentemente do estado do StatusEXL, o Cause-ExcCode é preenchido com TLBS para pedidos de STORE ou com TLBL para pedidos de LOAD.

Estados actualizados	**Bits**	**Valor**
Cause	ExcCode	TLBS ou TLBL

Tabela 4.41: Estados modificados pela excepção TLB Refill.

Se o StatusEXL se encontrar desligado na altura da excepção, o vector utilizado será o *TLB Refill Vector* com o deslocamento 0x000. Caso o StatusEXL se encontrar activo na altura da excepção, o vector utilizado será o *General Exception Vector* com o deslocamento 0x180.

Para informações detalhadas sobre o funcionamento da *TLB Refill* ver 4.3.4 .

8. Excepção Execute-Inhibit

Esta excepção ocorre sempre que um endereço virtual pedido pela MMU se encontre mapeado na TLB mas possua o bit XI activo. A excepção *Execute-Inhibit* só é válida para processadores que implementem os bits XI activos no registo PageGrainXIE.

Estados actualizados	Bits	Valor
Cause	ExcCode	TLBL ou TLBXI

Tabela 4.42: Estados modificados pela excepção TLB Execute-Inhibit.

O vector de excepção utilizado é o *General Exception Vector* com o deslocamento 0x180. Para informações detalhadas sobre o funcionamento da *Execute-Inhibit*, ver 4.3.4.

9. Excepção Read-Inhibit
Esta excepção ocorre sempre que um endereço virtual pedido pela MMU se encontre mapeado na TLB mas possua o bit XI activo. A excepção *Read-Inhibit* só é válida para processadores que implementem os bits RI activos no registo PageGrainRIE.

Estados actualizados	Bits	Valor
Cause	ExcCode	TLBL ou TLBRI

Tabela 4.43: Estados modificados pela excepção TLB Read-Inhibit.

O vector de excepção utilizado é o *General Exception Vector* com o deslocamento 0x180. Para informações detalhadas sobre o funcionamento da *Read-Inhibit*, ver 4.3.4.

10. Excepção TLB Invalid
A excepção *TLB Invalid* ocorre sempre que o endereço virtual pedido pelo MMU se encontre mapeado na TLB e a página física (PFN) referenciada, respectivamente *EntryLo0* ou *EntryLo1*, possui o bit V *(Valid)* desligado.

Estados actualizados	Bits	Valor
Cause	ExcCode	TLBL ou TLBS

Tabela 4.44: Estados modificados pela excepção TLB Invalid.

O vector de excepção utilizado é o *General Exception Vector* com o deslocamento 0x180. Para informações detalhadas sobre o funcionamento da *TLB Invalid*, ver 4.3.4.

11. Excepção TLB Modified
Da mesma forma que acontece com a *TLB Invalid*, a excepção *TLB Modified* ocorre sempre que o endereço virtual pedido pela MMU se encontre mapeado na TLB mas a página física referenciada possua o bit D *(Dirty)* activo.

O vector de excepção utilizado é o *General Exception Vector* com o deslocamento 0x180. Para informações detalhadas sobre o funcionamento da *TLB Modified*, ver 4.3.4.

Estados actualizados	Bits	Valor
Cause	ExcCode	Mod

Tabela 4.45: Estados modificados pela excepção TLB Modified.

12. Excepção Cache Error
Este tipo de excepções ocorrem sempre que a execução de uma instrução detecte erros nos rótulos da *cache*, *cache miss* ou sejam detectados problemas na paridade dos bits. Nestes casos, o vector de excepções aponta para uma zona de memória não mapeada e sem utilização de *cache* para resolução dos erros.

Estados actualizados	Bits	Valor
Cause	ExcCode	CacheErr
Status	ERL	1
Status	NMI	0
Status	SR	0
CacheErr	-	Estado de erro
ErrorEPC	-	PC de retorno

Tabela 4.46: Estados modificados pela excepção Cache.

O vector de excepção utilizado é o *Cache Exception Vector* com o deslocamento 0x100. Se o StatusBEV se encontrar activo, o PC é modificado para 0xBFC0 0200 + 0x0100. Caso contrário, PC é calculado utilizando as regras descritas anteriormente como podemos ver no quadro 4.33. Se a revisão da arquitectura for a 2, o PC é calculado através da expressão:

```
PC = EBase 31 .. 30 || 1 || EBase 28 .. 12 || 0x100
```

Caso contrário, o PC é modificado para 0xA000 0000 + 0x0100 de forma a manter retro compatibilidade.

13. Excepção Bus Error
A excepção Bus Error ocorre quando uma instrução, informação ou introdução de instruções no processador gera um pedido (através dos mecanismos de *cache*) que termina em erro.

O vector de excepção utilizado é o *General Exception Vector* com o deslocamento 0x180.

Estados actualizados	**Bits**	**Valor**
Cause	ExcCode	IBE ou DBE

Tabela 4.47: Estados modificados pela excepção Bus Error.

14. Excepção Integer Overflow
Esta excepção ocorre sempre que ocorre um *overflow* em instruções aritméticas como o ADD, ADDI ou SUB.

Estados actualizados	**Bits**	**Valor**
Cause	ExcCode	Ov

Tabela 4.48: Estados modificados pela excepção Integer Overflow.

O vector de excepção utilizado é o *General Exception Vector* com o deslocamento 0x180.

15. Excepção Trap
Este tipo de excepções ocorre sempre que seja verdade a condição de validação das instruções que gerem TRAP, como exemplo: TEQ, TEQI, TGE,TGEI, TGEI, TGEIU, TGEU, TLT, TLTI, TLTIU, TLTU, TNE ou TNEI.

Estados actualizados	**Bits**	**Valor**
Cause	ExcCode	Tr

Tabela 4.49: Estados modificados pela excepção Trap.

O vector de excepção utilizado é o *General Exception Vector* com o deslocamento 0x180.

16. Excepção System Call
Este tipo de excepção ocorre apenas se a instrução SYSCALL for executada.

Estados actualizados	**Bits**	**Valor**
Cause	ExcCode	Sys

Tabela 4.50: Estados modificados pela excepção System Call.

O vector de excepção utilizado é o *General Exception Vector* com o deslocamento 0x180.

17. Excepção Breakpoint
Sempre que a instrução BREAK é executada é gerada a excepção Breakpoint.

O vector de excepção utilizado é o *General Exception Vector* com o deslocamento 0x180.

Estados actualizados	Bits	Valor
Cause	ExcCode	Bp

Tabela 4.51: Estados modificados pela excepção Break Point.

18. Excepção Reserved Instruction

Sempre que é executada uma instrução cuja codificação (*opcode*) se encontre marcada como reservada para implementações futuras, o processador é responsável pela geração por *hardware* da excepção *Reserved Instruction*. Para mais informações sobre as codificações reservadas para a arquitectura MIPS32 Revisão 2, ver [MT10b].

Estados actualizados	Bits	Valor
Cause	ExcCode	RI

Tabela 4.52: Estados modificados pela excepção Reserved Instruction.

O vector de excepção utilizado é o *General Exception Vector* com o deslocamento 0x180.

19. Excepção Coprocessor Unusable

Esta excepção ocorre sempre que uma das seguintes condições seja verdade:

1. Execução de instruções do subconjunto da codificação COP0 ou a instrução *Cache*, com o processador a correr em modo *User* ou *Super User* com o *StatusCU0* bit a zero.
2. Execução de instruções do subconjunto da codificação COP1, COP1X, LWC1, SWC1, LDC1, SDC1 ou MOVCI com o *StatusCU1* desligado.
3. Execução de instruções do subconjunto da codificação COP2, LWC2, SWC2, LDC2 ou SDC2 com o *StatusCU2* desligado.

Estados actualizados	Bits	Valor
Cause	ExcCode	CpU
Cause	CE	Número do coprocessador gerador da excepção

Tabela 4.53: Estados modificados pela excepção Coprocessor Unusable.

O vector de excepção utilizado é o *General Exception Vector* com o deslocamento 0x180.

20. Excepção Floating Point

Este tipo de excepção ocorre quando sucede uma excepção na execução do coprocessador um, sendo em seguida encaminhada para o coprocessador zero.

Estados actualizados	Bits	Valor
Cause	ExcCode	FPE
FCSR	-	Indica a causa da excepção da unidade de vírgula flutuante

Tabela 4.54: Estados modificados pela excepção Floating Point.

O vector de excepção utilizado é o *General Exception Vector* com o deslocamento 0x180.

21. Excepção Coprocessor 2
Este tipo de excepção ocorre quando sucede uma excepção na execução do coprocessador dois, sendo posteriormente sinalizada para o coprocessador zero.

Estados actualizados	Bits	Valor
Cause	ExcCode	C2E

Tabela 4.55: Estados modificados pela excepção do Coprocessor 2.

O vector de excepção utilizado é o *General Exception Vector* com o deslocamento 0x180.

22. Excepção Watch
A excepção *Watch* faz parte do mecanismo de Debug da arquitectura MIPS32, e ocorre sempre que o endereço guardado nos registos *WatchHi* e *WatchLo* correspondem ao endereço da instrução actual ou da referência a dados na memória.

Por defeito, a excepção *Watch* só pode ocorrer caso ambos os bits StatusEXL e StatusERL se encontrem desactivados, caso contrário, através de *hardware* o processador activará o bit CauseWP indicando que o a excepção se encontra pendente.

Estados actualizados	Bits	Valor
Cause	ExcCode	WATCH
Cause	WP	Indica se o WATCH foi atrasado ou não

Tabela 4.56: Estados modificados pela excepção Watch.

O vector de excepção utilizado é o *General Exception Vector* com o deslocamento 0x180.

23. Excepção Interrupt
A excepção *Interrupt* ocorre sempre que é feito um pedido de interrupção através do mecanismo de interrupções da arquitectura MIPS32.

Estados actualizados	Bits	Valor
Cause	ExcCode	Int
Cause	IP	Indica as interrupções que se encontram pendentes

Tabela 4.57: Estados modificados pela excepção Interrupt.

O vector de excepção utilizado é o *General Exception Vector* com o deslocamento 0x180 se o CauseIV bit se encontrar desligado. Caso contrário, o vector de excepção utilizado é o *Interrupt Vector* com o deslocamento 0x200.

Para informações detalhadas relativas aos mecanismos de excepções da arquitectura MIPS32, ver [MT10c].

Simulação em Software

O mecanismo de excepções encontra-se implementado dentro do motor de execução do processador nos ficheiros "Mips32_core.c" e "Mips32_core.h", sendo que todas as estruturas associadas utilizadas encontram-se descritas em 4.1. É através das funções `RaiseException3(CPU *cpu)` e `RaiseException(CPU *cpu, int ExcCode)` que o sistema executa todas as rotinas das excepções acima descritas.

A primeira função é utilizada para os três casos especiais, uma vez que não possuem código no *CauseExcCode*, nomeadamente: *Reset*, *Soft Reset* e *NMI*. Estes três casos não podem/devem ser gerados por *software* sob pena de funcionamento imprevisível, dado que o seu despertar é feito exclusivamente por *hardware* através de injecção de sinais directamente no processador. Desta forma, optou-se por criar uma função, que através dos estados dos bits *StatusNMI* e *StatusSR* consiga deduzir qual a operação que deverá ser executada, sem adicionar comparações desnecessárias no tratamento das restantes excepções. Na figura 4.10 pode-se observar o diagrama de actividades para as excepções *Reset*, *Soft Reset* e *NMI*.

Como visto anteriormente, o funcionamento pode variar de excepção para excepção, sendo que em determinados casos, os estados resguardados pela arquitectura em cada excepção podem variar. De forma a resolver este problema, optou-se pela criação da segunda função para o processamento geral das excepções identificadas através do *CauseExcCode*. De forma a entender melhor o funcionamento do mecanismo de excepções, veja-se o diagrama de execução para o tratamento geral de excepções na figura 4.11.

De forma a simplificar a implementação do mecanismo, e uma vez que o processamento das excepções possui um tronco comum, à excepção de *Reset*, *Soft Reset* e *NMI*, decidiu-se criar a função `RaiseException(CPU *cpu, int ExcCode)` para o tratamento global

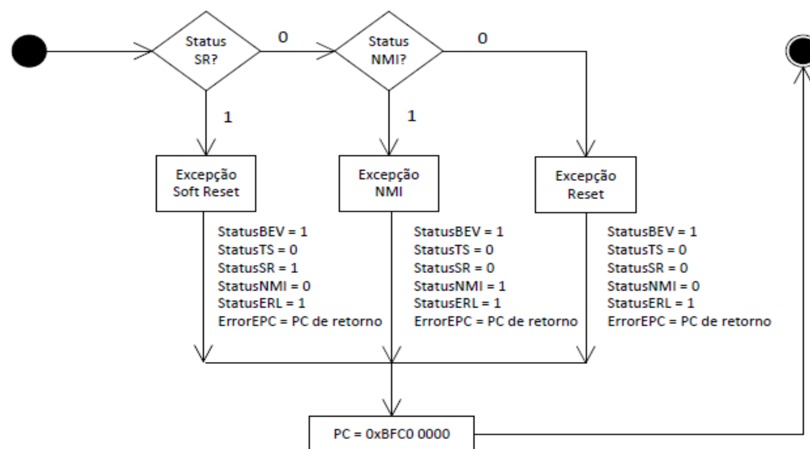


Figura 4.10: Diagrama de actividade para as excepções Reset, Soft Reset e NMI.

das excepções.

O processo de tratamento geral de excepções encontra-se dividido em quatro fases:

1. Validação da excepção;
2. Cálculo do deslocamento
3. Tratamento específico de cada excepção;
4. Cálculo do endereço base e geração do ponto de entrada.

A primeira fase consiste em identificar o valor do *StatusEXL*, uma vez que se o valor se encontrar a zero, o registo EPC será modificado e serão gerados novos ShadowSet's (apenas para a revisão 2). Em seguida, é necessário identificar se a instrução geradora da excepção se encontra em modo "Delay Slot" ou "Normal", uma vez que irá influenciar o valor do registo EPC. Caso contrário, não serão criados novos ShadowSet's nem será modificado o valor do EPC, forçando o tratamento pelo vector geral com deslocamento 0x180.

Na segunda fase é efectuado o cálculo do deslocamento do vector de entrada, sendo o seu valor directamente influenciada pelo tipo de excepção identificado pelo CauseExcCode e pelo valor do *StatusEXL*.

Como referido anteriormente, algumas excepções necessitam modificar o estado do processador através de bits específicos. Assim, a terceira fase consiste no processo de modificação do estado do processador através dos respectivos bits, ver 4.1.8

Por último temos a quarta fase, no qual o processador gerará através do registo EBase, para a revisão 2 da arquitectura, o endereço base utilizado para calcular o endereço do ponto de entrada. Assim que o endereço é calculado, este é introduzido no registo PC do processador mudando assim o fluxo de execução para o novo endereço.

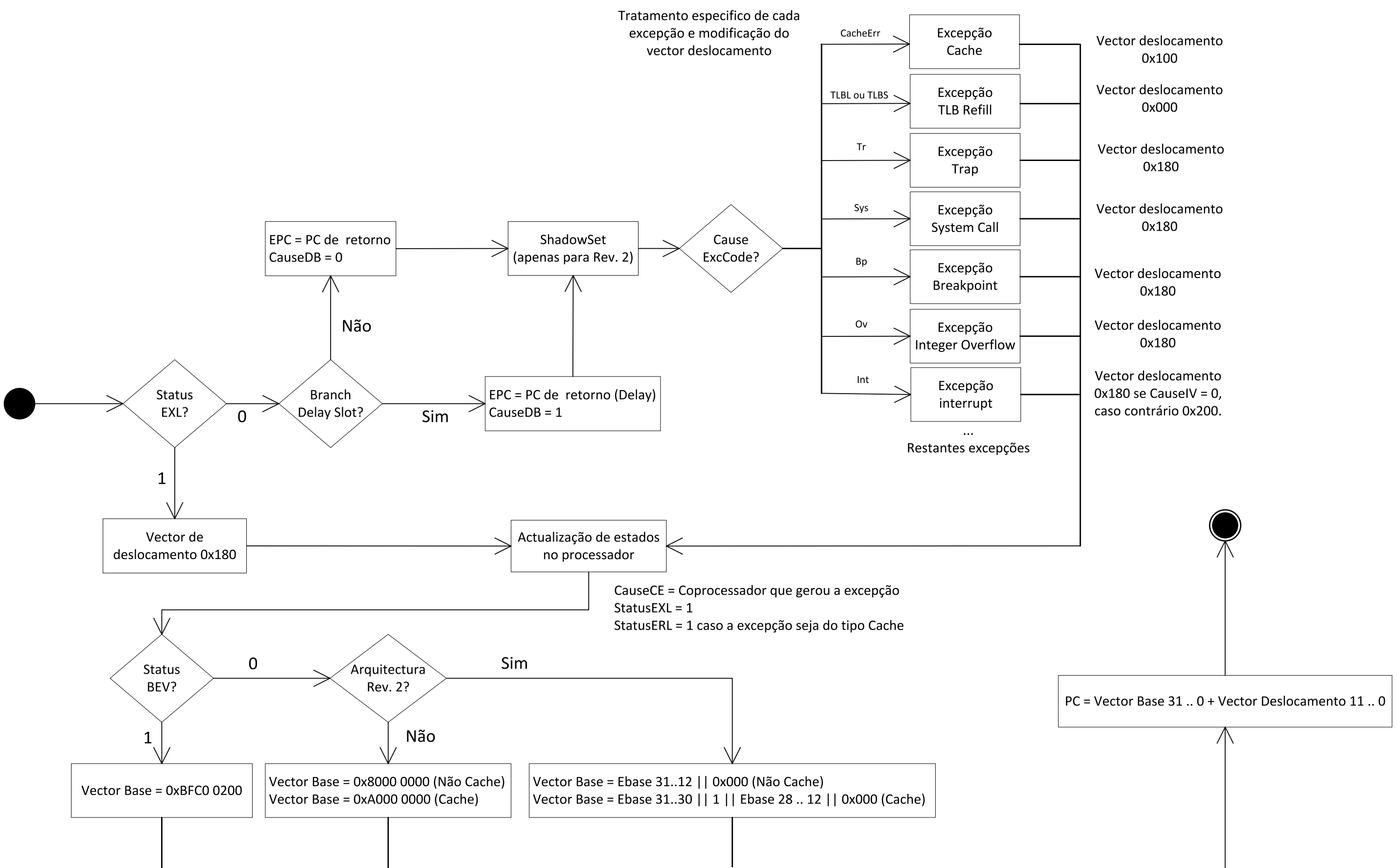


Figura 4.11: Diagrama de actividade para as excepções diferentes de Reset, Soft Reset e NMI.

4.2 Unidade de Gestão de Memória (MMU)

O módulo de gestão de memória, conhecido por MMU (*Memory Management Unit*), é um circuito integrado no processador responsável por efectuar a gestão da memória virtual. Este processo consiste: na validação dos modos de operação, atributos de cache, permissões de acesso a segmentos de memória virtual e tradução de endereços virtuais em endereços físicos, através do mecanismo de tradução implementado.

Por defeito, a arquitectura MIPS32 suporta exclusivamente a utilização de paginação, sendo o processo de tradução de endereços influenciado pelo mecanismo de tradução e respectivas funcionalidades implementadas no processador. No presente trabalho apenas será abordado o modelo de tradução de endereços utilizando o mecanismo TLB sem suporte a páginas físicas de tamanho 1KB.

É igualmente sua responsabilidade, a coordenação na comunicação entre o processador e a memória RAM, uma vez que todos os pedidos de tradução de endereços são enviados pelos coprocessadores zero e um para o módulo MMU internamente. Em seguida será apresentada a forma como se encontra estruturada a memória virtual, assim como os mecanismos de comunicação entre a MMU e a memória RAM.

4.2.1 Memória Virtual

A arquitectura MIPS32 dispõe de um espaço de endereçamento virtual com o tamanho 2^{32} bits perfazendo um total de 4GB. Este espaço de endereçamento virtual encontra-se dividido em cinco segmentos que podem ser classificados como: *Mapped*, *Unmapped*, *Cached* e *Uncached*.

Mapped Todos os endereços neste domínio são traduzidos pelo mecanismo de tradução TLB.

Unmapped Os endereços neste domínio não são traduzidos pela TLB, mas sim mapeados directamente para os primeiros 512 MBytes da memória RAM começando no endereço físico 0x00000000.

Cached Apenas os segmentos USEG, KSEG0, KSSEG e KSEG3 podem ou não utilizar os mecanismos de cache.

Uncached Endereços virtuais que não utilizam o mecanismo de cache. Alerta-se para o facto de a cache ser um componente desejável a qualquer processador, sendo não obrigatório.

Figura 4.12: Estrutura da memória virtual para arquitecturas de 32 bits.

Endereço Virtual ($VA_{31..29}$)	**Segmento**	**Espaço de endereçamento**	**Modo de execução**	**Acesso permitido**	**Tamanho**
0b111	kseg3	0xFFFF FFFF até 0xE000 0000	kernel	kernel	2^{29} bytes
0b110	sseg ksseg	0xDFFF FFFF até 0xC000 0000	supervisor	supervisor kernel	2^{29} bytes
0b101	kseg1	0xBFFF FFFF até 0xA000 0000	kernel	kernel	2^{29} bytes
0b100	kseg0	0x9FFF FFFF até 0x8000 0000	kernel	kernel	2^{29} bytes
0b0xx	useg suseg0 kuseg	0x7FFF FFFF até 0x0000 0000	user	user superuser kernel	2^{31} bytes

Tabela 4.58: Zonas de endereçamento na memória virtual.

Associado a cada segmento encontra-se o modo de operação do processador, uma vez que existem zonas da memória virtual que só podem ser acedidas em modo privilegiado. Esta técnica é utilizada para restringir o acesso não autorizado, por parte de aplicações que corram em modo *user* a segmentos de memória associados ao modo *kernel* ou *super user*.

Existem outras técnicas para evitar o acesso ilegal entre processos a correr no segmento USEG, tais como paginação linear ou multinível, abordadas mais à frente neste capítulo. No quadro 4.58 é possivel visualizar os modos associados a cada segmento, assim como o seu tamanho e respectivos endereços virtuais de início e fim.

Os segmentos KSEG0 e KSEG1 apontam para a mesma zona de memória física, nomeadamente os primeiros 512 MBytes da memória RAM. Esta zona de memória física varia entre 0x00000000 e 0x1FFFFFFF, sobrepondo-se um ao outro, no qual o KSEG0 utiliza o mecanismo de *cache* e o KSEG1 não utiliza o mecanismo de *cache*.

Actualmente as placas mãe (*motherboard*) tiram partido da técnica MMIO (*Memory Mapped Input/ Output*) para mapear endereços físicos para componentes ligados na mesma. Desta forma, é possível a comunicação entre os vários periféricos através de uma arquitectura de LOAD e STORE, como explanado em detalhe no capítulo 4.5.

		Acção despoletada a partir do modo de execução		
Espaço de endereçamento	**Segmento**	**Modo User**	**Modo Superuser**	**Modo Kernel**
0xFFFF FFFF até 0xE000 0000	kseg3	Address Error	Address Error	Mapped
0xDFFF FFFF até 0xC000 0000	sseg ksseg	Address Error	Mapped	Mapped
0xBFFF FFFF até 0xA000 0000	kseg1	Address Error	Address Error	Unmapped, Uncached
0x9FFF FFFF até 0x8000 0000	kseg0	Address Error	Address Error	Unmapped
0x7FFF FFFF até 0x0000 0000	useg suseg kuseg	Mapped	Mapped	StatusERL = 1, Unmapped StatusERL = 0, Mapped

Tabela 4.59: Zonas de endereçamento em função do modo de operação do processador.

No quadro 4.59 é possível visualizar o comportamento e respectivas acções tomadas pela MMU no processo de validação dos endereços virtuais e segmentos relativamente aos modos de operação. Qualquer tentativa de acesso por parte de um modo de execução com menos privilégios a zonas de memória priviligiadas, levará a MMU a gerar uma excepção do tipo *Address Error*.

No caso particular, em que o processador se encontre a correr em modo *Kernel* e o *StatusERL* se encontrar activo, o mapeamento para o segmento USEG é modificado de mapeado pela TLB para não mapeado. Esta mudança deve-se ao facto do processador se encontrar num estado de erro de *cache* ou arranque do sistema (*Reset*, *Soft Reset* ou NMI), no qual as interrupções se encontram desligadas. Desta forma, o processador pode aceder directamente ao conteúdo da memória física RAM.

4.2.2 Paginação

Com o surgimento da técnica de paginação, a memória física passou a ser dividia em blocos iguais, podendo o seu tamanho variar, mas sempre numa potência de 2. O tamanho das páginas suportado é definido exclusivamente pela implementação do processador, podendo a sua utilização ser configurável pelo sistema operativo. Cada um destes blocos

é chamado de página e é composto por um número de página e um deslocamento. O seu objectivo é oferecer aos programadores um espaço de endereçamento virtual contíguo, de forma a transferir a preocupação da gestão da memória para o sistema operativo. Assim, o sistema operativo poderá gerir os seus recursos internos, páginas físicas, de forma transparente ao programa que se encontra em execução.

Veja-se o seguinte exemplo: um processador no qual cada página possui 4KBytes ($2^{12} = 4096$ bytes) e o espaço de endereçamento com 32bits. Os 12 bits menos significativos indicam o deslocamento e os 20 bits mais significativos indicam uma dada página virtual dos 2^{20} bits (mais de um milhão de páginas possíveis).

Na secção seguinte, veremos a forma como os endereços virtuais traduzidos pela TLB variam com o tamanho das páginas virtuais. Será igualmente abordado a forma como o sistema operativo influencia o comportamento da tradução dos endereços, uma vez que a tradução dos endereços na TLB é mecânica.

Numa abordagem superficial, cada processo é composto por uma tabela de páginas que reside em memória e cuja manutenção é exclusiva do sistema operativo. Dos vários tipos de organizações existentes para tabelas de páginas dos processos, serão utilizados a paginação linear e multinível para explanar a influência do sistema operativo na tradução dos endereços virtuais em físicos.

Em ambos os casos, o sistema operativo possui uma ou várias tabelas de páginas por processo, que é composta por vários descritores de cada página física, sendo o descritor apelidado de PTE (*Page Table Entry*). Cada descritor contém o endereço físico da página (base) e informações de protecção e estado da página em causa, nomeadamente os bits P (*Present*), V (*Valid*) e D (*Dirty*), sendo o deslocamento calculado através dos bits adequados no endereço virtual.

Como se pode ver na figura 4.13, quando o processador emite um pedido para a MMU, o seu primeiro passo é aceder à TLB à procura do mapeamento para a página física. Caso o mapeamento não exista, é através do tratador da excepção *TLB Refill* que o sistema operativo, de acordo com a técnica de paginação utilizado, acederá à tabela de páginas do processo e introduzirá uma nova entrada na TLB. Em seguida, o processador deverá executar novamente a instrução geradora da excepção de forma a prosseguir com a execução do código binário.

Inerente à utilização da técnica de paginação surge um acontecimento conhecido como *Page Fault*, que ocorre quando uma página física não se encontra presente na memória RAM. A gestão das páginas físicas é efectuada exclusivamente pelo sistema operativo sendo este processo transparente à própria arquitectura. Uma vez que a memória física tem limite, assim como o espaço de endereçamento virtual, existe a necessidade de

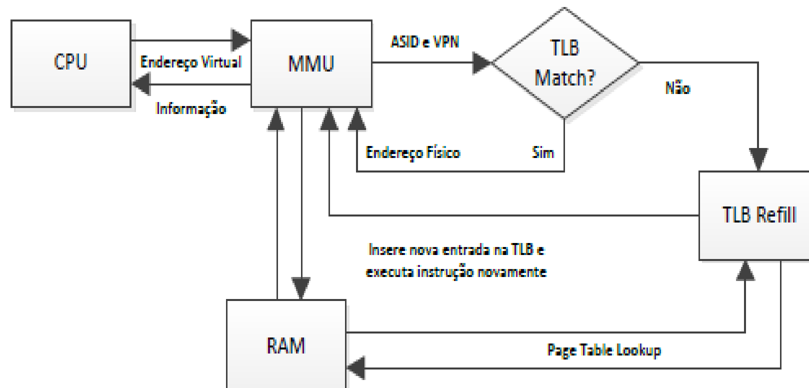


Figura 4.13: Mecanismo de tradução em sistemas com paginação.

remover as páginas físicas da memória RAM para uma zona em disco, conhecida como área de SWAP, a fim de libertar espaço.

Desta forma, o sistema operativo consegue manter em simultâneo, teoricamente com espaço interminável, as páginas físicas utilizadas pelos processos. O evento *Page Fault* pode surgir quando há uma tentativa de acesso a uma entrada na TLB, no qual a PFN correspondente contenha o bit V igual 0. Nestes casos, o sistema operativo (através da excepção *TLB Invalid*) alocará espaço na memória e copiará a página da área de SWAP para o novo espaço alocado, actualizando também o endereço na tabela de páginas do processo em execução. Nos casos em que não existe mapeamentos na TLB, o surgimento de um *Page Fault* é tratado na excepção *TLB Refill*.

Por defeito, a arquitectura MIPS32 disponibiliza o registo *Context* do coprocessador zero, no qual o sistema operativo registará o PTB da tabela de páginas do processo em execução. Este registo é actualizado sempre que haja uma mudança de contexto na execução no processador.

Em sistemas que utilizem paginação linear, o registo *Context* aponta para o endereço base da única tabela de páginas do processo. Nos casos em que a paginação seja multinível, o registo aponta para o endereço base da tabela de páginas global. Segue-se uma breve descrição dos métodos de paginação linear e multinível.

Linear

Os sistemas operativos que usufruem da técnica de paginação linear, utilizam uma única tabela de páginas para mapear as páginas físicas de cada processo. Pegando no exemplo anterior, com páginas de tamanho 4KBytes e sistema de endereçamento virtual de 32bits, um processo carregado em memória ocupará 2^{20} * 4 Bytes = 4MBytes.

O seu deslocamento é calculado utilizando os 12 bits menos significativos do endereço virtual. Por defeito, o sistema operativo inicia todos os PTE's com a *flag* V e P igual a 0, de forma a evitar ocupação de espaço e processamento desnecessário, uma vez que das 2^{20} páginas físicas possíveis de endereçar, apenas um pequeno conjunto será utilizado para programas relativamente pequenos. Segue-se a disposição de um processo com paginação linear.

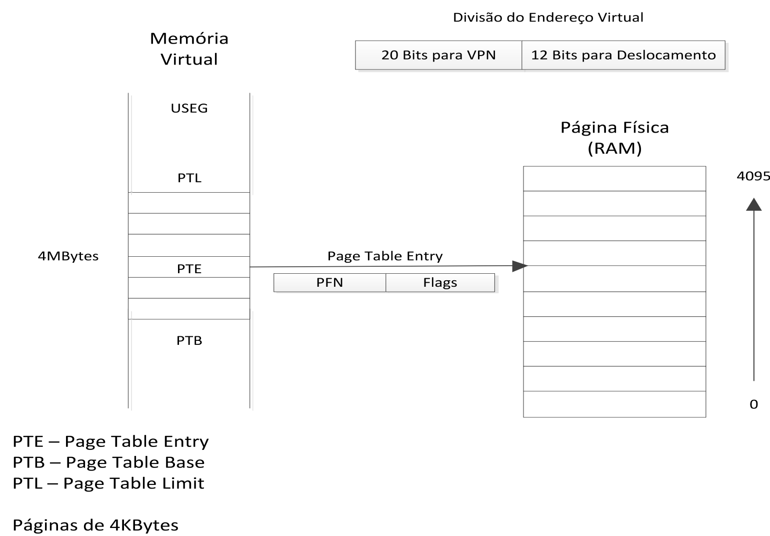


Figura 4.14: Mecanismo de paginação linear.

Multinível

Em sistemas que utilizem a técnica de paginação multinível, como exemplo o Linux, utilizam-se várias tabelas de páginas para mapear as páginas físicas de cada processo, existindo assim por processo uma tabela de páginas global, no qual cada entrada aponta para outra tabela de páginas. A quantidade de sub tabelas de páginas utilizada varia com o nível de profundidade, sendo o último nível a tabela páginas final que aponta para a página física.

Pegando novamente no exemplo anterior, assumindo uma paginação multinível de profundidade 2, onde os 10 bits mais significativos definem a tabela de páginas global, os 10 bits intermédios a tabela de páginas de 2º nível e os 12 bits menos significativos o deslocamento dentro da página.

Neste modelo pode-se ver que com páginas de 4KBytes, apenas serão alocados em memória 2^{10} (1024 bytes) * 4 bytes = 4096 bytes para a tabela de páginas global. A este tamanho é necessário adicionar o espaço ocupado pela tabela de páginas de 2º nível que perfaz 2^{10} (1024 bytes) * 4 bytes = 4096 bytes. Assim, o espaço minímo ocupado por uma página num sistema com paginação multinível e profundidade 2 é (1024 + 1024) * 4 = 8 Kbytes.

Como se pode ver existe um ganho significativo tanto no espaço ocupado pelo o processo em memória, passando de 4 MB para 8 Kbytes, como no tempo para a resolução do endereço físico uma vez que no pior dos casos serão percorridas 1024 + 1024 entradas = 2048 entradas de 4 Mbytes. Segue-se a disposição de um processo com paginação multinível com profundidade 2.

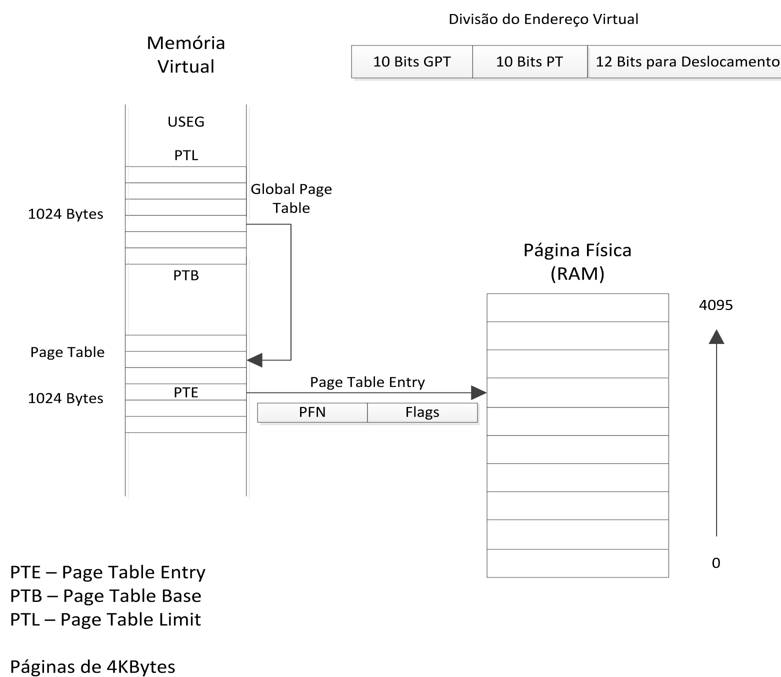


Figura 4.15: Mecanismo de paginação multinível.

4.2.3 Simulação em Software

A MMU é um módulo de presença obrigatória no processador e a sua implementação encontra-se dentro do ficheiro "Mips32_core.c" e "Mips32_core.h". A sua definição consiste numa estrutura com um apontador para o mecanismo de tradução definido no ficheiro "Config.h", assim como num conjunto de funções de validação, tradução de endereços e acesso à memória RAM.
Segue-se a definição da estrutura da MMU:

```
typedef struct MMU{
        //Mecanismo de Traducao de Enderecos (TLB)
        MT *mt;
} MMU;
```

Listing 4.13: Representação da MMU em C.

Segue-se a definição das funções básicas para o funcionamento da MMU dentro do processador:

```
/* Arquitectura de LOAD e STORE
 *
 * (MemElem) <- LoadMemory(CCA, AccessLength, pAddr, vAddr,
     IorD)
 * (void) <- StoreMemory(CCA, AccessLength, MemElem, pAddr,
     vAddr)
 *
 * CCA - Cacheability and Coherency Attribute
 * pAddr - Physical Address
 * vAddr - virtual Address
 * IorD - Instruction or Data
 * LorS - Load or Store
 * AccessLength - Tamanho da informacao
 *
 */
WORD LoadMemory  (CPU *cpu, BYTE AcessLength, WORD vAddr);
void StoreMemory (CPU *cpu, BYTE AcessLength, WORD vAddr,
    WORD MemElem);

/*
 * Mecanismo de Traducao de Enderecos
 * Endereco Virtual para Endereco Fisico
 *
 * (pAddr, CCA) <- AddressTranslation (vAddr, IorD, LorS)
 *
 */
```

```
WORD AddressTranslation (MT *mt, WORD SegValidCode, WORD
    vAddr);

/* Valida Segmento e Modo de Execucao
 *
 * Return Codes:
 *
 * 0 - ADDRESS_ERROR
 * 1 - USER_MAPPED
 * 2 - SUPER_USER_MAPPED
 * 3 - KERNEL_MAPPED
 * 4 - KERNEL_UNMAPPED
 * 5 - KERNEL_UNMAPPED_UNCACHED
 *
 */
WORD SegmentValidation(WORD vaddr);
```

Listing 4.14: Representação das funções de validação e acesso à memória da MMU.

A função *SegmentValidation* é utilizada pela MMU para validar os modos de operação e endereços virtuais pedidos pelo processador.

No processo de validação, a MMU confirmará o modo de execução do processador através dos bits do registo do coprocessador zero *StatusKSU*, *StatusERL* e *StatusEXL*. Em seguida, de acordo com o mapa dos segmentos referidos no quadro 4.59, a função retornará um código identificativo do segmento em questão ou um código de erro de endereçamento (*Address Error*). Nos casos em que o processador se encontre a correr em modo *kernel* e o *StatusERL* se encontre activo, para traduções no USEG, a função não retornará KERNEL_MAPPED mas sim o código correspondente a KERNEL_UNMAPPED.

A função *AddressTranslation* é utilizada, caso a validação do segmento da memória virtual se confirme, para chamar a função de tradução respectiva do mecanismo de tradução de endereços mapeada no ficheiro "Config.h" como se pode ver no excerto de código seguinte.

```
#ifndef TRANSLATIONTYPE
#define TRANSLATIONTYPE

    #if defined( TRANSLATION_TLB ) && defined( TRANSLATION_FMT )
        #error "Dois mecanismos de traducao seleccionados!"
    #endif

    #ifdef TRANSLATION_TLB
        //Mecanismo de Traducao MT
        #define MT TLB
```

```
        //Nome do Mecanismo de Traducao
        #define MTTYPE "TLB - Translation Lookaside Buffer"

        //Mappeamento de Funcoes da TLB com paginacao Linear
        #define traduz  traduz_tlb
    #endif

    #ifdef TRANSLATION_FMT
        //Mecanismo de Traducao MT
        #define MT FMT

        //Nome do Mecanismo de Traducao
        #define MTTYPE "FMT - Fixed Mapped translation"

        //Mappeamento de Funcoes da FMT
        #define traduz  traduz_fmt
    #endif

#endif
```

Listing 4.15: Mapeamento de funções da MMU.

As funções *LoadMemory* e *StoreMemory* são utilizadas pelo módulo MMU para aceder à memória RAM, funcionando como o ponto de entrada e saída de informação no processador.

Desta forma o processador deve respeitar o formato de mensagem definida pelo simulador através do ficheiro “ConnDef.h”, representado pelo bloco de código abaixo:

```
typedef struct mem_request{
    //Load or Store
    WORD LorS;

    //Access Length
    WORD Access;

    //Physical Address
    WORD Physical;

    //Virtual Address
    WORD Virtual;

    //Data for Store
    WORD Data;
```

```
        //RAM
        //Control Flags exclusivo para utilizacao do
            Simulador
        WORD Control;

        //Return Code
        WORD RC;
} mem_request;
```

Listing 4.16: Definição das mensagens entre o CPU e RAM.

Sempre que a MMU necessita enviar informação para a RAM, é necessário criar uma estrutura do tipo `struct mem_request` e preencher com a respectiva informação, nomeadamente: tipo de acesso (LOAD ou STORE), tamanho da informação (BYTE, HALFWORD ou WORD)[8], endereço físico previamente calculado e o endereço virtual. Caso a operação seja do tipo STORE o campo “Data” é preenchido com o valor que se deseja enviar para a RAM, caso contrário este campo é ignorado.

Os campos “Control” e “RC” são exclusivos para utilização do simulador como veremos em 4.5. Uma vez criada a mensagem, o processador necessita encaminhar a informação para os canais correctos, respectivamente: Tx (Transmissor) e Rx (Receptor). Como vimos na estrutura do processador em 2, estas variáveis guardam o valor dos descritores dos canais estabelecidos previamente pelo simulador, passados como argumento na inicialização do processador. Desta forma, a cada pedido de LOAD ou STORE o processador envia a estrutura para o canal Tx e aguarda resposta da RAM, lendo a informação, retornada no mesmo formato, pelo canal Rx.

A razão pelo qual os descritores são passados do simulador para o processador, deve-se ao facto de existirem várias formas de comunicação por descritores em Linux, mormente: *Named Pipes*, *Unnamed Pipes*, *Pseudo Terminals* ou *Network Sockets*. Assim, de forma a simplificar a implementação do simulador, optou-se por atribuir as responsabilidades de escolha e manutenção dos formatos de comunicação ao simulador. A definição dos canais, assim como os protocolos de comunicação suportados, serão abordados em detalhe na secção 4.5.

Todas as instruções definidas na ISA para acesso à memória RAM, como *LW*, *LH*, *LB*, *SW*, *SH*, *SB*, entre outras, utilizam estas duas funções para enviar informação para os canais de comunicação. Uma vez que o simulador não suporta a utilização de *cache* os atributos desta funcionalidade não foram incluídos nas funções *LoadMemory* e *StoreMemory*.

[8]BYTE = 1 byte, HALFWORD = 2 bytes, WORD = 4 bytes, DWORD = 8 bytes.

4.3 Mecanismo de Tradução de Endereços (TLB)

Como visto anteriormente, a memória virtual encontra-se segmentada em cinco áreas com diferentes tamanhos e permissões de acesso. Algumas destas áreas são designadas de mapeadas, e necessitam que um mecanismo fisíco traduza os endereços virtuais para endereços físicos.

Estes mecanismos são extremamente importantes para a performance do sistema, uma vez que funcionam como uma *cache* no processador, que guarda temporariamente um conjunto de pares de páginas virtuais e físicas. Desta forma, há um ganho significativo no tempo de tradução dos endereços uma vez que o processador conhece os endereços físicos onde quer chegar e não necessita de aceder à tabela de páginas em memória, mantida pelo sistema operativo, para encontrar o endereço físico que deseja aceder. Por norma a sua implementação é feita a nível de *hardware*, sendo que as mais conhecidas para a arquitectura MIPS são a TLB *(Translation Lookaside Buffer)* e o FMT*(Fixed Mapping Translation)*.

Em sistemas operativos complexos é obrigatório a utilização de uma TLB não só pelo aumento do desempenho, mas também para facilitar e melhorar a gestão da memória. Por outro lado, em sistemas que não exijam tamanha complexidade, nomeadamente microcontroladores em sistemas embebidos, é comum a utilização da técnica FMT como mecanismo de tradução de endereços uma vez que o conjunto de recursos é muito reduzido assim como a complexidade da sua gestão.

Neste capítulo apenas serão abordados a estrutura e o modo de funcionamento da TLB não abordando assim a organização FMT.
O objectivo da TLB é manter, numa memória interna do processador, a tradução dos endereços virtuais em endereços físicos, de forma a acelerar o processo de tradução de endereços. Desta forma, sempre que o processador tentar traduzir um endereço, primeiro consultará a sua tabela de endereços interna e, caso não exista uma correspondência, gerará uma excepção do tipo *TLB Refill* no qual o *kernel* do sistema operativo deverá descobrir a que endereço físico corresponde o endereço virtual pedido, sendo o novo mapeamento introduzido na TLB para futura utilização.

Em seguida será apresentada a forma como é feita a comunicação entre a MMU e a TLB através dos registos do coprocessador zero e as instruções existentes na ISA.

4.3.1 Interface de comunicação MMU/TLB

O mecanismo de tradução de endereços TLB reside dentro do chip do processador, sendo a interface de comunicação entre os dois componentes feita através de um conjunto de registos do coprocessador zero e um subconjunto de instruções definidas na ISA da arquitectura.

Os registos de comunicação[9] base utilizados pela TLB são: *EntryHi*, *EntryLo0*, *EntryLo1*, *Wired*, *Index*, *Random*, *Context*, *ContextConfig* e *Pagemask*.

CP0 Registo 0, Select 0 (Index)

O registo *Index* guarda o índice da TLB que será utilizado pelas instruções TLBWI, TLBR e TLBP. O acesso e a modificação do valor do índice é permitida por *software* dependendo do valor dos bits *StatusCU*. Para a instrução TLBP, sempre que ocorra uma correspondência o bit 31 (P) é escrito por *hardware* com o valor 0, caso não exista correspondência o *hardware* escreverá o valor 1.

31	30 ... n	n-1 ... 0
P	0	Index

Tabela 4.60: Registo interno Index.

CP0 Registo 1, Select 0 (Random)

O registo *Random* gera um valor aleatório e guarda-o nos primeiros n bits, sendo n o tamanho da TLB. Na realidade a geração do valor aleatório é basicamente um contador que está sempre a correr e é utilizado pela instrução TLBWR. Para os primeiros n bits o *software* detem permissões para escrita e leitura.

31 ... n	n-1 ... 0
0	Random

Tabela 4.61: Registo interno Random.

[9]Existem outros registos do coprocessador zero que são utilizados em configurações mais complexas da TLB, como exemplo duas TLBs a funcionar em conjunto. Estas técnicas não são abordadas neste documento.

CP0 Registo 2 e 3, Select 0 (EntryLo0 e EntryLo1)

Este par de registos contém a informação referente à localização das páginas físicas na memória RAM. O registo *EntryLo0* guarda as páginas pares (*even*) e o registo *EntryLo1* guarda as páginas ímpares (*odd*).

31 30	29 6	5 3	2	1	0
Fill	PFN	C	D	V	G

Tabela 4.62: Registo interno EntryLo0 e EntryLo1.

Campo	Descrição
Fill	Retorna sempre zeros em leitura.
Page Frame Number (PFN)	Corresponde ao número de página na memória RAM.
Cacheability and Coherency Attribute (C)	Estes dois bits identificam se a página encontra-se num segmento que utiliza a cache no processador.
Dirty (D)	O bit "Dirty" serve para identificar se é possível escrever na página. Se o bit estiver a 1 é possível escrever na na página, caso o bit estiver a 0, processador gerará uma excepção "TLB Modified Exception".
Valid (V)	Este bit indica se a TLB Entry e a sua respectiva página na memória são validas. Se o bit for 1 então a página é válida, caso contrário o sistema irá gerar uma excepção "TLB Invalid Exception".
Global (G)	Se este bit estiver activo todas as comparações com os bits ASID são ignoradas.

Tabela 4.63: Especificação dos bits do registo *EntryLo0 e 1*.

CP0 Registo 4, Select 0 (Context)

Este registo é muito importante em sistemas que utilizem vários processos, uma vez que é através deste mecanismo que se irá identificar o "contexto" ou a tabela de páginas do processo em execução.

O seu funcionamento é condicionado pelo o estado do bit *ContextCTXTC* que identifica se o registo do coprocessador central (CP0 Registo 4, Select 1) *ContextConfig* se encontra implementado no processador, sendo este opcional para a arquitectura MIPS32. Assim, se o bit *ContextCTXTC* se encontrar desligado, valor por defeito, o funcionamento do registo terá a seguinte representação:

31 ... 23	22 ... 4	3 ... 0
PTE Base	BadVPN2	0

Tabela 4.64: Registo interno Context.

Neste modelo, sempre que ocorre uma excepção do tipo *TLB Refill, TLB Invalid* ou *TLB Modified* o processador introduzirá os bits VA $_{31..13}$ no campo BadVPN2 do registo *Context*. O valor do campo PTE (*Page Table Entry*) será preenchido pelo o sistema operativo com um apontador para um array de PTE's do processo actual[10].
Uma vez que os quatro bits menos significativos do registo *Context* são zero, o registo aponta para uma estrutura em memória (PTE) com tamanho 16 Bytes.

O tamanho e o formato das estruturas PTE são definidos pelos sistemas operativos. Na secção 4.2.2 viu-se que em sistemas com paginação linear, uma tabela de páginas ocupa 4 MBytes e em sistemas com paginação multinível ocuparia 4 KBytes, isto para estruturas PTE com tamanho 8 Bytes. Em sistemas no qual a estrutura PTE possua um tamanho 16 bytes, a tabela de páginas ocupará quatro vezes o tamanho anterior, 16 MBytes e 16 KBytes, respectivamente.

Pelo facto do registo *Context* apontar directamente para uma estrutura de 16 Bytes não implica que os sistemas operativos sejam obrigados a utilizar este formato. Um procedimento comum é a aplicação de máscaras e *shifts* ao registo *Context* de forma a criar apontadores para o tamanho do PTE desejado.

Se o bit *ContextCTXTC* se encontrar activo, o registo *Context* pode apontar para qualquer estrutura PTE na memória cujo o endereço se encontre alinhado a uma potência de base dois. Dependendo da configuração do registo *ContextConfig*, é possível apontar para uma estrutura PTE mais pequena, 8 bytes, numa tabela de páginas linear ou tabela de páginas global em ambientes com paginação multinível.

CP0 Registo 5, Select 0 (Pagemask)

Este registo é utilizado pela TLB para identificar o tamanho das páginas, físicas e virtuais. O seu conteúdo pode ser modificado tanto por *hardware* como por *software*. A arquitectura MIPS suporta a utilização de páginas com vários tamanhos sendo que o mais comum seja páginas com tamanhos de 1KB (1024 bytes) e 4KB (4096 bytes).

[10]Em sistemas Linux o responsável por esta operação é o processo *scheduler*.

31 29	28 13	12 11	10 0
0	Mask	MaskX	0

Tabela 4.65: Registo interno Pagemask.

A codificação dos campos encontram-se no quadro abaixo.

Tamanho das páginas	**Bits 28..13 da máscara**	**Bits 12..11 da máscara**
1 KByte	0x0	0x0
4 KByte	0x0	0x3
16 KByte	0x3	0x3
64 KByte	0xF	0x3
256 KByte	0x3F	0x3
1 MByte	0xFF	0x3
4 MByte	0x3FF	0x3
16 MByte	0xFFF	0x3
64 MByte	0x3FFF	0x3
256 MByte	0xFFFF	0x3

Tabela 4.66: Codificação do tamanho das páginas.

CP0 Registo 6, Select 0 (Wired)

Este registo tem como função guardar o número de entradas na TLB que são consideradas fixas, não podendo ser alteradas. O registo pode ser lido e escrito o que permite que o sistema operativo modifique o número de entradas fixas conforme a sua necessidade. Esta é uma técnica utilizada pelo sistema operativo para poder manter sempre em memória as páginas mais requisitadas ou mapeamentos permanentes.

31 n	n-1 0
0	Wired

Tabela 4.67: Registo interno Wired.

CP0 Registo 10, Select 0 (EntryHi)

Este registo contém a informação relativa ao número da página virtual (VPN2) e ao seu respectivo ASID. As instruções TLBR, TLBP, TLBWR e TLBWI lêem sempre este registo para pesquisar ou inserir entradas novas na TLB. O campo VPN2X só é utilizado caso o sistema dê suporte a páginas de 1K, caso contrario este campo retorna sempre 0.

A implementação do campo ASID (*Address Space Identifier)* não é obrigatório mas obviamente desejável. Em sistemas que utilizem vários processos e cuja mudança de contexto seja uma grande necessidade, o que para um sistema operativo minimamente complexo é muito alta, a sua utilização é importante para aumentar o nível de desempenho.

31 – 13	12 – 11	10 – 8	7 – 0
VPN2	VPN2X	0	ASID

Tabela 4.68: Registo interno EntryHi.

Para informações mais detalhadas dos registos do coprocessador zero ver [MT10c].

4.3.2 Instruções de controlo MMU/TLB

O subconjunto de instruções existentes na ISA MIPS32r2 para comunicação entre a MMU e a TLB são: TLBWI (*TLB Write Index*), TLBWR (*TLB Write Random*), TLBR (*TLB Read*) e TLBP (*TLB Probe*). Por si só estas instruções não são o suficiente para escrever vários valores de 32 bits numa estrutura como a TLB. Desta forma, a arquitectura MIPS32 definiu que estas instruções teriam argumentos fixos sendo o seu valor variável, de forma que através dos registos do coprocessador zero acima citados, seja feita a interface entre o programador e a estrutura TLB.

As instruções TLBWR e TLBWI têm como objectivo introduzir uma nova entrada na estrutura que representa a TLB, sendo que a primeira gera um índice aleatoriamente e a segunda utiliza como índice o valor definido no registo *Index*.

A instrução TLBP é utilizada para verificar se existe uma correspondência da página virtual (VPN) e do espaço de endereçamento ASID no registo *EntryHi* com os valores existentes na TLB. Nos casos em que exista uma correspondência com a página virtual mas o bit (G) *Global* das páginas físicas se encontre activo numa chave da TLB, significa que os bits identificadores do ASID serão ignorados na comparação e a instrução TLBP retornará verdade para correspondência. Por último temos a instrução TLBR que em oposição às instruções de escrita, lê para os registos *EntryLo0* e *EntryLo1* os valores correspondentes à página virtual existente no registo *EntryHi* do coprocessador zero.

Das quatro instruções descritas, a instrução TLBR é a que tem menos utilização em sistemas que utilizem "TLB Flush" em vez de ASID's, uma vez que o sistema operativo introduz entradas na TLB até haver uma mudança de contexto. Em sistemas que utilizem ASID, a sua utilização já é mais importante pois o processo escalonamento necessita de obter informação relativa ao estado da TLB para gerir a forma como serão disponibilizados e atribuídos os ASID's aos novos processos.

4.3.3 TLB Flush vs ASID's

A forma como o sistema operativo controla a TLB é um aspecto fundamental para a segurança e o bom funcionamento de todo o sistema. Sempre que a execução de um processo é transferida para outro processo, acontece uma mudança de contexto que deve ser actualizada na TLB para que o novo processo possa aceder correctamente à sua informação e não à informação do processo anterior.

Já se sabe que a TLB guarda um conjunto de entradas, sendo cada entrada constituída por uma VPN, ASID, *Pagemask*, um par de FPN e seu conjunto de bits relativo ao seu estado. Esta informação que se encontra na TLB deve ser limpa ou filtrada quando há uma mudança de contexto de modo a que um processo não aceda intencionalmente ou não intencionalmente a zonas de memória de outros processos, o que tornaria todo o sistema instável. Imagine-se um sistema com dois processos a correr e havendo várias vezes mudanças de contexto.

VPN	**PFN**	**Válida**	**Permissão**
0	20	1	r-x
-	-	0	-
0	43	1	r-x
-	-	0	-

Tabela 4.69: Exemplo de mapeamento de páginas virtuais para físicas.

Como se pode ver no quadro 4.69 o sistema não iria funcionar correctamente pois as traduções para a página virtual zero dos dois processos carregados na TLB apontam para páginas físicas diferentes, conforme o processo. Tal situação, iria fazer com que ambos os processos acedessem à página física 20, levando a um estado de erro.

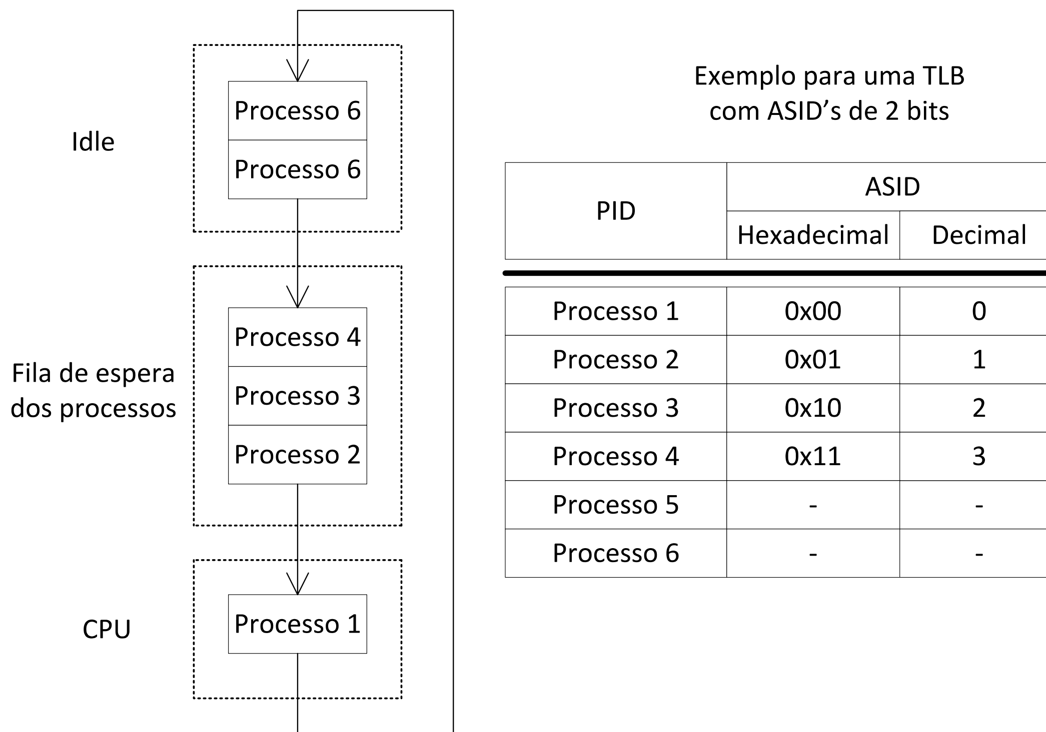
Devido a esta situação, os sistemas operativos criaram várias técnicas para melhorar a gestão da memória e ultrapassar este tipo de problemas, umas mais elaboradas que outras, sendo as mais conhecidas a *"TLB Flush on context switch"* e a utilização de ASID - *"Address Space Identifier"* para identificação de zonas de memória dos processos na TLB. Alerta-se para o facto que toda a gestão da memória é feita pelo sistema operativo e não pela arquitectura do sistema.

Uma forma bastante simples utilizada pelos sistemas operativos para gerir a mudança de contexto é limpando todo o conteúdo da TLB, também conhecido por "TLB Flush", assegurando assim que não existem entradas inválidas na TLB para aquele processo. É fácil de compreender que em todos os processos existe uma queda de desempenho quando começam a sua execução, pois é necessário gerar uma quantidade de *TLB Refill's* até

que existam referências válidas na TLB para o processo poder executar fluentemente. Se um sistema se encontrar num estado em que dois processos estejam sempre a comutar a sua execução, a descida de desempenho é notável.

Para sistemas que utilizem ASID's a gestão é um pouco mais complexa mas, em contrapartida há um aumento considerável no desempenho de todo o sistema. O sistema operativo é responsável por atribuir a cada processo um ASID único, que identificará o seu espaço de endereçamento. Assim é possível ter mapeados na TLB vários processos ao mesmo tempo sem que haja a necessidade de se fazer um "TLB Flush".

Nos casos em que haja uma mudança de contexto a probabilidade de o processo novo reutilizar as suas últimas entradas é bastante elevado o que por si só é um aumento no desempenho relativamente ao mecanismo "*TLB Flush on context switch*". Uma vez que o registo *EntryHi* da arquitectura MIPS apenas reserva 8 bits para o campo ASID o que acontece quando os ASID's estão todos atribuídos?



PID	ASID	
	Hexadecimal	Decimal
Processo 1	0x00	0
Processo 2	0x01	1
Processo 3	0x10	2
Processo 4	0x11	3
Processo 5	-	-
Processo 6	-	-

Figura 4.16: Escalonamento de processos com ASID.

Como se pode verificar na figura 4.16 o número máximo de ASID's esgotar-se-á mais rapidamente que o número de PID's (*Process Identifier*) uma vez que só temos 8 bits

para os ASID's o que nos dá um total de 256 ASID's diferentes na TLB. Nestes casos o sistema operativo tem a obrigação de confiscar um ASID's atribuído a um processo e atribuí-lo ao novo processo. A forma como é escolhido o processo que terá os seus mapeamentos removidos depende exclusivamente dos algoritmos utilizados pelo sistema operativo para gerir a memória, sendo os mais conhecidos: FIFO (*First In First Out*), LIFO (*Last In First Out*), LRU (*Least Recently Used*) e *Random*.

Outro aspecto importante a referir é a partilha de páginas físicas entre processos. Independentemente da técnica utilizada para gerir as entradas na TLB, os processos podem manter mapeamentos de páginas virtuais diferentes para a mesma página física mas o contrário já não é verdade como se pode ver no quadro 4.70.

VPN	**PFN**	**Válida**	**Permissão**	**Processo**
15	66	1	r-x	1
-	-	0	-	-
24	66	1	r-x	2
-	-	0	-	-
16	100	1	r-x	1

Tabela 4.70: Exemplo de mapeamentos com páginas físicas partilhadas.

No quadro 4.70 pode-se observar que existem dois processos com VPN's diferentes que apontam para a mesma PFN, respectivamente o processo 1 contém uma página virtual 15 que aponta para a página física 66, assim como o processo 2 contém uma página virtual 24 que aponta para a mesma página física. Esta situação é válida em sistemas operativos que permitam a partilha de memória entre processos. Em sistemas Unix existe um conjunto de técnicas apelidadas de IPC (*Inter-Process Comunication*) que permitem não só a comunicação entre processos mas também a partilha de zonas de memória assim como a partilha de código binário em *Shared library's*. A arquitectura MIPS32 permite a utilização das duas técnicas para a gestão da memória, mas pese o facto que nem todas as arquitecturas permitem a utilização de ASID's.

4.3.4 Excepções da TLB

A arquitectura MIPS32 dispõe de vários tipos de excepções criadas com o intuito de interromper o fluxo de execução e permitir ao sistema correr código binário de forma a resolver problemas ou estados de erro. Deste conjunto de excepções foram criados cinco tipos específicos para controlar o conteúdo e funcionamento da TLB.

Na secção 4.3.2 observou-se a forma como o programador pode controlar directamente

o conteúdo da TLB através das instruções disponibilizadas na ISA. Nesta secção serão abordados os mecanismos criados pela arquitectura para permitir ao sistema operativo gerir a TLB de forma coerente. Como visto anteriormente, a TLB possui uma tabela com mapeamentos de endereços virtuais para físicos, utilizada pelo processador para aceder à memória. Assim, surge uma importante questão:

O que acontece quando o processador tenta aceder a um endereço virtual que não se encontra mapeado na TLB?

Por defeito, quando o processador necessita aceder a um endereço virtual, ele acede à MMU que, após validar o segmento encaminha para a TLB, caso pertença ao USEG ou KSEG3. Nestas situações, a página virtual associada ao endereço virtual é procurada nas entradas da TLB e caso não exista nenhum mapeamento, o processador lançará uma excepção que suspenderá o fluxo de execução, alterando o seu PC para um endereço pré-calculado com a finalidade de resolver o endereço pedido. É de notar que há uma quebra no desempenho do sistema, uma vez que é necessário efectuar vários acessos à memória até que seja possível encontrar o respectivo mapeamento.

Assim que o mapeamento é descoberto pelo processador, este é adicionado na tabela interna da TLB de forma a permitir que futuros acessos, a curto prazo, sejam efectuados com rapidez. A interrupção do fluxo de execução que falamos acima é conhecida como *TLB Refill* e ocorre exactamente quando não existem mapeamentos na TLB que permitam resolver endereços virtuais em físicos. O seu funcionamento difere das restantes excepções, dado que pode gerar dois códigos de excepção e não apenas um, respectivamente a TLBL ou TLBS.

Independentemente do código de excepção atribuído, o valor é escrito no registo do coprocessador zero *StatusExcCode*. O primeiro caso ocorre se a instrução geradora da excepção for do tipo LOAD ou uma *instruction fetch* no processador, sendo a segunda situação não visível em contexto de simulação. No segundo caso, o processador encontra-se a executar uma instrução do tipo STORE. Em ambas as situações, o processador é responsável por guardar um conjunto de informação relativa ao seu estado na altura em que ocorre a excepção de forma a retomar o seu tratamento. No quadro 4.71 encontram-se os registos do coprocessador zero guardados no momento da excepção.

O tratamento da excepção *TLB Refill* contém um comportamento particular dado que os seus pontos de entrada dependem exclusivamente dos bits StatusBEV e StatusEXL. O primeiro bit indica o estado em que se encontra o processador, nomeadamente se ainda se encontra na fase de arranque (*Bootstrap*) ou não. O segundo bit indica se o processador se encontra actualmente a tratar uma excepção dado que iremos lançar uma nova excepção criando assim uma excepção aninhada ou *Nested Exception*.

Registo	**Estado do Registo**
BadVAddr	Endereço virtual que não tem mapeamento
Context	Se *Config3CTXTC* = 0, então *ContextBadVPN2* = $VA_{31..13}$; Se *Config3CTXTC* = 1, então é aplicada a máscara *ContextConfigVirtualIndex* com o registo *Context*.
EntryHi	Guarda o valor do VPN2 e o ASID do endereço virtual
EntryLo0	Imprevisível
EntryLo1	Imprevisível

Tabela 4.71: Registos do coprocessador zero salvaguardados quando ocorre a excepção TLB Refill.

Nestes casos em particular, é necessário que existam cuidados adicionais por parte dos programadores devido ao retorno das excepções.

No quadro 4.72 pode-se ver os pontos de entrada para a excepção *TLB Refill* conforme a configuração dos bits controladores.

Excepção	**StatusBEV**	**StatusEXL**	**Ponto de Entrada**
TLB Refill Vector	0	0	0x8000.0000
TLB Refill, General Vector	0	1	0x8000.0180
TLB Refill Vector	1	0	0xBFC0.0200
TLB Refill, General Vector	1	1	0xBFC0.0380

Tabela 4.72: Pontos de entrada para a excepção TLB Refill.

Como se pode ver, quando o StatusBEV se encontra activo todos os endereços são redireccionados para a zona alta do segmento KSEG1 que é uma zona não mapeada e que não utiliza *cache*. Neste estado o processador ainda se encontra em fase de arranque e os tratadores da excepção encontram-se mapeados na ROM da placa *board*. Quando o valor do StatusBEV é igual a zero o sistema encontra-se pronto para correr código do sistema operativo. Como podemos ver, os endereços têm como base o início do segmento KSEG0 que é uma zona não mapeada mas que utiliza *cache*.

O bit StatusEXL influência o valor do deslocamento no ponto de entrada em 0x180 no qual o tratamento da excepção pode mudar. Isto acontece, porque o *TLB Refill* é uma excepção que pode ocorrer dentro de outra excepção, no qual o sistema operativo deve ter em consideração alguns aspectos como salvaguardar manualmente os registos do coprocessador zero e executar o código de resolução de mapeamentos de páginas.

Caso o bit StatusEXL seja 0 o deslocamento é 0x0 e é considerado uma excepção do tipo *TLB Refill*. Mas, caso o bit StatusEXL seja 1 o deslocamento passa a ser 0x180 passando

a ser tratada como uma *General Exception.* A grande diferença consiste no facto de o código binário do vector *TLB Refill* correr directamente, enquanto que no caso em que passa pelo vector geral de excepções *TLB Refill, General Vector* será tratada com as restantes excepções existentes. Assim, é necessário efectuar várias comparações até se encontrar o tratador adequado para a excepção *TLB Refill* o que torna a resolução um pouco mais lenta, comparativamente com a primeira forma.

A excepção *TLB Modified* ocorre apenas em situações em o processador efectua uma operação do tipo STORE, no qual exista uma correspondência na TLB e contenha o bit *D (Dirty)* da página física activo. Nestes casos, o facto de o bit D da página física se encontrar activo, significa que a página física foi modificada ou se encontra a ser modificada por outro programa e não possui permissão para escrita.

Da mesma forma que acontece com o *TLB Refill*, é necessário salvaguardar os mesmos estados do processador, nomeadamente:

Registo	**Estado do Registo**
BadVAddr	Endereço virtual que não tem mapeamento
Context	Se *Config3CTXTC* = 0, então *ContextBadVPN2* = $VA_{31..13}$; Se *Config3CTXTC* = 1, então é aplicada a máscara *ContextConfigVirtualIndex* com o registo *Context.*
EntryHi	Guarda o valor do VPN2 e o ASID do endereço virtual
EntryLo0	Imprevisível
EntryLo1	Imprevisível

Tabela 4.73: Registos do coprocessador zero salvaguardados quando ocorre a excepção TLB Modified.

Em oposição do que acontece com a *TLB Refill*, a *TLB Modified* apenas contém um código de excepção Mod que é actualizado no registo *StatusExcCode.* O ponto de entrada do seu tratamento utiliza o deslocamento 0x180, o que significa que irá partilhar o mesmo tratador que todas as excepções que passam pelo *General Exception Vector*, à semelhança do que acontece com o *TLB Refill (General Vector).*

A excepção *TLB Invalid* ocorre quando existe uma tentativa de acesso a uma página física pelo processador, cuja correspondência na TLB exista e contenha o bit *V (Valid)* desactivado. Nos casos em que não exista mapeamento na TLB e o StatusEXL esteja activo (caso *TLB Refill* com *General Exception Vector*) não é possível distinguir os dois tipos de excepções uma vez que ambos geram os códigos TLBS e TLBL no registo StatusExcCode. A única forma de poder identificar inequivocamente o tipo de excepção é através da instrução TLBP, que validará se houve uma correspondência ou não na

TLB. Caso tenha ocorrido uma correspondência na TLB, então foi gerada uma *TLB Invalid*. Caso contrário ocorreu uma *TLB Refill*. Como acontece nas restantes excepções da TLB, é necessário salvaguardar os seguintes estados do processador como demonstra a tabela 4.74.

Registo	**Estado do Registo**
BadVAddr	Endereço virtual que não tem mapeamento
Context	Se *Config3CTXTC* = 0, então *ContextBadVPN2* = $VA_{31..13}$; Se *Config3CTXTC* = 1, então é aplicada a máscara *ContextConfigVirtualIndex* com o registo *Context*.
EntryHi	Guarda o valor do VPN2 e o ASID do endereço virtual
EntryLo0	Imprevisível
EntryLo1	Imprevisível

Tabela 4.74: Registos do coprocessador zero salvaguardados quando ocorre a excepção TLB Invalid.

Da mesma forma que acontece com a *TLB Modified*, o ponto de entrada do processador é o 0x180 e a excepção é tratada utilizando o *General Exception Vector*.

Além das excepções acima abordadas existem mais duas *Execute-Inhibit Exception* e *Read-Inhibit Exception*, cuja utilização depende da implementação do registo do coprocessador zero *PageGrain*. Dado que a sua implementação é obrigatória apenas em sistemas que suportem páginas com tamanho 1k (1024 bytes) e uma vez que o "SimuladorUE" não suporta esta funcionalidade, essas excepções não foram implementadas.

4.3.5 Simulação em Software

A TLB é um componente de carácter não obrigatório, sendo possível a sua anexação à MMU no processador. Este encontra-se implementado como um módulo externo ao processador que funciona como um repositório de mapeamentos.

A sua estrutura consiste num *array* de estruturas do tipo *struct TLB_Entry* que guarda um mapeamento de memória referente aos registos do coprocessador zero *EntryHi*, *EntryLo0 e 1*, *Pagemask* e *Wired*. Não existe uma única forma de implementação para a TLB uma vez que existem outros tipos de disposições e implementações possíveis para a TLB como o *array* simples, duplo, suportando ou não páginas de 1 KB de tamanho por exemplo. No presente trabalho apenas será abordado o modelo de TLB com um *array* simples sem suporte para páginas de tamanho 1 KB.

Uma vez que o sistema foi pensado para ser modularizado, o desenvolvimento de novos

formatos de implementação para a TLB podem ser anexadas no processador. Para tal é necessário respeitar algumas regras de comunicação, nomeadamente as assinaturas dos métodos que se encontram mapeadas no ficheiro “Config.h” do código fonte.

```
#ifdef TRANSLATION_TLB
        //Mecanismo de Traducao (MT)
        //Nome generico para a estrutura
        #define MT TLB

        //Nome do Mecanismo de Traducao
        #define MTTYPE "Translation Lookaside Buffer (TLB)"

        //Mapeamento de Funcoes da TLB
        #define traduz  procura_endereco

        #define tlbp_imp probe_for_match
        #define tlbwi_imp write_indexed_entry
        #define tlbwr_imp write_random_entry
        #define tlbr_imp read_indexed_entry

        #define mt_init init_tlb
        #define mt_print print_tlb
        #define mt_flush flush_tlb

#endif
```

Listing 4.17: Mapeamento de funções da TLB.

No bloco de código acima pode-se observar os mapeamentos para a TLB que se encontram no ficheiro “Config.h”. À semelhança do que acontece com o processador, o mecanismo de tradução de endereços também deve ser representado por um nome genérico, sendo este mapeado para uma implementação concreta, permitindo assim a adição e modificação de novos formatos. Para a implementação da TLB com *array* simples definiu-se a seguinte estrutura:

```
typedef struct TLB{
        //Array de entradas na TLB
        TLB_Entry *array[TLB_SIZE];

} TLB;

typedef struct TLB_Entry{
        /* Definir se a entrada e hardwired
         *
         * 0 - Not Wired
```

```
         * 1 - Wired
         */
        BYTE wired;

        //Virtual Page Number e ASID
        WORD entryHi;

        //Physical Page Number e Flags
        WORD entryLo0;
        WORD entryLo1;

        //Pagemask
        WORD pagemask;
} TLB_Entry;
```

Listing 4.18: Definição de um mapeamento na TLB.

A estrutura *TLB_Entry* guarda um conjunto de informação representado pelos registos do coprocessador zero e consistem num mapeamento da TLB. Cada índice da TLB é responsável por guardar um par de endereços virtuais e físicos, sendo que cada entrada é indexada pelo valor da página virtual, VPN. Em cada entrada pode-se ver os quatro registos necessários para a comunicação e mapeamento da TLB, nomeadamente: *EntryHi*, *EntryLo0*, *EntryLo1* e *Pagemask*. Estas variáveis guardam os valores necessários para se obter a resolução de um endereço virtual em físico sem necessidade de mecanismos adicionais, independentemente do tamanho de página.

É verdade que em cada entrada encontra-se um mapeamento, mas a leitura das variáveis por si só não retorna o valor do endereço físico correspondente. Para tal, é necessário construir o endereço através da figura 4.17.

Sempre que a MMU solicita uma tradução à TLB, a primeira é responsável por disponibilizar o endereço virtual assim como o ASID do processo em execução, sendo a responsabilidade de manter o ASID do processo no registo *EntryHi* do coprocessador zero exclusiva do sistema operativo. Assim, o primeiro passo na tradução dos endereços consiste, numa fase inicial, na divisão do endereço virtual em duas componentes: página virtual e no respectivo deslocamento. O tamanho utilizado para extrair os bits necessários, tanto para a VPN como para o deslocamento variam conforme o tamanho definido no valor da variável do mapeamento *Pagemask*. Desta forma, é possível que a TLB mantenha em memória mapeamentos de páginas com vários tamanhos.

Pegue-se como exemplo a tradução para uma página de tamanho 4K: os 20 bits mais significativos seriam utilizados para representar a VPN, e os 12 bits menos significativos para representar o deslocamento dentro da página física ou virtual. Agora, se o tamanho

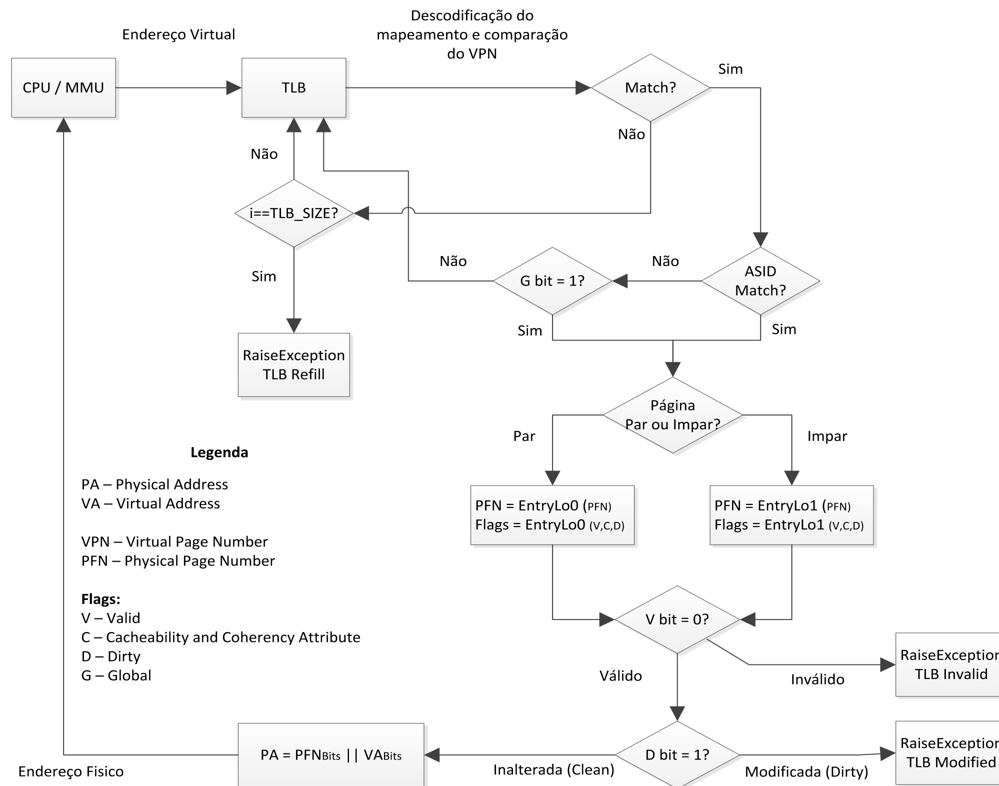


Figura 4.17: Diagrama de sequência para a tradução de endereços virtuais em endereços físicos.

das páginas for de 16K serão utilizados 18 bits para identificar a VPN e 14 bits para identificar o deslocamento. No quadro 4.75 pode-se observar a forma como os tamanhos da VPN e do deslocamento podem variar em função do tamanho das páginas. O quadro 4.75 não incluí suporte para PAE (*Physical Address Extension*) nem páginas físicas com tamanho 1K para a revisão 2 da arquitectura MIPS32.

Para cada entrada existente na TLB será necessário efectuar o processo de descodificação do endereço virtual, de forma a obter-se os valores para a VPN e o respectivo deslocamento. Desta forma, a TLB percorrerá a sua tabela interna à procura de ocorrências nas quais os valores da VPN e do ASID correspondam com os valores do mapeamento actual. É necessário comparar tanto a VPN como o ASID, uma vez que o processo em execução pode tentar aceder a uma zona de memória de outro processo o que não pode acontecer. Se a VPN e o ASID do endereço virtual corresponderem, significa que o mapeamento existe na TLB e pertence ao nosso processo.

Tamanho das Páginas	Bit Par/Impar	FPN	Desloc.	Endereço Físico
4K Bytes	VA_{12}	20 bits	12 bits	$PA = PFN_{25..6} \| VA_{11..0}$
16K Bytes	VA_{14}	18 bits	14 bits	$PA = PFN_{23..6} \| VA_{13..0}$
64K Bytes	VA_{16}	16 bits	16 bits	$PA = PFN_{21..6} \| VA_{15..0}$
256K Bytes	VA_{18}	14 bits	18 bits	$PA = PFN_{19..6} \| VA_{17..0}$
1M Bytes	VA_{20}	12 bits	20 bits	$PA = PFN_{17..6} \| VA_{19..0}$
4M Bytes	VA_{22}	10 bits	22 bits	$PA = PFN_{15..6} \| VA_{21..0}$
16M Bytes	VA_{24}	8 bits	24 bits	$PA = PFN_{13..6} \| VA_{23..0}$
64M Bytes	VA_{26}	6 bits	26 bits	$PA = PFN_{11..6} \| VA_{25..0}$
256M Bytes	VA_{28}	4 bits	28 bits	$PA = PFN_{9...6} \| VA_{27..0}$

Tabela 4.75: Tabela de geração de endereços físicos em função do tamanho de página.

Ao mesmo tempo que ocorre a validação do ASID, é igualmente feita uma comparação com o bit (G) *Global* das páginas físicas do mapeamento *EntryLo0 e 1*, PFN. Nos casos em que o bit se encontre activo, a comparação com o ASID é ignorada uma vez que a página física é partilhada no sistema inteiro. Caso contrário a entrada não é válida para o processo actual e a TLB passa para o mapeamento seguinte.

O passo seguinte consiste em identificar se o endereço virtual aponta para uma página física par ou impar, *EntryLo0* e *EntryLo1* respectivamente, através do bit *EvenOdd* que varia com o tamanho das páginas, como verificámos no quadro acima. Se o bit *EvenOdd* for zero, então a página física é par e será utilizado o PFN existente no *EntryLo0.* Caso contrario, a página é impar e será utilizado o PFN do *EntryLo1* no mapeamento da TLB.

Uma vez identificado o PFN correspondente ao endereço virtual é necessário validar duas situações, respectivamente se a página é válida e se não foi acedida por outro processo. Na primeira validação, a TLB verifica o valor do bit (V) *Valid* de forma a confirmar se o mapeamento existente é válido. Se o bit estiver activo, os acessos à página são válidos caso contrário a TLB lançará a excepção *TLB Invalid.* Na segunda verificação, o valor do bit (D) *Dirty* indica se a página foi acedida por outro processo. Se o bit não estiver activo, a página encontra-se limpa e pode ser acedida caso contrário a TLB lançará a excepção *TLB Modified.*

Oficialmente, a TLB verifica ainda o bit (C) *Cacheability and Coherency Attribute* mas, uma vez que a *Cache* não se encontra implementada, esta verificação não foi implementada. Neste estado, a TLB encontra-se pronta para gerar o endereço físico correspondente ao endereço virtual pedido pela MMU. Para tal, é necessário truncar os bits extra existentes no PFN encontrado e em seguida aplicar a operação lógica OR com o valor do deslocamento da página virtual.

Veja-se o seguinte exemplo para páginas de tamanho 4K: como descrito na tabela 4.75, para páginas de tamanho 4K o deslocamento utilizado corresponde aos doze bits menos significativos do endereço virtual. Relativamente aos 24 bits existentes no PFN dos *EntryLo0 e 1* do mapeamento, estes serão 'shiftados' para a esquerda N bits correspondentes ao tamanho das páginas, neste caso 12 bits. Em seguida será aplicada a operação lógica OR entre o PFN 'shiftado' e o deslocamento, formando assim o endereço físico correspondente.

A razão pelo qual os bits do PFN serão truncados nesta implementação deve-se ao facto de na arquitectura MIPS existir um símbolo PABITS (*Physical Address Bits*) que corresponde ao número de bits utilizado para representar o espaço de endereçamento físico. Desta forma, é possível utilizar 24 bits + 12 bits = 36 bits, em paginas de 4 KB, o que perfaz aproximadamente 2^{36} bits = 64 GB em endereços físicos, que é bastante superior ao convencional 2^{32} bits, 4 GB.

Esta técnica é conhecida como PAE (*Physical Address Extention*) e é exclusiva para arquitecturas de 32bits. Com a utilização de PAE, os processadores conseguem utilizar mais 4 bits na geração de endereços físicos permitindo ao sistema operativo utilizar memória RAM com tamanhos superiores a 4GB em arquitecturas de 32bits. Por defeito, os sistemas operativos Linux que correm na arquitectura MIPS32 utilizam o valor do PABITS igual a 32bits. Uma vez que o simulador não suporta a extensão PAE a implementação desta técnica não será abordada no processo de tradução de endereços.

Última fase na geração do endereço físico consiste em concatenar numa variável de 32 bits o valor do PFN truncado e o respectivo deslocamento, sendo os 4 bits utilizados pelo PAE ignorados. Após a geração do endereço físico a TLB envia o seu valor para a MMU que prosseguirá com o fluxo de execução de código binário. Nos casos em que não exista correspondência em nenhuma entrada da tabela interna da TLB, será lançada a excepção *TLB Refill*. Neste caso, o sistema operativo é responsável por encontrar e introduzir um mapeamento novo na TLB.

4.4 Memórias RAM e ROM

Como em qualquer sistema, é necessário a utilização de uma memória para armazenar dados e instruções pedidas pelo processador. Esta é a segunda peça essencial ao funcionamento de um sistema, sendo a primeira obviamente o processador, uma vez que todos os dados processados serão guardados temporariamente na memória. Assim, o segundo passo na implementação do simulador para a arquitectura MIPS32 é a criação de um dispositivo que funcione como uma memória RAM (*Random Access Memory*), de forma a criar uma área de trabalho para o processador.

Actualmente existem vários tipos de memórias, mas no âmbito do processo de simulação, apenas serão abordados dois tipos de memória, nomeadamente: memória principal RAM e memória de leitura ROM (*Read Only Memory*), para simulação do arranque do sistema, como veremos em seguida.

A memória RAM tem como função principal o armazenamento de informação, no qual está incluído todo o *kernel*, assim como toda a informação gerada pelos processos existentes no sistema operativo. Sem a RAM não é possível correr programas simples, ou complexos como sistemas operativos, uma vez que não existe uma área com permissões de leitura e escrita onde se possa guardar informação relevante para a execução dos programas. O seu funcionamento consiste em manter internamente informação de forma temporária e responder a todos os pedidos efectuados pelo processador. Qualquer pedido emitido pelo processador detém permissões de leitura e escrita em qualquer endereço existente na sua estrutura interna.

Devido à sua natureza, o conteúdo é considerado volátil uma vez que toda a informação existente na memória será perdida assim que o sistema for desligado. Em oposição do funcionamento da memória RAM, a ROM possui características exclusivas de leitura não permitindo a modificação do seu código interno. Em sistemas actuais, esta memória guarda a seguinte informação: BIOS (*Basic Input Output System*), POST (*Power On Self Test*) e configuração.

A BIOS é um programa que disponibiliza suporte básico de acesso ao *hardware*, enquanto que o programa POST é utilizado para validar o estado dos componentes ligados na placa mãe. Os modelos mais recentes permitem a parametrização da BIOS de forma a permitir ao utilizador escolher qual o dispositivo de arranque. Após a validação do *hardware*, é responsabilidade da BIOS encaminhar o PC do processador para um dispositivo de arranque, de forma a carregar o código do sistema operativo para memória, como exemplo um disco rígido ou um CD ROM. Em contexto de simulação, a memória ROM não possui carácter obrigatório, podendo ser utilizada ou não.

Pegando em simuladores actuais, temos o *software* VirtualBox da Oracle, que não utiliza uma BIOS para arranque das suas máquinas virtuais. Por outro lado pode-se ver que os *softwares* VMWare ou QEMU dispõem de BIOS para arranque dos sistemas instalados. De forma a tornar o comportamento do simulador o mais real possível, optou-se por permitir a utilização de uma BIOS para configuração e arranque do processador.

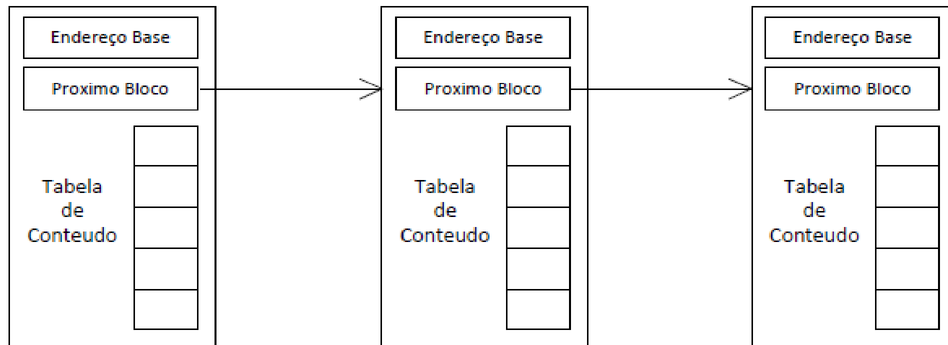


Figura 4.18: Disposição da memória RAM.

4.4.1 Simulação em Software

A memória RAM é um componente de carácter obrigatória em contexto de simulação. A sua representação encontra-se implementada nos ficheiros “MemoriaPrincipal.c” e “MemoriaPrincipal.h”. O processo de implementação da memória RAM não seguiu uma especificação em particular, sendo a sua representação feita utilizando um modelo genérico do funcionamento de uma RAM. Assim a componente RAM possui as seguintes características:

1. Toda a informação armazenada na RAM encontra-se organizada em blocos de 32 bits;
2. É possível aceder e armazenar informação em blocos de 32 bits (WORD), alinhadas com blocos de 16 (HALFWORD) ou 8 bits (BYTE);
3. A comunicação é feita através dos canais Tx (Transmissor) e Rx (Receptor) na estrutura RAM;
4. O modelo de mensagens utilizado é o `struct mem_request`, que encontra-se definido no ficheiro “ConnDef.h”;

Na construção da memória RAM, o primeiro impulso foi a criação de um `array` muito grande para guardar a informação. Esta representação foi rapidamente abandonada, uma vez que o espaço ocupado seria muito elevado tornando impraticável a manipulação e tratamento da mesma. Desta forma, optou-se pela divisão da memória em segmentos através da criação de uma estrutura de dados em formato de “lista ligada” como demonstra a figura 4.18.

O principal objectivo é tornar a manipulação da RAM prática e flexível. Dado que a grande maioria dos sistemas utilizam a técnica de paginação com páginas de tamanho 4K, decidiu-se que a divisão dos blocos seria feita à semelhança do que acontece com a descodificação dos endereços virtuais[11].

Os vinte bits mais significativos serão utilizados para agrupar um subconjunto de endereços num bloco, sendo os restantes doze bits utilizados como índice dentro do próprio bloco para aceder à informação. Com este mecanismo, a RAM possuirá um comportamento "on demand" uma vez que o espaço apenas crescerá em função dos pedidos efectuados pelo processador.

Veja-se o seguinte exemplo: se o simulador utilizar uma RAM com tamanho 4 MBytes, com blocos de tamanho 4KBytes e o programa em execução no simulador apenas utilizar endereços no intervalo 0x80000000 e 0x80004000[12], a memória RAM apenas utilizará ((0x4000 / 4K) * tamanho de cada bloco) em vez dos 4 MBytes. Desta forma, o espaço ocupado é reduzido drasticamente, permitindo uma melhor manipulação assim como a *depuração* do estado da memória e seu conteúdo.

Como se pode observar no quadro 4.18, cada bloco da RAM é composto por um endereço base que identifica o bloco e o subconjunto de endereços, a tabela de conteúdo que consiste num `array` com tamanho $2^{12} = 4096$ e um apontador para o próximo bloco. Sempre que é necessário adicionar um novo bloco, a adição não é feita de modo desorganizado, isto é, o novo bloco é adicionado de forma ordenada como demostra a figura 4.19.

Ao contrário do processador, a RAM não funciona dentro do processo principal, sendo invocada num novo processo. Assim, a RAM possui internamente um tratador de pedidos, que funciona de forma interminável respondendo aos pedidos efectuados pelo processador. A comunicação é efectuada através das duas variáveis Tx (Transmissor) e Rx (Receptor), como demonstra o bloco de código em 4.19, que representam os canais de comunicação entre a RAM e o processador.

Da mesma forma que o processador, a RAM utiliza a estrutura `struct mem_request` para receber e enviar os pedidos efectuados pelo processador.

Cada pedido emitido é descodificado, identificando primeiro o tipo de operação, LOAD ou STORE através do campo LorS, sendo em seguida preenchido uma estrutura idêntica com a nova informação recolhida da memória. Se ocorrer algum erro na execução, o

[11]Não confundir a tradução de endereços com a divisão de bits na RAM. Este mecanismo é utilizado para reduzir o espaço ocupado pela RAM no simulador não tendo qualquer ligação com a arquitectura MIPS32.

[12]Excluindo o endereço 0x80004000, caso contrário seria necessário adicionar um novo bloco na RAM.

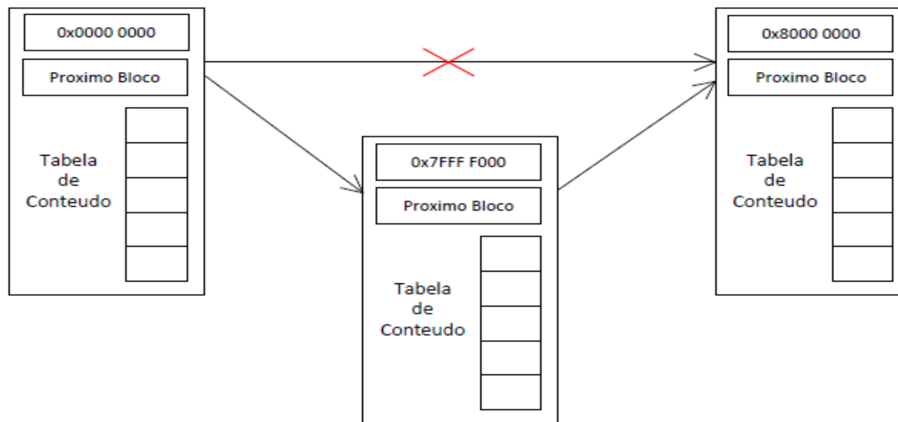


Figura 4.19: Introdução de um novo bloco na RAM.

campo RC será preenchido com um código representativo do erro. O campo Control é utilizado exclusivamente pelo o simulador para enviar pedidos de PRINT ao processo responsável pela RAM, de forma a imprimir o conteúdo da RAM num ficheiro.

Todas as operações efectuadas pela RAM são registadas localmente, de forma a auxiliar o processo de depuração da mesma. Desta forma, são gerados dois documentos na execução da RAM: *dump* da memória e o registo de operações. O primeiro consiste numa fotografia da RAM num determinado momento da sua existência, enquanto o segundo é utilizado para registar todas as operações efectuadas na RAM, nomeadamente: LOAD, STORE, PRINT e SHUTDOWN.

```
//Tamanho da RAM em bytes (4 Mb = 4,194,304)
#define RAM_SIZE      (4*1024*1024)

//Canal de transmissao (RAM Out, CPU In)
int Tx;
//Canal de recepcao (RAM In,  CPU Out)
int Rx;

typedef struct RAM_List {
        //Endereco Base na RAM
        WORD endereco_base;

        //Tabela com deslocamentos
        WORD tabela[SIZE];

        //Proximo No da Lista (RAM)
```

```
        struct RAM_List *next_block;
    } RAM_List;
```

Listing 4.19: Representação da estrutura memória RAM.

Em oposição à RAM, a memória ROM não dispõe de carácter obrigatório, sendo a decisão de utilização e implementação exclusiva do simulador. No presente trabalho, optou-se por dar suporte à utilização de uma memória ROM, e a sua implementação encontra-se nos ficheiros "MemoriaROM.c" e "MemoriaROM.h".

Uma vez que este componente possui relativamente pouca informação, decidiu-se utilizar um `array` simples com tamanho pré definido igual a 128 bytes para guardar a sua informação.

```
#define ROM_SIZE      128

typedef struct ROM_t {
        //0 - open
        //1 - lock
        int lock;

        int Tx;
        int Rx;

        WORD tabela[ROM_SIZE];
} ROM_t;
```

Listing 4.20: Representação da estrutura memória ROM.

O funcionamento da memória ROM é muito semelhante ao funcionamento da RAM, uma vez que em ambos os casos o processo responsável pelo simulador, gera um novo processo para a execução da memória. O processo de comunicação funciona exactamente da mesma maneira que acontece com a memória RAM, através da troca de mensagens no formato `struct mem_request`.

Assim como acontece com a RAM, o processador pode fazer pedidos, com a diferença que deverão ser sempre LOAD's[13], caso contrário a ROM não executará a instrução e retornará um código de erro na variável RC da mensagem, sendo tratado pelo o 'Dispatcher' do simulador. A existência da variável *lock* serve precisamente para não permitir a execução de um STORE na memória ROM por parte do processador, sendo este gerido pelo o simulador.

[13] O processo de inicialização da ROM é descrito em detalhe na secção 4.5

As variáveis Tx e Rx são utilizadas da mesma forma, representando os canais de comunicação criados pelo simulador, no qual a memória lê do Rx informação proveniente do processador e o Tx é utilizado para enviar a resposta de volta para o processador.

4.5 Simulador UE

Uma vez que todos os componentes e mecanismos utilizados na arquitectura MIPS32 foram devidamente explanados, está na altura de introduzir o componente central que possibilita a integração entre os vários módulos.

Nesta secção serão abordadas a forma como o simulador inicia o seu funcionamento preenchendo as suas estruturas de dados internas com informação em 4.5.1. Será também abordada a forma como o simulador carrega os ficheiros binários no formato ELF, passados como argumento do simulador.

Nas secções 4.5.2 e 4.5.3 serão demonstrados o motor de execução do simulador, assim como os seus mecanismos internos e procedimentos de encerramento.

4.5.1 Carregamento e arranque do sistema

O procedimento de inicialização do sistema consiste em três fases: validação de argumentos, criação e instanciação das estruturas definidas no ficheiro de configuração "Config." e carregamento do programa para memória. A primeira fase é composta pela validação dos argumentos uma vez que o simulador pode iniciar a sua execução de uma das seguintes formas:

```
./SimuladorUE Programa.bin
ou
./SimuladorUE Programa.bin ROM.bin
```

Em ambos os casos o programa passado como argumento deverá respeitar o formato ELF, caso contrário o sistema não compreenderá a organização da sua informação e abortará a execução. Caso o segundo argumento seja utilizado, este deverá respeitar igualmente o formato ELF e será carregado para a componente ROM existente no simulador[14]. Nestes casos o PC não é inicializado uma vez que o arranque do sistema será feito através da inserção do sinal *Reset* directamente no processador, como se de uma sistema real se tratasse. Se o segundo argumento não for utilizado, então o PC do processador

[14] Os mapeamentos para o dispositivo são efectuados pelo simulador, como acontece em placas mâe na actualidade. Por defeito a ROM encontra-se no endereço virtual 0xBFC00000, que corresponde ao endereço físico 0x1FC00000.

é inicializado com o endereço existente no *Entry Point* do ficheiro programa binário.

Quando se iniciar o ciclo de execução do simulador, o processador começará a executar as instruções apontadas pelo registo PC, como definido na arquitectura MIPS32. Uma vez concluída com sucesso a validação dos argumentos, inicia-se o processo de criação das estruturas de dados definidas e mapeadas através do ficheiro "Config.h" que darão suporte ao funcionamento do simulador. Todas as estruturas são definidas de forma genérica, seria bastante deselegante para o simulador apenas permitir um tipo de processador, ou cada vez que se pretendesse mudar de processador seja necessário fazer modificações no código fonte do simulador.

Para tal situação não acontecer, optou-se pela utilização de macros com a finalidade de mapear todas as estruturas existentes, assim, o simulador apenas vê um tipo genérico de processador, memória ou mecanismo de tradução de endereços e não as suas implementações específicas. Desta forma é possível desenvolver um periférico novo customizado e integra-lo no simulador sem modificações no código fonte do mesmo. Embora a implementação seja livre é necessário que o programador siga um conjunto de regras de forma a mapear correctamente as funções genéricas para funções específicas do seu periférico no ficheiro de configuração.

No excerto de código abaixo pode-se visualizar a forma como o simulador mapeia a estrutura do processador.

```
// MIPS CPUs
// MIPS Core 32
#define MIPS32_Core

// MIPS Core 64
//#define MIPS64_Core

//Mapeamento de Variaveis e Funcoes
#ifndef CORETYPE
#define CORETYPE

        //ERRO
        #if defined( MIPS32_Core ) && defined( MIPS64_Core )
                #error "Nao pode haver dois processadores diferentes
                    a funcionar ao mesmo tempo!"
        #endif

        // Processadores de 32 Bits
        #ifdef MIPS32_Core

                //Definir Revision
```

```
            #define ARCH_REV 2

            //Nome do Core
            #define CORE "MIPS 32 bits"

            //Nome da Struct CPU
            #define CPU CPU_32

            //Nome do MMU
            #define MMUTYPE "MMU para arquitectura de 32Bits"

            //Executa
            #define executa executa32

    #elif defined( MIPS64_Core )
            // Processadores de 64 Bits

            //Nome do Core
            #define CORE "MIPS 64 bits"

            //Nome da Struct CPU
            #define CPU CPU_64

            //Nome do MMU
            #define MMUTYPE "MMU para arquitectura 64Bits"

            //Executa
            #define executa executa64

    #endif
#endif
```

Listing 4.21: Mapeamento de funções e estruturas do Simulador.

No exemplo acima, pode-se ver a definição e mapeamento de dois tipos de processadores, um processador para a arquitectura MIPS32 e outro para a arquitectura MIPS64. É fácil de compreender que não é possível ter dois processadores, de arquitecturas diferentes a funcionar ao mesmo tempo, assim é necessário configurar qual a arquitectura que deve ficar activa.

Veja-se o seguinte exemplo: a estrutura `struct CPU_32`, representa o nome da estrutura internamente na implementação do processador de 32 bits. Os nomes utilizados em cada implementação não são importantes mas sim o nome utilizado pelo simulador para reconhecer a estrutura.

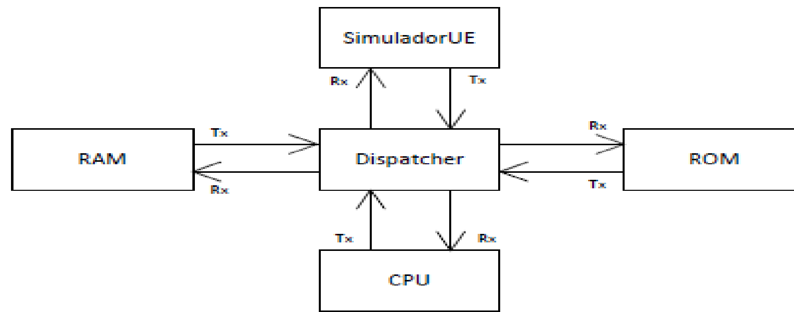


Figura 4.20: Disposição dos canais no simulador.

Desta forma, a estrutura `CPU` no código do simulador deverá mapear os nomes das estruturas dos processadores, neste caso, o nome `CPU_32`.

O mesmo acontece com as funções que mapeiam a função responsável pelo controlo do fluxo de execução do processador, nomeadamente: "executa32" para "executa". A vantagem da utilização de macros é o facto de esta criar uma camada de abstracção entre o simulador e os periféricos de forma a permitir que os mesmos possam ser utilizados em outros simuladores ou versões melhoradas do actual.

O passo seguinte consiste na criação e instanciação dos canais de comunicação utilizado pelos periféricos ligados no simulador. Para tal, o simulador criará e manterá internamente numa tabela de descritores, todos os canais de comunicação entre os vários periféricos. Na função de inicialização de cada periférico, são passados adicionalmente dois canais, o Tx e o Rx respectivamente, para que possuam um canal de transmissão e outro para recepção de dados.

Oficialmente, o processador e a RAM não se encontram ligados directamente, mas sim através da placa mãe. Esta placa dispõe de um chip interno que funciona como *hub* de dispositivos, no qual todos os periféricos se ligam, permitindo o mapeamento e o encaminhamento de endereços para dispositivos diferentes. Em contexto de simulação este chip é conhecido como "Dispatcher" e a sua representação encontra-se implementada nos ficheiros "Dispatcher.c" e "Dispatcher.h". Na figura 4.20 encontra-se demonstrado a disposição do simulador após o estabelecimento dos canais de comunicação.

O *Dispatcher* funciona como um módulo separado do simulador, sendo lançado em um novo processo. A sua principal função é redireccionar pedidos efectuados pelo processador para o periférico correspondente, de acordo com a tabela 4.76.

Esta técnica de redirecionamento de endereços é conhecida como MMIO (*Memory Mapped Input/ Output*) e é utilizada pelos processadores para mapear endereços físicos entre

Dispositivo	Endereço início	Endereço fim
RAM 1	0x20000000	0xFFFFFFFF
ROM	0x1FC00000	0x1FFFFFFC
Unused	0x08C00000	0x1FBFFFFC
IODEV	0x08400000	0x08BFFFFC
Clock	0x08000000	0x083FFFFC
RAM 0	0x00000000	0x07FFFFFC

Tabela 4.76: Tabela de mapeamento de endereços fisícos para dispositivos.

os diferentes dispositivos. A forma como os endereços são mapeados varia de acordo com a implementação da placa mãe, neste caso do simulador. A tabela de mapeamentos utilizada baseou-se nos mapeamentos utilizados pela placa mãe Malta [MT02].

Uma vez criadas as estruturas, mapeamento de funções, assim como estabelecidos os canais de comunicação, o simulador encontra-se pronto para carregar o código do programa binário para memória. No carregamento do programa para a memória, surge uma situação, apenas em configurações específicas, que necessitam de inicialização e suporte por parte do simulador.

Estas situações ocorrem exactamente quando os programas binários contêm código ou dados em zona de memória mapeada. Nestes casos o simulador não sabe onde colocar o código uma vez que não existem, préviamente, mapeamentos na TLB, uma vez que essa responsabilidade pertence ao sistema operativo. Em sistemas reais este tipo de problema não existe uma vez que o *kernel* do sistema operativo é carregado para segmentos de memória não mapeados, nomeadamente KSEG0, sendo em seguida introduzidos mapeamentos na TLB, permitindo o carregamento de novos programas para memória mapeada.

Actualmente, o simulador não dispõe de suporte para controladores de periféricos de armazenamento permanente, como os discos rígidos. Desta forma, os programas que correm no simulador não conseguem aceder a ficheiros de forma a permitir o carregamento de novos programas para zonas de memória mapeadas. Assim, optou-se pela introdução de mapeamentos manualmente, de forma a permitir o carregamento de código e dados para zonas mapeadas. Estes mapeamentos manuais dispõem de um limite máximo de dez mapeamentos, endereços virtuais e físicos na TLB, que serão utilizados para o carregamento inicial dos programas.

Para tal é necessário introduzir as seguintes directivas no ficheiro "Config.h":

```
//TLB - Mappings
#define tlb_map_entryHI_0   0x7FFFE000 //VPN + ASID + FLAGS
#define tlb_map_entryLO0_0  0x00000100 //Pagina Fisica Par
#define tlb_map_entryLO1_0  0x00000140 //Pagina Fisica Impar
#define tlb_map_mask_0      0x00001800 //Paginas de 4k

#define tlb_map_entryHI_1   0x00400000 //VPN + ASID + FLAGS
#define tlb_map_entryLO0_1  0x00000080 //Pagina Fisica Par
#define tlb_map_entryLO1_1  0x000000C0 //Pagina Fisica Impar
#define tlb_map_mask_1      0x00001800 //Paginas de 4k

#define tlb_map_entryHI_2   0xE0000000 //VPN + ASID + FLAGS
#define tlb_map_entryLO0_2  0x00000160 //Pagina Fisica Par
#define tlb_map_entryLO1_2  0x00000180 //Pagina Fisica Impar
#define tlb_map_mask_2      0x00001800 //Paginas de 4k

#define tlb_map_entryHI_3   -1
#define tlb_map_entryHI_4   -1
#define tlb_map_entryHI_5   -1
#define tlb_map_entryHI_6   -1
#define tlb_map_entryHI_7   -1
#define tlb_map_entryHI_8   -1
#define tlb_map_entryHI_9   -1
```

Listing 4.22: Mapeamentos manuais na TLB.

Desta forma é possível gerar programas que corram em qualquer segmento de memória, mapeada ou não mapeada. Resolvidos os problemas de mapeamentos, o simulador pode introduzir o código de cada secção respectivamente para o seu endereço de memória virtual. O carregamento é feito por secção e a ordem de introdução é decidida pelo ficheiro binário. O simulador apenas se encarrega de copiar o conteúdo de cada secção para o endereço físico resolvido, através das funções *standard* do processador, nomeadamente:

```
WORD LoadMemory(CPU *cpu, BYTE AcessLength, WORD vAddr);
void StoreMemory(CPU *cpu, BYTE AcessLength, WORD vAddr, WORD Info);
```

Listing 4.23: Funções standard de leitura e escrita do SimuladorUE.

Uma vez que no ficheiro binário existem apenas referências a endereços virtuais, não há necessidade, por parte do simulador, aceder directamente à memória física para introduzir código. Embora os ficheiros no formato ELF permitam, para determinadas arquitecturas, guardar informação relativa a endereços físicos, esta funcionalidade não é suportada pelo simulador.

Todos os endereços são processados pela unidade de gestão de memória do processador em funcionamento, daí a necessidade de haver mapeamentos na TLB antes do carregamento do programa. De modo a possibilitar a detecção do estado final da execução do código binário, decidiu-se introduzir duas instruções, `NOP` e `END` no final de cada secção. Nas secções posteriores serão abordados em detalhe a forma e os motivos pelo quais é necessário a utilização das duas instruções.

Nos casos em que não seja utilizada memória ROM, após o carregamento do ficheiro binário para a memória do sistema, o simulador procede à alteração do PC do processador sobrepondo-o com o valor do *Entry Point* definido no ficheiro binário, iniciando assim o processo de execução. Caso seja passado uma memória ROM como argumento, o simulador copiará o conteúdo do ficheiro binário ROM para dentro da memória ROM interna no simulador. Em seguida, de forma a iniciar o processo de execução, o simulador introduzirá um sinal de "Cold Reset" através da função `RaiseException3(CPU *cpu)` de forma a simular um arranque real do processador.

Como visto na secção 4.1.8, quando o sinal de *Reset* é introduzido no processador o valor do PC é modificado para 0xBFC00000, passando assim a execução para a "BIOS" residente na memória ROM, sendo esta responsável por dar continuidade à execução do código.

4.5.2 Execução

Após a conclusão da inicialização do sistema o simulador encontra-se pronto para executar o código carregado em memória. A sua execução inicia-se com a primeira instrução na memória, RAM ou ROM dependendo dos argumentos utilizados, que se situa no endereço apontado pelo registo PC.

Esta execução consiste num ciclo infinito no qual o simulador envia sucessivos pedidos de iterações ao processador, através da função `executa`, responsável pelo o controlo do fluxo de execução interna do mesmo.

```
void executa(CPU *cpu);
```

Listing 4.24: Função iterativa executa.

No capítulo 4.1 observou-se em detalhe o funcionamento e a estrutura acima mencionada. Em todas as iterações do ciclo de execução, o simulador valida se o estado do processador é *CPU_HALT* (*Idle*) com o objectivo de suspender ou abortar o fluxo de execução, sendo no segundo caso activado o procedimento de encerramento.

Uma vez que a velocidade de execução do código binário é elevada, impossibilitando a

leitura em tempo de execução, desenvolveu-se um mecanismo de *debug* que permite aos utilizadores validar a informação que se encontra em processamento num determinado instante ou endereço. Para tal é necessário configurar uma lista de endereços de *debug* definidos no ficheiro "Config.h", que parametrizará o simulador de forma a possibilitar a interrupção momentânea do fluxo de execução nos endereços customizados pelo utilizador. Este mecanismo permitirá também a execução de funções auxiliares como "fotografias" do estado de cada componente no sistema, nomeadamente, os registos do processador, mapeamentos na TLB, gerar *dumps* da memória física ou da tabela de mapeamentos da TLB. Será possível, em determinadas situações, a modificação manual de alguns componentes, como eliminar todos os mapeamentos não fixos existentes na TLB com o objectivo de demonstrar alguns comportamentos da arquitectura.

4.5.3 Término de Execução

O simulador permite a execução de código binário, mas o que acontece quando o código desenvolvido em memória termina? Na realidade a memória contém informação, independentemente da sua validade, o que levaria o processador a tentar interpretar e executar essa mesma informação.

Desta forma, levanta-se uma importante questão que é a forma como o simulador termina a sua execução. Os programadores não devem ter de se preocupar com esta questão, sendo a mesma responsabilidade do sistema operativo. Em sistemas reais, o sistema operativo gera uma excepção do tipo NMI, responsável por enviar um sinal de encerramento para a fonte de alimentação. Neste caso em concreto é o simulador que detém a responsabilidade de detectar o estado final e encerrar o processo de execução. Para tal, decidiu-se introduzir as instruções `NOP` e `END` no fim de cada segmento de código presente no ficheiro binário.

Segundo [MT10b] a instrução `NOP` é constituída por uma WORD de 32 bits, no qual todos os bits são iguais a zero. Esta instrução é interpretada pelo processador como uma instrução SLL *(Shift Left Logical)*. Relativamente à instrução `END`, esta não existe por defeito na ISA MIPS32 revisão 2, sendo que a sua codificação se encontra disponível para utilização customizada como se pode observar no quadro 4.77.

Desta forma decidiu-se utilizar os primeiros seis bits da tabela de codificação *opcode*, nomeadamente 011000, para codificar a instrução `END`, que activará o estado *CPU_HALT* no processador, que por sua vez servirá como ponto de saída para o simulador. No quadro 4.77 pode-se ver a codificação base das instruções para a ISA MIPS32 revisão 2 segundo [MT10b] e o local escolhido para codificar a instrução `END`.

OPCODE		bits 28..26							
		0	1	2	3	4	5	6	7
bits 31..29		000	001	010	011	100	101	110	111
0	000	SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	010	COP0	COP1	COP2	COP1X	BEQL	BNEL	BLEZL	BGTZL
3	011	**END**	*	*	*	SPECIAL2	JALX	*	SPECIAL3
4	100	LB	LH	LWL	LW	LBU	LHU	LWR	*
5	101	SB	SH	SWL	SW	*	*	SWR	CACHE
6	110	LL	LWC1	LWC2	PREF	*	LDC1	LDC2	*
7	111	SC	SWC1	SWC2	*	*	SDC1	SDC2	*

Tabela 4.77: Codificação para a instrução END.

Sempre que surja uma instrução com esta codificação o processador terminará a sua execução e por consequência o simulador encerrará o seu funcionamento. A razão pelo qual foi escolhido esta codificação e não outra deve-se ao facto de que esta é a tabela global que mapeia todas as instruções MIPS. É fácil de notar que não se encontram nesta tabela todas as instruções existentes na ISA MIPS32, uma vez que existem codificações que são subconjuntos de outras codificações, nomeadamente a instrução `ADD` que é uma sub codificação do grupo `SPECIAL`.

No capítulo 4.1, na interpretação das instruções, é posível ver que existem instruções que só necessitam de uma comparação para identificar a instrução existindo outros casos, em que é necessário fazer duas ou mais comparações. Tome-se como exemplo a instrução `ADD`: para que o processador consiga identificar a instrução e os seus argumentos, a sua primeira tarefa é identificar o tipo de codificação através dos seus seis bits mais significativos, respectivamente os bits 31 a 26. Neste caso, identificará o sub conjunto `SPECIAL`, no qual será necessário interpretar os seis bits menos significativos, bits 0 a 5 que irá identificar a instrução `ADD`, sendo os restantes bits da instrução utilizados para identificar os seus argumentos.

De forma a minimizar o número de comparações decidiu-se utilizar a primeira codificação disponível na tabela global de codificações. Esta técnica de introdução das duas instruções, `NOP` e `END`, levanta alguns perigos na sua utilização, exclusivamente quando existem secções no ficheiro binário alinhadas e desordenadas.

Veja-se o seguinte exemplo: um ficheiro binário contém duas secções com endereços não-alinhados, sendo que a secção A possui 50 instruções no endereço 0x80000000 e a secção B possuí 20 instruções no endereço 0x80000180. Quando estas secções forem carregadas para memória não haverá problemas de sobreposição de instruções.

Agora nos casos em que as secções se encontrem alinhadas e desordenadas, relativamente aos seus endereços no ficheiro binário, pode existir sobreposições de instruções caso não sejam tomadas precauções. Peguemos em outro exemplo no mesmo seguimento, um ficheiro binário com duas secções sendo que a secção A possui 10 instruções no endereço 0x80000000 e esta se encontra alinhada com a secção B com 5 instruções no endereço 0x80000028.

Se a secção B for introduzida na memória em primeiro lugar, quando a secção A for introduzida haverá sobreposição de informação. De forma a resolver este problema, o simulador antes de introduzir as duas instruções na memória verifica se os seus valores são iguais ao valor preenchido por defeito na RAM de forma a preservar qualquer informação carregada em memória anteriormente. Este é um comportamento perigoso por parte do simulador mas uma vez que os ficheiros de teste não contêm secções alinhadas optou-se por esta solução.

Após a ocorrência da instrução `END` o simulador abortará o seu ciclo de execução, iniciando assim o processo de encerramento. Este processo consiste no envio do sinal `SIGKILL` aos processos filhos, o qual será interceptado pelos mesmos, iniciando igualmente em seguida o processo de encerramento. Assim, em todos os processos filhos, são terminados e fechados os ficheiros de logs, sendo que em alguns casos como a RAM, será gerado um dump adicional com o estado da memória, assim como o processador e seus respectivos registos.

Capítulo 5

Utilização do Sistema

Este capítulo é dedicado à utilização, interface e ao modo como o utilizador interage com o sistema. Aqui serão demonstrados alguns exemplos da utilização do simulador, destacando as funcionalidades mais importantes:

- Modos de execução *Kernel* e *User*;
- Registos do coprocessador central CP0;
- Excepções associadas ao vector geral;
- Utilização das excepções *TLB Miss*, *TLB Invalid* ou *TLB Modified* para alocação de páginas na TLB e na memória RAM;
- Utilização de chamadas ao sistema (*System Calls*);
- Geração e tratamento de erros de execução (*runtime error*) como: *Address Error* (AdES ou AdEL), *Coprocessor Unusable* ou erros na TLB (TLBL, TLBS ou Mod);
- Integração de código compilado utilizando GCC no simulador;

A secção 5.1 descreve a interface desenvolvida para o sistema e a estrutura que os programas devem respeitar, de forma a permitir a sua execução no simulador. Na secção 5.2 são apresentados alguns casos aplicados com sucesso no simulador.

5.1 Interface e funcionamento

Em oposição aos simuladores SPIM e MARS, que permitem a programação do código fonte, o presente simulador apenas permite a execução de código binário préviamente compilado. De forma a possibilitar a execução de código no simulador, é necessário que cada programa seja compilado e linkado através de um compilador.

Para programas desenvolvidos directamente através de código `assembly`, ficheiros com extensão ".asm" ou ".S", é necessário utilizar um `assembler` para converter o código fonte em código binário executável. Uma vez que nos encontramos a desenvolver código binário numa arquitectura diferente da arquitectura MIPS, nomeadamente x86, é necessário utilizar ferramentas que suportem a compilação cruzada de código.

De forma a ultrapassar esta questão, optou-se pela utilização do *Workbench* da MIPS Inc, dado que dispõe de um conjunto de ferramentas para desenvolvimento, compilação e depuração de código binário para a arquitectura MIPS, suportando várias ISA's. Assim, utilizou-se o *software* `mips-gnu-linux-as` para compilar o código fonte `assembly` para código máquina. Existe no entanto outra forma, nomeadamente através do desenvolvimento de código na linguagem de programação C/C++. Nestes casos é necessário a utilização do compilador `mips-gnu-linux-gcc`, existindo ainda alguns problemas na integração entre o simulador e as bibliotecas dinâmicas da linguagem, como exemplo a biblioteca `glibc`.

Em ambos os casos, em determinadas situações, é necessário modificar alguns elementos dos ficheiros executáveis gerados, nomeadamente os endereços das secções existentes ou o ponto de entrada do ficheiro. A situação ocorre, uma vez que o `mips-gnu-linux-gcc` e o `mips-gnu-linux-as` geram secções cujo o endereço base começa no endereço virtual 0x00000000, sendo uma zona de memória mapeada, que não é possível aceder sem mapeamento prévio.

Existe também a necessidade de modificar os endereços de algumas secções, uma vez que o objectivo da demonstração foca-se nos mecanismos da arquitectura, no qual os testes às excepções necessitam de tratadores em endereços pré calculados, como abordado na secção 4.1.8. O mesmo se aplica ao ponto de entrada do ficheiro ELF, uma vez que este pode variar conforme o tipo de demonstração, sendo os endereços de entrada mais comums 0x00400000 ou o 0x80000000.

Na figura 5.1 é possível visualizar o processo de desenvolvimento e compilação dos ficheiros de teste gerados, com o `assembler as` ou pelo compilador GCC.

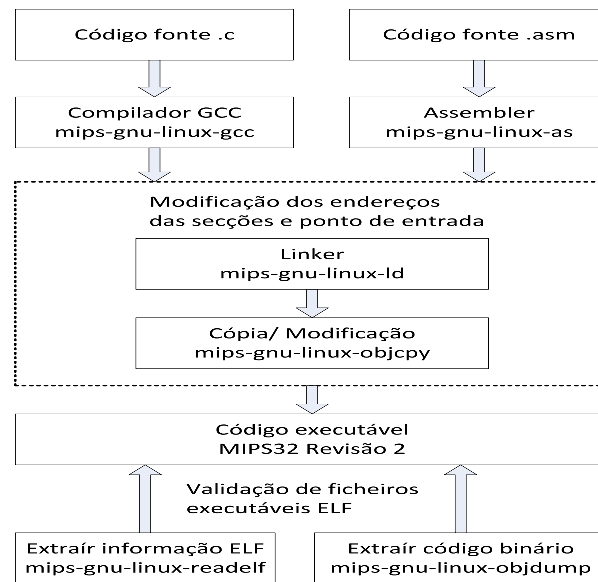


Figura 5.1: Desenvolvimento e compilação de programas para o SimuladorUE.

No anexo A pode-se observar o conteúdo do ficheiro Makefile, utilizado para compilar todas as demonstrações do presente trabalho.

Uma vez compilado o programa a ser testado, encontramo-nos prontos a executá-lo no simulador. Desta forma, para se iniciar a execução do simulador é necessário correr na consola um dos seguintes comandos:

```
./simuladorUE programa_ELF
ou
./simuladorUE programa_ELF memoria_ROM_ELF
```

Como se pode observar nos dois formatos acima, o simulador pode ser iniciado de duas formas: sem memória ROM e com memória ROM. No primeiro formato, apenas é passado como argumento o programa executável na norma ELF, sendo responsabilidade do código do programa a inicialização dos estados do processador, designadamente: os modos de execução, os estados de arranque, o mapeamento de páginas na TLB, estados de controlo, entre outros.

Por defeito, é possível parametrizar o estado inicial e os registos internos do processador no simulador, sendo os valores de inicialização utilizados sugeridos em [MT10c]. Neste caso, o endereço do PC é modificado com endereço do ponto de entrada definido no ficheiro executável ELF, sendo este considerado o ponto inicial de execução do código.

```
Program Console

Simulador MIPS32
Por David João Maia

Arquitectura:                                MIPS32 (Processador Genérico - 32bits)
Unidade de Gestão de Memória:                MMU para arquitectura de 32Bits
Mecanismo de Tradução de Endereços:          TLB - Translation Lookaside Buffer
Unidade de Memória Principal:                Implementação com Listas ligadas

A configurar canais de ligação utilzando 'Unix - Unnamed Pipes'...

                       Placa Board Mappings

| Dispositivos    | Endereço Inicio        | Endereço Fim

| 0x20000000      | 0xffffffff             | RAM Slot 2 (512Mb a 4Gb)
| 0x1fc00000      | 0x1ffffffc             | ROM (508Mb a 512Mb)
| 0x8c00000       | 0x1fbffffc             | Unused (140Mb a 508Mb)
| 0x8400000       | 0x8bffffc              | IO Devices (132Mb a 140Mb)
| 0x8000000       | 0x83ffffc              | Clock (128Mb a 132Mb)
| 0x00000000      | 0x7fffffc              | RAM Slot 1 (0 a 128Mb)


Canais de Comunicação

CPU: 10 9
RAM: 12 11
DISP: 8 7

Processador MIPS32 iniciado com o PID: (2463)
Memória R.A.M. iniciado com o PID: (2466)
Dispatcher iniciado com o PID: (2467)

Entry Point definido no ficheiro ELF: 0x80000000
```

Figura 5.2: Terminal VMM do simuladorUE.

O segundo caso, é utilizado exclusivamente em sistemas que simulem um funcionamento mais realista, nomeadamente os mecanismos de arranque utilizados num sistema real. Nestes casos o endereço do PC não é modificado pelo o valor do ponto de entrada no ficheiro binário ELF, sendo por sua vez actualizado através da inserção do sinal *Reset* no processador.

Assim que o simulador inicia a sua execução, é apresentado num terminal Linux um conjunto de informação relativamente à configuração do sistema, assim como os dispositivos emulados como se pode ver na figura 5.2. A componente gráfica do simulador consiste simplesmente num terminal Linux, de forma semelhante ao que acontece com o simulador QEMU, com a diferença que esta consiste numa visão directa do estado do monitor (VMM) e não num terminal TTY dentro do sistema virtualizado.

Inicialmente, ponderou-se a adopção de uma solução no qual fosse possível a utilização de duas consolas, uma para o VMM e ou outra através de um terminal TTY dentro do sistema virtualizado. Uma vez que o mecanismo de interrupções não se encontra implementado, apenas foi possível implementar um terminal com a informação proveniente do VMM.

No terminal é possível visualizar informações como a descrição dos componentes ligados

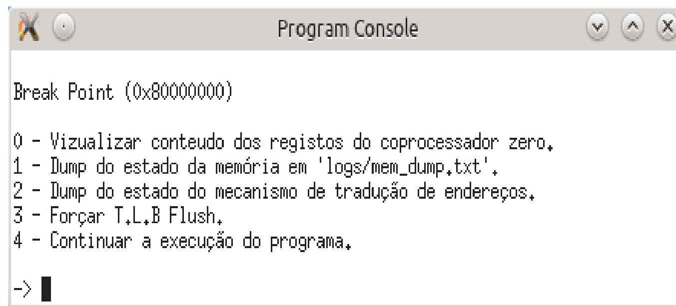


Figura 5.3: Funções do SimuladorUE.

no simulador ou o tipo de implementação utilizados. É igualmente possível validar algumas características mais técnicas como os mapeamentos definidos na placa mãe para associação dos vários dispositivos, como exemplo a memória RAM ou a memória ROM.

À medida que o simulador procede na sua execução, pode-se ver a definição dos vários canais, utilizados pelos componentes, assim como o protocolo de comunicação adoptado para comunicação. É igualmente possível visualizar que a execução dos vários componentes são lançados em processos individuais, no qual comunicarão exclusivamente com o módulo *Dispatcher*. Após a incialização dos componentes e respectivos módulos, é necessário proceder à introdução do programa binário na memória RAM do sistema virtualizado. Caso seja utilizada uma memória ROM o seu conteúdo será igualmente carregado para o dispositivo respectivo.

Em seguida, a consola do monitor (VMM) apresentará todas a instruções executadas no simulador em tempo de execução. Por defeito, o simulador possui um sistema de depuração interno, sendo possível mapear até dez endereços virtuais, no qual permitirá ao mesmo suspender a execução de instruções de forma a permitir ao utilizador a aplicação de um conjunto de funções. O objectivo destas funções consiste na modificação dos estados internos do sistema virtualizado assim como dos seus componentes.

Suponha-se a seguinte situação: a tabela de depuração interna no simulador possui apenas uma entrada com o endereço 0x80000000, desta forma, sempre que o PC do processador atingir o endereço mapeado, o simulador suspenderá a seu execução e apresentará a janela descrita na figura 5.3.

As funções disponíveis pelo simulador para modificar e recolher informação do sistema hóspede são:

- A primeira opção imprime no terminal os conteúdo dos registos internos e genéricos do processador do sistema hóspede;

- A segunda opção pega no conteúdo da memória RAM virtualizada e escreve num ficheiro de texto no sistema anfitrião:
- A terceira opção é muito semelhante à segunda uma vez que escreve o conteúdo da tabela de mapeamentos TLB interna no processador num ficheiro de texto no sistema anfitrião;
- A quarta opção permite ao utilizador executar uma *TLB Flush*, esvaziando assim a tabela de mapeamentos da TLB, forçando a uma mudança de contexto;
- A última opção é utilizada para retornar à execução das instruções no simulador.

5.2 Demonstrações

Em seguida serão abordados exemplos, através de programas binários no formato ELF, cujo objectivo consiste na demonstração das funcionalidades da arquitectura MIPS32 revisão 2 implementadas na secção 4.

Todos os exemplos apresentados utilizam programas gerados utilizando o `assembler mips-linux-gnu-as` directamente ou o compilador `mips-linux-gnu-gcc`.

5.2.1 Cálculo do factorial em modo e zona de *Kernel*

O primeiro exemplo tem como objectivo a demonstração do funcionamento de um programa, compilado directamente através do `mips-linux-gnu-as`, com as seguintes características:

- É responsabilidade do programa a configuração inicial do processador;
- Será utilizado apenas o modo de execução *kernel*;
- Serão utilizados endereços dentro do espaço de endereçamento *kernel*;
- Serão pré definidos no simulador mapeamentos na TLB para utilização da STACK no segmento KSEG3 para o modo *kernel*;
- O ponto de entrada do programa será o endereço 0x80000000;
- Será executado o programa com o factorial do número seis.

Uma vez que o primeiro exemplo não dispõe de memória ROM, é responsabilidade do programa configurar o processador com os estados adequados para a sua execução.

Apenas os endereços de memória dentro da zona *kernel*, nomeadamente KSEG0, KSEG1 e KSEG3 serão utilizados, no qual nos dois primeiros residirá o código binário do programa e no segmento KSEG3 será alocada uma página na TLB de forma a poder-se utilizar como STACK. O ponto de entrada do programa é no endereço 0x80000000, pertencente ao KSEG0 que é uma zona não mapeada e que não utilizada *cache*.

Uma vez que o modo de operação *kernel* não possui restrições de acesso, uma das metas da demonstração é permitir a execução do programa sem problemas, tanto a nível de permisssões como no funcionamento dos mecanismos. É importante referir que a função matemática factorial necessita de acesso à pilha STACK de forma a poder guardar os resultados e retornos das chamadas recursivas de forma estruturada.

Um requisito da utilização da STACK é a inicialização do registo $sp com um endereço para uma zona de memória que disponha de espaço livre. Para resolver este problema, será utilizado um mapeamento “*hardcoded*” no simulador, de uma página física na TLB, dentro do segmento KSEG3. No bloco de código seguinte pode-se ver os mapeamentos introduzidos na TLB.

```
//TLB - Mapeamentos
#define tlb_map_entryHI_0   0xE0000000 //Pagina Virtual + ASID + FLAGS
#define tlb_map_entryLO0_0  0x00000100 //Pagina Fisica Par
#define tlb_map_entryLO1_0  0x00000140 //Pagina Fisica Impar
#define tlb_map_mask_0      0x00001800 //Paginas de 4k
```

Listing 5.1: Mapeamentos na TLB para demonstração 1.

Sem o mapeamento, o programa não funcionará correctamente, uma vez que o endereço virtual apontado pelo $sp, será utilizado para guardar informação e caso aponte para uma zona de código ou uma zona não mapeada, levará o programa a um estado de erro.

Em sistemas operativos Linux, cada processo detém uma área reservada para STACK no seu espaço de endereçamento para ambos os modos, *user* e *kernel*, sendo ambas mapeadas na TLB. De forma a ultrapassar este problema, optou-se pelo mapeamento pré carregado na TLB do processador.

O comando seguinte exemplifica a forma como se iniciará o exemplo1.

```
./SimuladorUE exemplos/bin/exemplo1.bin
```

Na figura 5.4 pode-se ver a janela com a execução do programa exemplo1[1]. Após o arranque do programa, é possível visualizar em tempo real as instruções que vão sendo executadas no processador.

[1] O código fonte do programa pode ser consultado no Anexo B.

Program Console

```
Simulador MIPS32
Por David João Maia

Arquitectura:                           MIPS32 (Processador Genérico - 32bits)
Unidade de Gestão de Memória:           MMU para arquitectura de 32Bits
Mecanismo de Tradução de Endereços:     TLB - Translation Lookaside Buffer
Unidade de Memória Principal:           Implementação com Listas ligadas

A configurar canais de ligação utilzando 'Unix - Unnamed Pipes'...

                        Placa Board Mappings

| Dispositivos    | Endereço Inicio        | Endereço Fim

| 0x20000000      | 0xffffffff             | RAM Slot 2 (512Mb a 4Gb)
| 0x1fc00000      | 0x1ffffffc             | ROM (508Mb a 512Mb)
| 0x8c00000       | 0x1fbffffc             | Unused (140Mb a 508Mb)
| 0x8400000       | 0x8bffffc              | IO Devices (132Mb a 140Mb)
| 0x8000000       | 0x83ffffc              | Clock (128Mb a 132Mb)
| 0x00000000      | 0x7fffffc              | RAM Slot 1 (0 a 128Mb)


Canais de Comunicação

CPU: 10 9
RAM: 12 11
DISP: 8 7

Processador MIPS32 iniciado com o PID: (3671)
Memória R.A.M. iniciado com o PID: (3674)
Dispatcher iniciado com o PID: (3675)

Entry Point definido no ficheiro ELF: 0x80000000

---------------------- || ----------------------

(0)     Normal - 0x80000000 -     addi $a0, $0, 0x6
(1)     Normal - 0x80000004 -     lui $at, $0, 0x7fff0000
(2)     Normal - 0x80000008 -     ori $at, $at, 0xeffc
(3)     Normal - 0x8000000c -     add $sp, $0, $at
(4)     Normal - 0x80000010 -     jal 0x80000024
(5)     Delay  - 0x80000014 -     nop

(6)     Normal - 0x80000024 -     add $t0, $0, $a0
```

Figura 5.4: Execução do exemplo 1.

```
Estado do Processador:

$0:  0x0         0
$at: 0x7fffeffc  2147479548
$v0: 0x2d0       720
$v1: 0x0         0
$a0: 0x6         6
$a1: 0x0         0
$a2: 0x0         0
$a3: 0x0         0
$t0: 0x0         0
$t1: 0x1         1
$t2: 0x2d0       720
$t3: 0x6         6
$t4: 0x0         0
$t5: 0x0         0
$t6: 0x0         0
$t7: 0x0         0
$s0: 0x0         0
$s1: 0x0         0
$s2: 0x0         0
$s3: 0x0         0
$s4: 0x0         0
$s5: 0x0         0
$s6: 0x0         0
$s7: 0x0         0
$t8: 0x0         0
$t9: 0x0         0
$k0: 0x0         0
$k1: 0x0         0
$gp: 0x0         0
$sp: 0x7fffeffc  2147479548
$fp: 0x0         0
$ra: 0x80000018  -2147483624
HI:  0x0         0
LO:  0x0         0
```

Figura 5.5: Validação dos resultados.

Uma vez que este é um programa relativamente simples, o objectivo principal consiste em validar se o simulador foi capaz de executar o código completamente, confirmando o valor final no registo \$v0. Como se pode ver na figura 5.5, no argumento \$a0, o programa cálcula o factorial de seis, sendo o seu resultado 720.

Na figura 5.5 são apresentados os estados dos registos genéricos do processador, e é possível confirmar o valor do registo \$v0 com o resultado 720. O registo \$sp possui o valor actual da pilha, encontrando-se com o valor inicial, uma vez que todos os dados introduzidos foram removidos com a execução. Relativamente ao registo \$ra, este possuí o valor do endereço de retorno da última instrução JAL executada, como se pode ver na figura 5.5.

Desta forma, no primeiro exemplo, o simulador é capaz de executar código binário utilizando mecanismos complexos, como exemplo as funções recursivas do qual practicamente

todas as funções minimamente elaboradas necessitam. É igualmente possível a execução de código binário em modo *kernel*, alimentado apenas pelos bits *StatusKSU* e com acesso exclusivo a zonas de memória *kernel*.

5.2.2 Cálculo da sequência de Fibonacci em modo e zona de *User*

O segundo exemplo possui um objectivo semelhante ao utilizado na primeira demonstração, mas com uma ligeira modificação. A diferença consiste no facto de o programa correr inteiramente em segmentos de memória de *user* e em modo *user*.

Da mesma forma que o exemplo anterior, a demonstração foi compilada directamente através do `mips-linux-gnu-as` e possui as seguintes características:

- É responsabilidade do programa a configuração inicial do processador;
- O programa será executado em modo *user*;
- Serão utilizados exclusivamente endereços dentro do espaço de endereçamento *user* para execução do programa;
- Serão pré definidos no simulador mapeamentos na TLB para utilização da área de código, assim como da STACK para o modo *user*;
- O ponto de entrada do programa será o endereço 0x80000000;
- Será executado o programa com a sequência de Fibonacci do número quinze;

Como a presente demonstração não dispõe de uma memória ROM, é necessário inicializar o estado do processador para modo *user*, sendo obrigatório alterar os bits *StatusKSU* para 0b10, *StatusEXL* e *StatusBEV* para zero.

O programa arranca inicialmente para o endereço 0x80000000, no qual será executada uma rotina de inicialização do processador em modo *kernel*, uma vez que o *StatusERL* se encontra activo. Após a inicialização do processador, o registo *ErrorEPC* é modificado com o endereço do programa, nomeadamente 0x00400000, sendo em seguida executada a instrução ERET com o objectivo de alterar o PC, para em seguida passar para o modo *user*.

Será utilizado apenas o segmento de memória USEG para guardar as páginas com o código binário do programa Fibonacci, assim como os dados referentes à pilha STACK. Dado que todo o código arranca directamente para uma zona de memória mapeada, é necessário que o simulador possua inicialmente duas páginas alocadas na TLB, de

forma a permitir o carregamento e execução do programa. O bloco de código seguinte, apresenta os mapeamentos efectuados na TLB pelo o simuladorUE no seu arranque.

```
//TLB - Mappings
#define tlb_map_entryHI_0   0x7FFFE000 //Pagina Virtual + ASID + FLAGS
#define tlb_map_entryLO0_0  0x00000100 //Pagina Fisica Par
#define tlb_map_entryLO1_0  0x00000140 // Pagina Fisica Impar
#define tlb_map_mask_0      0x00001800 // Paginas de 4k

#define tlb_map_entryHI_1   0x00400000 // Pagina Virtual + ASID +
    FLAGS
#define tlb_map_entryLO0_1  0x00000080 // Pagina Fisica Par
#define tlb_map_entryLO1_1  0x000000C0 // Pagina Fisica Impar
#define tlb_map_mask_1      0x00001800 // Paginas de 4k
```

Listing 5.2: Mapeamentos na TLB para demonstração 2.

Assim como acontece com a função Factorial, a função Fibonacci requer a utilização da pilha igualmente de forma a poder guardar os resultados e retornos de chamadas recursivas de forma estruturada. Da mesma forma, sem os mapeamentos o programa não funcionará correctamente, uma vez que o endereço virtual apontado pelo $sp será utilizado para guardar informação. Caso este aponte para uma zona de código, zona não mapeada ou zona de memória sem permissões, levará o sistema a um estado de erro, podendo em alguns casos gerar um excepção do tipo *Address Error* ou *TLB Refill Exception*.

Para executar o programa do exemplo dois, basta correr o comando:

```
./SimuladorUE exemplos/bin/exemplo2.bin
```

Na figura 5.6, pode-se ver a execução do programa exemplo 2[2]. Após o arranque do programa, é possível observar as instruções que vão sendo executadas no processador em tempo real. Relativamente ao programa anterior, este é um pouco mais complexo uma vez que requer mais comparações e chamadas recursivas. Da mesma forma que a demonstração anterior, o objectivo final consiste em validar se o simulador foi capaz de executar o código completamente, confirmando o valor final no registo $v0.

Na figura 5.7, pode-se ver o valor de todos os registos genéricos, incluindo o registo $v0, no qual é possível confirmar que Fibonacci de 15 é igual a 610.

No registo $sp podemos ver o valor actual da pilha, encontrando-se com o valor inicial uma vez que todos os dados introduzidos foram removidos com a execução. Neste exemplo, o valor do $sp difere do valor da demonstração anterior uma vez que o segmento

[2]O código fonte do programa pode ser consultado no apêndice C.

```
Program Console

Simulador MIPS32
Por David João Maia

Arquitectura:                        MIPS32 (Processador Genérico - 32bits)
Unidade de Gestão de Memória:        MMU para arquitectura de 32Bits
Mecanismo de Tradução de Endereços:  TLB - Translation Lookaside Buffer
Unidade de Memória Principal:        Implementação com Listas ligadas

A configurar canais de ligação utilzando 'Unix - Unnamed Pipes'...

                        Placa Board Mappings

| Dispositivos   | Endereço Inicio      | Endereço Fim

| 0x20000000     | 0xffffffff           | RAM Slot 2 (512Mb a 4Gb)
| 0x1fc00000     | 0x1ffffffc           | ROM (508Mb a 512Mb)
| 0x8c00000      | 0x1fbffffc           | Unused (140Mb a 508Mb)
| 0x8400000      | 0x8bffffc            | IO Devices (132Mb a 140Mb)
| 0x8000000      | 0x83ffffc            | Clock (128Mb a 132Mb)
| 0x00000000     | 0x7fffffc            | RAM Slot 1 (0 a 128Mb)


Canais de Comunicação

CPU: 10 9
RAM: 12 11
DISP: 8 7

Memória R.A.M. iniciado com o PID: (3363)
Processador MIPS32 iniciado com o PID: (3360)
Dispatcher iniciado com o PID: (3364)

Entry Point definido no ficheiro ELF: 0x80000000

---------------------- || ----------------------

(0)     Normal - 0x80000000 -     mfc0 $t6, C0_Status, 0
(1)     Normal - 0x80000004 -     ori $t6, $t6, 0x10
(2)     Normal - 0x80000008 -     lui $at, $0, 0x400000
(3)     Normal - 0x8000000c -     ori $at, $at, 0x0
(4)     Normal - 0x80000010 -     mtc0 $at, , 0
(5)     Normal - 0x80000014 -     mtc0 $t6, C0_Status, 0
(6)     Normal - 0x80000018 -     eret

(7)     Normal - 0x400000 -       addi $a0, $0, 0x8
(8)     Normal - 0x400004 -       lui $at, $0, 0x7fff0000
```

Figura 5.6: Execução do exemplo2.

```
Estado do Processador:

$0:  0x0         0
$at: 0x7fffeffc  2147479548
$v0: 0x262       610
$v1: 0x0         0
$a0: 0x1         1
$a1: 0x0         0
$a2: 0x0         0
$a3: 0x0         0
$t0: 0x179       377
$t1: 0xe9        233
$t2: 0x0         0
$t3: 0x0         0
$t4: 0x0         0
$t5: 0x0         0
$t6: 0x900014    9437204
$t7: 0x0         0
$s0: 0x0         0
$s1: 0x0         0
$s2: 0x0         0
$s3: 0x0         0
$s4: 0x0         0
$s5: 0x0         0
$s6: 0x0         0
$s7: 0x0         0
$t8: 0x0         0
$t9: 0x0         0
$k0: 0x0         0
$k1: 0x0         0
$gp: 0x0         0
$sp: 0x7fffeffc  2147479548
$fp: 0x0         0
$ra: 0x400014    4194324
HI:  0x0         0
LO:  0x0         0
```

Figura 5.7: Registos genéricos.

```
Registos do Coprocessador 0

Registo: 0  (Index)             - 0x0
Registo: 1  (Random)            - 0x3f
Registo: 2  (EntryLo0)          - 0x0
Registo: 3  (EntryLo1)          - 0x0
Registo: 4  (Context)           - 0x0
Registo: 4  (ContextConfig)     - 0x3ffff8
Registo: 4  (UserLocal)         - 0x0
Registo: 5  (Pagemask)          - 0x1800
Registo: 6  (Wired)             - 0x4
Registo: 7  (HWREna)            - 0x0
Registo: 8  (BadVAddr)          - 0x0
Registo: 9  (Count)             - 0x0
Registo: 10 (EntryHi)           - 0x0
Registo: 11 (Compare)           - 0x0
Registo: 12 (Status)            - 0x900010
         StatusCU:              - 0x0
         StatusBEV:             - 0x0
         StatusTS:              - 0x0
         StatusKSU:             - 0x2 (User Mode)
         StatusERL:             - 0x0
         StatusEXL:             - 0x0
         StatusIE:              - 0x0

Registo: 12 (IntCtl)            - 0x0
Registo: 12 (SRSCtl)            - 0x0
Registo: 12 (SRSMap)            - 0x0
Registo: 13 (Cause)             - 0x0
         CauseCE:               - 0x0
         CauseExcCode:          - 0x0

Registo: 14 (EPC)               - 0x0
Registo: 15 (PRId)              - 0x0
Registo: 15 (EBase)             - 0x80000000
         EBase_31_30:           - 0x2

Registo: 16 (Config)            - 0x80000408
Registo: 16 (Config1)           - 0x1
Registo: 25 (PerfCnt)           - 0x0
Registo: 30 (ErrorPC)           - 0x400000
```

Figura 5.8: Registos internos do coprocessador zero.

mapeado KSEG3 foi modificado para o segmento USEG, respectivamente 0xE0000100 para 0x7FFFEFFC.

Como se pode ver no segundo exemplo, o simulador permite a configuração e execução de código binário, no qual é possível a execução de programas relativamente longos, uma vez que são necessárias aproximadamente trinta mil instruções para correr o Fibonacci(15). Na sua execução, a função Fibonacci consome exclusivamente recursos em modo de *user*, nomeadamente no segmento de memória USEG.

Na figura 5.8, pode-se verificar o valor dos registos internos do coprocessador zero, no qual assentam os modos de execução e estados do processador.

5.2.3 Casos de erro e mecanismos MIPS

Em oposição aos dois exemplos anteriores, a presente demonstração tem como objectivo a geração de vários erros, forçando o programa em execução a proceder á correcção dos respectivos erros através dos mecanismos definidos na arquitectura MIPS32.

O programa da presente demonstração foi compilado directamente através do compilador `mips-linux-gnu-as` e possui as seguintes características:

- É responsabilidade do programa a configuração inicial do processador;
- O programa será executado utilizando ambos os modos *user* e *kernel*;
- Serão utilizados endereços dentro do espaço de endereçamento de *user* e *kernel*;
- Serão utilizados pré mapeamentos na TLB através do simulador para as áreas STACK e USER_CODE, sendo adicionados novos mapeamentos na execução do programa;
- O ponto de entrada do programa será o endereço 0x80001000;
- Serão geradas várias situações de erro no qual a arquitectura deverá ser capaz de disponibilizar mecanismos para tratamento.

A demonstração funcionará de forma diferente das anteriores, sendo gerado um erro de cada vez, demonstrando assim um exemplo do seu funcionamento. Através dos mecanismos fornecidos pela arquitectura MIPS32 serão efectuados para cada caso a sua correcção.

O programa iniciará a sua execução no endereço 0x80001000, sendo efectuado em primeiro lugar a configuração do processador, começando em seguida o processo de geração

```
Program Console
(29)    Normal - 0x800010ac -    mtc0 $s3, C0_PageMask, 0
(30)    Normal - 0x800010b0 -    mtc0 $s4, C0_Index, 0
(31)    Normal - 0x800010b4 -    tlbwi

********* Excepção *********

Registos do Coprocessador 0 (Momento da excepção):

StatusCU:  0x0
StatusBEV: 0 (Bootstap)
StatusTS:  1
StatusERL: 0 (Normal Level)
StatusEXL: 0 (Normal Level)
StatusKSU: 0 (Kernel Mode)

Delay Branch desligado

Tipo de excepção:              MCheck
BadVirtualAddress:             0x0

Revisão da arquitectura: 2

Registos do Coprocessador 0 (Após cálculo de vectores):

StatusBEV: 0 (Bootstap)
StatusERL: 0 (Normal Level)
StatusEXL: 1 (Exception Level)
StatusKSU: 0 (Kernel Mode)

EPC:                    0x800010b4
Error EPC:              0x0
Endereço do Tratador:   0x80000180

****************************
```

Figura 5.9: Excepção Machine Check.

de estados de erros. A configuração do processador consistirá simplesmente no desligar dos bits que possam influenciar os modos de operação ou o mecanismo de tradução de endereços, nomeadamente: *StatusKSU*, *StatusEXL*, *StatusERL* e *StatusBEV*. A configuração dos bits encontra-se no bloco de código no anexo D na etiqueta "Config" (linha 24).

O primeiro erro será a geração da excepção *Machine Check* em modo *kernel*. Como vimos na secção 4.1.8, a excepção *Machine Check* ocorre sempre que existam inconsistências no processador ou na TLB. De forma a possibilitar a demonstração deste comportamento, serão introduzidos dois mapeamentos iguais na TLB, forçando a ocorrência da referida excepção.

No anexo D pode-se ver o bloco de código referente aos mapeamentos introduzidos na TLB na etiqueta "MCheck" (linha 67) e consequente geração do estado de erro. Na figura 5.9 pode-se ver os registos do coprocessador central, no qual é possível confirmar que o *StatusTS* se encontra activo no momento da excepção, quando executada a segunda instrução TBLWI.

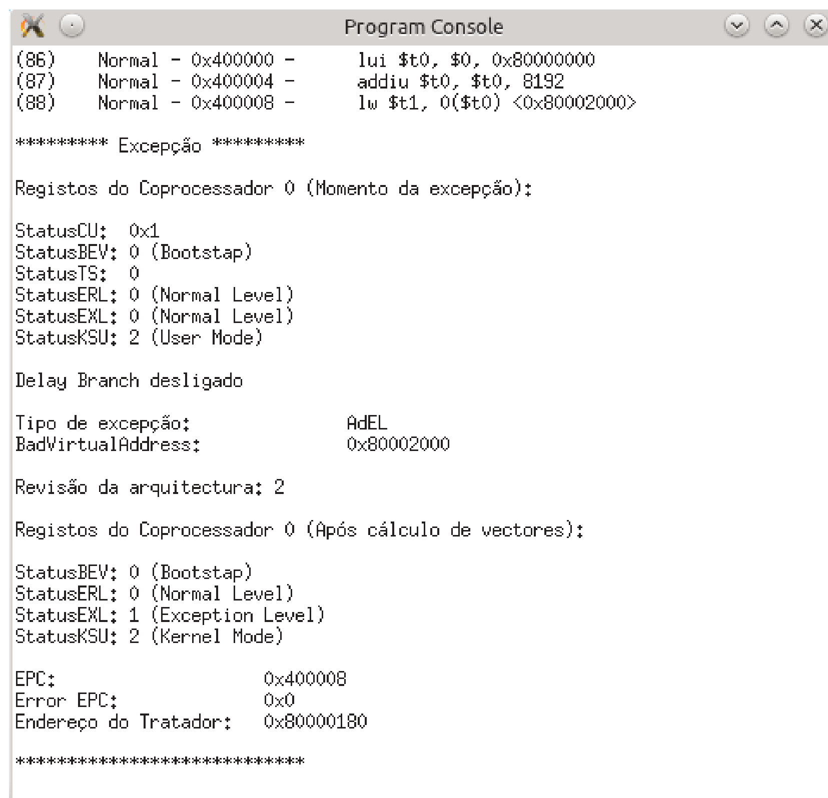


Figura 5.10: Excepção Address Error.

Como se pôde observar em 4.3.2, na execução de instruções de escrita na TLB, nomeadamente em TLBWI ou TLBWR, é efectuado a validação da mesma, sendo gerada a excepção *Machine Check* caso exista uma inconsistência. É de notar que na geração da excepção o PC é modificado para o endereço *EBase* + 0x180, perfazendo o endereço 0x80000180.

No anexo D, pode-se observar igualmente o código de tratamento do programa no rótulo "handle_mcheck" (linha 177), sendo efectuado um simples *flush* do primeiro mapeamento como correcção da excepção. Será igualmente efectuado a passagem de 1 para 0 do bit *StatusTS*.

Chama-se à atenção que a permissão na resolução das inconsistências por *software* na TLB é dependente da implementação. Para a revisão dois da arquitectura MIPS32, a detecção e tratamento, apenas pode ocorrer em instruções de escrita, designadamente TLBWI ou TLBWR. Os próximos exemplos serão executados em modo *user*, uma vez que dizem respeito a mapeamentos de páginas, assim como o acesso sem permissões a determinados recursos do sistema. Assim, o código dos próximos exemplos será executado

```
Program Console
(119)   Normal - 0x400020 -      mtc0 $s4, C0_Index, 0
(120)   Normal - 0x400024 -      tlbwi
(121)   Normal - 0x400028 -      lw $t1, -32($sp)       <0x7fffefdc>

********* Excepção *********

Registos do Coprocessador 0 (Momento da excepção):

StatusCU:  0x1
StatusBEV: 0 (Bootstap)
StatusTS:  0
StatusERL: 0 (Normal Level)
StatusEXL: 0 (Normal Level)
StatusKSU: 2 (User Mode)

Delay Branch desligado

Tipo de excepção:               TLBL
BadVirtualAddress:              0x7fffefdc

Revisão da arquitectura: 2

Registos do Coprocessador 0 (Após cálculo de vectores):

StatusBEV: 0 (Bootstap)
StatusERL: 0 (Normal Level)
StatusEXL: 1 (Exception Level)
StatusKSU: 2 (Kernel Mode)

EPC:                    0x400028
Error EPC:              0x0
Endereço do Tratador:   0x80000000

****************************
```

Figura 5.11: Excepção TLBL (TLB Refill).

em espaço de *user* começando no endereço 0x00400000. O bloco de código responsável pela mudança de modo de execução pode ser consultado no anexo D na etiqueta "Switch" (linha 41).

Como referido anteriormente, sempre que seja efectuado um pedido de LOAD ou STORE num determinado endereço sem permissões de acesso, a MMU é responsável por gerar uma excepção do tipo *Address Error* indicando o acesso não autorizado. Assim, de forma a demonstrar esta situação tentar-se-á aceder ao endereço 0x80002000 (.data) através do modo de execução *user*.

Como indicado, este é um modo de execução menos privilegiado, sem autorização de acesso ao espaço de endereçamento *kernel*. Na tentativa de acesso será gerada uma excepção do tipo *AdEL*, sendo o PC modificado para o endereço *EBASE* + 0x180, perfazendo 0x80000180 como se pode observar na figura 5.10. O tratador utilizado para resolver este caso irá simplesmente somar quatro bytes ao valor de retorno do EPC, de forma a fazer o *bypass* da instrução geradora do erro.

Chama-se á atenção que o código utilizado para resolver o problema de acesso indevido

```
Program Console
(144)   Normal - 0x400040 -
********* Excepção *********

Registos do Coprocessador 0 (Momento da excepção):

StatusCU:  0x0
StatusBEV: 0 (Bootstap)
StatusTS:  0
StatusERL: 0 (Normal Level)
StatusEXL: 0 (Normal Level)
StatusKSU: 2 (User Mode)

Delay Branch desligado

Tipo de excepção:                CpU
BadVirtualAddress:               0x7fffefdc

Revisão da arquitectura: 2

Registos do Coprocessador 0 (Após cálculo de vectores):

StatusBEV: 0 (Bootstap)
StatusERL: 0 (Normal Level)
StatusEXL: 1 (Exception Level)
StatusKSU: 2 (Kernel Mode)

EPC:                     0x400040
Error EPC:               0x0
Endereço do Tratador:    0x80000180

*****************************

        mfc0 $t7, C0_Status, 0
```

Figura 5.12: Excepção Coprocessor Unusable.

é exclusivo do sistema operativo, sendo neste caso utilizado para demonstrar a sua existência e funcionamento. Na etiqueta "handle_adel" (linha 152) no anexo D, pode-se observar o *bypass* da excepção no código do tratador.

Uma das excepções que mais vezes ocorre num sistema operativo é sem dúvida a excepção *TLB Refill*, dado que esta é invocada várias vezes em cada mudança de contexto na TLB. De forma a exemplificar o seu comportamento serão efectuados acessos a dados existentes na STACK, cujo mapeamento será removido préviamente da TLB.

Na figura 5.11 pode-se observar a geração da excepção *TLB Refill*, no qual se pode ver o endereço virtual da instrução geradora da excepção, assim como o endereço virtual que o processador não conseguiu aceder (*BadVAddr*).

O código binário do tratamento da excepção pode ser consultado no anexo D em ".section .TLBrefill" (linha 245), consistindo no remapeamento da STACK na TLB. Em oposição das excepções anteriores, a *TLB Refill* possui um ponto de entrada diferente caso o *StatusEXL* se encontre desligado, nomeadamente o endereço *EBASE* + 0x00, o que perfaz o endereço virtual 0x80000000.

```
Program Console
(176)   Normal - 0x400044 -      syscall (User Mode)

********* Excepção *********

Registos do Coprocessador 0 (Momento da excepção):

StatusCU:  0x0
StatusBEV: 0 (Bootstap)
StatusTS:  0
StatusERL: 0 (Normal Level)
StatusEXL: 0 (Normal Level)
StatusKSU: 2 (User Mode)

Delay Branch desligado

Tipo de excepção:              Sys
BadVirtualAddress:             0x7fffefdc

Revisão da arquitectura: 2

Registos do Coprocessador 0 (Após cálculo de vectores):

StatusBEV: 0 (Bootstap)
StatusERL: 0 (Normal Level)
StatusEXL: 1 (Exception Level)
StatusKSU: 2 (Kernel Mode)

EPC:                   0x400044
Error EPC:             0x0
Endereço do Tratador:  0x80000180

****************************
```

Figura 5.13: Excepção System Call.

A demonstração da TLB Refill não tem por objectivo a programação de algoritmos complexos de gestão de memória, mas sim a exemplificação do funcionamento do mecanismo. Sempre que este tipo de situação ocorre o sistema operativo é responsável por introduzir um mapeamento novo na TLB.

Uma vez que o programa se encontra a correr em modo de execução *user* com os bits $StatusCU_{3..0}$ ligados, é possível aceder a determinadas funcionalidades no coprocessador central. De forma a demonstrar a restrição no acesso às referidas funcionalidades é necessário desligar estes bits. Após desactivar os bits *StatusCU*, tentar-se-á a modificação do modo de execução de *user* para *kernel*, de forma a obter-se novamente acesso privilegiado a todas as funcionalidades no processador.

No anexo D na etiqueta “cpuerror” (linha 224) pode-se observar a tentativa de modificação dos bits *StatusKSU* para 0b00. Tal modificação não pode ocorrer a partir do modo de execução *user* dado que seria uma enorme falha de segurança para o sistema. Na figura 5.12 pode-se observar a geração da excepção na execução do programa no simulador.

De forma semelhante que excepção *Address Error* é responsabilidade do sistema opera-

tivo decidir o que fazer com o programa gerador da excepção. Mais uma vez, neste caso será feito novamente o *bypass* das instruções problemáticas como se pode ver na etiqueta “handle_cpu” (linha 169) no anexo D.

Por último têm-se a execução do mecanismo mais conhecido na área dos sistemas operativos, nomeadamente as chamadas ao sistema *Syscall*. O seu funcionamento é relativamente simples, uma vez que a execução da instrução SYSCALL invoca directamente uma excepção no processador, sendo o PC modificado para o endereço $EBASE$ + 0x180 que perfaz 0x80000180.

Na figura 5.13 pode-se observar a geração da excepção na execução do programa no simulador. De forma a terminar a execução do programa é executada uma SYSCALL, através do modo *user* sem permissões nos bits $StatusCU$, que irá simplesmente escrever o valor cinco no registo \$v0. O código binário do tratamento da excepção pode ser consultado na etiqueta “handle_syscall” (linha 160) no anexo D.

Como se pode observar no código do tratador “handle_syscall”, a instrução SYSCALL possui apenas uma função, designadamente escrever o valor cinco no registo \$v0, contrariamente ao que acontece em sistemas operativos reais, como o Linux, no qual possuem nesta localização, uma tabela de mapeamentos para os tratadores das chamadas ao sistema implementadas. Chama-se no entanto à atenção, para a necessidade de modificação do registo EPC na excpção SYSCALL, uma vez que o endereço de retorno aponta novamente para a mesma instrução. Assim, através de *software* o tratador da excepção deve ter este comportamento em consideração, somando quatro bytes ao valor de retorno.

5.2.4 Cálculo de números primos em C (GCC)

No presente exemplo têm-se como objectivo a demonstração da possibilidade de execução de código binário gerado através de um compilador externo ao simulador. Desta forma, o programa utilizado na demonstração, foi compilado directamente através do compilador `mips-linux-gnu-gcc` e possui as seguintes características:

- É responsabilidade do simulador a configuração inicial do processador;
- O programa será executado em modo *user*;
- Serão utilizados endereços dentro do espaço de endereçamento de *user*;
- Serão utilizados pré mapeamentos na TLB através do simulador;
- O ponto de entrada do programa dependerá do compilador GCC, aproximadamente 0x0040xxxx;

- Será feita a validação se quatro números inteiros são primos.

A configuração do processador será feita directamente através do simulador uma vez que o programa compilado através do GCC não tem em conta o estado do processador. Desta forma, os bits mais relevantes na configuração do processador serão modificados préviamente no simulador, nomeadamente: *StatusKSU*, *StatusEXL*, *StatusERL* e *StatusBEV* para 0b10, 0, 0 e 0 respectivamente.

Em oposição aos exemplos anteriores, o processo de compilação difere, sendo utilizado o compilador para a linguagem C `mips-linux-gnu-gcc` e o linker `mips-linux-gnu-ld`. No bloco de código seguinte pode-se observar a forma como o exemplo quatro foi compilado.

```
#Compilador e Linker
LK = mips-linux-gnu-ld
GCC = mips-linux-gnu-gcc

BIN = bin
SRC = src
AUX = aux

exemplo4:
	# Compilar sem linkar
	$(GCC) -EL -mips32r2 -mabi=32 -c $(SRC)/primos.c -o
	$(AUX)/exemplo4.o

	# Linkagem sem entry point
	$(LK) -EL -N $(AUX)/exemplo4.o -o $(BIN)/exemplo4.bin
```

Listing 5.3: Sequência de compilação para a demonstração 4. (Makefile)

Inicialmente ponderou-se a geração de código sem modificações, mas uma vez que o compilador geraria um enorme conjunto de informação desnecessária para a demonstração, mas necessária para o *loader* do sistema operativo, esta opção teve de ser abandonada.

Optou-se assim por uma abordagem mais simples, nomeadamente na compilação sem linkagem. Isto é, o compilador GCC, através da opção -c [3], apenas compilará o código C para código máquina, sendo gerado um ficheiro do tipo “Relocatable”. Em seguida é necessário efectuar a linkagem do ficheiro gerado, criando um ficheiro executável sem informações desnecessárias, nomeadamente preparações ou referências para bibliotecas dinâmicas.

[3] No processo de compilação do código fonte, apenas é feito o assemble do código sem se efectuar a linkagem.

```
Estado do Processador:

$0:  0x0        0
$at: 0x0        0
$v0: 0x4        4
$v1: 0x10       16
$a0: 0xf        15
$a1: 0x0        0
$a2: 0x0        0
$a3: 0x0        0
$t0: 0x4        4
$t1: 0x6        6
$t2: 0x2        2
$t3: 0x7        7
$t4: 0x2        2
$t5: 0xb        11
$t6: 0x4        4
$t7: 0xf        15
$s0: 0x0        0
$s1: 0x0        0
$s2: 0x0        0
$s3: 0x0        0
$s4: 0x0        0
$s5: 0x0        0
$s6: 0x0        0
$s7: 0x0        0
$t8: 0x0        0
$t9: 0x0        0
$k0: 0x0        0
$k1: 0x0        0
$gp: 0x0        0
$sp: 0x7fffefd4 2147479508
$fp: 0x7fffefd4 2147479508
$ra: 0x4000e4   4194532
HI:  0x0        0
LO:  0x1        1
```

Figura 5.14: Registos genéricos.

O ponto de entrada utilizado para a execução do programa é deixado inalterado, sendo o mesmo gerado pelo GCC. Como objectivo, o programa irá validar se os números 6, 7, 11 e 15 são números primos, sendo o seu resultado guardado no registo genérico $v0 para cada chamada da função. O programa não foi pensado para ser complexo mas sim possível de correr dentro do simulador. Na figura 5.15 pode-se observar a execução do programa no simulador[4].

Na figura 5.14 pode-se ver o valor final dos registos genéricos do processador. De forma a facilitar a interpretação da execução do código, foram introduzidas instruções `assembly` directamente no código C, como se pode ver no anexo E (linha 5).

Estas instruções tem como objectivo a intercepção do resultado, assim como do argumento utilizado, de forma a introduzir num registo temporário para apresentação como

[4] O código fonte do programa pode ser consultado no Anexo E.

```
Program Console

Processador MIPS32 iniciado com o PID: (2121)
Memória R.A.M. iniciado com o PID: (2124)
Dispatcher iniciado com o PID: (2125)

Entry Point definido no ficheiro ELF: 0x400090

---------------------- || ----------------------

(0)     Normal - 0x400090 -     addiu $sp, $sp, -40
(1)     Normal - 0x400094 -     sw $ra, 36($sp)         <0x7fffeff8>
(2)     Normal - 0x400098 -     sw $fp/$s8, 32($sp)     <0x7fffeff4>
(3)     Normal - 0x40009c -     addu $fp/$s8, $sp, $0   <move $fp/$s8, $sp>
(4)     Normal - 0x4000a0 -     addiu $a0, $0, 6
(5)     Normal - 0x4000a4 -     jal 0x4000fc
(6)     Delay  - 0x4000a8 -     nop

(7)     Normal - 0x4000fc -     addiu $sp, $sp, -24
(8)     Normal - 0x400100 -     sw $fp/$s8, 20($sp)     <0x7fffefd0>
(9)     Normal - 0x400104 -     addu $fp/$s8, $sp, $0   <move $fp/$s8, $sp>
(10)    Normal - 0x400108 -     sw $a0, 24($fp/$s8)     <0x7fffefd4>
(11)    Normal - 0x40010c -     sw $0, 8($fp/$s8)       <0x7fffefc4>
(12)    Normal - 0x400110 -     sw $0, 12($fp/$s8)      <0x7fffefc8>
(13)    Normal - 0x400114 -     addiu $v0, $0, 1
(14)    Normal - 0x400118 -     sw $v0, 12($fp/$s8)     <0x7fffefc8>
(15)    Normal - 0x40011c -     j 0x400158
(16)    Delay  - 0x400120 -     nop

(17)    Normal - 0x400158 -     lw $v1, 12($fp/$s8)     <0x7fffefc8>
(18)    Normal - 0x40015c -     lw $v0, 24($fp/$s8)     <0x7fffefd4>
(19)    Normal - 0x400160 -     slt $v0, $v0, $v1
(20)    Normal - 0x400164 -     beqz $0, 0x400124
(21)    Delay  - 0x400168 -     nop

(22)    Normal - 0x400124 -     lw $v1, 24($fp/$s8)     <0x7fffefd4>
(23)    Normal - 0x400128 -     lw $v0, 12($fp/$s8)     <0x7fffefc8>
(24)    Normal - 0x40012c -     div $v1, $v0
(25)    Normal - 0x400130 -     teq $v0, $0
(26)    Normal - 0x400134 -     mfhi $v0, hi
(27)    Normal - 0x400138 -     bnez $0, 0x10
(28)    Delay  - 0x40013c -     nop

(29)    Normal - 0x400140 -     lw $v0, 8($fp/$s8)      <0x7fffefc4>
```

Figura 5.15: Execução do exemplo 4.

se pode ver na figura 5.14. Desta forma, os resultados das quatro chamadas da função "primos" serão guardados dois a dois, nomeadamente o valor de retorno no registo par e o argumento no registo impar, começando no registo $t0.

Os valores de retorno iguais a dois indicam que o argumento é um número primo uma vez que só possui dois divisores, nomeadmento o número um e ele próprio. Nos casos em que o valor seja superior a dois, indica que o argumento não é um número primo, sendo o valor retornado o número de divisores.

Como se pôde observar com o presente exemplo, é possível compilar código na linguagem C, através do compilador GCC com suporte para compilação cruzada, para código binário sendo possível a sua execução no simulador, existindo no entanto, ainda alguns problemas na integração de ambos.

Capítulo 6

Conclusões

Nesta tese desenvolveu-se um simulador da arquitectura MIPS32 revisão 2 com capacidade de execução em modo user e kernel, incluindo os mecanismos de memória virtual e tradução de endereços, excepções e interrupções. A abordagem seguida teve em vista a flexibilidade na análise da execução de código em oposição ao desempenho.

O simulador permite ler um ficheiro executável no formato ELF, gerado por um compilador, e executá-lo num ambiente virtual controlado.

As secções seguintes apresentam em maior detalhe os objectivos alcançados (secção 6.1), limitações (secção 6.2), análise comparativa com outras soluções existentes (secção 6.3) e finalmente tópicos em aberto para trabalho futuro (6.4).

6.1 Objectivos alcançados

Um dos objectivos colocados é virtualização da arquitectura MIPS32 revisão 2. Para o efeito é necessário cumprir um conjunto de requisitos mínimos impostos pela arquitectura. Os requisitos referidos foram implementados com sucesso e enumeram-se em seguida:

ISA para a arquitectura MIPS32 revisão 2
O primeiro requisito consiste na implementação do conjunto de instruções existentes na arquitectura MIPS32 revisão 2, nomeadamente as instruções aritméticas, de

controlo do mecanismo de tradução de endereços, saltos condicionais e incondicionais, assim como do controlo do fluxo de execução do processador.

Simulação dos pipelines
Um aspecto muito importante é a simulação do comportamento da execução das instruções dentro dos *pipelines*.
Este comportamento é recriado através do estado *delay_slot* dentro da máquina de estados central no processador.

Registos genéricos
Um dos pré-requisitos na implementação da ISA é o conjunto de registos genéricos, nos quais serão depositadas a informação proveniente da memória ou gerada pelas instruções descritas na ISA.

Registos internos do coprocessador de controlo
De forma semelhante aos registos genéricos, os registos internos do coprocessador de controlo, consistem num pré-requisito não só para um subconjunto da ISA, mas igualmente para o PRA (*Privileged Resource Architecture*).

Modos de execução
O mecanismo responsável pela introdução das funcionalidades de segurança, nomeadamente a restrição de acessos a registos internos, assim como a zonas de memória, são os modos de execução do processador, designadamente: modo *kernel* e *user*.

Unidade de gestão de memória
A MMU é o mecanismo responsável pela validação e tradução dos endereços pedidos pelo processador no acesso à memória.
Inerente ao seu funcionamento encontra-se os mecanismos de tradução de endereços, FMT e TLB nos quais os mapeamentos para a memória são estáticos e dinâmicos respectivamente.

Uma vez que o simulador foi concebido para utilizar exclusivamente a TLB, o sistema virtualizado dispõe de mecanismos de paginação, com tamanhos iguais ou superiores a 4K.

Mecanismo de excepções
Este é sem sombra de dúvidas o mecanismo mais importante no processador, no qual assentam vários mecanismos como: *syscalls*, controlo do fluxo de execução, interrupções, modos de execução ou *Shadow Set's*.

Existem, no entanto, dois mecanismos considerados obrigatórios pela revisão 2 que não foram implementados totalmente, nomeadamente o mecanismo de interrupções e o con-

junto *Shadow Set's*. Estas limitações serão detalhadamente abordadas na secção 6.2 deste capítulo.

Adicionalmente aos requisitos da arquitectura MIPS32, foi necessário emular quatro tipos de componentes físicos de forma a possibilitar o funcionamento correcto de um sistema simples no simulador, nomeadamente: a placa mãe, processador genérico MIPS, memória RAM e memória ROM.

O simulador permite a execução de ficheiros binários executáveis na norma ELF como argumentos, tornando possível a execução de código binário gerado por qualquer compilador.

6.2 Limitações

Embora muitas das metas delineadas para o simulador tenham sido cumpridas, ficaram igualmente objectivos por cumprir. Desta forma, determinados aspectos da arquitectura MIPS32 Revisão 2 do simulador possuem limitações nos seus mecanismos, tal como na sua execução e na sua implementação.

Um exemplo deste comportamento, provavelmente o mais importante, é o facto de a implementação do mecanismo de interrupções não ter ficado completa. Uma vez que este mecanismo é responsável pela comunicação entre o processador e os vários periféricos ligados no sistema, o seu impacto repercute-se no simulador e na própria interacção entre o utilizador e o sistema virtualizado.

Associado à especificação da arquitectura MIPS32, encontra-se o mecanismo de *hardware* conhecido como *Shadow Set's*, no qual, para a revisão dois, é introduzida uma grande optimização e diferenciação na forma como são salvaguardados e repostos os registos genéricos, sempre que ocorrem excepções ou interrupções no processador.

Tanto o mecanismo de interrupções como os *Shadow Set's* assentam sobre mecanismos prioritários, como o mecanismo de tradução de endereços ou o mecanismo de excepções. Por este motivo foram forçados a ser implementados em último lugar, ficando a sua implementação incompleta.

Por outro lado, para a especificação da revisão dois, existem outros aspectos, como o coprocessador de vírgula flutuante ou a memória *cache*, os quais são funcionalidades desejáveis em qualquer sistema real, mas sendo opcionais em contexto de simulação. Desta forma, ambos os mecanismos foram excluidos dos objectivos no processo de implementação do simulador.

Ainda dentro das características da arquitectura MIPS32 revisão dois, no mecanismo de

tradução de endereços existe a possibilidade de utilização de páginas físicas e virtuais com tamanho igual a 1K (1024 bytes). O simulador foi concebido para correr um sistema operativo Linux, no qual a tendência é utilizar páginas com tamanho igual ou superior 4K, sendo as páginas de tamanho 1K utilizadas em sistemas menos complexos, como os microcontroladores. Desta forma, optou-se pelo não suporte à paginação com páginas de tamanho 1K.

Uma limitação do sistema prende-se no facto de a representação do mecanismo da TLB se encontrar implementada apenas para o modelo composto por um `array` simples, dado que a arquitectura MIPS32 dispõe de configurações mais complexas da mesma, como `array's` duplos ou com extensões.

Relativamente à composição do simulador, a sua maior limitação consiste no conjunto de periféricos emulados, uma vez que o presente simulador apenas virtualiza quatro tipos de componentes, mormente: placa mãe[1], processador MIPS genérico, memória RAM e ROM. Devido ao limitado conjunto de componentes emulados, o simulador não consegue correr sistemas muito complexos, uma vez que não dispõe de memórias permanentes (disco rígido) nos quais possa carregar informação, como bibliotecas de sistema ou um sistema operativo completamente funcional.

6.3 Comparação com trabalhos relacionados

Uma das aplicações do simulador é a execução de um sistema operativo como o Linux, à semelhança do que acontece com emuladores mais avançados como o QEMU[2] ou o GXemul[3]. Existem no entanto outros tipos de simuladores, que simulam igualmente a arquitectura MIPS32, mas com um objectivo pedagógico, nomeadamente o SPIM[4] (*A MIPS32 Simulator*) ou o MARS[5] (*Mips Assembly and Runtime Simulator*).

Estes dois últimos simuladores, SPIM e MARS, apresentam um ambiente de desenvolvimento com possibilidade de execução e depuração de código `assembly` MIPS.

Nestes simuladores, a simulação da arquitectura é feita através da interpretação do código fonte, sendo em seguida simulada a sua execução. Por este motivo, em ambos os simuladores não é possível a execução de código binário gerado por outro tipo de compiladores, que não o do próprio simulador, como por exemplo o compilador GCC. É também importante referir que a arquitectura MIPS32 não se encontra totalmente

[1] A placa mãe utilizada inspirou-se na *motherboard* Malta MIPS.
[2] Site oficial do simulador QEMU: `http://wiki.qemu.org/Main_Page`
[3] Site oficial do simulador GXemul: `http://gxemul.sourceforge.net/`
[4] Site oficial do simulador SPIM:`http://pages.cs.wisc.edu/~larus/spim.html`
[5] Site oficial do simulador MARS: `http://courses.missouristate.edu/kenvollmar/mars/`

simulada, dado que alguns mecanismos como os modos de execução, excepções ou registos internos não se encontram implementados.

O presente trabalho possui em comum alguns aspectos com os simuladores acima referidos, nomeadamente a possibilidade de execução instrução a instrução.

Embora existam algumas parecenças com os emuladores anteriores, o objectivo do presente trabalho consiste em subir um pouco mais a "fasquia", de forma a permitir a emulação completa de *hardware* com o objectivo de executar programas ou sistemas operativos desenhados para a arquitectura MIPS32, independentemente do tipo de compilador utilizado.

Simuladores mais poderosos como o QEMU ou GXemul, simulam não uma mas várias arquitecturas, necessitando assim de um nível de portabilidade superior.

A forma que ambos utilizam para satisfazer esta necessidade, consiste na utilização de técnicas como a *Dynamic Binary Translation*, nas quais é possível a criação de uma representação intermédia de instruções, conhecida como *Micro code* ou *Micro Operations*. À semelhança do que acontece com a linguagem de programação Java, a utilização de *Micro code* é uma poderosa funcionalidade e vantagem na portabilidade do sistema, permitindo assim o mapeamento das instruções do *Micro code* em funções específicas, dependendo da arquitectura do sistema anfitrião.

Uma vez que o presente trabalho apenas simula uma arquitectura em específico, MIPS32, a utilização de *Micro code* produziria desnecessáriamente um maior *overhead* na geração do código no sistema anfitrião.

Dado que a emulação de sistemas se encontra com os níveis de desempenho mais baixos nos tipos de virtualização actuais, existe uma enorme necessidade de optimização dos emuladores. Desta forma, surgiram técnicas como o JIT (*Just In Time*), o LLVM (*Low Level Virtual Machine*) ou *Nested Page Tables*, nas quais é possível optimizar a execução dos sistemas virtualizados.

Ambos os emuladores QEMU ou GXemul tiram partido da DBT (*Dynamic Binary Translation*), não só por motivos de portabilidade, mas igualmente com objectivo de optimizar o funcionamento do sistema, dado que com a *Static Binary Translation* não é possível, em determinadas situações, aceder ao código binário com uma só passagem.

Com a utilização de DBT, é possível integrar técnicas como a compilação por blocos de código ou a compilação de código em tempo de execução, nomeadamente compiladores JIT. Uma vez que o emulador desenvolvido no presente trabalho não tira partido

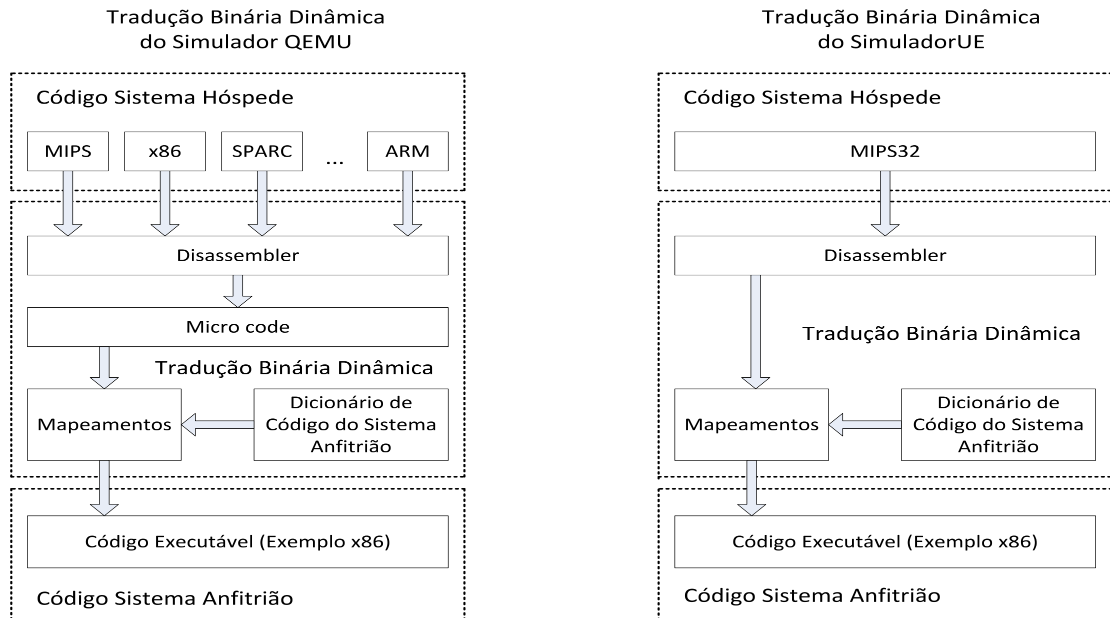


Figura 6.1: Tradução binária dinãmica dos simuladores QEMU e SimuladorUE.

de nenhuma representação intermédia através de *Bytecode*[6], *Micro code*[7] ou *Bitcode*[8], a tradução do código binário é feita, de forma semelhante aos emuladores referidos anteriormente, através do mapeamento de instruções binárias para funções do núcleo de execução do processador desenvolvido na linguagem C/C++.

Na figura 6.1, podemos ver a forma como é feita a tradução dinâmica binária nos simuladores QEMU e no simulador desenvolvido no presente trabalho.

Uma vertente directamente ligada à virtualização, é a emulação de componentes físicos, que será ligada ao simulador, permitindo o sistema hóspede aceder aos mesmos.

Simuladores mais simples como o SPIM não dispõem de suporte para periféricos como discos rígidos, placas de rede ou dispositivos USB, uma vez que a sua principal função é validação do código `assembler` MIPS.
Existem, no entanto, simuladores no mesmo âmbito, como o emulador MARS, que suporta a integração de dispositivos ao simulador, mas de forma limitada.

Relativamente a simuladores mais complexos e completos, como QEMU e o GXemul,

[6]Representação intermédia utilizado pelo compilador da linguagem Java;
[7]Representação intermédia utilizado pelos simuladores QEMU e GXemul;
[8]Representação intermédia utilizado pelo compilador LLVM;

estes dispõem de um grande leque de dispositivos que o sistema hóspede pode aceder. Exemplos destes dispositivos são:

- Placa mãe no qual todos os dispositivos serão interligados;
- Processador genérico MIPS ou baseado num modelo específico;
- Memória RAM;
- Placa gráfica através de adaptadores VESA, AGP ou PCI Express;
- Memória ROM/EROM/EPROM ou EEPROM;
- Discos rígidos através da implementação de controladoras como ICH ou PIIX;
- Controlo para ACPI (Advanced Configuration and Power Interface) através da implementação de controladoras como ICH ou PIIX;
- Placas de redes (NIC);
- Dispositivos USB;
- Dispositivos PCI;
- Dispositivos ISA;
- Dispositivos ópticos como CD ROM/DVD;

Neste aspecto, existe uma grande desvantagem entre o presente simulador e os simuladores complexos acima referidos, dado que a quantidade de dispositivos virtualizados pelo simulador é bastante reduzida. Da lista acima citada apenas um pequeno subconjunto se encontra actualmente suportado no simulador.

Um aspecto muito valorizado em qualquer produto desenvolvido é a sua componente gráfica, com a qual o utilizador irá interagir.

O presente simulador, de forma semelhante ao simulador QEMU, dispõe de uma interface bastante simples, consistindo apenas na utilização de uma consola do Linux para apresentação e interacção entre o utilizador e o sistema hóspede. Optou-se pela simplicidade na interface gráfica do simulador, dando uma maior prioridade na implementação do núcleo de execução e seus respectivos mecanismos. A interface difere da interface utilizada no simulador QEMU no modo como é feita a ligação com o sistema virtualizado. No caso do simuladorUE trata-se de um monitor para a máquina virtual, enquanto que no simulador QEMU é criado um interface série ao qual se liga um terminal.

Uma vez que o mecanismo de interrupções ficou incompleto, não foi possível implementar uma solução semelhante à utilizada no simulador QEMU.

6.4 Trabalho Futuro

Em seguida serão identificados alguns tópicos que ficaram em aberto para desenvolvimento futuro.

Enriquecimento das funcionalidades do processador
Em contexto de virtualização, o mecanismo de interrupção do processador não possui carácter prioritário, uma vez que o seu funcionamento assenta sobre mecanismos mais importantes, nomeadamente os registos do coprocessador central ou o mecanismo de excepções. É claro que a sua utilização é de suma importante, dado que sem esta não é possível a existência de comunicação entre o processador emulado e os periféricos ligados na placa mãe, designadamente os periféricos básicos de *Input/Output* (como o monitor, o rato ou o teclado).

Temos a emulação da memória *cache*, cujo o objectivo é a análise do desempenho de algoritmos no que respeita à utilização da *cache*.

De forma opcional, temos o coprocessador de vírgula flutuante interno no processador, cuja funcionalidade consiste na execução de operações aritméticas sobre números de vírgula flutuante.

Enriquecimento dos periféricos possíveis de virtualizar
No desenvolvimento do emulador, utilizou-se como referência para a placa mãe, a placa física *Malta MIPS*, sendo apenas um subconjunto das suas funcionalidades implementadas.

Outro aspecto relevante é implementação destas funcionalidades, que através do sistema de interrupções permitirão a comunicação entre o utilizador e a máquina virtual dentro do emulador. Desta forma, será possível a ligação de periféricos como: rato, teclado, monitor através de terminais, placa de rede, disco rígido ou portas USB no sistema.

Uma vez que o simulador não dispõe de suporte para discos rígidos nem para controlo da ACPI (*Advanced Configuration and Power Interface*), seria uma poderosa adição no emulador, o suporte para este tipo de componentes através da simulação da uma controladora do tipo ICH (*Input/Output Controller Hub*) ou PIIX (*PCI IDE ISA Xcelerator*), como objectivo a comunicação entre o processador e os periféricos de *Input/ Output*.

Possibilidade de execução de sistemas operativos Linux
Uma das grandes limitações no desenvolvimento do sistema hóspede, foi o facto de este ter sido programado utilizando a linguagem assembler MIPS directamente,

dado que o Workbench mostrou-se inicialmente uma ferramenta muito sofisticada, devido à utilização de bibliotecas dinâmicas do sistema.

Oficialmente, o código gerado por programas compilados pelo `GCC` corre fluentemente no simulador, sendo o único problema a referencia para funções que não se encontram carregadas na memória do sistema. A tarefa de carregamento das bibliotecas dinâmicas, por exemplo o `glibc`, é responsabilidade do sistema operativo, no qual o simulador apenas se limita a executar código binário.

Actualmente, devido a alguns problemas na integração do código gerado pelo compilador `GNU GCC`, o simulador apenas suporta programas compilados pelo `GCC` no qual, o ponto de entrada assim como os endereços utilizados sejam customizados. Para tal, é necessário carregar todas as bibliotecas necessárias com o ficheiro que será executado, uma vez que o simulador não dispõe de suporte para memórias permanentes.
Outra forma é a utilização da opção `static` na compilação do programa, forçando a linkagem de toda a biblioteca num ficheiro único executável, sendo uma opção pouco elegante.

Desta forma, para que seja possível a execução de um sistema operativo Linux no simulador, é necessário que a integração do código gerado por esta plataforma e as bibliotecas da qual estas dependem, se encontrem de alguma forma, carregadas na memória do simulador.

Optimização da execução do emulador
Uma vez que os emuladores convencionais, cujo o funcionamento se baseia num ciclo de leitura, descodificação e execução, se encontram com o desempenho mais baixo nos tipos de virtualização, seria interessante introduzir optimizações tanto na execução como na portabilidade do emulador, que permitissem atingir níveis satisfatórios na virtualização de sistemas que possuam arquitecturas diferentes.

Estes tipos de optimização poderiam ser, à semelhança do que acontece com o emulador QEMU, a tradução binária dinâmica, através da utilização de *bytecode* e de um compilador JIT (*Just In Time*), ou através de técnicas de optimização de código para linguagens convencionais, como LLVM (*Low Level Virtual Machine*).

Bibliografia

[BE94] Farquhar Bunce, Philip and Erin. *The Mips Programmer's Handbook*. Morgan Kaufmann, 1994.

[Bri03] Robert Britton. *MIPS Assembly Language Programming*. Prentice Hall, Illustrated Edition, 2003.

[CgC] Vitaly Chipounov and george Candea. Dynamically translating x86 to LLVM using QEMU.

[Dan06] Sivarama P. Dandamudi. *Guide to RISC Processors for Programmers and Engineers*. Springer, 2006.

[Gor04] Mel Gorman. *Understanding The Linux Virtual Memory Manager*. 2004.

[IA95] Inc IDT and Ltd Algorithmics. *IDT R30xx Family Software Referenc Manual*. Printice-Hall, 1 edition, 1995.

[JAM09] Carlos Ribeiro Luís Veiga e Rodrigo Rodrigues José Alves Marques, Paulo Ferreira. *Sistemas Operativos*. FCA - Editora de Informática Lda, 1 edition, 2009.

[Lov10] Robert Love. *Linux Kernel Development*. Pearson Education, Inc, 3 edition, 2010.

[Mai] David João Maia. Arquitectura MIPS32. *Seminários - Mestrado Engenharia Informática*.

[Mat02] Matías Zabaljáuregui. *Hardware Assisted Virtualization Intel Virtualization Technology*. MIPS Technologies, Inc, 1 edition, 2002.

[MB] David Maia and Miguel Barão. MIPS32 Architecture Simulator. *2ª Jornadas de Informática da Universidade de Évora.*

[Mic09] Microchip. *PIC32 MX Family 5xx/6xx/7xx Data Sheet.* Microchip Technology Inc, 2009.

[MMS01] Jeffrey Oldham Mark Mitchell and Alex Samuel. *Advanced Linux Programing.* David Dwyer, 1 edition, 2001.

[MT02] Inc MIPS Technologies. *Malta^{TM} User's Manual.* MIPS Technologies, Inc, 1 edition, 2002.

[MT08] Inc MIPS Technologies. *MIPS32 Instruction Set Quick Reference*, 1 edition, 2008.

[MT10a] Inc MIPS Technologies. *MIPS Architecture For Programmers, Volume I-A: Introduction to the MIPS32 Architecture.* MIPS Technologies, Inc, 3 edition, 2010.

[MT10b] Inc MIPS Technologies. *MIPS Architecture For Programmers, Volume II-A: The MIPS32 Instruction Set.* MIPS Technologies, Inc, 3 edition, 2010.

[MT10c] Inc MIPS Technologies. *MIPS Architecture For Programmers Volume III: The MIPS32 and microMIPS32 Privileged Resource Architecture.* MIPS Technologies, Inc, 3 edition, 2010.

[PH05] David A. Patterson and John L Hennessy. *Computer Organization and Design - The Hardware/ Software Interface.* Morgan Kaufmann, 2005.

[Sta01] Tool Interface Standards. *Executable and Linkable Format (ELF).* Portable Formats Specification, 1 edition, 2001.

[Sta05] William Stallings. *Operating Systems, Internals and Design Principles.* Prentice Hall, Illustrated Edition, 5 edition, 2005.

[Swe06] Dominic Sweetman. *See MIPS Run - Second Edition.* Denise E. M. Penrose, 2006.

[WGP] Rodolfo Wottrich, Thiago Genez, and Walisson Pereira. Uma Visão Geral Sobre Sistemas Virtualizados.

Anexos

Anexo A

Compilação de código binário - Makefile

Segue-se o conteúdo do ficheiro Makefile utilizado para geração do código binário para os exemplos demonstrados no capítulo 5.

```
1  # Compilacao Cruzada para arquitectura MIPS
2  # Compilador Assembly
3
4  AC = mips-linux-gnu-as
5  LK = mips-linux-gnu-ld
6  OC = mips-linux-gnu-objcopy
7  GCC = mips-linux-gnu-gcc
8
9  CFLAGS = -c
10 CDEBUG = -g
11
12 BIN = bin
13 SRC = src
14 AUX = aux
15
16 clean:
17         rm -rf *~
18         rm -rf $(BIN)/*
19         rm -rf $(AUX)/*
```

```
20          rm -rf $(SRC)/*~
21
22 exemplo1:
23          $(AC) -EL -mips32r2 $(SRC)/exemplo1.asm -o $(AUX)/exemplo1.o
24          $(LK) -EL -N -e 0x80000000 -Ttext=0x80000000 -Tdata=0
                x80010000
25          $(AUX)/exemplo1.o -o $(BIN)/exemplo1.bin
26
27 exemplo2:
28          $(AC) -EL -mips32r2 $(SRC)/exemplo2.asm -o $(AUX)/exemplo2.o
29          $(OC) -F elf32-tradlittlemips --remove-section=.reginfo
30          $(AUX)/exemplo2.o $(LK) -EL -N -e 0x80000000 -Ttext=0
                x00400000
31          -Tdata=0x00410000 --section-start=.config=0x80000000
32          $(AUX)/exemplo2.o -o $(BIN)/exemplo2.bin
33
34 exemplo3:
35          $(AC) -EL -mips32r2 $(SRC)/exemplo3.asm -o $(AUX)/exemplo3.o
36          $(OC) -F elf32-tradlittlemips --remove-section=.reginfo $(AUX
                )/
37          exemplo3.o $(LK) -EL -N -e 0x80001000 -Ttext=0x80001000
38          -Tdata=0x80002000 --section-start=.TLBrefill=0x80000000
39          --section-start=.userText=0x00400000 --section-start
40          =.generalVector=0x80000180 $(AUX)/exemplo3.o -o $(BIN)/
41          exemplo3.bin
42
43 exemplo4:
44          $(GCC) -EL -mips32r2 -mabi=32 -c $(SRC)/primos.c -o
45          $(AUX)/exemplo4.o
46          $(LK)  -EL -N $(AUX)/exemplo4.o -o $(BIN)/exemplo4.bin
47
48 compile: clean exemplo1 exemplo2 exemplo3 exemplo4
```

Listing A.1: Ficheiro Makefile das demonstrações.

Anexo B

Código Assembler MIPS - Demonstração 1

O seguinte código fonte `assembler` é utilizado para demonstrar o exemplo 1, no qual é configurado o processador no seu estado inicial sendo, em seguida executado o factorial do número seis.

```
int fact(int n){
        if(n<1)
                return (1);
        else
                return (n*fact(n-1));
}
```

Listing B.1: Demonstração 1 em C.

```
1  .set noat
2  .text
3  main:    # Preparacao do Processador
4
5           # valor de n para teste , n = 5
6           addi $a0,$zero,6
7
8           # STACK Pointer = 0xE0000100
9           lui     $at, 0xe000
10          ori     $sp, $at, 0x0100
11
12          jal FACT
13          nop
14
15          j END
16          nop
17
18 FACT:    add $t0,$zero,$a0        # parametro
19          slti $t1,$t0,1           # if(n<1)
20          beq $t1,$zero,ELSE       #
21          nop
22                                   # THEN
23          addi $v0,$zero,1         # returna 1
24          jr $ra                   #
25          nop
26
27 ELSE:    addi $sp,$sp,-8          # alocar espaco na STACK
28          sw $a0, 4($sp)           # guarda parametro n
29          sw $ra, 0($sp)           # guarda registo de retorno
30
31          add $a0,$t0,-1           # passa o (n-1) como parametro
32          nop
33          jal FACT                 # chama procedimento FACT novamente
34          nop
35
36          lw $ra, 0($sp)           # restaura o valor de $ra
37          lw $a0, 4($sp)           # restaura o valor de $a0
38          addi $sp,$sp, 8          # desaloca pilha
39
40                                   # multiplicacao
41          add $t2,$zero,$zero      # variavel temporaria X
42          add $t3,$zero,$zero      # contador = 0
43
44 FOR:                              # n*$v0
```

```
45          beq $t3  ,$a0 , FIMFOR      # condicao de saida do loop
46          nop
47          add $t2 , $t2 , $v0         # x+=v0
48          addi $t3 , $t3 ,1           # contador ++
49          j FOR                       # loop
50          nop
51
52 FIMFOR:  add $v0 ,$t2 ,$zero         # resultado
53          jr $ra                      # returna n*fact(n-1)
54          nop
55
56 END:
```

Listing B.2: Demonstração 1 em Assembly MIPS.

Anexo C

Código Assembler MIPS - Demonstração 2

O seguinte código fonte `assembler` é utilizado para demonstrar o exemplo 2, no qual é configurado o processador no seu estado inicial, sendo em seguida executado o factorial do número seis.

```
int fibonacci(int n){
        if n == 0
                return 0;

        if (n < 2)
                return 1;

        return fibonacci(n - 1) + fibonacci(n - 2);
}

# 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610
```

Listing C.1: Demonstração 2 em C.

```
1  .set noat
2  .text
3  main:   addi $a0,$zero,15        # valor de n para teste, n = 8
4
5          lui   $at, 0x7fff
6          ori   $at, $at, 0xeffc
7          add   $sp, $0, $at
8
9          jal fib
10         nop
11
12         j END
13         nop
14
15 fib:    beq $a0, $zero, zero
16         nop
17
18         slti $t0, $a0, 2         # if i < 2 ( i == 1)
19         beq $t0, $zero, cont     # if i >= 2 salta para cont
20         nop
21         addi $v0, $zero, 1       # else retorna 1
22
23         jr $ra
24         nop
25
26 zero:   addi $v0, $zero, 0       # else retorna 0
27         jr $ra
28         nop
29
30 cont:
31
32         # Alocar espaco na STACK
33
34         addi $sp, $sp, -16              # 3 elementos para a stack
35         sw $ra, 0($sp)                  # Return Address
36         sw $a0, 4($sp)
37
38         # Chamadas recursivas
39
40         addi $a0, $a0, -1        # calcular n - 1
41         jal fib                  # calcular fib(n - 1)
42         nop
43
```

```
44          sw $v0, 8($sp)            # Guardar na STACK o resultado de fib
                (n - 1)
45          lw $a0, 4($sp)            # Carregar valor de n
46
47          addi $a0, $a0, -2         # calcular n - 2
48          jal fib
49          nop
50
51          sw $v0, 12($sp)           # Guardar na STACK o resultado de fib
                (n - 2)
52
53          # Retirar valores
54          #da STACK
55
56          lw $ra, 0($sp)            # Carregar return address
57          lw $t0, 8($sp)            # Carregar valor de fib(n - 1)
58          lw $t1, 12($sp)           # Carregar valor de fib(n - 2)
59          addi $sp, $sp, 16         # Remover da STACK
60
61          # Calculo
62          add $v0, $t0, $t1         # fib(n - 1) + fib(n - 2)
63
64          jr $ra
65          nop
66  END:
67
68  .section .config
69          # Configurar Processador
70          # Obter registo Status
71          mfc0 $t6, $12, 0
72
73          # Activar modo User KSU
74          ori $t6, $t6, 0b0000000000010000
75
76          # Introduzir valor 0x0040000 no ErrorEPC
77          lui $at, 0x0040
78          ori $at, $at, 0x0
79          mtc0 $at, $30, 0
80
81          # Escrever Registo Status
82          mtc0 $t6, $12, 0          #Escrever no coprocessador 0
83
84          eret              # Comeca o programa
```

Listing C.2: Demonstração 2 em Assembly MIPS.

Anexo D

Código Assembler MIPS - Demonstração 3

O seguinte código fonte `assembler` é utilizado para demonstrar o exemplo 3, no qual é configurado o processador no seu estado inicial, sendo em seguida apresentado vários casos de erro, assim como mecanismos MIPS.

```
1  .data
2  info:           .asciiz "SimuladorUE"
3  info2:          .word 10, 20, 30, 40
4
5  .text
6  .set noat
7  main:    # Configurar processador
8           jal Config
9           nop
10
11          # Gerar excepcao Machine Check
12          jal MCheck
13          nop
14
15          # Modificar modo de execucao no processador
16          jal Switch
17          nop
18
```

```
19          j end
20          nop
21
22  #————————————————————————————————————————————————
23
24  Config: # Configurar Processador
25
26          # Obter registo Status
27          mfc0 $t6, $12, 0
28
29          # Desligar StatusBEV e StatusERL
30          lui $t0, 0b1111111110111111
31          ori $t0, $t0, 0b1111111111111011
32
33          and $t6, $t6, $t0
34
35          # Escrever Registo Status
36          mtc0 $t6, $12, 0         #Escrever no coprocessador 0
37
38          jr $ra                   # Comeca o programa
39
40  #————————————————————————————————————————————————
41
42  Switch: # Mudar para modo User
43
44          # Obter registo Status
45          mfc0 $t6, $12, 0
46
47          # Deixar Bit CU3..0 activo
48          lui $t0, 0b0001000000000000
49          or $t6, $t6, $t0
50
51          ori $t6, $t6, 0b0000000000010010
52
53          # Escrever Registo Status
54          #Escrever no coprocessador 0
55          mtc0 $t6, $12, 0
56
57
58          # Introduzir valor 0x0040000 no EPC
59          lui $at, 0x0040
60          ori $at, $at, 0x0
61          mtc0 $at, $14, 0
62
```

```
63          eret                        # Comeca o programa em user
64
65  #-----------------------------------------------------------------
66
67  MCheck: addi $s0, $0, 0xF000                #EntryHi
68          addi $s1, $0, 0x6000                #EntryLo0
69          addi $s2, $0, 0x5000                #EntryLo1
70          addi $s3, $0, 0x1000                #PageMask
71          addi $s4, $0, 10                    #Index
72
73          mtc0 $s0, $10
74          mtc0 $s1, $2
75          mtc0 $s2, $3
76          mtc0 $s3, $5
77          mtc0 $s4, $0
78
79          tlbwi
80
81          addi $s0, $0, 0xF000                #EntryHi
82          addi $s1, $0, 0x6000                #EntryLo0
83          addi $s2, $0, 0x5000                #EntryLo1
84          addi $s3, $0, 0x1000                #PageMask
85          addi $s4, $0, 11                    #Index
86
87          mtc0 $s0, $10
88          mtc0 $s1, $2
89          mtc0 $s2, $3
90          mtc0 $s3, $5
91          mtc0 $s4, $0
92
93          tlbwi
94          nop
95
96          jr $ra
97  #-----------------------------------------------------------------
98  end:
99
100 #
101 # Handler do entry point 0x8000.0180
102 #
103 .section .generalVector
104         #Guardar na STACK os registos Status, EPC e afins
105         mfc0 $k0, $14, 0                          #Status EPC
106         sw $k0, 0($sp)
```

```
107         addi $sp, $sp, -4
108
109         mfc0 $k0, $13, 0                        #Status Cause
110         sw $k0, 0($sp)
111         addi $sp, $sp, -4
112         #-----------------------------------------
113
114         mfc0 $k0, $13, 0                        #Get Cause Register
115         andi $k0, $k0, 0b0000000001111100       #Extrair ExcCode
116
117         addi $k1, $0, 0x08                      #Codigo do TLBL sem
118         beq $k0, $k1, handle_tlbl               #shift 0x08 => 0x02
119
120         addi $k1, $0, 0x10                      #Codigo do AdEL sem
121         beq $k0, $k1, handle_adel               #shift 0x10 => 0x04
122
123         addi $k1, $0, 0x20                      #Codigo do syscall
                sem
124         beq $k0, $k1, handle_syscall            #shift 0x20 => 0x08
125
126         addi $k1, $0, 0x2c                      #Codigo do CpU sem
127         beq $k0, $k1, handle_cpu                #shift 0x2c => 0x0b
128
129         addi $k1, $0, 0x60                      #Codigo do MCheck sem
130         beq $k0, $k1, handle_mcheck             #shift 0x60 => 0x18
131
132         #-----------------------------------------
133
134 return: # Repor os registos da STACK
135         addi $sp, $sp, 4
136         lw $t6, 0($sp)
137         mtc0 $t6, $13, 0         #Status Cause
138
139         addi $sp, $sp, 4
140         lw $t6, 0($sp)
141         mtc0 $t6, $14, 0         #Status EPC
142
143         #Return para a instrucao geradora do erro
144         eret
145         nop
146
147         #-----------------------------------------
148
149 handle_tlbl:     j return
```

```
150
151
152 handle_adel:     # Incrementar o EPC em 4 para fazer o bypass
153                  lw $k0, 8($sp)           #EPC
154                  addi $k0, $k0, 4         #Bypass
155                  sw $k0, 8($sp)
156
157                  j return
158                  nop
159
160 handle_syscall:  addi $v0, $zero, 5
161                  lw $k0, 8($sp)           #EPC
162                  addi $k0, $k0, 4         #Bypass
163                  sw $k0, 8($sp)
164
165                  j return
166                  nop
167
168
169 handle_cpu:      lw $k0, 8($sp)           #EPC
170                  addi $k0, $k0, 4         #Bypass
171                  sw $k0, 8($sp)
172
173                  j return
174                  nop
175
176
177 handle_mcheck:   # Desligar bit StatusTS
178
179                  mfc0 $k0, $12, 0                # Status
180                  lui  $k1, 0xFFDF                # Mascara para TS
181                  and  $k0, $k0, $k1              # Desligar TS
182                  mtc0 $k0, $12, 0                # Escrever Status
183                                                  # no coprocessador 0
184
185                  # Apagar o primeiro mapeamento
186                  addi $s0, $0, 0x00              #EntryHi
187                  addi $s1, $0, 10                #Index
188
189                  mtc0 $s0, $10
190                  mtc0 $s1, $0
191
192                  tlbwi
193
```

```
194                 j return
195
196 #
197 # Segmento de Text User 0x00400000
198 #
199 .section .userText
200
201                 # Gerar erro AdEL
202                 la $t0, info
203                 lw $t1, 0($t0)
204
205                 # Gerar erro TLBL
206                 # Limpar mapeamento da STACK e tentar aceder
207                 li $t0, 2048
208                 sw $t0, -32($sp)
209
210                 # Remover mapeamento da Stack na TLB
211                 addi $s0, $0, 0x00      #EntryHi
212                 addi $s4, $0, 0         #Index
213
214                 mtc0 $s0, $10
215                 mtc0 $s4, $0
216
217                 tlbwi
218
219                 lw $t1, -32($sp)
220
221                 # Gerar erro Coprocessor Unusable
222                 # Remover permissao de modificao
223
224 cpuerror:       mfc0 $t6, $12, 0        # Status
225
226                 # Deixar Bit CU3..0 activo
227                 lui $t0, 0b1110111111111111
228                 ori $t0, $t0, 0b1111111111111111
229                 and $t6, $t6, $t0
230
231                 # Escrever Status no coprocessador 0
232                 mtc0 $t6, $12, 0
233
234                 # A partir daqui da erro no acesso aos registos
                        internos
235                 mfc0 $t7, $12, 0        # Status
236
```

```
237 userexit:      syscall
238                nop
239                #Fim do programa
240
241
242 #
243 # Segmento de Text User 0x80000000
244 #
245 .section .TLBrefill
246                # Remapear pagina
247                lui  $s0, 0x7FFF       #EntryHi
248                ori  $s0, $s0, 0xE000
249
250                lui  $s1, 0x0000       #EntryLo0
251                ori  $s1, $s1, 0x0100
252
253                lui  $s2, 0x0000       #EntryLo1
254                ori  $s2, $s2, 0x0140
255
256                lui  $s3, 0x0000       #PageMask
257                ori  $s3, $s3, 0x1800
258
259                addi $s4, $0, 8        #Index
260
261                mtc0 $s0, $10
262                mtc0 $s1, $2
263                mtc0 $s2, $3
264                mtc0 $s3, $5
265                mtc0 $s4, $0
266
267                tlbwi
268
269                #Return para a instrucao geradora do erro
270                eret
271                nop
```

Listing D.1: Demonstração 3 em Assembly MIPS.

Anexo E

Código Assembler MIPS - Demonstração 4

O seguinte código fonte `assembler` é utilizado para demonstrar o exemplo 4, no qual é configurado directamente no simulador o estado inicial do processador, sendo em seguida executada várias chamadas de validação dos números primos.

```
//#include <stdio.h>

void main(){
        int res;
        res = primos(6);
        __asm__ ("move $t0, $v0;");
        __asm__ ("li $t1, 6;");
        //printf("-> %d\n", res);

        res = primos(7);
        __asm__ ("move $t2, $v0;");
        __asm__ ("li $t3, 7;");
        //printf("-> %d\n", res);

        res = primos(11);
        __asm__ ("move $t4, $v0;");
        __asm__ ("li $t5, 11;");
        //printf("-> %d\n", res);
```

```
        res = primos(15);
        __asm__ ("move $t6, $v0;");
        __asm__ ("li $t7, 15;");

        __asm__ ("j end;");
        //printf("-> %d\n", res);

        //return 2;
}

// 0 - primo
// 1 - nao primo
int primos(int num){
int res = 0;

int i=0;
for(i=1; i<=num ; i++){
                if ( (num % i) == 0 )
                res += 1;
}

//printf("-> %d\n", res);
return res;
}

__asm__ ("end: nop;");
```

Listing E.1: Demonstração 4 em C.

Segue-se o código fonte desmontado através do `mips-gnu-linux-objdump`:

```
Disassembly of section .text:

00400090 <main>:
  400090:     27bdffd8     addiu   sp,sp,-40
  400094:     afbf0024     sw      ra,36(sp)
  400098:     afbe0020     sw      s8,32(sp)
  40009c:     03a0f021     move    s8,sp
  4000a0:     0c100042     jal     400108 <primos>
  4000a4:     24040006     li      a0,6
  4000a8:     afc20018     sw      v0,24(s8)
  4000ac:     00404021     move    t0,v0
  4000b0:     24090006     li      t1,6
  4000b4:     0c100042     jal     400108 <primos>
  4000b8:     24040007     li      a0,7
```

```
  4000bc:       afc20018      sw      v0,24(s8)
  4000c0:       00405021      move    t2,v0
  4000c4:       240b0007      li      t3,7
  4000c8:       0c100042      jal     400108 <primos>
  4000cc:       2404000b      li      a0,11
  4000d0:       afc20018      sw      v0,24(s8)
  4000d4:       00406021      move    t4,v0
  4000d8:       240d000b      li      t5,11
  4000dc:       0c100042      jal     400108 <primos>
  4000e0:       2404000f      li      a0,15
  4000e4:       afc20018      sw      v0,24(s8)
  4000e8:       00407021      move    t6,v0
  4000ec:       08100064      j       400190 <end>
  4000f0:       240f000f      li      t7,15
  4000f4:       03c0e821      move    sp,s8
  4000f8:       8fbf0024      lw      ra,36(sp)
  4000fc:       8fbe0020      lw      s8,32(sp)
  400100:       03e00008      jr      ra
  400104:       27bd0028      addiu   sp,sp,40

00400108 <primos>:
  400108:       27bdffe8      addiu   sp,sp,-24
  40010c:       afbe0014      sw      s8,20(sp)
  400110:       03a0f021      move    s8,sp
  400114:       afc40018      sw      a0,24(s8)
  400118:       afc00008      sw      zero,8(s8)
  40011c:       afc0000c      sw      zero,12(s8)
  400120:       24020001      li      v0,1
  400124:       afc2000c      sw      v0,12(s8)
  400128:       08100059      j       400164 <primos+0x5c>
  40012c:       00000000      nop
  400130:       8fc30018      lw      v1,24(s8)
  400134:       8fc2000c      lw      v0,12(s8)
  400138:       0062001a      div     zero,v1,v0
  40013c:       004001f4      teq     v0,zero,0x7
  400140:       00001010      mfhi    v0
  400144:       14400004      bnez    v0,400158 <primos+0x50>
  400148:       00000000      nop
  40014c:       8fc20008      lw      v0,8(s8)
  400150:       24420001      addiu   v0,v0,1
  400154:       afc20008      sw      v0,8(s8)
  400158:       8fc2000c      lw      v0,12(s8)
  40015c:       24420001      addiu   v0,v0,1
  400160:       afc2000c      sw      v0,12(s8)
```

```
  400164:       8fc3000c        lw      v1,12(s8)
  400168:       8fc20018        lw      v0,24(s8)
  40016c:       0043102a        slt     v0,v0,v1
  400170:       1040ffef        beqz    v0,400130 <primos+0x28>
  400174:       00000000        nop
  400178:       8fc20008        lw      v0,8(s8)
  40017c:       03c0e821        move    sp,s8
  400180:       8fbe0014        lw      s8,20(sp)
  400184:       27bd0018        addiu   sp,sp,24
  400188:       03e00008        jr      ra
  40018c:       00000000        nop

00400190 <end>:
        ...
```

Listing E.2: Demonstração 4 em Assembly MIPS.