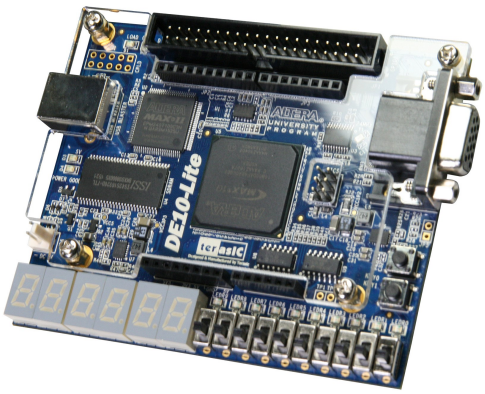
TÉCNICO LISBOA



Processador Didático P4

Especificação e Implementação do Processador e Ambiente de Desenvolvimento

Dinis Pedro Pinto Marcos Madeira

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

Orientadores: Prof. Rui António Policarpo Duarte
Prof. José Carlos Alves Pereira Monteiro

Júri

Presidente: Prof. Alberto Manuel Rodrigues da Silva
Orientador: Prof. Rui António Policarpo Duarte
Vogal: Prof. Arlindo Manuel Limede de Oliveira

Abril 2019

Resumo

O livro “Arquitectura de Computadores: dos Sistemas Digitais aos Microprocessadores” [1] apresenta dois processadores didáticos, o Pequeno Processador Pedagógico (P3), um processador CISC, e o Pequeno Processador Pedagógico com *Pipeline* (P4), um processador RISC. O P3 foi implementado em hardware como parte de uma plataforma didática utilizada nas aulas de laboratório de Arquitetura de Computadores no Instituto Superior Técnico. Estando prevista uma mudança de ênfase nesta unidade curricular de *Complex Instruction Set Computer* (CISC) para *Reduced Instruction Set Computer* (RISC), nesta dissertação foi feita a especificação do P4, a sua implementação num sistema de hardware reconfigurável, com suporte para um conjunto de periféricos de entrada e saída, incluindo *LEDs*, interruptores, botões de pressão, mostradores hexadecimais, um mostrador alfanumérico, um ecrã VGA, um teclado PS/2 e um acelerómetro de 3 eixos. Adicionalmente, foi criado um ambiente de desenvolvimento baseado numa plataforma *web*, incluindo um editor de *assembly*, um assemblador, ferramentas de depuração e um simulador, que podem ser acedidos *online* através de um *navegador* de Internet, dispensado a transferência de ficheiros ou a instalação de programas no sistema do utilizador.

Palavras-chave: Processador, P4, RISC, Sistema Didático.

Abstract

The “Arquitectura de Computadores: dos Sistemas Digitais aos Microprocessadores” [1] book presents two didactic processors, the Pequeno Processador Pedagógico (P3), a CISC processor, and the Pequeno Processador Pedagógico com *Pipeline* (P4), a RISC processor. The P3 processor has been implemented before in hardware as a learning platform for the laboratory classes of the Computer Architecture course for the Computer Science degree at Instituto Superior Técnico. Considering an evolution of the course from a focus on CISC to a focus on RISC, the work on this dissertation consists of the specification of the P4, and its implementation on a reconfigurable hardware, supporting a set of I/O devices, including LEDs, switches, push buttons, hexadecimal displays, an alphanumeric display, a VGA display, a PS/2 keyboard and a 3-axis accelerometer. Additionally, the creation of the development environment based in a web platform, including an assembly editor, an assembler, debugging tools and a simulator, which can be accessed online with a web browser, with no need to download files or installing software in the user’s system.

Keywords: Processor, P4, RISC, Learning Platform.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Acrónimos

P3 Pequeno Processador Pedagógico

P4 Pequeno Processador Pedagógico com *Pipeline*

CISC *Complex Instruction Set Computer*

RISC *Reduced Instruction Set Computer*

ISA *Instruction Set Architecture*

PC *Program Counter*

RI Registo de Instrução

ULA Unidade Lógica e Aritmética

VHDL *Very High Speed Integrated Circuit Hardware Description Language*

FPGA *Field-Programmable Gate Array*

1 Introdução

O processador é o componente do computador responsável por realizar operações aritméticas, lógicas e de controlo, e ainda controla a entrada e saída de dados de vários dispositivos, como memórias, *caches* e periféricos. As operações executadas pelo processador são especificadas através de instruções em linguagem máquina, que estão organizadas em conjuntos de bits.

Um processador é caracterizado por trabalhar sobre conjuntos de bits de um determinado tamanho, realizar diferentes operações, com maior ou menor complexidade, que sejam capazes de aceder diretamente à memória ou apenas fazer operações entre valores de registos. Estas características determinam a arquitetura de um processador.

A *Instruction Set Architecture* (ISA) é uma das primeiras abstrações do processador, fornecendo uma interface entre o hardware e o software ao mais baixo nível. Uma ISA pode ter diferentes realizações em hardware, mas ainda assim é possível correr diretamente em todas elas o software que tenha sido desenvolvido para essa ISA.

Existem dois tipos de ISA: *Complex Instruction Set Computer* (CISC) e *Reduced Instruction Set Computer* (RISC). As arquiteturas CISC caracterizam-se por possuírem instruções que realizam operações mais complexas que podem requerer vários ciclos de relógio para serem executadas. Por outro lado, as arquiteturas RISC, implementam apenas instruções que realizam operações simples e são executadas a uma média próxima de um ciclo de relógio graças à técnica de *pipelining*, que consiste em dividir a execução de cada instrução em vários passos sequenciais correspondentes a diferentes estágios. Desta forma obtém-se uma execução sobreposta de várias instruções, cada uma num estágio diferente, à semelhança de uma linha de montagem.

Para demonstrar o funcionamento de um *pipeline*, é ilustrada na Figura 1.1 a execução de seis instruções, I1 a I6, em nove ciclos de relógio, T1 a T9, cuja execução acontece em quatro estágios. É possível observar-se que cada instrução demora quatro ciclos de relógio a ser executada. No entanto, a partir do instante T4, quando o *pipeline* fica preenchido, há quatro instruções a serem executadas simultaneamente em estágios diferentes, conseguindo-se desta forma uma média de uma instrução executada por ciclo de relógio.

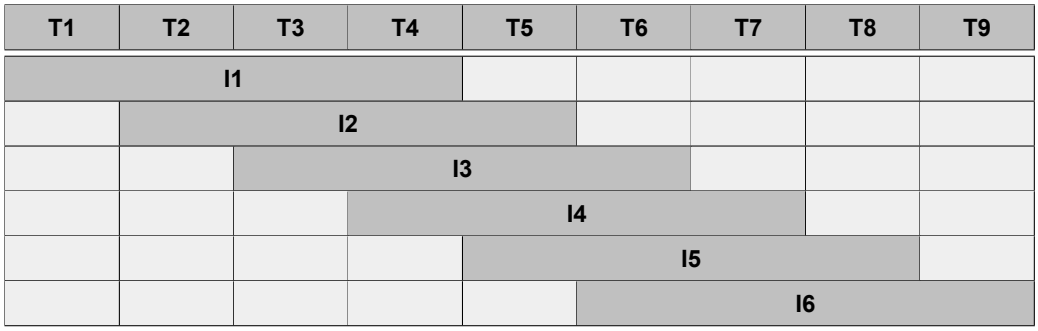


Figura 1.1. Demonstração do funcionamento de um *pipeline* com quatro andares.

A técnica de *pipelining* é facilitada pela complexidade reduzida das operações correspondentes a uma instrução em arquiteturas RISC, que torna possível a divisão das instruções em estágios correspondentes aos andares do *pipeline*. No entanto, o facto de a execução de uma instrução começar antes de as instruções que a precedem terminarem a sua execução leva à existência de conflitos quando a execução de uma instrução depende da conclusão das instruções anteriores. A resolução destes conflitos é uma das questões abordadas na especificação e implementação em hardware do P4.

As arquiteturas CISC e RISC distinguem-se pelas seguintes características:

- O número e complexidade das instruções: as arquiteturas CISC disponibilizam instruções para operações complexas, enquanto que as arquiteturas RISC apenas oferecem instruções para operações mais simples e mais usadas pelos programas.
- O formato das instruções: as arquiteturas RISC oferecem um formato mais regular e de comprimento fixo, que permitem que os circuitos de descodificação das instruções sejam menos complexos, enquanto que as arquiteturas CISC têm instruções com formatos mais complexos e de comprimento variável.
- Modos de endereçamento e acesso à memória: as arquiteturas RISC oferecem um número mais limitado de modos de endereçamento. Processadores baseados em arquiteturas RISC são descritos como máquinas carregar-guardar (*load-store*), visto que para realizar operações carrega-se, em primeiro lugar, os operandos para registos e posteriormente guarda-se o resultado na memória, ao contrário de arquiteturas CISC, que normalmente possuem modos de endereçamento que permitem fazê-lo numa única instrução que trabalha diretamente com os valores na memória.

As arquiteturas CISC têm a vantagem de ser necessário um menor número de instruções *assembly*, o que se traduz numa menor utilização da memória, em relação ao número de instruções numa arquitetura RISC para realizar a mesma tarefa, visto que num processador RISC é frequentemente necessário utilizar várias instruções para realizar as mesmas operações realizadas por uma única instrução num processador CISC. A maior complexidade das operações realizadas pelas instruções em arquiteturas CISC também significa que o processo de compilação de uma linguagem de alto nível para *assembly* requer menos trabalho por parte do compilador [2].

Por outro lado, os processadores RISC têm uma arquitetura mais simples, que requer menos recursos, o que se traduz na vantagem de ter o circuito a funcionar mais rapidamente. Apesar de serem necessárias mais instruções para realizar uma mesma tarefa, consegue obter-se um desempenho superior ao de processadores CISC. Isto deve-se ao facto de as operações executadas por estas instruções serem mais simples e terem um tempo de execução constante, permitindo tirar partido de técnicas de *pipelining*. As arquiteturas carregar-guardar têm ainda a vantagem de não ser preciso carregar os operandos novamente a partir da memória caso tenham sido usados na operação anterior e que sejam necessários numa nova operação, visto que estes mantém-se nos registos até que outros valores sejam carregados para esses registos, reduzindo assim os acessos à memória.

Pela sua maior simplicidade em relação aos processadores CISC, os processadores RISC são bons objetos de estudo numa introdução à arquitetura de computadores. Contudo, um processador comercial capaz de suportar um sistema operativo como Linux ou Windows tem de cumprir requisitos que

vão muito além daquilo que é desejável numa primeira abordagem, pois estes requisitos traduzem-se num aumento de complexidade que dificulta a compreensão do sistema como um todo e dos princípios básicos que se pretendem estudar.

O P4 é um processador que foi inicialmente proposto no livro "Arquitectura de Computadores: dos Sistemas Digitais aos Microprocessadores" [1], a par com o P3, um processador CISC. O P3 está totalmente especificado no livro e já se encontrava implementado em hardware reconfigurável, usado para experimentar nas aulas laboratoriais. O P4 pode ser visto como uma versão RISC do P3, mas ao contrário deste, não está completamente especificado no livro e nunca tinha sido implementado antes em hardware.

Esta dissertação descreve a especificação do P4 e o desenvolvimento de um sistema didático para facilitar o seu estudo. O sistema didático consiste numa placa de desenvolvimento com um dispositivo *Field-Programmable Gate Array* (FPGA) [3] onde o P4 foi implementado, com vários periféricos de entrada e saída que permitem a interação com o utilizador. Como parte desta solução foi também criado um ambiente de desenvolvimento com ferramentas de apoio que permitem criar programas *assembly* para o P4, que podem ser testados e depurados num simulador, ou carregados no P4 configurado na FPGA de forma a poder ser testado o seu funcionamento num sistema real.

Esta solução traduz-se num conjunto de ferramentas que tornam possível estudar a fundo um processador com arquitetura RISC, desde a sua implementação ao nível de portas lógicas e registos, até ao nível da sua programação em *assembly*, fazendo assim a ponte entre o hardware e o software.

1.1 Objetivos

Este trabalho teve como objetivos:

- Estudar e completar a especificação do processador P4 proposto no livro [1].
- Implementar o P4 numa FPGA.
- Suportar periféricos de entrada e de saída.
- Criar um assemblador para o P4.
- Criar um simulador e depurador para o P4, incluindo os periféricos.
- Criar uma ferramenta de carregamento, depuração e controlo do P4 na FPGA.

1.2 Organização do Documento

Esta dissertação está organizada em oito capítulos. Neste capítulo foi contextualizado e apresentado o trabalho realizado no âmbito desta tese. No Capítulo 2 serão analisadas outras soluções já existentes e como estas se comparam com a solução desenvolvida nesta dissertação. O Capítulo 3 introduz o leitor às várias tecnologias utilizadas no desenvolvimento deste trabalho. A solução desenvolvida começa a ser apresentada no Capítulo 4, com a especificação completa do P4, e de seguida, no Capítulo 5, descrevem-se os desenvolvimentos ao nível de software. O Capítulo 6 foca-se no processo de desenvolvimento da solução ao nível do hardware, onde a implementação em hardware do P4 é descrita em

maior detalhe, incluindo o projeto de vários componentes, ao mesmo tempo que são indicados os desafios encontrados na sua conceção e as soluções encontradas. No Capítulo 7 é feita a avaliação do sistema, analisando-se os recursos usados, o seu desempenho, e é demonstrado o seu correto funcionamento. O documento termina com o Capítulo 8, onde são feitas algumas reflexões sobre o trabalho desenvolvido e quais poderão ser os desenvolvimentos futuros.

2 Trabalho Relacionado

Inicialmente os programas eram feitos diretamente em *assembly* e as memórias mais rápidas tinham um custo muito elevado, por estes motivos, programas compactos eram desejáveis, o que levou à popularização das arquiteturas CISC por permitirem a realização de operações complexas com uma única instrução. Nomeadamente, o processador CISC de 16 bits Intel 8086 introduzido em 1979, deu origem a uma das famílias de processadores mais bem sucedidas da Intel.

Os processadores RISC representam uma mudança de paradigma no sentido de um hardware mais simples, os quais têm tido cada vez maior desenvolvimento nos últimos tempos, especialmente suportado pelo desenvolvimento de sistemas embebidos.

Alguns processadores RISC começaram como projetos académicos, e existe muito material para o seu estudo. Neste capítulo, vamos analisar alguns processadores RISC: o Nios II [4], o RISC-V [5], o MIPS [6] e o ARM [7], bem como simuladores para alguns destes e outros processadores.

2.1 Processadores

2.1.1 MIPS

O MIPS começou como um projeto na *Stanford University* em 1981 [8]. Este foi concebido sob o paradigma de simplificar o hardware, transferindo a complexidade para o compilador. Implementa uma ISA com instruções simples e que são executadas num único ciclo de relógio utilizando um *pipeline* dde 5 andares. MIPS é um acrónimo para *Microprocessor without Interlocked Pipeline Stages*, em referência à técnica de *pipelining* utilizada neste processador, que foi o primeiro processador RISC e ajudou a popularizar este tipo de arquitetura.

Por ter começado como um projeto académico, possuir uma arquitetura simples, e ter servido de base a vários textos e livros pedagógicos, é um bom candidato no estudo de Arquitetura de Computadores. Existem alguns simuladores de processadores MIPS que facilitam o seu estudo, nomeadamente: ViSiMIPS [9], SPIM [10], MARS [11], WinMips64 [12], entre outros.

2.1.2 RISC-V

O RISC-V começou como um projeto académico na *University of California, Berkeley*. Em 2015 foi criada a *RISC-V Foundation* onde membros de todo o mundo colaboram no desenvolvimento do RISC-V.

O RISC-V é uma ISA com princípios RISC, desenhada para poder corresponder a diferentes necessidades. A sua principal vantagem é ser uma ISA aberta. Isto significa que os desenvolvedores de sistemas podem fazer a sua própria implementação do RISC-V em hardware, sem pagar quaisquer licenças ou direitos de autor. No entanto, as suas implementações podem ser fechadas, salvaguardando a propriedade intelectual de quem desenvolve a implementação personalizada.

Embora não seja a única ISA aberta, o RISC-V destaca-se também pela sua modularidade. Existe um sistema base com um conjunto de requisitos mínimos que todas as implementações devem suportar, que pode ser estendido com módulos padrão. Estes módulos padrão são compatíveis entre si e implementam as funcionalidades mais utilizadas nos processadores.

A abertura e modularidade do RISC-V traduzem-se numa grande vantagem para os desenvolvedores de sistemas. Atualmente teriam de usar uma ISA já existente, que normalmente implica o pagamento de licenças e direitos de autor, e que poderia não estar adaptada às suas necessidades. Por outro lado, caso optassem por adaptar uma ISA já existente às suas necessidades, ou criar uma de raiz, teriam de sacrificar todo o ecossistema de bibliotecas e ferramentas desenvolvidas para essa ISA, o que não acontece no RISC-V, pois todas as implementações do sistema base são compatíveis entre si.

O RISC-V tem ainda a vantagem de não ser uma ISA projetada para nenhuma microarquitetura em particular, que poderia limitar a sua usabilidade noutras microarquiteturas. Desta forma o RISC-V pode ser usado em sistemas com diferentes requerimentos de desempenho, consumo energético, etc.

2.1.3 ARM

A possibilidade de implementar processadores RISC usando hardware mais simples torna-os vantajosos para o uso em dispositivos embebidos, onde a portabilidade e o baixo consumo são fatores desejáveis. A arquitetura ARM é uma arquitetura RISC amplamente usada neste tipo de dispositivos, o que faz com que em termos de quantidade seja das ISA mais usadas [13].

2.1.4 Nios II

O Nios II é um *soft processor* desenvolvido pela Intel que pode ser instanciado em qualquer FPGA Intel.

A principal vantagem do Nios II é a sua flexibilidade, podendo ser configurado em duas versões com desempenhos distintos. Os desenvolvedores de sistemas podem criar núcleos Nios II adaptados às suas necessidades, tendo à sua disponibilidade várias ferramentas pela Intel para automatizar este processo. Isto faz do Nios II o *soft processor* mais usado em FPGA.

A Intel disponibiliza um programa académico, o *Intel FPGA University Program* [14], baseado nas FPGA Intel e no processador Nios II. Neste programa existem tutoriais de introdução às FPGA e às ferramentas de desenvolvimento.

2.2 Simuladores

2.2.1 ViSiMIPS

O ViSiMIPS é um simulador do processador MIPS32 desenvolvido em Java. Na interface deste simulador é possível observar o *pipeline* em diagrama de blocos e os valores à entrada e saída de cada bloco à medida que se simula um programa *assembly* introduzido pelo utilizador. O utilizador controla a simulação avançando ou recuando ciclos de relógio. Esta ferramenta é usada para ajudar à compreensão do conceito de *pipelining*.

2.2.2 SPIM

O SPIM é um simulador do processador MIPS32 que permite a execução de programas *assembly*. Está disponível em terminal bem como numa interface gráfica. É possível carregar um programa *assembly*

a partir de um ficheiro. Uma vez carregado, é possível corrê-lo e ver as alterações feitas aos registos e à memória, bem como a saída do programa numa consola.

Este simulador também oferece funcionalidades de depuração, como correr o programa passo a passo e definir pontos de paragem. É também possível visualizar e alterar o conteúdo dos registos e da memória.

2.2.3 MARS

O MARS é um simulador do processador MIPS32. É possível carregar programas *assembly* a partir de um ficheiro, editá-los e assemblá-los. É possível visualizar e alterar o conteúdo dos registos e da memória. A velocidade da simulação pode ser controlada e também é possível definir pontos de paragem e fazer a execução passo a passo, avançando ou recuando ciclos de relógio.

2.2.4 WinMIPS64

O WinMIPS64 é um simulador do processador MIPS64. Os programas *assembly* são carregados a partir de um ficheiro e existem alguns programas de demonstração incluídos. Na interface gráfica é possível visualizar o *pipeline*, ver e editar o conteúdo dos registos e da memória. É possível executar o programa passo a passo, um ou quatro ciclos de relógio de cada vez. Também é possível executar até ao próximo ponto de paragem definido pelo utilizador.

2.2.5 *Simple 8-bit Assembler Simulator*

O *Simple 8-bit Assembler Simulator* [15] é um simulador *online* desenvolvido em JavaScript. Trata-se do simulador de um processador de 8 bits baseado na arquitetura x86, com 4 registos genéricos e com acesso a 256 bytes de memória. Tem ainda uma saída baseada numa consola que está mapeada em memória, ou seja, para escrever para a consola basta escrever para uma posição específica da memória. Este simulador é capaz de correr programas numa linguagem de *assembly* baseada na linguagem do assemblador NASM [16].

A interface gráfica do simulador permite introduzir numa caixa de texto um programa em *assembly* e assemblá-lo. Após a assemblagem é possível ver as etiquetas definidas pelo programa e o conteúdo da memória. A simulação pode ser feita passo a passo ou a uma velocidade de relógio definida pelo utilizador. Durante a simulação é possível ver as alterações aos valores da memória e dos registos, bem como a saída na consola.

2.2.6 Simulador do P3

O simulador do P3 [17] permite correr programas a partir de ficheiros executáveis do P3. Trata-se de uma aplicação desenvolvida em Java com uma interface gráfica onde é possível observar e alterar o conteúdo dos registos e da memória. Quando se carrega um ficheiro executável do P3, é mostrado o programa desassemblado. A simulação corre continuamente, ou até um ponto de paragem definido pelo

utilizador. Além disso, é possível executar passo a passo, um ciclo de relógio ou uma instrução de cada vez.

A saída do programa pode ser observada na janela de texto disponibilizada no simulador, e também na janela da placa que representa os periféricos disponíveis na placa e permite a interação com estes.

2.2.7 *P3JS Assembler and Simulator*

O P3JS [18] é um assemblador e simulador não oficial para o P3 que está disponível *online* numa plataforma *web*. Também está disponível em linha de comandos na sua versão *offline*.

A interface gráfica do P3JS possui um editor integrado de *assembly* no qual é possível criar programas *assembly*. Na mesma plataforma é possível assemblar e carregar os programas no simulador. No simulador é possível visualizar o conteúdo dos registos e das memórias. A velocidade de simulação é configurável, é possível definir pontos de paragem, e é possível executar passo a passo um ciclo de relógio de cada vez. À semelhança do simulador do P3, existe uma representação gráfica interativa dos periféricos da placa onde é possível definir e observar as entradas e saídas do programa.

2.3 Conclusões

Após análise das funcionalidades destes simuladores, ficaram patentes algumas funcionalidades essenciais que foram tidas em conta para a realização do trabalho desta dissertação, nomeadamente a visualização do conteúdo dos registos e da memória, e a possibilidade de criar pontos de paragem e de executar programas passo a passo. Outras funcionalidades também se revelaram importantes, como a existência de um editor de *assembly* integrado, bem como a possibilidade de assemblar ou executar diretamente os programas *assembly*, uma vez que permitem desenvolver e depurar os programas de forma mais expedita. A disponibilização de uma versão *online* permite aceder ao simulador através de um navegador de Internet sem necessitar descarregar ou instalar ficheiros, o que constitui um entrave à adoção e utilização destas ferramentas, tal como se observou com o Simulador do P3, por vezes preterido a favor do P3JS. Alguns simuladores do processador MIPS permitem a visualização do *pipeline*, o que é uma funcionalidade útil para ajudar na compreensão deste conceito e do seu funcionamento.

Na Tabela 2.1 estão sumarizadas as funcionalidades dos simuladores analisados bem como as funcionalidades que foram implementadas no simulador do P4.

Tabela 2.1. Comparação das funcionalidades de diversos simuladores.

Características \ Simulador	P4	P3JS	P3	MARS	ViSiMIPS	SPIM	WinMIPS64	Simple 8-bit
Carregar programa a partir de ficheiro	✓	✗	✓	✓	✗	✓	✓	✗
Editor de *assembly* integrado	✓	✓	✗	✓	✗	✗	✗	✓
Execução passo a passo	✓	✓	✓	✓	✓	✓	✓	✓
Execução passo a passo para trás	✗	✗	✗	✓	✓	✗	✗	✓
Pontos de paragem	✓	✓	✓	✓	✗	✓	✓	✗
Frequência de relógio configurável	✓	✓	✗	✓	✗	✗	✗	✓
Visualizar o *pipeline*	✗	✗	✗	✗	✓	✗	✓	✗
Visualizar conteúdo dos registos	✓	✓	✓	✓	✓	✓	✓	✓
Visualizar conteúdo da memória	✓	✓	✓	✓	✗	✓	✓	✓
Editar conteúdo dos registos e da memória	✗	✗	✓	✓	✗	✗	✓	✗
Versão *online*	✓	✓	✗	✗	✗	✗	✗	✓
Versão *offline*	✓	✓	✓	✓	✓	✓	✓	✗

3 Tecnologias Utilizadas

Este trabalho está dividido em duas partes. A primeira parte corresponde à especificação e implementação do P4 em hardware. A segunda parte corresponde à criação de um ambiente de desenvolvimento com ferramentas de apoio fundamentais ao sistema didático desenvolvido: um editor de *assembly*, assemblador, depurador, simulador e ferramenta de interface com a FPGA.

A implementação em hardware do P4 foi concretizada em FPGA, enquanto que as ferramentas de apoio foram desenvolvidas com tecnologias *web*: HTML5, JavaScript e CSS.

3.1 Plataforma de Desenvolvimento da Implementação em Hardware

A plataforma escolhida para a concretização em hardware do P4 foi a placa *Terasic DE10-Lite* [19] baseada numa FPGA *MAX 10* [20] da Intel, ilustrada na Figura 3.1.

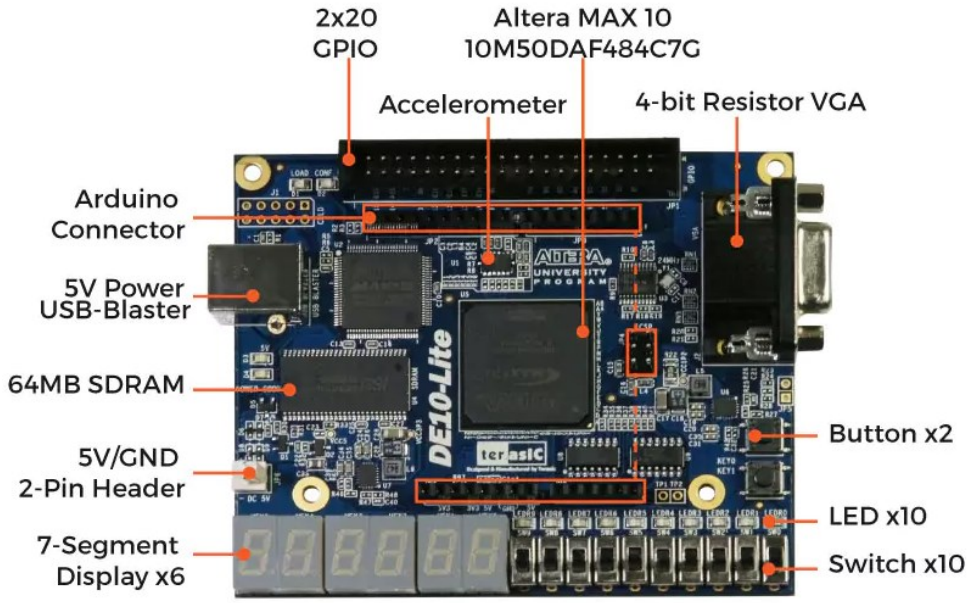


Figura 3.1. Fotografia da face superior da placa *Terasic DE10-Lite*.

Esta placa foi escolhida pelo facto de ter um custo acessível para equipar o laboratório, integrar vários periféricos úteis para um sistema didático: *LEDs*, interruptores, mostradores, botões de pressão e ainda permitir acoplar placas de expansão compatíveis com *Arduino Uno* [21], oferecendo a possibilidade de adicionar novos periféricos. Além do mais, a tecnologia FPGA utilizada nesta placa permite outros usos pedagógicos, como a concretização de circuitos lógicos criados pelos próprios estudantes, pelo que poderá ser usada não só no contexto do ensino de Arquitetura de Computadores mas também de outras unidades curriculares como Sistemas Digitais.

As principais características desta placa são:

- FPGA *Intel MAX 10 10M50DAF484C7G*.
  - 50 000 elementos lógicos programáveis.
  - 1638 Kb de memória organizada em blocos de 9 Kb (M9K).
- 10 *LEDs* vermelhos.
- 10 interruptores.
- 2 botões de pressão.
- 6 mostradores hexadecimais de 7 segmentos.
- 64 MB de memória SDRAM (*bus* de 16 bits).
- Saída VGA de 4 bits.
- Acelerómetro de 3 eixos.
- Alimentação, comunicação e reprogramação via USB.

Para disponibilizar um conjunto maior de periféricos, juntou-se a esta solução uma placa de expansão, ilustrada na Figura 3.2, que disponibiliza seis botões de pressão adicionais e um mostrador alfanumérico LCD de duas linhas por 16 caracteres.

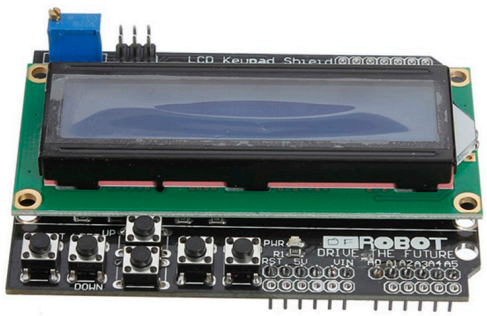


Figura 3.2. Fotografia da face superior da placa de expansão com 6 botões de pressão e mostrador alfanumérico LCD de duas linhas por 16 caracteres.

Outro periférico que se pretendia disponibilizar nesta solução é um teclado PS/2. Para tal foi desenvolvido um protótipo de uma segunda placa de expansão com uma porta USB por forma a permitir a ligação de teclados que suportem o protocolo PS/2, explicada em maior detalhe no Capítulo 6.10.

Uma vantagem da placa DE10-Lite é o facto de ser facilmente programada através do software *Quartus Prime Lite* disponibilizado pela *Intel*. A maioria dos componentes do P4 foram especificados no *Quartus* usando a ferramenta *Schematic Design*. Esta ferramenta fornece um ambiente gráfico onde é possível especificar circuitos lógicos através de um diagrama de blocos. Neste caso, um circuito é criado através de portas lógicas, registos ou outros componentes, e as respetivas ligações entre estes.

A utilização de diagramas de blocos permite criar facilmente circuitos simples que dependam apenas da interligação de vários blocos, criando-se assim uma estrutura hierárquica fácil de visualizar. Por outro lado, para circuitos que implementam lógica sequencial mais complexa, foi utilizada a linguagem *Very High Speed Integrated Circuit Hardware Description Language* (VHDL), uma linguagem de descrição de

hardware, pois permite especificar estes circuitos com um maior nível de abstração e com maior foco na sua funcionalidade.

Em comparação com a plataforma do P3, a do P4 consiste em hardware mais recente, mais compacto, com interfaces atuais, disponibiliza novos periféricos como o acelerómetro, bem como periféricos melhorados como o terminal que passa a ser a cores e com maior resolução. Ainda assim, no geral a nova plataforma mantém muitas das características da plataforma do P3, o que permite uma utilização em laboratório similar à da plataforma do P3. Em particular, todos os periféricos disponíveis na plataforma do P3 estão também disponíveis na plataforma do P4. Esta similaridade deixa em aberto a possibilidade de uma futura adaptação do P3 para esta nova plataforma, o que iria permitir ter ambos os processadores numa única solução de hardware.

3.2 Plataforma de Desenvolvimento das Ferramentas de Apoio

Para permitir o desenvolvimento de programas para o P4, foram criados um editor de *assembly*, assemblador, depurador e simulador. Foi ainda criada uma ferramenta de interface com a FPGA que permite configurá-la com o P4 e assim carregar programas na memória do P4, bem como depurá-los, sendo ainda possível controlar o funcionamento do processador, como parar a execução de programas, executá-los passo a passo ou executar uma qualquer instrução.

Optou-se por desenvolver estas ferramentas com tecnologias *web*: HTML5, JavaScript e CSS. Estas são tecnologias atuais, em constante desenvolvimento, e para as quais há uma grande disponibilidade de documentação e bibliotecas devido ao seu vasto uso na *web* e cada vez mais noutras plataformas, como em aplicações móveis e para *desktop*. Com estas tecnologias é possível implementar interfaces gráficas e funcionalidades de forma mais simples e rápida do que com outras tecnologias tradicionalmente utilizadas no desenvolvimento de aplicações nativas. Além disso, estas tecnologias não são específicas para uma plataforma, o que significa que as ferramentas podem ser utilizadas em diferentes plataformas.

Para o utilizador a principal vantagem é o facto de poder utilizar as ferramentas *online*, bastando para isso aceder a uma página *web* através de um navegador de Internet, sem ter de instalar software adicional no sistema do utilizador.

As ferramentas de apoio foram também disponibilizadas numa aplicação desenvolvida usando a *framework* NW.js [22]. Assim, é possível utilizar as ferramentas numa aplicação que não requer uma ligação à Internet. Nesta aplicação são disponibilizadas funcionalidades adicionais relativas ao controlo do P4 na FPGA, que não estão disponíveis na versão *web*. As distribuições para as plataformas Linux, Windows e macOS são criadas automaticamente a partir da versão *web* utilizando a ferramenta nwjs-builder-phoenix [23].

Na versão *offline* está também disponível um assemblador para o P4 em linha de comandos, que utiliza a plataforma Node.js [24].

4 Estrutura do P4

Neste capítulo é apresentado o P4, começando com a sua ISA, a sua arquitetura e os seus mecanismos de entradas e saídas. O P4 é um processador RISC com as seguintes características:

- Tem um conjunto de 38 instruções.
- Tem três modos de endereçamento.
- Instruções de tamanho fixo (16 bits).
- Tem três formatos de instrução.
- Cada instrução utiliza no máximo dois modos de endereçamento.
- Tem um conjunto de 8 registos de uso geral.
- Tem um *pipeline* de 4 andares.
- Suporta interrupções.

4.1 Conjunto de Instruções do Processador P4

O P4 possui instruções para realizar operações aritméticas, lógicas, de deslocamento, de controlo, de transferência e genéricas. Na Tabela 4.1 estão listadas as instruções do P4 agrupadas pelo tipo de operação que executam. Cada instrução está explicada em maior detalhe no Anexo A.

Tabela 4.1. Conjunto de instruções do processador P4.

Aritméticas	Lógicas	Deslocamento	Controlo	Transferência	Genéricas
NEG	COM	SHR	BR	MOV	NOP
INC	AND	SHL	BR.*cond*	LOAD	ENI
DEC	OR	SHRA	JMP	STOR	DSI
ADD	XOR	SHLA	JMP.*cond*	MVI	STC
ADDC	TEST	ROR	JAL	MVIH	CLC
SUB		ROL	JAL.*cond*	MVIL	CMC
SUBB		RORC	RTI		
CMP		ROLC	INT		

De seguida estão indicadas as operações realizadas por cada instrução:

Operações Aritméticas

`NEG` Converte o operando para o seu complemento para dois.

`INC` Incrementa o operando.

`DEC` Decrementa o operando.

`ADD` Adiciona dois operandos e guarda o resultado num terceiro operando.

`ADDC` Adiciona dois operandos com transporte e guarda o resultado num terceiro operando.

`SUB` Subtrai dois operandos e guarda o resultado num terceiro operando.

`SUBB` Subtrai dois operandos com empréstimo e guarda o resultado num terceiro operando.

`CMP` Compara dois operandos e apenas atualiza os bits de estado.

Operações Lógicas

`COM` Complementa cada bit do operando.

`AND` Faz a conjunção de dois operandos e guarda o resultado num terceiro operando.

`OR` Faz a disjunção de dois operandos e guarda o resultado num terceiro operando.

`XOR` Faz a disjunção exclusiva de dois operandos e guarda o resultado num terceiro operando.

`TEST` Faz a disjunção de dois operandos e apenas atualiza os bits de estado.

Operações de Deslocamento

`SHR` Deslocamento dos bits do operando uma posição para a direita.

`SHL` Deslocamento dos bits do operando uma posição para a esquerda.

`SHRA` Deslocamento aritmético dos bits do operando uma posição para a direita.

`SHLA` Deslocamento aritmético dos bits do operando uma posição para a esquerda.

`ROR` Rotação dos bits do operando uma posição para a direita.

`ROL` Rotação dos bits do operando uma posição para a esquerda.

`RORC` Rotação dos bits do operando uma posição para a direita com transporte.

`ROLC` Rotação dos bits do operando uma posição para a esquerda com transporte.

Instruções de Controlo

`BR` Salto relativo incondicional.

`BR.cond` Salto relativo condicional.

`JMP` Salto absoluto incondicional.

`JMP.cond` Salto absoluto condicional.

`JAL` Salto absoluto com ligação incondicional.

`JAL.cond` Salto absoluto com ligação condicional.

`RTI` Retorno de rotina de serviço a uma interrupção.

`INT` Chamada de rotina de serviço a uma interrupção.

Operações de Transferência

`MOV` Copia um registo para outro registo.

`LOAD` Carrega um valor da memória para um registo.

`STOR` Guarda o valor de um registo para a memória.

`MVI` Copia uma constante para um registo.

`MVIH` Copia uma constante para o octeto mais significativo do registo.

`MVIL` Copia uma constante para o octeto menos significativo do registo.

Operações Genéricas

`NOP` Não tem qualquer efeito.

`ENI` Ativa o tratamento de interrupções.

`DSI` Inibe o tratamento de interrupções.

`STC` Ativa o bit de transporte.

`CLC` Desativa o bit de transporte.

`CMC` Complementa o bit de transporte.

4.2 Arquitetura do P4

Nesta secção são dados detalhes sobre a arquitetura do P4, nomeadamente o *pipeline*, os registos, bits de estado (*flags*), memórias, modos de endereçamento e formatos de instrução.

4.2.1 O Pipeline do P4

O P4 é estruturalmente organizado num *pipeline* de 4 andares que está ilustrado de forma simplificada na Figura 4.1. Entre cada andar, todos os sinais são guardados em registos. Desta forma os sinais gerados num andar só entram no andar seguinte no ciclo de relógio seguinte. Consegue-se assim ter uma instrução em diferentes fases de processamento:

- 1.º andar: carregamento da instrução.
- 2.º andar: descodificação da instrução e carregamento dos operandos.
- 3.º andar: execução.
- 4.º andar: escrita do resultado.

Em maior detalhe, no *primeiro andar*, onde é feito o carregamento da instrução, encontramos o registo *Program Counter* (PC) cujo valor indica o endereço da instrução a ser executada. Encontramos também a Memória de Programa, que recebe o valor do PC e obtém a instrução (RI) a ser executada. Estes dois sinais, PC e RI, passam ainda por um circuito de controlo responsável pelas funcionalidades de depuração do P4, descritas no Capítulo 5.3.2, e de seguida passam para o andar seguinte do *pipeline*.

No *segundo andar* é feita a descodificação da instrução e o carregamento dos operandos. O descodificador de instruções avalia a instrução e gera sinais de controlo, nomeadamente, os sinais que identificam os registos que serão utilizados como operandos. O banco de registos recebe estes sinais e coloca o valor destes registos à sua saída.

Os multiplexadores, MUX4 e MUX5, que se encontram neste andar, fazem o *forwarding* de dados, uma técnica de otimização do *pipeline* que será explicada em maior detalhe no Capítulo 6.7.

Existe ainda neste andar a unidade de controlo de saltos, que analisa a instrução e o registo de estado para determinar quando deve ocorrer um salto. Para acelerar a resolução dos saltos, existe um somador dedicado ao cálculo do endereço dos saltos relativos. O multiplexador MUX6 seleciona entre o valor do registo PC somado com os 8 bits menos significativos da instrução, para os saltos relativos, e o valor de um registo selecionado pela instrução, para os saltos absolutos.

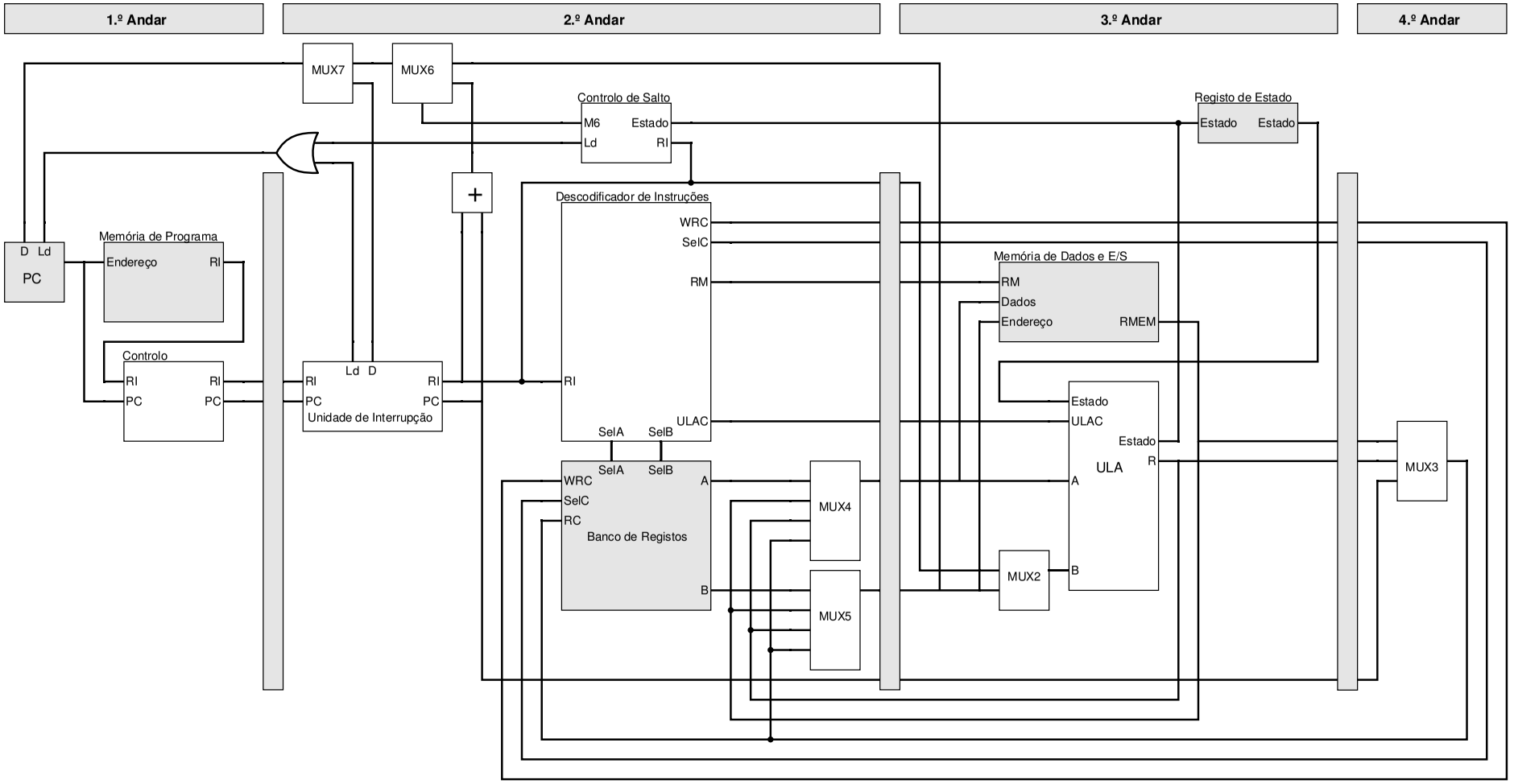


Figura 4.1. Diagrama simplificado do P4.

Para suportar o tratamento das interrupções existe, no segundo andar, a Unidade de Interrupção que controla um multiplexador MUX7 e gera o sinal `Ld` que permite carregar o PC com o endereço da rotina de tratamento da interrupção.

No *terceiro andar* são executadas as operações, englobando não só as operações realizadas pela Unidade Lógica e Aritmética (ULA), como também as operações de leitura ou de escrita na memória de dados e de dispositivos de entrada e de saída. Os componentes principais deste andar são a ULA, a memória de dados e periféricos de entrada e saída, e o registo de estado.

Neste andar, o multiplexador MUX2 permite selecionar entre a saída B do andar anterior e os 8 bits menos significativos da instrução, que permite implementar as instruções `MVI`, `MVIH` e `MVIL`.

No *quarto andar* é feita a escrita do resultado no banco de registos. Neste andar apenas encontramos um multiplexador, o MUX3, que encaminha a fonte dos dados a serem escritos: ULA, entradas/saídas, ou registo PC, para a entrada do banco de registos. Só neste andar o banco de registos recebe os sinais de controlo, relativos à escrita do resultado, que foram gerados pelo descodificador de instruções no segundo andar do *pipeline*.

4.2.2 Registos

O P4 possui 8 registos genéricos de 16 bits, R0 a R7, o *Program Counter* (PC) e o registo de estado. O registo R0 não pode ser alterado e tem sempre valor zero, o PC contém o endereço de memória da instrução a ser executada, e o registo de estado guarda um conjunto de *flags* relevantes ao resultado de operações aritméticas, lógicas e de deslocamento, i.e., se o resultado foi zero, negativo, se houve transporte, ou excesso. Todos os registos têm valor zero após a reinicialização do P4.

4.2.3 Bits de Estado

Cada *flag* é guardada no registo de estado como um bit, cujo o valor é 1 quando essa flag está ativa e 0 quando está desativada. Na Tabela 4.2 mostra-se a estrutura do registo de estado.

Tabela 4.2. Bits do registo de estado.

4	3	2	1	0
Interrupção (E)	Zero (Z)	Transporte (C)	Negativo (N)	Excesso (O)

O significado de cada bit de estado está descrito na seguinte lista:

- O: *overflow* ou excesso. É definido pelas operações aritméticas e indica que o resultado está incorreto quando interpretado em complemento para 2 por exceder o número de bits disponíveis no registo de destino.
- N: *negative* ou sinal. É definido pelas operações aritméticas, lógicas e de deslocamento. Indica que o resultado da operação corresponde a um número negativo quando interpretado em complemento para 2, ou seja, o bit mais significativo tem valor 1.

- C: *carry* ou transporte. É definido pelas operações aritméticas e de deslocamento. Nas operações aritméticas indica que o resultado pode estar incorreto quando interpretado como um número sem sinal por exceder o número de bits disponíveis no registo de destino. Nas operações de deslocamento corresponde ao bit deslocado para fora por estas operações. Pode ser modificado através das instruções `STC`, `CLC` e `CMC`.
- Z: zero. É definido pelas operações aritméticas, lógicas e de deslocamento, e indica que o resultado da operação foi zero.
- E: *enable interrupts*. Qualquer rotina de serviço de uma interrupção só será chamada quando este bit tiver o valor 1. Este bit pode ser modificado através das instruções `ENI` e `DSI`.

4.2.4 Memória

O P4 possui duas memórias separadas para o programa e para os dados. Cada memória consegue guardar 32768 palavras de 16 bits. Os endereços superiores a 7FFFh estão mapeados para 15 bits ignorando o bit mais significativo. Por exemplo, o endereço 8000h está mapeado para 0000h.

4.2.5 Modos de Endereçamento

O P4 possui três modos de endereçamento: endereçamento por registo, endereçamento imediato e endereçamento indireto por registo.

Endereçamento por registo.
É o único modo possível para as instruções aritméticas, lógicas e de deslocamento. O operando corresponde ao registo indicado. Por exemplo: `ADD R1, R2, R3`.

Endereçamento imediato.
Este modo é utilizado em instruções de carregamento de constantes em registos, instruções de salto relativo e na instrução `INT`. O operando corresponde à constante indicada. Por exemplo: `BR 127`.

Endereçamento indireto por registo.
Este modo é utilizado pelas instruções `LOAD` e `STOR`. O operando corresponde à posição de memória cujo endereço é dado pelo valor do registo indicado. Por exemplo: `LOAD R1, M[R2]`.

4.2.6 Formatos de Instrução

Todas as instruções têm 16 bits e podem ter um dos três formatos possíveis: o formato I, utilizado pelas instruções aritméticas, lógicas e de deslocamento; o formato II, utilizado pelas instruções de controlo; e o formato III, utilizado pelas restantes instruções. Tanto o formato II como o formato III dividem-se em dois subformatos, que determinam o significado dos 8 bits menos significativos da instrução. A Tabela 4.3 mostra os formatos de instrução utilizados pelos vários tipos de instruções. Note-se que nem todas as instruções utilizam todos os campos do seu formato.

Tabela 4.3. Formato das instruções do processador P4.

Formato	Bits da Instrução																Tipos de Operações
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
I	1	0	RC			OP					RA			RB			Aritméticas, Lógicas e Deslocamento
II.a	0	0	0	X		Cond			Destino								Controlo
II.b	0	0	1	OP		Cond			X					RB			
III.a	0	1	RC			OP			X		RA			RB			Transferência
III.b	1	1	RC			OP			Constante								
	0	1	RC			OP			Constante								Interrupção
	1	1	RC			OP			X								Genéricas

Os dois bits mais significativos da instrução, 15 e 14, indicam o formato da instrução. O subformato é indicado pelo terceiro bit mais significativo, 13, no caso do formato II; e pelo bit 10, o mais significativo do código de operação, `OP`, no caso do formato III.

O campo `RC` indica o registo para a escrita do resultado, e os campos `RA` e `RB` indicam os registos utilizados para os operandos. Em algumas instruções o operando é uma constante codificada nos 8 bits menos significativos. Nas instruções correspondentes a saltos condicionais, a condição é codificada no campo `Cond`, de acordo com a Tabela 4.4.

Esta nomenclatura é diferente daquela utilizada no livro [1], em que os formatos II.a, II.b, III.a e III.b são chamados de formatos II, III, IV e V, respetivamente.

O livro não especifica o formato a ser utilizado nas instruções genéricas, nem previa a existência das instruções relativas às interrupções. Escolheu-se utilizar o formato III.b com códigos de operação em que o bit mais significativo tem valor 1 para as instruções genéricas e relativas às interrupções.

Por questões de normalização, e para facilitar a inicialização, optou-se por codificar a instrução `NOP` como uma instrução com todos os bits a zero. Isto significa que esta instrução é na realidade codificada como um salto relativo que nunca ocorre, o que é conseguido através da criação de uma condição que é sempre falsa e que é codificada como `0000b`. Esta codificação das condições é diferente daquilo que estava previsto inicialmente no livro [1]. A nova codificação das condições de teste passou a ser como indicado na Tabela 4.4.

Tabela 4.4. Codificação das condições de teste.

Condição	Mnemónica	OPCODE	Condição	Mnemónica	OPCODE
Falso		`0000`	Verdadeiro		`0001`
Zero	Z	`0010`	Não-zero	NZ	`0011`
Transporte	C	`0100`	Não-transporte	NC	`0101`
Negativo	N	`0110`	Não-negativo	NN	`0111`
Excesso	O	`1000`	Não-excesso	NO	`1001`
Positivo	P	`1010`	Não-positivo	NP	`1011`

Foi feita outra alteração em relação ao que estava especificado no livro com o intuito de simplificar a descodificação das instruções de transferência, o que é conseguido ao tornar-se o uso dos operandos mais coerente com aquele que é feito nas restantes operações. Neste sentido, o formato III.a passou a utilizar também o campo `RA`, de forma a ser usado pela instrução `STOR` ao invés do campo `RC`, visto o campo `RC` tratar-se de um campo utilizado noutras instruções para indicar o registo onde é feita a escrita do resultado, o que não acontece no caso da instrução `STOR`, que apenas tem de utilizar o valor de dois registos sem os alterar.

4.3 Interrupções

As interrupções indicam ao processador que existe um evento externo ao *pipeline* que necessita de atenção imediata. No P4 existem 9 fontes possíveis de interrupções: o pressionar de cada um dos 7 botões de pressão, o pressionar de uma tecla no teclado PS/2 e o final da contagem do temporizador.

Quando é gerada uma interrupção, é testado se o bit E do registo de estado e o bit correspondente na máscara de interrupções têm valor 1, nesse caso, o processador interrompe a execução do programa e começa a executar a rotina de serviço dessa interrupção, que deverá estar no endereço calculado pela soma do endereço 7F00h com o valor do vetor de interrupção com um deslocamento de 4 bits para a esquerda. Por exemplo, a rotina de serviço às interrupções do temporizador, que tem um vetor de interrupção com valor Fh, deverá ser colocada no endereço 7FF0h (7F00h + F0h). Este deslocamento significa que cada rotina de interrupção pode utilizar um total de 16 posições de memória sem se sobrepor a outras rotinas de interrupção.

O valor dos vetores de interrupção e endereços das rotinas de serviço correspondentes estão indicados na Tabela 4.5.

Tabela 4.5. Vetores de interrupção e endereços das rotinas de serviço.

Origem	*Key0*	*Key1*	*Right*	*Up*	*Down*	*Left*	*Select*	Teclado	Temporizador
Vetor	0h	1h	2h	3h	4h	5h	6h	7h	Fh
Endereço	7F00h	7F10h	7F20h	7F30h	7F40h	7F50h	7F60h	7F70h	7FF0h

4.4 Entradas e Saídas

Os endereços de FF00h a FFFFh estão reservados para a interface com os periféricos, que é feita através de portos que estão mapeados nestes endereços de memória. A Tabela 4.6 mostra os portos disponíveis no P4 e o respetivo mapeamento em memória. As escritas para portos de leitura ou não atribuídos são ignoradas, enquanto que as leituras de portos de escrita ou não atribuídos retornam o valor FFFFh.

Tabela 4.6. Mapeamento em memória dos dispositivos de entrada e saída.

Endereço	Dispositivo	Função	Leitura	Escrita
FFFF	Terminal	Ler última tecla premida	X	
FFFE	Terminal	Escrever caráter		X
FFFD	Terminal	Testar se houve tecla premida	X	
FFFC	Terminal	Posicionar cursor		X
FFFB	Terminal	Definir cor		X
FFFA	Máscara de Interrupções	Ler ou definir vetores de interrupção ativados	X	X
FFF9	Interruptores	Ler valor dos interruptores	X	
FFF8	LED	Definir estado ligado/desligado dos LED		X
FFF7	Temporizador	Iniciar/parar ou ler estado	X	X
FFF6	Temporizador	Ler ou definir valor da contagem	X	X
FFF5	Mostrador LCD	Escrever caráter		X
FFF4	Mostrador LCD	Posicionar cursor		X
FFF3	Mostrador 3	Escrever valor no mostrador 3		X
FFF2	Mostrador 2	Escrever valor no mostrador 2		X
FFF1	Mostrador 1	Escrever valor no mostrador 1		X
FFF0	Mostrador 0	Escrever valor no mostrador 0		X
FFEF	Mostrador 5	Escrever valor no mostrador 5		X
FFEE	Mostrador 4	Escrever valor no mostrador 4		X
FFED	Acelerómetro Z	Ler aceleração no eixo Z	X	
FFEC	Acelerómetro Y	Ler aceleração no eixo Y	X	
FFEB	Acelerómetro X	Ler aceleração no eixo X	X	

4.4.1 Terminal (Ecrã e Teclado)

O terminal serve para mostrar ao utilizador informação sob a forma de texto e receber as teclas pressionadas por ele. O terminar disponível no P4 permite mostrar uma janela de texto de 45 linhas por 80 colunas num ecrã VGA, ligado à saída VGA da placa, e a entrada é obtida a partir de um teclado compatível com PS/2 ligado à porta USB da placa de expansão.

A leitura do porto de estado, FFFDh, permite saber se houve alguma tecla premida desde a última leitura do porto de leitura. Caso tenha havido, o valor lido é 1, caso contrário, é 0.

Ao fazer a leitura do porto de leitura, FFFFh, obtém-se o valor da última tecla premida, ou zero caso nenhuma tecla tenha sido premida desde a última leitura. O valor da tecla é dado pelo valor ASCII estendido *Code Page 437* do caráter correspondente à tecla, ou no caso de teclas sem correspondência a um caráter, o valor é dado pela correspondência indicada na Tabela 6.2. Os bits 8, 9 e 10 do valor indicam o estado das teclas Shift, Ctrl e Alt, respetivamente, tal como é explicado em maior detalhe no Capítulo 6.10.

A escrita para o porto de escrita, FFFEh, resulta na impressão de um caráter na janela de texto na posição atual do cursor, fazendo-o deslocar-se para a posição seguinte, da esquerda para a direita e de cima para baixo, voltando à posição inicial após a última posição visível na janela de texto.

O caráter impresso é determinado pelos 8 bits menos significativos do valor escrito para este porto e pela sua correspondência em ASCII estendido *Code Page 437*. A Tabela 4.7 mostra todos os caracteres que fazem parte desta codificação e o seu valor equivalente em Unicode, indicado em hexadecimal por baixo de cada caráter.

A escrita para o porto de controlo, FFFCh, define a posição do cursor. Os 8 bits mais significativos do valor escrito determinam a linha, e os 8 bits menos significativos determinam a coluna. O valor da linha deverá estar entre 0 e 44, enquanto que o valor da coluna deverá estar entre 0 e 79. Outra função deste porto é permitir alternar entre a resolução de 1920x1080 e a resolução 1280x960, o que é feito, respetivamente, através da escrita dos valores FF00h e FF01h para este porto.

Tabela 4.8. Formato dos comandos para o controlo da posição do cursor no terminal.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Linha 0 – 44								Coluna 0 – 79							

A escrita para o porto de cor, FFFBh, define a cor do caráter e do fundo nas escritas subsequentes. Os 8 bits mais significativos determinam a cor do fundo, enquanto que os 8 bits menos significativos determinam a cor do caráter.

Na palete de cores predefinida os bits de cor estão codificados como RGB, em que R corresponde ao valor da intensidade da componente vermelha, G corresponde à intensidade da componente verde e B à intensidade da componente azul. A intensidade das componentes vermelha e verde estão codificadas em 3 bits cada, enquanto a componente azul está codificada, com menor resolução, em 2 bits, uma vez que a menor resolução da cor azul não é tão perceptível para o olho humano como as restantes [25]. A título de exemplo, para um fundo laranja, codificado como 11101100b (ECh), com caracteres de cor azul, codificada como 00000011b (03h), deve ser escrito o valor EC03h para o porto de cor.

Tabela 4.9. Formato dos comandos para o controlo da cor do texto e do fundo do terminal.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Vermelho			Verde			Azul		Vermelho			Verde			Azul	
Fundo								Texto							

4.4.2 Máscara de Interrupções

A máscara de interrupções permite ativar ou desativar cada um dos 9 vetores de interrupção. Cada bit da máscara de interrupções corresponde a um vetor de interrupção. Quando o valor de um bit é 1, o vetor de interrupção correspondente fica ativado, enquanto que o valor 0 desativa o vetor de interrupção correspondente. Esta correspondência está indicada na Tabela 4.10.

A escrita para o porto FFFAh permite definir o valor para a máscara de interrupções. Por exemplo, a escrita do valor 8000h para este porto faz com que apenas o vetor 15 fique ativo, que corresponde às

Tabela 4.7. ASCII Estendido *Code Page* 437.

\	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0000	☺ 263A	☻ 263B	♥ 2665	♦ 2666	♣ 2663	♠ 2660	• 2022	◘ 25D8	○ 25CB	◙ 25D9	♂ 2642	♀ 2640	♪ 266A	♫ 266B	☼ 263C
1	► 25BA	◄ 25C4	↕ 2195	‼ 203C	¶ 00B6	§ 00A7	▬ 25AC	↨ 21A8	↑ 2191	↓ 2193	→ 2192	← 2190	∟ 221F	↔ 2194	▲ 25B2	▼ 25BC
2	0020	! 0021	” 0022	# 0023	$ 0024	% 0025	& 0026	’ 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
3	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
4	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
5	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
6	‘ 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
7	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	⌂ 2302
8	Ç 00C7	ü 00FC	é 00E9	â 00E2	ä 00E4	à 00E0	å 00E5	ç 00E7	ê 00EA	ë 00EB	è 00E8	ï 00EF	î 00EE	ì 00EC	Ä 00C4	Å 00C5
9	É 00C9	æ 00E6	Æ 00C6	ô 00F4	ö 00F6	ò 00F2	û 00FB	ù 00F9	ÿ 00FF	Ö 00D6	Ü 00DC	¢ 00A2	£ 00A3	¥ 00A5	Pts 20A7	ƒ 0192
A	á 00E1	í 00ED	ó 00F3	ú 00FA	ñ 00F1	Ñ 00D1	ª 00AA	º 00BA	¿ 00BF	⌐ 2310	¬ 00AC	½ 00BD	¼ 00BC	¡ 00A1	« 00AB	» 00BB
B	▒ 2591	▒ 2592	▓ 2593	| 2502	┤ 2524	╡ 2561	╢ 2562	╖ 2556	╕ 2555	╣ 2563	║ 2551	╗ 2557	╝ 255D	╜ 255C	╛ 255B	┐ 2510
C	└ 2514	┴ 2534	┬ 252C	├ 251C	─ 2500	┼ 253C	╞ 255E	╟ 255F	╚ 255A	╔ 2554	╩ 2569	╦ 2566	╠ 2560	═ 2550	╬ 256C	╧ 2567
D	╨ 2568	╤ 2564	╥ 2565	╙ 2559	╘ 2558	╒ 2552	╓ 2553	╫ 256B	╪ 256A	┘ 2518	┌ 250C	█ 2588	▄ 2584	▌ 258C	▐ 2590	▀ 2580
E	α 03B1	ß 00DF	Γ 0393	π 03C0	Σ 03A3	σ 03C3	µ 00B5	τ 03C4	Φ 03A6	Θ 0398	Ω 03A9	δ 03B4	∞ 221E	φ 03C6	ε 03B5	∩ 2229
F	≡ 2261	± 00B1	≥ 2265	≤ 2264	⌠ 2320	⌡ 2321	÷ 00F7	≈ 2248	° 00B0	∙ 2219	· 00B7	√ 221A	ⁿ 207F	² 00B2	■ 25A0	00A0

Tabela 4.10. Correspondência entre os bits da máscara de interrupções e vetores de interrupção.

Bit	15	7	6	5	4	3	2	1	0
Origem	Temporizador	Teclado	*Select*	*Left*	*Down*	*Up*	*Right*	*Key1*	*Key0*
Vetor	Fh	7h	6h	5h	4h	3h	2h	1h	0h

interrupções originadas pelo temporizador. A leitura do porto FFFAh obtém o valor atual da máscara de interrupções.

4.4.3 Interruptores

Existem 10 interruptores dispostos horizontalmente na parte inferior direita da placa, identificados como SW0 a SW9. Ao fazer a leitura do porto FFF9h obtém-se o estado dos interruptores, indicado pelos dez bits menos significativos do valor lido. Quando o interruptor estiver para cima, o bit correspondente tem valor 1, quando estiver para baixo, o bit correspondente tem valor 0. O interruptor mais à direita corresponde ao bit menos significativo. Os seis bits mais significativos têm sempre valor 0.

4.4.4 *LEDs*

Por cima dos interruptores, na parte inferior direita da placa, existem 10 *LEDs* vermelhos, identificados com LEDR0 a LEDR9. A escrita para o porto FFF8h permite definir o estado ligado ou desligado de cada um dos *LEDs* através dos dez bits menos significativos do valor escrito. Cada *LED* fica ligado quando o bit correspondente tem valor 1, e fica desligado quando o bit correspondente tem valor 0. O *LED* mais à direita corresponde ao bit menos significativo. Os seis bits mais significativos são ignorados.

4.4.5 Temporizador

É possível definir um temporizador que gera uma interrupção após um intervalo de tempo real definido pelo utilizador.

A escrita para o porto FFF7h permite controlar o funcionamento do temporizador. Quando o bit menos significativo do valor escrito é 1, o temporizador inicia a contagem decrescente, caso seja 0, o temporizador pára a contagem. Os restantes bits são ignorados. A leitura deste porto devolve o valor 1 caso o temporizador esteja em contagem e 0 caso contrário.

O valor escrito para o porto FFF6h define o valor da contagem, que corresponde à duração do temporizador em décimas de segundo. A leitura deste porto devolve o valor atual da contagem, ou seja, as décimas de segundo restantes para o final da contagem.

4.4.6 Mostrador Alfanumérico LCD

Na parte superior esquerda da placa, numa placa de expansão, existe um mostrador alfanumérico LCD que permite mostrar uma janela de texto de duas linhas por 16 caracteres.

A escrita para o porto de escrita, FFF5h, resulta na impressão de um caráter na janela de texto na posição atual do cursor, fazendo o cursor deslocar-se para a posição seguinte, da esquerda para a direita e de cima para baixo, voltando à posição inicial após a última posição.

O caráter a ser impresso é definido pelos 8 bits menos significativos do valor escrito para este porto e a sua correspondência em ASCII. Para valores superiores a 127, o caráter correspondente está definido no *datasheet* do mostrador alfanumérico LCD Hitachi modelo HD44780U (ROM Code: A00) [26].

A escrita para o porto de controlo, FFF4h, permite definir a posição do cursor, o estado de ligado ou desligado do mostrador, bem como limpar o seu conteúdo. Estas ações são determinadas pelos bits do valor escrito, como indicado pela Tabela 4.11.

Tabela 4.11. Formato dos comandos para o controlo do mostrador alfanumérico LCD.

Bit	Ação
15	Liga (1) ou desliga (0) o mostrador alfanumérico LCD.
5	Limpa o conteúdo do mostrador alfanumérico LCD (1).
4	Posiciona o cursor na primeira (0) ou segunda linha (1).
3 a 0	Posiciona o cursor na coluna especificada (0 a 15).

4.4.7 Mostradores Hexadecimais de 7 Segmentos

No canto inferior esquerdo da placa existem, dispostos horizontalmente, seis mostradores hexadecimais de 7 segmentos que são controlados individualmente pelo processador para formar dígitos hexadecimais.

A escrita nos portos FFEEh a FFF3h permite definir o valor hexadecimal, de 0h a Fh, mostrado pelo mostrador correspondente a cada porto, de acordo com a Tabela 4.6. O valor mostrado é determinado pelos 4 bits menos significativos do valor escrito para este porto, os restantes bits são ignorados. O mostrador 0 situa-se mais à direita, o mostrador 1 situa-se à esquerda do mostrador 0, e assim sucessivamente.

4.4.8 Acelerómetro

A placa do P4 possui um acelerómetro de três eixos perpendiculares, x, y e z, capaz de medir acelerações compreendidas entre -2g e 2g em cada um destes três eixos. Numa vista superior da placa, o eixo x é paralelo à lateral mais comprida da placa, o eixo y é paralelo à lateral mais curta, e o eixo z é perpendicular aos anteriores.

A leitura dos portos FFEBh, FFECh e FFEDh, permite obter o valor da aceleração medido pelo acelerómetro da placa para o eixo x, y e z, respetivamente. O valor de 16 bits deverá ser interpretado em complemento para 2. Os valores 8000h e 7FFFh correspondem às acelerações de -2g e 2g, respetivamente, e os valores FF00h e 00FFh correspondem às acelerações de -1g e 1g, respetivamente.

4.4.9 Botões de Pressão

A placa do P4 possui um total de oito botões de pressão, dois no lado direito da placa, com as etiquetas KEY0 e KEY1, os restantes estão por cima dos mostradores hexadecimais de 7 segmentos, na parte inferior da placa de expansão e possuem as etiquetas *Select*, *Left*, *Down*, *Up*, *Right* e *Reset*.

Estes botões não estão mapeados em memória. Ao pressionar cada um destes botões é gerada uma interrupção correspondente a esse botão, de acordo com a Tabela 4.5, à exceção do botão Reset que reinicializa o P4, colocando todos os registos a 0, incluindo o PC.

5 Implementação em Software

Os programas em *assembly* podem ser carregados, editados e assemblados numa interface gráfica disponível numa plataforma *web*, que foi desenvolvida em JavaScript, HTML5 e CSS, e que está acessível através de um navegador de Internet em `https://web.tecnico.ulisboa.pt/dinismadeira/p4/`.

A partir da versão *online* é possível descarregar a aplicação, que além das funcionalidades da versão *online*, permite ainda interagir com a *FPGA* e assemblar ficheiros *assembly* a partir da linha de comandos. A evocação da aplicação é feita através da execução do ficheiro p4, no caso de um sistema operativo Linux, ou p4.exe num sistema operativo Windows. A utilização da linha de comandos é explicada no Capítulo 5.1.6.

5.1 Assemblador

O assemblador gera código máquina a partir de um programa em *assembly*. Para executar o programa, o código máquina tem de ser colocado na memória de programa do P4. Além disso, o programa *assembly* pode inicializar posições da memória de dados com valores especificados no programa, que têm de ser colocados na memória de dados para o seu correto funcionamento.

Para especificar o conteúdo das memórias, são utilizados ficheiros no formato *Memory Initialization File* (MIF) [27]. Mais concretamente, ao assemblar um programa em *assembly*, o assemblador para o P4 gera um ficheiro prog.mif que especifica o conteúdo da memória de programa, e um ficheiro data.mif que especifica o conteúdo da memória de dados, e comprime estes dois ficheiros num único ficheiro de formato ZIP com extensão p4z.

5.1.1 Constantes

O tamanho máximo das constantes é de 16 bits, ou seja, valores compreendidos entre 0 e 65535 quando interpretadas como números sem sinal, ou entre -32768 e 32767 quando interpretadas como números em complemento para 2.

É possível especificar, no código *assembly*, constantes em decimal, hexadecimal, binário, octal e como caracteres:

- **Decimal:** valor constituído pelos algarismos de 0 a 9, opcionalmente seguido do caráter `d`. São válidos os valores entre `-32768` e `65535`.
- **Hexadecimal:** valor constituído pelos algarismos de 0 a 9 e pelas letras de A a F, seguido do caráter `h`. São válidos os valores entre `-8000h` e `FFFFh`.
- **Binário:** valor constituído pelos algarismos 0 e 1 seguido do caráter `b`. São válidos os valores entre `-100000000000000b` e `1111111111111111b`.
- **Octal:** valor constituído pelos algarismos de 0 a 7 seguido do caráter `o`. São válidos os valores entre `-100000o` e `177777o`.
- **Caráter:** um caráter entre plicas. Esta constante tem como valor o código do caráter na codificação de caracteres ASCII estendido *Code Page 437*, ou Unicode, caso essa codificação tenha sido ativada através da pseudoinstrução `OPT`.

Por norma, as constantes são definidas no início do programa *assembly*, sendo-lhes associada uma etiqueta, o que permite tornar o código mais legível e facilita a atualização dessas constantes posteriormente.

5.1.2 Comentários

Os comentários começam com o caráter "`;`". Todo o texto até ao final da linha é considerado comentário.

5.1.3 Pseudoinstruções

As pseudoinstruções são declarações no programa *assembly* que não correspondem a instruções do P4, mas sim a diretivas para o assemblador. As pseudoinstruções suportadas pelo assemblador para o P4 são:

ORIG

Formato: `ORIG <endereço>`

Descrição: Define a primeira posição de memória em que um bloco de programa ou dados é carregado em memória, podendo ser usado múltiplas vezes de forma a definir blocos em diferentes zonas de memória. Não pode haver zonas sobrepostas.

EQU

Formato: `<símbolo> EQU <const>`

Descrição: Associa o valor `<const>` ao símbolo `<símbolo>`.

WORD

Formato: `<etiqueta> WORD <const>`

Descrição: Reserva uma posição de memória e inicializa-a com o valor `<const>`, associando o endereço dessa posição de memória ao nome `<etiqueta>`.

STR

Formato: `<etiqueta> STR '<texto>'|<const>(,'<texto>'|<const>)*`

Descrição: Reserva posições consecutivas de memória e inicializa-as com o valor de cada caráter de `<texto>` ou com o valor da constante `<const>`. É possível usar um número arbitrário de operandos, separando-os por vírgula, que serão avaliados de forma sequencial da esquerda para a direita. O nome `<etiqueta>` é associado à primeira posição de memória. Dois caracteres `'` consecutivos em `<texto>` são interpretados como um único caráter `'`.

TAB

Formato: `<etiqueta> TAB <const>`

Descrição: Reserva `<const>` posições consecutivas de memória sem as inicializar. O nome `<etiqueta>` é associado à primeira posição de memória.

OPT

Formato: `OPT <opção>`

Descrição: Permite configurar a assemblagem ativando a opção indicada pelo campo `<opção>`. As opções possíveis são:

- `ENABLE_DELAY_SLOTS`: ativa a utilização das *delay slots*. A primeira instrução *assembly* após cada instrução de salto é executada independentemente de o salto se realizar ou não.
- `DISABLE_DELAY_SLOTS`: desativa a utilização das *delay slots*. Esta opção faz com que o assemblador preencha automaticamente as *delay slots* com instruções `NOP`.
- `ASCII`: ativa a utilização da codificação de caracteres ASCII estendido CP437 no assemblador. Os caracteres em `'<texto>'` serão tratados como ASCII estendido CP437. Por exemplo, `'É'` será interpretado como a constante 0090h.
- `UNICODE`: ativa a utilização da codificação de caracteres Unicode no assemblador. Os caracteres em `'<texto>'` serão tratados como Unicode. Por exemplo, `'É'` será interpretado como a constante 00C9h. Esta opção não altera a codificação utilizada pelos periféricos.

As opções `DISABLE_DELAY_SLOTS` e `ASCII` encontram-se ativadas por omissão.

Os símbolos e etiquetas definidos pelas pseudoinstruções `EQU`, `WORD`, `STR` e `TAB` podem ser compostos por caracteres alfanuméricos (A-Z, a-z e 0-9) mais o caráter _, e o primeiro caráter não pode ser um algarismo.

5.1.4 Instruções Assembly

As instruções *assembly* suportadas pelo assemblador para o P4 estão apresentadas no Anexo A por ordem alfabética.

5.1.5 Etiquetas

As instruções podem ser precedidas por "`<etiqueta>:`", que associa `<etiqueta>` à posição de memória onde irá ficar aquela instrução. Com isto, `<etiqueta>` pode ser usada como argumento em instruções de salto, o que permite aliviar o programador da tarefa de calcular o endereço ou o deslocamento necessário para realizar o salto. Por exemplo, se tivermos a instrução:

```
ReadLoop: LOAD R1, M[R2]
```

É possível realizar um salto absoluto, especificando a etiqueta, ao invés de um registo com o endereço da instrução:

```
JMP ReadLoop
```

O assemblador gera o código para carregar previamente o endereço de destino, `ReadLoop`, no registo R7 antes de executar a instrução `JMP R7`. Isto tem como consequência a perda do conteúdo do registo R7.

Também é possível utilizar etiquetas nos saltos relativos. O assemblador calcula automaticamente o deslocamento e usa-o na instrução de salto. Os saltos relativos estão limitados a um máximo de 128 posições para trás (PC-128) e 127 posições para a frente (PC+127) no programa.

As etiquetas podem ainda ser iniciadas pelo caráter ".", tornando-as em etiquetas locais, o que significa que ficam associadas à última etiqueta global definida. Desta forma, é possível haver múltiplas etiquetas locais com o mesmo nome, desde que associadas a diferentes etiquetas globais. É também possível referir uma etiqueta local associada a uma etiqueta global diferente da última etiqueta global definida, concatenando-se a etiqueta global com a etiqueta local. Por exemplo:

```
Rotina1:    ; código
.Loop       ; código
            JMP.Z   .Loop
Rotina2:    ; código
.Loop       ; código
            JMP.Z   .Loop
Rotina3:    ; código
            JMP.Z   Rotina1.Loop
```

Neste fragmento de código, ambas as instruções `JMP.Z .Loop` saltam para a instrução que as precedem, isto porque a etiqueta `.Loop` utilizada nestas instruções está associada à etiqueta global anterior.

Na realidade, a primeira definição de `.Loop` define o símbolo `Rotina1.Loop`, enquanto a segunda definição define o símbolo `Rotina2.Loop`. Desta forma, a instrução `JMP.Z Rotina1.Loop` salta para a instrução correspondente à etiqueta `.Loop` associada à etiqueta global `Rotina1`.

5.1.6 Linha de Comandos

É disponibilizada uma versão em linha de comandos do assemblador para o P4. Para tal é necessário ter o Node.js [24] instalado no sistema.

A assemblagem pela linha de comandos é feita através do comando: `node p4as ficheiro.as`, em que `ficheiro.as` corresponde ao ficheiro de código fonte *assembly* para o P4 a ser assemblado. Este comando gera um ficheiro `ficheiro.p4z`, sendo possível configurar o nome do ficheiro gerado concatenando `-o novo_ficheiro` ao comando, que faz com que seja gerado um ficheiro com o nome `novo_ficheiro.p4z`.

A opção de ver os símbolos definidos pelo programa *assembly* e os respetivos valores pode ser ativada concatenando `--show-refs` ao comando.

5.1.7 Interface Gráfica

A interface gráfica do assemblador foi criada para facilitar a interação do utilizador com o sistema e divide-se em duas zonas: a que contém o editor de texto ③, mostrada na Figura 5.1, e a zona que contém o painel de resultado, mostrada à direita ou por baixo da zona do editor de texto, onde se podem ver as mensagens de erro, depuramento, etiquetas, ficheiro MIF gerado, etc.

No topo da interface encontram-se quatro botões ①:

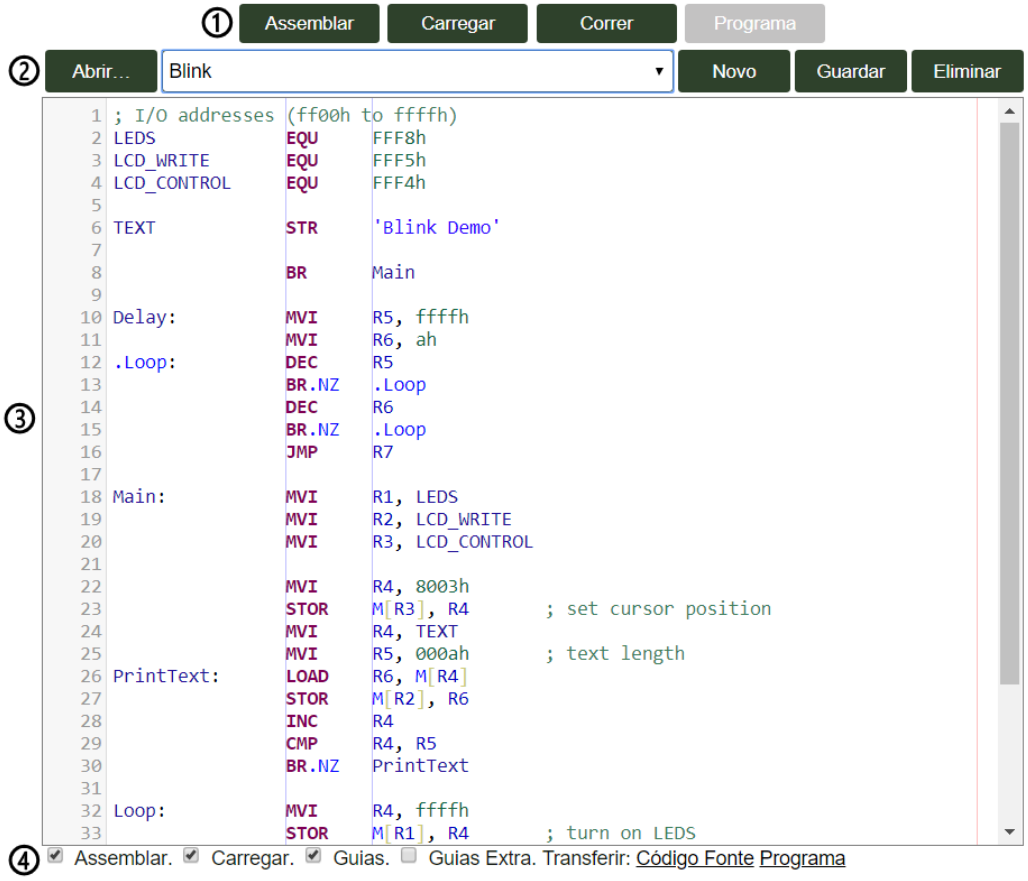


Figura 5.1. Interface gráfica do assemblador.

- Assemblar: faz a assemblagem do programa e carrega-o no simulador.
- Carregar: carrega o programa assemblado no simulador.
- Correr: corre o programa atualmente carregado no simulador.
- Programa: carrega as memórias do P4 com o programa assemblado.

Por cima do editor, encontram-se botões e menus relativamente à gestão de programas ②:

- Abrir…: permite carregar um programa *assembly* a partir de um ficheiro de texto com código *assembly* (.as), ou a partir de um ficheiro executável do P4 (.p4z) que é desassemblado e carregado no editor.
- Lista de programas: menu *dropdown* que permite carregar um programa anteriormente guardado ou de demonstração.
- Novo: limpa o editor de texto e começa a edição de um novo programa.
- Guardar: guarda as alterações ao programa atual.
- Eliminar: elimina o programa atual da lista de programas guardados.

Por baixo do editor, encontram-se algumas opções ④ relativas ao editor:

- Assemblar: ativa a assemblagem automática do programa *assembly* quando o conteúdo do editor é modificado.

- Carregar: ativa o carregamento automático do programa *assembly* no simulador quando o programa é assemblado.
- Guias: Mostrar ou ocultar as guias de indentação principais.
- Guias Extra: Mostrar ou ocultar as guias de indentação secundárias.

Adicionalmente, existem duas ligações para transferir o programa *assembly*:

- Código Fonte: transfere um ficheiro de texto .as com o código *assembly* do programa.
- Programa: transfere um ficheiro .p4z com os ficheiros MIF para inicialização da memória de programa e de dados correspondentes ao programa assemblado.

Por cima do painel de resultado é mostrada uma mensagem a indicar o tempo despendido a assemblar o programa, bem como o número de erros e avisos. Além disso, existe o menu para selecionar o tipo de resultado a mostrar:

- Erros: mostra uma lista de todas as mensagens de erro e avisos ocorridos na assemblagem do programa.
- Depuramento: permite visualizar o código máquina em representação binária gerado por cada linha de código *assembly*.
- Etiquetas: permite visualizar o valor associado a cada etiqueta, em binário, decimal e hexadecimal.
- MIF: permite visualizar o conteúdo do ficheiro MIF correspondente ao programa assemblado.
- BIN: permite visualizar o código máquina gerado em binário.
- HEX: permite visualizar o código máquina gerado em hexadecimal.

5.2 Simulador

O simulador facilita o desenvolvimento de programas para o P4, pois permite controlar a execução dos programas numa interface gráfica que representa o sistema real, e observar os valores em estruturas internas do P4. São suportadas as seguintes funcionalidades:

- Ver o conteúdo da memória de dados, o valor dos registos, do *Program Counter* e os bits de estado.
- Ver o conteúdo da memória de programa e respetivas instruções desassembladas.
- Definir pontos de paragem e fazer execução passo a passo.
- Definir a frequência do relógio na simulação.
- Ver o total de ciclos de relógio simulados.
- Ver a instrução que vai ser executada, bem como a linha *assembly* correspondente a essa instrução.
- Emulação dos periféricos numa interface gráfica interativa semelhante ao sistema real que permite definir e observar as entradas e saídas.

5.2.1 Controlos do Simulador

Para controlar a execução da simulação, existem três botões ① no topo da interface do simulador, que está ilustrada na figura Figura 5.2:

– Iniciar/Parar: permite iniciar ou parar a simulação.
– Passo: permite simular um único ciclo de relógio.
– Reiniciar: simula o reinício do P4.

Figura 5.2. Interface gráfica do simulador.

Por baixo, existe o painel de visualização e controlo do relógio ②. Aqui é mostrada a frequência atual do relógio e o total de ciclos de relógio simulados. Além disso, é possível definir uma frequência máxima para o relógio, desde 1 Hz até ao máximo suportado pelo sistema do utilizador.

5.2.2 Visualização dos Registos

No painel de visualização dos registos ③ é mostrado o valor dos registos do P4 (R1 a R7), bem como do *Program Counter*, Registo de Instrução e Registo de Estado, à medida que a simulação decorre.

Ao clicar neste painel é possível alternar entre várias representações: decimal com sinal, decimal sem sinal, binário e hexadecimal.

5.2.3 Emulação da Placa

O painel de emulação placa ④ é uma interface gráfica que representa o sistema real e permite definir e observar as entradas e saídas disponíveis na placa do P4.

Abaixo listam-se, a partir do topo e da esquerda para a direita, os componentes representados nesta interface:

– Mostrador alfanumérico LCD de duas linhas por 16 caracteres.
– 6 mostradores hexadecimais de 7 segmentos.
– 10 *LEDs*: `LEDR9` a `LEDR0`.
– 10 interruptores: `SW9` a `SW0`.
– 3 controlos para definir a aceleração no eixo do X, Y e Z.
– 7 botões de pressão: `SELECT`, `LEFT`, `UP`, `DOWN`, `RIGHT`, `KEY0` e `KEY1`.

5.2.4 Visualização das Memórias

Os painéis de visualização das memórias encontram-se por baixo do painel de emulação da placa e estão ilustrados na Figura 5.3. Estes permitem observar o conteúdo da memória de programa e da memória de dados.

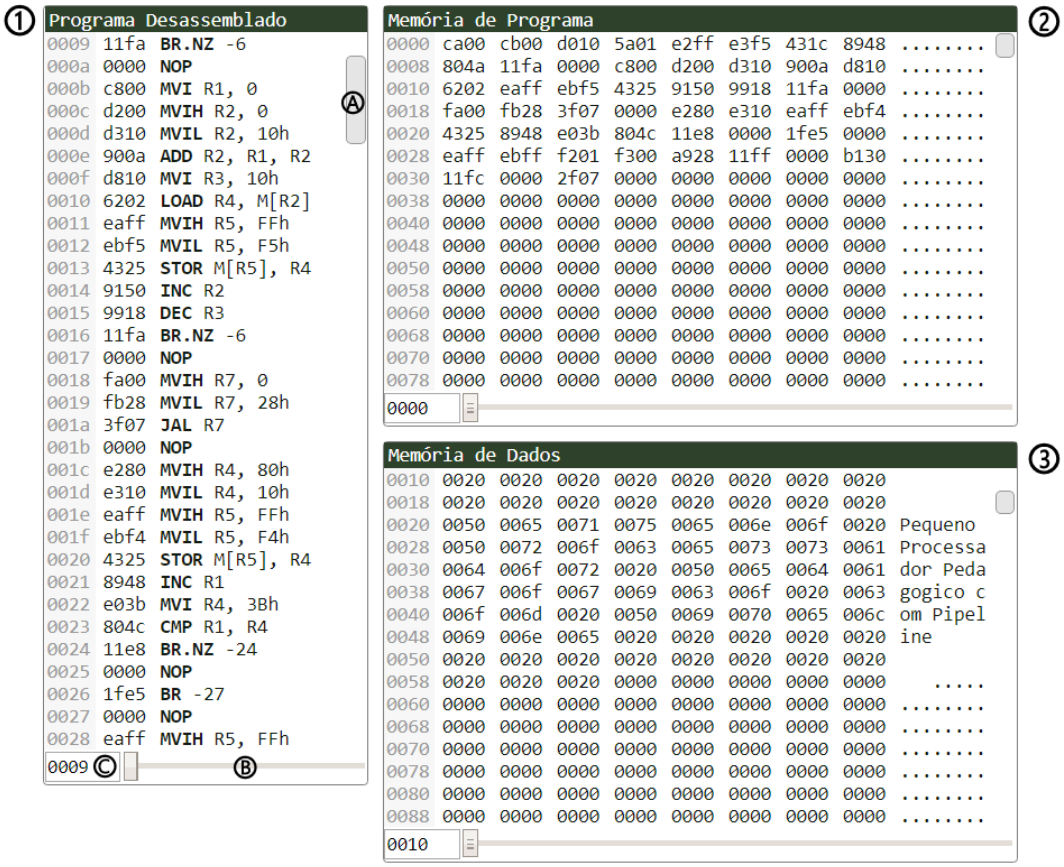


Figura 5.3. Painéis de visualização das memórias.

O painel à esquerda mostra o conteúdo da memória de programa ① em hexadecimal a par das instruções *assembly* correspondentes.

Os dois painéis à direita mostram o conteúdo da memória ② de programa e da memória de dados ③, em hexadecimal, e o seu valor correspondente em ASCII.

Do lado direito de cada memória existe uma barra de deslocamento Ⓐ que permite alternar a visualização dos 8 bits menos significativos de um dado endereço.

Por baixo, existe uma barra de deslocamento Ⓑ que permite definir os 7 bits mais significativos dos endereços a serem mostrados.

Também é possível definir diretamente o endereço base introduzindo-o na caixa de texto Ⓒ que se encontra na parte inferior esquerda do painel.

5.2.5 Terminal (Ecrã e Teclado)

A janela do terminal, ilustrada na Figura 5.4, mostra o conteúdo do terminal tal como seria mostrado num ecrã VGA ligado à placa.

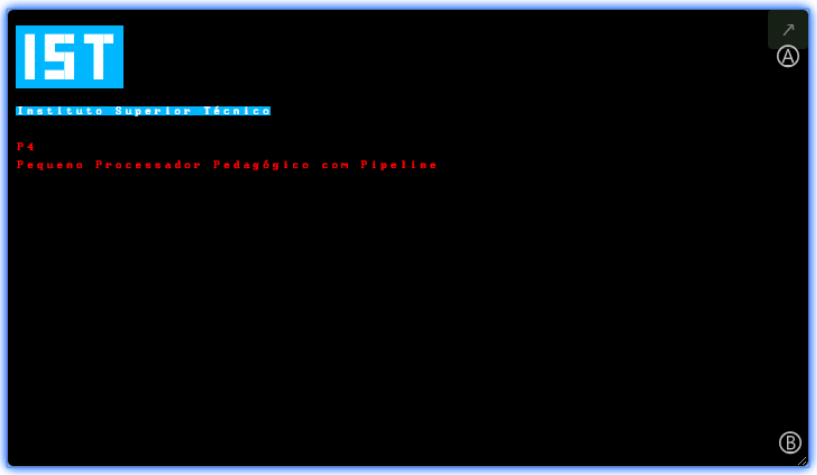


Figura 5.4. Janela do Terminal.

É possível alternar para o modo de ecrã inteiro, que mostra a janela do terminal em todo o ecrã do sistema do utilizador. Para tal, basta clicar na seta visível no canto superior direito da janela do terminal Ⓐ, ou fazer duplo clique em qualquer zona da janela do terminal. O mesmo procedimento deve ser repetido para sair deste modo, ou, carregar na tecla `Escape`.

É também possível redimensionar esta janela clicando no canto inferior direito Ⓑ e, com o botão pressionado, mover o rato na diagonal para aumentar ou diminuir o tamanho da janela.

A simulação do teclado é feita através do teclado do sistema do utilizador. Para tal, a janela do terminal deve estar selecionada, o que é indicado por um sombreado azul, e assim, as teclas premidas no teclado serão replicadas na simulação.

5.2.6 Pontos de Paragem

Uma funcionalidade importante para o depuramento de um programa é a possibilidade de definir pontos de paragem de forma a poder observar-se o estado do programa em determinados pontos da sua execução. O simulador permite definir ou remover pontos de paragem clicando sobre o número da linha de uma instrução no editor de *assembly*.

Quando uma instrução está marcada com um ponto de paragem, a simulação pára automaticamente antes de executar essa instrução.

Quando a simulação está parada ou a correr com uma frequência de relógio não superior a 10 Hz, a linha do programa *assembly* correspondente à próxima instrução a ser executada pelo simulador aparece sombreada no editor de *assembly*.

5.3 Interface com a FPGA

É possível interagir com a FPGA que contém o P4 a partir da aplicação, para tal, é necessário ter o programa Intel Quartus instalado no sistema. Esta funcionalidade não está disponível na versão *online*.

A interface gráfica para interface com a FPGA, mostrada na Figura 5.5, divide-se em tres secções principais: a secção de carregamento de memórias ①, a secção de depuramento do P4 ② e a janela de histórico ③.

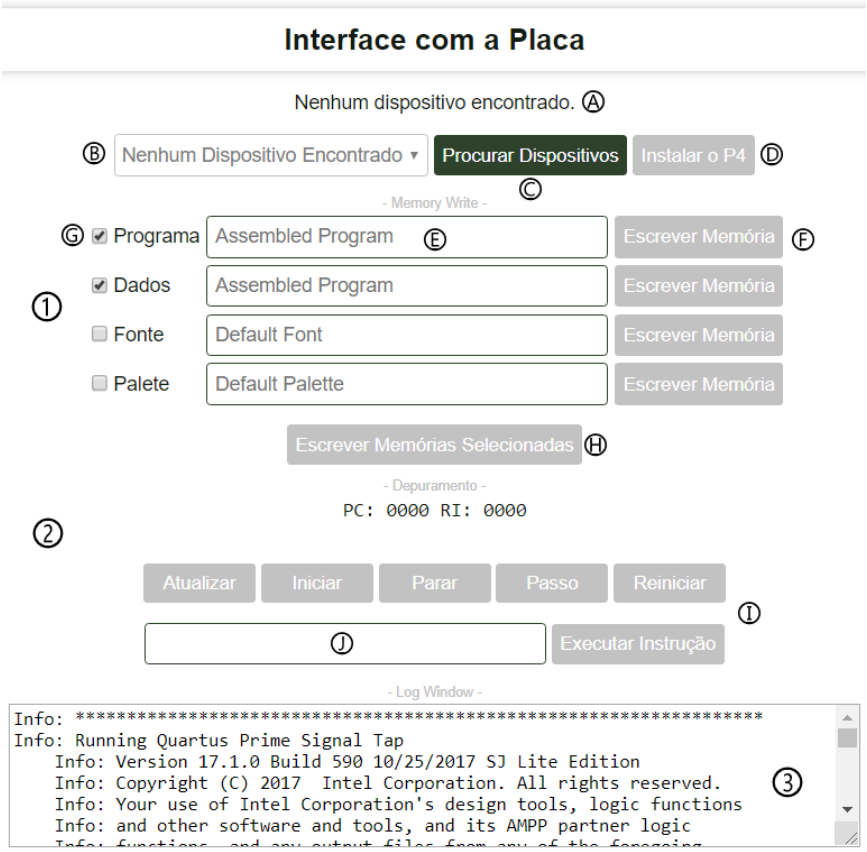


Figura 5.5. Interface gráfica para interface com a FPGA.

No topo da interface é mostrada uma mensagem de estado Ⓐ que indica o estado atual da ligação à FPGA, bem como, o resultado das últimas operações de comunicação.

Por baixo da mensagem de estado há um menu de seleção Ⓑ onde é possível escolher de entre as várias FPGA ligadas ao computador. Para atualizar esta lista deve carregar-se no botão «Procurar Dispositivos» Ⓒ, à direita deste menu.

Do lado direito encontra-se ainda o botão «Instalar o P4» Ⓓ, que configura a FPGA com o processador P4. Esta ação é executada automaticamente sempre que se tenta carregar memórias numa FPGA que ainda não esteja configurada com o P4.

5.3.1 Carregamento de Memórias

Nesta secção ② é possível carregar o conteúdo das várias memórias do P4 a partir de ficheiros MIF, ou, no caso da memória de programa e da memória de dados, com o conteúdo correspondente ao programa anteriormente assemblado, de forma a correr esse programa no P4.

É também possível, carregar a memória da fonte e a memória da palete, que permitem alterar a fonte e palete de cores utilizadas no terminal.

Para fazer o carregamento de uma memória, basta clicar na caixa de texto Ⓔ e selecionar o ficheiro MIF a partir do menu de seleção que abre automaticamente, de seguida, clica-se no botão «Escrever Memória» Ⓕ, à direita, para escrever o ficheiro selecionado na memória correspondente.

Também é possível escrever de uma só vez todas as memórias marcadas com um visto Ⓖ à sua esquerda, bastando para isso clicar no botão «Escrever Memórias Selecionadas» Ⓗ.

Quando o campo do ficheiro é deixado em branco, a memória de programa e a memória de dados são carregadas com o conteúdo correspondente ao programa assemblado, enquanto que a memória da fonte e da palete são carregadas com a fonte e palete predefinidas.

5.3.2 Depuramento na FPGA

É possível comunicar com o P4 para obter em tempo real o valor do *Program Counter* e da instrução a ser executada no P4, e também para enviar comandos que controlam o funcionamento do P4.

Esta funcionalidade permite, por exemplo, parar a execução de um programa. Isto é conseguido através de desativação da contagem do *Program Counter* e substituindo a instrução a ser executada por `NOP`. É possível também reiniciar o P4, fazer execução passo a passo, bem como executar uma instrução específica que o utilizador pode configurar na interface. Estão disponíveis os seguintes botões Ⓘ:

- Atualizar: obtém o valor atualizado do *Program Counter* e do Registo de Instrução.
- Iniciar: retoma a execução do programa.
- Parar: pára a execução do programa.
- Passo: executa uma única instrução.
- Reiniciar: reinicia o P4, colocando o PC e todos os registos a 0.
- Executar Instrução: executa a instrução *assembly* introduzida na caixa de texto Ⓙ junto a este botão.

5.3.3 Janela de Histórico

A comunicação com a FPGA é feita através da ferramenta *Quartus Prime Signal Tap* que permite executar comandos para obter informações da FPGA e atualizar o conteúdo das memórias.

Os comandos executados, bem como o resultado e mensagens de erro, são mostrados na janela de histórico ③, permitindo depurar a comunicação com a FPGA.

5.4 Ferramentas Adicionais

Por baixo da janela do terminal encontram-se ligações para as seguintes ferramentas: o editor de texto do terminal e o editor da fonte do terminal. Ao clicar numa destas ligações abre-se uma janela modal onde é mostrada a interface gráfica da ferramenta.

5.4.1 Editor de Texto do Terminal

Esta ferramenta oferece uma interface gráfica que permite editar o conteúdo do terminal, incluindo a possibilidade de mudar a cor do texto e do fundo. Ao clicar no botão "Gerar *Assembly*" é gerado um programa em *assembly* que imprime na janela do terminal o conteúdo que foi definido através desta ferramenta.

O conteúdo a mostrar no terminal é codificado na memória. Cada palavra na memória corresponde a um caráter em ASCII estendido. Existem sequências de controlo iniciadas por uma palavra com valor `0000h` seguida de uma palavra cujo valor corresponde a um comando. O comando `0000h` indica o final do conteúdo, o comando `0001h` permite definir a posição do cursor, e o comando `0002h` permite definir a cor utilizada. O valor da posição ou da cor é definido pela palavra que sucede a sequência de controlo.

O programa *assembly* gerado inicializa o conteúdo da memória com o conteúdo a mostrar no terminal, codificado desta forma, de seguida, obtém em sequência cada palavra da memória. Quando é encontrada uma sequência de controlo, o programa executa o comando correspondente, caso contrário, o programa imprime o caráter correspondente à palavra lida da memória. O programa termina quando é encontrada a sequência de paragem.

5.4.2 Editor da Fonte do Terminal

Esta ferramenta permite editar a fonte utilizada pelo terminal. Ao abrir a ferramenta é carregada automaticamente a fonte predefinida do terminal. É possível carregar outra fonte a partir de uma imagem PNG de 576 por 576 píxeis que contenha 256 caracteres numa grelha de 16 por 16 caracteres, ou, a partir de um ficheiro MIF.

A janela do editor da fonte do terminal, demonstrada na Figura 5.6, divide-se em duas regiões. No lado esquerdo, há uma grelha de 16 por 16 posições com os caracteres da fonte atual. Ao clicar num desses caracteres, este é carregado numa grelha de 24 por 24 posições, do lado direito, correspondente aos píxeis desse caráter.

Ao premir o botão esquerdo do rato sobre a grelha do lado direito, os píxeis sob o cursor serão preenchidos, ao passo que o botão direito do rato limpa os píxeis sob o cursor.

Por baixo da grelha encontram-se os seguintes botões:

- Limpar: limpa toda a grelha.
- Inverter: inverte todos os píxeis da grelha.
- Copiar: copia o conteúdo da grelha para a área de transferência.
- Colar: preenche a grelha com o conteúdo previamente guardado na área de transferência.
- Guardar: atualiza o caráter atual com o conteúdo da grelha.

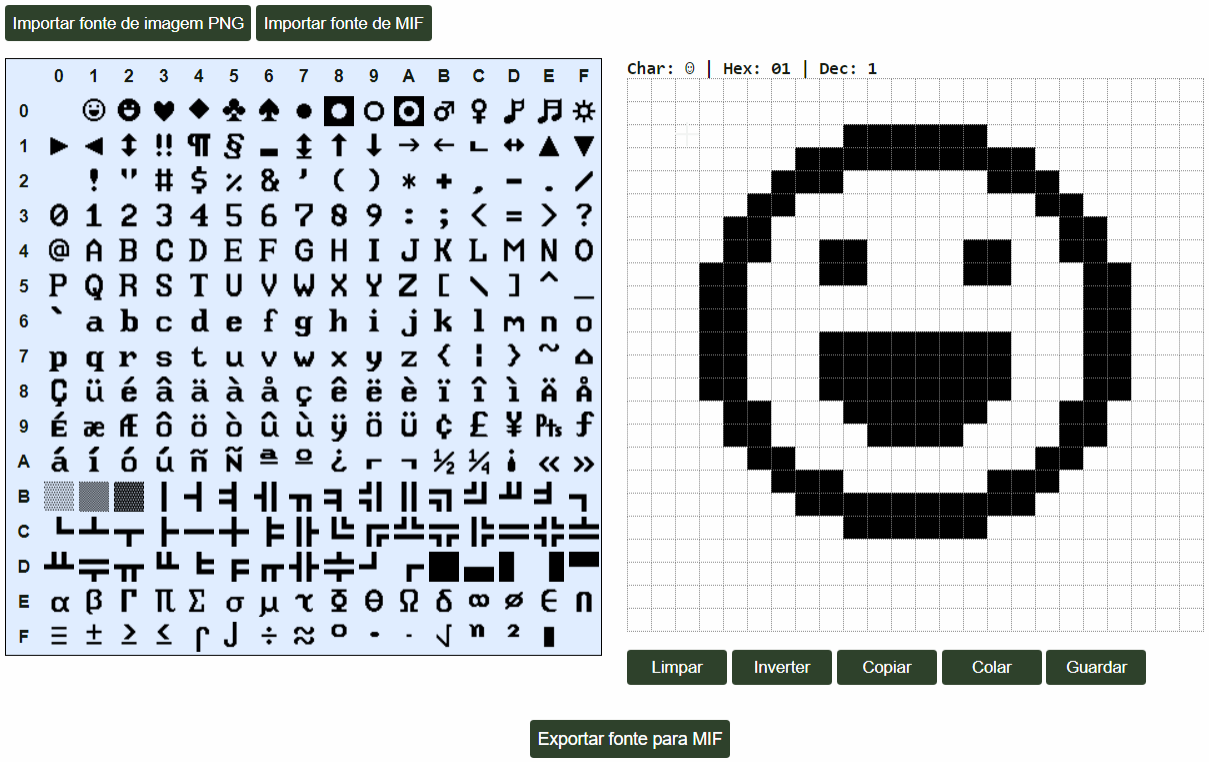


Figura 5.6. Editor da fonte do Terminal.

Após editar a fonte, esta pode ser exportada para um ficheiro MIF que pode ser carregado na memória do P4 de forma a utilizar a fonte criada, ou, que pode voltar a ser importado nesta ferramenta de forma a voltar a editar essa fonte.

5.5 Configurações

É possível alterar o idioma da interface a partir de um menu de seleção Ⓑ disponível no canto inferior direito da interface, mostrado na Figura 5.7. Os idiomas disponíveis são inglês e português.

Todas as definições, incluindo os programas *assembly* guardados, podem ser exportadas para um ficheiro. Esta opção está disponível através de uma ligação no canto inferior esquerdo da interface Ⓐ, bem como a opção de importar um ficheiro previamente exportado.

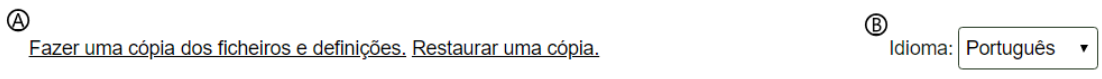


Figura 5.7. Controlos para as configurações da interface.

6 Implementação em Hardware

O desenvolvimento da implementação em hardware foi feito usando uma abordagem *bottom-up*, ou seja, primeiro criaram-se e validaram-se os componentes mais básicos que depois foram utilizados como caixas pretas noutros componentes mais complexos.

Neste capítulo apresenta-se em maior detalhe a implementação de alguns componentes do P4, com especial ênfase no processo de desenvolvimento, indicando-se alguns dos desafios e as soluções encontrados.

6.1 Unidade Lógica e Aritmética

A Unidade Lógica e Aritmética (ULA) é composta por quatro circuitos principais: a unidade aritmética, a unidade lógica, a unidade de deslocamento, e a unidade de seleção de octetos, tal como ilustrado na Figura 6.1. Consoante o código de operação (ULAC), um multiplexador seleciona qual o resultado destas quatro unidades deve ser colocado na saída (R).

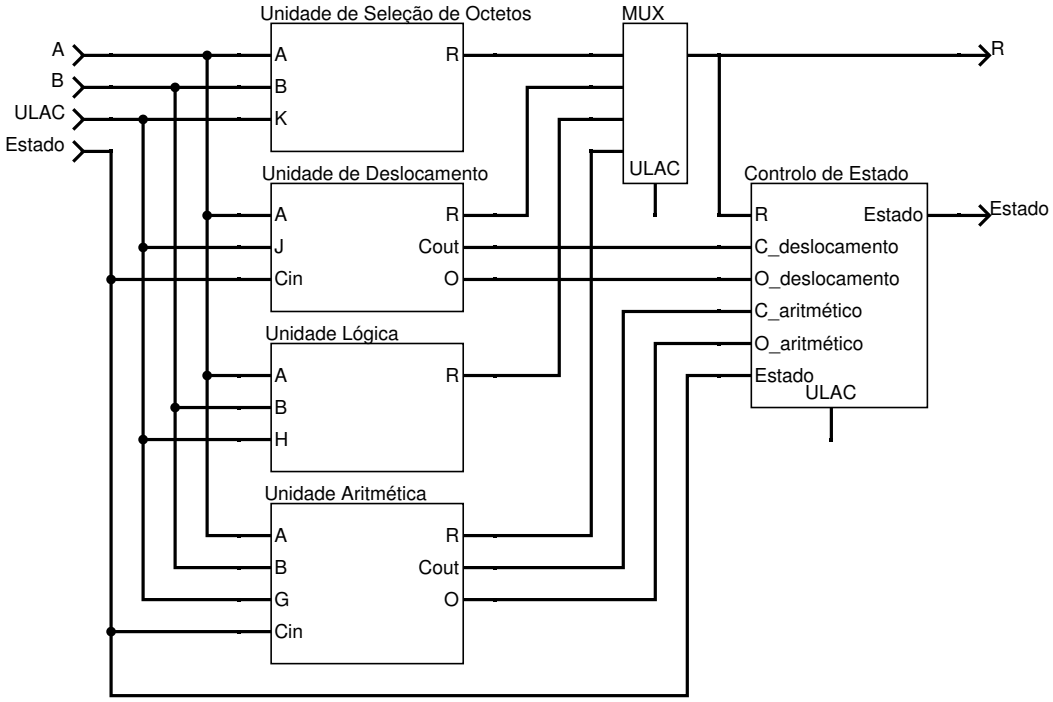


Figura 6.1. Visão geral da Unidade Lógica e Aritmética.

6.1.1 Unidade Aritmética

O primeiro componente da ULA a ser criado foi unidade aritmética, ilustrada na Figura 6.2. O componente principal da ULA é um somador de 16 bits. Usando uma abordagem *bottom-up*, começou-se por fazer um semissomador (*hald-adder*), que consiste numa porta `XOR` e numa porta `AND` tal como se pode ver na Figura 6.3. Usando dois somadores parciais fez-se o somador completo (*full-adder*) de 1 bit ilustrado na

Figura 6.4. Com 16 somadores em paralelo fez-se um somador *ripple-carry* de 16 bits. A ocorrência de *overflows* é detetada fazendo a disjunção exclusiva dos bits de transporte dos dois somadores completos que calculam os bits mais significativos do resultado.

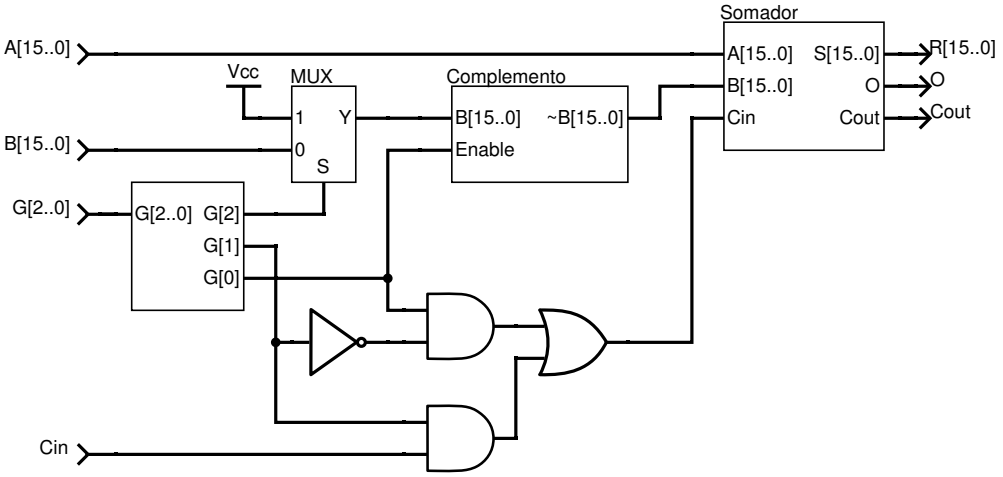


Figura 6.2. Unidade Aritmética.

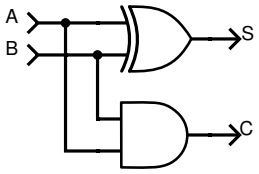


Figura 6.3. Semissomador.

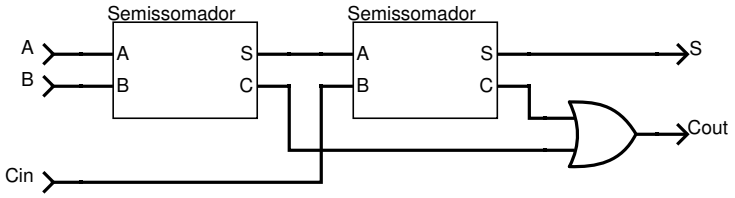


Figura 6.4. Somador Completo.

Além da soma, a unidade aritmética também realiza subtrações. As subtrações em binário são conseguidas fazendo a soma do primeiro operando com o complemento para dois do segundo operando. O complemento do segundo operando é obtido fazendo-se um `XOR` de cada bit com 1. Para que este seja um complemento para dois, é ainda preciso somar 1, o que é conseguido ativando a entrada *carry-in* do somador de 16 bits.

Para as operações `INC` e `DEC`, usa-se um multiplexador que substitui a entrada B por -1 em complemento para dois, que é complementada no caso da operação `INC`.

Tratando-se de circuitos relativamente simples, foi possível testá-los diretamente na placa ligando as entradas da unidade aos interruptores para definir o valor dos operandos, sendo a saída da unidade ligada aos *LEDs* para visualizar o resultado.

6.1.2 Unidade Lógica

A unidade lógica realiza quatro operações: `XOR`, `OR`, `AND` e `NOT`, que correspondem, respetivamente, à disjunção exclusiva, disjunção, conjunção e negação.

Cada uma destas operações é realizada por um bloco composto de 16 portas lógicas, uma para cada bit dos operandos, que realizam a operação em questão. Este circuito está representado de forma simplificada na Figura 6.5.

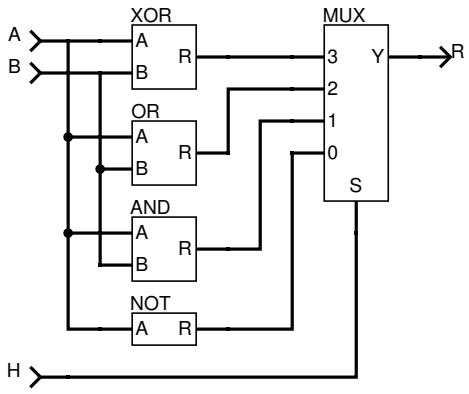


Figura 6.5. Unidade Lógica.

As operações são realizadas em paralelo e um multiplexador seleciona o resultado da operação pretendida, usando o código da operação como sinal de seleção. A implementação em bloco de diagramas desta unidade está no Anexo B.1.

6.1.3 Unidade de Deslocamento

A unidade de deslocamento é construída a partir de multiplexadores que para o bit n da saída, selecionam o bit $n-1$ ou $n+1$ da entrada, consoante se trate de um deslocamento para a esquerda ou para a direita. É ainda necessária alguma lógica adicional para determinar o bit mais significativo e o bit menos significativo da saída, visto que no caso das operações de rotação o valor do bit inserido corresponde ao valor do bit deslocado para fora, no caso das operações de deslocamento com transporte corresponde ao valor do bit de transporte, e nos deslocamentos aritméticos para a direita o bit mais significativo não se altera. O diagrama de blocos desta unidade está no Anexo B.2.

6.1.4 Unidade de Seleção de Octetos

Optou-se por implementar as operações `MVI`, `MVIH` e `MVIL` utilizando a ULA, através da Unidade de Seleção de Octetos, representada na Figura 6.6. Estas operações consistem em carregar uma constante de 8 bits para os 8 bits mais significativos ou menos significativos de um registo.

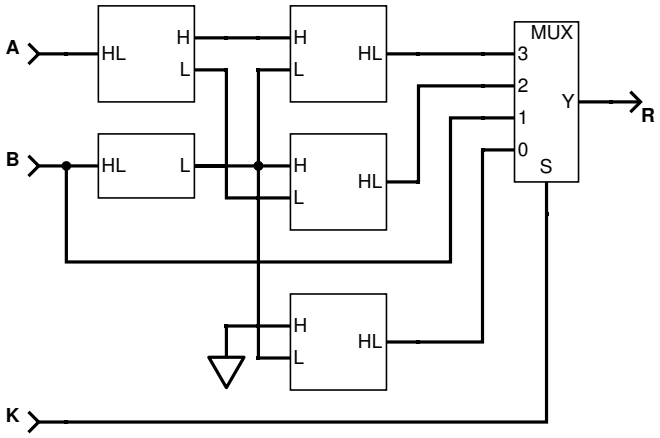


Figura 6.6. Unidade de Seleção de Octetos.

O valor atual do registo a ser alterado é colocado na entrada A desta unidade, enquanto que na entrada B é colocado o valor da constante. Consoante a operação em causa, o octeto mais significativo ou menos significativo da entrada A é substituído pelo octeto menos significativo da entrada B.

Utilizou-se ainda este circuito para implementar a operação `MOV`, que copia o valor de um registo para outro. Neste caso, o valor da entrada A é descartado e o valor da entrada B, correspondente ao valor do registo a ser copiado, é colocado inalterado à saída desta unidade.

6.1.5 Controlo de Estado

Outro circuito também presente nesta unidade é o controlador do registo de estado, que é responsável por atualizar os vários bits de estado, representado no Anexo B.3. Os bits de estado que devem ser atualizados são determinados consoante a operação realizada.

Nas operações em que os bits de estado sinal e zero são atualizados, estes são determinados a partir do bit mais significativo do resultado e a partir do resultado da disjunção lógica negada entre todos os bits do resultado, respetivamente.

Nas operações aritméticas, os bits de estado transporte e excesso são atualizados usando os bits de transporte e excesso gerados pela unidade aritmética, enquanto que nas operações de deslocamento em que estes bits são atualizados, estes bits de estado são atualizados utilizando-se os bits gerados pela unidade de deslocamento.

A atualização dos bits é conseguida utilizando multiplexadores controlados pelo código de operação, que selecionam entre as possíveis origens do novo valor do bit de estado, incluindo o valor anterior para o caso das operações que não alteram esse bit de estado.

As operações `CLC`, `STC` e `CMC`, que desativam, ativam e complementam o bit de transporte, respetivamente, foram também implementadas neste circuito. Para tal foram criados códigos de operação específicos para estas operações e que são interpretados por este circuito de forma a fazer a atualização respetiva do bit de transporte.

Note-se que o registo de estado é sempre atualizado em cada ciclo de relógio com os bits de estado gerados pela ULA, pelo que mesmo quando o processador está a realizar operações que não envolvem

a ULA, é necessário que o sinal de controlo enviado para a ULA corresponda a uma operação que mantém os bits de estado inalterados. O valor do sinal de controlo escolhido para este tipo de operações foi `11000`, correspondente à operação `MVI`.

6.2 Banco de Registos

Na criação do banco de registos, novamente usando a abordagem *bottom-up*, começou-se por criar um registo individual de 16 bit, usando 16 *flip-fops* do tipo D. O banco de registos consiste em 7 registos de 16 bits ligados a dois multiplexadores que selecionam os dois registos cujos valores são colocados nas saídas A e B, usando os sinais de seleção `SelA` e `SelB`; e um descodificador, responsável por colocar o sinal `WRC` à entrada do registo a ser escrito, consoante o valor do sinal `SelC`. A Figura 6.7 mostra um esquema simplificado deste circuito, cuja implementação em diagrama de blocos está no Anexo B.4.

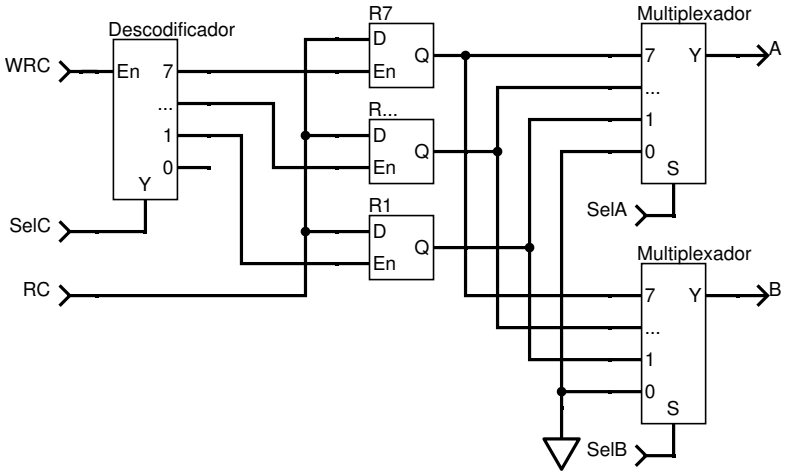


Figura 6.7. Banco de Registos.

Este circuito também foi testado diretamente na placa, usando um botão de pressão como sinal de relógio, os interruptores para definir as entradas e os *LEDs* para mostrar a saída.

6.3 Registo de Estado

O registo de estado, representado na Figura 6.8, consiste num registo de 4 bits que guarda os bits de estado de sinal, transporte, excesso e zero. Este registo guarda o valor à entrada, proveniente da ULA, em cada ciclo de relógio, e coloca-o à saída até ao ciclo seguinte.

Quando é tratada uma interrupção, é necessário manter o valor do estado de forma a que possa ser recuperado mais tarde. Isto é conseguido através de um registo extra que guarda o valor atual do estado quando a entrada `store` está ativa. Por outro lado, a entrada `load` ativa faz com que seja colocado à saída o valor guardado neste registo, recuperando-se assim o estado anteriormente guardado quando a entrada `store` foi ativada.

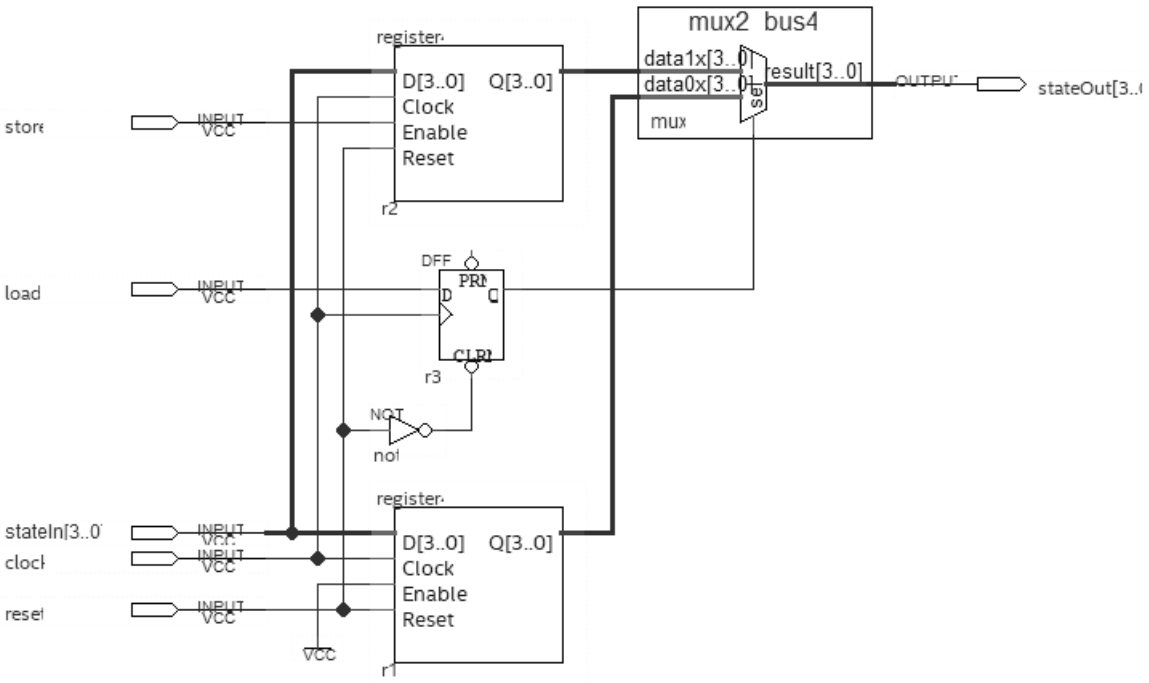


Figura 6.8. Registo de Estado.

6.4 *Program Counter*

O registo *Program Counter* (PC), representado em diagrama de blocos na Figura 6.9, consiste num registo que incrementa uma unidade a cada ciclo de relógio e que possui a possibilidade de ser carregado com um dado valor (`data`) de forma assíncrona, o que significa que o valor carregado é colocado à saída do PC no mesmo ciclo de relógio em que a entrada `Ld` é ativada.

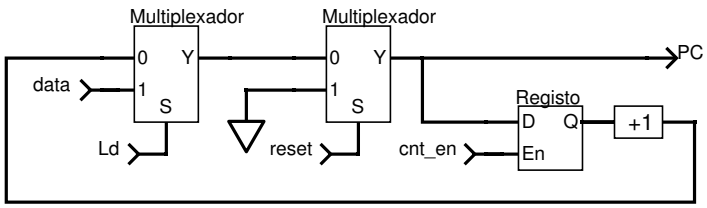


Figura 6.9. *Program Counter*.

6.5 Unidade de Controlo de Salto

A unidade de controlo de salto, representada na Figura 6.10, tem à sua entrada o valor do registo de instrução e o valor do registo de estado, que são utilizados para determinar quando deve ou não ocorrer o salto, consoante a condição e o valor dos bits de estado.

É utilizado um multiplexador no qual entra cada bit de estado, e ainda o resultado da disjunção negada do bit de estado zero e sinal, que corresponde a um resultado positivo. Desta forma, cada uma destas entradas indica se as condições zero, transporte, excesso, sinal e positivo é verdadeira. Usando

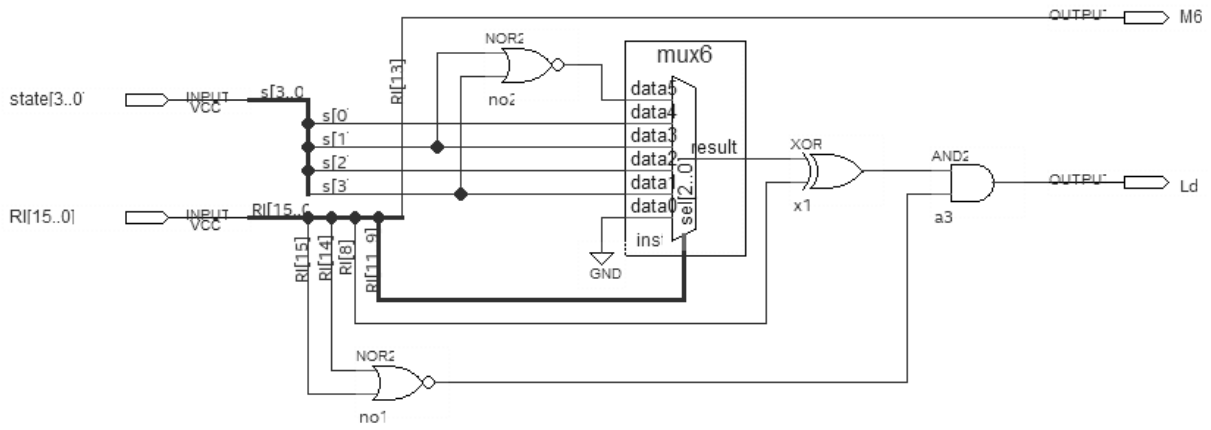


Figura 6.10. Unidade de Controlo de Salto.

os bits mais significativos do campo da instrução que indica a condição do salto, seleciona-se a entrada correspondente à condição do salto. É então realizada a disjunção exclusiva deste valor com o bit menos significativo da condição de salto, que nos indica que o salto deve ser realizado quando a condição é falsa. O valor resultante indica se a condição do salto é cumprida ou não.

Note-se que há também uma entrada do multiplexador cujo valor é sempre 0. Isto traduz-se numa condição que nunca ocorre, ou uma condição que ocorre sempre, no caso em que o bit menos significativo do código da condição é 1. Estas duas condições são usadas para codificar a instrução `NOP` e os saltos incondicionais, respetivamente, como já foi referido no Capítulo 4.2.6, onde também está a Tabela 4.4 com a codificação das condições de teste.

O salto está ainda dependente de os dois bits mais significativos da instrução terem valor 0, pois apenas instruções que cumprem esta condição correspondem a instruções de salto. Quando é determinado que o salto deve ocorrer, é ativada a saída `Ld`, que faz com que o PC seja carregado com o endereço de destino do salto.

6.5.1 Resolução de Saltos

Para acelerar a resolução dos saltos, adicionou-se ao pipeline do P4 o multiplexador MUX6 e um somador dedicado para somar o valor do PC com os 8 bits menos significativos da instrução, como se pode observar na Figura 4.1. Com isto, é possível obter o valor de destino do salto logo no segundo andar do *pipeline*. Além disso, contrariamente à especificação no livro [1], optou-se por colocar a unidade de controlo de salto também no segundo andar do *pipeline*.

A unidade de controlo de salto foi alterada de forma a gerar a saída `M6`, que indica se se trata de um salto absoluto ou relativo, e que controla o multiplexador MUX6, que tem como entradas o endereço de destino do salto, obtido a partir de um registo, para o caso dos saltos absolutos, e o endereço de destino do salto, calculado como a soma do PC e os 8 bits menos significativos da instrução, para o caso dos saltos relativos.

6.5.2 *Delay Slot*

Como resultado das escolhas arquiteturais em relação à resolução dos saltos, apenas uma instrução entra no *pipeline* antes de o salto ser resolvido. Esta instrução constitui uma *delay slot*, ou seja, no P4 a instrução que segue a uma instrução de salto é executada independentemente de o salto se realizar ou não.

6.6 Descodificador de Instruções

O descodificador de instruções do P4 é um circuito combinatório relativamente simples, visto tratar-se de uma arquitetura RISC. Este circuito tem como entrada o valor do registo de instrução e, a partir deste, gera vários sinais de controlo usados por outros componentes do P4. Estes sinais de controlo são:

- `M2` a `M5`: sinais de controlo dos multiplexadores MUX2 a MUX5.
- `SelA` e `SelB`: sinais que selecionam os registos cujos valores devem ser colocados nas saídas A e B do banco de registos, respetivamente.
- `SelC`: sinal que seleciona o registo a ser escrito.
- `WRC`: sinal que permite ativar a escrita no banco de registos.
- `RM`: sinal que permite ativar a escrita na memória.
- `ALUC`: sinal de controlo da ULA que indica a operação a ser realizada.

Em primeiro lugar, foi criada a tabela presente no Anexo C.2 que mostra o valor dos bits que compõem cada instrução e os valores dos respetivos sinais de controlo. Cada linha da tabela representa uma instrução, que estão agrupadas pelo formato e pelo tipo de operação a que correspondem. Na parte esquerda da tabela estão representados os bits de cada instrução e os valores ou campos correspondentes a esses bits. Na parte direita da tabela são mostrados os vários sinais de controlo. As colunas `M1`, `M2`,`M3`, `WRC`, `RM`, `M6`, `Ld` e `ALUC`, indicam o valor destes sinais para cada instrução. Os campos preenchidos com X indicam valores *don't care*, isto é, bits da instrução e sinais de controlo cujos valores não são relevantes para a instrução em causa e por isso podem ter qualquer valor. As colunas `SelA`, `SelB` e `SelC` indicam os bits da instrução que devem ser usados como valor para estes sinais. A coluna *Flags* indica os bits de estado que devem ser atualizados pela instrução.

Por simples análise desta tabela é possível deduzir a lógica necessária para gerar cada sinal de controlo. Analisando-se a coluna respondente a um sinal de controlo verifica-se de que forma este valor é determinado pelos bits de cada instrução. Por exemplo, o sinal `M2` tem valor 0 para todas as instruções, à exceção das instruções `MVI`, `MVIH` e `MVIL`, em que tem valor 1. Estas instruções têm em comum o facto de terem os dois bits mais significativos, os bits 15 e 14, com valor 1, e o bit mais significativo do código de operação, o bit 10, com valor 0. Uma vez que estas condições aplicam-se única e exclusivamente a estas instruções, podem ser utilizadas para determinar este sinal de controlo.

O descodificador de instruções foi criado de forma progressiva. Numa versão inicial apenas suportava algumas instruções relacionadas com a ULA e posteriormente foi adicionada a lógica para a geração de sinais de controlo necessários para a implementação de outras instruções. Além disso, começou

por ser implementado em diagramas de blocos, o que embora seja relativamente simples de fazer por se tratarem apenas de operações lógicas, no final o circuito tornou-se difícil de compreender, o que dificulta o seu depuramento e possíveis alterações futuras. Por este motivo, optou-se por substituir por uma descrição em VHDL. A lógica para gerar o sinal do exemplo anterior é facilmente descrita da seguinte forma:

```
M2 <= '0';
if format = "11" and op(2) = '0' then M2 <= '1'; end if; -- MVI, MVIH, MVIL
```

A segunda linha indica que o sinal deve ser 1 quando os dois bits mais significativos da instrução têm valor 1 e quando o bit mais significativo do código de operação tem valor 0, ou seja, quando se tratam das instruções `MVI`, `MVIH` e `MVIL`. A primeira linha garante que o valor deste sinal é 0 para todas as outras instruções. No Anexo D.1 está a descrição completa em VHDL do descodificador de instruções.

A partir da tabela é também possível verificar-se que o sinal `M1`, que controla o multiplexador MUX1 que estava previsto no livro [1], deve ser apenas 0 ou *don't care*. Isto significa que este multiplexador já não é necessário, o que se deve ao facto de se ter acelerado a resolução de saltos utilizando um somador dedicado para calcular a soma do PC com o deslocamento, deixando de ser necessário usar a ULA para este cálculo. Como tal, já não é necessário o MUX1 para colocar o valor do PC à entrada da ULA.

6.7 *Forwarding* de Dados

O *forwarding* de dados é uma otimização do processador que permite evitar paragens no *pipeline* causadas pela existência de dependências de dados entre instruções consecutivas. Isto é conseguido através de dois multiplexadores, MUX4 e MUX5, presentes no segundo andar do *pipeline*, como se observa na Figura 4.1, que nas alturas apropriadas, substituem as saídas A e B do banco de registos por valores obtidos diretamente de andares posteriores do *pipeline*.

Estes multiplexadores devem ser controlados por um sinal de controlo de forma que quando este sinal tem valor 0 é utilizada a saída do banco de registos, quando tem valor 1 é utilizado o valor do resultado da ULA, quando tem valor 2 é utilizado o valor de saída da memória de dados/periféricos, e quando tem valor 3 é utilizado o valor à saída do MUX3.

O princípio de funcionamento do componente que gera os sinais de controlo destes multiplexadores, representado no Anexo B.5, consiste em analisar os sinais de controlo `SelA`, `SelB`, `SelC` e `WRC`. Através do sinal `WRC` e `SelC` é possível determinar os registos no qual houve escritas nos ciclos de relógio anteriores. `SelA` e `SelB` indicam os registos cujos valores atualizados devem ser colocados na saída dos multiplexadores MUX5 e MUX6.

As unidades FWA1 e FWB1, do tipo `op_forwarding1` analisam os sinais `SelA/B`, `SelC` e `WRC` ocorridos no ciclo de relógio anterior. Caso `WRC` esteja ativo e `SelA/B` e `SelC` sejam iguais, é ativada a saída `FW`, que indica que deve haver *forwarding* de dados. O sinal de controlo deverá ser 1 ou 2, consoante a origem dos dados seja a ULA ou a memória de dados/periféricos, o que pode ser determinado a partir do sinal de controlo do multiplexador `M3`.

As unidades FWA2 e FWB2, do tipo `op_forwarding2` analisam os sinais `SelA/B`, `SelC` e `WRC` ocorridos no segundo ciclo de relógio anterior. Caso `WRC` esteja ativo e `SelA/B` e `SelC` sejam iguais, o sinal de controlo gerado será 3, de forma a ser utilizado o valor no final do pipeline, caso contrário, o sinal de controlo gerado é 0, utilizando-se assim o valor atual à saída do banco de registos.

Em suma, as unidades FWA1 e FWA2 determinam quando é necessário fazer o *forwarding* do operando A a partir do terceiro e do quarto andar do *pipeline*, respetivamente. O mesmo se aplica às unidades FWB1 e FWB2, mas neste caso para o operando B.

6.8 Entradas e Saídas (IO)

Este circuito está representado no Anexo B.6 e é responsável pelas entradas e saídas mapeadas em memória, e também, pela leitura e escrita na memória de dados.

Existem dois componentes principais deste circuito, o `IO_write`, que é essencialmente um descodificador, que seleciona o dispositivo onde a escrita deve ocorrer consoante o endereço, e o `IO_read`, que é um multiplexador que seleciona a saída do dispositivo que deve ser lida consoante o endereço.

6.9 Memória de Dados

De acordo com o diagrama do P4 apresentado no livro [1], a entrada de dados desta memória estava ligada ao *Program Counter*, o que se presume tratar de uma gralha visto que desta forma não seria possível escrever na memória outro valor que não o do *Program Counter*.

Como já referido no Capítulo 4, o formato da instrução `STOR` foi alterado de forma a utilizar os campos RA e RB, realizando a operação `M[RB] = RA`, além disso a instrução `LOAD` utiliza os campos RC e RB, realizando a operação `RC = M[RB]`.

Desta forma, a utilização de cada campo é consistente em todas as operações, permitindo simplificar o descodificador de instruções e as ligações à memória de dados. Mais concretamente, a entrada de endereço da memória foi ligada à saída B do banco de registos, visto que o endereço é sempre especificado pelo valor do registo indicado pelo campo RB, por outro lado, a entrada de dados da memória foi ligada à saída A do banco de registos, visto que na instrução `STOR` é utilizado o valor do registo indicado pelo campo RA, tal como se pode observar na Figura 4.1.

6.10 Terminal (Ecrã e Teclado)

O terminal é um dispositivo de entrada e de saída. A entrada é feita através da ligação a um teclado compatível com PS/2, enquanto que a saída é feita através da ligação a um ecrã VGA.

Teclado.

Uma vez que a placa do P4 não possui uma entrada para teclado, foi necessário desenvolver o protótipo, ilustrado na Figura 6.11, de uma segunda placa de expansão com uma porta USB por forma a permitir a ligação de teclados que funcionem com o modo de compatibilidade PS/2, ou, alternativamente, qualquer outro teclado que suporte o protocolo PS/2 utilizando-se um adaptador passivo para fazer a ligação.

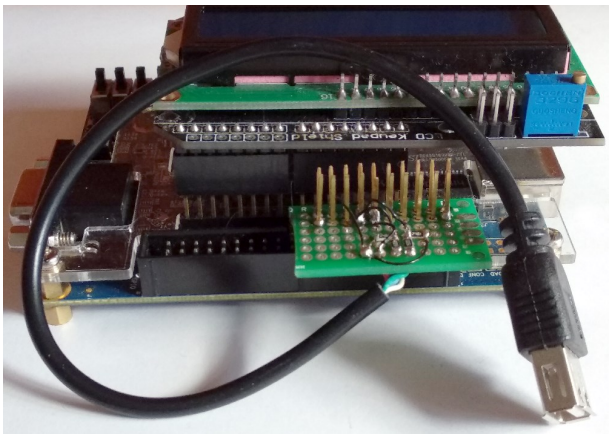


Figura 6.11. Fotografia do protótipo da placa de expansão com uma porta USB acoplado à entrada GPIO da placa.

A porta USB foi ligada a 4 pinos da entrada GPIO da placa, de acordo com a Tabela 6.1. Para o funcionamento correto, tanto a linha de relógio como a de dados foram ligadas a uma resistência *pull-up* de 2kΩ, além disso, também foram usadas resistências 120Ω para fazer a adaptação de níveis de tensão entre o teclado (5V) e a placa (3,3V), de acordo com o esquema da Figura 6.12.

Tabela 6.1. Correspondência entre os pinos da placa, teclado e adaptador USB.

Placa		PS/2			USB		
Nome	Número	Nome	Número	Cor	Nome	Número	Cor
VCC	11	VCC	4	Vermelho	VCC	1	Vermelho
GPIO11	14	Data	1	Branco	Data-	2	Branco
GPIO10	13	Clock	5	Castanho	Data+	3	Verde
Ground	12	Ground	3	Preto	Ground	4	Preto

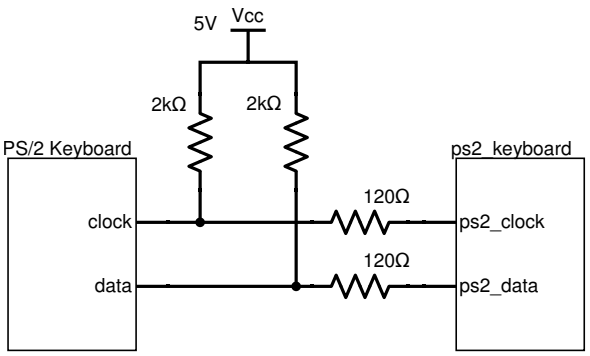


Figura 6.12. Ligação entre o teclado e a placa.

A comunicação com o teclado é feita através do protocolo PS/2. Utilizou-se o componente `ps2_keyboard` cujo funcionamento e VHDL estão descritos na página da *wiki* da *DigiKey* sobre a interface PS/2 para um teclado [28]. Existem dois sinais fornecidos pelo teclado, o sinal de relógio e o sinal de dados. Quando

inativo, ambos os sinais têm valor 1. Quando o teclado envia dados para a placa, ativa o sinal de relógio, alternando-o sucessivamente entre 0 e 1. Em cada flanco descendente do sinal de relógio o sinal de dados terá o valor de um bit de dados. Cada transação consiste num bit de dados inicial com valor 0, seguido de 8 bits correspondentes a um *Scan Code*, com o bit menos significativo em primeiro lugar, seguido de um bit de paridade e por fim um bit com valor 1.

Quando uma tecla é premida ou libertada, um ou vários *Scan Codes*, correspondentes a essa tecla e ação, são enviados pelo teclado. Estes valores são utilizados pelo componente `terminal_input`, cujo papel é determinar que teclas foram premidas a partir das sequências de *Scan Codes*, tal como indicado no Anexo C.1, e consoante a sequência de teclas premidas, determinar o valor de entrada correspondentes ao ASCII estendido do caráter introduzido.

O esquema de teclado suportado é o português (Portugal) e o seu comportamento, embora com algumas diferenças, é similar ao de um teclado utilizado num computador:

- A tecla Caps Lock alterna entre o modo de introdução de minúsculas e maiúsculas.
- A tecla Shift, enquanto premida, inverte o modo de introdução de minúsculas e maiúsculas. Permite também introduzir o caráter superior de algumas teclas, por exemplo, o caráter `!` pode ser introduzido premindo a tecla `1` enquanto a tecla Shift está premida.
- A tecla Alt Gr tem um comportamento equivalente a pressionar simultaneamente a tecla Shift e Ctrl e permite introduzir o caráter alternativo de algumas teclas, por exemplo, o caráter `@` pode ser introduzido premindo a tecla `2` enquanto a tecla Alt Gr está premida.
- O caráter € do teclado, visto não existir em ASCII estendido CP437, é substituído pelo caráter ϵ.
- É possível introduzir caracteres acentuados premindo a tecla correspondente ao acento e de seguida a tecla correspondente ao caráter a ser acentuado.
- Quando a versão acentuada do caráter não existe em ASCII estendido, considera-se que foi introduzida a versão não acentuada desse caráter, caso esta exista.
- É possível introduzir o caráter correspondente a um acento premindo a tecla correspondente ao acento e de seguida a tecla de espaço. Caso este não exista em ASCII estendido, o pressionar da tecla de espaço é ignorado, comportando-se como se apenas tivesse sido pressionada a tecla de acento.
- Pressionar uma ou diferentes teclas de acento consecutivamente é equivalente a apenas ter pressionado a última tecla de acento pressionada.

Para permitir um maior leque de funcionalidades, as teclas que não correspondem a caracteres são também mapeadas para caracteres em ASCII estendido, de acordo com a Tabela 6.2.

Além do valor ASCII estendido, que é colocado no octeto menos significativo do porto de leitura do terminal, é também colocado nos bits 8, 9 e 10, o estado das teclas Shift, Ctrl e Alt, como esquematizado na Tabela 6.3, tomando estes o valor 1 quando a respetiva tecla se encontrava premida, e 0 caso contrário.

Tabela 6.2. Mapeamento entre teclas especiais e caracteres.

Tecla	Caráter	Dec.	Hex.
F1	☺	1	01
F2	☻	2	02
F3	♥	3	03
F4	♦	4	04
F5	♣	5	05
F6	♠	6	06
F7	•	7	07
F8	◘	8	08
F9	○	9	09
F10	◙	10	0A
F11	♂	11	0B
F12	♀	12	0C
Print Screen	♪	13	0D
Scroll	♫	14	0E
Pause	☼	15	0F
End	►	16	10
Home	◄	17	11
Insert	↕	18	12
Backspace	‼	19	13
Enter	¶	20	14
Tab	▬	22	16
Escape	↨	23	17
Up Key	↑	24	18
Down Key	↓	25	19
Right Key	→	26	1A
Left Key	←	27	1B
Num	∟	28	1C
Delete	↔	29	1D
Page Up	▲	30	1E
Page Down	▼	31	1F
Menu	⌂	127	7F
Win Left	⌐	169	A9
Win Right	¬	170	AA

Tabela 6.3. Significado dos bits do valor lido a partir do porto de leitura do Terminal.

10	9	8	7	6	5	4	3	2	1	0
Alt	Ctrl	Shift	Caráter							

Ecrã VGA.

Este componente é baseado no `VGA_Interface` do P3, com algumas adaptações de forma a suportar uma resolução superior e cores. Existem duas principais funções, por um lado, guardar a informação referente ao texto a ser mostrado e alterá-lo consoante as escritas no porto de escrita e de cor, por outro, gerar um sinal VGA para mostrar o conteúdo da janela de texto num ecrã.

A resolução da janela de texto do P4 é de 1920x1080 píxeis ao invés dos 640x480 do P3, e a fonte passou de 8x8 píxeis para uma fonte configurável de 24x24 píxeis. Além disso, passou de 24 linhas por 80 colunas para 45 linhas por 80 colunas. Outra alteração é o facto de o terminal do P4 passar a suportar cores, usando uma palete configurável de 256 cores de 12 bits. Por questões de compatibilidade também é disponibilizada a resolução 1280x960 píxeis, que pode ser selecionada através do porto de controlo.

A interface com o ecrã VGA é controlada pelo componente `VGA_Interface`, descrito em VHDL. Este componente é responsável por aceitar os comandos de escrita e controlo do terminal e gerar os sinais VGA de forma a apresentar a janela do terminal num ecrã VGA.

O conteúdo do terminal é mantido em duas memórias, `char_memory`, que contém o valor ASCII do caráter a imprimir em cada posição do terminal, e a `color_memory` que, similarmente, contém o índice da cor do caráter e do fundo. Uma escrita para o porto de escrita escreve o valor escrito para este porto na `char_memory`, enquanto que o valor do índice da cor atual, guardado num registo, é escrito na `color_memory`.

O endereço no qual é feita a escrita nestas memórias corresponde à posição do cursor, que é mantida num registo cujo valor pode ser definido por uma escrita para o porto de controlo, e é incrementado a cada escrita para o porto de escrita. Por outro lado, uma escrita para o porto de cor atualiza o registo que mantém a cor atual com o valor escrito para este porto, e que será utilizado nas próximas escritas na `color_memory`.

Para a geração dos sinais VGA, existem dois contadores que mantêm a contagem da linha e coluna do píxel que está a ser mostrado. Para se obter uma resolução de 1920 colunas por 1080 linhas com uma frequência de atualização de 60 Hz, utilizada pelo terminal do P4, são necessárias 220 colunas e 45 linhas adicionais de forma a permitir a sincronização do ecrã nesta resolução [29], além disso, durante o envio das colunas e linhas adicionais são ativados dois sinais de sincronização, vertical e horizontal, que permitem ao ecrã determinar o início de cada nova linha e quadro.

Para além dos contadores referidos anteriormente, que indicam os valores das coordenadas horizontais e verticais do píxel a ser escrito no ecrã, existem outros dois contadores que mantêm as coordenadas de texto para esse mesmo píxel.

Para determinar o valor do píxel a ser mostrado, ou seja, o valor de 12 bits para cada uma das componentes de cor (vermelho, verde e azul), existem vários estágios que ocorrem em diferentes ciclos do relógio.

Em primeiro lugar, é obtido da `char_memory` o valor ASCII, e da `color_memory`, o índice de cor, do caráter na posição de texto correspondente ao píxel a ser mostrado.

Num segundo estágio, é obtido, o valor das componentes de cor para o índice de cor obtido. Este valor está guardado na memória `palette_memory`. Esta é uma memória com 256 posições e palavras de 12 bits, onde os bits 0 a 3 correspondem à componente azul, os bits 4 a 7 à componente verde, e os bits 8 a 11 à componente vermelha. Tem-se portanto 4 bits para cada componente, que corresponde à resolução suportada pelos conversores digital-analógico da placa responsáveis por gerar as saídas VGA das componentes de cor.

Neste estágio é também obtido da `font_memory` o valor do píxel para as coordenadas e o caráter a ser imprimido. A fonte é codificada na memória da seguinte forma: cada caráter ocupa 576 bits e estão organizados na memória sequencialmente e por ordem ascendente do seu valor ASCII. Por exemplo, o caráter cujo valor ASCII é 1 está codificado nas posições 576 a 1151 da memória. Os 576 bits de cada caráter codificam cada píxel do caráter, sendo que o valor *um* representa um píxel ativo, enquanto o valor *zero* representa um píxel inativo. Cada 24 bits sequenciais representam uma linha, representando os píxeis dessa linha da esquerda para a direita, enquanto que as linhas estão organizadas do topo para o fundo.

Por último, os valores obtidos para as componentes de cor são enviados para o ecrã.

6.11 Interrupções

6.11.1 Máscara de Interrupções

A máscara de interrupções consiste num registo para o qual se pode escrever utilizando o porto correspondente à máscara de interrupções. Este valor é utilizado pelo codificador de interrupções para determinar a próxima interrupção a ser tratada.

6.11.2 Codificador de Interrupções

O codificador de interrupções é descrito em VHDL e consiste num *priority encoder* que determina a próxima interrupção a ser tratada, fazendo a conjunção lógica entre a máscara de interrupções e o registo de interrupções pendentes. Caso o valor resultante seja diferente de zero, existem interrupções pendentes, e a próxima a ser tratada é aquela correspondente ao bit diferente de zero menos significativo, ou seja, a interrupção cujo vetor é mais baixo.

Este componente recebe *flags* das várias fontes de interrupção. Sempre que uma dessas *flags* é ativada, o bit correspondente à respetiva interrupção é ativado no registo de interrupções pendentes. Por outro lado, quando é ativada a *flag Interrupt Acknowledgement* (IAK), que é gerada pela unidade de interrupção para indicar que a interrupção foi tratada, o bit correspondente à interrupção a ser tratada é colocado a zero.

6.12 Botões de Pressão

A placa do P4 possui 8 botões de pressão, sendo um deles utilizado para fazer a reinicialização do P4, colocando o *Program Counter* e todos os registos a zero. Os restantes botões geram uma interrupção quando são premidos.

6.12.1 Botões da Placa de Expansão

À exceção do botão de reinicialização, os demais botões da placa de expansão estão ligados sobre a forma de divisores de tensão a uma única entrada analógica que recebe várias gamas de valores consoante os botões que estão premidos num dado momento, o que significa que é necessário avaliar este valor para determinar os botões premidos.

Para tal, foi utilizado o ADC (*Analog-to-Digital Converter*) da FPGA que gera um valor de 12 bits a partir do valor lido nesta entrada. De seguida, observaram-se os valores gerados quando os vários botões são premidos isoladamente e determinaram-se intervalos de valores, indicados na Tabela 6.4, que englobam os valores lidos para cada botão.

Tabela 6.4. Mapeamento dos botões analógicos.

Botão	Right	Up	Down	Left	Select
Valor	000h - 1FFh	200h - 3FFh	400h - 5FFh	600h - 7FFh	800h - BFFh

Posteriormente, criou-se o circuito da Figura 6.13, que faz o mapeamento destes intervalos para cinco saídas, cada uma ativada quando o valor se encontra dentro de um destes intervalos, indicando desta forma o botão premido.

6.12.2 *Debouncing*

Devido à natureza mecânica dos botões, quando um botão é premido é comum que durante alguns instantes o valor da entrada varie entre premido e não premido até estabilizar no novo valor [30], o que poderia levar um programa a comportar-se como se o botão tivesse sido pressionado várias vezes pelo utilizador.

Para evitar esta situação é aplicado um filtro de *debouncing* que descarta variações momentâneas do valor da entrada, ilustrado na Figura 6.14. Isto é conseguido através de um contador que é reinicializado sempre que existe uma variação do valor da entrada. Quando a entrada se mantém estável pelo tempo suficiente para o contador chegar ao final da contagem, é ativada a escrita do registo da saída com o valor da entrada.

É utilizado um contador de 19 bits ligado a um sinal de relógio com uma frequência de 50 MHz, desta forma o sinal de saída só é atualizado quando o sinal de entrada se mantém estável por pelo menos 10 ms.

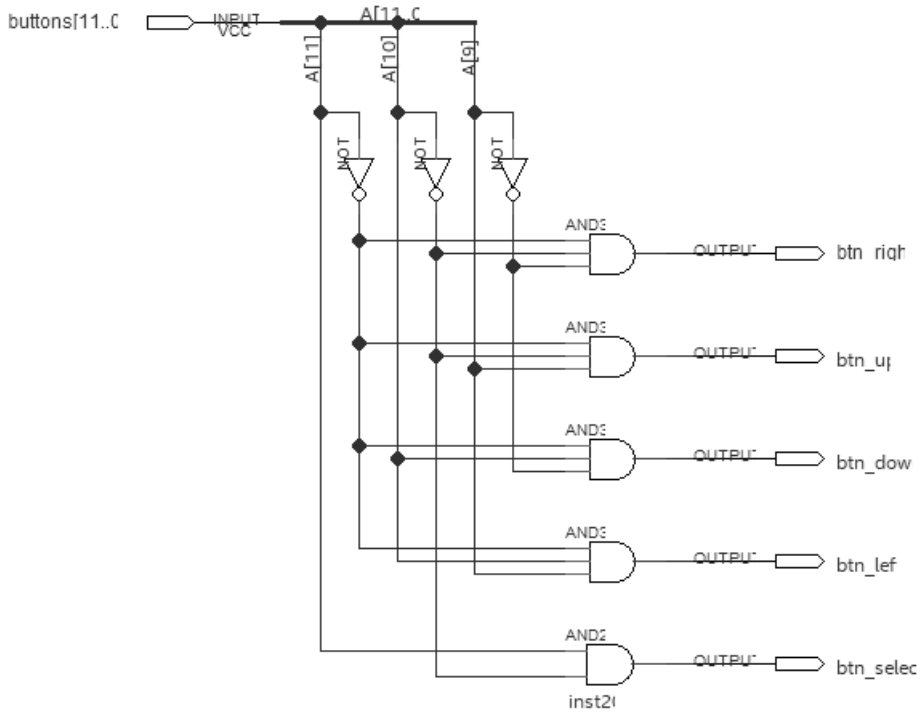


Figura 6.13. Circuito que descodifica a entrada correspondente aos botões analógicos.

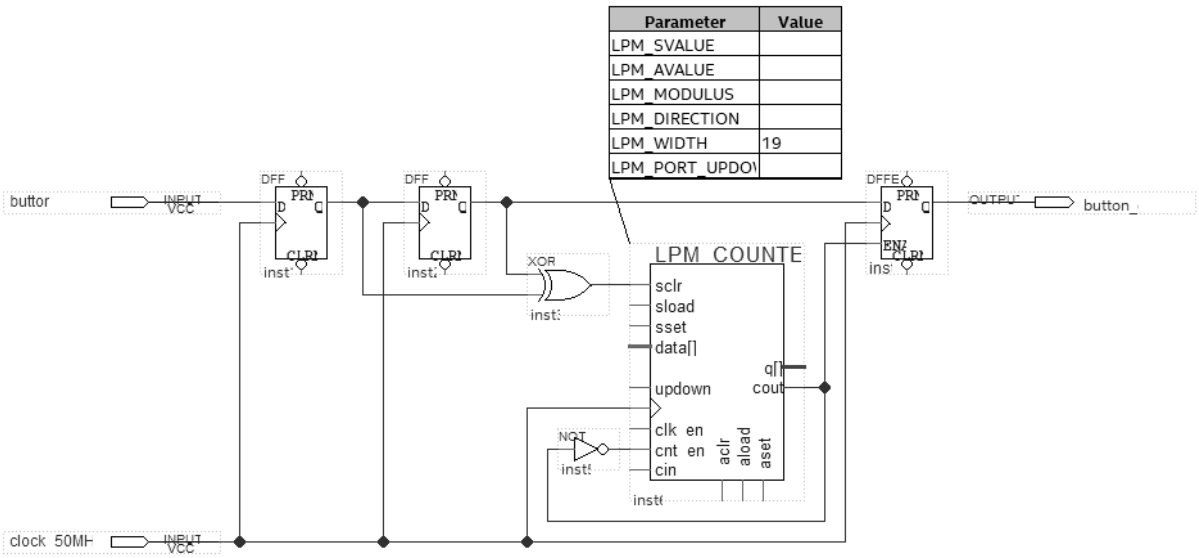


Figura 6.14. Circuito que implementa o filtro de *debouncing*.

6.13 *LEDs*

O circuito de controlo dos *LEDs* consiste num registo de 10 bits cuja escrita é ativada quando é feita uma escrita para o porto correspondente a este periférico. Cada bit deste registo está ligado a um *LED*, de forma que os *LEDs* se mantêm acesos ou apagados consoante o último valor escrito para este registo. Inicialmente os *LEDs* encontram-se apagados e voltam ao estado de apagados quando o P4 é reiniciado.

6.14 Temporizador

O temporizador do P4 é implementado por um circuito descrito em VHDL. Este componente contém dois contadores, um que conta o número de ciclos de relógio equivalentes a 100 ms e que, quando o temporizador está ativo, decrementa outro contador sempre que a sua contagem chega ao fim.

Uma escrita para o porto que serve para definir a contagem do temporizador inicializa este contador com esse valor. Uma escrita do valor 1 para o porto de inicialização do temporizador faz com que este comece a decrementar a cada 100 ms.

No final da contagem, é ativada uma *flag* que dá origem à interrupção correspondente ao temporizador.

6.15 Mostrador Alfanumérico LCD

O controlo de baixo nível do mostrador alfanumérico LCD é feito pelo componente `lcd_controller`, descrito em VHDL. Este componente é responsável pela inicialização do LCD e serialização dos comandos, bem como dos tempos de execução de cada comando, gerando uma saída que indica quando o LCD está disponível para receber um novo comando. Os comandos suportados pelo controlador do LCD e que são utilizados por este componente estão listados e descritos no Anexo C.3.

Este componente é utilizado pelo `lcd_control`, descrito em VHDL, responsável por converter os comandos de escrita e de controlo feitos através de escritas para os respetivos portos, em comandos apropriados para o LCD.

Uma vez que o LCD funciona a uma velocidade inferior à do P4 e fica indisponível durante vários ciclos de relógio durante a sua inicialização e após a execução de cada comando, é utilizada uma fila FIFO com 64 posições onde são inseridos os comandos de escrita e de controlo antes de serem consumidos pelo controlador.

Isto permite que o programador possa fazer escritas consecutivas nos portos de escrita e controlo do mostrador, sem ter de salvaguardar entre cada escrita o tempo de execução dos comandos ao nível do hardware.

Por exemplo, para fazer uma escrita completa do mostrador são executados pelo controlador do LCD, 16 comandos para escrita do caráter e 2 para posicionamento do cursor no início da primeira e da segunda linha. Uma vez que o controlador demora cerca de 204 µs para consumir cada comando, isto totaliza 3,672 ms.

Para evitar um possível comportamento anómalo do mostrador causado pelo enchimento da fila, o programador apenas tem de garantir que não faz uma média superior a 272 escritas completas do mostrador por segundo. Isto é mais do que suficiente para um mostrador deste tipo, pois atualizações muito frequentes tornam impossível a leitura do seu conteúdo.

O consumo da fila FIFO é controlado por três sinais: o sinal *busy* do `lcd_control` que indica se o LCD está disponível para aceitar comandos, o sinal *empty* da fila FIFO que indica se há comandos para consumir, e o próprio sinal que ativa o consumo da fila, de forma a prevenir dois consumos consecutivos. Isto está demonstrado a Figura 6.15.

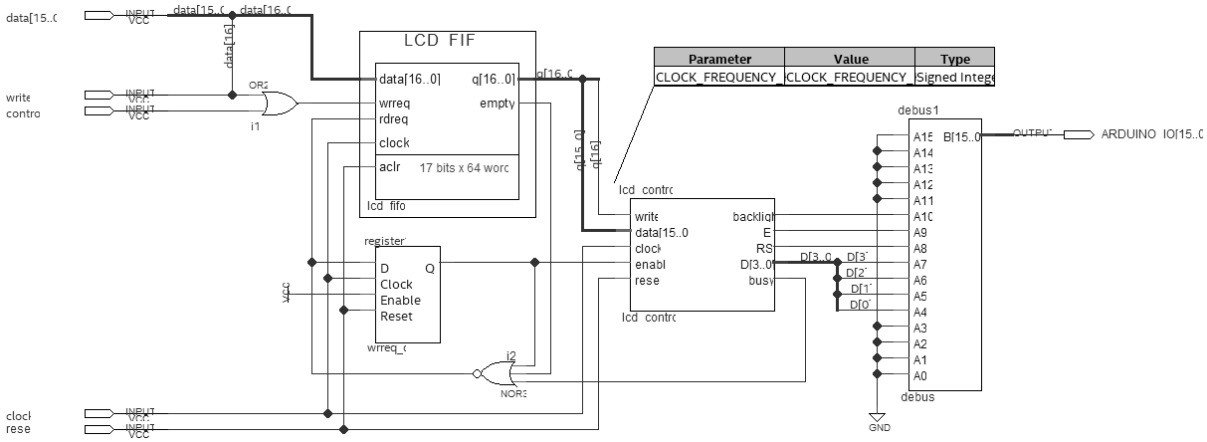


Figura 6.15. Circuito de controlo do LCD.

6.16 Mostradores Hexadecimais de 7 Segmentos

Cada mostrador possui um circuito de controlo que à semelhança do circuito de controlo dos *LEDs* contém um registo para guardar os valores escritos para o porto correspondente ao mostrador. Adicionalmente, existe um circuito que gera a saída para ativar individualmente cada um dos segmentos do mostrador de forma a gerar o número hexadecimal correspondente ao valor guardado no registo.

Quando o P4 se inicia, os mostradores encontram-se apagados até que seja feita uma escrita, mantendo-se ligados a partir daí até que o processador seja reinicializado. Isto é conseguido, através de um registo cuja saída controla um multiplexador, tal como está esquematizado na Figura 6.16.

6.17 Acelerómetro

A comunicação com o acelerómetro da placa utiliza uma interface SPI, uma interface de comunicação em série usada na comunicação entre microprocessadores e pequenos periféricos [31], e está implementada por componentes descritos em Verilog disponíveis na *Design Store* da Intel [32], aos quais foram feitas as alterações necessárias para obter os valores da aceleração em todos os eixos, uma vez

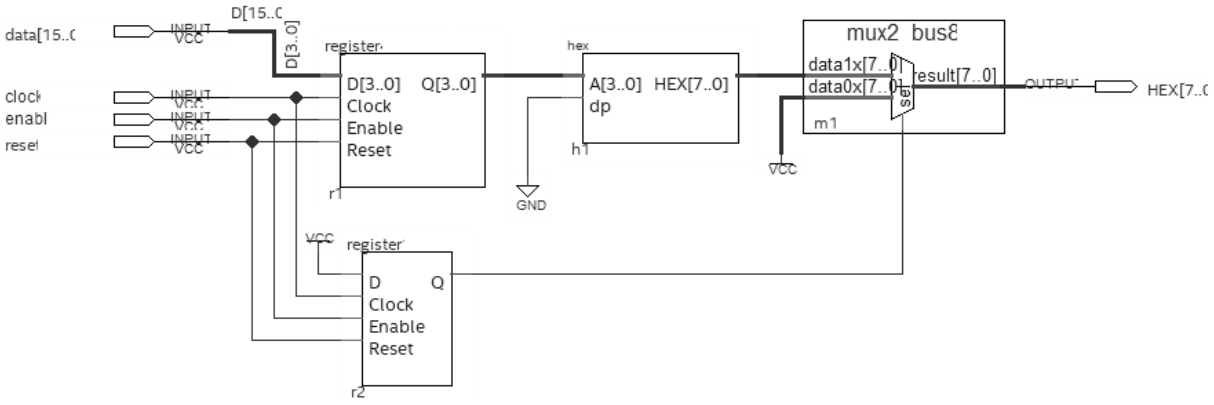


Figura 6.16. Circuito de controlo de um mostrador hexadecimal de 7 segmentos.

que apenas estava a ser feita a leitura do eixo dos X. Estes valores são guardados em registos que podem ser lidos usando os portos correspondentes a cada um dos eixos.

7 Avaliação da Solução

O sistema desenvolvido foi validado e avaliado em termos de recursos usados, desempenho e funcionamento. Foram feitos também alguns programas *assembly* para o P4 de forma a demonstrar o correto funcionamento do sistema e dos seus periféricos.

7.1 Simulação Funcional do P4

Devido à propriedade reconfigurável do hardware usado, a maior parte dos circuitos foram testados diretamente na FPGA, quer individualmente, usando as entradas e saídas da placa para definir os possíveis valores de entrada dos circuitos e observar os resultados, quer através de pequenos programas quando o sistema já estava suficientemente desenvolvido para os correr.

No entanto, esta forma de testar está limitada aos periféricos de saída ligados à FPGA, não permitindo ter uma visão total do estado interno do sistema. Para poder observar-se os valores dos sinais em várias partes do circuito e como estes evoluem ao longo do tempo, foi utilizada a ferramenta de *Simulation Waveform Editor* do *Quartus*.

Nesta ferramenta é possível correr uma simulação que gera um diagrama com a evolução do valor dos sinais que se quer observar ao longo dos vários ciclos de relógio, permitindo testar e validar a implementação em hardware do P4. Por exemplo, para testar o funcionamento do P4 ao realizar a operação de deslocamento à esquerda, criou-se o seguinte programa *assembly*:

```
MVI R1,1
SHL R1
```

Este programa, em código máquina, corresponde a:

```
1100100000000001
1000110001001000
```

Estes valores foram colocados na memória de programa e correu-se a simulação funcional do circuito do processador, a partir da qual se obteve o diagrama da Figura 7.1. Note-se que foram acrescentadas duas instruções `NOP` após a primeira instrução para evitar conflitos de dados, uma vez que o P4 ainda não possuía circuitos de *forwarding* de dados nesta simulação.

Pode verificar-se que o programa funciona corretamente analisando a evolução dos seguintes sinais:

- O sinal `PCOut` indica-nos o valor do PC no primeiro andar do *pipeline*. O seu valor é incrementado em uma unidade a cada flanco ascendente do sinal de relógio, como esperado para este programa.
- O sinal `RIOut` indica-nos a instrução a ser executada no segundo andar do *pipeline*. Começa por ser `0000000000000000`, que corresponde a um `NOP`. No segundo flanco ascendente do relógio, quando o sinal da primeira instrução chega ao segundo andar do *pipeline*, o seu valor é `1100100000000001`, que corresponde à primeira instrução do programa. Posteriormente, toma o valor `1000110001001000`, que corresponde à segunda instrução.

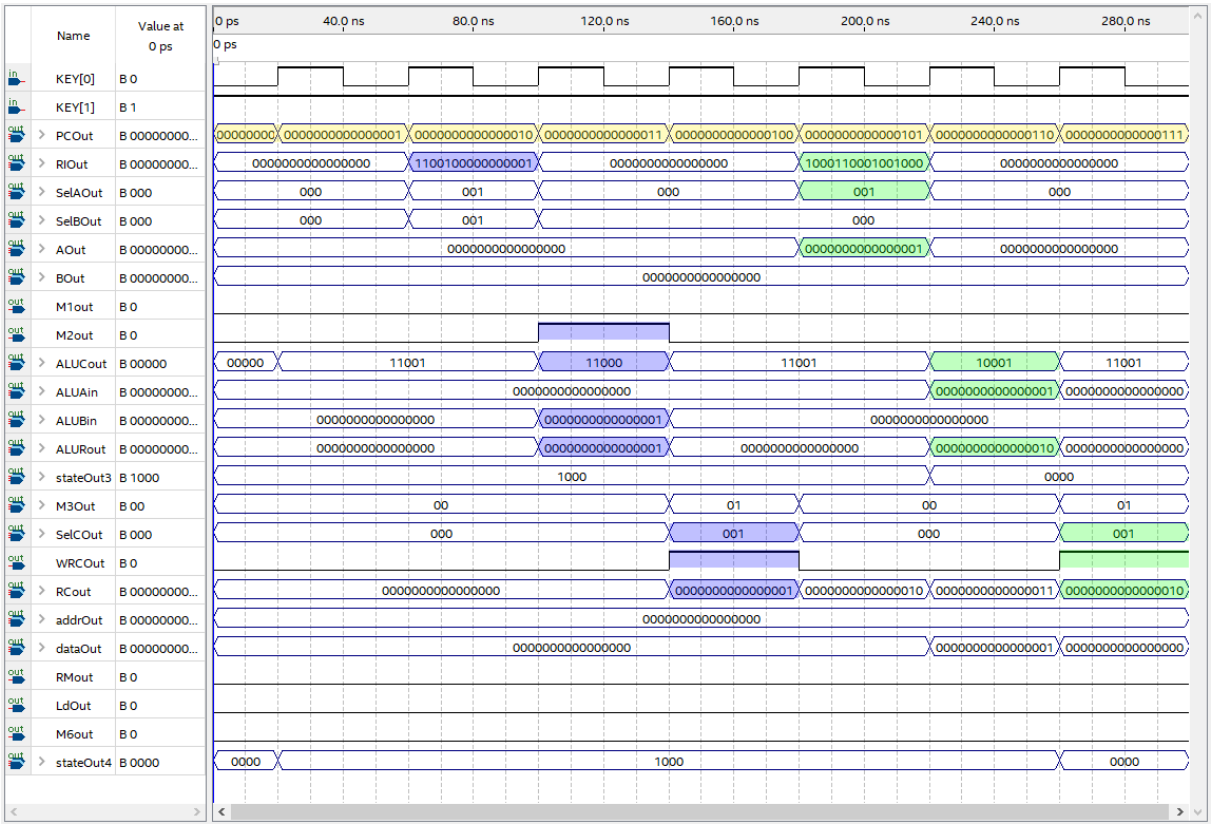


Figura 7.1. Janela da ferramenta *Simulation Waveform Editor* com o resultado da simulação do programa no P4.

- `SelAOut` é o sinal de controlo à saída do descodificador de instruções que indica o registo cujo valor será colocado na saída A do banco de registos, `AOut`, e que no andar seguinte será colocado na entrada A da ULA, `ALUAin`. Para a segunda instrução deste programa o valor deste sinal é `001`, visto estar a operar sobre o valor do registo `R1`.
- `AOut`, e que no andar seguinte será colocado na entrada A da ULA, `ALUAin`. Para a segunda instrução deste programa o valor deste sinal é `001`, visto estar a operar sobre o valor do registo `R1`.
- `ALUCout` é o sinal de controlo para a ULA, que no caso da instrução `MVI` tem valor `11000`, e no caso da instrução `SHL` tem valor `10001`.
- `M2Out` é o sinal de controlo do multiplexador `MUX2` no terceiro andar do *pipeline*, que tal como esperado, tem valor `1` na execução da instrução `MVI`.
- Verifica-se que quando a instrução passa ao quarto andar do *pipeline*, os sinais `M3Out`, `WRCOut`, `SelCOut` e `RCOut`, que correspondem respetivamente ao sinal de controlo do multiplexador `MUX3`, ao sinal que ativa a escrita no banco de registos, ao sinal que determina o registo a ser escrito, e ao valor que é escrito no registo, possuem os valores `01` (usar o resultado da ULA), `1` (ativar escrita), `001` (escrever no registo `R1`) e `0000000000000001` (escrever o valor 1).
- Após a primeira instrução chegar ao final do *pipeline*, a instrução `SHL` entra no segundo andar do *pipeline*. Mais uma vez os vários sinais tomam os valores esperados e quando esta instrução chega ao

final do *pipeline* o valor `0000000000000010` é escrito no registo `R1`, que corresponde ao deslocamento para a esquerda do valor `0000000000000001` anteriormente guardado no registo `R1`.

Portanto conclui-se que o sistema funciona corretamente na execução deste programa.

7.2 Portabilidade Para Outras Placas

O P4 pode ser portado para outras placas da Intel FPGA. Para atestar este facto, foi feito o porte para a placa Terasic DE0-CV[33] baseada numa FPGA Intel Cyclone V. Para tal, alterou-se a configuração do projeto no Quartus para o novo dispositivo e atualizou-se o mapeamento dos pinos para esta placa. Uma vez que o novo mapeamento de pinos utiliza os mesmos nomes para os pinos que têm a mesma função, quase não foi necessário fazer alterações ao nível dos pinos dentro do projeto. Foi no entanto necessário remover o circuito ADC, uma vez que a FPGA desta placa não o possui. Além disso, a placa também não possui acelerómetro nem ligação para a placa de expansão utilizada no sistema do P4, pelo que estes periféricos não estariam disponíveis num sistema que utilizasse esta placa.

7.3 Recursos Usados

Na Tabela 7.1 estão indicados os recursos da FPGA usados pelo P4. A segunda coluna corresponde à placa DE10-Lite, utilizada nesta solução, enquanto a terceira coluna corresponde à placa Terasic DE0-CV, para a qual foi feito o porte experimental.

Tabela 7.1. Recursos da FPGA usados pelo P4.

Família	MAX 10	Cyclone V
Dispositivo	10M50DAF484C6GES	5CEBA4F23C7
Total de elementos lógicos	3900 / 49 760 (8%)	1603 / 18 480 (9%)
Total de registos	1639	1617
Total de pinos	111 / 360 (31%)	111 / 224 (50%)
Total de pinos virtuais	0	0
Total de bits de memória	1 286 473 / 1 677 312 (77%)	1 286 313 / 3 153 920 (41%)
Elementos multiplicadores de 9-bits embebidos	0 / 288 (0%)	n/a
Total de PLL	4 / 4 (100%)	3 / 4 (75%)
Blocos de memória flash de utilizador	0 / 1 (0%)	n/a
Blocos conversores analógico-digital	1 / 2 (50%)	n/a

7.4 Frequência Máxima e Caminho Crítico

A frequência do relógio do P4 é de 80 MHz. Utilizando a ferramenta *TimeQuest Timing Analisys* determinou-se uma frequência máxima estimada de funcionamento de 83,08 MHz no modelo *Slow 1200mV 85C* e um caminho crítico entre a memória de dados e o PC, passando pelos MUX5, MUX6 e MUX7.

Este caminho crítico corresponde à implementação do mecanismo de *forwarding* de dados no caso em que haja uma instrução `LOAD` seguida de uma instrução de salto absoluto, como `JMP`, utilizando o mesmo registo da instrução anterior, o que implica que este mecanismo controle o MUX5 de forma a utilizar o valor à saída da memória de dados como o valor a ser carregado no PC.

Perante este resultado, foi estudada uma forma de reduzir o caminho crítico. Uma solução encontrada foi adicionar um registo à entrada `Ld`. Desta forma, obtém-se uma frequência máxima de 102,97 MHz, um aumento de cerca de 24%. Por outro lado, esta solução implica adicionar um atraso nos saltos e significa um aumento da complexidade do funcionamento do processador. Por este motivo, optou-se por manter a solução anterior.

7.5 Avaliação de Desempenho

O desempenho do processador pode ser calculado como o número total de instruções executadas para realizar uma determinada tarefa, multiplicado pelo número médio de ciclos de relógio por instrução (CPI) e pelo inverso da frequência do relógio.

Devido ao mecanismo de *pipelining* utilizado no P4 e à sua arquitetura mais simples, é expectável que se observe um aumento considerável de desempenho em relação ao P3. Por tratar-se de um processador RISC, o P4 deverá atingir valores de CPI próximos de 1.

Na Tabela 7.2 é apresentado um programa que soma as primeiras 64 posições de memória e escreve o resultado na memória, tanto para o P3 como para o P4.

Tabela 7.2. Programa que soma as primeiras 64 posições de memória e escreve o resultado na memória.

P3			P4		
	MOV	R2, 0040h		MVI	R2, 0040h
	MOV	R3, 0000h		MVI	R3, 0000h
L1:	ADD	R3, M[R2]	L1:	LOAD	R4, M[R2]
	DEC	R2		DEC	R2
	BR.NZ	L1		BR.NZ	L1
				ADD	R3, R3, R4
L2:	MOV	M[R1], R3	L2:	STOR	M[R1], R3

No P3, este programa corresponde a um total de 195 instruções executadas em 2070 ciclos de relógio, enquanto no P4 corresponde a um total de 259 instruções executadas em 259 ciclos de relógio, não considerando o tempo de enchimento e esvaziamento do *pipeline*. Isto dá um CPI de aproximadamente 10,6 para o P3 e de 1 para o P4.

Uma vez que a frequência de funcionamento do sistema do P4 é de 80 MHz, o tempo de execução deste programa é de 3,24 µs, o que corresponde a 80 MIPS. O tempo de execução no P3, com uma frequência de 6,25 MHz, é de 331,2 µs, correspondente a cerca de 0,6 MIPS.

Conclui-se que apesar de o P4 ter de executar mais instruções, consegue realizar esta tarefa com um desempenho 100 vezes superior em relação ao P3.

7.6 Demonstração

Na plataforma *web*[1] estão disponíveis alguns programas que demonstram o funcionamento do P4 e dos seus periféricos. A título de exemplo, ao selecionar-se o programa *Keyboard Interrupts* aparece no editor *assembly* o programa listado no Anexo E.2. Este programa ativa as interrupções do terminal, e sempre que uma tecla é premida, lê o valor do terminal e escreve esse valor de volta para o terminal, para o mostrador alfanumérico LCD, para os mostradores hexadecimais de 7 segmentos e para os *LEDs*.

Ao correr o programa no simulador, e de seguida pressionar as teclas `P` e `4` no terminal, obtém-se o resultado demonstrado na Figura 7.2. Este programa escreve no terminal e no mostrador alfanumérico LCD o texto introduzido através do teclado, neste caso `P4`. Além disso é possível observar o valor `34h` nos mostradores, e o valor `0000110100b` nos *LEDs*, que corresponde ao valor ASCII da última tecla premida. Na Figura 7.3 pode ver-se o resultado na placa após ter-se feito o mesmo procedimento.

Figura 7.2. Demonstração do programa *Keyboard Interrupts* no simulador.

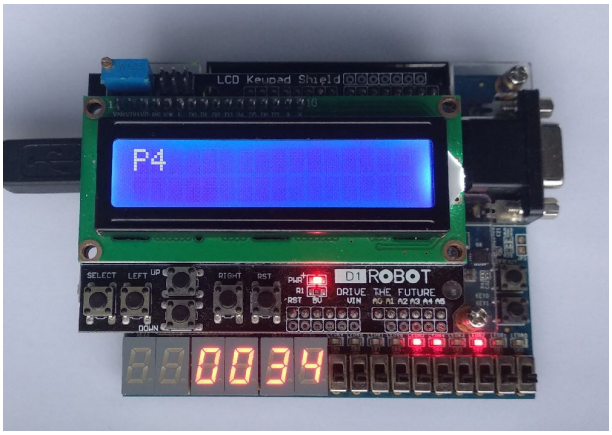


Figura 7.3. Demonstração do programa *Keyboard Interrupts* na placa.

Outro programa de demonstração disponível é o *Logo P4*, que imprime no terminal um logótipo do P4. Este programa, que pode ser consultado no Anexo E.1, foi gerado através da ferramenta de edição

[1] Plataforma *web* disponível no endereço: `https://web.tecnico.ulisboa.pt/dinismadeira/p4/`

de texto do terminal, apresentada no Capítulo 5.4.1. Na Figura 7.4 é possível ver o resultado num ecrã VGA ligado à placa após correr este programa.

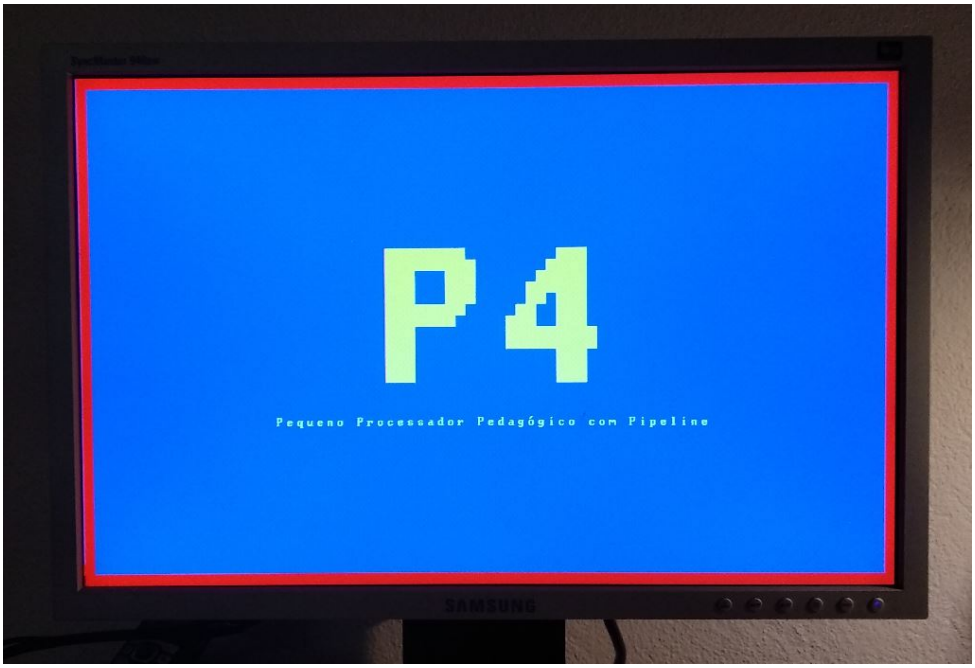


Figura 7.4. Demonstração do programa *Logo P4* na placa.

8 Conclusões e Trabalho Futuro

O desenvolvimento deste sistema didático para o P4 nasceu a partir da especificação apresentada no livro “Arquitectura de Computadores: dos Sistemas Digitais aos Microprocessadores” [1]. Ao longo do seu desenvolvimento foram encontrados desafios e adotadas soluções. Neste capítulo serão feitas algumas reflexões sobre o sistema final e possíveis desenvolvimentos futuros.

8.1 Conclusões

Os processadores RISC são tradicionalmente utilizados em sistemas embebidos que se têm tornado cada vez mais ubíquos. Embora estes processadores sejam mais limitados ao nível da complexidade das operações executadas por cada instrução e isso se traduza num maior número de instruções necessárias para realizar operações complexas, que numa arquitetura CISC são normalmente realizadas por uma única instrução, a simplicidade das operações realizadas por cada instrução, aliadas à técnica de *pipelining*, que explora o paralelismo na execução destas operações, os processadores RISC conseguem obter desempenhos semelhantes aos processadores CISC.

O P4, proposto no livro [1], está quase totalmente especificado. Ainda assim, para a sua implementação em hardware, foi necessário definir algumas questões arquiteturais que não foram contempladas no livro. Também foram feitas algumas alterações pelo facto de estas permitirem fazer uma realização em hardware mais simples. Foi ainda implementado um sistema de interrupções que não estava previsto no livro.

A implementação em hardware foi concebida através de uma descrição ao nível de registos e portas lógicas, tanto através de uma interface visual de diagramas de blocos como através da linguagem de descrição de hardware VHDL e foi implementada em FPGA.

Além da implementação em hardware, foram também criadas ferramentas para criar, assemblar e depurar programas *assembly*, bem como simulá-los em software com uma interface que representa o sistema real.

Esta solução permite estudar um processador RISC desde a sua arquitetura e implementação ao nível de registos e portas lógicas, até à execução de programas *assembly* num sistema real.

O facto de se tratar de um processador não comercial, criado apenas com o propósito de ser implementado num sistema didático, aliado ao facto de se tratar de um processador RISC, permite manter um nível de complexidade suficientemente acessível para o seu estudo no âmbito da unidade curricular de Introdução à Arquitetura de Computadores, ao contrário de outras soluções já existentes que revelam um nível de complexidade que constitui um entrave à compreensão do funcionamento básico do processador.

Este sistema vem inovar o ensino de Arquitetura de Computadores, apresentando uma arquitetura RISC em alternativa à alternativa CISC do P3. Por outro lado o P3 está implementado em hardware mais antigo que se torna cada vez mais difícil de manter em laboratório à medida que as suas interfaces entram em desuso. Futuramente o P3 pode ser adaptado à placa FPGA utilizada para o P4, o que

permitiria ter um sistema didático para um processador CISC e RISC utilizando a mesma solução de hardware.

8.2 Trabalho Futuro

Embora o trabalho esteja concluído e pronto a funcionar em ambiente laboratorial, a forma modular como ele foi realizado permite serem realizados outros trabalhos sobre ele. A implementação em hardware e as ferramentas de software estão disponíveis num repositório GitHub, acessível através do endereço `https://github.com/dinismadeira/p4`, o que facilitará alterações futuras e contribuições independentes.

Como se verificou através do porte experimental para a placa Terasic DE0-CV, é possível portar o P4 para diferentes placas com relativa facilidade, deixando aberta a possibilidade de portar o P4 para placas que ofereçam diferentes periféricos. Além disso, poderão ser adicionados novos periféricos ao atual sistema do P4 através do conector de expansão compatível com *Arduino Uno* ou do conector GPIO.

Ao nível do ambiente de desenvolvimento, poderão ser feitos melhoramentos ao simulador de forma a ter-se uma melhor visualização do funcionamento do *pipeline*, bem como a possibilidade de alterar o conteúdo dos registos e da memória, que são duas funcionalidades encontradas em alguns dos simuladores analisados no Capítulo 2. A depuração do sistema real também poderá ser melhorada através de uma funcionalidade que permita definir pontos de paragem na execução de programas no P4. Poderão ser criadas mais ferramentas adicionais, como um editor da palete de cores utilizada no terminal, e a ferramenta de edição do texto do terminal poderia passar a contemplar também ferramentas de desenho, que poderia ser útil para criar ecrãs para jogos, por exemplo.

O P4, sendo um processador RISC, requer mais instruções para executar as mesmas operações em relação ao P3, que é um processador CISC, o que significa um maior esforço por parte de um programador de *assembly* para um processador RISC. Isto normalmente não é problemático quando se programa em linguagens de alto nível que são depois compiladas para *assembly*. Para facilitar a criação de programas para o P4, poderia ser desenvolvido um compilador que compilasse uma linguagem de mais alto nível para o *assembly* do P4. Isto abriria também a possibilidade de utilizar o P4 como um sistema didático no estudo do funcionamento de compiladores.

Referências

1. Guilherme Arroz, José Monteiro e Arlindo Oliveira: Arquitectura de Computadores: dos Sistemas Digitais aos Microprocessadores. 3.ª Edição. IST Press (dezembro 2014)
2. RISC vs CISC, https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/. Último acesso: 15/04/2019.
3. Andrew Moore com Ron Wilson: *FPGAs For Dummies, 2nd Intel Special Edition*, https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/misc/fpgas_for_dummies_ebook.pdf. Último acesso: 30/05/2019.
4. Nios II, https://www.altera.com/niosII. Último acesso: 30/05/2018.
5. RISC-V, https://riscv.org/. Último acesso: 30/05/2018.
6. *MIPS – Market-leading RISC CPU IP processor solutions*, https://www.mips.com/. Último acesso: 30/05/2018.
7. *Architecting a Smarter World – Arm*, https://www.arm.com/. Último acesso: 30/05/2018.
8. *John Hennessy: Engineering Solutions 10/00*, https://news.stanford.edu/news/2000/october18/hensci-1018.html. Último acesso: 30/05/2018.
9. ViSiMIPS, https://ieeexplore.ieee.org/document/6028756/. Último acesso: 29/05/2018.
10. SPIM, http://spimsimulator.sourceforge.net/. Último acesso: 29/05/2018.
11. MARS, http://courses.missouristate.edu/KenVollmar/mars/. Último acesso: 29/05/2018.
12. WinMips64, http://indigo.ie/~mscott/. Último acesso: 29/05/2018.
13. *ARM Holdings eager for PC and server expansion*, https://www.theregister.co.uk/2011/02/01/arm_holdings_q4_2010_numbers/. Último acesso: 29/05/2018.
14. *Intel® FPGA University Program*, https://www.intel.com/content/www/us/en/programmable/support/training/university/overview.html. Último acesso: 29/05/2018.
15. *Simple 8-bit Assembler Simulator*, https://schweigi.github.io/assembler-simulator/. Último acesso: 21/04/2019.
16. Linguagem NASM, https://www.nasm.us/xdoc/2.10.09/html/nasmdoc3.html. Último acesso: 21/04/2019.
17. P3, http://algos.inesc-id.pt/arq-comp/?Material_Did%C3%A1tico___Processador_P3. Último acesso: 21/04/2019.
18. *P3JS Assembler and Simulator*, https://p3js.goncalomb.com/. Último acesso: 30/05/2018.
19. Terasic DE10-Lite, http://de10-lite.terasic.com/. Último acesso: 29/05/2018.
20. Intel FPGAs, https://www.intel.com/content/www/us/en/products/programmable.html. Último acesso: 01/05/2019.
21. *Arduino Uno Rev3*, https://store.arduino.cc/arduino-uno-rev3. Último acesso: 01/05/2019.
22. NW.js, https://nwjs.io/. Último acesso: 08/03/2019.
23. nwjs-builder-phoenix, https://github.com/evshiron/nwjs-builder-phoenix. Último acesso: 08/03/2019.
24. Node.js, https://nodejs.org/. Último acesso: 02/02/2019.
25. Acuidade da visão humana à cor azul, https://nfggames.com/games/NTSC/visual.shtm. Último acesso: 26/04/2019.
26. *Datasheet* do mostrador alfanumérico LCD Hitachi modelo HD44780U, https://www.sparkfun.com/datasheets/LCD/HD44780.pdf. Último acesso: 19/03/2019.
27. *Memory Initialization File (.mif)*, https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/reference/glossary/def_mif.htm. Último acesso: 07/04/2019.
28. Página da *wiki* da *DigiKey* sobre a interface PS/2 para um teclado, https://www.digikey.com/eewiki/pages/viewpage.action?pageId=28278929. Último acesso: 22/02/2019.

29. Temporizações de Vídeo: VGA, SVGA, 720P, 1080P, https://timetoexplore.net/blog/video-timings-vga-720p-1080p. Último acesso: 10/04/2019.
30. *Switch Debouncing*, https://electrosome.com/switch-debouncing/. Último acesso: 27/04/2019.
31. *Serial Peripheral Interface* (SPI), https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi/all. Último acesso: 10/04/2019.
32. Exemplo *GSensor* na *Design Store* da Intel, https://fpgacloud.intel.com/devstore/platform/16.1.0/Standard/gsensor-max10-de10-lite/. Último acesso: 18/01/2019.
33. Terasic DE0-CV, http://de0-cv.terasic.com. Último acesso: 11/03/2019.

9 Anexos

A Instruções Assembly

Salvo indicação em contrário, os operandos são registos.

ADD *Flags*: ZCNO

Formato: `ADD op1, op2, op3`

Ação: $op1 \leftarrow op2 + op3$

Descrição: Soma op3 a op2 e guarda o resultado em op1.

ADDC *Flags*: ZCNO

Formato: `ADDC op1, op2, op3`

Ação: $op1 \leftarrow op2 + op3 + C$

Descrição: Soma op3 a op2 e ao valor do bit de estado transporte, guardando o resultado em op1.

AND *Flags*: ZN

Formato: `AND op1, op2, op3`

Ação: $op1 \leftarrow op2 \wedge op3$

Descrição: Faz a disjunção lógica bit a bit entre op2 a op3 e guarda o resultado em op1.

BR *Flags*: Nenhuma

Formato: `BR <deslocamento>`

Ação: $PC \leftarrow PC + deslocamento$

Descrição: Salto relativo incondicional. A nova posição é determinada em relação à posição da instrução de salto somando-se o valor de <deslocamento>. O valor tem de estar compreendido entre -128 e 127. É possível utilizar uma etiqueta de forma a realizar o salto para a instrução precedida por essa etiqueta.

BR.*cond* *Flags*: Nenhuma

Formato: `BR.cond <deslocamento>`

Ação: $PC \leftarrow PC + deslocamento$

Descrição: Salto relativo condicional. Funciona de modo similar ao salto relativo incondicional com a diferença de que caso a condição não se verifique esta instrução não surte qualquer efeito. As condições possíveis são:

Condição	Zero	Negativo	Positivo	Transporte	Excesso
Verdadeiro	`BR.Z`	`BR.N`	`BR.P`	`BR.C`	`BR.O`
Falso	`BR.NZ`	`BR.NN`	`BR.NP`	`BR.NC`	`BR.NO`

CLC *Flags*: C

Formato: `CLC`

Ação: $C \leftarrow 0$

Descrição: *Clear Carry*. Coloca o bit de estado transporte a 0.

CMC *Flags*: C

Formato: `CMC`

Ação: $C \leftarrow \overline{C}$

Descrição: *Complement Carry*. Complementa o valor do bit de estado de transporte.

CMP *Flags*: ZCNO

Formato: `CMP op1, op2`

Ação: $op1 - op2$

Descrição: Compara os operandos op1 e op2, atualizando os bits de estado. É equivalente à instrução `SUB R0, op1, op2`. É normalmente utilizada antes de saltos condicionais.

COM *Flags*: ZN

Formato: `COM op`

Ação: $op \leftarrow \overline{op}$

Descrição: Faz o complemento bit a bit de op.

DEC *Flags*: ZCNO

Formato: `DEC op`

Ação: $op \leftarrow op - 1$

Descrição: Decrementa op em uma unidade.

DSI *Flags*: E

Formato: `DSI`

Ação: $E \leftarrow 0$

Descrição: *Disable Interrupts*. Coloca o bit de estado E a 0, inibindo o tratamento de interrupções.

ENI *Flags*: E

Formato: `ENI`

Ação: $E \leftarrow 1$

Descrição: *Enable Interrupts*. Coloca o bit de estado E a 1, ativando o tratamento de interrupções.

INC *Flags*: ZCNO

Formato: `INC`

Ação: $op \leftarrow op + 1$

Descrição: Incrementa op em uma unidade.

INT *Flags*: ZCNOE

Formato: `INT <constante>`

Ação: $PC \leftarrow 7F00h + constante, \quad PC_E \leftarrow PC, \quad RE_E \leftarrow RE$

Descrição: Gera uma interrupção independentemente do valor do bit de estado E. O endereço da rotina de tratamento da interrupção é calculado como a soma entre 7F00h e <constante>, que deverá estar compreendida entre 0 e 255. Guarda o valor do *Program Counter* e o Registo de Estado para ser utilizado mais tarde pela instrução `RTI`.

JAL *Flags*: Nenhuma

Formato: `JAL <endereço>`

Ação: $PC \leftarrow$ ***endereço***, $\quad R7 \leftarrow PC$

Descrição: Salto absoluto incondicional. A nova posição é determinada pelo valor de <endereço>. A posição atual é guardada em R7. É possível utilizar uma etiqueta de forma a realizar o salto para a instrução precedida por essa etiqueta.

JAL.*cond* *Flags*: Nenhuma

Formato: `JAL.`*`cond`* `<endereço>`

Ação: $PC \leftarrow$ ***endereço***, $\quad R7 \leftarrow PC$

Descrição: Salto absoluto condicional. Funciona de modo similar ao salto absoluto incondicional com a diferença de que caso a condição não se verifique esta instrução não surte qualquer efeito. As condições possíveis são:

Condição	Zero	Negativo	Positivo	Transporte	Excesso
Verdadeiro	`JAL.Z`	`JAL.N`	`JAL.P`	`JAL.C`	`JAL.O`
Falso	`JAL.NZ`	`JAL.NN`	`JAL.NP`	`JAL.NC`	`JAL.NO`

JMP *Flags*: Nenhuma

Formato: `JMP <endereço>`

Ação: $PC \leftarrow$ ***endereço***

Descrição: Salto absoluto incondicional. A nova posição é determinada pelo valor de <endereço>. É possível utilizar uma etiqueta de forma a realizar o salto para a instrução precedida por essa etiqueta.

JMP.*cond* *Flags*: Nenhuma

Formato: `JMP.`*`cond`* `<endereço>`

Ação: $PC \leftarrow$ ***endereço***

Descrição: Salto absoluto condicional. Funciona de modo similar ao salto absoluto incondicional com a diferença de que caso a condição não se verifique esta instrução não surte qualquer efeito. As condições possíveis são:

Condição	Zero	Negativo	Positivo	Transporte	Excesso
Verdadeiro	`JMP.Z`	`JMP.N`	`JMP.P`	`JMP.C`	`JMP.O`
Falso	`JMP.NZ`	`JMP.NN`	`JMP.NP`	`JMP.NC`	`JMP.NO`

LOAD *Flags*: Nenhuma

Formato: `LOAD op1, op2`

Ação: $op1 \leftarrow op2$

Descrição: Copia o valor de op2 para op1. op2 deverá ser uma posição de memória endereçada por registo indireto `M[Rx]`, em que o endereço dessa posição é o conteúdo do registo `Rx`.

MOV *Flags*: Nenhuma

Formato: `MOV op1, op2`

Ação: $op1 \leftarrow op2$

Descrição: Copia o valor de op2 para op1.

MVI *Flags*: Nenhuma

Formato: `MVI op1, op2`

Ação: $op1 \leftarrow op2$

Descrição: Copia o valor de op2 para op1. op2 deverá ser uma constante compreendida entre -32768 e 65535. Para valores superiores a 255 ou inferiores a -128 esta instrução é automaticamente convertida num par de instruções MVIH e MVIL.

MVIH *Flags*: Nenhuma

Formato: `MVI op1, op2`

Ação: $op1 \leftarrow (op1 \wedge 00FFh) \vee (op2 << 8)$

Descrição: Copia o octeto de maior peso de op2 para o octeto de maior peso de op1. op2 deverá ser uma constante compreendida entre -128 e 255.

MVIL *Flags*: Nenhuma

Formato: `MVI op1, op2`

Ação: $op1 \leftarrow (op1 \wedge FF00h) \vee op2$

Descrição: Copia o octeto de menor peso de op2 para o octeto de menor peso de op1. op2 deverá ser uma constante compreendida entre -128 e 255.

NEG *Flags*: ZCNO

Formato: `NEG op`

Ação: $op \leftarrow -op$

Descrição: Inverte o sinal de op. Equivalente à instrução `SUB op, R0, op`.

NOP *Flags*: Nenhuma

Formato: `NOP`

Ação: $\emptyset$

Descrição: *No operation*. Não tem qualquer efeito.

OR *Flags*: ZN

Formato: `OR op1, op2, op3`

Ação: $op1 \leftarrow op2 \vee op3$

Descrição: Faz a conjunção lógica bit a bit entre op2 a op3 e guarda o resultado em op1.

ROL *Flags*: ZCN

Formato: `ROL op`

Ação: $op \leftarrow (op << 1) \vee (op >> 15)$

Descrição: *Rotate Left*. Faz a rotação à esquerda dos bits de op. Similar ao deslocamento à esquerda com a diferença de que o bit mais à esquerda passa a ser o bit mais à direita.

ROLC *Flags*: ZCN

Formato: `ROLC op`

Ação: $op \leftarrow (op << 1) \vee C, \quad C \leftarrow op >> 15$

Descrição: *Rotate Left with Carry*. Similar à rotação à esquerda com a diferença de que o bit de estado transporte passa a ser o bit mais à direita e o bit mais à esquerda é colocado no bit de estado transporte.

ROR *Flags*: ZCN

Formato: `ROR op`

Ação: $op \leftarrow (op >> 1) \vee ((op \wedge 1) << 15)$

Descrição: *Rotate Right*. Faz a rotação à direita dos bits de op. Similar ao deslocamento à direita com a diferença de que o bit mais à direita passa a ser o bit mais à esquerda.

RORC *Flags*: ZCN

Formato: `ROLC op`

Ação: $op \leftarrow (op >> 1) \vee (C << 15), \quad C \leftarrow op \wedge 1$

Descrição: *Rotate Right with Carry*. Similar à rotação à direita com a diferença de que o bit de estado transporte passa a ser o bit mais à esquerda e o bit mais à direita é colocado no bit de estado transporte.

RTI *Flags*: ZCNOE

Formato: `RTI`

Ação: $PC \leftarrow PC_E, \quad RE \leftarrow RE_E$

Descrição: *Return from Interrupt*. Executa um salto para a instrução que iria ser executada antes do

tratamento da interrupção e repõe o registo de estado com o valor do estado antes do tratamento da interrupção.

SHL *Flags*: ZCN

Formato: `SHL op`

Ação: $op \leftarrow op << 1, \quad C \leftarrow op >> 15$

Descrição: *Shift Left*. Faz o deslocamento à esquerda dos bits de op. É inserido um 0 na posição mais à direita e o bit mais à esquerda é colocado no bit de estado transporte.

SHLA *Flags*: ZCNO

Formato: `SHLA op`

Ação: $op \leftarrow op << 1, \quad C \leftarrow op >> 15$

Descrição: *Shift Left Arithmetic*. Similar ao deslocamento à esquerda com a diferença de que os bits de estado correspondente às operações aritméticas são atualizados.

SHR *Flags*: ZCN

Formato: `SHR op`

Ação: $op \leftarrow op >> 1, \quad C \leftarrow op \wedge 1$

Descrição: *Shift Right*. Faz o deslocamento à direita dos bits de op. É inserido um 0 na posição mais à esquerda e o bit mais à direita é colocado no bit de estado transporte.

SHRA *Flags*: ZCNO

Formato: `SHRA op`

Ação: $op \leftarrow op >> 1, \quad C \leftarrow op \wedge 1$

Descrição: *Shift Right Arithmetic*. Similar ao deslocamento à direita com a diferença de que os bits de estado correspondente às operações aritméticas são atualizados.

STC *Flags*: C

Formato: `STC`

Ação: $C \leftarrow 1$

Descrição: *Set Carry*. Coloca o bit de estado transporte a 1.

STOR *Flags*: Nenhuma

Formato: `STOR op1, op2`

Ação: $op1 \leftarrow op2$

Descrição: Copia o valor de op2 para op1. op1 deverá ser uma posição de memória endereçada por registo indireto `M[Rx]`, em que o endereço dessa posição é o conteúdo do registo `Rx`.

SUB *Flags*: ZCNO

Formato: `SUB op1, op2, op3`

Ação: $op1 \leftarrow op2 - op3$

Descrição: Subtrai op3 a op2 e guarda o resultado em op1.

SUBB *Flags*: ZCNO

Formato: `SUBB op1, op2, op3`

Ação: $op1 \leftarrow op2 - op3 - (1 - C)$

Descrição: Subtrai op3 e o complemento do valor do bit de estado transporte a op2, guardando o resultado em op1.

TEST *Flags*: ZN

Formato: `TEST op1, op2`

Ação: $op1 \wedge op2$

Descrição: Testa os bits dos operandos op1 e op2, atualizando os bits de estado. É equivalente à instrução `AND R0, op1, op2`. É normalmente utilizada antes de saltos condicionais.

XOR *Flags*: ZN

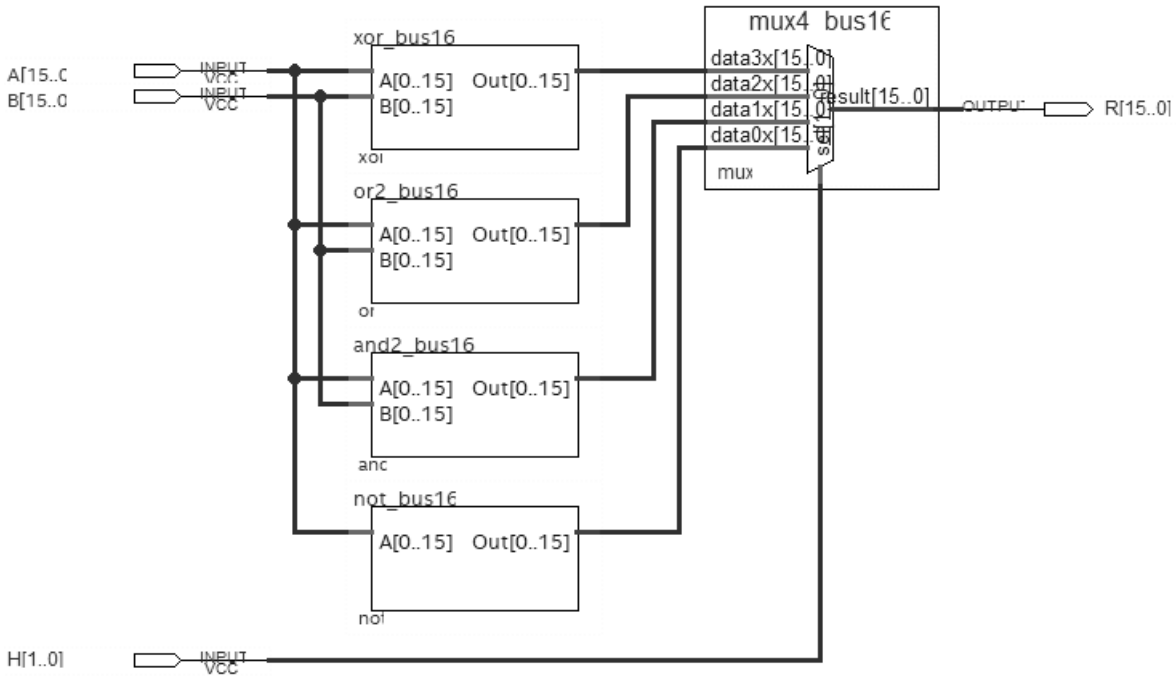
Formato: `XOR op1, op2, op3`

Ação: $op1 \leftarrow op2 \oplus op3$

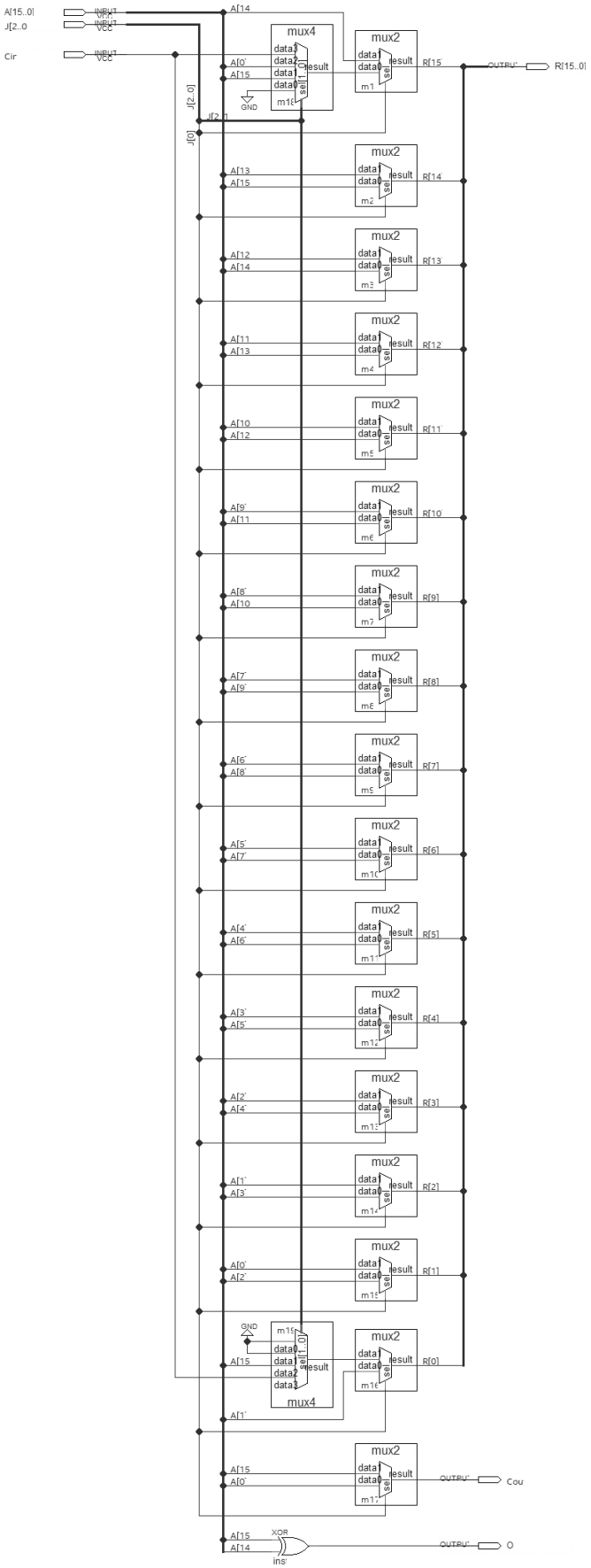
Descrição: Faz a conjunção lógica exclusiva bit a bit entre op2 e op3 e guarda o resultado em op1.

B Diagramas de Blocos

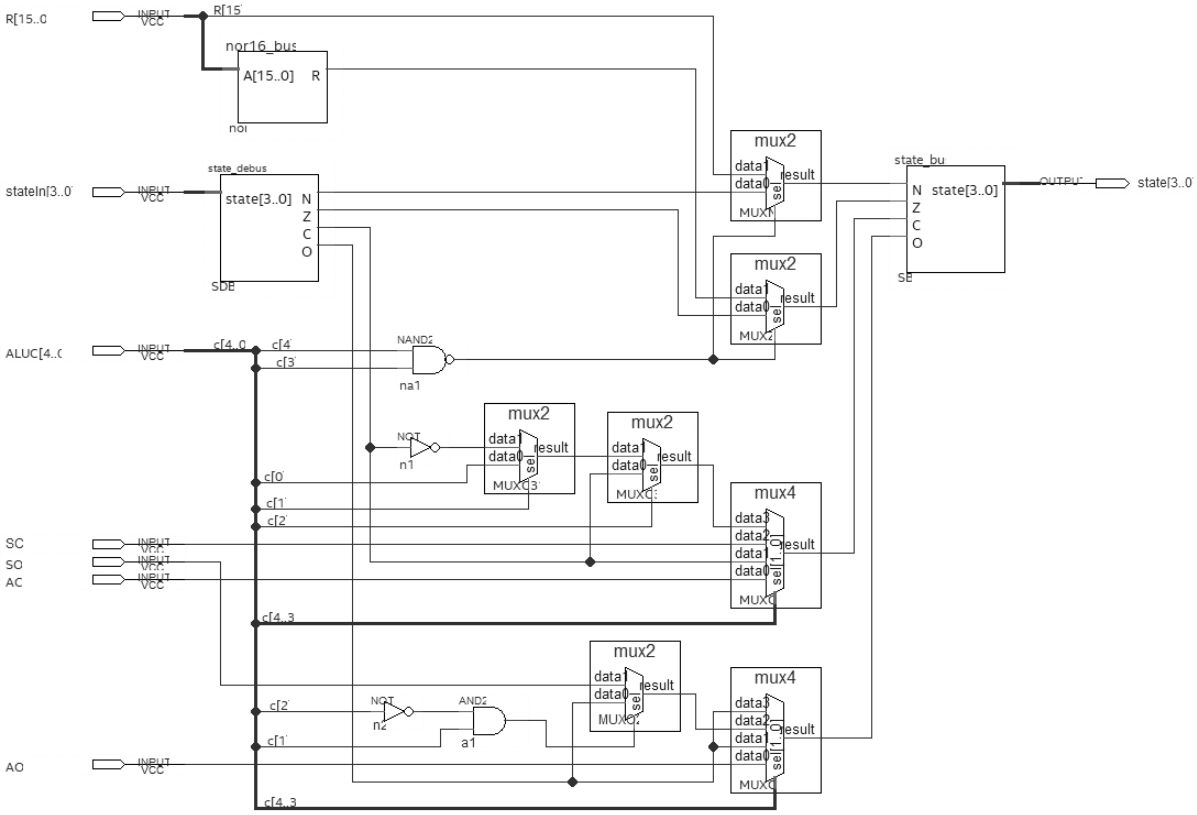
B.1 Unidade Lógica



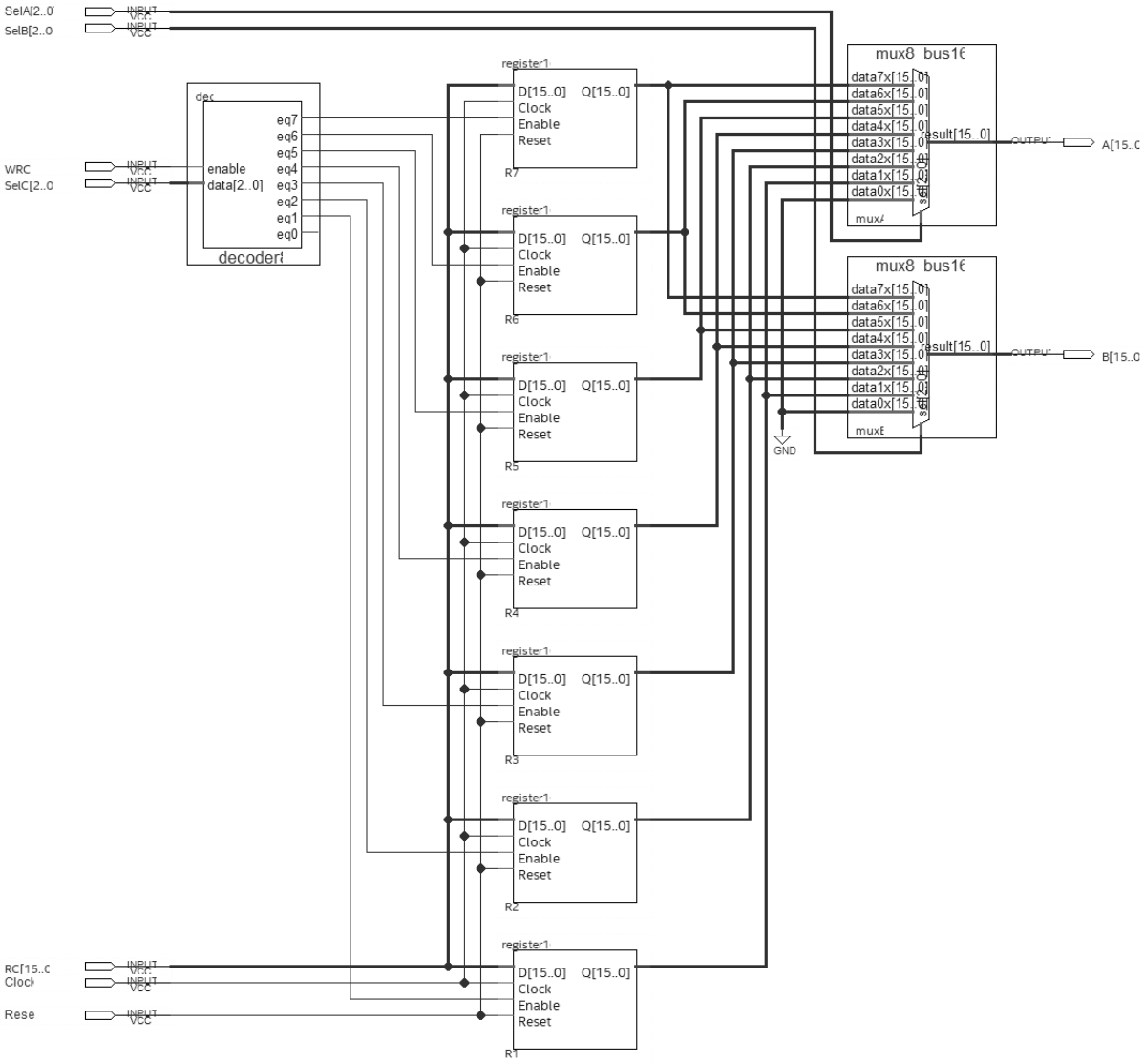
B.2 Unidade de Deslocamento



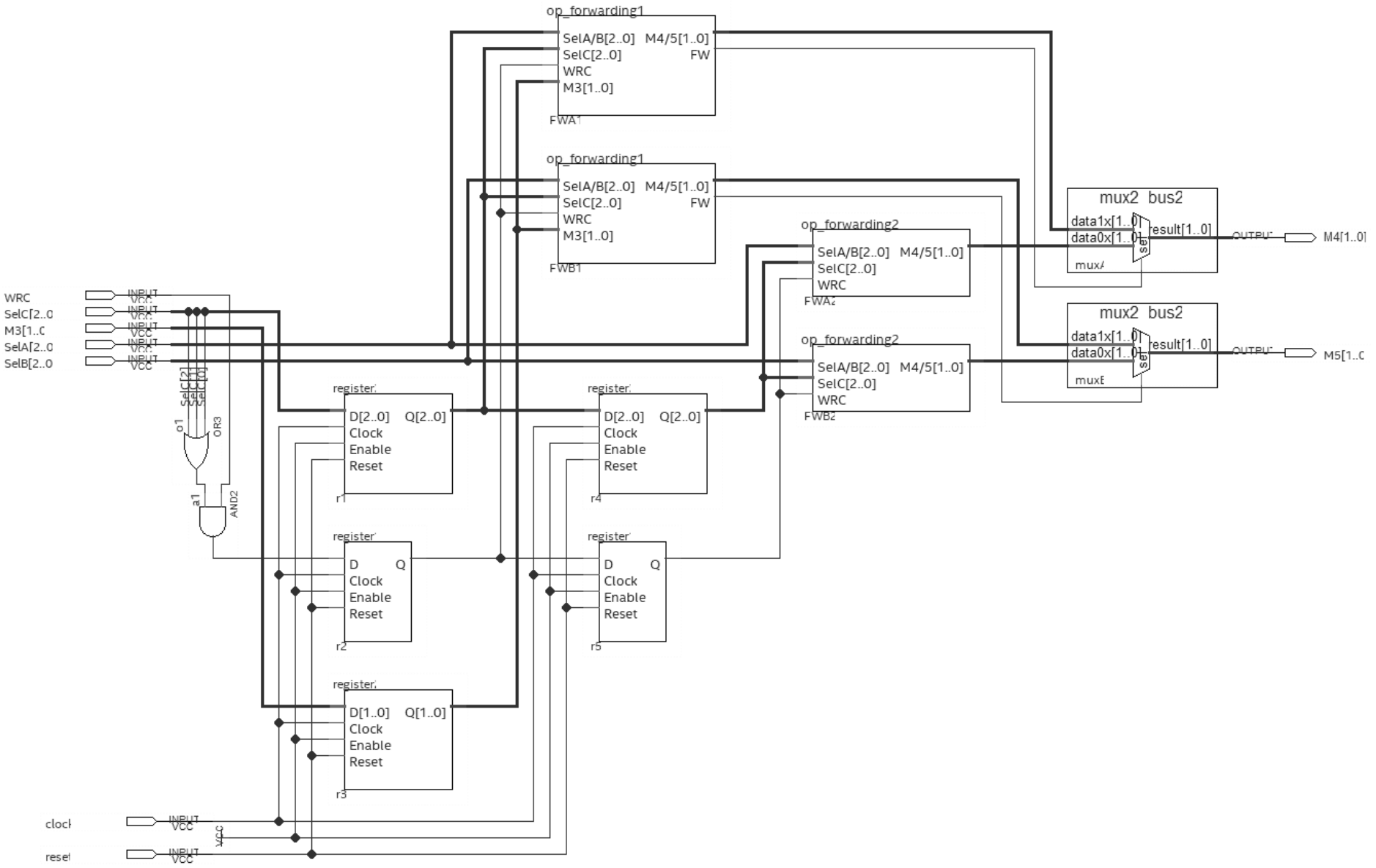
B.3 Controlador do Registo de Estado



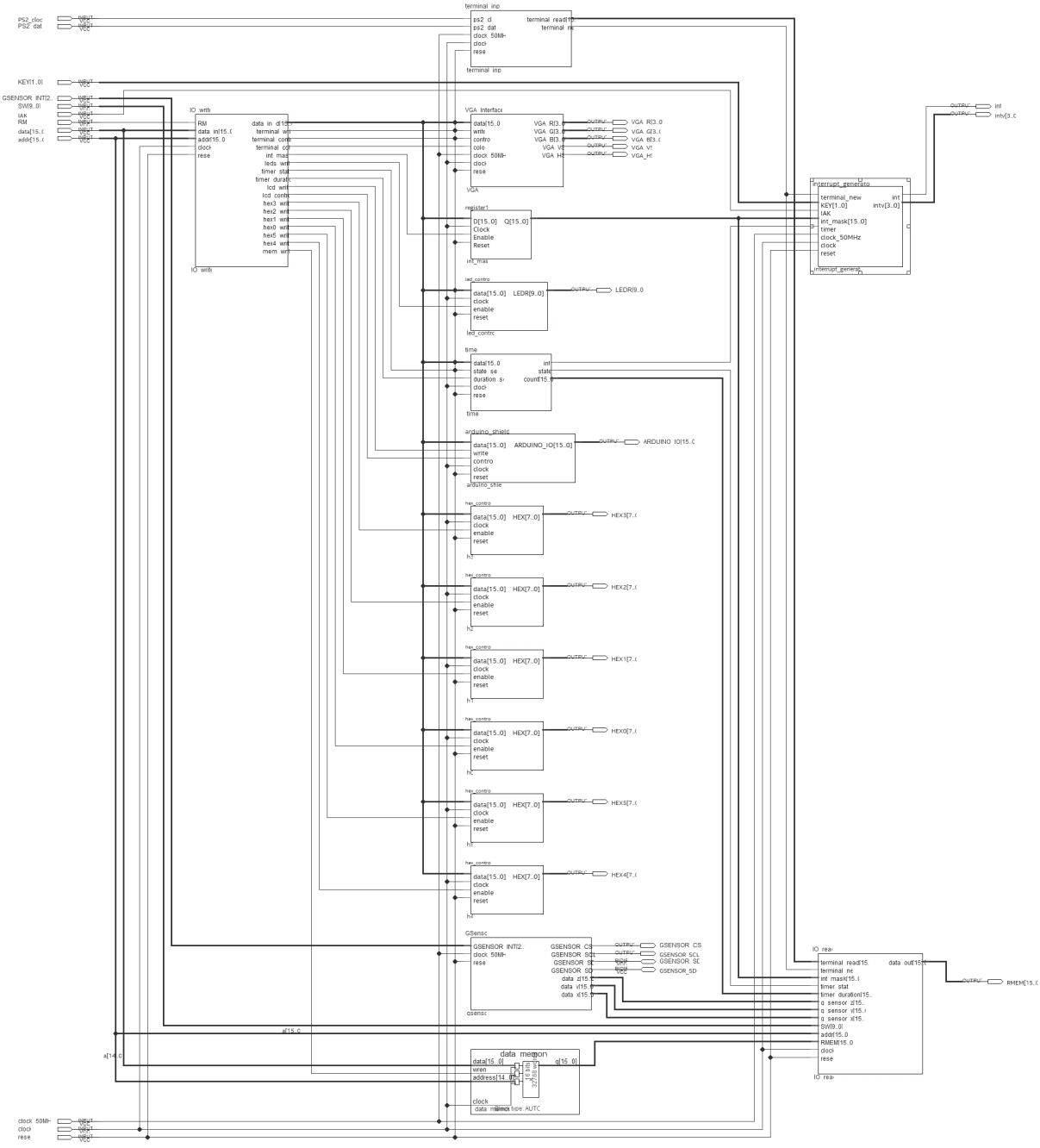
B.4 Banco de Registos



B.5 Circuito de *Forwarding* de Dados.



B.6 Entradas e Saídas (IO)



C Tabelas

C.1 *Scan Codes* de Teclado com Esquema Português (Portugal)

Tecla	*Scan Codes*	
	Premida	Libertada
A	1C	F0 1C
B	32	F0 32
C	21	F0 21
Ç	4C	F0 4C
D	23	F0 23
E	24	F0 24
F	2B	F0 2B
G	34	F0 34
H	33	F0 33
I	43	F0 43
J	3B	F0 3B
K	42	F0 42
L	4B	F0 4B
M	3A	F0 3A
N	31	F0 31
O	44	F0 44
P	4D	F0 4D
Q	15	F0 15
R	2D	F0 2D
S	1B	F0 1B
T	2C	F0 2C
U	3C	F0 3C
V	2A	F0 2A
W	1D	F0 1D
X	22	F0 22
Y	35	F0 35
Z	1A	F0 1A

Tecla	*Scan Codes*	
	Premida	Libertada
0	45	F0 45
1	16	F0 16
2	1E	F0 1E
3	26	F0 26
4	25	F0 25
5	2E	F0 2E
6	36	F0 36
7	3D	F0 3D
8	3E	F0 3E
9	46	F0 46
’	4E	F0 4E
«	55	F0 55
+	54	F0 54
´	5B	F0 5B
º	52	F0 52
~	5D	F0 5D
<	61	F0 61
,	41	F0 41
.	49	F0 49
-	4A	F0 4A
Espaço	29	F0 29

Tecla	*Scan Codes*	
	Premida	Libertada
Escape	76	F0 76
F1	5	F0 05
F2	6	F0 06
F3	4	F0 04
F4	0C	F0 0C
F5	3	F0 03
F6	0B	F0 0B
F7	83	F0 83
F8	0A	F0 0A
F9	1	F0 01
F10	9	F0 09
F11	78	F0 78
F12	7	F0 07
Backspace	66	F0 66
Tab	0D	F0 0D
Enter	5A	F0 5A
Caps Lock	58	F0 58
Left Shift	12	F0 12
Right Shift	59	F0 59
Left Ctrl	14	F0 14
Win Left	E0 1F	E0 F0 1F
Left Alt	11	F0 11
Right Alt	E0 11	E0 F0 11
Win Right	E0 27	E0 F0 27
Menu	E0 2F	E0 F0 2F
Right Ctrl	E0 14	E0 F0 14

Tecla	*Scan Codes*	
	Premida	Libertada
Print Screen	E0 12 E0 7C	E0 F0 7C E0 F0 12
Scroll Lock	7E	F0 7E
Pause	E1 14 77 E1 F0 14 F0 77	-
Insert	E0 70	E0 F0 70
Home	E0 6C	E0 F0 6C
Page Up	E0 7D	E0 F0 7D
Delete	E0 71	E0 F0 71
End	E0 69	E0 F0 69
Page Down	E0 7A	E0 F0 7A
Up Key	E0 75	E0 F0 75
Left Key	E0 6B	E0 F0 6B
Down Key	E0 72	E0 F0 72
Right Key	E0 74	E0 F0 74
Num Lock	77	F0 77
KP /	E0 4A	E0 F0 4A
KP *	7C	F0 7C
KP -	7B	F0 7B
KP 0	70	F0 70
KP 1	69	F0 69
KP 2	72	F0 72
KP 3	7A	F0 7A
KP 4	6B	F0 6B
KP 5	73	F0 73
KP 6	74	F0 74
KP 7	6C	F0 6C
KP 8	75	F0 75
KP 9	7D	F0 7D
KP +	79	F0 79
KP Enter	E0 5A	E0 F0 5A
KP .	71	F0 71

C.2 Tabela de Instruções

	Instrução	Bits da Instrução																Bits de Controlo											Flags
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	M1	M2	M3	SelA	SelB	SelC	WRC	RM	M6	Ld	ALUC	
Aritméticas	***Formato I***																												
	ADD	1	0	RC			00000					RA			RB			0	0	01	5..3	2..0	13..11	**1**	0	X	0	**00000**	ONCZ
	SUB	1	0	RC			00001					RA			RB			0	0	01	5..3	2..0	13..11	**1**	0	X	0	**00001**	ONCZ
	ADDC	1	0	RC			00010					RA			RB			0	0	01	5..3	2..0	13..11	**1**	0	X	0	**00010**	ONCZ
	SUBB	1	0	RC			00011					RA			RB			0	0	01	5..3	2..0	13..11	**1**	0	X	0	**00011**	ONCZ
	DEC	1	0	RC			00100					RC			X			0	0	01	5..3	X	13..11	**1**	0	X	0	**00100**	ONCZ
	INC	1	0	RC			00101					RC			X			0	0	01	5..3	X	13..11	**1**	0	X	0	**00101**	ONCZ
	CMP	1	0	R0			00001					RA			RB			0	0	X	5..3	2..0	13..11	X	0	X	0	**00001**	ONCZ
	NEG	1	0	RC			00001					R0			RC			0	0	X	5..3	2..0	13..11	X	0	X	0	**00001**	ONCZ
Lógicas	COM	1	0	RC			01000					RC			X			0	0	01	5..3	X	13..11	**1**	0	X	0	**01000**	NZ
	AND	1	0	RC			01001					RA			RB			0	0	01	5..3	2..0	13..11	**1**	0	X	0	**01001**	NZ
	OR	1	0	RC			01010					RA			RB			0	0	01	5..3	2..0	13..11	**1**	0	X	0	**01010**	NZ
	XOR	1	0	RC			01011					RA			RB			0	0	01	5..3	2..0	13..11	**1**	0	X	0	**01011**	NZ
	TEST	1	0	R0			01001					RA			RB			0	0	X	5..3	2..0	13..11	X	0	X	0	**01001**	NZ
Deslocamento	SHR	1	0	RC			10000					RC			X			0	0	01	5..3	X	13..11	**1**	0	X	0	**10000**	NCZ
	SHL	1	0	RC			10001					RC			X			0	0	01	5..3	X	13..11	**1**	0	X	0	**10001**	NCZ
	SHRA	1	0	RC			10010					RC			X			0	0	01	5..3	X	13..11	**1**	0	X	0	**10010**	ONCZ
	SHLA	1	0	RC			10011					RC			X			0	0	01	5..3	X	13..11	**1**	0	X	0	**10011**	ONCZ
	ROR	1	0	RC			10100					RC			X			0	0	01	5..3	X	13..11	**1**	0	X	0	**10100**	NCZ
	ROL	1	0	RC			10101					RC			X			0	0	01	5..3	X	13..11	**1**	0	X	0	**10101**	NCZ
	RORC	1	0	RC			10110					RC			X			0	0	01	5..3	X	13..11	**1**	0	X	0	**10110**	NCZ
	ROLC	1	0	RC			10111					RC			X			0	0	01	5..3	X	13..11	**1**	0	X	0	**10111**	NCZ
Controlo	***Formato II.a***																												
	NOP	0	0	0	X		0000					X						X	X	X	X	X	X	0	0	X	0	110XX	-
	BR	0	0	0	X		Cond					Constante						X	X	X	X	X	X	0	0	**0**	**1**	110XX	-
	Formato II.b																												
	JMP	0	0	1	0		Cond					X				RB		X	X	X	X	2..0	X	0	0	**1**	**1**	110XX	-
	JAL	0	0	1	1		Cond					X				RB		X	X	**00**	X	2..0	**111**	**1**	0	**1**	**1**	110XX	-
Transferência	***Formato III.a***																												
	MOV	0	1	RC			000			X		X			RB			X	0	01	X	2..0	13..11	**1**	0	X	0	**11001**	-
	LOAD	0	1	RC			010			X		X			RB			X	X	**10**	X	2..0	13..11	**1**	0	X	0	110XX	-
	STOR	0	1	X			011			X		RA			RB			X	X	X	5..3	2..0	X	0	**1**	X	0	110XX	-
	Formato III.b																												
	MVI	1	1	RC			000					Constante						0	**1**	01	X	X	13..11	**1**	0	X	0	**11000**	-
	MVIH	1	1	RC			010					Constante						0	**1**	01	**13..11**	X	13..11	**1**	0	X	0	**11010**	-
	MVIL	1	1	RC			011					Constante						0	**1**	01	**13..11**	X	13..11	**1**	0	X	0	**11011**	-
Interrupção	ENI	0	1	X			100					X						X	X	X	X	X	X	0	0	X	0	110XX	-
	DSI	0	1	X			101					X						X	X	X	X	X	X	0	0	X	0	110XX	-
	RTI	0	1	X			110					X						X	X	X	X	X	X	0	0	X	X	110XX	-
	INT	0	1	X			111					Constante						X	X	X	X	X	X	0	0	X	X	110XX	-
Genéricas	CLC	1	1	X			100					X						X	X	X	X	X	X	0	0	X	0	**11100**	C
	STC	1	1	X			101					X						X	X	X	X	X	X	0	0	X	0	**11101**	C
	CMC	1	1	X			110					X						X	X	X	X	X	X	0	0	X	0	**11110**	C

C.3 Controlador do Mostrador Alfanumérico LCD Hitachi HD44780

Tabela 9.1. Códigos de alguns comandos suportados pelo controlador do mostrador alfanumérico LCD.

Comando	Código									Tempo Máximo de Execução
	RS	7	6	5	4	3	2	1	0	
Limpar mostrador	0	0	0	0	0	0	0	0	1	1,52 ms
Modo de entrada	0	0	0	0	0	0	1	I/D	S	37 µs
Ligar/desligar mostrador	0	0	0	0	0	1	D	C	B	37 µs
Definir deslocamento	0	0	0	0	1	S/C	R/L	X	X	37 µs
Definir funções	0	0	0	1	DL	N	F	X	X	37 µs
Definir posição do cursor	0	1	Linha			Coluna				37 µs
Escrever caráter	1	Caráter								37 µs

Tabela 9.2. Descrição dos campos de configuração do mostrador alfanumérico LCD.

Campo	Valor	
	0	1
I/D	Posição do cursor decrementa automaticamente	Posição do cursor incrementa automaticamente
S	Deslocamento automático do conteúdo desativado	Deslocamento automático do conteúdo ativado
D	Ecrã desligado	Ecrã ligado
C	Cursor não visível	Cursor visível
B	Cursor sem piscar	Cursor a piscar
S/C	Deslocar o cursor	Deslocar o conteúdo
R/L	Deslocar para a esquerda	Deslocar para a direita
DL	Interface de 4 bits	Interface de 8 bits
N	Ecrã de 1 linha	Ecrã de 2 linhas
F	Fonte 5x8	Fonte 5x10

D Ficheiros VHDL

D.1 Descodificador de Instruções

```
-----------------------------------------------------------------------------------------------------
-- Project : P4 (Pequeno Processador Pedagógico com Pipeline)
-- File    : instruction_decoder_base.vhd
-- Author  : Dinis Madeira (dinismadeira@tecnico.ulisboa.pt)
-- Date    : 2018
-----------------------------------------------------------------------------------------------------
-- Description: Decodes an instruction and generates control signals.
-----------------------------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity instruction_decoder_base is
   port (
      -- Inputs
      RI          : in  std_logic_vector(15 downto 0); -- Instruction Register
      -- Outputs
      WRC         : out std_logic;                     -- Write Register C
      SelC        : out std_logic_vector(2 downto 0);  -- Select Register C
      RM          : out std_logic;                     -- Write Memory
      ALUC        : out std_logic_vector(4 downto 0);  -- ALU Control
      M2          : out std_logic;                     -- MUX2
      M3          : out std_logic_vector(1 downto 0);  -- MUX3
      SelA        : out std_logic_vector(2 downto 0);  -- Select Register A
      SelB        : out std_logic_vector(2 downto 0)   -- Select Register B
   );
end instruction_decoder_base;

architecture Architecture_1 of instruction_decoder_base is begin
   process (RI)
      variable format : std_logic_vector(1 downto 0) := RI(15 downto 14);
      variable op     : std_logic_vector(2 downto 0) := RI(10 downto 8);
   begin

      WRC <= '0';
      if format = "10" then WRC <= '1'; end if;                                  -- Arithmetic Operations
      if format = "00" and RI(13 downto 12) = "11" then WRC <= '1'; end if;      -- JAL
      if format = "01" and (op = "000" or op = "010") then WRC <= '1'; end if; -- Transfer Operations
      if format = "11" and op(2) = '0' then WRC <= '1'; end if;                  -- Transfer Operations

      SelC <= RI(13 downto 11);
      if format = "00" and RI(13 downto 12) = "11" then SelC <= "111"; end if; -- JAL

      RM <= '0';
      if format = "01" and op = "011" then RM <= '1'; end if; -- STOR

      ALUC <= "11000";
      if format = "10" then ALUC <= RI(10 downto 6); end if;                        -- Arithmetic Operations
      if format = "01" and op = "000" then ALUC <= "11001"; end if;                 -- MOV
      if format = "11" and op(2) = '0' then ALUC <= "110" & op(1 downto 0); end if; -- MVI, MVIH, MVIL
      if format = "11" and op(2) = '1' then ALUC <= "111" & op(1 downto 0); end if; -- CLC, STC, CMC

      M2 <= '0';
      if format = "11" and op(2) = '0' then M2 <= '1'; end if; -- MVI, MVIH, MVIL

      M3 <= "01";
      if format = "00" and RI(13 downto 12) = "11" then M3 <= "00"; end if; -- JAL
      if format = "01" and op = "010" then M3 <= "10"; end if;              -- LOAD

      SelA <= RI(5 downto 3);
      if format = "11" and (op = "010" or op = "011") then SelA <= RI(13 downto 11); end if; -- MVIH, MVIL

      SelB <= RI(2 downto 0);

   end process;
end Architecture_1;
```

E Programas *Assembly*

E.1 Programa Logo P4

Programa em *assembly* do P4 gerado pela ferramenta de edição de texto do terminal que imprime um logótipo do P4 no terminal.

```
; Endereços de E/S
TERM_WRITE      EQU     FFFEh
TERM_CURSOR     EQU     FFFCh
TERM_COLOR      EQU     FFFBh

                MVI     R1, TERM_WRITE
                MVI     R2, 3600
ClearLoop:      STOR    M[R1], R0
                DEC     R2
                BR.NZ   ClearLoop

TerminalStr     STR     0,2,7e0h,'…',0,0 ; Conteúdo do Terminal
                MVI     R1, TERM_WRITE
                MVI     R2, TERM_CURSOR
                MVI     R3, TERM_COLOR
                MVI     R4, TerminalStr

TerminalLoop:
                LOAD    R5, M[R4]
                INC     R4
                CMP     R5, R0
                BR.Z    .Control
                STOR    M[R1], R5
                BR      TerminalLoop

.Control:
                LOAD    R5, M[R4]
                INC     R4
                DEC     R5
                BR.Z    .Position
                DEC     R5
                BR.Z    .Color
                BR      .End

.Position:
                LOAD    R5, M[R4]
                INC     R4
                STOR    M[R2], R5
                BR      TerminalLoop

.Color:
                LOAD    R5, M[R4]
                INC     R4
                STOR    M[R3], R5
                BR      TerminalLoop

.End:
                BR      0
```

E.2 Programa *Keyboard Interrupts*

Programa em *assembly* do P4 para demonstração das interrupções do teclado e de vários periféricos. O programa começa por limpar a janela do terminal e ativar as interrupções do teclado. Quando uma tecla é premida é chamada uma rotina que imprime o valor da tecla premida no terminal, nos *LEDs*, no mostrador alfanumérico LCD e nos mostradores hexadecimais de 7 segmentos.

```
; Endereços de E/S
TERM_READ       EQU     FFFFh
TERM_WRITE      EQU     FFFEh
INT_MASK        EQU     FFFAh
LEDS            EQU     FFF8h
LCD_WRITE       EQU     FFF5h
DISP7SEG_3      EQU     FFF3h
DISP7SEG_2      EQU     FFF2h
DISP7SEG_1      EQU     FFF1h
DISP7SEG_0      EQU     FFF0h

                ; Limpar Terminal
                MVI     R1, TERM_WRITE
                MVI     R2, 3600
ClearLoop:      STOR    M[R1], R0
                DEC     R2
                BR.NZ   ClearLoop

                ; Ativar Interrupções do Teclado
                MVI     R1, INT_MASK
                MVI     R2, 80h
                STOR    M[R1], R2
                ENI

Loop:           CMP     R7, R0
                BR.Z    Loop

                ; Mostrar no Terminal
                MVI     R1, TERM_WRITE
                STOR    M[R1], R7
                ; Mostrar nos LEDs
                MVI     R1, LEDS
                STOR    M[R1], R7
                ; Mostrar no Ecrã LCD
                MVI     R1, LCD_WRITE
                STOR    M[R1], R7
                ; Mostrar nos Mostradores
                MVI     R1, DISP7SEG_0
                STOR    M[R1], R7
                SHR     R7
                SHR     R7
                SHR     R7
                SHR     R7
                MVI     R1, DISP7SEG_1
                STOR    M[R1], R7
                SHR     R7
                SHR     R7
                SHR     R7
                SHR     R7
                MVI     R1, DISP7SEG_2
                STOR    M[R1], R7
                SHR     R7
                SHR     R7
                SHR     R7
                SHR     R7
                MVI     R1, DISP7SEG_3
                STOR    M[R1], R7
                MVI     R7, 0
                BR      Loop

                ; Interrupção do Teclado
                ORIG    7F70h
                ; Ler Tecla Premida Para R7
                MVI     R1, TERM_READ
                LOAD    R7, M[R1]
                RTI
```