FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

FEUP

Simulador e Compilador de Micro-Código para Processador Vectorial Dedicado

Pedro Manuel Marques Martins Carneiro

Dissertação realizada no âmbito do
Mestrado Integrado em Engenharia Electrotécnica e de Computadores
Major Telecomunicações

Orientador: José Carlos dos Santos Alves, Prof. Dr.

Setembro de 2011

Resumo

Os processadores escalares, que operam apenas um dado ou um conjunto de dados em cada execução de uma instrução, são por vezes limitados. Desta forma, é relevante e necessário o desenvolvimento de soluções alternativas que explorem paralelismo ao executar as operações, para que o desempenho do processamento possam aumentar.

Seguindo este ponto de vista, os processadores vetoriais são uma opção credível, com uma arquitetura do tipo SIMD ou MIMD, em que operam simultaneamente vários operandos com apenas uma instrução. Por outro lado, a flexibilidade que apresentam ao nível da sua implementação reforça a sua validade.

De modo a ser possível tirar o máximo partido desta caraterística plataformas reconfiguráveis, como por exemplo FPGA, são o parceiro ideal para implementações deste tipo. Nomeadamente nos últimos anos, estes dispositivos têm sofrido uma evolução notável, passando a incorporar milhares e mesmo milhões de blocos lógicos. A flexibilidade das arquiteturas vetoriais aliada a todas as potencialidades de um FPGA torna possível a realização de processadores dedicados, feitos à medida de uma dada aplicação.

Os trabalhos no contexto desta Dissertação compreendem duas componentes, conjugadas para produzir uma cadeia completa que proceda à geração e simulação de arquiteturas vetoriais dedicadas.

Em primeiro lugar, foi construído um assemblador de linguagem simbólica que realiza a tradução de ficheiros de descrição da arquitetura a concretizar. O objetivo alvo desse micro-assemblador é a geração de código binário, o tipo de código que o processador executa.

Por outro lado, foi desenvolvida outra ferramenta de *software* que gera os módulos *Verilog* da arquitetura dedicada, devidamente ligados e configurados de acordo com as especificações exigidas. O programa gerador cria ainda uma bancada de teste *Verilog* que tem como função simular o processador resultante do processo de geração.

Assim, a arquitetura gerada é posteriormente, através de cálculos e estímulos contidos na bancada de teste referida, testada e avaliada.

Abstract

Scalar processors, operating just only one data or a set of data in each execution of an instruction, are sometimes limited. In this way, it's relevant and necessary the development of alternative solutions that explore parallelism in executing operations, so processing performance can increase.

Following this point of view, vector processors are a credible option, with an architecture of SIMD or MIMD type, operating at the same time many operands with only one instruction. On the other hand, the flexibility of implementation reinforces their validity.

In order to take full advantage of this feature reconfigurable platforms, such as FPGA, are the ideal partner for this kind of implementations. Namely over recent years, these devices have undergone a remarkable evolution, incorporating thousands and even millions of logic cells. The flexibility of vector architectures combined with all the potentialities of an FPGA makes it possible to perform dedicated processors, tailor-made for a given application.

The work in the context of this Dissertation include two components, together to produce a complete chain to generate and simulate dedicated vector architectures.

Firstly, it was built a symbolic language translator which performs the translation of the description files of the architecture to materialize. The target objective of the micro-assembler is the generation of binary code, the type of code that the processor executes.

In the other side, it was developed other software tool which creates *Verilog* modules properly connected and configured, according to the required specifications. The program generator also creates a *Verilog* test bench which is designed to simulate the processor resulting from the generation process.

Finally, the architecture generated is then, through calculations and stimulus contained in the mentioned test bench, tested and evaluated.

Agradecimentos

Ao longo de todo o meu percurso académico, em especial nestes anos de faculdade, existiram muitas pessoas que me influenciaram positivamente e, por esse motivo, gostaria de expressar aqui a minha gratidão e pequena homenagem para com elas.

Em primeiro lugar quero agradecer ao Prof. Dr. José Carlos Alves, meu orientador, por todo o tempo dispensado ao longo desta Dissertação. Durante esse período deu-me várias orientações e sugestões que foram um contributo essencial para o bom desenvolvimento destes trabalhos.

Deixo também o meu agradecimento à Universidade do Porto, Faculdade de Engenharia em particular, por todo o caminho que me proporcionaram, com os meios humanos e materiais que puseram ao meu dispor, incluindo a oportunidade de fazer Erasmus.

Aos meus companheiros de laboratório, os meus agradecimentos por todo o convívio e espírito de entreajuda que proporcionaram durante estes trabalhos.

A todos os meus verdadeiros amigos, em Portugal e no resto do mundo, o meu muito obrigado por todos os bons e maus momentos que partilhei convosco.

Em especial àqueles que vestiram um traje igual ao meu, porque me ensinaram e ajudaram a compreender um conjunto de regras para o presente e futuro; de forma peculiar um abraço sentido para os amigos que ingressaram em 2006 comigo, Rafael Lopes, Miguel Garcia, André Matos, Vítor Sobrado, Filipe Pereira, Paulo Félix, Joel Ramos, João Teixeira, Victor Veloso, Eurico Fonseca, Inês Queirós, Ramiro Cortez, Ricardo Brito, Joana Fonseca, Telmo Oliveira, Daniel Silva, João Marques e tantos outros que não cabem no papel mas estão cá dentro.

Um enorme abraço para os elementos do TBPK por todos os momentos únicos, em particular para o meu grande amigo Presidente Gabriel Vieira.

Um grande bem-haja a todos os companheiros de aventura de Erasmus na República Checa. Todos vocês contribuíram positivamente para uma das melhores e inesquecíveis experiências da minha vida.

Saudações para o grupo de BTT Os Tartarugas, pela amizade e por terem proporcionado momentos de lazer e convívio durante esta caminhada.

Agradeço ainda aos amigos que já o eram antes e hoje ainda continuam a me apoiar da mesma forma. Um abraço especial para o amigo de sempre Diogo e um beijo grande para a Isabel pelo seu incansável apoio.

Todos vocês ajudaram a tornar-me numa pessoa diferente ao longo destes anos.

Um enorme obrigado à família Santos, por me terem ensinado que não temos uma só família e por me terem feito parte da vossa. Sempre que precisei deram um contributo que não dá para

medir. Um beijo grande em específico para a Marta, por ter sido em todos os sentidos a irmã que nunca tive mas afinal tenho.

Por último, mas com a maior importância, agradeço do fundo do coração à minha família, principalmente aos meus pais. Por terem sido o suporte e inspiração ao longo deste percurso e por sempre me terem incentivado e apoiado em todas as minhas conquistas, que assim se tornaram as nossas conquistas.

Pedro Manuel Marques Martins Carneiro

"(...) And a new day will dawn for those who stand long (...)"

Led Zeppelin - Stairway to Heaven

Conteúdo

Lista de Figuras

Lista de Tabelas

Abreviaturas e Símbolos

ASIC	Application Specific Integrated Circuit
Bit	Binary Digit
CDC	Control Data Corporation
CISC	Complex Instruction Set Computing
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
DDR	Double Data Rate
DSP	Digital Signal Processor
FEUP	Faculdade de Engenharia da Universidade do Porto
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
GCC	GNU Compiler Collection
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IC	Integrated Circuit
IIR	Infinite Impulse Response
MB	Mega Byte
MISD	Multiple Instruction Single Data
MIMD	Multiple Instruction Multiple Data
MIPS	Microprocessor without Interlocked Pipeline Stages
MOSFET	Metal Oxide Semiconductor Field Effect Transistor
PAM	Programmable Active Memories
RAM	Random Access Memory
RISC	Reduced Instruction Set Computing
RO	Read-Only
ROM	Read-Only Memory
RTL	Register Transfer Level
RW	Read & Write
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
STAR	Strings and Arrays
VAS	Vetorial, Assembly e Simbólica
VESPA	Vector Extended Soft Processor Architecture
VHDL	Very high speed integrated circuit Hardware Description Language
VLIW	Very Long Instruction Word
VLR	Vector Lenght Register
WO	Write-Only

Capítulo 1

Introdução

Atualmente, e com base no progresso das décadas anteriores, a área da microeletrónica demonstra um desenvolvimento crescente. Em 1965, Gordon Moore constatou com base em dados passados que o número de transístores numa pastilha duplica, sensivelmente, a cada 18 meses, mantendo-se todavia o seu custo [1]. A tendência nos anos subsequentes até ao tempo corrente vem confirmar essa suposição. Na verdade, este crescimento levou à criação de dispositivos cada vez mais complexos e poderosos. Por exemplo, hoje em dia é possível adquirir um computador pessoal por menos de 300 € com uma capacidade de processamento e memória bastante superiores do que um computador de há 15 anos atrás.

A indústria dos semicondutores deu um contributo fulcral para todo este desenrolar positivo. Nomeadamente a evolução dos MOSFET, especificamente da tecnologia CMOS, conduziu à implementação de circuitos integrados cada vez mais pequenos, mas constituídos por milhões de transístores. Deste modo, rapidamente se tornaram na célula base dos equipamentos eletrónicos atuais.

Uma das plataformas que incorpora este tipo de componente são os FPGA. Grande parte dos circuitos integrados digitais, como é o caso dos ASIC, vêm com a sua função definida de fabrico, não podendo ser posteriormente mudada. O *hardware* constituinte de um FPGA pode ser reconfigurado, isto é, permite que a lógica seja moldada de acordo com os objetivos pretendidos por quem faz uso dela.

Neste contexto, os FPGA são vistos como um ótimo suporte para a implementação de processamento vetorial. Uma das principais vantagens desta relação é a flexibilidade alcançada para um processador deste tipo, dado que se torna possível a sua configuração de acordo com a aplicação em que será usado. Outro dos grandes benefícios é a portabilidade para qualquer arquitetura FPGA [2].

O processamento vetorial, como o próprio nome indica, proporciona operações sobre os elementos de um vetor. Para tal tem a capacidade de carregar uma única instrução e executá-la sobre vários elementos de uma só vez.

Os trabalhos desta Dissertação enquadram-se no desenvolvimento de ferramentas de *software* para uma arquitetura vetorial e dedicada, já parcialmente desenvolvida. O objetivo destas ferramentas é a geração automática da arquitetura, do código binário que executa e sua simulação.

1.1 Motivação

A motivação dos trabalhos desta Dissertação surge com a arquitetura vetorial e dedicada previamente implementada. Uma vez que esta se encontra desenvolvida para ser trabalhada com base em FPGA, ocorre a necessidade de se poder configurar e adaptar o processador à aplicação para a qual é feito à medida. Assim, esta necessidade conduz a que seja imperativa a implementação de ferramentas de *software* que tenham como função a geração dos micro-programas a executar pelo processador. Devem ainda ser capazes de gerar e parametrizar toda a arquitetura, conforme os requisitos exigidos pela aplicação e utilizador. Por fim, é igualmente imprescindível a criação de um mecanismo que permita simular todo o processo de geração descrito no parágrafo anterior. Esta simulação visa confirmar o bom funcionamento do processador e ainda tirar algumas conclusões acerca do desempenho deste tipo de unidades.

Estes trabalhos encontram-se enquadrados no Projeto VECTOR - *Matlab Compilation and Hardware Synthesis of Custom-Vector Processing for Image and Signal Processing Algorithms*.

1.2 Objetivos

No seguimento dos objetivos gerais e da motivação apresentada na secção anterior 1.1 foram projetadas as seguintes metas mais detalhadas:

1. Estudo dos módulos *Verilog* da implementação do processador já existente;
2. Definição da constituição dos ficheiros que descrevem a arquitetura a implementar;
3. Criação de uma ferramenta de *software* que realize a tradução de pseudo-código de alto nível para micro-código binário;
4. Tornar a ferramenta de assemblagem mais robusta e intuitiva, para que o seu uso seja mais fácil e parecido com um compilador/assemblador corrente;

5. Geração dos módulos *Verilog* de acordo com as especificações pretendidas assim como da bancada de teste que permite a simulação da arquitetura;
6. Garantia que as restrições impostas são satisfeitas ou o utilizador é informado do motivo do erro;
7. Simulação de toda a arquitetura e do seu desempenho, com recurso a alguns exemplos;
8. Escrita do documento final da Dissertação.

1.3 Estrutura do Documento

Para além deste capítulo introdutório, o capítulo 1, este documento conta com mais 5 capítulos, devidamente divididos e identificados.

No capítulo 2, é apresentado o estado da arte realizado previamente ao trabalho propriamente dito. Trata-se de uma revisão bibliográfica acerca da base teórica e do progresso conseguido nas áreas que envolvem esta Dissertação. Na parte final deste capítulo são ainda mencionadas algumas ferramentas, de entre as quais foram selecionadas algumas para o decorrer deste trabalho.

O capítulo 3 contém a exposição da arquitetura dedicada já implementada. São referidos todos os aspetos daquela que será a principal base do restante trabalho.

O conteúdo do capítulo 4 gira em torno de todo o trabalho desenvolvido ao longo da Dissertação. Desta forma, engloba a descrição pormenorizada das tarefas realizadas, a justificação das decisões tomadas e a metodologia usada em cada passo.

No capítulo 5 são abordados todos os resultados obtidos, com indicação dos testes efetuados para a sua validação.

O capítulo 6 inclui as conclusões retiradas com este trabalho, com uma componente crítica para os resultados conseguidos. Tem também uma secção onde são relatadas perspetivas de trabalho futuro no contexto do tema e do trabalho realizado.

No início de cada capítulo anterior, com exceção dos capítulos 1 e 6, é dada uma informação detalhada sobre o seu teor. No final de cada um é igualmente feito um pequeno resumo dos pontos mais importantes referidos ao longo desse capítulo.

No Anexo A são apresentados os resultados verificados com a utilização dos argumentos facultativos das ferramentas de *software*.

O Anexo B ilustra algumas situações de erro na execução das mesmas ferramentas.

Por seu turno o Anexo C contém um exemplo da constituição de um processador gerado com base nos ficheiros de configuração da arquitetura vetorial.

Finalmente, o documento termina com a inclusão das Referências, isto é, todas as fontes consultadas para a realização da Dissertação.

Capítulo 2

Estado da Arte

2.1 Introdução

Neste capítulo é feita uma revisão bibliográfica sobre as áreas onde incidem os trabalhos desta Dissertação.

Em primeiro lugar, na secção 2.2 é detalhado o conceito de arquitetura dedicada, com maior incidência para as arquiteturas vetoriais e para as arquiteturas baseadas em FPGA. São apresentados os princípios e base teórica de ambos e são expostos alguns exemplos para uma melhor perceção.

De seguida, na secção 2.3 é mostrado um estudo sobre memórias e o seu funcionamento. Dentro das memórias ROM e RAM, bastante usadas em sistemas digitais e em arquiteturas deste género em particular, as memórias *cache* merecem especial ênfase porque exibem algumas diferenças para os restantes dispositivos de armazenamento.

Finalmente, na secção 2.4 são mencionadas algumas ferramentas que são os meios a usar para alcançar os objetivos propostos para esta Dissertação.

2.2 Arquiteturas

No mundo dos computadores, o termo *arquitetura* toma um significado no sentido mais lato da palavra. Com efeito, segundo [3], dentro desse domínio *arquitetura* revela-se como o conjunto de 3 componentes:

- **Conjunto de instruções** — inclui o conjunto de instruções, modos de endereçamento de memória e registos do processador.

- **Organização** — elemento da arquitetura de mais alto nível. Conta com a descrição das memórias, CPU e respetivas interligações.

- ***Hardware*** — atributos mais específicos do computador em questão, como o projeto detalhado da lógica e o encapsulamento de toda a tecnologia no computador.

Na sequência do estudo realizado e que é apresentado ao longo deste capítulo 2, a definição de arquitetura assenta essencialmente nos dois primeiros pontos mencionados anteriormente. Essa definição atenta especialmente à Organização e ao conjunto de instruções referido no primeiro ponto.

Deste modo, é notória a grande diversidade de arquiteturas existente. Contudo, no contexto desta análise será dado mais realce às arquiteturas vetoriais e arquiteturas baseadas em FPGA, descritas nos sub capítulos 2.2.2 e 2.2.3, respetivamente.

As arquiteturas podem também ser divididas de acordo com as funções pretendidas: dedicadas e de propósito geral. Os processadores de propósito geral têm uma esfera de ação superior, podendo ser usados em qualquer tipo de implementação. Por seu turno, processadores dedicados são concebidos tendo em vista um papel específico numa dada aplicação ou tarefa.

Desta forma, arquiteturas dedicadas conduzem a um melhor desempenho, sobretudo no que concerne a tempos de execução. Porém, carregam a grande desvantagem do seu uso ser limitado e não poder ser generalizado. O desempenho dos processadores, nomeadamente das arquiteturas de propósito geral, aumentou cerca de 80% cada ano, de 1985 a 2000. Este progresso deve-se à exploração de paralelismo bem como a melhorias na largura de banda e na latência de acesso das memórias [4].

Na próxima secção são abordadas várias formas de tirar proveito do processamento em paralelo, para que possam ser atingidos melhores níveis de desempenho.

2.2.1 Modos de Paralelismo

Como foi referido atrás, a grande meta a alcançar com o uso de paralelismo é o aumento de desempenho do processamento. Existem diversas maneiras de implementar esta técnica. Todas elas têm em comum a característica que está inerente ao próprio sentido da palavra paralelismo: a realização de tarefas em simultâneo com o objetivo de diminuir os tempos das suas execuções.

Em 1967, Gene Amdahl modelou o progresso positivo conseguido com o uso de paralelismo com a relação hoje conhecida como lei de Amdahl. Ao longo do tempo foi sofrendo algumas reestruturações, acabando por ser estabelecida em 2000 pelo próprio Gene Amdahl através da

Equação 2.1 [5]:

$$speedup = \frac{1}{(1-f)+\frac{f}{N}}, \tag{2.1}$$

onde f é a percentagem do processamento realizado em N processadores paralelos. Na expressão anterior $1-f$ traduz o fragmento temporal de processamento em série, sem paralelismo.

A partir da Equação 2.1 é possível confirmar o aumento da velocidade no processamento quando existe recurso a vários processadores em paralelo. Por outro lado quando N tende para infinito, o *speedup* tende para $\frac{1}{(1-f)}$. Nesta situação, e apesar de haver um acréscimo em f e consequente aumento do *speedup*, este não é significativo face ao número de processadores, teoricamente infinito. É por este motivo que o paralelismo só deve ser explorado em situações em que haja uma percentagem de processamento paralelo muito grande ou quando o número de processadores usado é relativamente pequeno [6].

Em suma, o paralelismo tem como grande bandeira uma melhoria significativa no desempenho no processamento. O facto de serem usadas várias unidades de processamento permite que haja uma melhor partilha entre todas. Assim, a complexidade de cada uma dessas unidades é inferior comparada à complexidade "geral" sem paralelismo.

Não obstante, também apresenta desvantagens em relação ao processamento convencional em série. O facto de haver processamento em simultâneo implica que haja um sistema de controlo, a fim de haver sincronização. É este fator que garante a inexistência de erros quando ocorrem concorrência e dependência de dados[1].

Relativamente às formas de implementar processamento paralelo, existem 4 níveis diferentes segundo os quais o paralelismo pode ser avaliado [3, 7]:

- **Paralelismo ao nível do *bit*** — técnica que tem a ver com o número de dígitos binários que um processador consegue suportar por ciclo de relógio numa instrução. Um bom exemplo é a subtração de 2 números de 64 bits cada; um processador de 64 bits consegue realizar a operação com uma só instrução, ao passo que um processador de 32 bits necessita de 2 instruções para operar a parte mais significativa e a parte menos significativa separadamente, incluindo a propagação do *borrow*. Atualmente, as duas arquiteturas mencionadas atrás são as mais utilizadas.

- **Paralelismo ao nível da tarefa** — alternativa que explora o paralelismo através da divisão de uma tarefa maior em tarefas menores e usar diferentes processadores para cada tarefa mais simples. É o que acontece hoje em dia nos processadores com vários núcleos de processamento (processadores *multicore*).

[1] Estado que em inglês se designa por *race condition*.

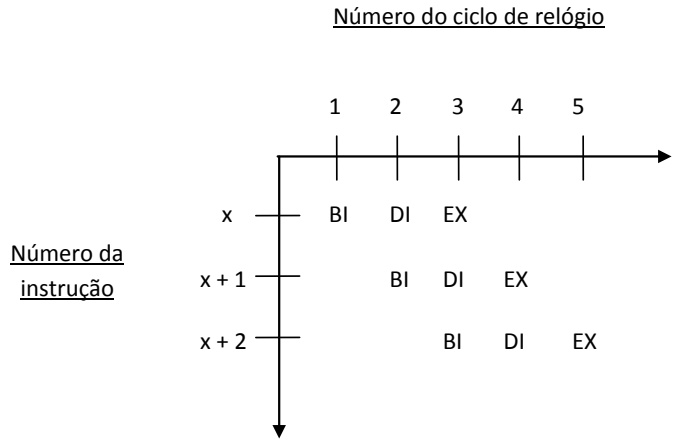

Figura 2.1: Excerto de *Pipeline* com 5 andares. No excerto apresentado são ilustrados apenas 3 dos 5 andares de *Pipeline* de cada instrução. Os nomes dos andares de *Pipeline* são os nomes dos ciclos na versão sem esta estratégia: BI = Busca da Instrução (*Instruction Fetch*), DI = Descodificação da Instrução (*Instruction Decode*) e EX = Execução (*Execution*). Adaptado de [3].

- **Paralelismo ao nível de dados** — opção que também é aproveitada nos processadores com diversos núcleos. Baseia-se em dividir uma operação sobre um determinado dado pelos váriados núcleos, em que cada núcelo opera sobre uma parte do dado.

- **Paralelismo ao nível da instrução** — forma de paralelismo que tira partido da execução em simultâneo de instruções cujos operandos não dependem dos resultados das operações entre si. Processadores deste tipo usam técnicas para evitar a dependência de dados e permitir o uso de paralelismo, designadamente, renomeação de registos, execução de instruções fora de ordem e execução especulativa.

  A técnica com maior utilização hoje em dia é a introdução de andares de *pipeline*. A ilustração deste mecanismo encontra-se na Figura 2.1. A cada ciclo de relógio uma nova instrução é carregada para o *pipeline* e a parte da instrução apresentada demora 3 ciclos de relógio a ser processada. Como são executadas várias instruções em paralelo, consegue-se um aumento de velocidade para o triplo em relação a um processador sem *pipeline*, na ausência de *hazards*.

2.2.2 Arquiteturas Vetoriais

Em 1972, Michael Flynn definiu um meio de classificação de arquiteturas. Esse método, conhecido como Taxonomia de Flynn, encontra-se representado na Tabela 2.1. As 4 combinações

Tabela 2.1: Taxonomia de Flynn.

	Instruções	
Conjunto de Dados	SISD	MISD
	SIMD	MIMD

ilustradas baseiam-se nas instruções executadas e no conjunto de dados operados. Em ambos podem acontecer 2 situações: simples ou múltiplos [8, 9].

No caso das arquiteturas SISD é executada somente uma instrução sobre um conjunto de dados ou um operando apenas. Trata-se, portanto, do processamento tradicional escalar em série. No entanto, é possível realizar implementações paralelas deste tipo de arquitetura, nomeadamente com o uso de *pipelines*.

As arquiteturas do tipo MISD, por seu turno, caraterizam-se por haver várias operações sobre um único grupo de dados. Apesar deste género de arquitetura tirar proveito do paralelismo, as suas complexidade e redundância que introduz fazem com que não seja uma arquitetura de uso corrente. Uma boa ilustração deste tipo de arquitetura é o cálculo matricial; por exemplo, uma arquitetura MISD permite o produto de 2 matrizes, sendo estas os dados de entrada e não cada elemento de cada linha e coluna de uma matriz separadamente.

No que concerne à exploração de paralelismo as arquiteturas SIMD e MIMD são, de facto, as mais proveitosas. Este último tipo distingue-se por arquiteturas onde várias instruções operam simultaneamente sobre vários conjuntos de dados diferentes. É a situação que acontece nos processadores com vários núcleos de processamento. Este tipo de arquitetura, geralmente, aproveita também o paralelismo ao nível da instrução, mais concretamente por intermédio da técnica de uso de instruções longas[2].

Falta apenas referir uma arquitetura de todas aquelas incluídas na taxonomia proposta por Flynn, a arquitetura do tipo SIMD. Esta diferencia-se das demais por descrever processamentos em que uma instrução opera ao mesmo tempo vários dados. Por isso, as arquiteturas do tipo SIMD são uma ferramenta poderosa quando é necessário operar sobre um conjunto de informação ao mesmo tempo, como amostras ou píxeis. Por este motivo são muito eficientes em aplicações de cálculo de filtros, como por exemplo no cálculo de respostas FIR e IIR, e em aplicações gráficas, em que os dados multimédia necessitam de, por vezes, sofrer algum tipo de tratamento antes de serem usados para algum fim mais prático. Um exemplo típico é a subtração de elementos de vários vetores, ilustrada na Figura 2.2. Com uma arquitetura do tipo SISD seria necessário carregar a instrução subtração e realiza-la VLR vezes: para o primeiro elemento de cada vetor, depois para o segundo elemento de cada vetor e assim sucessivamente. Recorrendo a uma arquitetura do tipo SIMD, esta sequência de subtrações pode ser feita através da execução de uma só instrução: neste caso, a instrução VSUB que subtrai os elementos do vetor *v2* aos elementos do vetor *v1* de uma só vez. O resultado é colocado no vetor *v3*.

[2] VLIW

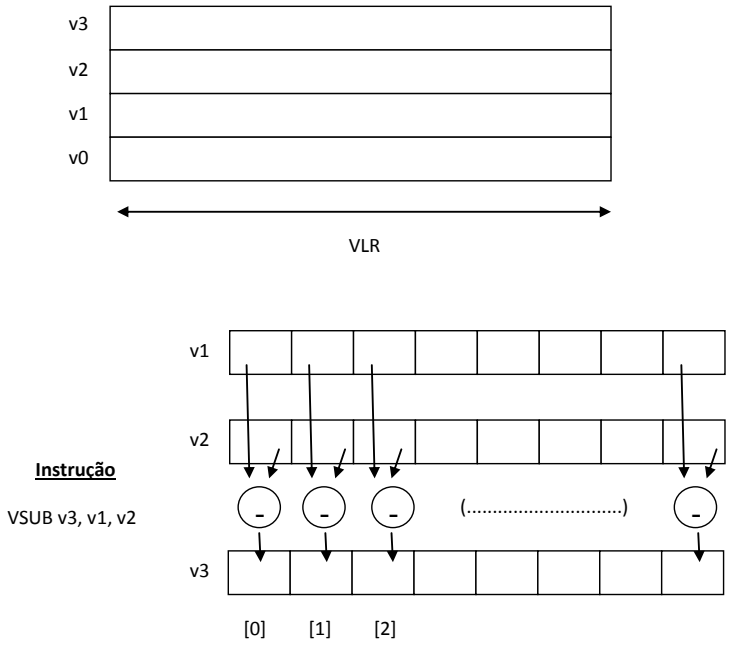

Figura 2.2: Exemplo de instrução e de um registo vetoriais. Na parte de cima da imagem está representado um registo vetorial, com capacidade para 4 vetores com comprimento VLR cada. Na parte de baixo encontra-se ilustrado o funcionamento da instrução vetorial de subtração entre os elementos de 2 vetores. Adaptado de [11].

Devido a estas caraterísticas as arquiteturas SIMD são ainda um ótimo meio para a implementação de ciclos, tais como os ciclos *for* ou *while* em linguagem C, por exemplo [10].

Todavia, este tipo de arquitetura também apresenta desvantagens. Em primeiro lugar, faz uso de muitos registos e/ou memórias para armazenar operandos e resultados. Este é um fator que contribui negativamente para que haja um aumento do consumo e da área ocupada. Por outro lado, deste modo, a latência no acesso às memórias também tende a ser um problema. Adicionalmente as instruções que são executadas podem exigir o acesso a dados guardados com uma ordem diferente da ordem em que são necessários ao longo do processamento. Como tal, são usados mecanismos de reordenação de dados e renomeação de memórias, o que contribui para um decréscimo da eficiência do processamento e a ocorrência de possíveis erros na leitura dos dados. [10, 12].

Os processadores de arquitetura vetorial são do tipo SIMD ou tipo MIMD e herdam todas as características descritas anteriormente. Logo aí diferem dos processadores escalares, tipicamente processadores do tipo SISD. O processamento vetorial, de uma forma geral, recorre ao paralelismo de dados e ao nível da instrução, não fazendo uso do paralelismo ao nível da tarefa. Com efeito, a grande diferença entre uma arquitetura vetorial e uma arquitetura escalar é ao nível do conjunto de instruções. Enquanto que uma implementação escalar opera sobre dados individuais, as operações

de uma implementação vetorial trabalham sobre conjuntos de dados individuais ordenados, ou por outras palavras, vetores [13].

Contudo, o facto de as instruções serem específicas do tipo de arquitetura também traz alguns inconvenientes. Em primeiro lugar, o facto de serem diferentes leva à necessidade de serem compiladas por um processador adequado a esse tipo de instrução. Este processo nem sempre é simples e exige trabalho humano. Do mesmo modo, por vezes é complicado transformar um algoritmo em instruções passíveis de serem executadas com um processador vetorial.

Apesar das suas diferenças, é possível aliar uma arquitetura vetorial com uma arquitetura escalar. Por vezes é proveitoso usar ambas as implementações, principalmente quando não é possível vetorizar uma determinada sequência de operações. Nesta situação, o processador vetorial executa as instruções vetoriais normais, sendo auxiliado na sua lacuna pelo processador escalar. Outro exemplo em que esta relação produz benefícios é o acoplamento de *soft-cores* com o objetivo de acelerar o processamento geral. Um *soft-core* é um processador que pode ser implementado recorrendo a *software* de alto nível, como linguagem C, e posteriormente compilado com vista a ser integrado num FPGA ou ASIC, por exemplo. Esta última alternativa é vantajosa porque permite reduzir o tempo de processamento em virtude das capacidades do processamento vetorial; proporciona ainda uma redução no custo total e tempo de implementação do processador. Por último não exige larga experiência e habilidade no desenvolvimento de *hardware* [12, 2, 19].

As implementações de processadores vetoriais também variam no que concerne à forma de ler e armazenar operandos e resultados. Nessa linha de pensamento, são consideradas 2 tipos de arquitetura vetorial: arquitetura memória-memória e arquitetura de registo. Os supercomputadores CDC STAR-100 e Cray-1, são, respetivamente, exemplos de ambos os tipos de arquitetura. Os processadores do tipo memória-memória leêm e escrevem todos os dados envolvidos nas operações bem como os dados obtidos em memórias. Esta solução é melhor em termos de capacidade, mas perde em termos de tempo. Todos estes acessos às memórias provocam um aumento da latência.

Por sua vez, a arquitetura de registo usa bancos de registos mais pequenos mas também mais próximos das unidades de cálculo. Deste modo conseguem combater o problema da latência e largura de banda da memória. De um modo geral, a opção ideal passa por fazer uma implementação híbrida de ambas as arquiteturas. Uma implementação em que os resultados finais sejam armazenados em memórias, mas os resultados intermédios, necessários a cálculos futuros, sejam guardados em registos.

De referir ainda que num processador actual o acesso à memória costuma dar-se de 2 formas diferentes: acesso por passo e acesso indexado. No primeiro, o endereço de memória acedido é condicional e depende do endereço actual. A posição acedida depende do valor de um passo x, em que a posição do acesso é o endereço atual somado ou subtraído de x posições. O acesso indexado também é condicional, mas dependente do endereço base, posição de memória onde começa o armazenamento de dados. A posição acedida é obtida com a soma de uma constante a

esse endereço base [11, 12].

Em 1963 com o desenvolvimento do CDC 6600, o primeiro processador a usar instruções do tipo RISC, surgia um dos precursores dos processadores vetoriais. Apesar de realizar processamento escalar, já apresentava algumas características dos supercomputadores vetoriais. Neste leque incluem-se o comprimento das instruções e a capacidade de memória, anormalmente grandes para a altura [14].

Os primeiros verdadeiros supercomputadores vetoriais apareceram no início da década de 70. Um exemplo deste tipo de computadores é o STAR-100, desenvolvido pela CDC, contemporâneo dos processadores CDC 6600 e CDC 7600. Já usava instruções vetoriais capazes de operar vários dados numa só execução e explorava algum paralelismo através de *pipelines* e a ajuda de uma unidade escalar. Essas instruções eram inovadoras na altura, pois eram de 64 bits e não 60, como nas arquiteturas anteriores. Eram portanto instruções semelhantes às instruções CISC atuais [14].

Os processadores vetoriais foram progredindo e a sua época de renome chegou por volta de 1976, principalmente com as inovações trazidas com a implementação do processador Cray-1. A sua maior contribuição era colmatar a falha do STAR-100, apresentando um processamento escalar eficiente e não apenas um super processamento vetorial. Ao contrário dos seus antecessores, continha registos que permitiam reutilizar dados de uma forma mais rápida, aumentando a largura de banda da memória [14].

Com o desenvolvimento da tecnologia CMOS os processadores vetoriais foram perdendo terreno. Essa tecnologia permitiu o fabrico de processadores com menor área e consumo mas, acima de tudo, a um custo menor. Para tal, contribuiu bastante a quantidade de produção bastante alta, oposto da produção de algumas dezenas de processadores vetoriais. No entanto, e apesar dos poucos avanços a nível da sua arquitetura, os processadores vetoriais continuam a ser objeto de estudo e de uso nos dias correntes. Um exemplo concreto dessa utilização são os DSP, que incorporam caraterísticas vetoriais como o uso de instruções longas e uma arquitetura do tipo SIMD. Um DSP tira partido destas credenciais para realizar processamento de sinal e implementar algoritmos como filtros FIR ou FFT. Finalmente, os processadores vetoriais têm sido explorados tendo como alvo a sua implementação em plataformas reconfiguráveis, como os FPGA [14, 15]. Estas implementações são abordadas na secção 2.2.3.

No futuro, segundo [14], os possíveis panoramas de evolução são essencialmente 2 e encontram-se ilustrados na Figura 2.3. Por um lado, os processadores vetoriais podem-se manter na perspetiva atual, com exercício em aplicações dedicadas científicas e gráficas. Todavia, o dilema do custo e das unidades vendidas manter-se-á, inviabilizando a sua competitividade. Neste ramo, a provável solução será os processadores passarem a incorporar unidades super escalares e vetoriais, em que ambas tiram partido simultaneamente de paralelismo ao nível da tarefa.

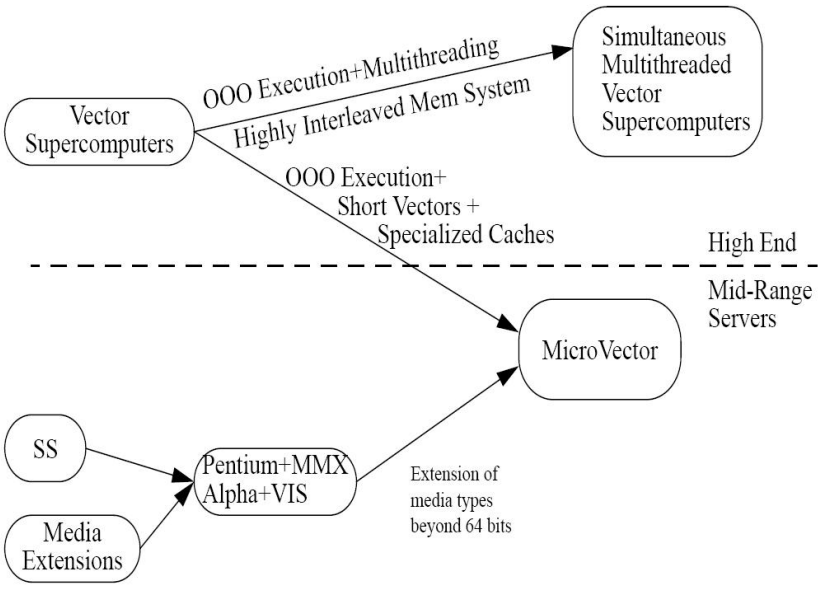

Figura 2.3: Perspetiva de evolução das arquiteturas vetoriais. Encontram-se representadas 2 das possíveis evoluções dos processadores vetoriais [14].

2.2.3 Arquiteturas Baseadas em FPGA

Um FPGA é um dispositivo reconfigurável, bastante usado nos dias de hoje na implementação de sistemas digitais complexos. Tem a sua origem sensivelmente a meio dos anos 80, criado pela empresa Xilinx. Sendo um dispositivo semicondutor, sofreu a mesma evolução que toda essa indústria e a tecnologia CMOS. Inicialmente tinham poucos milhares de blocos lógicos e, hoje em dia, apresentam bastantes milhões desses blocos. Por todas estas razões são dispositivos flexíveis, eficientes e com custos de desenvolvimento reduzidos. Derivado da sua produção em larga escala, atualmente, os FPGA apresentam ainda custos de produção baixos [16, 17, 18].

Este tipo de sistema reconfigurável, com base em plataformas como FPGA, nem sempre é formado apenas pela lógica inserida no FPGA. Na verdade, é bastante comum o uso de um processador independente, com o objetivo de conciliar a execução das instruções. As partes mais complexas e que possam não ser implementadas em FPGA são processadas no processador adjacente. Tudo o resto é implementado no FPGA, através de *software* de síntese digital.

Portanto, os sistemas reconfiguráveis são geralmente constituídos por um processador hospedeiro (a chamada componente fixa) e a lógica reconfigurável do FPGA ou de um dispositivo semelhante (denominada componente variável).

O desempenho de sistemas reconfiguráveis, principalmente com recurso a FPGA, nem sempre tem o mesmo desempenho em termos de frequência de relógio que um processador de propósito geral. Existe por isso uma necessidade de usar técnicas que aumentem essa eficácia [16, 17].

Segundo [17], de todas essas técnicas podemos salientar o aumento de largura de banda de

memória. O FPGA deve explorar o paralelismo de dados e adaptar o comprimento de dados lidos em função do comprimento das instruções que executa. Num processador de uso genérico este comprimento de memória é sempre fixo.

De igual forma, outro processo utilizado é a inclusão de unidades otimizadas para a realização de determinadas execuções específicas. Nos computadores de uso geral as instruções foram concebidas para qualquer aplicação. Quando estamos perante uma aplicação em específico, pode ser produtivo usar uma unidade igualmente específica para essa tarefa. Essas unidades são otimizadas para a tarefa em questão. Um caso concreto são os blocos multiplicadores que um FPGA por vezes incorpora.

Por fim, o desempenho pode ser melhorado com exploração de paralelismo, nomeadamente com a introdução de *pipelines*. Os diversos blocos lógicos que o FPGA dispõe podem ser configurados para implementar este tipo de lógica paralela. Assim, consegue-se uma gestão mais eficiente dos recursos do FPGA bem como um acesso às memórias mais rápido [17, 19].

Com base nas caraterísticas dos processadores vetoriais mencionadas na sub-secção 2.2.2, os FPGA são uma plataforma que proporciona uma implementação deste tipo. Uma aliança deste género tira partido de 3 características dos processadores vetoriais implementados com recurso a *software* de síntese digital [19]:

- **Escalabilidade** — uma variação no número de caminhos de dados dos vetores leva a uma aceleração do processamento. A título de exemplo, um aumento de 2 para 16 caminhos de dados leva a uma aceleração de 1.8 para 6.3 vezes mais.

- **Flexibilidade** — atendendo à capacidade de reconfiguração no que toca aos recursos de um FPGA e, contrariamente ao que sucede com os IC, como é o caso dos ASIC, é possível moldar a implementação do processador a uma dada aplicação em concreto.

- **Portabilidade** — apesar da sua flexibilidade e capacidade de configuração, a vetorização de um processador deste tipo por *software* é de certa forma independente da plataforma onde é implementado. Desta forma, permite que seja exportado para outras famílias de plataformas.

Um dos exemplos de um sistema reconfigurável é o DECPeRLe-1, concebido em 1991. Foi um dos primeiros trabalhos nesta área e carateriza-se por possuir um computador hospedeiro, 4 blocos de memória local de 1MB cada, uma rede de 4x4 FPGA XC3090-10 da Xilnx (também conhecido como PAM) e dispositivos externos de entrada e saída.

O seu funcionamento consiste no carregamento de um ficheiro de configuração do processador hospedeiro para a rede de FPGA. Em seguida, esse processador envia e recebe dados da

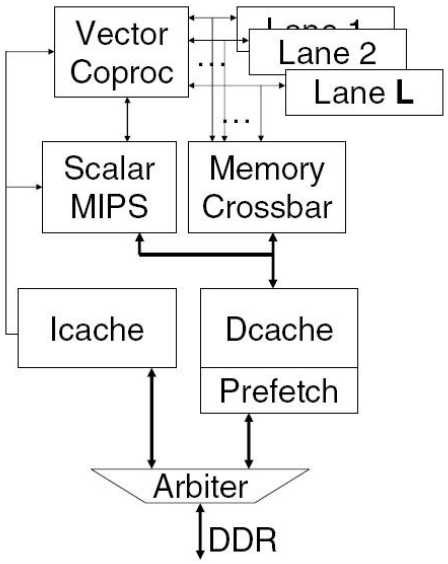

Figura 2.4: Arquitetura VESPA melhorada [19].

rede de FPGA. A lógica do FPGA realiza operações sobre os dados provenientes do processador hospedeiro e devolve-lhe os resultados [20, 17].

Outro exemplo de processador *soft* reconfigurável é o VESPA. Trata-se do uso coletivo de um processador escalar, implementado em *software*, com um processador vetorial passível de ser parametrizado; toda esta arquitetura dedicada é concretizada em *Verilog*.

Segundo [19], genericamente, um processador escalar *soft* é mais lento na execução de código C do que *hardware* dedicado para FPGA. Um dos objetivos do VESPA era baixar este défice de performance. Para isso, a arquitetura inicial do processador sofreu algumas mudanças. Na Figura 2.4 encontra-se a arquitetura VESPA depois das melhorias referidas.

Um dos pontos onde o VESPA, com o seu *hardware* reconfigurável, ganha ao processador escalar é no número de ciclos gasto na execução de uma dada sequência de instruções. Isto deve-se ao facto da introdução de vários caminhos de dados de processamento, passando a ser explorado o paralelismo ao nível da instrução e ao nível da tarefa. Na implementação anterior o caminho de dados só permitia a execução de uma instrução de cada vez.

Também a forma como os resultados são entregues à memória *cache* de dados sofreu alterações. Com a inclusão de um bloco de pré-busca [3], este processo é mais robusto e eficiente porque a falta de um dado na memória *cache* provoca um estado de falha na linha de *cache* em questão e nas linhas que irão ser lidas em seguida.

[3] *Prefetch*

No que concerne aos recursos usados pelo VESPA, o processador também sofreu evolução. Sendo o VESPA um processador de uso geral, as suas instruções respeitam sempre um dado número de bits. A partir da parametrização implementada, as instruções do VESPA podem ser fracionadas em pedaços mais pequenos consoante a necessidade da aplicação a realizar.

Finalmente, uma vez que os vários caminhos de dados de cálculo vetorial e o processador escalar partilham uma memória *cache* de dados, foi criado um módulo controlador [4], que mapeia os dados que vai recebendo nas linhas da memória.

Resumindo, desta forma a flexibilidade dos processadores VESPA é a sua bandeira. Configurando parâmetros como o tamanho dos vetores operados e do módulo regulador é possível causar melhorias substanciais no desempenho [19].

A arquitetura da Figura 2.4 mostra a operação conjunta do processador escalar, que usa instruções MIPS, com o co-processador vetorial, que recorre às instruções da arquitetura VIRAM [21]. Estas instruções baseiam-se no conjunto de instruções MIPS, mas são uma extensão para o processamento vetorial. Os caminhos de dados *Lane 1, Lane 2 ... Lane L* implementam este processamento de uma forma paralela. Ambos os processadores partilham uma memória *cache* que contém as instruções a executar. Finalmente, os processadores VESPA têm um controlador de memória, que regula o acesso e interface das memórias *cache* à memória externa DDR, onde podem ir buscar dados [19, 2].

2.3 Memórias

A disposição da memória segue uma hierarquia para tirar partido do princípio da localização. Este, por seu turno, encerra 2 perspetivas diferentes: localização espacial e localização temporal.

A hierarquia da memória funciona eficientemente quando contém níveis bem definidos. O desempenho de uma arquitetura é amplamente influenciado pela disposição dos dados em memória. Desta forma é crucial uma alocação ponderada dos dados às memórias de forma a aumentar a largura de banda no seu acesso. Neste contexto, o caso ideal seria ter muitas memórias em paralelo ligadas ao processador, com débito de dados em simultâneo. Contudo, esta abordagem não é flexível em arquiteturas de propósito geral.

Em arquiteturas dedicadas e/ou baseadas em FPGA é possível definir a forma de disposição da memória. Nomeadamente, este tipo de arquitetura permite níveis intermédios entre 2 tipos de abordagens de hierarquia de memória. Por um lado, a situação em que existe uma memória de elevada capacidade com todos os dados e latência maior; por outro lado, o caso em que coexistem várias memórias temporárias, mais pequenas, mas que fornecem dados a um débito superior e conduzem a um processamento mais rápido.

[4] *Memory Crossbar*

Na sub-secção 2.3.1, são estudados em mais pormenor 2 casos concretos de memórias RAM. O interesse do seu estudo é o uso corrente nos dias de hoje.

2.3.1 Scratch e Cache

As memórias *scratch* ou memórias locais são tradicionalmente pequenas e rápidas quando comparadas com as memórias *cache*. Por conseguinte, ocupam menos área e consomem menos do que este último tipo de memória; um estudo [22] revela que essa diferença se situa em 34% no que concerne a área utilizada e 40% em termos de consumo de potência.

São assim memórias com características propícias para o armazenamento de dados por pouco tempo. Por outro lado, são memórias para ocupar o topo da hierarquia de memória. Devido a todos estes fatores e a serem memórias com um bloco único, são usadas para simplificar a lógica das memórias *cache* [23]. Assim, o acesso é linear, pois a memória contém os blocos de cada endereço de forma sucessiva, num único bloco.

Deste modo, as memórias *scratch* são usadas em processadores de aplicação específica para implementar um nível de memória local, de armazenamento temporário.

Por seu lado as memórias *cache* são dispositivos não tão rápidos como as memórias locais mas muito eficientes. Operam em conjunto com um componente de armazenamento maior, servindo de ponto intermédio. Por vezes o acesso a esse tipo de componente é mais prolongado e recorrendo às *cache* esse tempo é reduzido. Quando se pretende aceder a um dado, caso esse dado já esteja na *cache* não é preciso aceder ao dispositivo de armazenamento maior. Todas as operações leitura e escrita são sinalizadas com os estados de sucesso ou falha.

Este género de memória é constituído por 2 memórias mais pequenas, uma memória de etiquetas e uma memória de dados. A identificação de acesso a um determinado dado é conseguida através de uma etiqueta e de uma codificação binária para o endereço. A etiqueta é direcionada para a memória de etiquetas que trata de verificar se esta é válida ou não e gerar o estado de sucesso ou falha. A codificação binária do endereço é composta pelo número do bloco e por um número que identifica qual a parte do bloco em concreto é que se pretende o acesso [24, 25].

As memórias *cache* tiram igualmente partido do princípio da localização e tentam otimizar o acesso a dados perto, espacial e/ou temporalmente. Segundo essa perspetiva, as *cache* são localizadas segundo uma hierarquia com vários níveis, por exemplo L1, L2, L3. No nível L1 situam-se as *cache* mais rápidas e pequenas. O tempo de acesso vai aumentando ao longo dos níveis L2 e L3 bem como a capacidade das memórias. A tendência é que as *cache* deste nível sejam colocadas num encapsulamento diferente do processador.

Devido às características referidas anteriormente, as memórias *cache* encontram-se num nível intermédio na hierarquia das memórias. A sua localização reside entre os registos e memórias *scratch*, dispositivos de armazenamento mais rápidos e pequenos, e a memória RAM, de maior capacidade. O enquadramento das memórias *cache* na hierarquia da memória, bem como dos seus

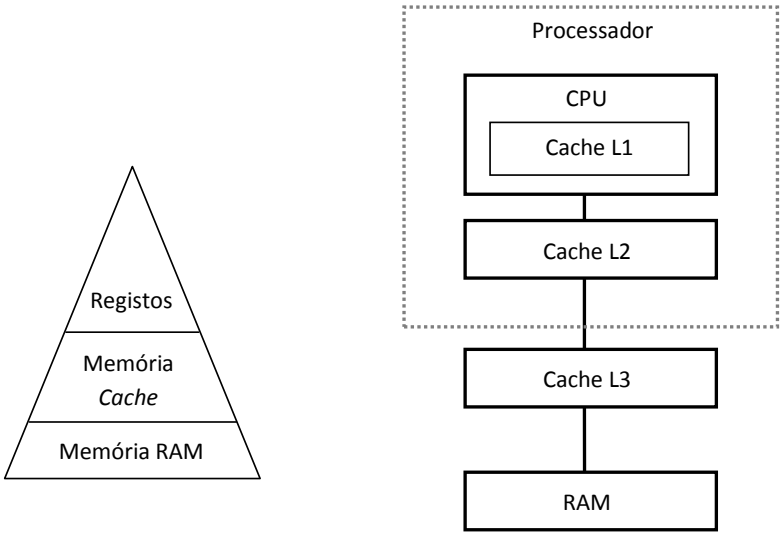

Figura 2.5: Localização das memórias *cache* na hierarquia da memória e estratificação dos seus níveis.

diferentes níveis, encontra-se disposto na Figura 2.5 [24, 25].

Atendendo à associatividade, existem 2 tipos específicos de memória *cache* [24, 26]:

- ***Cache* de Mapeamento Direto** — caso em que cada conjunto da memória *cache* não tem nenhuma divisão. Apenas são usados *bits* para identificar o conjunto pretendido, pois a sua identificação é unívoca. É um caso mais específico do que aquele representado na Figura 2.6, onde a associatividade (n) é 1.

- ***Cache* Associativa** — situação em que a memória é dividida em conjuntos e cada conjunto apresenta n divisões, n linhas de *cache*. Um endereço de acesso à memória é atribuído a um conjunto e dentro desse pode ser relativo a uma das n linhas, dependendo do valor de uma etiqueta. A associatividade é dada pelo número de divisões, n.

  Uma memória deste género com um bloco apenas denomina-se *cache* totalmente associativa. No caso de ter vários blocos com n divisões diz-se que se trata de uma memória associativa de n conjuntos.

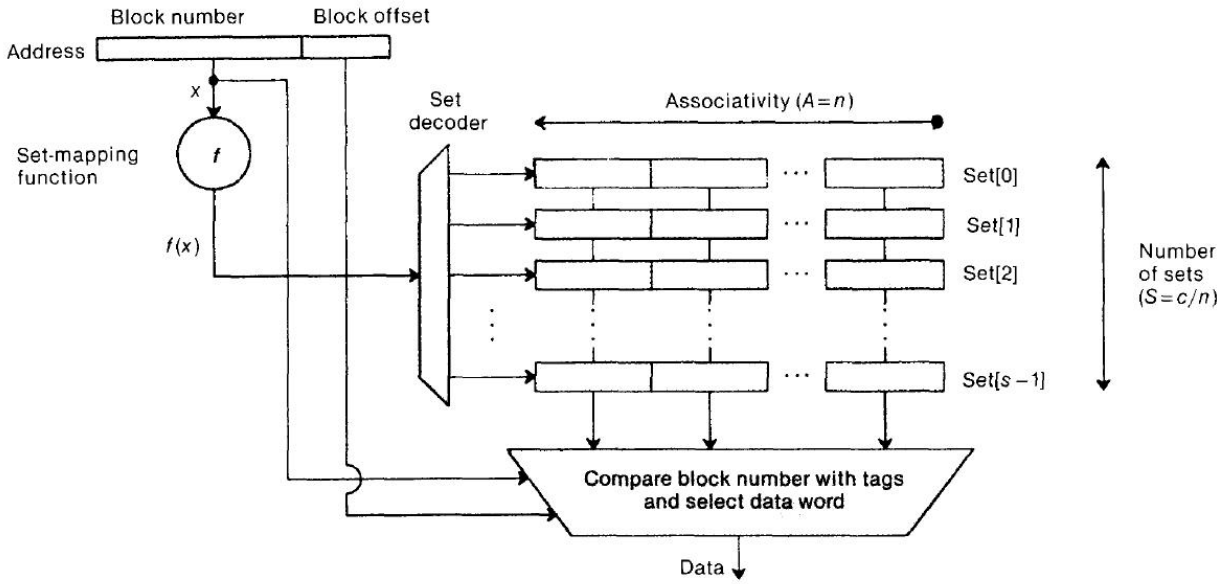

Figura 2.6: Constituição genérica de uma memória *cache* associativa [24].

2.4 Outras Ferramentas

2.4.1 Ferramentas de Apoio ao Projeto de *Hardware* Digital

A linguagem *Verilog* é uma linguagem de descrição de *hardware*[5]. O intuito de *Verilog* é modelizar sistemas eletrónicos e digitais. Apesar de não ser uma linguagem de programação, a sua síntaxe é parecida com a da linguagem C. Nomeadamente no que concerne à implementação de condições e ciclos, *Verilog* também permite a implementação das instruções *if else*, *for* e *while*.

Existem outras linguagens de descrição de *hardware*, como é o caso de VHDL. *Verilog* tem algumas vantagens em relação a estas linguagens e VHDL em particular. Um código escrito em *Verilog* é geralmente mais simples do que VHDL. Desta forma, é preferível usar Verilog para criar as bancadas de teste para simulação de circuitos digitais, por exemplo.

Por outro lado, *Verilog* tem uma desvantagem em relação a VHDL ao permitir uma menor flexibilidade nas funcionalidades que proporciona [27, 28].

Uma vez escrita essa descrição da composição do circuito no geral, a descrição é sintetizada. O resultado dessa síntese é uma rede em que são representados os blocos básicos do circuito, ao nível do registo[6], como por exemplo *flip-flops* e portas de adição e multiplicação lógica [27].

[5] HDL
[6] RTL

Após a fase de síntese lógica, um sistema digital deve ser simulado, para testar o seu funcionamento. A simulação consiste numa série de estímulos aplicados às entradas do sistema e verificação dos resultados obtidos à saída face aos esperados. Para isso, a linguagem *Verilog* dispõe de uma série de funções, como por exemplo as funções *$display* e *$write* que servem para imprimir mensagens no ecrã. Existem várias ferramentas que permitem a implementação desta funcionalidade.

Dentro destes podem ser destacados os *softwares* da *Cadence Design Systems*, *Synopsys*, *Mentor Graphics* e *Xilinx*. Todas estas plataformas são comercializadas e não livres, exigindo por isso uma licença para o seu funcionamento [27, 28].

De entre as opções citadas foram consideradas as ferramentas *ModelSim* da *Mentor Graphics* e o *Xilinx* ISE da *Xilinx*. O *ModelSim* é um simulador de circuitos digitais que permite depuração do seu funcionamento através do acesso a um conjunto de dados gerado pela bancada de teste. Um exemplo destes dados é a visualização das formas de onda dos sinais do circuito simulado.

Por seu turno, o *Xilinx* ISE é um *software* que permite a síntese de modelos de descrição de *hardware*.

Assim, atendendo às funcionalidades que ambas ferramentas proporcionam, a sua utilização é feita em conjunto. Deste modo, os *softwares* atuam de forma complementar no processo de síntese e teste de sistemas digitais.

Capítulo 3

Arquitetura Dedicada

3.1 Introdução

Os trabalhos desta Dissertação compreendem o desenvolvimento de ferramentas de *software* com o objetivo de traduzir e permitir a reconfiguração de uma arquitetura dedicada. Assim, neste capítulo será apresentada a arquitetura proposta como a base de todo o trabalho desenvolvido. Trata-se de uma arquitetura que se engloba na categoria das arquiteturas reconfiguráveis e tem em vista a aceleração de processos de cálculo vetorial.

Importa ainda salientar que a arquitetura descrita já se encontrava desenvolvida antes do início dos trabalhos desta Dissertação. Apenas foi escolhida como arquitetura dedicada alvo, para a qual se devem implementar e adequar o assemblador e simulador, descritos no capítulo 4.

Na secção 3.2 é descrita detalhadamente toda a arquitetura dedicada, com ênfase para as suas funcionalidades e constituição. Na sub-secção 3.2.1 são expostas as partes que constituem e concretizam a arquitetura anterior. As instruções que o processador é capaz de executar assim como a sua explicação encontram-se ilustradas na sub-secção 3.2.2.

3.2 Arquitetura

A micro-arquitetura proposta, embora sendo genérica, está orientada para a sua integração em plataformas FPGA. Este tipo de dispositivos tem progredido bastante nos últimos anos e o resultado é a coexistência na mesma placa de uma imensidão de componentes. Entre a sua constituição contam-se, atualmente, muitos milhares de blocos lógicos, núcleos de cálculo como multiplicadores e blocos de memória RAM interna. Esta evolução permite a adaptação das arquiteturas

para FPGA com vista à implementação de aplicações científicas, como por exemplo algoritmos de cálculo.

Contudo, com vista a atingir os melhores índices de eficiência este género de arquitetura deve incluir andares de *pipeline*. Para alcançar esse objetivo, as unidades *pipelined* precisam de receber os operandos provenientes das memórias a uma taxa máxima. Este débito é conseguido quando é lido um operando por ciclo de relógio. A estratégia típica passa por usar os blocos de RAM interna do FPGA para implementar armazenamento distribuído de dados de forma a aumentar a largura de banda de acesso à memória.

No entanto, esta abordagem não é a melhor dado que as arquiteturas resultantes trazem, geralmente, algumas dificuldades. Em primeiro lugar, os blocos de memória de um FPGA oferecem uma quantidade de memória limitada. Desta maneira é extremamente importante o planeamento da organização da alocação dos dados à memória. Os dados podem ser associados a memória externa, mais lenta e maior, e/ou a memórias distribuídas, com um grande débito de dados.

Esta micro-arquitetura, ao invés disso, apresenta uma estrutura de memórias diferente. A arquitetura usa os recursos de memória do FPGA para constituir um sistema de memórias *cache* programável e que pode ser ajustado à aplicação em concreto. A estrutura em questão explora os recursos internos de memória do FPGA de modo a conseguir a construção de 2 tipos de blocos de memória. Por um lado, blocos heterogéneos (blocos com diferentes organizações, tipo de *cache* e dimensão) de memória para um armazenamento maior e que envolve uma interface com uma memória externa. Por outro lado, memórias locais, mais pequenas, propícias para um armazenamento temporário [29].

A arquitetura mencionada é vetorial, isto é, as suas instruções têm como operandos vetores e não apenas escalares. Desta forma, trata-se de um processador SIMD, em que uma instrução opera simultaneamente sobre vários dados, neste caso os elementos de um vetor. O objetivo desta arquitetura é o processamento de cálculos envolvendo os ditos vetores.

A orientação da micro-arquitetura abordada é o processamento de cálculos do tipo *dataflow*. Este tipo de cálculos caracteriza-se por seguir um fluxo sequencial de operações em que elementos podem ser usados várias vezes em diferentes alturas como operandos. Um exemplo possível encontra-se ilustrado na Figura 3.1, em que é implementada a expressão $D \times [(A-B)+(B-C)]$.

Numa aplicação desenvolvida pelo processador proposto, um núcleo de cálculo do tipo *dataflow* consome operandos vetoriais para gerar resultados também eles vetoriais. Os núcleos de cálculo são compostos por um ou mais ciclos encadeados cujas instruções são embebidas em blocos lógicos do FPGA. Esta representação está patente na Figura 3.2, onde os conteúdos de 2 ciclos *for* são mapeados nos blocos básicos A e B; cada um destes blocos base traduz uma sequência de cálculos do tipo *dataflow* [29].

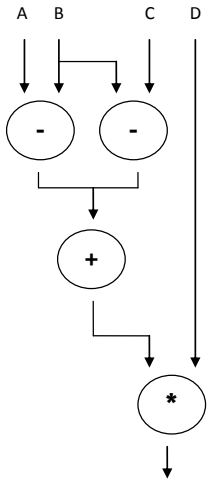

Figura 3.1: Sequência de operações que caracterizam o tipo de cálculo *dataflow*.

```
for(i=0; i<640; i++)
{ ... block A
  for(j=0; j<480; j++)
  { ... block B
  }
}
```


Figura 3.2: Mapeamento de ciclos *for* em diferentes blocos de cálculo. Adaptado de [29].

3.2.1 Partes Constituintes

A arquitetura que fundamenta os trabalhos desta Dissertação inclui na sua constituição um caminho de dados, gerado em conformidade com os blocos básicos encontrados e traduzidos nos ciclos que representam a aplicação. O processador conta ainda com um controlador de micro-código que sequencia as instruções a executar, envolvidas ou não em ciclos encadeados de limites constantes, e vários portos para acesso aos operandos armazenados nas memórias. Esta abordagem genérica da arquitetura deste processador está representada na Figura 3.3.

Uma das diferenças introduzidas pela micro-arquitetura referida anteriormente reside no acesso às memórias. Cada porto de memória tem associado um gerador de endereços, específico para os dados pré-associados a esse porto. A função deste componente é determinar o endereço absoluto em memória de um elemento de um vetor em função de alguns comandos de iteração sobre os seus índices.

Na etapa de síntese do *hardware* de toda a arquitetura cada gerador de endereços é parametrizado com uma série de valores. Estas constantes especificam os endereços inicial e final, dimensões e outros valores relativos a vetores alocados na memória associada a um dado gerador de endereços.

Assim, embebendo estes parâmetros na lógica do gerador de endereços, o cálculo do endereço absoluto torna-se bastante mais simples e eficiente. Nomeadamente, o acesso à memória reduz-se a uma série de subtrações e adições, através de comandos de iteração de pós-incremento/decremento [29].

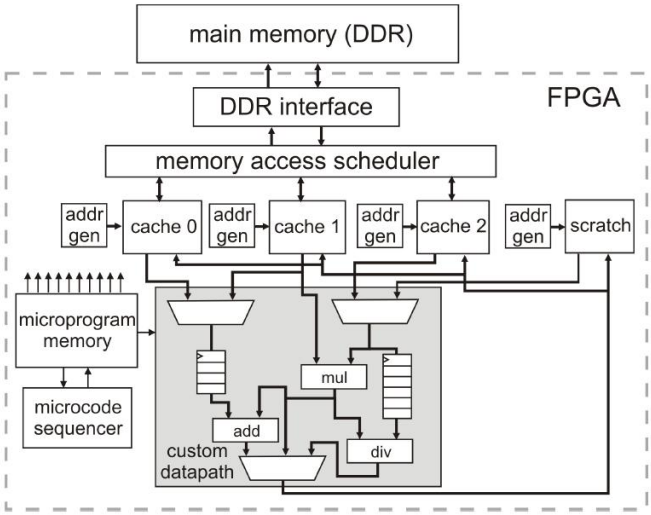

Figura 3.3: Arquitetura vetorial usada [29].

Por outro lado, a latência de acesso às memórias é um dos fatores que mais afeta negativamente o desempenho de um processador. Para que a sua performance seja otimizada é necessário que os operandos sejam "injetados" no caminho de dados por portos de memória distintos. Da mesma forma é imperativo que os dados acedidos já se encontrem nas memórias em questão. Apenas nestas condições é garantido que o débito de leitura é igual a um dado por ciclo de relógio. Atendendo a estas características, o ideal seria ter uma configuração para a estrutura das memórias que mudasse em tempo real, consoante a reconfiguração do caminho de dados. Mas esta opção é inviável uma vez que em tempo real é impraticável a reconfiguração das memórias para uso reutilizável, derivado das suas limitações físicas [29].

Por conseguinte, a solução sugerida passa por atribuir uma memória *cache* a cada porto. Estas memórias são configuradas individualmente no que toca ao seu tamanho e organização. O objetivo passa por associar a cada porto de memória uma série de vetores e fazer posteriormente a configuração tendo em conta as características destes dados. Inicialmente os vetores encontram-se armazenados numa memória externa e são pré-alocados para a estrutura de memórias *cache*. Esta distribuição é feita tendo em conta as necessidades do núcleo de cálculo, para que seja alcançado o estado em que a taxa de dados em paralelo é máxima e a latência de acesso às memórias é mínima.

A Figura 3.4 exemplifica o processo mencionado anteriormente. Considera-se a execução do bloco de código aí descrito, que numa fase lê os vetores A e C e numa etapa posterior lê os vetores M e P. Realizando a alocação dos vetores A e M na memória RAM1 e dos vetores C e P na memória RAM2 é conseguido o acesso em paralelo, com latência mínima no acesso às memórias.

Realizando a mesma análise sobre o exemplo da Figura 3.1 verifica-se que os vetores A, B,

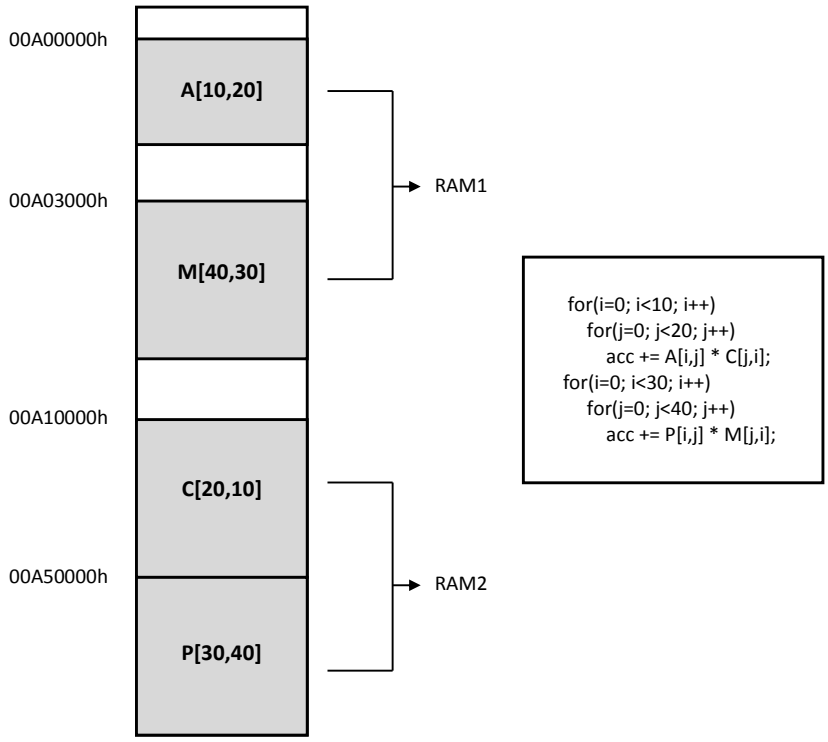

Figura 3.4: Alocação de vetores armazenados na memória externa para as memórias *cache*.

C e D devem ser atribuídos a memórias diferentes de forma a que o seu tempo de leitura seja mínimo. Por uma questão de facilidade, são admitidos os nomes F e G para os vetores dos resultados de $A-B$ e $B-C$, respetivamente, e H para o vetor que contém o resultado da expressão $(A-B)+(B-C)$. É fulcral que os resultados intermédios não sejam armazenados na mesma memória do que entradas do próximo bloco de cálculo. Por exemplo, os vetores F e G devem ser alocados a memórias distintas bem como o vetor H, que não deverá ficar na mesma memória do que o vetor D [29].

Por seu turno, o controlador de micro-código é construído com recurso a uma memória que guarda um conjunto de instruções a executar. O controlador sequencia esse conjunto de instruções segundo comandos de endereçamento cujo formato é específico da aplicação. Estes comandos são explorados na sub-secção 3.2.2.

O controlador de micro-código envolve ainda uma outra secção, responsável pelo controlo da execução de ciclos. O controlo é feito com um campo de apenas 2 *bits* da micro-instrução e permite a execução de ciclos encadeados com um número de iterações constante. Existe a possibilidade de contornar esta restrição, com a utilização de uma memória que contém uma tabela com os limites dos ciclos, embebida em toda a lógica da aplicação. O número de iterações é guardado com um índice que identifica não só a sua posição na tabela mas também o número da ocorrência

de ciclo; por exemplo, a posição 1 da tabela conterá o número de iterações do 1º ciclo a executar e assim sucessivamente. Esta memória pode ser do tipo RAM, passível de ser lida e reconfigurada em tempo real permitindo ciclos com iterações variáveis, ou do tipo ROM, cujo conteúdo é definido juntamente com a restante síntese da arquitetura. A complexidade da parte de controlo de ciclos está mais relacionada com o número de ciclos encadeados e a ordem de grandeza dos contadores de iterações do que com o número total de ciclos a executar no micro-código [29].

A micro-arquitetura anterior foi modelada em *Verilog* HDL, estando organizada nos módulos seguintes:

- **addrgen.v** — é o gerador de endereços, que recebe um comando de 8 *bits* da instrução a executar.
- **datamemory.v** — implementa a memória RAM local de dados, que suporta operações de leitura e/ou escrita com o módulo de topo do processador. Este bloco recebe um endereço de 19 *bits* para efetuar uma dessas operações, definida por um sinal de leitura/escrita de 1 *bit*. Os dados armazenados numa memória deste género são de 8 *bits*.
- **dm_cache.v** — realiza a interface com a memória externa, implementando uma memória *cache* de mapeamento direto. Existem vários sinais que codificam os estados de pronto e permissão para ler e escrever, de forma a não haver leituras ou escritas em simultâneo. Os dados que este módulo passa ao caminho de dados são de 32 *bits*.
- **assocn_cache.v** — módulo semelhante ao módulo *dm_cache.v*, que realiza a interface com a memória externa. A diferença entre ambos é que este módulo regula a transferência de dados de e para memórias *associativas* de n conjuntos; este parâmetro pode tomar o valor 2 ou 4. Os dados que este módulo passa ao caminho de dados são de 32 *bits*.
- **codememory.v** — descreve a memória que guarda o micro-código binário das instruções a processar.
- **lcontrol.v** — módulo cuja função é sequenciar as micro-instruções armazenadas em memória e controlar a execução de ciclos.
- **datapath.v** — módulo responsável por implementar os blocos de operações identificados na aplicação, a parte que efetua cálculos sobre os dados lidos das memórias.
- **toplevel.v** — módulo de nível superior do processador e que compreende todos os módulos anteriores, à exceção das memórias de dados, sejam elas locais (*scratch*) ou *cache*. É também o bloco onde os módulos prévios são interligados entre si.

Apesar de não fazer parte da arquitetura, surge um outro módulo *Verilog* que serve de bancada de teste para simulação de todo o processador. É neste módulo que são instanciados os módulos

das memórias de dados e feitas as respetivas ligações com o módulo de nível superior. No caso das memórias *cache* é também incluído neste módulo o modelo de simulação da memória externa e interface entre si. Por último, são definidos estímulos e valores para os sinais de entrada de modo a poder simular e verificar se os resultados observados à saída estão corretos.

3.2.2 Instruções

O processador mencionado anteriormente usa instruções específicas, adaptadas com vista ao seu uso nesta arquitetura em concreto. Recordando a Figura 3.2, cada ciclo a executar tem um bloco básico que contém as instruções a executar nesse ciclo. Estas instruções, escritas em linguagem C ou semelhante, são de alguma forma transformadas nas instruções dedicadas, compreendidas e passíveis de serem processadas pela arquitetura em foco. Cada uma destas micro-instruções é constituída por alguns campos que unidos compõem uma instrução completa. O conjunto destes campos está representado na Tabela 3.1.

Tabela 3.1: Campos de instrução implementados [29].

Campos de uma Instrução	**Nº de argumentos**	**Exemplo**	**Função**
Noloop	0	——	Execução de instrução sem início nem fim de ciclo
Loop	1	Loop 480	Indicação de início de novo ciclo, com 480 iterações
Next	0	——	Indicação de final do último ciclo em execução
Read	1	Read A[i++,j]	Leitura do vetor A em memória, com incremento da dimensão linha após a execução da instrução
Write	1	Write A[i,j–]	Escrita do vetor A em memória, com decremento da dimensão coluna após a execução da instrução
Set	1	Set A[i++,j]	Incrementar o índice de linha do vetor A, sem operação de leitura ou escrita
Sel	1	Sel A	Seleção do vetor A, para posterior manuseamento
Nop	0	——	Sem operação nenhuma
Halt	0	——	Instrução que indica fim do processamento

A tabela referida encontra-se dividida em 3 partes. Na parte superior são apresentados os campos da instrução responsáveis pelo controlo de ciclos. Na parte central da tabela estão os campos que realizam funções sobre os operandos vetoriais do caminho de dados do processador. Por último, no final da tabela estão os comandos *Halt* e *Nop*, cuja utilização é mais específica e não se engloba em nenhuma função geral dos campos anteriores.

Como é possível verificar na Tabela 3.1, todos os comandos que constituem uma instrução têm um nome próprio. Nalguns casos, esta designação é suficiente para explicitar todo o conteúdo e funções pretendidas com esse campo. Noutros casos, porém, o comando só ganha total sentido quando acompanhado por um argumento que complementa a função principal do campo da instrução. Os campos nesta última situação têm um exemplo da sua utilização com o respetivo argumento na secção Exemplo da tabela anterior; os comandos sem argumento não foram referenciados nessa secção porque a sua utilização é feita recorrendo apenas ao nome do campo. Finalmente, na secção Função da tabela em questão é apresentada com mais detalhe a tarefa desempenhada por cada comando [29].

Os campos das instruções podem ser expressos por uma linguagem textual, de mais alto nível, o tipo de linguagem abordado na tabela anterior. Por outro lado, cada um dos campos de uma instrução tem associada uma codificação binária, de modo a ser executado pelo processador. Desta forma, uma instrução, sendo constituída por vários comandos, irá ter também uma codificação binária, resultante da junção das codificações de todos os campos que a compõem.

Assim, o aspeto típico de uma instrução será a representação textual de um conjunto de campos. Adicionalmente, pode surgir um campo para configuração da função do caminho de dados. Este, embora não fazendo parte da instrução nem da lista de comandos válidos, é incluído na mesma linha que estes. Um exemplo poderá ser a série de comandos:

```
Noloop; Nop; Set A[i--,j--]; DP_CONF_1;
```

O número de campos de uma instrução depende diretamente do número de portos de memória associado ao caminho de dados do processador. Uma instrução é composta por $N+1$ comandos, sendo N o número de portos de acesso à memória. Cada um dos N comandos é constituído, pela ordem seguinte, por [29]:

- **2 conjuntos de 3 *bits*** — cada campo destes controla a operação a realizar sobre uma dada dimensão de um vetor em memória. Um dos conjuntos destina-se à dimensão linha, *i*, e outro à dimensão coluna, *j*. As combinações representadas por estes 3 *bits* estão ilustradas na Tabela 3.2.

Tabela 3.2: Comandos para manipulação dos índices dos vetores. Adaptado de [29].

Código Binário	**Função**	**Mnemónica**
000	Manter índice	M[i,j]
001	Pós-incremento	M[i++,j]
010	Pós-decremento	M[i–,j]
011	Somar constante	M[i+S,j]
100	Subtrair constante	M[i-S,j]
101	Colocar a 0	M[i=0,j]
110	Colocar no valor máximo	M[i=MAX,j]
111	Reservado	——

A tabela contém a devida função para cada uma das 8 codificações binárias possíveis. Para além disso, em cada caso indica ainda a mnemónica respetiva, de forma a tornar mais percetível e explícita a forma como este campo de 3 *bits* é usado numa descrição das micro-instruções de nível superior.
No caso do vetor afeto pelo comando ser unidimensional, a combinação referente à dimensão *j* fica com o valor 000 e à dimensão *i* é atribuído o valor adequado, em função da operação correspondente. A combinação 111 apenas foi usada para o comando *Halt*, estando reservada a essa utilização apenas.

- **2 *bits*** — este campo controla a operação a realizar sobre o vetor implícito no comando da instrução. Existem 4 combinações que codificam as operações possíveis nesta arquitetura e estão representadas na Tabela 3.3.

Tabela 3.3: Comandos para operar vetores armazenados nas memórias. Adaptado de [29].

Código Binário	**Função**
00	Sem operação
01	Ler dados
10	Escrever dados
11	Selecionar vetor

Assim, cada comando de uma instrução associado a um porto de memória tem 8 *bits*. O comando sempre presente em qualquer instrução é um campo de 2 *bits* que regula a execução de ciclos. As combinações binárias implementadas para este comando são 4 e estão descritas na Tabela 3.4.

De realçar que a combinação 10 não é atribuída em nenhum dos comandos em uso; um possível destino seria a implementação de mecanismos que permitam a finalização ou continuação de

Tabela 3.4: Comandos para o controlo de ciclos. Adaptado de [29].

Código Binário	**Função**
00	Sem ciclo
01	Início de ciclo
10	Sem uso
11	Fim de ciclo

um ciclo a meio do mesmo, semelhantes às diretivas *break* e *continue* em linguagem C, respetivamente [29].

Por conseguinte, cada instrução terá $N \times (2 \times 3 + 2) + 2$ *bits*, onde N é o número de portos de acesso à memória. A Figura 3.5 mostra a disposição dos diversos campos para a construção de uma instrução. O exemplo concreto refere-se a uma instrução onde ocorrem operações sobre vetores bidimensionais e com recurso a 2 portos de memória distintos [29].

Para além disso, a micro-instrução poderá ainda conter um campo de controlo do caminho de dados, para configuração das operações a realizar. Este campo é, naturalmente, específico da aplicação e não intefere com o restante mecanismo de controlo do processador.

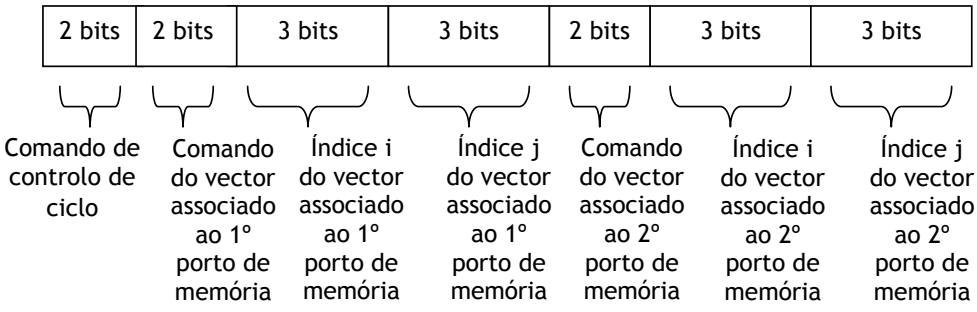

Figura 3.5: Constituição de uma instrução.

3.3 Resumo

Neste capítulo foi exibida a arquitetura vetorial dedicada, usada no decorrer dos trabalhos da Dissertação. Nesse contexto, foi dada uma noção geral da arquitetura, com foco na sua descrição e das suas funcionalidades. Em seguida, foi detalhada a organização da arquitetura com especial

detalhe para as partes componentes da mesma. Finalmente, foram ainda abordadas as micro-instruções passíveis de serem executadas pelo processador, com pormenor para a sua constituição.

Capítulo 4

Implementação

4.1 Introdução

No capítulo 3 foi abordada a micro-arquitetura dedicada de um processador vetorial, que servirá como base durante esta Dissertação. Durante a realização dos trabalhos inerentes à mesma, o objetivo passa por criar um conjunto de ferramentas de *software* que permita 2 funções essenciais. Por um lado, a tradução de uma linguagem simbólica de alto nível em micro-código binário. Por outro lado, a geração e posterior simulação da arquitetura dedicada, resultante de alguns parâmetros escolhidos para a sua configuração.

Neste capítulo é relatado o desenvolvimento do trabalho em torno das ferramentas com funções de tradutor, assemblador e simulador. Na secção 4.2 é exposto o fluxo no contexto global do projeto total desde que uma aplicação é representada em ficheiros de descrição até à sua implementação e simulação.

Na secção 4.3 são mostrados os ficheiros que descrevem a arquitetura do processador; nestes ficheiros são incluídas algumas informações sobre a arquitetura, traduzidas para possibilitar a sua reconfiguração.

O conteúdo da secção 4.4 visa explicitar a estrutura do micro-assemblador que realiza a tradução do código simbólico em código binário. Além disso, são apresentadas as funcionalidades implementadas em detalhe.

O processo de geração dos módulos *Verilog* da arquitetura é exibido em pormenor na secção 4.5.

Finalmente, na secção 4.6 é detalhado o algoritmo Sobel, exemplo considerado para implementação do caminho de dados da arquitectura.

4.2 Fluxo Geral de Projeto

O trabalho relativo a esta Dissertação enquadra-se num projeto mais geral. De uma forma global, o objetivo reside na implementação de uma dada aplicação por parte de uma arquitetura gerada de forma dedicada e posteriormente simulada. O fluxo geral de todas estas atividades está ilustrado na Figura 4.1 e é detalhado em seguida.

O ponto de partida é uma aplicação qualquer que envolva processos de cálculo com vetores. Tipicamente estas aplicações são descritas com recurso a uma linguagem de alto nível, como por exemplo C. De modo a criar uma arquitetura que implemente esta aplicação, é necessária uma descrição da implementação desejada. Nesta fase, esse processo é manual e o utilizador define ficheiros de descrição da arquitetura que a caracterizam.

Os trabalhos desenvolvidos situam-se nesta secção do fluxo de projeto, assinalada na Figura 4.1 dentro de um retângulo em evidência. Assim, envolvem a criação de uma forma que descreva e defina a arquitetura. Numa etapa posterior, ferramentas de *software* usam os ficheiros produzidos para 2 funções principais. Por um lado, a assemblagem de micro-código binário que traduz uma série de instruções que implementam o bloco de cálculos requerido. Por outro lado, recorrendo ainda a uma base de modelos *Verilog* previamente desenvolvidos, a geração de toda a arquitetura que concretiza a aplicação pretendida. Todos estes aspetos são abordados em detalhe nas secções 4.3, 4.4 e 4.5.

É importante realçar ainda que o processo de tradução entre as codificações do caminho de dados e os respetivos módulos é igualmente manual.

Finalmente, o ciclo de projeto termina com a síntese e simulação da arquitetura criada. Para alcançar tal objetivo, são usados 2 *softwares* para tarefas distintas. O *ModelSim* é utilizado com a função de simulação do desempenho da arquitetura. Os resultados podem ser observados através das formas de onda dos sinais que este programa gera, por exemplo.

Por seu turno, o *Xilinx* ISE sintetiza a arquitetura realizada automaticamente pelo processo de geração. A implementação da arquitetura é feita através de um ficheiro *.bit*, que define as informações necessárias para a sua integração numa plataforma FPGA, neste caso.

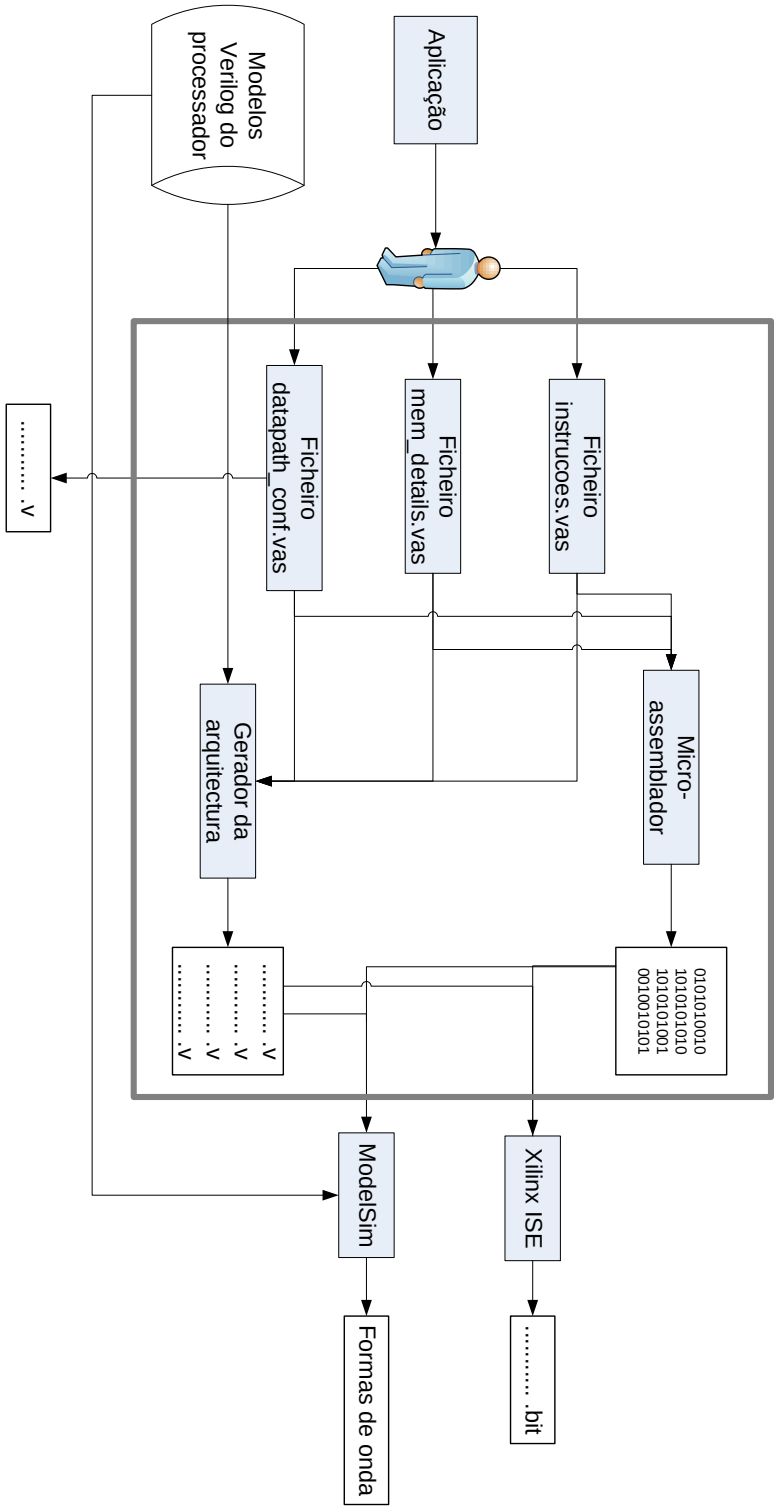

Figura 4.1: Fluxo geral do projeto, com ênfase para a parte trabalhada ao longo desta Dissertação. Esta encontra-se encerrada no retângulo em realce.

4.3 Ficheiros de Descrição da Arquitetura

Um dos grandes objetivos deste trabalho consistiu na configuração da arquitetura vetorial, de forma a possibilitar a sua moldagem a uma aplicação em concreto. Com vista em alcançar essa meta surgiu a necessidade de definir um método de descrição da arquitetura. Em particular, este formato tem que proporcionar detalhes e parâmetros sobre os blocos constituintes da arquitetura. Adicionalmente, esta descrição deve incluir a configuração das memórias da arquitetura e a respetiva alocação de vetores às mesmas.

Usar um único ficheiro para descrever vários detalhes da arquitetura tornaria mais complicado identificar as várias partes componentes da mesma. Recorrendo a vários ficheiros, é possível dividir a arquitetura de uma forma mais intuitiva e simples. Assim, é proposta uma separação da descrição por vários ficheiros. A abordagem considerada realiza a divisão referida em relação à funcionalidade dos vários blocos da arquitetura. Tendo em conta a constituição genérica da arquitetura da Figura 3.3, foram consideradas 3 divisões principais e, como tal, foi atribuído um ficheiro a cada uma dessas divisões.

Em primeiro lugar, um dos ficheiros em causa contém as micro-instruções que são traduzidas para micro-código. Outro dos ficheiros do conjunto de descritores da arquitetura tem como objeto a configuração do caminho de dados do processador. Finalmente, surge um terceiro ficheiro que descreve os portos de acesso à memória da arquitetura, respetiva alocação de vetores e suas características; o ficheiro em questão encerra ainda a definição de entradas e saídas do caminho de dados.

Deste modo, verifica-se que as informações patentes no conjunto de 3 ficheiros são complementares entre si. Para além disso, existem alguns dados relativos à mesma parte da arquitetura descritos em ficheiros diferentes. Um bom exemplo disso é a descrição das entradas e saídas do caminho de dados no ficheiro relativo aos portos de memória. A razão para isto acontecer prende-se com a circunstância das entradas e saídas do caminho de dados estarem associados aos portos de memória. Assim, a localização do seu relato poderia estar num desses 2 ficheiros.

Em seguida é apresentado o conteúdo dos ficheiros, separadamente, em detalhe. Aquando da sua utilização foi especificado um nome para cada descritor considerado. Ao longo desta secção, os ficheiros são, algumas vezes, mencionados recorrendo a essa identificação exemplo.

4.3.1 Ficheiro com Micro-Instruções

A constituição deste ficheiro envolve, como referido anteriormente, a sequência de micro-instruções a executar. Nomeadamente, como visto na secção 3.2.2, cada instrução é composta por alguns comandos. A codificação textual destes comandos é apresentada neste ficheiro que contém o conjunto completo de campos; por outras palavras, encerra na totalidade um micro-programa executável pelo processador vetorial.

O formato definido para este ficheiro está representado na Figura 4.2. A estrutura adotada tem o seu início com um cabeçalho de comentários; o objetivo desta parte do ficheiro é contribuir para a sua identificação bem como deixar indicações sobre o modelo usado. Na parte descritiva propriamente dita, o formato consiste em exprimir uma instrução por linha. O primeiro número de cada linha identifica igualmente o número da instrução.

A estrutura utilizada implica escrever sempre o número da linha em primeiro lugar, prosseguido do sinal dois pontos. Depois deste sinal de início de nova linha, aparecem os campos da instrução devidamente ordenados. A separação dos vários comandos é feita com recurso ao ponto e vírgula. Com este método torna-se possível identificar claramente não só a função mas também o número de comandos que compõem cada instrução.

O nome dado ao ficheiro neste exemplo concreto foi *instrucoes.vas*, devido ao conteúdo que armazena. O pedaço de código apresentado é escrito para uma arquitetura com 2 portos de memória associados. Os vetores TESTE e EXP estão alocados respetivamente a cada um desses portos de acesso. Os nomes atribuídos aos vetores são meramente experimentais, com a codificação a permitir o uso de qualquer nome.

A nomenclatura considerada permite que o comando *Halt* codifique sozinho toda a instrução de paragem do processador. Assim, ao invés de todas as outras instruções que necessitam de vários campos para a sua representação, este comando traduz todos os bits desta instrução.

```
//Micro-instructions
//Number of line: Field 1; Field 2; .....
  0: Noloop;     Sel TESTE; /*Sel B*/     Sel EXP;                DP1;
/*0: Noloop;     Sel TESTE;               Sel EXP;*/
  1: Noloop;     Nop;                     Set EXP[i--,j--];       DP2;
  2: Loop 480;   Read TESTE[i,j++];       Write EXP[i,j++];       DP1;
  3: Loop 319;   Read TESTE[i,j++];       Write EXP[i,j++];       DP2;
  4: Next;       Read TESTE[i,j++];       Write EXP[i,j++];       DP1;
  5: Noloop;     Read TESTE[i++,j=0];     Write EXP[i++,j=0];     DP1;
  6: Next;       Nop;                     Set EXP[i,j--];         DP3;
  7: Halt;
```

Figura 4.2: Ficheiro com micro-código simbólico (*instrucoes.vas*).

Em último lugar, este ficheiro inclui uma indicação do caminho de dados usado em cada instrução. A ideia é codificar esta designação e incluir 2 *bits* adicionais no micro-código gerado. Deste modo, é possível reconfigurar o núcleo de cálculo do processador entre instruções.

4.3.2 Ficheiro de Configurações do Caminho de Dados

Outro dos ficheiros criados é o ficheiro que especifica o caminho de dados a incluir na arquitetura. Este ficheiro foi chamado de *datapath_conf.vas*, pois apresenta configurações relativas ao caminho de dados. Um exemplo deste ficheiro está representado na Figura 4.3.

Foi incluído ainda um cabeçalho que identifica o ficheiro e explicita em que consiste cada campo de uma linha. Neste caso o primeiro campo é o nome pretendido para o módulo do caminho de dados. Por seu turno, o segundo campo é uma identificação binária para diferenciar da designação textual e que é incluída no micro-código binário gerado. Finalmente, o terceiro e último campo traduz a latência do respetivo caminho de dados dessa linha; a razão da inclusão deste valor é poder passar essa informação ao módulo de topo do processador, para que este induza um atraso com esse número de ciclos de relógio no processamento. Assim, o módulo de mais alto nível atrasa os sinais de controlo para escrita dos resultados.

Todos os campos mencionados anteriormente estão delimitados por ponto e vírgula. A escolha entre as alternativas possíveis para o caminho de dados é feita com base na indicação dada no ficheiro *instrucoes.vas*. Caso a designação seja válida, é assumida a configuração correspondente presente no ficheiro *datapath_conf.vas*.

```
//Datapath configuration
//Identifier BinaryCode Latency
SOBEL;        0;    5;
PIXEL_COMP;   1;    6;
```

Figura 4.3: Ficheiro com configurações do caminho de dados (*datapath_conf.vas*).

4.3.3 Ficheiro de Portos de Memória

O único ficheiro do conjunto ainda não apresentado caracteriza o ambiente em torno dos portos de acesso à memória. Este ficheiro usa uma forma própria, criada neste passo de raíz, para proceder à descrição desta parte constituinte da arquitetura. Entre os aspetos incluídos nessa descrição estão a alocação de vetores aos portos de memória, a configuração das memórias componentes do processador e identificação de entradas e saídas do caminho de dados. Pelas razões citadas, o nome dado a este ficheiro foi *mem_details.vas*. A Figura 4.4 ilustra esse ficheiro descritor.

```
//Memory description
//Memory name  Vectors stored in memory  Type of memory
//Vector[dimensions]:starting address:end address:S dimension:Stride i:Stride j
//scratch       : local memory
//cachedm       : direct mapped cache
//cachenwa      : cache n ways associative
//Cache specification:
//4             : number of ways
//32            : number of sets
//8             : number of blocks per line (K*128 bits or K*32 bytes)
//Memory work mode
MEM0:  TESTE[5,6]:1:100:5:1:2; /*IGN[2,3]:4:5;*/ \\
       B[10,50]:200:300:10:5:10; C[5,3]:400:500:1:1:1; K[5,3]:550:650:1:1:1;
       cachenwa; 4; 32; 8; RW;
MEM1:  DF[100]:700:800:10:10; EXP[34,7]:850:950:10:10:1; scratch; 1024; RW;
/*MEM2:F[1,1]:500:25:100; G[12,50]; H[100]; cachenwa; 4; 32; 512; RW;*/
input: INA:MEM0;
input: INB:MEM0;
input: INC:MEM1;
output: SQRT; MEM0;
output: SQRT2; MEM1;
```

Figura 4.4: Ficheiro com especificações dos portos de acesso às memórias (*mem_details.vas*).

Tal como nos casos anteriores, este ficheiro foi dotado de um cabeçalho que explicita o seu conteúdo. Também de forma análoga aos ficheiros anteriores, o cabeçalho identifica o conteúdo do ficheiro e dá uma noção básica da sua organização. Neste sentido, são detalhados alguns pormenores mais específicos e passíveis de dúvida, como os parâmetros da organização da *cache* associativa.

A estrutura do ficheiro *mem_details.vas* divide-se em 2 partes. Na primeira fração, é apresentada uma descrição dos portos de memória. Para esse efeito foram definidos vários campos. Em primeiro lugar é indicado o nome da memória a que o porto está associado, seguido do sinal dois pontos. Depois da ocorrência deste delimitador inicia-se a enumeração dos vetores alocados a cada porto de memória.

A identificação de um vetor inicia-se pelo seu nome, acompanhado da indicação do número de elementos de cada dimensão. Estes valores são incluídos dentro de parêntesis retos e separados por vírgula. Após a declaração do vetor, delimitados pelo sinal dois pontos, aparecem os seguintes parâmetros desse vetor:

- Endereço inicial em memória;
- Endereço de fim em memória;
- Ordem de grandeza das constantes de linha e coluna;

- Constante da dimensão linha;
- Constante da dimensão coluna;

A descrição de um vetor termina com ponto e vírgula. Em seguida à representação de todos os vetores, são indicadas as configurações da memória. Estas começam com o tipo de memória e os dados respetivos. A Tabela 4.1 resume os tipos de memória considerados e respetivos parâmetros usados para cada um. Na tabela referida, a coluna do meio traduz os termos usados para codificar os diferentes tipos.

Como a Tabela 4.1 assinala, os parâmetros das memórias são igualmente separados por ponto e vírgula. O último campo da descrição, que surge depois destes parâmetros, é o modo de funcionamento das memórias. Os 3 modos em uso encontram-se revelados na Tabela 4.2. A primeira coluna apresenta a codificação definida para cada um dos modos. Finalmente, a segunda coluna explicita o seu significado.

A segunda parte do ficheiro *mem_details.vas* define as entradas e saídas do caminho de dados. Uma vez que estas se encontram associadas a portos de memória, nesta fração são descritas também essas ligações. A nomenclatura escolhida determina que a identificação das entradas e saídas é feita com recurso às palavras *input* e *output*, respetivamente. Este nome é seguido pelo sinal dois pontos, que delimita o primeiro campo.

No caso das entradas, após a identificação surge o nome da entrada. A relação entre uma entrada e uma memória é unívoca, isto é, uma entrada liga a um e um só porto de memória. Desta forma, o modelo criado estipulou que depois do nome da entrada, separado por dois pontos, surge o nome do porto de memória. A descrição de uma entrada termina com ponto e vírgula.

Relativamente às saídas, a notação é semelhante mas não existe a restrição de apenas uma saída ligar a uma memória. Assim, depois da indicação de *output* aparece igualmente o nome da saída. Em seguida, é escrita a lista das memórias onde liga a saída em questão, separadas por

Tabela 4.1: Descrição das memórias.

Tipo de memória	**Codificação**	**Parâmetros descritos**
Scratch	scratch	Tipo de memória; Tamanho do bloco de memória;
Cache Associativa	cachenwa	Tipo de memória; Número de linhas da cache; Tamanho de cada bloco da cache; Número de maneiras associativas;
Cache de Mapeamento Direto	cachedm	Tipo de memória; Número de linhas da cache; Tamanho de cada bloco da cache;

Tabela 4.2: Modos de funcionamento da memória.

Modos de funcionamento da memória	Significado
RO	Apenas leitura
WO	Só escrita
RW	Leitura e escrita

ponto e vírgula.

Assim, é possível configurar o processador recorrendo aos ficheiros anteriormente apresentados. A configuração exemplo proporcionada pelos ficheiros referidos está ilustrada na Figura C.1.

4.4 Micro-Assemblador

4.4.1 Estrutura e Função Principal

Como referido anteriormente, o objetivo principal proposto para o micro-assemblador é realizar uma interpretação de ficheiros que configuram e descrevem o processador. Assim, deve ser capaz de ler o conteúdo dos ficheiros referidos na secção 4.3, processar o seu conteúdo atendendo ao seu significado simbólico e, por fim, gerar o micro-código binário que o processador executa. A geração dos módulos *Verilog* que irão compor a arquitetura de todo o sistema ficou entregue a um programa separado, explorado na secção 4.5.

Surge, portanto, deste modo a necessidade da seleção de uma linguagem de programação para a implementação destes *softwares*. Desta feita, a escolha recaiu sobre a linguagem C, sendo os *softwares* Cygwin a opção para compilador. Trata-se de um grupo de *softwares* criados para Microsoft Windows com a meta de proporcionar um ambiente semelhante ao sistema Unix. Desta forma, entre outros instrumentos, o Cygwin possibilita a integração do compilador tradicional de ambiente Unix, o GCC [30].

Com a escolha das ferramentas definida, iniciou-se o processo de escrita do código C. Inicialmente, a execução do micro-assemblador era feita sem argumentos. Considerando a função de um micro-assemblador, posteriormente foram incluídos alguns argumentos para tornar a sua execução mais funcional e flexível. Desta forma a estrutura do código do micro-assemblador inicia-se com a análise de cada um destes argumentos. Estes encontram-se apresentados na Tabela 4.3.

A função de tradução do micro-assemblador implica que este seja capaz de abrir os ficheiros descritores e gere um ficheiro com o micro-código do processador. A extensão exemplo usada

Tabela 4.3: Argumentos obrigatórios do micro-assemblador.

Argumentos	**Função**
-r	Próximo nome de ficheiro que sucede este argumento é um ficheiro de leitura
-w	Próximo nome de ficheiro que sucede este argumento é um ficheiro de escrita
-m	Precede e identifica o ficheiro de descrição das memórias da arquitetura
-c	Precede e identifica o ficheiro das instruções a executar no processador
-d	Precede e identifica o ficheiro com as configurações do caminho de dados

para este ficheiro foi a extensão *.bin*. Assim, e tomando os nomes exemplo dos ficheiros da secção 4.3, o conjunto de argumentos usados para executar o micro-assemblador deverá ser o seguinte:

```
./assemblador -r -m mem_details.vas -r -d datapath_conf.vas -r -c instrucoes.vas -w
    code.bin
```

O primeiro argumento diz respeito ao executável do programa do micro-assemblador. Os 9 argumentos seguintes traduzem a abertura para leitura dos ficheiros da descrição da arquitetura. Por último, é aberto o ficheiro *code.bin*, que irá conter o código binário resultante do processo de tradução.

Aquando da sua execução, o programa do micro-assemblador começa por ler os argumentos todos. Na ausência de erros, o passo seguinte é verificar a ordem dos ficheiros descritores. A ordem pela qual estes são referenciados não é irrelevante. Foi estabelecido previamente que a ordem dos ficheiros deverá ser: *mem_details.vas*, *datapath_conf.vas* e depois *instrucoes.vas*.

Caso a ordem anterior esteja correta, o processo de tradução segue o seu percurso normal. Por outro lado, caso ocorra alguma imprecisão, o micro-assemblador retorna com a explicitação do erro ocorrido.

O esquema geral da execução do micro-assemblador encontra-se ilustrado na Figura 4.5. Nos próximos parágrafos é detalhado o processamento do micro-assemblador sobre os argumentos e os ficheiros de leitura lidos.

O micro-assemblador lê todos os argumentos explicitados na sua execução, sem exceção. O primeiro tratamento consiste em contar os argumentos para verificar que os argumentos mínimos estão listados. Estes consistem no conjunto de 11 argumentos apresentado atrás, exceto o argumento do executável do programa. O número pode ser excedido porque foram implementados

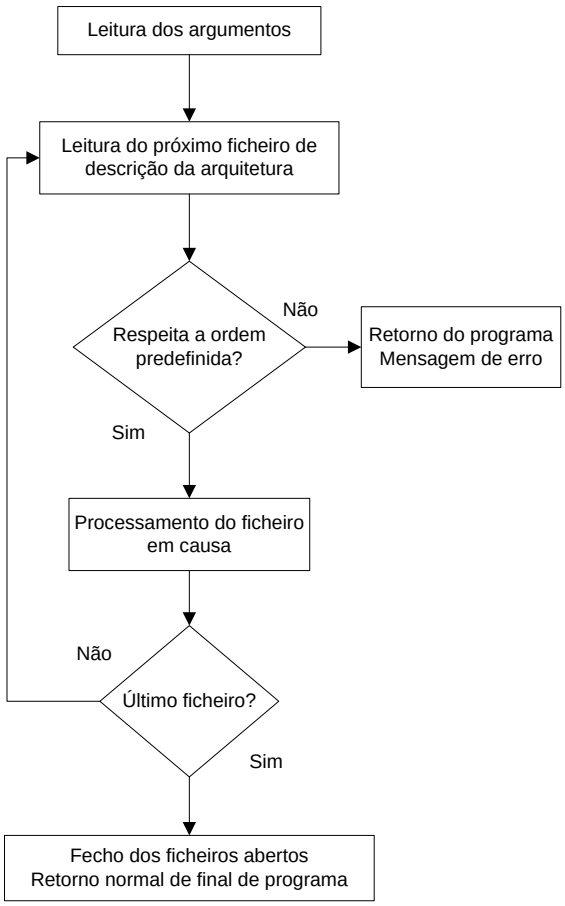

Figura 4.5: Sequência de tarefas do micro-assemblador.

outros argumentos, facultativos, com funcionalidades adicionais. Deste modo, é conseguido o controlo na ocorrência de erro nos 2 casos possíveis; a situação em que o número limiar de argumentos não é atingido ou de algum argumento não fazer parte do conjunto dos argumentos obrigatórios ou facultativos. Em qualquer uma das circunstâncias a execução do programa é suspensa e é emitida uma mensagem de erro.

Na inexistência de situações anormais, o micro-assemblador continua a ler os argumentos um a um e a cumprir a sua respetiva função. Este ciclo termina quando surge o último argumento. Depois da execução desse argumento o micro-assemblador entra na fase de processamento dos ficheiros de descrição da arquitetura vetorial.

Todo o processo de interpretação dos argumentos está resumido na Figura 4.6.

O primeiro ficheiro a ser processado, como referido anteriormente, é o ficheiro de descrição das memórias da arquitectura. O micro-assemblador lê o ficheiro linha a linha e trabalha sobre cada uma antes de ler a próxima. A tarefa inicial consiste em ignorar completamente qualquer

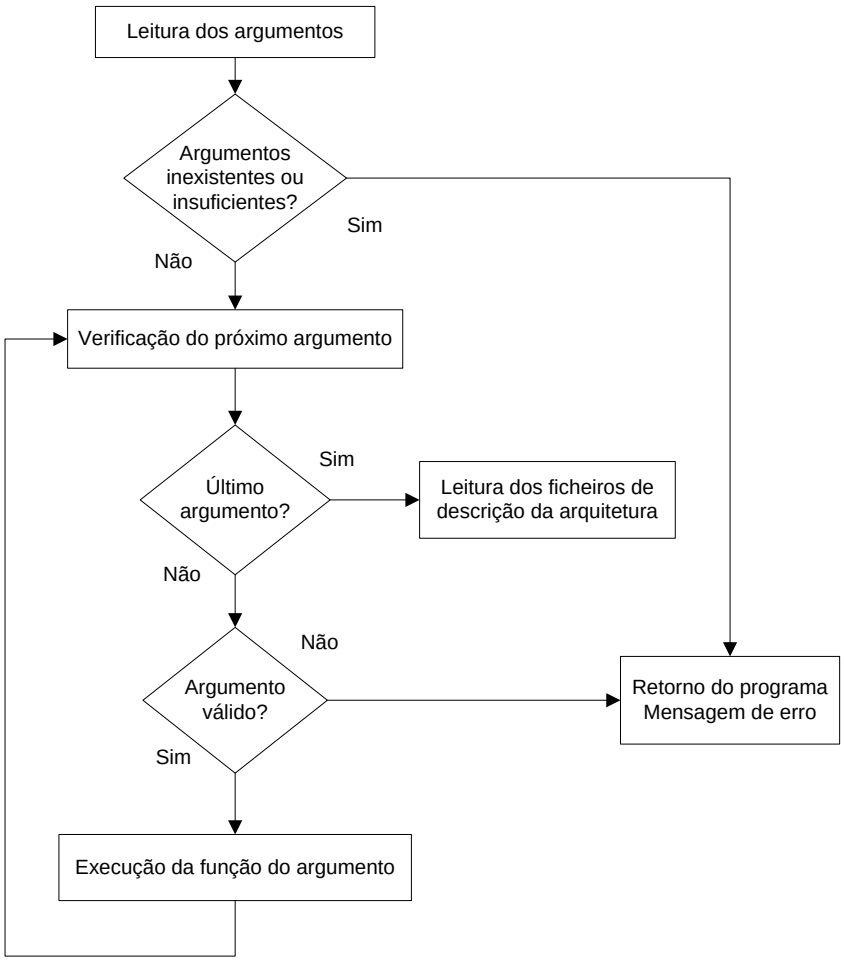

Figura 4.6: Tratamento dos argumentos na execução das ferramentas de *software*.

linha comentada ou os fragmentos comentados na sua constituição.

Após a filtragem anterior o conteúdo do ficheiro é interpretado pelo micro-assemblador. Nesta fase, onde apenas é gerado o micro-código binário, a parte do ficheiro que detalha as entradas e saídas não é tratada porque não influencia a tradução. O processamento recai, assim, sobre os portos de memória e respetivos valores. Este funcionamento pode ser consultado na Figura 4.7. Como referido, nesta fase o micro-assemblador opera sobre a primeira parte do ficheiro *mem_details.vas*, o que corresponde apenas à metade esquerda do fluxograma da Figura 4.7.

A leitura dos vetores começa pelos valores das suas dimensões e restantes constantes. O micro-assemblador testa os valores lidos com o intuito de detetar inconsistências entre o número de dimensões declarado e respetivas constantes de linha e coluna. Se não estiver presente uma situação de erro, o programa guarda os elementos da linha nas variáveis respetivas. Os vetores

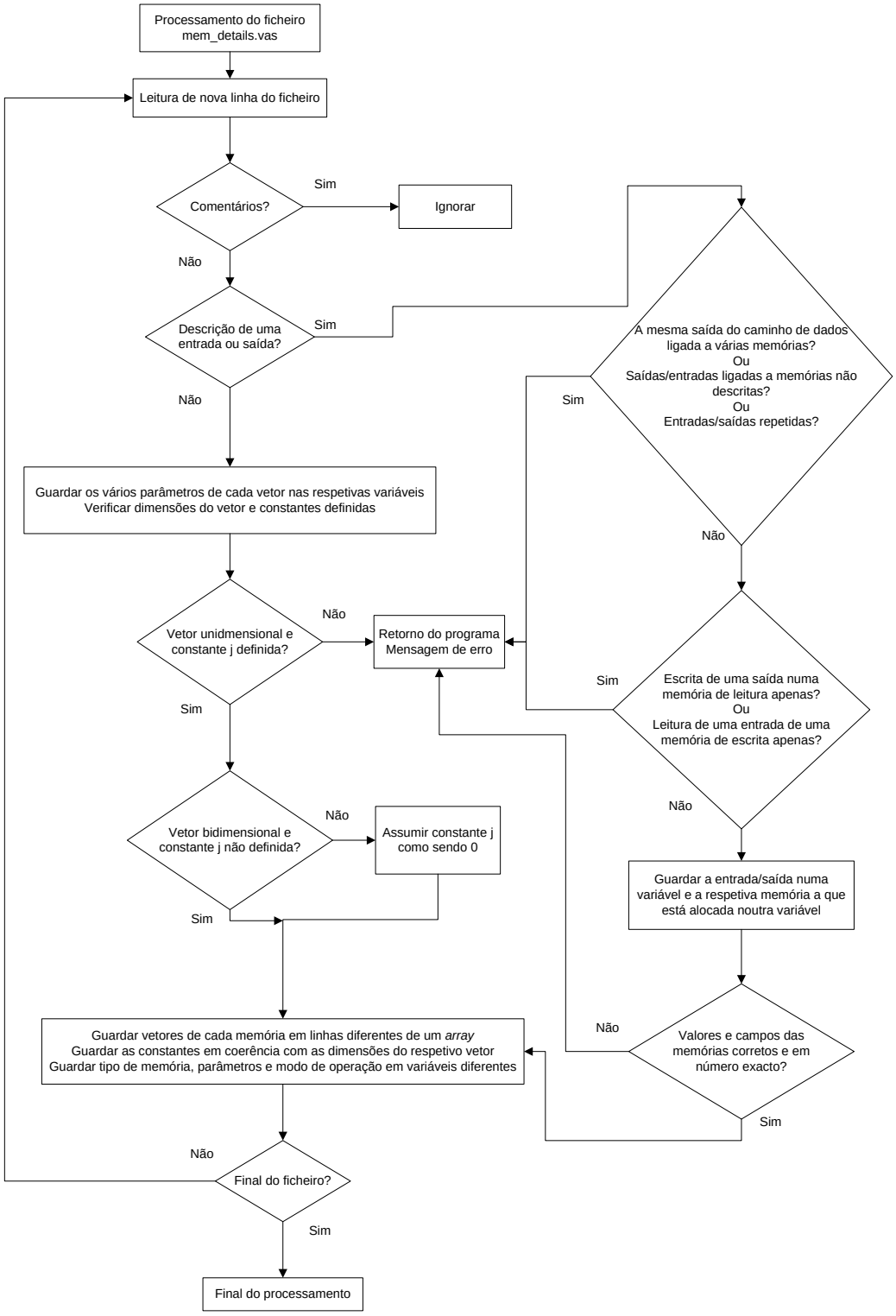

Figura 4.7: Processamento do ficheiro *mem_details.vas* pelas ferramentas de *software*.

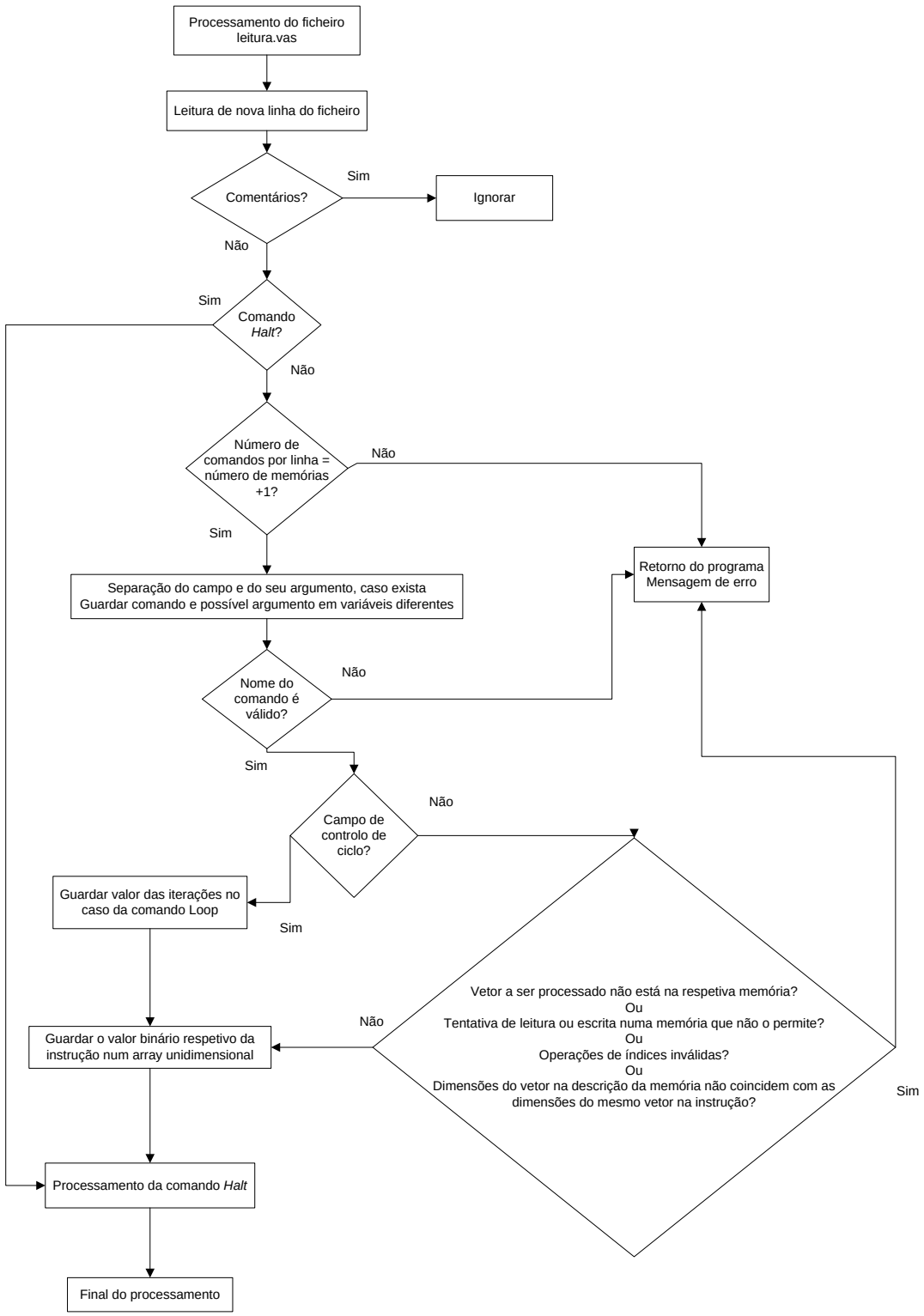

Figura 4.8: Interpretação do ficheiro *instrucoes.vas* pelo micro-assemblador.

alocados aos portos de memória são armazenados num *array* de caracteres, em que cada linha contém um vetor e seus valores associados.

O micro-assemblador repete este processamento para todos os portos de memória descritos no ficheiro, até ao seu fim.

Depois de processar o ficheiro de descrição das memórias o micro-assemblador interpreta o ficheiro das configurações do caminho de dados. A única ação que o assemblador realiza com o conteúdo deste ficheiro é usar a codificação binária do caminho de dados. Mediante a opção escolhida em cada linha do ficheiro *instrucoes.vas*, a combinação binária respetiva é acrescentada no micro-código gerado. Assim, o processamento do ficheiro *datapath_conf.vas* por parte do micro-assemblador está patente na Figura 4.9.

Por fim, o último ficheiro traduzido pelo micro-assemblador é o ficheiro *instrucoes.vas*. O processamento deste ficheiro está ilustrado na Figura 4.8.

Tal como no ficheiro *mem_details.vas*, o micro-assemblador lê o ficheiro das instruções linha a linha e começa por ignorar possíveis comentários existentes. Com apenas o bruto das instruções para processar, o micro-assemblador verifica se se trata do campo da instrução *Halt* que simboliza final de programa. Em caso afirmativo, interpreta esse comando e finaliza o processo de tradução; caso contrário, traduz a sequência de campos da linha para gerar a totalidade de *bits* dessa instrução.

O processamento dos campos de uma instrução, à exceção do comando *Halt*, começa por confirmar a condição do número de portos de memória. Como explicado na secção 3.2.2, o número de campos de uma instrução tem de ser igual ao número de portos de memória + 1. O micro-assemblador averigua esta situação, com o processo de tradução a decorrer normalmente apenas com a condição verificada. Em seguida, antes de prosseguir para a interpretação propriamente dita, o *software* confirma ainda a inexistência de campos das instruções inválidos.

O tratamento conferido aos comandos das instruções difere entre os campos de controlo de ciclos e os restantes. Este grupo de comandos permite uma tradução direta da sua representação textual para código binário. Nos restantes campos, o comando *Nop* segue a mesma lógica de processamento. Todos os outros campos ainda não referidos exigem um tratamento especial porque têm argumentos que especificam operações sobre os vetores alocados aos portos de memória.

Desta forma, a interpretação dos campos *Sel*, *Set*, *Read* e *Write* inicia-se com a separação do comando e respetivo argumento. A tradução de todos estes comandos requer, em primeiro lugar, a verificação da existência do vetor do argumento em memória. Somente em caso afirmativo é que o micro-assemblador traduz o comando e argumento inerente. Os campos *Set*, *Read* e *Write* têm de satisfazer restrições adicionais devido a operarem sobre os índices dos vetores alocados em memória; as operações dos índices são restritas ao conjunto implementado e o número de índices apresentado tem de coincidir com as dimensões reais dos vetores. Em último lugar, os comandos

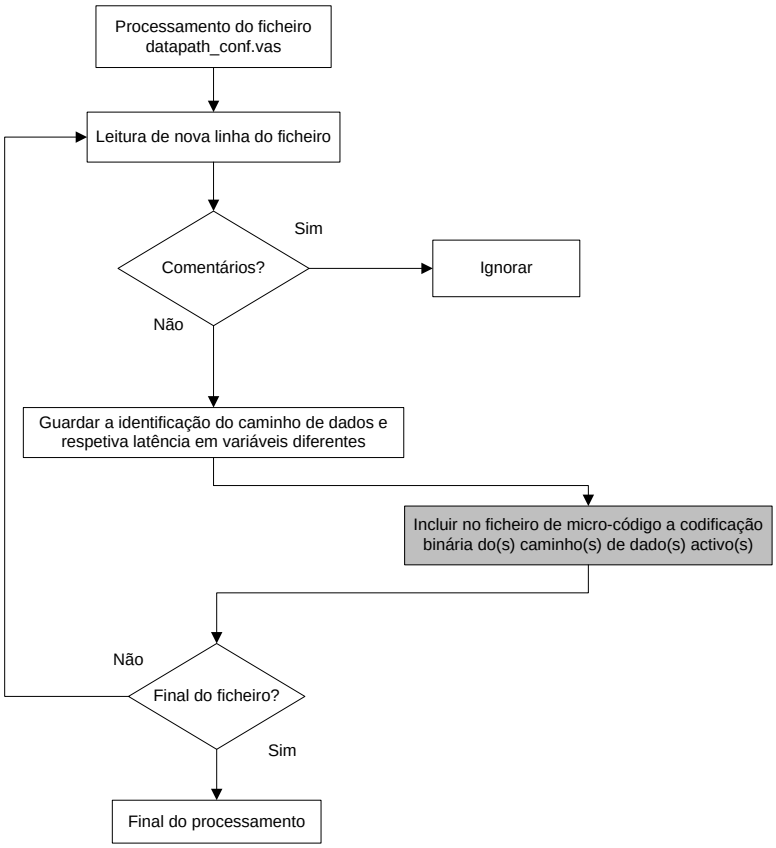

Figura 4.9: Processamento do ficheiro *datapath_conf.vas* pelas ferramentas de *software*. A ação da caixa a sombreado é relativa apenas ao *software* assemblador, não sendo executada pelo gerador da arquitetura.

Read e *Write* têm de respeitar as restrições do modo de funcionamento da memória; nomeadamente, é impossível fazer a operação de leitura numa memória apenas de escrita e vice-versa.

Com toda a estrutura anterior implementada, e na ausência das situações anormais referidas, o micro-assemblador entra na fase final do processamento. Nesta etapa, concretiza a interpretação com a tradução dos campos das instruções e argumentos segundo as combinações binárias vistas na secção 3.2.2.

Um exemplo desta tradução está patente na Figura 4.10. Como se pode verificar, cada campo da instrução é traduzido através de uma divisão das suas diversas partes constituintes. No caso do campo de controlo de ciclos a sua tradução é direta, com os 2 *bits* que resultam desse processo a integrar a instrução como os 2 *bits* mais significativos. Os restantes campos envolvem a interpretação das várias partes, que em conjunto, formam a representação binária do campo.

Considerando o campo *Read* da Figura 4.10, verifica-se que a codificação binária do campo é definida em função dos seus argumentos e do seu significado intrínseco. Assim, o campo *Read*

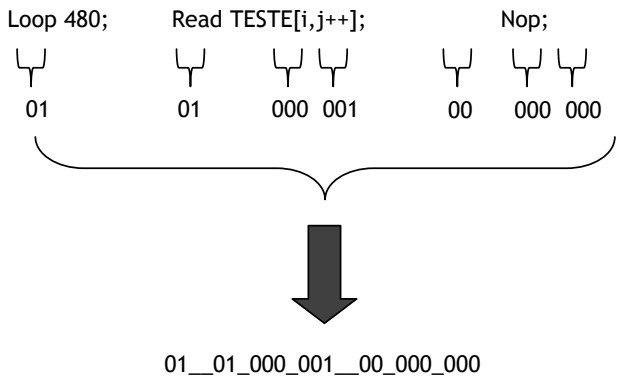

Figura 4.10: Exemplo do processo de tradução do micro-assemblador.

pressupõe uma leitura e é codificado com o valor binário 01. Os seus argumentos *i* e *j++* são traduzidos para os valores binários 000 e 001, respetivamente. Por último, o comando *Nop* segue a mesma abordagem descrita para o campo *Read*. O facto de representar um comando sem operação em vetores e não possuir argumentos leva a que todos os 8 *bits* do campo em questão sejam codificados com o valor binário 0.

À medida que cada instrução é codificada, o resultado é guardado num *array* de caracteres que é impresso no ficheiro de escrita que contém o código binário; no caso exemplo anterior o valor armazenado seria 010100000100000000. Este armazenamento é feito linha a linha, com cada linha de código binário a representar a respetiva linha de instrução. Finalmente, também o campo *Halt* é traduzido e adicionado ao restante micro-código. Este processamento consiste na escrita do número de valores binários 1 correspondente ao número de *bits* das instruções.

4.4.2 Funcionalidades Secundárias

Para além das funcionalidades básicas para interpretação dos ficheiros e geração do micro-código, foram desenvolvidas outras funcionalidades. Estas visam aumentar a robustez e compreensão do *software* criado e ficheiros usados.

Um dos exemplos é a introdução de comentários nos ficheiros de descrição, para além da representação dos conteúdos mencionados. Este tipo de texto visa apoiar e explicitar toda a estrutura criada, embora não seja parte necessária da descrição da arquitetura. Um dos objetivos é tornar a linguagem simbólica mais compreensível. Por outro lado, a sua utilização prevê maior facilidade para testar diferentes alternativas; por exemplo, no caso do utilizador querer utilizar 2 comandos diferentes para ver o seu efeito, pode comentar um deles e testar o outro. O processo recíproco é rápido e traz vantagens quando as diferenças entre comandos são grandes e a sua mudança exige muito tempo.

Os caracteres escolhidos para a implementação dos comentários são os mesmos do que na linguagem C, com a adição do caracter #. A função dos caracteres # e // é comentar linhas completas. A sequência /**/ pode ser usada para o mesmo fim ou comentar apenas pequenos excertos de código.

No ficheiro *mem_details.vas* foi implementado um meio de evitar a escrita do código todo na mesma linha. No caso em particular deste ficheiro, em que a descrição dos vetores alocados às memórias pode ser bastante extensa, justifica-se a inclusão desta delimitação. Com recurso à sequência de carateres \\, é dada a indicação às ferramentas de *software* que a descrição do porto de memória continua na linha seguinte. Assim, quando o ficheiro é processado o programa que o interpreta sabe que a linha de código simbólico continua na linha seguinte do ficheiro. Este assume que a descrição da linha chega ao fim quando não surge a sequência \\.

De igual forma, foram também implementados alguns argumentos opcionais. Estes argumentos podem ser usados em conjunto com os argumentos obrigatórios, na execução do micro-assemblador. O grupo de argumentos nestas condições está descrito na Tabela 4.4.

O argumento *-help*, quando invocado, imprime um menu de ajuda no ecrã. Na situação em que este argumento é chamado simultaneamente com os argumentos obrigatórios é-lhe dada prioridade. Por outras palavras, quando o *-help* é usado o micro-assemblador imprime sempre o menu de ajuda e suspende todo o processamento usual. O menu de ajuda contém uma descrição dos argumentos possíveis de executar bem como dos campos das instruções e respetivas operações sobre os índices. O menu referido encontra-se ilustrado na Figura A.2.

Por seu turno, o argumento *-comment* seleciona a geração do ficheiro de micro-código binário com comentários e separações. Por omissão, o micro-código é gerado e incluído no ficheiro de escrita sem comentários, apenas com valores 0 e 1. Recorrendo a este argumento, o ficheiro de saída toma a forma da Figura A.1. Como pode ser visto na figura em causa, são adicionados comentários que identificam cada linha e a sequência de campos da instrução que originam o código binário. Do mesmo modo, são inseridas separações entre os grupos de *bits* de cada comando, o que ajuda à compreensão do código obtido.

Finalmente, o argumento *-vNUM* regula o nível de texto impresso no momento da execução do micro-assemblador. O controlo da verbosidade é conseguido através de valores atribuídos ao parâmetro NUM. Este parâmetro pode tomar valores de 1 a 3, com ordem crescente do nível de texto. Com o argumento em nível 1 são impressas no ecrã apenas mensagens de carácter geral; estas indicações dão uma ideia das fases que o micro-assemblador segue no processo de tradução mas não o que faz em concreto.

Caso seja usada a opção -v3, o nível do texto é máximo e todas as ações efetuadas pelo *software*

Tabela 4.4: Argumentos facultativos do micro-assemblador.

Argumento	Função
-help	Imprimir um menu de ajuda
-comment	Permitir a geração do micro-código com comentários e separações
-vNUM	Regular o nível de comentários na execução do programa, em função do valor NUM

são listadas; com esta alternativa é possível verificar em pormenor o que o micro-assemblador vai fazendo, como por exemplo as características das memórias que vai lendo ou quais os campos das instruções que vai processando. Nas Figuras A.3 e A.4 podem ser observados os efeitos dos argumentos -v1 e -v3, respetivamente.

4.5 Geração da Arquitetura

Foi criada uma plataforma de *software* idêntica à do micro-assemblador com capacidade de proporcionar esta nova função. Nomeadamente, apesar de se tratar de um programa independente do micro-assemblador, a sua estrutura de leitura dos ficheiros descritivos é, em traços gerais, a mesma. De igual forma, esta ferramenta foi gerada com recurso a linguagem C e compilada com o Cygwin. O motivo da escolha da divisão dos 2 programas está relacionado com a função principal de ambos e com a extensão do código C que os implementa. Juntar ambos implicaria formar um programa bastante confuso, extenso e com algumas funcionalidades em comum mas com 2 funções principais diferentes. Assim surgiu a separação entre o código do micro-assemblador e o código do gerador da arquitetura vetorial.

Com o *software* do micro-assemblador a realizar o processo de tradução do micro-código, o segundo grande objetivo é gerar a arquitetura do processador. Este processo de geração tem de permitir a configuração da arquitetura segundo os parâmetros definidos nos ficheiros de descrição. Uma vez que os módulos que concretizam a arquitetura já se encontram parcialmente desenvolvidos, a ideia passa pela sua correta parametrização e ligação entre si.

Como referido na secção 3.2.1, a linguagem usada para a descrição do *hardware* da arquitetura foi *Verilog*. Esta decisão teve como base a arquitetura previamente desenvolvida na mesma linguagem. Atendendo às suas características, a forma encontrada de configurar a arquitetura foi o uso conjunto das diretivas *defparam* e *parameter*. Esta última permite a definição de parâmetros

intrínsecos a um módulo *Verilog*. Por seu lado, a diretiva *defparam* possibilita definir um parâmetro de um módulo instanciado dentro de outro módulo; neste caso o parâmetro inicialmente definido é substituído pelo valor definido com a diretiva *defparam*.

Existe, contudo, uma situação que estas diretivas por si só não conseguem solucionar. A diretiva *defparam* não permite definir parâmetros vetoriais, isto é, que sejam compostos por vários parâmetros escalares. Por outro lado, também não permite definir parâmetros de sub-módulos de níveis sucessivamente inferiores. Desta maneira, é impossível definir os valores dos parâmetros dos vetores alocados às memórias a partir do módulo de nível superior. A alternativa encontrada foi definir cada um desses parâmetros com a diretiva *parameter* e fazer a substituição individual com a *defparam*. Para evitar a escrita de um código demasiado extenso foi usada a diretiva *'include*; esta possibilita a execução na íntegra de uma porção de código incluída noutro ficheiro. A ideia é definir todos os parâmetros de todos os vetores com a diretiva *defparam* num ficheiro à parte e incluir no módulo de topo da arquitetura.

Por conseguinte, o objetivo desta parte do *software* é a geração do módulo de topo da arquitetura e respetiva bancada de teste, com todos os sub-módulos devidos e corretamente interligados. O programa gerador tem de ser igualmente capaz de criar o ficheiro com os parâmetros dos vetores, para ser incluído posteriormente.

A geração da arquitetura vetorial dedicada é feita com recurso a um programa semelhante ao micro-assemblador. A estrutura e rotinas de ambos são bastante parecidas, dado que ambos têm como função interpretar os ficheiros de descrição da arquitetura. Existem, contudo, algumas diferenças porque o objetivo principal de ambos é distinto. Assim, a abordagem considerada foi adaptar a estrutura do micro-assemblador de forma a conseguir a geração da arquitetura.

Como tal, a forma como o programa gerador é executado é em tudo análoga à do micro-assemblador. A sequência de argumentos para o executar é dada por:

```
./gerador -r -m mem_details.vas -r -d datapath_conf.vas -r -c instrucoes.vas -w
    toplevel.v -w toplevel_tb.v -w def_data_parameters.v
```

Como pode ser observado no conjunto de argumentos anterior, uma das diferenças existentes é o facto do programa de geração escrever em 3 ficheiros diferentes, que irão conter os códigos *Verilog* do ficheiro de topo da arquitetura, da bancada de teste e da definição dos parâmetros dos vetores em memória.

Em geral, o ficheiro *mem_details.vas*, é lido e processado da mesma forma por ambos micro-assemblador e gerador no que toca à descrição dos portos de memória e vetores alocados. A única exceção é a verificação dos campos dessa descrição por parte do gerador da arquitetura; são verificadas as codificações dos campos (se correspondem apenas e só a combinações válidas) e o número de campos especificado em função do tipo de memória (por exemplo, uma memória do tipo *scratch* deve ter exatamente 2 campos: o tipo de memória e o tamanho dos blocos de

memória). Qualquer imprecisão detetada é tratada como uma ocorrência de erro, com a impressão de um aviso e paragem de execução do *software* gerador.

Adicionalmente, o programa processa ainda a parte do ficheiro que descreve as entradas e saídas do caminho de dados.

Toda esta porção de processamento corresponde à secção direita do fluxograma da Figura 4.7. Desta maneira, todo o processamento do ficheiro *mem_details.vas* feito pelo micro-assemblador é também feito pelo gerador, sendo que este último executa ainda as funcionalidades acima.

O *software* gerador verifica um grupo de restrições de forma a garantir que não há situações de erro na descrição das entradas e saídas. Na inexistência deste tipo de estado, o gerador guarda a indicação das entradas, saídas e respetivas memórias a que estão alocadas em variáveis. Na altura de escrever o código *Verilog* dos ficheiros de saída são lidos os dados destas variáveis.

Em seguida, o programa gerador interpreta o ficheiro *datapath_conf.vas*. Esse processo está detalhado na Figura 4.9. Como pode ser confirmado pelo conteúdo da mesma figura, o processamento deste ficheiro é bastante simples. O gerador limita-se a ler o ficheiro linha a linha, ignorando os comentários, e a armazenar a informação importante sobre o nome do caminho de dados e sua latência em variáveis. Deste modo, não realiza a operação da inclusão da codificação do(s) caminho(s) de dado(s) no código binário. Esta acção está em realce na Figura 4.9 e é exclusiva do *software* assemblador.

O último ficheiro lido pelo *software* gerador é o ficheiro com as instruções a executar. As rotinas usadas para ler o ficheiro são iguais às descritas para o caso do micro-assemblador. No entanto, no caso da geração da arquitetura, a única informação que o programa retém é o número de iterações de possíveis ciclos a implementar.

O gerador verifica ainda se o número de campos de uma instrução é igual ao número de portos de memória + 1, abortando a sua execução em caso de erro.

Com todos os dados que necessita para a geração e parametrização da arquitetura, o programa gerador inicia a escrita do código *Verilog* dos ficheiros de saída. A geração do código destes módulos é feita de acordo com o código *Verilog* dos módulos e estrutura da arquitetura apresentados no capítulo 3.

Recorrendo à estrutura dos módulos já existentes, e em função dos resultados do processamento dos 3 ficheiros descritores armazenados nas variáveis, o gerador escreve o código dos módulos de topo da arquitetura e da bancada de teste. Este processo envolve a instanciação dos módulos da arquitetura, suas ligações e a declaração de todas as saídas, entradas, fios e registos[1]. A escrita do código destes ficheiros é completa com a parametrização correta dos módulos, recorrendo às diretivas mencionadas anteriormente. Assim, consegue-se a configuração e flexibilidade pretendidas para arquitetura vetorial.

[1] Em linguagem *Verilog*, sinais do tipo output, input, wire e reg.

No caso de haver portos de memória associados a memórias *cache* é adicionado um modelo *Verilog* que implementa o comportamento da memória externa. Este modelo de funcionamento é incluído na bancada de teste para representar a troca de dados entre a memória *cache* e a memória externa. Neste trabalho houve uma restrição que limitou o uso de apenas uma memória deste tipo. A utilização de várias memórias *cache* exige um módulo que controle o acesso à memória externa, a fim de evitar acessos simultâneos. Uma vez que este módulo não se encontra desenvolvido foi necessário atender a esta limitação.

Finalmente, a parte de geração é completa com a escrita do ficheiro que define os parâmetros dos vetores em memória. Este ficheiro será adicionado ao módulo de topo da arquitetura com a diretiva *'include*.

4.6 Aplicação Exemplo - Sobel

O algoritmo Sobel tem como objetivo detetar os limites de uma imagem. Este algoritmo assume especial importância no domínio de processamento de imagem. Nesta área é crucial que as imagens possibilitem dados que as identifiquem de forma fácil e inequívoca. Assim, o algoritmo Sobel ajuda na diferenciação dos conteúdos de imagem, nomeadamente a nível de cores e fronteiras com outros conteúdos. Por outro lado, permite retirar alguma informação irrelevante da imagem, mantendo contudo o conteúdo importante que permite a sua identificação [31].

O algoritmo Sobel baseia-se no cálculo do gradiente segundo as direções x e y. Para tal realiza a convolução da imagem a processar com as máscaras, representadas nas seguintes matrizes:

$$T_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad T_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Em seguida, é calculado o módulo do gradiente total, segundo a equação $g(x,y) = \sqrt{T_x^2 + T_y^2}$. Posteriormente, o valor do módulo do gradiente é comparado com um valor limiar, que funciona como decisor. Este valor tem de ser bem escolhido porque caso seja muito alto os limites podem não ser detetados. Na situação do valor limiar ser muito baixo podem ser encontradas arestas falsas e a representação da imagem processada não corresponder à realidade.

Na Figura 4.11 encontra-se ilustrado um exemplo da execução deste algoritmo. A imagem da esquerda representa a imagem inicial, ao passo que a imagem da direita encerra os limites detetados com o algoritmo [31].

Este algoritmo é o caso de estudo exemplo no contexto desta arquitetura. O caminho de dados do processador implementa o algoritmo referido. Ao longo do trabalho realizado no decorrer desta Dissertação foi usado este caminho de dados como exemplo de concretização da arquitetura.

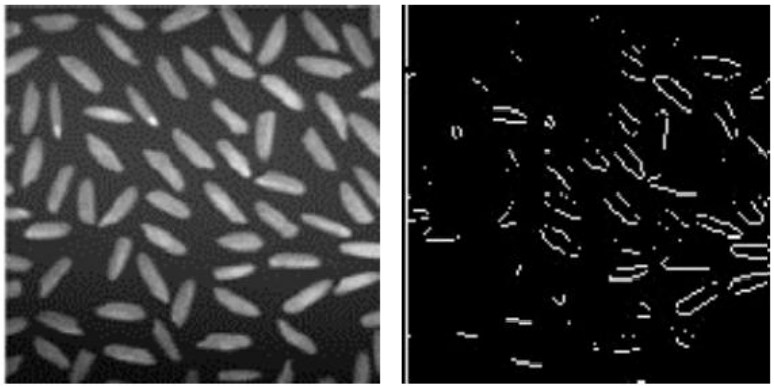

Figura 4.11: Exemplo de aplicação do algoritmo Sobel [31].

4.7 Resumo

Neste capítulo foram referidos todos os aspetos relacionados com a implementação das ferramentas de *software* deste trabalho. Em primeiro lugar, surge o desenvolvimento de um formato que suporte a descrição da arquitetura. Este modelo foi conseguido com uma linguagem simbólica que traduz as características passíveis de ser parametrizadas na arquitetura.

Em seguida, foram relatados os 2 *softwares* criados para ler e interpretar os ficheiros descritores. Ambos possuem uma estrutura semelhante, mas objetivos principais diferentes. Por um lado, o micro-assemblador tem como meta a tradução de código de alto nível para micro-código binário.

Por seu turno, o gerador da arquitetura tem a missão de gerar a arquitetura dedicada, com os seus componentes ligados e configurados segundo as indicações fornecidas nos ficheiros descritores. Esta ferramenta deve ser ainda capaz de gerar uma bancada de teste que permita a simulação do funcionamento da arquitetura de acordo com os parâmetros exigidos.

Finalmente, foi apresentado o algoritmo Sobel, um dos casos de estudo em torno desta arquitetura e uma das aplicações passíveis de ser implementada pelo caminho de dados.

Capítulo 5

Resultados e Testes

5.1 Introdução

Neste capítulo serão exibidos os resultados obtidos com as ferramentas de *software* desenvolvidas e os ensaios realizados em torno das mesmas. Da mesma forma, serão apresentadas as condições em que os resultados foram conseguidos. Esta apresentação será completa com uma breve crítica acerca dos resultados referidos.

Na secção 5.2 são descritos os resultados alcançados com o micro-assemblador no processo de tradução do micro-código. Esta secção mostra ainda excertos dos módulos *Verilog* obtidos com o gerador da arquitetura dedicada.

Por último, na secção 5.3 estão explicitados os resultados obtidos com a simulação realizada para validar os modelos de processadores gerados.

5.2 Assemblagem e Geração da Arquitetura

Como referido anteriormente, a meta da fase de assemblagem passou pela tradução do código simbólico de alto nível em micro-código binário. Com o intuito de verificar essa funcionalidade, numa primeira fase foram usadas algumas combinações aleatórias de campos de instruções. Os ensaios realizados não representam situações reais e úteis. O objetivo passou sim por testar a grande maioria de possibilidades de comandos de instruções e respetivos argumentos, principalmente os argumentos de iteração sobre índices de vetores.

Um dos exemplos experimentados foi a configuração definida pelos ficheiros de descrição da Figura 5.1. Como pode ser observado, a arquitetura ensaiada conta com 4 portos de memória e permutações entre diferentes caminhos de dados.

```
                    /* FICHEIRO mem_details.vas */

MEM0: TESTE[2,2]:1:100:2:1:2; B[10,50]:200:300:10:5:10; \\
      C[5,3]:400:500:1:1:1;   K[5,3]:550:650:1:1:1;     cachenwa; 2; 8; 8; RW;
MEM1: DF[100]:700:800:10:10;  EXP[34,7]:200:400:3:8:8;  scratch; 1024; RO;
MEM2: H[100]:500:700:1:1;                               cachedm; 32; 256; RW;
MEM3: G[50,50]:800:1000:2:4:2;                          scratch; 256; WO;
output: dout;  MEM3;
output: dout2; MEM2;
input:  din:MEM0;
input:  din2:MEM0;
input:  din3:MEM1;
```

```
                    /* FICHEIRO instrucoes.vas */

0: Noloop;  Sel TESTE;  Sel EXP;      Sel H;              Sel G;              SOBEL;
1: Loop 10; Nop;        Set EXP[i--,j--]; Set H[i];       Set G[i=MAX,j=0];   MUL;
2: Next;    Read TESTE[i--,j=MAX];    Read EXP[i+S,j++]; Nop; Nop;           SOBEL;
3: Loop 20; Nop;        Nop;          Write H[i++];       Nop;                SOBEL;
4: Next;    Nop;        Set DF[i-S];  Nop;                Write G[i,j++];     DIV;
5: Halt;
```

```
                    /* FICHEIRO datapath_conf.vas */

SOBEL;  00;   5;
MUL;    01;   6;
DIV;    10;   4;
ADD;    11;   8;
```

Figura 5.1: Ficheiros de descrição da arquitetura para teste do processo de assemblagem.

O código binário gerado nestas condições está representado na Figura 5.2. Pelos resultados evidenciados, é possível aferir o perfeito funcionamento do processo de tradução do micro-código binário. O *software* não encontrou nenhum erro durante a assemblagem e as codificações binárias dos caminhos de dados e campos das intruções coincidem com o esperado. Foram ainda feitos mais testes, segundo a mesma lógica de usar várias combinações possíveis, que corroboram estas afirmações.

O passo seguinte consistiu na geração de uma arquitetura que obedece às configurações anteriores. Uma vez mais o objetivo não foi propriamente alcançar um resultado concreto mas antes verificar o processo de compilação da arquitetura.

Desta forma, os ficheiros de descrição foram posteriormente processados pela ferramenta de *software* de geração. Em seguida são apresentados fragmentos obtidos dos ficheiros gerados. O código *Verilog* dos ficheiros não foi incluído na íntegra devido à sua extensão. Do mesmo modo, nem todos os aspetos do código *Verilog* são analisados com este exemplo para evitar redundância entre os exemplos considerados. Assim, os conteúdos mais importantes dos resultados alcançados ao nível da geração da arquitetura são mostrados de maneira repartida; uma parte é analisada no

```
00000011000000110000001100000011000
00000000010010000000000010111000010 1
11001001001011010000000000000001100
00000000000000000000011000000000100
00000000000100000000000001000101110
11111111111111111111111111111111111
```

Figura 5.2: Micro-código binário para teste gerado pelo compilador.

contexto desta situação e outra parte refere-se à aplicação Sobel, cujos resultados são apresentados mais à frente nesta secção.

Recorrendo aos ficheiros de descrição da Figura 5.1, foram testadas as primeiras configurações da arquitetura no que respeita à sua geração. Na Figura 5.3 está patente uma parte dos resultados verificados.

```
defparam addrgen_0.strIdimJ_0=19'd3;
defparam addrgen_0.endAddr_0=19'd650;
defparam addrgen_0.strDimJ_0=4'd1;
defparam addrgen_0.strDimI_0=4'd1;
defparam addrgen_0.DIM_J_0=11'd3;
defparam addrgen_0.DIM_I_0=11'd5;
defparam addrgen_0.DIM_S_0=2'd1;
defparam addrgen_0.addrBase_0=19'd550;
```

```
defparam addrgen_1.strIdimJ=19'd0;
defparam addrgen_1.endAddr=19'd800;
defparam addrgen_1.strDimJ=4'd0;
defparam addrgen_1.strDimI=4'd10;
defparam addrgen_1.DIM_J=11'd0;
defparam addrgen_1.DIM_I=11'd100;
defparam addrgen_1.DIM_S=2'd10;
defparam addrgen_1.addrBase=19'd700;
```

```
defparam assoc4_cache_0.N_SETS = 8;
defparam assoc4_cache_0.N_WAYS = 2;
defparam assoc4_cache_0.N_CACHE_LINES = 16;
defparam assoc4_cache_0.N_BLOCKS_PER_LINE = 8;
defparam assoc4_cache_0.N_BITS_WAYS = 1;
defparam assoc4_cache_0.N_BITS_SET = 3;

defparam datamemory_0.BLOCK_SIZE = 1024;
defparam datamemory_0.mem_filename = "mem0.hex";

defparam dm_cache_0.N_CACHE_LINES = 32;
defparam dm_cache_0.N_BLOCKS_PER_LINE = 256;
```

```
datamemory_1(
        .clock( clock ),
        .address( address3 ),
        .wr( wr3 ),
        .rd( 1'b0 ),
        .din( mem_din3 ),
        .dout( mem_dout3 )
);
```

Figura 5.3: Exemplos de código *Verilog* obtidos com o *software* de geração, com recurso às configurações da Figura 5.1.

A primeira parte do código *Verilog* apresentado corresponde a um excerto do ficheiro que define os parâmetros dos vetores alocados em memória. Estes estão definidos no módulo gerador de endereços e são substituídos no momento da assemblação da arquitetura pelos parâmetros requisitados. A operação de substituição é levada a cabo pela diretiva *defparam*. No primeiro fragmento da Figura 5.3 encontram-se as parametrizações dos vetores K e DF dos portos de memória MEM0 e MEM1, respetivamente. Os valores gerados correspondem aos valores definidos no ficheiro de configuração; de salientar que no caso de um vetor unidimensional, como é o caso do vetor DF, os valores respeitantes à dimensão *j* são preenchidos com 0.

A segunda porção apresentada na Figura 5.3 é relativa à parametrização dos blocos de memória da arquitetura. Como se pode observar na figura, este processo é uma vez mais concretizado recorrendo à diretiva *defparam*. Os portos de memória incluídos neste excerto são, ordenadamente, o porto MEM0, MEM1 e MEM2. O processo de geração traduz as designações codificadas das memórias para transformar cada bloco numa instanciação do módulo respetivo. Em seguida a diretiva *defparam* trata de escrever o valor correto para cada parâmetro de cada bloco de memória inserido na arquitetura.

É ainda possível definir o ficheiro que inicializa os blocos de memória; o gerador usa uma designação por omissão que define um valor simbólico. Esta deve ser substituída pelo nome correto do ficheiro caso se queira inicializar a memória com um conjunto específico de dados. Assim, todos os parâmetros assumem os valores esperados para o processo de geração da arquitetura.

Por último, a terceira secção do código *Verilog* da Figura 5.3 mostra a interface e instanciação de um bloco de memória do tipo *scratch*. Este é um dos exemplos dos módulos que o programa gerador cria e interliga com os outros módulos da arquitetura. Através do código exemplo podemos ver que este módulo é controlado por um endereço e sinais de leitura e escrita que regulam o seu acesso; o mesmo se passa com os blocos de memória *cache*. Os sinais de leitura e escrita são controlados pelo gerador de endereços afeto a esse bloco de memória. No entanto, uma das indicações que o *software* gerador introduz é ativar ou inibir as ações de leitura ou escrita. Este processo é realizado em função do modo de funcionamento da memória detalhado no ficheiro de descrição das memórias. Por exemplo, o bloco de memória da Figura 5.3 está ligado ao porto de memória MEM3, que é de escrita apenas. Desta forma, o sinal de leitura é ligado permanentemente ao valor lógico 0, o que impossibilita essa ação.

Todas as configurações e caraterísticas anteriores, com exceção do primeiro fragmento de código, são parte integrante do ficheiro que contém a bancada de teste gerada. A análise destes pontos leva à verificação da operação correta do programa gerador. Tal como no caso do processo de tradução do micro-código, foram efetuados outros testes com combinações diferentes dos ficheiros de configuração. Esses ensaios seguem igualmente os resultados esperados.

Com as ferramentas de *software* verificadas por várias situações distintas, o passo seguinte foi aplicar o processo de compilação e assemblagem completo ao exemplo do algoritmo Sobel, explorado na secção 4.6. Os ficheiros de descrição usados para a implementação deste algoritmo estão ilustrados na Figura 5.4.

O ficheiro *mem_details.vas* descreve uma arquitetura que conta com 2 portos de memória distintos. Ambos estão associados a memórias locais, em que uma é memória de leitura apenas e a outra de escrita apenas. Desta forma, a ideia é uma configuração da arquitetura onde os dados são lidos de uma memória, tratados pelo processador e os resultados escritos na outra memória. Por conseguinte, o caminho de dados apresenta uma entrada que liga ao porto de memória de leitura, MEM0, e uma saída que liga ao porto de escrita, MEM1.

As instruções de código simbólico do ficheiro *instrucoes.vas* são o resultado dum processo de tradução manual, previamente realizado. O ponto de partida foi uma rotina escrita em linguagem C que implementa o algoritmo Sobel, posteriormente assemblada no código simbólico [29]. A sequência de instruções referida implica operações sobre um conjunto de valores que representam os píxeis de uma imagem. Neste caso, uma matriz de 480 valores em comprimento e 640 em largura é processada segundo as máscaras definidas para o algoritmo considerado. Deste modo, o único caminho de dados usado nesta implementação foi o caminho de dados que processa o algoritmo Sobel.

```
                        /* FICHEIRO mem_details.vas */

MEM0: A[480,640]:10:307199:2:2:1; B[480,640]:40:307199:2:2:1; scratch; 1024; RO;
MEM1: A[480,640]:10:307199:2:2:1; B[480,640]:40:307199:2:2:1; scratch; 1024; WO;
output: dout; MEM1;
input:  din:MEM0;
```

```
                        /* FICHEIRO instrucoes.vas */

0: Noloop;     Sel A;              Sel B;             SOBEL;
1: Noloop;     Set A[i--,j];       Nop;               SOBEL;
2: Loop 480;   Set A[i++,j=0];     Set B[i++,j--];    SOBEL;
3: Loop 640;   Read A[i++,j];      Nop;               SOBEL;
4: Noloop;     Read A[i++,j];      Nop;               SOBEL;
5: Next;       Read A[i-S,j++];    Write B[i,j++];    SOBEL;
6: Next;       Nop;                Set B[i,j=0];      SOBEL;
7: Halt;
```

```
                        /* FICHEIRO datapath_conf.vas */

SOBEL;  00;   5;
```

Figura 5.4: Ficheiros de descrição da arquitetura para implementação e teste do algoritmo Sobel. Ficheiro *instrucoes.vas* adaptado de [29].

```
00000011000001110000
00001000000000000000
10100100010001000100
00000101000000000100
00000101000000000000
00110001001000101100
00000000101000001100
11111111111111111111
```

Figura 5.5: Micro-código binário do algoritmo Sobel gerado pelo micro-assemblador.

Assim, em primeiro lugar, foi aplicado o processo de assemblagem aos ficheiros da Figura 5.4. O processamento por parte do *software* assemblador realizou-se sem erros e o resultado encontra-se demonstrado na Figura 5.5. Este apresenta, como os exemplos anteriores, os resultados aguardados. O ficheiro resultante da assemblagem, que contém o micro-código binário, é usado posteriormente para carregar a memória que alberga as instruções a executar.

Após a obtenção de resultados na etapa de assemblagem, os ficheiros de configuração da Figura 5.4 foram usados para gerar uma arquitetura dedicada para aplicação do algoritmo Sobel. Os ficheiros com código *Verilog* obtidos com este processo são configurados da forma vista para o exemplo anterior. Tal como nesses casos, existem ainda outros pedaços de código que ajudam a parametrizar e definir a arquitetura. Os excertos em questão estão indicados na Figura 5.6.

```
toplevel tl(
        .clock( clock ),
        .reset( reset ),
        .rd0( rd0 ),
        .wr0( wr0 ),
        .address0( address0 ),
        .rd1( rd1 ),
        .wr1( wr1 ),
        .address1( address1 ),
        .en( 1'b1 ),
        .din( mem_dout0 ),
        .dout( mem_din1 ),
        .halt( halt )
);
```

```
parameter DP_LATENCY = 5;
defparam codememory_1.PROGRAM_FILE = "code.bin";
```

Figura 5.6: Exemplos de código *Verilog* obtidos com o *software* de geração, com recurso às configurações da Figura 5.4 para implementação do algoritmo Sobel.

Na primeira secção de código apresentado na Figura 5.6 encontra-se a instanciação do módulo de nível superior da arquitetura. A declaração do módulo citado é feita no ficheiro de bancada de teste *Verilog*. Na sequência da instanciação do módulo de topo são também definidos os sinais que realizam a interface com os blocos de memória. Assim, são ligados os sinais de leitura, escrita e endereço que conduzem o acesso às memórias.

De igual modo, é definido um sinal de ativação ou inibição que comanda a execução das instruções e, por conseguinte, a operação de todo o processador. A existência deste sinal é devida à latência de leitura de dados das memórias. No caso das memórias *scratch* a latência de leitura é sempre de um ciclo de relógio; contudo, no caso das memórias *cache* podem ocorrer períodos de interação com a memória externa, onde os dados não estão disponíveis. Como tal, o sinal de permissão é permanentemente ativo quando são usadas apenas memórias locais e ligado a um sinal que simboliza a situação em que a memória está pronta ou não, no caso de utilização de memórias *cache*. O sinal referido é o sinal *en* representado na interface do módulo *toplevel*.

Finalmente são definidas nesta interface os barramentos de dados de entrada e saída do processador, que coincidem com as entradas e saídas do caminho de dados da arquitetura.

Por outro lado, na segunda parte do código são mostradas as parametrizações feitas no ficheiro de topo da arquitetura. Ao nível deste ficheiro são feitas configurações para moldar o caminho de dados à aplicação a executar. Nomeadamente, é neste ponto em que é definida a latência inserida pelo caminho de dados. Por fim, o parâmetro que contém o nome do ficheiro do micro-código binário é substituído pelo ficheiro gerado no processo de assemblagem. Desta forma, o módulo de controlo do micro-código carrega a memória com as instruções a executar para a operação desejada, neste caso o algoritmo Sobel.

Os resultados obtidos para as tarefas de assemblagem e geração da arquitetura no contexto do algoritmo Sobel foram os corretos. Uma vez mais, estes factos confirmam a correta implementação dessas funcionalidades por parte das ferramentas de *software* desenvolvidas.

Finalmente, foram ensaiadas algumas situações de erro deliberadas com o objetivo de verificar a robustez do micro-assemblador. Todas as possibilidades consideradas durante a etapa de implementação foram testadas com efeitos positivos. A título de exemplo, apenas serão consideradas 2 situações, a fim de evitar uma enumeração extensa de todos os testes efetuados. O critério de escolha destas situações foi a contribuição negativa que poderiam trazer caso não fossem devidamente tratadas. Os resultados alcançados nesta componente estão ilustrados no Anexo B.

5.3 Simulação

A fase final dos testes efetuados prendeu-se com a validação e simulação dos modelos de processadores gerados com as ferramentas de *software* implementadas.

A configuração considerada é conseguida através dos ficheiros da Figura 5.4. A arquitetura gerada desta forma conta com 2 portos de memória, um de leitura e outro de escrita, associados a memórias do tipo *scratch*. Esta implementação aplica o algoritmo Sobel, com os dados da memória de leitura como entrada do caminho de dados e os resultados a serem armazenados na memória de escrita.

A simulação efetuada, levada a cabo através do *software ModelSim*, teve como objetivo perceber o funcionamento da arquitectura nestas condições. A arquitetura lê os dados do porto de memória MEM0, processa-os segundo o algoritmo Sobel e coloca os resultados no porto de memória MEM1. O objetivo da simulação é implementar o algoritmo com recurso a ciclos *for* em *Verilog* e posteriormente comparar os resultados obtidos desta maneira com os resultados conseguidos com a implementação da arquitetura vetorial. Para melhor identificação desta comparação, é impressa uma matriz de valores binários, onde um 0 representa resultados diferentes e 1 resultados iguais.

Analisando a matriz binária resultante, verifica-se que os valores alcançados são os corretos, exceto os valores calculados nas margens. Estes desvios devem-se aos cálculos envolvidos no algoritmo Sobel; uma vez que cada cálculo implica o uso dos valores da iteração presente, passada e futura, os valores das margens são calculados erradamente por falta dos valores presentes e futuros nas primeiras e últimas iterações.

Deste modo, os resultados conseguidos coincidem com os resultados previstos e foi possível validar as ferramentas de *software* numa arquitetura que envolve memórias *scratch*.

A versão anterior do processador foi a única ensaiada, com uma configuração integrando apenas memórias *scratch*. Esta versão foi simulada com a bancada de teste previamente construída. Uma vez que foi usada exclusivamente essa bancada de teste, não foram testadas outras configurações de processadores. Da mesma forma, o único caminho de dados aplicado foi o algoritmo Sobel, pois a bancada de teste existente possibilita justamente a simulação desse caminho de dados.

5.4 Resumo

Ao longo deste capítulo foram apresentados os resultados obtidos no decorrer dos trabalhos desta Dissertação. Numa primeira fase, foram testados vários exemplos diferentes de descrição da arquitetura. Estes ficheiros foram alvo do processo de assemblagem e geração da arquitetura pelas ferramentas de *software*. Todos os testes efetuados neste sentido alcançaram com sucesso os resultados esperados.

Posteriormente, foi gerada uma configuração do processador para a implementação do algoritmo Sobel. Em seguida, a configuração foi alvo de simulação para a sua validação. Contando

com uma memória *scratch* de leitura e outra do mesmo tipo de escrita, a configuração ensaiada conseguiu obter os resultados aguardados para o algoritmo Sobel.

Capítulo 6

Conclusões e Trabalho Futuro

O conteúdo deste capítulo está relacionado com a análise crítica do desenrolar dos trabalhos desta Dissertação e respetivos resultados obtidos. Deste modo, aqui são apresentadas as conclusões retiradas dessa análise, ao longo da secção 6.1.

Na secção 6.2 são incluídas perspetivas de trabalho futuro, ou seja, sugestões para desenvolver e consolidar todo o fluxo de projeto em torno de uma arquitetura dedicada, em particular das ferramentas de *software* desenvolvidas.

6.1 Conclusões

O objetivo principal desta Dissertação foi o desenvolvimento de ferramentas de *software* que permitissem a geração de micro-código e respetiva arquitetura dedicada. Este objetivo foi alcançado na sua totalidade, sendo confirmado pelos resultados dos exemplos implementados.

Um outro objetivo consistiu na validação por simulação de diferentes configurações da arquitetura vetorial. Neste campo, foram obtidos resultados corretos em arquiteturas com memórias *scratch* apenas. A bancada de teste desenvolvida para a simulação de processadores com memórias *cache* não foi totalmente completa. Por esse motivo, não foram simuladas arquiteturas que recorrem a memórias *cache* e, portanto, não foram obtidos resultados conclusivos nesse sentido. Assim, a validação das arquiteturas geradas não foi completamente conseguida.

As ferramentas de *software* desenvolvidas revelaram-se bastante úteis, pois permitem a geração e configuração da arquitetura de forma automática e eficiente. Como tal, permitem poupar tempo e evitar erros relativamente ao processo de tradução manual. A implementação das ferramentas é um processo relativamente acessível mas ao mesmo tempo moroso porque é um conjunto de muitas rotinas pequenas. Nomeadamente, o desenvolvimento de ferramentas robustas e flexíveis exige muito trabalho e tempo gasto em testes para garantir o seu funcionamento perfeito. No

entanto, se este objetivo for alcançado, as ferramentas compensam o esforço dispendido pelo ganho em relação aos motivos já referidos.

A simulação das arquiteturas geradas é um processo que exige a compreensão dos módulos *Verilog* já desenvolvidos. Contudo, com base na bancada de teste previamente desenvolvida, é fácil adaptar o modelo de simulação a arquiteturas com recurso a apenas memórias *scratch*.

No caso de arquiteturas que incluem memórias *cache* a simulação revelou-se mais complicada. Uma das razões foi a adaptação da bancada de teste prévia para este tipo de situações em particular. Dado que a interface das memórias *cache* é diferente e envolve interação com a memória externa, torna-se menos evidente a fonte de erros na simulação. Outro dos motivos são as mudanças que a memória *cache* exige nos blocos da arquitetura devido às suas permissões de leitura e escrita, contrastando com a "disponibilidade" completa das memórias *scratch*.

Em suma, as ferramentas desenvolvidas trazem benefícios evidentes na tarefa de construção e configuração de uma arquitetura dedicada. Permitem ainda a simulação da arquitetura resultante, que necessita de sofrer melhorias para tornar o processo completo.

6.2 Trabalho Futuro

O trabalho desenvolvido ao longo desta Dissertação abre portas a outros trabalhos. No sentido de colmatar as lacunas evidenciadas ao nível da simulação é necessário um estudo mais aprofundado da interação entre memórias *cache* e restante processador. É igualmente imperativo o desenvolvimento de soluções que permitam a simulação a 100% desse tipo de arquiteturas.

Ainda no que concerne às memórias *cache* seria interessante e deveras vantajosa a implementação de um módulo que regule o acesso concorrente à memória externa por várias memórias deste tipo. Deste modo, tornar-se-ia possível a utilização de vários blocos de memória *cache*, sem a restrição do uso de apenas um bloco.

Uma outra perspetiva de evolução diz respeito à forma como o processador opera. Até ao momento o processador opera sempre com o mesmo caminho de dados. Uma possibilidade de trabalho futuro consiste na mudança do caminho de dados em tempo real. Para isso podem ser usados os 2 *bits* incluídos no micro-código que representam essa indicação.

Neste contexto, outro ponto de interesse é o desenvolvimento de campos de instrução diferentes que ofereçam novas funcionalidades. Por exemplo, poderiam ser criados os campos *Pause* e *Continue* que permitam a introdução de paragens entre o processamento. Estes intervalos de tempo podem ser usados para a reconfiguração do caminho de dados. Outra alternativa poderia ser a implementação de um comando *Restart* que possibilite um salto incondicional para o início

do micro-programa e estabelecer o seu reinício.

Finalmente, ao nível do fluxo de projeto em geral existem 2 tarefas que são feitas manualmente. Por um lado, surge o processo de tradução de código numa linguagem de programação ou semelhante, como C, em código simbólico. Por outro lado, a correspondência entre a codificação dos caminhos de dados e o respetivo módulo em *Verilog* que concretiza cada caminho de dados. A implementação de ferramentas de *software* que automatizem estes processos iria conferir mais rapidez, fiabilidade e facilidade no ponto de vista global do projeto.

Anexo A

Resultados das Funcionalidades Secundárias

Neste anexo são apresentados os resultados obtidos com o uso de argumentos facultativos das ferramentas de *software* desenvolvidas. As figuras apresentadas ajudam a perceber o efeito da utilização dos mesmos.

A.1 Argumento *-comment*

A Figura A.1 mostra o código binário obtido com a execução do compilador com recurso ao argumento *-comment*.

```
//Binary microcode:
//J command_I command_vector command
//2 last bits for loop command

//Line0: Noloop; Sel TESTE; Sel EXP;
000_000_11__000_000_11__00
//Line1: Noloop; Nop; Set EXP[i--,j--];
000_000_00__010_010_00__00
//Line2: Loop 480; Read TESTE[i,j++]; Write EXP[i,j++];
001_000_01__001_000_10__01
//Line3: Loop 319; Read TESTE[i,j++]; Write EXP[i,j++];
001_000_01__001_000_10__01
//Line4: Next; Read TESTE[i,j++]; Write EXP[i,j++];
001_000_01__001_000_10__11
//Line5: Noloop; Read TESTE[i++,j=0]; Write EXP[i++,j=0];
101_001_01__101_001_10__00
//Line6: Next; Nop; Set EXP[i,j--];
000_000_00__010_000_00__11
//Line 7: Halt program
111_111_11__111_111_11__11
```

Figura A.1: Ficheiro com micro-código binário gerado com o argumento *-comment*.

A.2 Argumento *-help*

O menu de ajuda impresso na linha de comandos com o argumento *-help* está representado na Figura A.2.

```
                << Lista de argumentos possiveis>>


-r ---------------- Opcao de leitura
-w ---------------- Opcao de escrita
-m ---------------- Ficheiro de descricao das memorias do processador
-c ---------------- Ficheiro com o pseudo codigo a processar
-vn ---------------- Nivel de volume de comentarios a incluir na execucao do
compilador, em ordem crescente, com n de 1 a 3
-switch ---------------- Opcao que permite activar a impressao de comentarios no
decorrer da execucao do compilador


                << Lista de comandos possiveis do micro-codigo >>


 - Comandos para controlo de loops:

Noloop ---------------- Sem ciclo
Loop x ---------------- Inicio de ciclo com x iteracoes
Next ---------------- Fecho do ultimo ciclo aberto


 - Comandos para execucao de instrucoes:

Sel X ---------------- Selecciona o vector X
Set X[i,j] ---------------- Activa o vector, podendo escolher como
movimentar os indices do vector X
Read X[i,j] ---------------- Le o conteudo do vector X, fazendo em seguida a
operacao seleccionada sobre os indices do mesmo
Write X[i,j] ---------------- Escreve no vector X, fazendo em seguida a operacao
escolhida sobre os indices do mesmo
Nop ---------------- Nao faz nenhuma operacao nem movimenta os indices de nenhum vector


 - Operacoes possiveis sobre os indices:

i,j ---------------- Mantem indice
i++,j ---------------- Executa instrucao e depois incrementa o indice
i--,j ---------------- Executa instrucao e depois decrementa o indice
i+S,j ---------------- Adiciona a constante S ao indice
i-S,j ---------------- Subtrai a constante S ao indice
i=0,j ---------------- Coloca o indice a 0
i=MAX,j ---------------- Coloca o indice no valor maximo definido
```

Figura A.2: Menu de ajuda disponível nas ferramentas de *software*.

A.3 Argumento *-v1*

O argumento -vNUM pode ser executado com vários níveis de verbosidade. Na Figura A.3 encontra-se ilustrado um exemplo de execução com este argumento. O nível do argumento usado foi 1, que define um nível de texto mínimo.

```
A abrir o ficheiro mem_details.vas...
A abrir o ficheiro instrucoes.vas...
A abrir o ficheiro code.bin...

        Ficheiro mem_details.vas a ser processado...


        Ficheiro instrucoes.vas a ser processado...


                !! Codigo binario gerado com sucesso !!

A fechar os ficheiros abertos...

                                    || Terminado ||
```

Figura A.3: Execução do compilador por omissão.

A.4 Argumento *-v3*

Finalmente, a Figura A.4 representa a função do argumento -v3. Uma vez que foi utilizado o nível 3, o nível de verbosidade alcançado é máximo, com o detalhe de todas as operações efectuadas.

```
A abrir o ficheiro code.bin...
Ficheiro binario sem comentarios.

        Ficheiro mem_details.vas a ser processado...

Memoria n ways associative so de leitura,4 vectores.
Memoria scratch de leitura e escrita, 2 vectores.
        Ficheiro instrucoes.vas a ser processado...

Instrucao Noloop a ser processada...
Concluida!
```

Figura A.4: Execução do compilador com recurso ao argumento *-v3*.

Anexo B

Situações de Erro

Neste anexo são apresentados os resultados obtidos na fase de testes de situações de erro no decorrer do processo de assemblação.

Deste modo, a primeira ação incorreta testada foi a seleção de um vetor não alocado na respectiva memória; na linha 0 é feita uma tentativa de seleção do vetor A, que não se encontra alocado na memória MEM0. Como tal, foram usados os ficheiros de descrição da Figura B.1 e, como seria de esperar, o micro-assemblador incorre num estado de erro.

```
                        /* FICHEIRO mem_details.vas */

MEM0: TESTE[2,2]:1:100:2:1:2; B[10,50]:200:300:10:5:10; \\
      C[5,3]:400:500:1:1:1;   K[5,3]:550:650:1:1:1;     cachenwa; 2; 8; 8; RW;
MEM1: DF[100]:700:800:10:10;  EXP[34,7]:200:400:3:8:8;  scratch; 1024; WO;
output: dout;  MEM1;
input:  din:MEM0;
```

```
                        /* FICHEIRO instrucoes.vas */

0: Noloop;    Sel A;                  Sel EXP;             SOBEL;
//0: Noloop;  inv TESTE;              Sel EXP;             SOBEL;
1: Loop 10;   Nop;                    Set EXP[i--,j--];    SOBEL;
2: Next;      Read TESTE[i--,j=MAX];  Write EXP[i,j++];    SOBEL;
3: Loop 20;   Nop;                    Write EXP[i,j];      SOBEL;
4: Next;      Nop;                    Set DF[i+S];         SOBEL;
5: Noloop;    Read TESTE[i-S,j=0];    Write DF[j=MAX];     SOBEL;
6: Noloop;    Nop;                    Nop;                 SOBEL;
7: Halt;
```

```
                        /* FICHEIRO datapath_conf.vas */

SOBEL;  00;   5;
```

Figura B.1: Ficheiros de descrição da arquitetura para teste de erros. O ficheiro *instrucoes.vas* apresenta propositadamente 2 situações de erro, presentes nas 2 primeiras linhas do ficheiro.

```
A abrir o ficheiro mem_details.vas...
A abrir o ficheiro instrucoes.vas...
A abrir o ficheiro code_teste.bin...

        Ficheiro mem_details.vas a ser processado...


        Ficheiro instrucoes.vas a ser processado...

Esta a tentar processar o vector A que nao existe na memoria 0.
Ver instrucao 2 da linha 0!


A fechar os ficheiros abertos...

                                || Terminado ||
```

Figura B.2: Situação de erro - vetor não alocado em memória.

Assim, imprime uma mensagem de aviso onde explicita detalhadamente a incorreção encontrada. A mensagem obtida pode ser consultada na Figura B.2.

A segunda ocorrência de erro representada deriva da utilização de um campo de instrução inválido. Com o intuito de ensaiar esta situação, foram usados os mesmos ficheiros do erro anterior. No entanto, foi comentada a linha 0 e descomentada a segunda linha 0, para teste do comando *inv*. O campo em questão é identificado como errado e apontado por uma informação de erro. A Figura B.3 ilustra esta situação de teste.

```
A abrir o ficheiro mem_details.vas...
A abrir o ficheiro instrucoes.vas...
A abrir o ficheiro code_teste.bin...

        Ficheiro mem_details.vas a ser processado...


        Ficheiro instrucoes.vas a ser processado...

2a instrucao da linha 0 desconhecida!


A fechar os ficheiros abertos...

                                || Terminado ||
```

Figura B.3: Situação de erro - instrução inválida.

Anexo C

Exemplo de Configuração do Processador

Neste anexo é incluída uma representação gráfica de um exemplo de uma configuração do processador. Esta respeita aos ficheiros apresentados na secção 4.3.

Dado que as ferramentas de *software* não configuram a estrutura de controlo do processador, na Figura C.1 apenas foram representadas as memórias e o caminho de dados. Estes são os blocos que são parametrizados de acordo com os ficheiros de descrição da arquitetura.

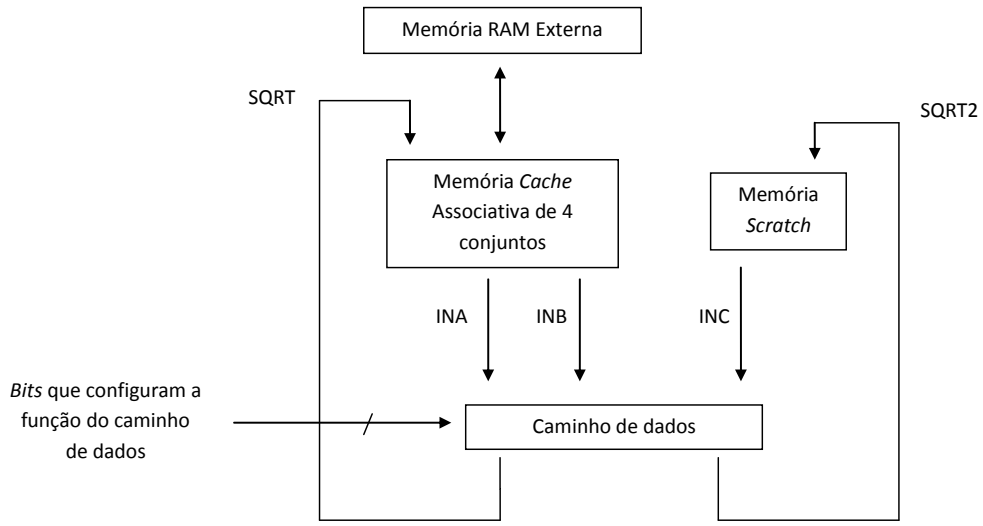

Figura C.1: Exemplo de configuração do processador.

Referências

[1] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics, Volume 38, Número 8*, Abril 1965.

[2] P. Yiannacouras, J.G. Steffan, e J. Rose. Vespa: portable, scalable, and flexible fpga-based vector processors. *CASES'08, Atlanta, EUA*, Outubro 2008.

[3] John L. Hennessy e David Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, Quarta edição, 2007.

[4] Rajeev Balasubramonian, David Albonesi, Alper Buyuktosunoglu, e Sandhya Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, Dezembro 2000.

[5] M.D. Hill e M.R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33 –38, Julho 2008.

[6] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31:532–533, Maio 1988.

[7] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, e Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Relatório té, EECS Department, University of California, Berkeley, Dezembro 2006.

[8] Michael J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948 –960, Setembro 1972.

[9] XuBang Shen. Evolution of mpp soc architecture techniques. *Science in China Series F: Information Sciences*, 51:756–764, 2008.

[10] Junho Cho, Hoseok Chang, e Wonyong Sung. An fpga based simd processor with a vector memory unit. Em *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*, página 4 pp., 2006.

[11] Krste Asanovic. *Vector Microprocessors*. Tese de doutoramento, Universidade de California, Berkeley, EUA, 1998.

[12] Jason Yu, Guy Lemieux, e Christpher Eagleston. Vector processing as a soft-core cpu accelerator. Em *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, FPGA '08, páginas 222–232, New York, NY, USA, 2008. ACM.

[13] Jae-Sung Yoon, Donghyun Kim, Chang-Hyo Yu, e Lee-Sup Kim. A 3d graphics processor with fast 4d vector inner product units and power aware texture cache. Em *Custom Integrated Circuits Conference, 2008. CICC 2008. IEEE*, páginas 539 –542, Setembro 2008.

[14] Roger Espasa, Mateo Valero, e James E. Smith. Vector architectures: past, present and future. Em *Proceedings of the 12th international conference on Supercomputing*, ICS '98, páginas 425–432, New York, NY, USA, 1998. ACM.

[15] Roger Espasa, Mateo Valero, e James E. Smith. Out-of-order vector architectures. Em *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, páginas 160–170, Washington, DC, USA, 1997. IEEE Computer Society.

[16] S. Brown. Fpga architectural research: a survey. *Design Test of Computers, IEEE*, 13(4):9 –15, winter 1996.

[17] Iouliia Skliarova e Antônio B Ferrari. Introdução à computação reconfigurável. *Revista do Departamento de Electrónica e Telecomunicações da Universidade de Aveiro.*, 2(6), Setembro 2003.

[18] *Xilinx.* www.xilinx.com.

[19] P. Yiannacouras, J.G. Steffan, e J. Rose. Data parallel fpga workloads: Software versus hardware. Em *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, páginas 51 –58, Setembro 2009.

[20] *DECPeRLe-1 Board. Disponível em* http://web.cecs.pdx.edu/~mperkows/RECONFIGURABLE/tsld003.html.

[21] C. Kozyrakis e D. Patterson. Vector vs. superscalar and vliw architectures for embedded multimedia benchmarks. Em *Microarchitecture, 2002. (MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*, páginas 283 – 293, 2002.

[22] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, e Peter Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. Em *Proceedings of the tenth international symposium on Hardware/software codesign*, CODES '02, páginas 73–78, New York, NY, USA, 2002. ACM.

[23] Yanqin Yang, Meng Wang, Haijin Yan, Zili Shao, e Minyi Guo. Dynamic scratch-pad memory management with data pipelining for embedded systems. *Concurrency and Computation: Practice and Experience*, 22(13):1874–1892, 2010.

[24] M.D. Hill. A case for direct-mapped caches. *Computer*, 21(12):25 –40, Dezembro 1988.

[25] Mahmut Kandemir, Taylan Yemliha, SaiPrashanth Muralidhara, Shekhar Srikantaiah, Mary Jane Irwin, e Yuanrui Zhnag. Cache topology aware computation mapping for multicores. *SIGPLAN Not.*, 45:74–85, Junho 2010.

[26] Michael D. Powell, Amit Agarwal, T. N. Vijaykumar, Babak Falsafi, e Kaushik Roy. Reducing set-associative cache energy via way-prediction and selective direct-mapping. 2001.

[27] Donald E. Thomas e Philip R. Moorby. *The Verilog hardware description language, Volume 1*. Kluwer Academic, Primeira edição, 2002.

[28] D.Nicula e M.Cirstea. Successful cad tools application to fpga/asic design. *International Journal of Engineering*, 15(1):72–76, 1999.

[29] R. for blind review. *A Programmable Microarchitecture for Streaming Computing*, 2010.

[30] *Cygwin.* www.cygwin.com.

[31] Zhang Jin-Yu, Chen Yan, e Huang Xian-Xiang. Edge detection of images based on improved sobel operator and genetic algorithms. Em *Image Analysis and Signal Processing, 2009. IASP 2009. International Conference on*, páginas 31 –35, april 2009.