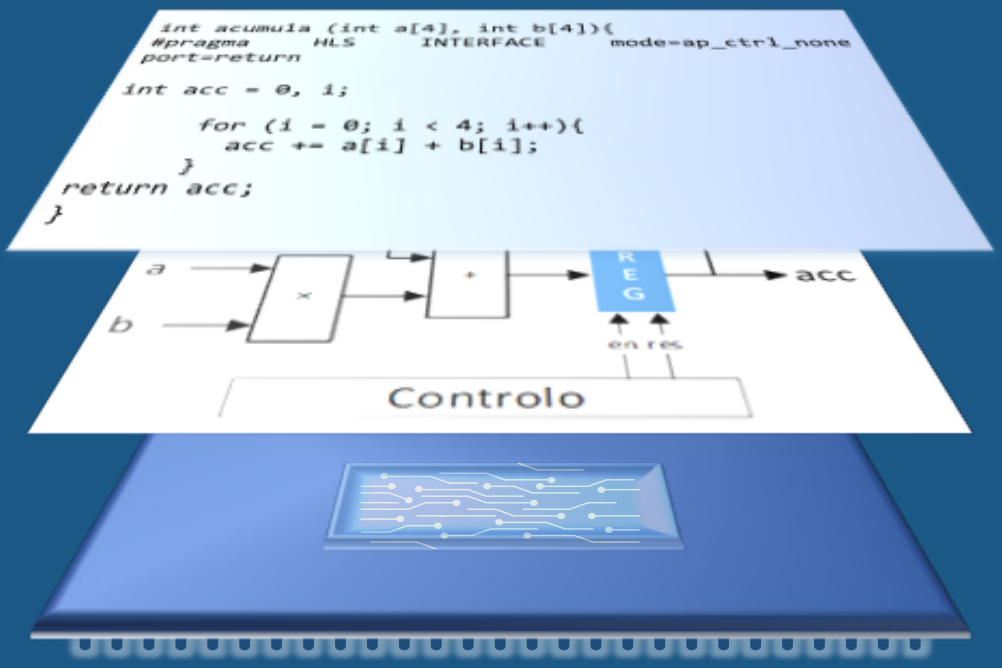
MÁRIO VÉSTIAS • PAULO FLORES • HORÁCIO C. NETO

COLEÇÃO **CAMINHOS DO CONHECIMENTO**

SÍNTESE DE ALTO NÍVEL EM FPGA



Síntese de Alto Nível em FPGA

Mário Véstias • Paulo Flores • Horácio C. Neto

Síntese de Alto Nível em FPGA

POLITÉCNICO DE LISBOA | POLYTECHNIC UNIVERSITY OF LISBON

TÍTULO
Síntese de Alto Nível em FPGA

AUTORES
Mário Véstias, Paulo Flores, Horácio C. Neto

EDITOR
Instituto Politécnico de Lisboa

DESIGN DA CAPA
Pedro Antunes

EXECUÇÃO GRÁFICA
Gráfica 99

Julho de 2025

ISBN 978-989-35158-5-3

DEP. LEGAL N.º 550632/25

ÍNDICE

PARTE I

Conceitos, Dispositivos e Ferramentas

1 Introdução

As metodologias e as ferramentas de projeto de sistemas digitais têm evoluído com o objetivo de conseguir circuitos melhores e mais eficientes. Com o aumento da complexidade dos sistemas digitais, surgiu uma nova dimensão no desenvolvimento destes sistemas relacionada com a eficiência de projeto. É necessário lidar com o aumento crescente da complexidade dos circuitos, com a redução do tempo disponível para o projeto, com a integração de equipas de projeto e com a dinâmica de evolução das novas tecnologias de circuito integrado.

A introdução de linguagens de descrição de hardware (HDL – *Hardware Description Language*) foi fundamental para dar resposta ao aumento da complexidade dos sistemas digitais. Numa metodologia de projeto de sistemas digitais baseada em linguagens de descrição de hardware, os circuitos são descritos com uma HDL (p. ex., VHDL ou Verilog). As vantagens deste tipo de abordagem ao projeto de hardware são várias, como a documentação formal do circuito, a integração com ferramentas de síntese e de simulação, a reutilização de código, a portabilidade, entre outros. Os fluxos de projeto de circuito digital baseados em linguagens de descrição de hardware e as ferramentas de síntese e de simulação foram fundamentais para dar resposta ao projeto de sistemas digitais com crescente complexidade.

Com o evoluir da tecnologia de circuito integrado e a complexidade das aplicações, bem como a redução do tempo de projeto, os fluxos de projeto baseados em linguagens de descrição de hardware tornam-se cada vez mais difíceis de conseguir e com maiores custos.

Para lidar de forma eficiente com esta complexidade, é necessária uma abordagem à síntese de sistemas digitais a um nível de abstração mais elevado. Este nível de abstração foi conseguido com a síntese de alto nível (HLS – *High Level Synthesis*). A síntese de alto nível permite traduzir funcionalidades descritas numa linguagem de alto nível, por exemplo a linguagem C ou a C++, de forma automática, análogo ao trabalho realizado pelos compiladores de software. O conceito começou a ser investigado no início dos anos 80, tendo surgido algumas ferramentas comerciais na segunda metade da década de 90. Contudo, foi só a partir da década seguinte que houve um investimento forte de várias

empresas na terceira geração de ferramentas de síntese de alto nível que alavancou o sucesso da síntese de alto nível. Um dos fatores que ajudou neste processo foi a tecnologia de hardware reconfigurável, permitindo não só um projeto rápido, como também a implementação rápida do circuito em hardware.

A HLS tornou-se numa importante ferramenta de projeto, com os maiores mercados a utilizarem a HLS no projeto dos seus circuitos integrados. A presente obra procura contribuir para o crescimento da HLS ao disponibilizar conteúdos necessários à formação de quem pretende trabalhar nesta área.

1.1 A Importância do Hardware Digital

Um sistema de computação para a execução de um algoritmo ou de uma aplicação baseia-se geralmente num processador de aplicação genérica. O projetista escolhe uma plataforma de processamento e uma linguagem de programação, seguindo-se o desenvolvimento e o teste do código sobre a plataforma escolhida.

O tempo de execução do algoritmo depende do código desenvolvido e do sistema de computação. Na maioria dos casos, o código compilado é executado num processador de aplicação genérica. Com o evoluir das tecnologias de circuito integrado, os processadores tornaram-se mais rápidos e, consequentemente, os algoritmos executam mais rápido. No entanto, a evolução da tecnologia não consegue acompanhar as exigências computacionais das novas aplicações. Foi necessário explorar novas arquiteturas de processamento que permitissem acelerar a execução dos algoritmos.

Duas das principais abordagens arquiteturais que permitem acelerar a execução das aplicações são as arquiteturas dedicadas e as de multiprocessamento. Alguns exemplos de processadores que exploram estes conceitos são os processadores de sinal digital (DSP – *Digital Signal Processor*) e as unidades de processamento gráfico (GPU – *Graphics Processing Unit*). Ambos incluem unidades de computação dedicadas à execução de funções específicas (p. ex., processamento de filtros, cálculo vetorial, etc.) e exploram o processamento paralelo. Em vez de melhorarem o desempenho com o aumento da frequência, melhoram-no com a utilização de hardware dedicado e de processamento em paralelo.

A especialização e o multiprocessamento são uma solução de aceleração, particularmente importantes na execução de algoritmos computacionalmente exigentes. No limite, podemos desenvolver o hardware de um sistema computacional dedicado à execução de um algoritmo ou um conjunto de algoritmos específicos. Para tal, dispomos das tecnologias ASIC (*Application-Specific Integrated Circuit*) e FPGA (*Field-Programmable Gate Array*). A tecnologia ASIC oferece-nos a solução mais eficiente, ou seja, mais rápida, com menos consumo de energia e uma densidade de circuito por área de silício maior. Contudo, implica o projeto e o fabrico da arquitetura computacional em circuito integrado. Em alternativa, a

FPGA disponibiliza uma plataforma de hardware programável que evita incluir no projeto do sistema digital o passo de fabrico do circuito integrado, mas a solução final é menos eficiente do que a conseguida com a utilização da tecnologia ASIC. A escolha de uma ou de outra tecnologia depende da relação entre os custos de projeto e o volume de vendas. Para um volume de produção médio, a tecnologia FPGA é em geral mais rentável, pois não tem custos de projeto do circuito integrado. Para um volume de produção elevado, a tecnologia ASIC é mais rentável, pois o custo unitário reduz com o aumento de unidades produzidas.

A adoção de soluções baseadas em sistemas computacionais digitais dedicados, com tecnologia FPGA ou ASIC, conduz-nos ao projeto de sistemas digitais.

1.2 Projeto de Sistemas Digitais com HLS

O projeto de um sistema digital baseia-se num paradigma diferente do de desenvolvimento de software. Uma aplicação é descrita em software com uma linguagem de programação que cria uma abstração da plataforma onde será executada, tipicamente um processador. O programador pode facilmente alterar a descrição software, o que permite melhorar o código, garantir a portabilidade, etc. A descrição software é depois convertida para um conjunto de instruções específicas do processador alvo. Esta tarefa é realizada pelo compilador. Este processo de desenvolvimento de software deu origem a um vasto conjunto de linguagens de programação, de compiladores e de processadores que procuram a implementação da aplicação mais eficiente.

Um aspeto central do desenvolvimento de software é que a arquitetura computacional alvo, o processador, é fixa, com um conjunto de instruções pré-definidos. A única forma de melhorar a execução da aplicação é otimizar o código ou optar por um processador mais rápido.

No caso do projeto de hardware temos uma dimensão adicional: a arquitetura computacional alvo. O projetista de hardware tem de desenvolver a arquitetura alvo e sintetizar o algoritmo para essa arquitetura. O projeto de um algoritmo em hardware é, por isso, mais moroso e complexo do que o projeto da funcionalidade em software. Contudo, esta abordagem permite obter um circuito dedicado com melhores características computacionais do que as conseguidas com a execução da aplicação software num processador genérico. A solução em hardware é, em geral, considerada quando a solução em software não responde às restrições de desempenho, de custo, de consumo, etc.

Quando se opta por desenvolver um determinado algoritmo em hardware, o ponto de partida é, em geral, a descrição do algoritmo numa linguagem de alto nível. Para o desenvolvimento do sistema hardware dedicado com HLS podemos partir da mesma descrição software do algoritmo, mas as abordagens de projeto de hardware e de software são diferentes. O programador, que tem como alvo um processador genérico, centra-se no problema de programação do algoritmo, deixando as otimizações a cargo do compilador.

No caso da arquitetura alvo se tratar de um processador de aplicação específica, já existe a preocupação em tirar proveito de algumas das unidades dedicadas do processador para otimizar a execução do algoritmo. Por exemplo, se um processador permite a execução de instruções vetoriais, a descrição software deve procurar cumprir os requisitos de organização que permitam ao compilador traduzir o código da forma mais eficiente que tire proveito das capacidades de cálculo vetorial do processador. No caso do projetista de hardware, é possível adaptar a arquitetura de processamento ao fluxo de execução do algoritmo. Assim, o código do algoritmo é ajustado para permitir sintetizar a arquitetura que melhor cumpre os requisitos de projeto. Por exemplo, separar um vetor de elementos para permitir o acesso a múltiplos elementos em simultâneo, desenrolar uma estrutura de repetição (por exemplo, uma estrutura `for`) para permitir a execução paralela de várias iterações ou alterar o tipo de interface de modo a melhorar os tempos de acesso aos dados. O projeto de hardware baseado em síntese de alto nível permite ao projetista, através de diretivas adicionadas ao código, explorar estes aspetos, orientando a ferramenta de síntese de modo a atingir os requisitos do projeto.

Esta distinção entre a programação de software e o projeto de hardware indica que o desenvolvimento de hardware com síntese de alto nível não consiste simplesmente em pegar num algoritmo descrito com uma linguagem de alto nível e sintetizá-lo. Uma vez que existem múltiplas implementações do algoritmo em hardware, o projetista tem de orientar a ferramenta de HLS de modo a gerar a arquitetura que melhor se ajusta aos requisitos do projeto.

1.3 Síntese de Alto Nível no Projeto de Hardware

A síntese de alto nível utiliza linguagens de programação para descrever o algoritmo a sintetizar, sendo as mais utilizadas as linguagens C, C++, SystemC e OpenCL.

A descrição inicial não inclui conceitos de baixo nível, como o sinal de relógio. Estes só surgem após a síntese de alto nível. O resultado da síntese corresponde a uma descrição de transferência de registos (RTL – *Register Transfer Level*) numa linguagem de descrição de hardware (e.g., VHDL ou Verilog). Esta descrição é depois tratada por uma ferramenta de síntese lógica para mapear a descrição RTL numa determinada tecnologia.

A síntese de alto nível permite a simulação do circuito a um nível de abstração mais elevado e facilita a exploração das diferentes opções arquiteturais. Esta exploração faz-se através de restrições ou diretivas que são consideradas automaticamente pela ferramenta de síntese. Os casos mais comuns de diretivas são as de interface de comunicação de dados, as de memória, as relacionadas com estruturas de repetição e as temporais. Muitas destas diretivas competem entre si ou só têm efeito em conjunto. Por exemplo, podemos aumentar o número de operações em paralelo sem obter qualquer efeito nos tempos de execução devido a limitações no acesso aos dados através da interface. Neste caso, é

preciso que a largura de banda da interface aumente com as necessidades de acesso aos dados.

A descrição de hardware com linguagens de programação de alto nível exige algum esforço do projetista no sentido de guiar a ferramenta de síntese, mas esta orientação é realizada a um nível de abstração elevado. Contudo, temos de ter presente que a redução do tempo de projeto através da utilização de ferramentas HLS tem, geralmente, um custo na solução final. A solução obtida poderá não estar tão otimizada quando comparada com uma solução obtida com a síntese do circuito a partir de uma descrição com uma linguagem de descrição de hardware. No entanto, este hiato tem sido reduzido à medida que as ferramentas HLS têm evoluído.

A síntese de alto nível garante um fluxo de projeto de hardware desde uma descrição abstrata numa linguagem de alto nível até uma descrição hardware do circuito. As vantagens na utilização de ferramentas de síntese de alto nível são diversas:

- Redução do tempo de projeto: ao descrever o circuito a um nível de abstração funcional omitem-se muitos detalhes relacionados com a tecnologia, protocolos de comunicação, etc. O projetista apenas tem de se preocupar com a funcionalidade do circuito.
  Para além de automatizar a geração da descrição do circuito hardware, a ferramenta de HLS gera estimativas de desempenho e de área que guiam o projetista na escolha da melhor arquitetura. Por exemplo, o projetista pode intervir na geração do hardware através de diretivas (p. ex., nível de paralelismo) integradas na descrição funcional. É assim possível explorar várias arquiteturas para o mesmo problema, com diferentes relações entre, por exemplo, área e desempenho;
- Redução do tempo de verificação: sendo a funcionalidade descrita a um nível de abstração elevado com uma linguagem de programação, é possível verificar a funcionalidade da descrição através da simulação do código. Sendo uma simulação funcional a um nível de abstração elevado, a verificação é mais rápida e permite aplicar mecanismos de depuração de erros;
- Reutilização de funções: tratando-se de uma descrição utilizando uma linguagem de alto nível, facilmente se reutilizam funções independentes da tecnologia. Os detalhes de tecnologia surgem apenas durante o processo de síntese;
- Projeto focado na funcionalidade: uma vez que o projetista não tem de se preocupar com detalhes de implementação, fica livre para se concentrar na funcionalidade do circuito e em encontrar o algoritmo que melhor responde às necessidades do projeto.

A descrição de um sistema digital com uma linguagem de programação de alto nível aproxima o hardware do software, ou seja, o projetista de hardware e o programador descrevem ambos o algoritmo com linguagens de programação. Podem inclusive utilizar a mesma linguagem de programação e até o mesmo código. No entanto, as implementações

seguem caminhos diferentes. No projeto de software, o código do algoritmo ou da aplicação desenvolvido pelo programador será executado num processador. No projeto de hardware, o código desenvolvido descreve um circuito hardware que executa o algoritmo ou a aplicação. Apesar de descrever o circuito com uma linguagem de alto nível, a preocupação do projetista de hardware é desenvolver um sistema digital.

1.4 Tecnologia Alvo

A síntese de alto nível é independente da tecnologia e, como tal, pode ter como alvo a tecnologia ASIC ou FPGA. Neste livro, considera-se como tecnologia alvo os dispositivos FPGA. Sendo independente da tecnologia, os conceitos de síntese de alto nível abordados nos capítulos seguintes aplicam-se igualmente a ambas as tecnologias.

Existem diversas ferramentas de síntese de alto nível para FPGA, como o *Vitis HLS* da *AMD-Xilinx*, o *HLS Compiler* da *Intel-Altera* e o *Catapult HLS* da Siemens, entre outras. As ferramentas variam entre si no conjunto de linguagens de especificação suportadas, na sintaxe das diretivas, na usabilidade e na qualidade dos resultados. No entanto, os conceitos e os métodos abordados neste livro surgem em qualquer ferramenta de HLS, pelo que a utilização de uma em particular não limita a aplicabilidade dos conteúdos do livro.

Neste livro, utilizamos a ferramenta *Vitis HLS* da *AMD-Xilinx* para exemplificar os métodos e conceitos apresentados. O *Vitis HLS* sintetiza código descrito com linguagens C ou C++ em código RTL especificado em VHDL ou em Verilog sintetizável e suporta simulação comportamental e co-simulação hardware/software. Para demonstrar os resultados obtidos, utilizou-se como dispositivo alvo a FPGA XCZU7EV-2 da família *ZYNQ Ultrascale+* da *Xilinx*.

1.5 Organização do Livro

O livro está organizado em três partes. A parte I introduz conceitos, dispositivos e ferramentas e está dividida em 4 capítulos:

- Capítulo 1: Faz uma breve introdução à síntese de alto nível e descreve a organização do livro;
- Capítulo 2: Descreve a tecnologia de lógica programável com dispositivos FPGA. Faz uma abordagem à arquitetura do dispositivo e de que forma implementa hardware, e por fim descreve a arquitetura da FPGA considerada nos exemplos deste livro;
- Capítulo 3: Descreve conceitos e arquiteturas de hardware típicas necessárias à melhor compreensão dos mecanismos de síntese de alto nível. Abordam-se métricas de desempenho, mecanismos de otimização, paralelismo e *pipeline*, circuitos de memória e protocolos de interface;
- Capítulo 4: Descreve a síntese de alto nível, nomeadamente o fluxo de síntese com os seus diversos passos de projeto. Descreve ainda de uma forma genérica os diversos mecanismos de descrição de hardware com as linguagens de programação.

A parte II descreve com detalhe os métodos de descrição dos circuitos, bem como as diretivas associadas às estruturas de repetição, às interfaces, às memórias, ao tipo de dados e ao projeto modular e hierárquico de hardware. Está igualmente dividida em 4 capítulos:

- Capítulo 5: Descreve os mecanismos de síntese de estruturas de repetição e de condições, bem como as respetivas diretivas que permitem ao projetista controlar a síntese dos mesmos;
- Capítulo 6: Aborda a síntese dos argumentos das funções de descrição dos circuitos em interfaces. Explica como é que cada um dos argumentos é sintetizado em interfaces, quais as opções de implementação de cada um e como otimizar as interfaces de modo a responder às exigências de comunicação do sistema digital;
- Capítulo 7: Introduz os tipos de dados suportados pela síntese de alto nível. Para além dos tipos de dados nativos das linguagens de programação, o HLS suporta tipos de dados com precisão arbitrária. O capítulo descreve de que forma se utilizam estes tipos de dados e quais as vantagens em utilizá-los na síntese de hardware;
- Capítulo 8: Aborda o projeto modular de hardware com interligação de vários componentes ao mesmo nível ou em hierarquia.

A parte III conclui o livro com exemplos práticos de algoritmos. Inclui a descrição de dez algoritmos/aplicações em C/C++ que podem ser sintetizados com a ferramenta de HLS e testados pelo leitor em FPGA. Pretende-se com este capítulo dar uma visão prática de utilização da HLS e ao mesmo tempo introduzir alguns mecanismos de otimização bastante utilizados. Muitos dos exemplos são configuráveis em termos de paralelismo, permitindo gerar diversas soluções com diferentes compromissos entre desempenho e ocupação de recursos através da alteração de apenas alguns parâmetros.

2 Projeto de Hardware com Lógica Programável

Para melhor perceber como funciona a síntese de alto nível e interpretar os seus resultados, é importante conhecer o dispositivo alvo e de que forma determinadas especificações são implementadas em hardware programável. Nas secções seguintes, descreve-se a arquitetura das FPGA e a sua evolução, bem como a forma como as características do hardware programável determinam a solução hardware final.

2.1 Arquitetura do Dispositivo de Lógica Programável FPGA

A FPGA é um circuito integrado cujo hardware pode ser reprogramado após ter sido fabricado, permitindo assim a implementação de qualquer circuito ou sistema digital, limitado apenas pela quantidade de lógica programável disponível na FPGA. Adicionalmente, a FPGA permite a reprogramação dinâmica total ou parcial, ou seja, o hardware pode ser reprogramado alternadamente ou em paralelo com a execução através da reprogramação parcial.

A FPGA é constituída por um conjunto de elementos base que permitem implementar circuitos lógicos, interligações entre os circuitos lógicos e blocos de entrada/saída que permitem ligar a FPGA a dispositivos externos. Todos estes elementos são programáveis e estão distribuídos pela FPGA de forma estruturada. Genericamente, a arquitetura de uma FPGA consiste numa estrutura regular dos vários elementos base (ver Figura 2-1).

A estrutura é composta pelos seguintes elementos:

- Bloco Lógico Configurável (CLB – *Configurable Logic Block*) – Bloco configurável que permite a implementação de lógica hardware. Os CLB estão dispostos no circuito integrado de forma regular segundo uma matriz a duas dimensões. O número de CLB disponíveis em uma FPGA depende da família de FPGA e da sua dimensão;
- Matriz de Interligação (IM – *Interconnection Matrix*) – Bloco de interligações programáveis que permite configurar as ligações entre linhas e colunas de CLB. As interligações também se dispõem na FPGA de forma regular;

- Bloco de Interligação (IB – *Interconnection Block*) – Bloco de interligações programáveis que estabelece a ligação entre os CLB e as IM. Os CLB têm vários IB vizinhos que permitem estabelecer ligações por linha e por coluna a outros CLB;
- E/S (Entrada/Saída) – Bloco configurável que estabelece a ligação entre os elementos lógicos e as interligações com os pinos do dispositivo para acesso externo. O número de blocos E/S depende, naturalmente, do número de pinos genéricos do dispositivo, ou seja, pinos a que o projetista tem acesso.

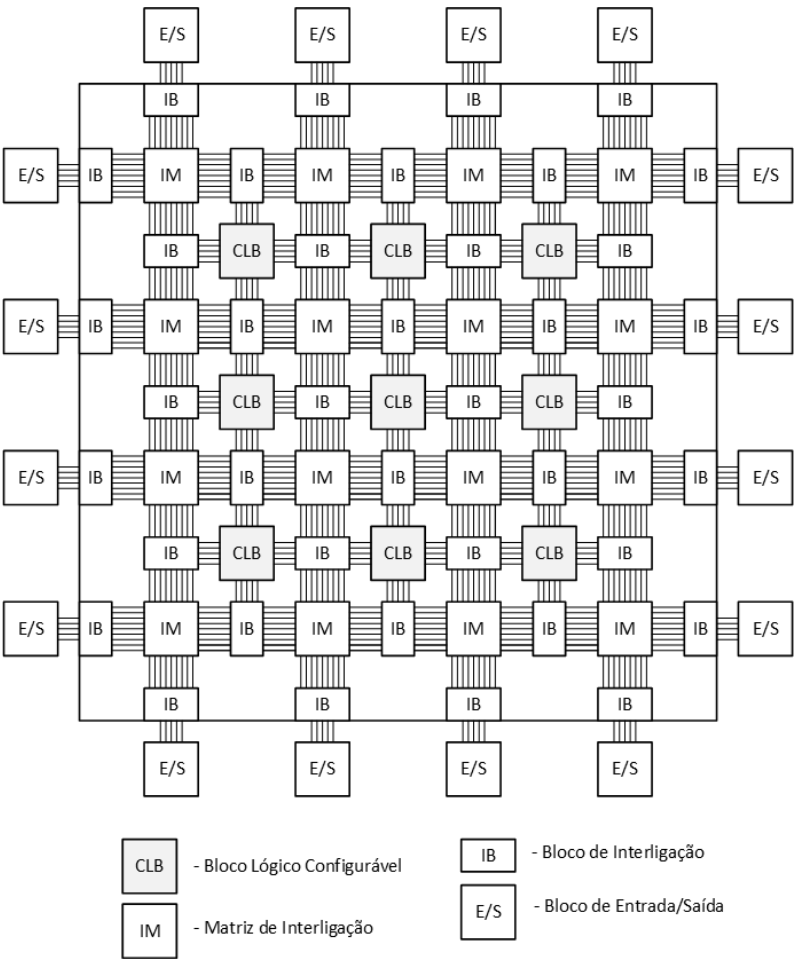


Figura 2-1 – Arquitetura genérica da FPGA

Os elementos programáveis estão dispostos de forma regular segundo uma matriz por todo o circuito integrado. As funções lógicas são implementadas nos blocos CLB, cuja arquitetura será vista mais à frente. Funções que não podem ser implementadas com um único CLB, são implementadas com vários CLB interligados através dos blocos de interligação: IM e IB.

O bloco IB permite o acesso dos blocos de lógica à rede de interligações e o bloco IM permite configurar as interligações entre os vários circuitos lógicos criando caminhos de dados entre módulos lógicos. Os blocos de E/S estabelecem a ligação dos circuitos implementados na FPGA com dispositivos externos. Estes blocos também são configuráveis.

A reconfigurabilidade da FPGA provém do facto de ser possível implementar uma nova funcionalidade em hardware após o fabrico do circuito integrado. Para tal, basta reprogramar cada um dos blocos CLB, IM, IB e E/S de acordo com o circuito digital pretendido.

Existem diferentes tecnologias de implementação da configurabilidade das FPGA. A tecnologia mais utilizada baseia-se em memória estática (SRAM – *Static Random-Access Memory*). Células de memória estática, SRAM, são utilizadas para guardar os bits de configuração de todos os blocos da FPGA, nomeadamente, para garantir a configurabilidade dos blocos lógicos e para realizar as interligações nos blocos MI e BI de interligação. Em alternativa à utilização de células SRAM, existe a tecnologia baseada em memória *flash*.

A principal vantagem em realizar a reprogramação com base em tecnologia SRAM, em comparação com a tecnologia *flash*, é a maior velocidade de operação, a menor dimensão dos elementos de memória e a necessidade de menor potência dinâmica. No entanto, as memórias SRAM são voláteis, ou seja, quando se desliga a alimentação do circuito perdem a programação. O sistema digital tem assim de voltar a ser reprogramado após ligar de novo a alimentação da FPGA. Para isso, é necessário que os dados de configuração estejam armazenados em dispositivos de memória não voláteis externos à FPGA.

A tecnologia predominante é a baseada em SRAM, devido principalmente ao facto de utilizar a mesma tecnologia de fabrico para as células de programação e para os circuitos lógicos e interligações, e por ser bastante mais rápida. Contudo, também existem algumas FPGA que têm integrada uma memória *flash*, evitando assim a necessidade de ter uma memória externa para armazenar a configuração.

Mais do que a tecnologia utilizada para implementar a configurabilidade da FPGA, é importante perceber como é que se implementam sistemas digitais nos blocos configuráveis da FPGA. Nas secções seguintes, descreve-se cada um dos blocos configuráveis e de que forma são utilizados na implementação de circuitos digitais.

2.1.1 Bloco Lógico Configurável

O bloco lógico configurável é o componente básico da FPGA que permite implementar lógica e memória. Sabemos que qualquer função lógica pode ser descrita com uma tabela de verdade e que uma tabela de verdade pode ser implementada com uma memória. Na FPGA, a tabela de verdade de uma função lógica é implementada com uma LUT (*Look-Up Table*). Uma LUT é formada por uma memória de N bits, seguida de um circuito de seleção (*multiplexer*) (ver Figura 2-2).

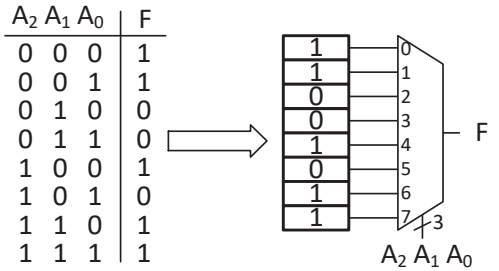


Figura 2-2 – Implementação de uma função de 3 variáveis com uma LUT de 3 entradas

O exemplo ilustra a implementação de uma função lógica de três variáveis binárias com uma LUT. Uma função lógica de três variáveis tem uma tabela de verdade com oito entradas, ou seja, com as oito combinações possíveis das variáveis binárias de entrada. Assim, a memória da LUT tem oito células e um *multiplexer* de oito entradas de dados e três entradas de seleção. O conteúdo da memória corresponde aos valores da função e o seletor do *multiplexer* recebe como entrada as variáveis binárias da função. Como se pode observar na figura, a memória é preenchida de acordo com os valores da função de saída.

O mecanismo pode ser generalizado para qualquer número de variáveis da função. Com M variáveis de entrada, é necessária uma memória com $N = 2^M$ posições e um *multiplexer* com N entradas de dados e M entradas de seleção. Uma LUT de M entradas implementa qualquer função lógica de M variáveis. A reprogramação da LUT faz-se simplesmente alterando o conteúdo da memória. Funções maiores que as permitidas por uma única LUT são implementadas à custa da interligação de várias LUT.

A LUT é assim o bloco lógico configurável da FPGA com a menor granularidade. A área ocupada por um circuito implementado nas LUT da FPGA é quantificada pelo número de LUT utilizadas. Naturalmente, esta métrica está dependente do tamanho da LUT. Ao longo da evolução da FPGA como dispositivo reprogramável, o tamanho da LUT tem variado. LUT maiores permitem que mais lógica seja implementada por LUT e são necessárias menos interligações, comparativamente ao que seria preciso com LUT mais pequenas. Contudo, LUT de maior dimensão têm tempos de propagação maiores e aumentam a possibilidade de parte da LUT ficar desperdiçada ao implementar funções lógicas mais pequenas do que a capacidade da LUT. Com LUT menores, temos menos atraso e menos desperdício, mas mais interligações. O tamanho da LUT tem evoluído de 8 posições para 16 para 64 e até para 256 posições, com algumas restrições.

Implementar as funções lógicas com LUT, em vez de utilizar portas lógicas, facilita a programação da função e garante que o tempo de propagação da função é sempre o mesmo.

Além da LUT, o bloco lógico configurável também contém elementos de memória na forma de flip-flop e ou *latch*. A possibilidade de ligar a saída da LUT a um flip-flop permite a implementação de lógica sequencial. A Figura 2-3 ilustra a estrutura base que interliga a LUT com um flip-flop de modo a permitir a implementação de circuitos lógicos sequenciais.

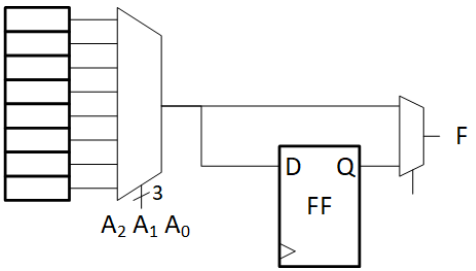


Figura 2-3 – Interligação da LUT com um flip-flop que permite a implementação de lógica sequencial.

O multiplexer à saída permite escolher entre uma função combinatória e uma função sequencial, em que a saída é registada num flip-flop.

Considerando que a implementação da lógica de configuração é feita com tecnologia SRAM, uma LUT com N posições é basicamente implementada com N células de SRAM.

Sendo implementada com células de memória, a LUT pode igualmente ser utilizada na implementação de memória. Uma LUT de 6 entradas pode ser utilizada como uma memória de 64 × 1 bits, sendo comummente designada de memória distribuída. A interligação de várias LUT permite implementar memórias de maior capacidade, ou seja, com mais posições de armazenamento e com mais bits armazenados por posição.

2.1.2 Blocos de Interligação

A função que é possível implementar num bloco lógico depende do tamanho da LUT. Quando se pretende implementar funções de maior complexidade do que a suportada por uma LUT, é necessário interligar várias LUT. Para isso, as LUT estão rodeadas de interligações configuráveis que permitem estabelecer ligações entre elas. A arquitetura mais comum de organização das LUT e das interligações é conhecida por *estilo-ilha*. Nesta arquitetura, as LUT encontram-se rodeadas de interligações, como se pode observar na Figura 2-1 da secção 2.1. Cabe às ferramentas de síntese lógica determinar como dividir a função lógica em várias LUT e como interligá-las por forma a executar a função lógica. As estruturas de interligação são bastante flexíveis e permitem interligar não só LUT vizinhas, mas também LUT mais afastadas.

O bloco lógico acede aos recursos de interligação através do bloco de interligação (BI), tanto na vertical, como na horizontal. Os canais de interligação atravessam o circuito integrado na vertical e na horizontal. O cruzamento das ligações verticais com as horizontais é feito com o módulo de interligação (MI). Este módulo consiste numa matriz de ligações

configuráveis que permitem ligar uma qualquer linha de comunicação horizontal a uma qualquer linha vertical. Um sinal que chega ao MI pode ser encaminhado para qualquer uma das direções e sentidos. Esta forma de interligação segmentada permite as ligações locais de vizinhança através dos blocos BI e as ligações de maior distância através de blocos MI, sem a intervenção dos blocos lógicos que surgem ao longo do caminho, ou seja, torna a comunicação de dados independente dos blocos lógicos.

Todos os pontos de interligação da arquitetura são programáveis com células de SRAM que definem se duas linhas que se intersetam estão ou não ligadas.

2.1.3 Otimização da Estrutura Básica da FPGA

A arquitetura da FPGA descrita nas secções anteriores é suficiente para implementar circuitos combinatórios e sequenciais. No entanto, procurando melhorar a eficiência da área do circuito integrado e os tempos de propagação das implementações, as novas FPGA incluem estruturas de lógica otimizadas e com ligações dedicadas, componentes aritméticos e blocos de memória local, com o objetivo de melhorar a área e o desempenho das funcionalidades hardware mais frequentes.

2.1.3.1 Estrutura de propagação do sinal de transporte das somas

Uma das otimizações nas estruturas lógicas está relacionada com a propagação do sinal de transporte das operações de soma. A operação de soma é bastante frequente nos circuitos digitais. Sabe-se que um somador completo recebe três bits à entrada (dois bits a somar mais um bit de transporte do somador completo anterior) e gera dois bits à saída (bit de soma e bit de transporte). O somador completo pode, assim, ser implementado com duas LUT, uma para cada bit de saída. Com a utilização de vários somadores completos consegue-se implementar a operação de soma com operandos de vários bits. Para tal, basta interligar os somadores completos em cascata, em que o sinal de transporte de saída de um somador completo é a entrada de transporte do somador completo seguinte. Encadeando N somadores completos, obtemos um somador de N bits.

A Figura 2-4 ilustra a implementação de um somador de 3-bits com três somadores completos, em que cada um é implementado com duas LUT. O circuito pode ser facilmente generalizado para operandos com qualquer número de bits.

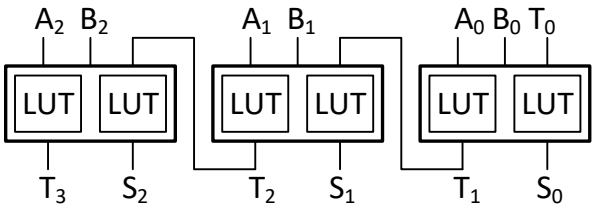


Figura 2-4 – Implementação de um somador de 3-bits com LUTs.

Começando pelos bits de menor peso à direita, em cada somador completo vai sendo gerado um bit de soma e um bit de transporte. O bit de transporte é propagado para o somador completo seguinte.

O tempo de propagação do somador é determinado pela propagação do sinal de transporte desde o primeiro somador completo até ao último. É assim proporcional ao tamanho em bits do somador. Quando as interligações dos sinais de transporte ao longo do somador são feitas com a malha de interligação genérica, o tempo de propagação do somador é fortemente penalizado.

A solução encontrada foi a de criar ligações dedicadas para a propagação das linhas de transporte entre LUT. Em vez de utilizar as ligações genéricas da arquitetura de interligações, criou-se uma estrutura dedicada para a propagação dos sinais de transporte.

Com esta estrutura dedicada de ligação, o atraso provocado pela propagação do sinal de transporte é consideravelmente reduzido, o que melhora bastante o tempo de propagação do somador, comparativamente à solução em que as linhas de transporte são implementadas com a arquitetura genérica de interligação.

2.1.3.2 Estrutura Hierárquica de LUT

LUT vizinhas são com grande probabilidade usadas em conjunto para implementar funções. Já referimos que LUT maiores favorecem a eficiência de utilização da área, mas ao aumentar a granularidade da LUT, estamos muito provavelmente a aumentar o subaproveitamento das LUT. Por outro lado, LUT mais pequenas exigem mais interligações entre si para implementar funções maiores o que pode piorar o desempenho. Uma solução mais flexível consiste em utilizar estruturas de LUT com tamanho variável e estruturas hierárquicas de LUT.

Uma abordagem comum na implementação de LUT consiste em permitir configurar o tamanho da LUT. Tipicamente, considera-se uma LUT que pode ser separada em duas. Por exemplo, ter uma LUT de 6 entradas que pode ser configurada como duas LUT de 5 entradas cada. A implementação é relativamente simples e consiste em ter duas LUT de 5 entradas seguidas de um multiplexer (ver Figura 2-5).

A LUT pode ser configurada como uma única LUT de 6 entradas. Para tal, basta considerar apenas a saída S1. Em alternativa, pode ser configurada como duas LUT de 5 entradas comuns. Para isso, a entrada F é fixada ao valor lógico ‘1’ e consideram-se as duas saídas S0 e S1. Esta estrutura de subdivisão de uma LUT de 6 entradas em duas LUT é considerada, por exemplo, na implementação de um somador completo que necessita de gerar duas saídas: a soma e o transporte.

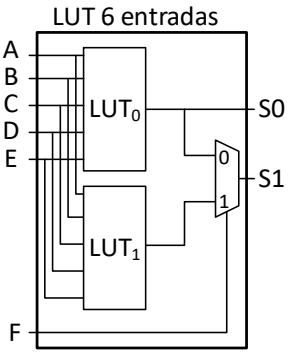


Figura 2-5 – Estrutura de uma LUT de 6 entradas configurável como duas LUT de 5 entradas.

2.1.3.3 Módulos Aritméticos

A operação de multiplicação também surge com frequência na implementação de sistemas digitais. Existem diversas implementações do multiplicador com LUT, desde uma solução iterativa com baixa utilização de LUT baseada no algoritmo de deslocamento e soma, até uma solução paralela bastante rápida, mas com um consumo elevado de LUT. Ambas as soluções têm limitações de desempenho. A solução iterativa é a mais lenta, pois demora vários períodos de relógio a executar. Por outro lado, a solução paralela, embora mais rápida, tem um caminho crítico penalizado pelas interligações entre LUT, para além de ocupar mais área.

Sendo a multiplicação uma operação frequente e procurando melhorar o seu desempenho e a área ocupada, integraram-se na FPGA multiplicadores dedicados com um tamanho fixo. Estes multiplicadores ocupam menos área do que a área das LUT necessárias à sua implementação, são mais rápidos, pois consideram lógica e interligações dedicadas, e consomem menos energia. Assim, libertam-se LUT e interligações configuráveis para implementar outras funcionalidades, enquanto se conseguem multiplicadores mais eficientes.

As operações de multiplicação-soma/subtração e de multiplicação-acumulação são também bastante comuns. Por esta razão, nas FPGA mais recentes o multiplicador dedicado evoluiu para um módulo aritmético de maior complexidade que permite realizar estas e outras operações, e é habitualmente designado DSP (*Digital Signal Processing*). Em geral, um DSP tem como elemento principal um multiplicador que se liga a somadores e a registos (ver Figura 2-6).

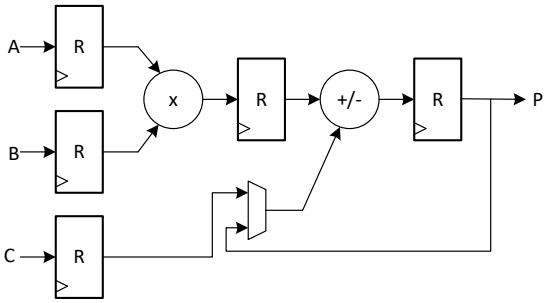


Figura 2-6 – Estrutura típica de um DSP, com registos às entradas e saídas dos módulos aritméticos

As saídas dos operadores, bem como as entradas do DSP, A, B e C, são todas registadas.

Uma característica adicional interessante dos DSP é a possibilidade de serem configurados dinamicamente, ou seja, a sua configuração pode ser alterada durante a execução do circuito de modo a realizar operações diferentes.

2.1.3.4 Módulos de Memória

Um outro módulo também bastante utilizado em sistemas digitais é a memória. As LUT permitem implementar memória, uma vez que são projetadas com base em células SRAM. Agregando várias LUT consegue-se implementar memória RAM e ROM com qualquer dimensão, limitada apenas pelos recursos disponíveis na FPGA. Para implementar memórias de muito pequena dimensão, as LUT são suficientes. No entanto, são ineficientes na implementação de memórias de média e grande dimensão devido às interligações necessárias e à utilização de uma estrutura de memória não dedicada.

Procurando melhorar a eficiência na implementação de memória, as FPGA passaram a incluir módulos de memória dedicados distribuídos pelo circuito integrado com dimensões de vários Kbits. As memórias dedicadas também podem ser interligadas para formar RAM e ROM de maiores dimensões. A utilidade destes blocos é imensa, desde ROM para a implementação de tabelas de coeficientes ou tabelas para cálculo lógico bastante rápido, até RAM para armazenar dados temporários dentro da FPGA, reduzindo assim as necessidades de acessos mais lentos a memórias externas. As FPGA atuais já comportam facilmente múltiplas memórias internas com uma capacidade total de vários Megabits.

As memórias dedicadas têm em geral a possibilidade de serem configuradas com um ou com dois portos de acesso, quer de leitura, quer de escrita. Uma memória com apenas um porto apenas permite apenas um acesso de cada vez. Por outro lado, uma memória com dois portos permite dois acessos simultâneos que, no caso da RAM, podem ser de leitura e ou de escrita. Por exemplo, podemos ter um porto de leitura e outro de escrita. Isto permite escrever dados na RAM ao mesmo tempo que são lidos dados. Podemos também ter ambos os portos de leitura ou de escrita.

Na configuração das memórias com duplo porto, os portos funcionam de forma independente, incluindo a possibilidade de ter um sinal de relógio independente para cada um (ver Figura 2-7).

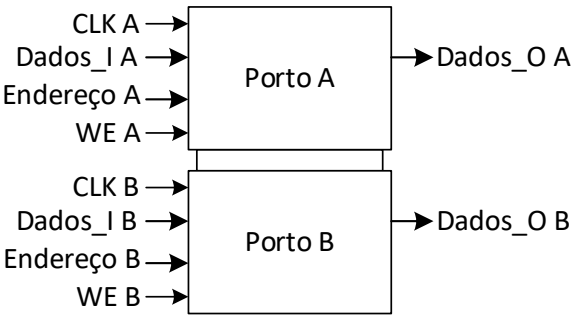


Figura 2-7 – Memória dedicada da FPGA com possibilidade de ser configurada com dois portos independentes.

Com sinais independentes de relógio, é possível, por exemplo, escrever e ler a ritmos diferentes. A sincronização no acesso aos dados é feita pelo circuito de controlo da memória.

A capacidade de armazenamento de uma memória dedicada é fixa. No entanto, tipicamente as FPGA permitem que uma memória possa ser configurada como duas memórias independentes, por exemplo como duas memórias com metade da capacidade. Além disso, é ainda possível configurar o tamanho do barramento de dados e o número de posições da memória. Por exemplo, uma memória de 32 Kbits poder ser configurada com dimensões de 4K × 8 ou 512 × 64, sem alterar o desempenho. Esta possibilidade de configuração permite melhorar a eficiência na utilização das memórias, pois é possível ajustar as dimensões da memória a cada caso em particular.

No caso de memórias com duplo porto, as configurações aplicam-se a ambos os portos, podendo inclusive ter os portos com configurações diferentes. Isto possibilita, por exemplo, escrever na memória dados com 32 bits e ler dados com 8 bits.

Em algumas FPGA, os blocos de memória dedicada podem ser configurados como estruturas *First-In-First-Out* (FIFO). As estruturas FIFO são memórias com acesso sequencial. Reduzem a complexidade da sincronização e da geração de endereços, tornando o processo de manipulação de endereços automático. Sempre que o acesso aos dados for feito de forma sequencial, é possível utilizar uma FIFO em vez de uma RAM. Estas estruturas de memória são, assim, bastante comuns no projeto de sistemas digitais e, por isso, alguns fabricantes incluem o suporte para configuração dos módulos de memória dedicada das FPGA como RAM, ROM ou FIFO.

Apesar de todas as vantagens na utilização dos módulos de memória dedicada para implementar RAM, ROM ou FIFO, para memórias mais pequenas é preferível utilizar LUT. A LUT é o bloco mais rápido da FPGA e está disponível por toda a FPGA. É assim o método mais eficiente de implementação de memórias de pequena dimensão.

Em algumas FPGA a LUT também pode ser configurada como um registo de deslocamento, que por sua vez pode ser utilizado de forma bastante eficiente na implementação de FIFO de pequena dimensão.

2.1.3.5 Módulos Dedicados Adicionais

As FPGA atuais têm outros blocos dedicados, para além dos referidos nas secções anteriores, que permitem aumentar a densidade computacional, a eficiência do dispositivo, gerar sinais de relógio com determinadas frequências e fases, aceder a memória externa e comunicar com o exterior. Entre os blocos dedicados adicionais mais comuns destacam-se os seguintes:

- PLL (*Phase-Lock Loop*) – Bloco utilizado para gerar sinais de relógio a partir de um sinal de relógio fonte. O PLL permite dividir ou multiplicar a frequência do relógio origem e assim ajustar a frequência do circuito de acordo com o seu caminho crítico. A FPGA pode conter mais do que um PLL, o que permite implementar circuitos digitais com módulos a funcionar a frequências diferentes;
- CME (Controlador de Memória Externa) – Bloco utilizado para aceder à memória externa. A utilização de um bloco dedicado para acesso à memória externa garante um desempenho mais elevado comparado com o que seria obtido se o controlador fosse implementado com LUT. Algumas FPGA contêm mais do que um controlador de memória e a possibilidade de configurar o protocolo de acesso em função do tipo de memória externa;
- TS (*Transceiver* Série) – Módulo utilizado nas comunicações série de alto débito. Surgem tipicamente em FPGA dedicadas à implementação de circuitos de comunicação de dados com elevado débito.

A distribuição dos blocos ao longo do circuito integrado é feita de forma regular. Algumas colocações de blocos estão interrelacionadas. Por exemplo, um bloco de memória dedicada está geralmente associado a um bloco DSP. Garante assim o acesso rápido de um DSP a dados contidos em memória dedicada.

Todos os blocos estão rodeados das interligações configuráveis, permitindo a interligação entre quaisquer tipos de blocos. No seu conjunto, todos estes blocos permitem implementar em FPGA qualquer circuito digital.

2.1.4 Processador Dedicado

A densidade das FPGA tem aumentado, o que permite projetar circuitos cada vez maiores, incluindo processadores genéricos. Como qualquer outro sistema digital, um processador pode ser implementado em lógica programável. Existem vários exemplos de núcleos de

processador disponíveis para serem implementados em FPGA: o MicroBlaze da AMD-Xilinx, o NIOS da Intel, além de outros processadores de código aberto que podem igualmente ser integrados na FPGA.

A existência de um ou mais processadores em FPGA permite o projeto de sistemas num único circuito integrado (SoC – *Systems-on-Chip*). Passa a ser possível projetar um sistema digital com um ou mais processadores, com memória e com módulos de hardware dedicados para acelerar determinadas funções tornando a sua execução mais eficiente do que seria possível apenas com o processador. Esta solução possibilita desenvolver sistemas hardware/software autónomos sem necessidade de ter um computador ligado à FPGA para controlar o acesso ao hardware.

De modo a melhorar as características do processador, surgiram FPGA com processadores integrados. As primeiras soluções implementavam um ou mais processadores juntamente com a lógica configurável, como um qualquer bloco de hardware dedicado a par com um DSP ou com uma memória dedicada. Estes processadores dedicados não ofereciam configurabilidade, mas foram um primeiro passo para o desenvolvimento de SoC em FPGA.

Apesar das possibilidades de projeto de hardware/software que este tipo de FPGA oferecia, o seu sucesso foi limitado, em parte, devido à complexidade do projeto de todo o sistema de processamento em torno do processador e da interligação entre o processador e o hardware. Não nos podemos esquecer que um processador precisa de um barramento para acesso à memória de dados e de instruções, podendo esta ser memória interna da FPGA ou memória externa. Para aceder à memória externa é necessário um controlador de memória. Adicionalmente, para melhorar a execução do software é importante incluir memória *cache*, bem como unidades de cálculo em vírgula flutuante. O projeto do sistema em torno do processador tinha de ser feito com lógica programável, o que tornava o projeto bastante complexo.

Como forma de reduzir a complexidade do projeto, surgiram as SoC FPGA que, como o nome indica, integram um ou mais processadores, mas com uma arquitetura orientada ao processador. Isto significa que as novas SoC FPGA integram um sistema de processamento completo separado da lógica reconfigurável e com interfaces também dedicadas para ligar à zona reconfigurável da FPGA. As SoC FPGA passam a considerar o sistema de processamento como o núcleo do sistema, que pode aceder a módulos hardware implementados na zona de lógica programável como sendo periféricos. A grande vantagem relativamente às soluções anteriores é que todo o sistema de processamento é implementado de forma integrada na SoC FPGA. Não só melhora o desempenho do sistema de processamento, como simplifica o seu projeto.

2.2 Arquiteturas FPGA Comerciais

Existem vários fabricantes de FPGA, como a Intel, a AMD-Xilinx, a Lattice Semiconductor e a Microchip Technology. Todos consideram uma organização da lógica reconfigurável com as estruturas genéricas descritas na secção anterior, mas com algumas variações que distinguem as arquiteturas de cada um dos fabricantes. Nesta secção, apresentamos uma descrição breve da arquitetura da família de SoC FPGA Ultrascale+ do fabricante AMD-Xilinx, de que faz parte a FPGA considerada nas implementações apresentadas neste livro.

2.2.1 FPGA da Família Ultrascale+ da AMD-Xilinx

As FPGA da AMD-Xilinx estão organizadas por famílias dentro de determinada tecnologia de fabrico. A Ultrascale+ utiliza as tecnologias de fabrico mais recentes e considera diferentes famílias de FPGA que variam na relação custo, desempenho, recursos de hardware e consumo. Em qualquer das famílias a arquitetura lógica da FPGA utiliza os mesmos componentes, nomeadamente:

- Lógica configurável: LUT de 6 entradas (LUT-6);
- Memória: Blocos de memória (BRAM) de 36Kb de duplo porto configurável como duas memórias independentes de duplo porto de 18 Kb. Blocos de memória (UltraRAM) de 288 Kb de duplo porto;
- Blocos Aritméticos: Blocos DSP (DSP48E2) com multiplicador 27 × 18, somador/acumulador de 48 bits e pré-somador de 27 bits.

Nas secções seguintes, descrevem-se com mais detalhe cada um destes blocos.

2.2.1.1 Bloco Lógico Configurável

A lógica reconfigurável da FPGA AMD-Xilinx está organizada em blocos lógicos configuráveis (CLB). Um CLB contém uma estrutura designada *slice*, que por sua vez contém oito LUT-6 (LUT de 6 entradas) e dezasseis *flip-flop*, dois por cada LUT-6, organizados por colunas. Uma função que não pode ser implementada apenas com uma LUT-6 é facilmente implementada com a interligação de várias LUT-6.

A LUT-6 pode ser configurada com apenas uma saída e 6 entradas ou com duas saídas, mas com 5 entradas comuns, idêntico ao descrito na secção 2.1.3.2. A saída da LUT-6 pode ser ligada diretamente à saída do *slice* (lógica combinatória) ou pode ser, opcionalmente, registada com um flip-flop ou *latch* (lógica sequencial). A existência de dois *flip-flop* por LUT-6 permite que ambas as saídas da LUT-6 (configuração com duas saídas e cinco entradas) sejam registadas (ver Figura 2-8).

Cada um dos *flip-flop* associados a uma LUT-6 pode receber uma das duas saídas da LUT-6 ou uma entrada direta do CLB sem passar pela LUT-6. Ambas as saídas da LUT-6 podem passar ou não pelos *flip-flop*.

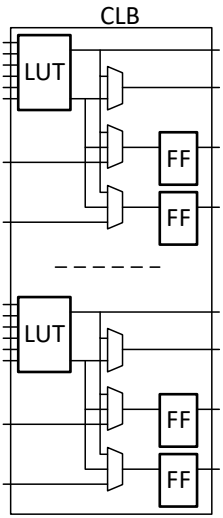


Figura 2-8 – Estrutura genérica do CLB.

Também, como genericamente referido nas secções anteriores, a CLB contém linhas dedicadas de propagação dos sinais de transporte que melhoram consideravelmente a implementação de operadores aritméticos, como somador, contador e multiplicador. As linhas dedicadas de transporte percorrem a FPGA na vertical.

Com o objetivo de otimizar a utilização dos *slices*, consideram-se dois tipos de *slices*: SLICEL (lógica) e SLICEM (memória). O SLICEL contém as LUT-6, os *flip-flop* e a lógica de propagação do sinal de transporte descritos anteriormente. O SLICEM oferece a possibilidade adicional de configurar as LUT-6 como memória RAM distribuída de 64-bit, com a adição de sinais de controlo de escrita, de leitura e de sinal de relógio. Como as LUT-6 podem ser configuradas com uma ou duas saídas, uma LUT-6 do SLICEM pode ser configurada como memória RAM de 64 × 1 ou 32 × 2. Juntando as 8 LUT-6 de um SLICEM, obtém-se uma memória RAM de 512 bits. O SLICEM também pode ser configurado como um registo de deslocamento de 32 bits. Combinando as 8 LUT-6 consegue-se implementar um registo de deslocamento de 256-bit em um SLICEM.

2.2.1.2 Bloco Aritmético

Os blocos aritméticos dedicados (DSP48E2) contêm, entre outros operadores, um multiplicador de 27×18 bits seguido de um somador/acumulador de 48 bits (ver Figura 2-9).

A figura apresenta genericamente os módulos e as interligações principais do DSP48E2. O bloco permite implementar diversas funções, nomeadamente multiplicação, multiplicação-acumulação, multiplicação-soma, quatro somadores independentes, registo de deslocamento, comparador, operações lógicas bit a bit, detetor de padrões e contador.

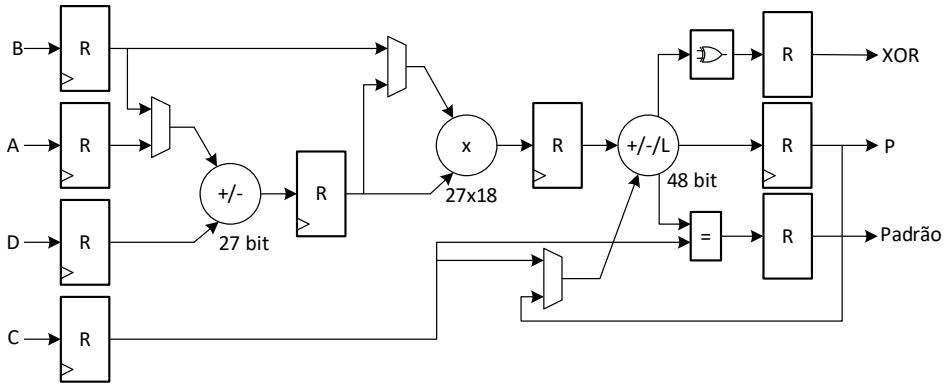


Figura 2-9 – Estrutura do bloco aritmético DSP48E2.

O DSP é utilizado frequentemente na realização de multiplicações. No entanto, também pode ser utilizado apenas como somador ou como acumulador, bem como qualquer uma das outras operações descritas anteriormente, sem necessidade de utilizar o multiplicador. De modo a reduzir o consumo, é possível desligar o multiplicador quando não está a ser utilizado.

2.2.1.3 Blocos de Memória

O bloco de memória RAM síncrona dedicada (BRAM) tem uma capacidade de armazenamento de 36 Kbits e pode ser configurado com diversos modos de funcionamento.

As BRAM não são apenas utilizadas para implementar RAM, mas também memórias do tipo FIFO (*First-In First-Out*). As FIFO podem ser implementadas de forma bastante eficiente com memória BRAM, uma vez que estas incluem lógica adicional para a implementação da FIFO, sem necessidade de recorrer a LUT. A configurabilidade da FIFO segue a da BRAM, ou seja, os portos de leitura e de escrita da FIFO podem operar a frequências diferentes e podem ter configurações diferentes. Por exemplo, é possível escrever na FIFO com um barramento de dados de 8 bits e ler dados a 32 bits.

De modo a implementar FIFO de maior dimensão do que a suportada por uma única BRAM, podem ser interligadas várias BRAM.

Estes dispositivos incluem adicionalmente blocos de memória dedicada com uma capacidade de 288 Kbits (UltraRAM), podendo estes também ser interligados entre si para formar memórias de maior capacidade. A UltraRAM é uma memória RAM síncrona de duplo porto com uma configuração fixa de 4096×72. Ambos os portos operam à mesma frequência, mas permitem um acesso para escrita ou leitura independente.

2.3 Mapeamento dos Circuitos Digitais em FPGA

O projeto de circuitos digitais com síntese de alto nível tendo como alvo a tecnologia FPGA produz uma solução hardware com os recursos configuráveis disponíveis na FPGA. Para além de gerar o circuito hardware, também produz um relatório com uma estimativa de ocupação de recursos (LUT, FF, DSP, BRAM, entre outros) e de desempenho. As estimativas de ocupação de recursos permitem determinar se a FPGA alvo tem hardware suficiente para implementar o circuito.

O modo como a funcionalidade do circuito é descrita determina os recursos necessários à sua implementação. Por este motivo, a estimativa de recursos também nos permite aferir se estamos a utilizar recursos em excesso, tendo em conta o circuito esperado. É por isso importante saber de que forma são mapeadas algumas das operações mais comuns em hardware, nomeadamente as operações aritméticas básicas (soma, subtração e multiplicação), contadores, registos e memórias.

2.3.1 Mapeamento de Somadores e Subtratores

As operações básicas de soma e de subtração podem ser implementadas com LUT ou com DSP. A implementação básica com LUT através de somadores completos encadeados é, em geral, a mais eficiente devido às linhas dedicadas de transporte. Um somador completo de 1-bit é implementado com uma LUT-6 configurada com duas saídas, como referido anteriormente. Um somador de N-bits é implementado com N somadores completos de 1-bit e ocupa N LUT. Existe deste modo uma relação linear entre o tamanho do somador e o número de LUT.

A subtração é implementada da mesma forma pois utiliza um somador para adicionar o simétrico do operando a subtrair (A + (-B)). Pela mesma razão, o número de LUT necessárias à implementação de um subtrator é proporcional ao número de bits do somador. Também é possível implementar um somador/subtrator com o mesmo número de recursos, bastando uma entrada adicional de seleção do tipo de operação. Note-se que as LUT do somador completo apenas estão a usar duas das cinco entradas disponíveis, sendo possível ocupar mais uma entrada com o sinal de seleção da operação.

As somas e as subtrações também podem ser implementadas com DSP, uma vez que tem disponível um somador de 48 bits que pode ser facilmente estendido para mais bits interligando vários DSP.

Havendo duas opções de implementação, coloca-se a questão de saber se a ferramenta de HLS gera o somador/subtrator com LUT ou com DSP. Testou-se a síntese de um somador no *Vitis HLS* com operandos de dimensão entre 8 e 128 bits e em todos os casos a ferramenta utilizou LUT. Para dimensões maiores, a ferramenta optou, por omissão, por realizar a soma iterativamente, ou seja, o mesmo somador é utilizado várias vezes para somar parcelas da soma. A opção por utilizar LUT resulta do facto de ser uma implementação bastante eficiente em termos de área e de desempenho comparável aos DSP.

2.3.2 Mapeamento de Registos

Os registos são implementados com flip-flop. Para implementar um registo de N-bits são necessários N flip-flop. Como uma LUT tem disponíveis à saída dois flip-flop, são necessários os flip-flop de N/2 LUT. Ao utilizar os flip-flop, as LUT não ficam necessariamente inutilizadas, uma vez que é possível utilizar simultaneamente as LUT e os seus flip-flop de forma independente, desde que não seja necessário registar a saída da LUT.

Também é possível implementar registos com DSP, uma vez que os DSP contêm registos às entradas, às saídas e em pontos intermédios. Nos casos em que uma determinada operação aritmética com registos à entrada e à saída é mapeada em DSP, os registos de entrada e de saída da operação são implementados com os existentes no DSP, não sendo necessário consumir flip-flop adicionais. A ferramenta de síntese de alto nível realiza esta otimização automaticamente.

2.3.3 Mapeamento de Contadores

O somador e o subtrator, juntamente com um registo, também são utilizados frequentemente na implementação de contadores. Os contadores são utilizados, por exemplo, para a implementação da variável de controlo de uma estrutura de repetição. Um contador crescente ou decrescente é facilmente implementado com um somador ou um subtrator, respetivamente (ver Figura 2-10).

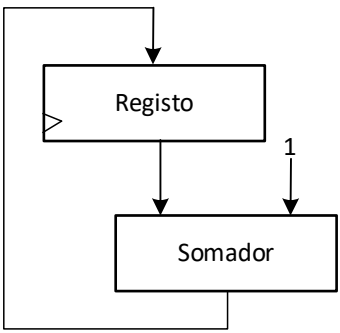


Figura 2-10 – Implementação de um contador com LUT.

O exemplo da figura ilustra um contador crescente simplificado implementado com um registo e com um somador. Neste caso, o contador incrementa um em cada soma. Qualquer outro incremento é possível, bastando alterar a constante presente na segunda entrada do somador.

Os recursos utilizados na implementação de um contador são assim os mesmos utilizados na implementação de um somador, pois o registo é implementado com os flip-flop disponíveis às saídas das LUT utilizadas na implementação da soma.

2.3.4 Mapeamento de Multiplicadores

A operação de multiplicação pode ser realizada com LUT, com DSP ou até com uma mistura dos dois tipos de recursos, dependendo do tamanho do multiplicador.

A implementação típica com LUT baseia-se no algoritmo de geração e acumulação de produtos parciais. Na multiplicação binária, um produto parcial corresponde à multiplicação de um bit (0 ou 1) do multiplicador (A) pelo multiplicando (B). O resultado é 0 quando multiplicado por '0' (B×0) ou o próprio multiplicando, quando multiplicado por '1' (B×1). Um produto parcial de N bits pode assim ser facilmente gerado com N LUT.

O produto P = A × B, ambos de N bits, calcula N produtos parciais que têm de ser somados (ver exemplo de um multiplicador de 4×4 implementado com LUT na Figura 2-11).

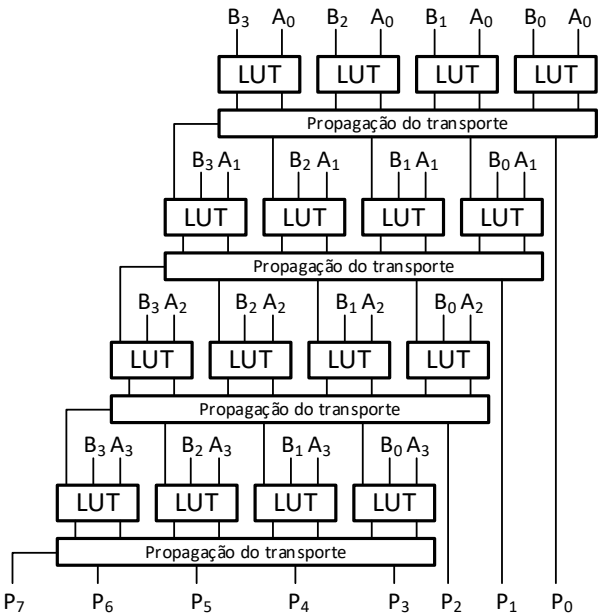


Figura 2-11 – Implementação de um multiplicador de 4 bits com LUT.

Como se pode observar na figura, uma LUT é utilizada para calcular o produto entre dois bits, de A e B, e somar com a saída do parcial anterior. Uma LUT é assim utilizada no cálculo de um produto-soma de 1 bit.

Com base nesta arquitetura, o número de LUT necessárias para implementar o produto entre dois operandos de N bits é teoricamente N^2. Dada a complexidade quadrática na ocupação de recursos da operação de multiplicação, a ferramenta de HLS considera quase sempre uma implementação com DSP, exceto para multiplicações de dimensão reduzida.

A implementação da multiplicação com DSP é o método utilizado, por omissão, pela ferramenta de síntese. O número de DSP necessários depende do tamanho do multiplicador pretendido. Sabendo que o tamanho do multiplicador do DSP é de 27×18 (inteiros), os multiplicadores são implementados em parcelas. Por exemplo, na multiplicação P = A × B, em que A e B são operandos de 35 bits, são necessários quatro DSP com a organização ilustrada na Figura 2-12.

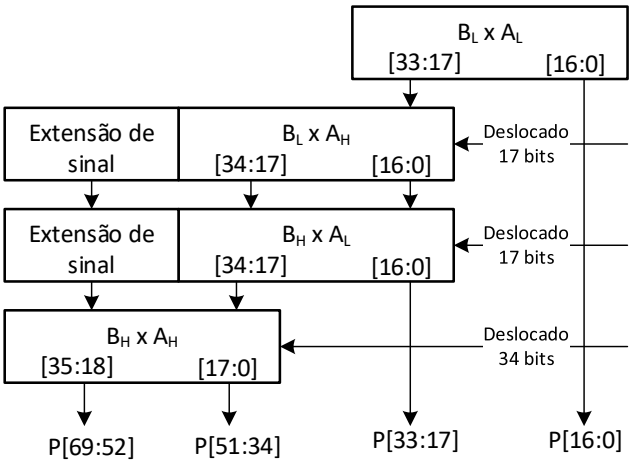


Figura 2-12 – Implementação de um multiplicador de 35 bits com DSP.

Os operandos A e B são subdivididos em A_H ([34:17]), A_L ([16:0]) e B_H ([34:17]), B_L ([16:0]). O produto P = A × B é calculado como:

$$P = (A_H \times 2^{17} + A_L) \times (B_H \times 2^{17} + B_L) =$$

$$(A_H \times B_H) \times 2^{34} + (A_H \times B_L + A_L \times B_H) \times 2^{17} + (A_L \times B_L)$$

No total, são necessárias 4 multiplicações resultando em 4 DSP. Os deslocamentos de 17 bits são implementados com interligações dedicadas entre DSP para esse efeito.

2.3.5 Mapeamento de Memórias

As memórias podem ser implementadas em FPGA com LUT (memória distribuída) ou com BRAM (memória dedicada). O tamanho da memória determina o tipo de recurso utilizado na sua implementação. Em geral, memórias de muito pequena dimensão são implementadas com LUT e as restantes com BRAM.

No caso de memória implementada com LUT, uma LUT implementa uma memória de 64×1 ou 32×2. Memórias maiores são implementadas com várias LUT. Por exemplo, uma memória de 128×2 necessita de 4 LUT (ver Figura 2-13).

Cada um dos bits de dados é gerado à custa de duas LUT mais um *multiplexer* dedicado. O método pode ser facilmente estendido para memórias de maior dimensão, quer em número de endereços, quer em tamanho da palavra de dados.

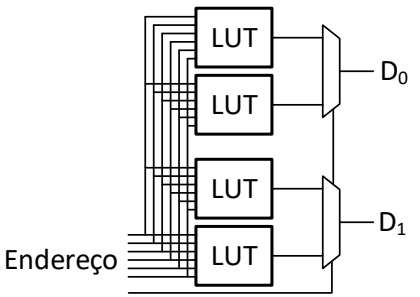


Figura 2-13 – Implementação de uma memória de 128×2 com LUT.

No caso da implementação de memória com BRAM, é possível estimar o número de BRAM necessárias tendo em conta o tamanho de cada BRAM (36 Kbits) e as configurações possíveis:

- BRAM 36Kbit: 32K × 1, 16K × 2, 8K × 4, 4K × 9, 2K × 18, 1K × 36, 512 × 72 (porto simples)
- BRAM 18Kbit: 16K × 1, 8K × 2, 4K × 4, 2K × 9, 1K × 18, 512 × 36 (porto simples)

Assim, basta adicionar tantas BRAM quantas as necessárias para perfazer a memória necessária. Por exemplo, para realizar uma memória de 64K × 8, podemos utilizar 16 BRAM com a configuração 4K × 9 (16 × 4K × 9 = 64K × 9). Com esta configuração sobra um bit de dados em cada posição.

O tamanho dos dados está limitado a 72 bits (porto simples) e a 36 bits (duplo porto). Quando é necessária uma memória com um tamanho de dados superior, são utilizadas várias memórias, cada uma armazenando uma parte da palavra de dados. Consideremos, como exemplo, uma memória de porto simples com capacidade 128 × 256 = 32 Kbits. Apesar de uma única BRAM conter capacidade suficiente para guardar 32 Kbits, são necessárias 4 BRAM para a sua implementação devido ao limite do tamanho do barramento de dados (72 bits no máximo). Repare-se que uma memória de 128 × 216 é implementada com apenas 3 BRAM (216 = 72 × 3).

Uma BRAM pode ser configurada com um ou dois portos. Nos casos em que são necessários mais do que dois portos, é necessário replicar as memórias, ou seja, incluir duas memórias com o mesmo conteúdo. A escrita dos dados é feita em simultâneo para as duas e a leitura é feita de forma independente com dois portos de cada memória.

3 Circuitos Hardware: Conceitos e Arquiteturas

O desenvolvimento de circuitos hardware com base em síntese de alto nível eleva o nível de abstração com que estes são descritos. Contudo, é necessário ter presente os conceitos e os modelos de computação em hardware durante a descrição dos circuitos de modo a gerar as melhores soluções e para poder interpretar os resultados da ferramenta de síntese.

Neste capítulo são abordados os principais conceitos, métodos e modelos de computação associados ao hardware. Procura-se, deste modo, criar uma base de conhecimento hardware que permita mais facilmente entender o projeto de hardware com ferramentas de síntese de alto nível.

3.1 Computação com Hardware Dedicado

Ao contrário de um processador que tem uma arquitetura de computação fixa, o projeto de hardware permite elaborar uma arquitetura de computação ajustada à funcionalidade que se pretende executar. Esta flexibilidade leva a que existam várias arquiteturas para o mesmo problema, diferenciando-se no tempo de execução, nos recursos utilizados, no consumo de energia, etc.

Para exemplificarmos esta ideia, consideremos o produto vetorial, `PV`, entre dois vetores de dimensão 16, `A` = $a_{15}a_{14}\ldots a_0$ e `B` = $b_{15}b_{14}\ldots b_0$, de acordo com a seguinte equação:

$$\text{PV} = \sum_{i=0}^{15} a_i \times b_i = a_0 \times b_0 + a_1 \times b_1 + \cdots + a_{15} \times b_{15}$$

Se executarmos o produto vetorial num processador genérico com apenas uma unidade aritmética que executa uma operação por ciclo de relógio, irá demorar 16 ciclos de relógio para executar as 16 multiplicações mais 16 ciclos de relógio para executar as acumulações dos produtos, com um total de 32 ciclos.

O mesmo problema implementado em hardware tem várias soluções resultantes do facto de poderem ser executadas várias operações em paralelo, ou seja, no mesmo ciclo de relógio. Consideremos uma solução em hardware dedicado com dois multiplicadores e dois somadores (ver Figura 3-1).

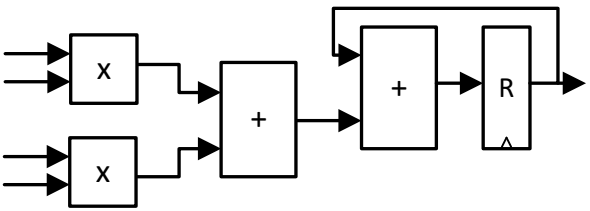


Figura 3-1 – Solução de hardware dedicado para o produto vetorial

Neste caso, a unidade realiza duas multiplicações e a acumulação da soma dos dois produtos num único ciclo de relógio. No total, irá demorar 8 ciclos de relógio para realizar o produto vetorial. A solução não é única, pois podemos considerar mais unidades de multiplicação e de soma em paralelo. No limite, podemos realizar o produto vetorial num único ciclo de relógio. Comparando as soluções em termos de número de ciclos de relógio e da quantidade de recursos, facilmente concluímos que a execução em menos ciclos implica a utilização de mais recursos.

O projeto de circuitos com hardware dedicado envolve assim uma nova dimensão de projeto, pois é necessário projetar o circuito hardware onde serão executadas todas as operações do algoritmo. Existe um espaço de projeto com várias soluções possíveis para um determinado problema que o projetista pode explorar. Para o orientar na procura da melhor solução, utiliza várias métricas, como desempenho, área, consumo, etc. Na secção seguinte, descrevemos um conjunto de métricas associadas ao desempenho.

3.2 Métricas de Desempenho

Um dos objetivos mais comuns no projeto de qualquer circuito de computação é otimizar o seu desempenho. Existem várias métricas de desempenho: o tempo de execução e frequência, a latência e a taxa de produção. Nas secções seguintes, descrevem-se cada uma destas métricas.

3.2.1 Tempo de Execução e Frequência

O número de ciclos de relógio de execução de um algoritmo numa determinada plataforma de computação não permite por si só identificar qual a solução mais rápida. É necessário juntar a frequência do sinal de relógio para poder determinar o tempo de execução. O tempo de execução, *Texec*, é determinado pelo produto entre o número de ciclos de relógio, *Nciclos*, e o período de relógio, *Tclk* (duração de cada ciclo de relógio), ou seja:

$$Texec = Nciclos \times Tclk$$

O período de relógio é determinado pelo caminho crítico do circuito, que corresponde ao maior atraso de propagação da lógica entre dois elementos de memória. Quanto maior for

o caminho crítico, maior será o período de relógio necessário e, consequentemente, menor será a frequência.

Consideremos de novo o exemplo do produto vetorial entre vetores de 16 elementos. Para a sua execução, iremos analisar três arquiteturas computacionais dedicadas e uma genérica baseada num processador:

- Arquitetura 1: Arquitetura dedicada com apenas um multiplicador e um somador (ver Figura 3-2a);
- Arquitetura 2: Arquitetura dedicada idêntica à apresentada na Figura 3-1, com dois multiplicadores e dois somadores;
- Arquitetura 3: Arquitetura dedicada com quatro multiplicadores e quatro somadores (ver Figura 3-2b);
- Arquitetura 4: Processador com capacidade de cálculo de 1 operação por ciclo de relógio com uma frequência de 1 GHz.

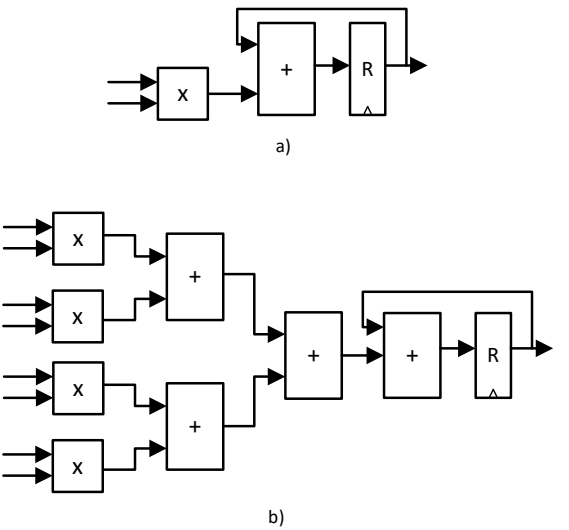


Figura 3-2 – Arquiteturas dedicadas para o cálculo vetorial. a) com apenas um multiplicador e um somador, b) com quatro multiplicadores e quatro somadores.

Para o cálculo do caminho crítico, consideremos que o multiplicador das arquiteturas dedicadas tem um atraso de 3 ns e que o somador tem um atraso de 1 ns. Para simplificar a análise, consideremos apenas os tempos de atraso do multiplicador e do somador, desprezando os tempos de atraso dos registos. No caso da arquitetura 1, o atraso total é igual à soma do atraso do multiplicador com o do somador, ou seja, 3 + 1 = 4 ns. Na arquitetura 2, o atraso é igual ao do multiplicador mais o dos dois somadores, um total de

3 + 1 + 1 = 5 ns e, na arquitetura 3, o atraso é dado pelo de um multiplicador mais o de 3 somadores, ou seja 3 + 1 + 1 + 1 = 6 ns.

Em termos de ciclos de relógio, a arquitetura 1 demora 16 ciclos (uma multiplicação-acumulação por ciclo), a arquitetura 2 demora 8 ciclos (2 multiplicações e acumulação das somas por ciclo) e a arquitetura 3 demora 4 ciclos (4 multiplicações e acumulação das somas por ciclo). Assumindo um período de relógio igual ao atraso máximo de cada uma das soluções, o tempo total de execução do produto vetorial, em cada uma, seria:

Arquitetura 1: 4 ns × 16 = 64 ns (frequência máxima de relógio = 250 MHz)

Arquitetura 2: 5 ns × 8 = 40 ns (frequência máxima de relógio = 200 MHz)

Arquitetura 3: 6 ns × 4 = 24 ns (frequência máxima de relógio ≈ 167 MHz)

A arquitetura 3 é a mais rápida, apesar de operar a uma frequência menor, mas necessita de mais recursos de cálculo.

No caso da arquitetura 4, o período de relógio é igual a 1 ns. Sabendo que demora 32 ciclos de relógio para executar o produto vetorial, o tempo de execução é:

Arquitetura 4: 1 ns * 32 = 32 ns

Como se pode concluir pelo exemplo, a frequência por si só não determina qual o circuito mais rápido. Basta verificar que a arquitetura com a menor frequência é a que executa o produto vetorial mais rápido. Para tal, foi necessário utilizar quatro multiplicadores em paralelo juntamente com quatro somadores, de modo a reduzir o número de períodos de relógio.

O exemplo também permite observar que as arquiteturas dedicadas 1 e 2 são mais lentas do que o processador genérico. Foi necessário aumentar o número de unidades paralelas para que se conseguisse obter uma arquitetura dedicada mais rápida do que o processador.

As arquiteturas dedicadas em hardware podem executar a frequências mais baixas e conseguir melhores desempenhos com a exploração da execução paralela de operações. Uma das vantagens em executar com uma frequência mais baixa é a de redução da potência que é necessária fornecer ao circuito.

3.2.2 Latência

Uma função em hardware pode ser executada num único ciclo de relógio ou em vários ciclos seguindo uma determinada sequência de operações. O tempo necessário para executar as operações designa-se *latência*. A latência pode ser expressa como um intervalo de tempo ou como um número de ciclos de relógio.

No exemplo do produto vetorial considerado na secção anterior, as quatro arquiteturas tinham latências diferentes, de acordo com o número de ciclos de relógio necessários para

o cálculo do produto vetorial. Assim, a latência de cada uma das arquiteturas no cálculo do produto vetorial entre vetores de 16 elementos é:

Arquitetura 1: 16 ciclos de relógio

Arquitetura 2: 8 ciclos de relógio

Arquitetura 3: 4 ciclos de relógio

Arquitetura 4: 32 ciclos de relógio

No caso de termos de realizar vários produtos vetoriais seguidos, a latência total é obtida pela latência de um produto vetorial multiplicado pelo número de produtos vetoriais, se considerarmos que as arquiteturas apresentadas apenas iniciam o cálculo de um novo produto vetorial após terminar o anterior. Neste cenário, o projetista da arquitetura procura reduzir o número de ciclos de relógio e aumentar a frequência de modo a conseguir a melhor latência.

3.2.3 Taxa de Produção

A taxa de produção corresponde ao número de ações executadas por unidade de tempo ou quantidade de resultados produzidos por unidade de tempo. A unidade da taxa de produção é dados/segundo ou dados/ciclos de relógio. A taxa de produção de dados pode ser, naturalmente, de um resultado por vários ciclos de relógio. Isto acontece, por exemplo, quando o circuito necessita de vários ciclos de relógio para processar os dados e não admite novos dados enquanto não terminar o processamento dos anteriores.

Os exemplos que vimos na secção anterior têm taxas de produção de dados diferentes. Por exemplo, a arquitetura 1 produz um novo produto vetorial a cada 16 ciclos de relógio, enquanto a arquitetura 2 produz um produto vetorial a cada 8 ciclos.

Esta forma de caracterizar a produção de dados é relativa ao período de relógio e, consequentemente, à frequência de operação. Circuitos com a mesma taxa de produção por ciclo de relógio podem ter desempenhos diferentes, dependendo da frequência a que operam. Assim, a unidade dados/ciclo é utilizada para comparar circuitos que operam à mesma frequência.

Para sabermos qual o circuito com melhor desempenho, temos de ter em conta a taxa de produção de dados e a frequência de operação. Neste caso, a unidade dados/segundo permite identificar de forma absoluta qual o circuito com melhor desempenho. Por exemplo, consideremos dois circuitos com a mesma taxa de produção por ciclo de relógio, mas com frequências de 100 e 50 MHz. Neste caso, um dos circuitos tem o dobro do desempenho relativamente ao outro, pois opera ao dobro da frequência, ou seja, tem uma taxa de produção de dados por segundo duas vezes maior.

Uma métrica similar é à taxa de consumo de dados que corresponde ao número vezes que os dados de entrada são consumidos por unidade de tempo.

3.3 Técnicas de otimização de Hardware

Existem vários métodos de otimização do desempenho de uma arquitetura hardware. Nesta secção, descrevem-se duas das mais comuns: *pipeline* e paralelização.

3.3.1 Pipeline

O caminho crítico de um circuito determina a sua frequência de operação. Quanto mais operações em sequência forem realizadas num ciclo de relógio, menor será a latência do circuito, mas mais longo será o caminho crítico, com a consequente redução da frequência de operação. Temos um compromisso entre latência e frequência, como visto nos circuitos do exemplo anterior de cálculo do produto interno.

Há, no entanto, uma técnica que permite reduzir o caminho crítico sem reduzir o número de operações que são realizadas por ciclo de relógio. A técnica designada *pipeline* tira proveito da execução repetida de operações sobre os operandos de entrada. Esta técnica permite que um novo cálculo possa ser iniciado sem que o anterior tenha terminado.

Consideremos de novo o produto vetorial, em que é realizada uma multiplicação seguida de uma soma para cada par de elementos dos vetores. O que a técnica de *pipeline* faz é permitir que se inicie uma nova multiplicação ao mesmo tempo que se executa a acumulação do produto anterior. Para tal, utiliza-se um registo intermédio para armazenar a saída do multiplicador (ver Figura 3-3).

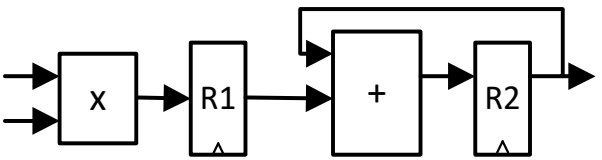


Figura 3-3 – Cálculo do produto vetorial com a arquitetura 1 em pipeline

O registo R1, à saída do multiplicador, permite armazenar o produto que é acumulado no registo R2 ao mesmo tempo que o multiplicador calcula o novo produto. A cada ciclo de relógio, a arquitetura armazena um novo produto em R1 e uma acumulação em R2. Considerando os vetores $\mathtt{A} = a_{15}a_{14}\ldots a_1a_0$ e $\mathtt{B} = b_{15}b_{14}\ldots b_1b_0$, teríamos o escalonamento da Figura 3-4.

O escalonamento determina a distribuição das operações realizadas em cada ciclo do sinal de relógio.

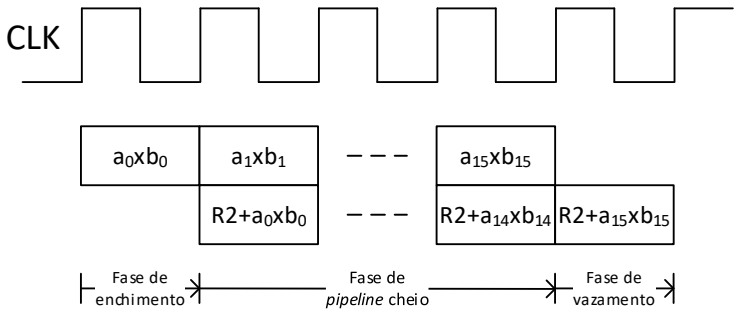


Figura 3-4 – Escalonamento do cálculo do produto vetorial com a arquitetura em pipeline

Como se pode observar, em cada ciclo de relógio são realizadas duas operações, uma de multiplicação e outra de soma. No total, são usados 17 ciclos de relógio para executar o produto vetorial, ou seja, a latência é de 17 ciclos, mais um ciclo de relógio que a arquitetura 1 sem *pipeline*. No entanto, o caminho crítico da arquitetura com *pipeline* é determinado pelo operador com maior atraso, o multiplicador ou o somador, ao contrário da arquitetura sem *pipeline*, em que o caminho crítico era determinado pela soma dos atrasos dos dois operadores.

Se calcularmos a latência das duas arquiteturas, temos que:

Arquitetura 1 sem *pipeline*: 16 × 4 ns = 64 ns

Arquitetura 1 com *pipeline*: 17 × max(3, 1) ns = 51 ns

A técnica de *pipeline* pode ser aplicada do mesmo modo às restantes arquiteturas, com registos à saída dos multiplicadores, fazendo reduzir o atraso no caminho crítico para apenas 3 ns. No caso das arquiteturas 2 e 3, teríamos os seguintes valores de latência:

Arquitetura 2 com *pipeline*: 9 × 3 ns = 27 ns

Arquitetura 3 com *pipeline*: 5 × 3 ns = 15 ns

Na técnica de *pipeline* identificam-se três fases, que podem ser observadas no escalonamento da Figura 3-4:

- Fase de enchimento: Fase inicial da *pipeline* em que as unidades do caminho de dados vão progressivamente recebendo os dados iniciais. Nesta fase, as unidades ainda não estão todas a processar dados. No exemplo, a fase de enchimento ocorre no primeiro ciclo de relógio, em que apenas o multiplicador está a executar;
- Fase de *pipeline* cheio: Fase da *pipeline* em que todas as unidades estão a processar dados. No exemplo, são todos os ciclos de relógio em que o multiplicador e o somador estão a executar em paralelo;

– Fase de vazamento: Fase da *pipeline* em que as unidades começam sucessivamente a parar o seu processamento, da entrada para a saída. No exemplo, ocorre no último ciclo de relógio, em que o multiplicador já não processa dados e o somador está ainda a processar o produto gerado no ciclo de relógio anterior.

O número de ciclos de relógio necessários nas fases de enchimento e de vazamento da *pipeline* aumenta com o número de níveis de *pipeline*.

A técnica de *pipeline* pode ser aplicada genericamente a qualquer caminho de dados com várias operações. Consideremos, genericamente, o caminho de dados de um circuito sem *pipeline* representado na Figura 3-5.

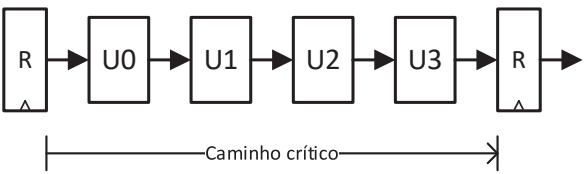


Figura 3-5 – Caminho de dados de um circuito sem pipeline

O exemplo considera quatro unidades de cálculo combinatórias entre dois registos. O caminho crítico é determinado como sendo o atraso entre o registo de entrada e o de saída. Assumindo o mesmo atraso para todas as unidades de cálculo igual a 5 ns, temos que o atraso total é de 20 ns. A latência é igual a um período de relógio com uma frequência máxima de 50 MHz.

Se pretendermos processar o mesmo circuito N vezes para vários valores de entrada, a latência total é dada por N ciclos de relógio, que corresponde a uma latência total de execução igual a N × 20 ns.

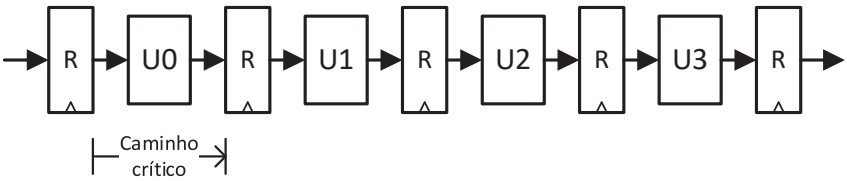


Figura 3-6 – Caminho de dados do circuito da Figura 3-5 com pipeline

Introduzindo registos entre as unidades (ver Figura 3-6), a latência aumenta para 4 períodos de relógio, mas o caminho crítico passa a ser de apenas 5ns (medido entre os registos), a que corresponde uma frequência de 200 MHz. Assumindo que os registos não têm atraso, a latência total do circuito é de 4 × 5 ns = 20 ns, o mesmo que o circuito anterior. O cálculo para N valores de entrada teria a mesma latência total igual a N × 20 ns.

A introdução dos registos entre unidades permite que se aplique a técnica de *pipeline*. Em cada ciclo de relógio pode ser iniciado o processamento de novos dados de entrada. Após a fase de enchimento (3 ciclos de relógio), todas as unidades começam a funcionar em paralelo (ver escalonamento na Figura 3-7).

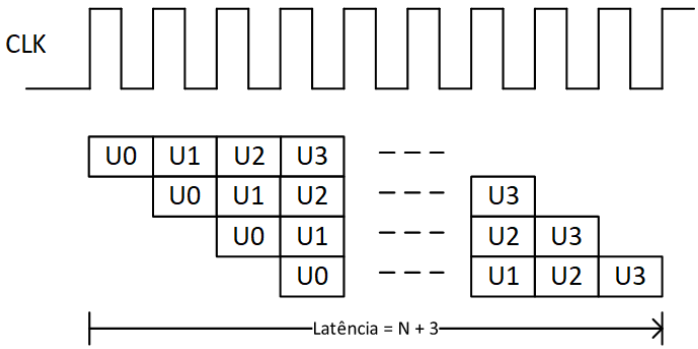


Figura 3-7 – Escalonamento da pipeline do circuito da Figura 3-6

No escalonamento da figura, verificamos o enchimento inicial durante 3 ciclos de relógio e o vazamento final também durante 3 ciclos de relógio. A latência total é dada por N + 3, a que corresponde um tempo total de execução de (N + 3) × 5 ns.

Para N = 100, o tempo de execução dos circuitos, com e sem *pipeline* será:

Sem *pipeline*: Latência = 100 × 20 ns = 2000 ns

Com *pipeline*: Latência = (100 + 3) × 5 ns = 515 ns

Com a introdução dos registos e o funcionamento em modo *pipeline* reduzimos a latência total do circuito em aproximadamente 4 vezes.

Uma opção de projeto a ter em consideração quando se aplica a técnica de *pipeline* é a de escolher o número de níveis de *pipeline*. No caso anterior, considerou-se uma *pipeline* completa, ou seja, introduziu-se um registo entre cada unidade funcional. No entanto, esta pode não ser a melhor opção, por várias razões:

- Necessidade de redução da quantidade de registos;
- Necessidade de redução da latência ou de ajustar o número de níveis de *pipeline*;
- Os dados de entrada ou de saída não podem ser produzidos ou consumidos, respetivamente, à frequência de operação da *pipeline*, o que permite um caminho crítico com maior atraso;
- O circuito tem de operar a uma frequência mais baixa, para reduzir a potência necessária ao seu funcionamento, permitindo um caminho crítico maior.

Nestes casos, pode reduzir-se o número de andares de pipeline omitindo alguns dos registos intermédios. No exemplo em análise, poderia considerar-se apenas um registo a meio do caminho de dados (ver Figura 3-8).

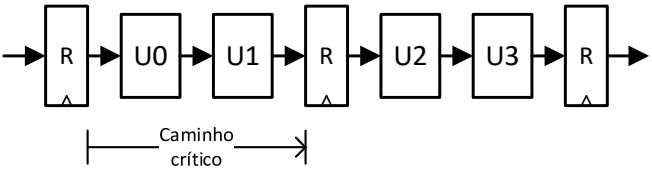


Figura 3-8 – Caminho de dados do circuito da Figura 3-5 com pipeline utilizando apenas um registo intermédio

Assim o circuito passa a ter apenas dois níveis de *pipeline*, um caminho crítico com o dobro do atraso e, consequentemente, uma frequência de relógio com metade do valor.

Para o exemplo com N = 100, o atraso total passaria a ser igual a (100 + 1) × 10 ns = 1010 ns, com uma frequência de operação de 100 MHz.

A técnica de *pipeline* não só melhora o desempenho do circuito, devido ao aumento da frequência, como também aumenta a taxa de produção de dados por unidade de tempo. No exemplo anterior com quatro andares de *pipeline*, a taxa de produção é de um resultado a cada 5 ns (após a fase de enchimento do *pipeline*), quatro vezes mais rápida do que a solução sem *pipeline* que produz um resultado a cada 20 ns.

3.3.2 Paralelização

Uma outra forma de melhorar o desempenho dos circuitos de computação é através da exploração do paralelismo. A computação paralela corresponde a executar operações em paralelo em contraste com a computação em série em que apenas se executa uma operação de cada vez. Para tal, é necessário que a arquitetura hardware tenha vários circuitos a operar em paralelo.

Na secção anterior considerou-se paralelismo quando se analisaram algumas arquiteturas para o cálculo do produto vetorial. Recordemos duas destas arquiteturas na Figura 3-9.

A arquitetura ilustrada na Figura 3-9a não tem unidades de processamento em paralelo. Realiza apenas uma multiplicação e uma soma em série. Já a arquitetura da Figura 3-9b tem quatro multiplicadores em paralelo, que permitem realizar quatro multiplicações ao mesmo tempo, seguidos de dois somadores em paralelo. Os restantes somadores já se encontram em série.

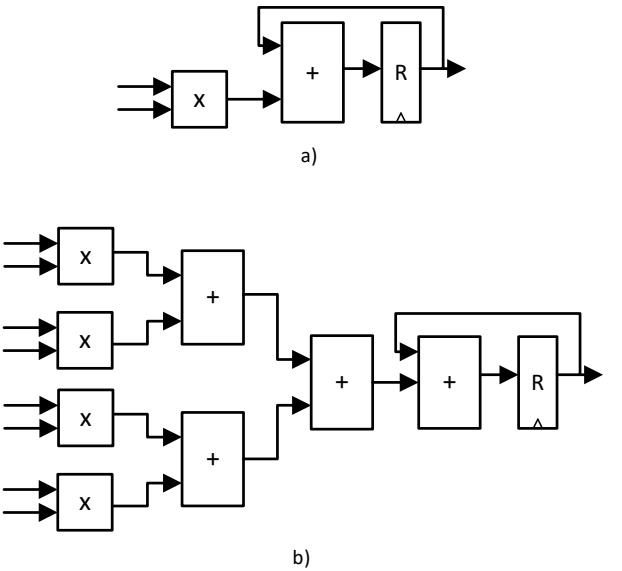


Figura 3-9 – Arquitetura dedicada para o cálculo vetorial a) sem paralelismo e b) com paralelismo. A arquitetura b) utiliza quatro multiplicadores em paralelo seguidos de dois somadores em paralelo e dois somadores em série

O paralelismo tem um custo em termos de recursos, pois implica a replicação de unidades para permitir a execução paralela. Assim, uma arquitetura paralela com melhor desempenho tem uma área maior. Esta relação desempenho/área está sempre presente no projeto de circuitos hardware.

Se compararmos as duas técnicas de otimização verificamos que no paralelismo é necessário replicar unidades de processamento, enquanto na técnica de *pipeline* é apenas necessário adicionar registos ao longo do caminho de dados. Além disso, com o método de *pipeline* obtém-se um aumento da frequência resultante da redução do caminho crítico. No entanto, com *pipeline*, as operações são executadas em série.

Aplicando *pipeline* à arquitetura não paralelizada da Figura 3-9a (ver Figura 3-3) e considerando a execução do produto interno entre vetores de 100 elementos (atraso do multiplicador = 3 ns e atraso da soma = 1 ns), o atraso total passaria a ser igual a (100 + 1) × 3 ns = 303 ns.

A execução do mesmo produto interno na arquitetura da Figura 3-9b demoraria 25 (paralelismo de 4 multiplicadores) × 6 ns = 150 ns. O tempo de execução é reduzido a cerca de metade, mas é necessário adicionar 3 multiplicadores e 3 somadores.

Se tivéssemos considerado apenas um paralelismo de 2 multiplicadores, o atraso seria de 50 (paralelismo de 2 multiplicadores) × 5 ns = 250 ns. O desempenho seria ainda menor que o da solução com *pipeline*, mas com mais um multiplicador e um somador.

Os mecanismos de paralelismo e de *pipeline* podem ser aplicados em simultâneo, ou seja, o método de *pipeline* pode ser aplicado a uma arquitetura paralela. Por exemplo, o método de *pipeline* aplicado à arquitetura da Figura 3-9b resulta no circuito da Figura 3-10.

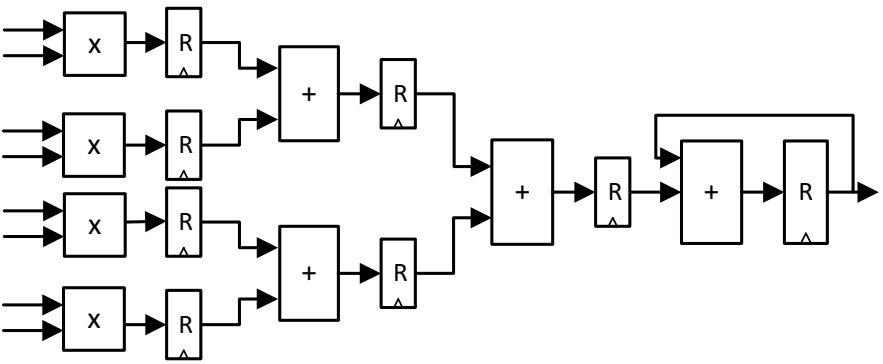


Figura 3-10 – Pipeline aplicado à arquitetura paralela

À arquitetura paralela acrescentaram-se registos à saída de todas as unidades de cálculo. Consegue-se assim reduzir o caminho crítico para 3 ns (atraso do multiplicador). A execução do produto vetorial na nova arquitetura tem um atraso de (25 + 3) × 3 ns = 84 ns.

O tempo de execução do produto vetorial da arquitetura anterior pode ser melhorado se atentarmos ao facto de que o atraso do multiplicador, 3 ns, é igual ao atraso de 3 somadores, 3 × 1 ns. Isto permite que se reduza o número de níveis de *pipeline* sem reduzir a frequência (ver Figura 3-11).

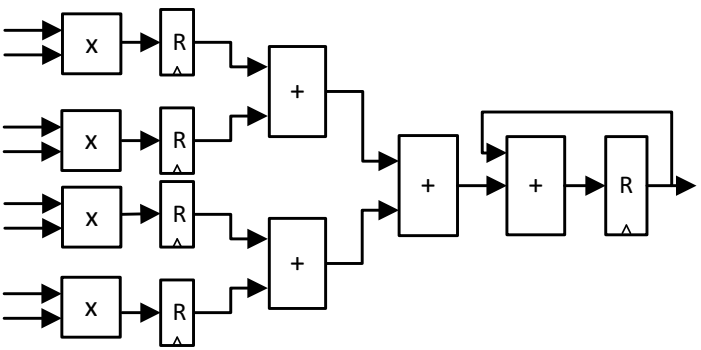


Figura 3-11 – Modificação da pipeline aplicada à arquitetura paralela

Nesta arquitetura, os atrasos dos níveis de p*ipeline* estão balanceados, ou seja, os caminhos críticos dos níveis de *pipeline* são próximos. Neste caso, são ambos iguais a 3 ns. A execução do produto vetorial na arquitetura modificada tem um atraso de (25 + 1) × 3 ns = 78 ns. Comparando com a arquitetura anterior, com quatro níveis de *pipeline*, tem um atraso menor (78 em vez de 84 ns) e tem menos três registos.

Aplicando as técnicas de paralelismo e de *pipeline* ao problema do produto interno conseguiram-se projetar várias arquiteturas com diferentes relações entre desempenho e área. Mais arquiteturas podem ser geradas, bastando variar o fator de paralelismo e/ou o número de níveis de *pipeline*. No limite, seria possível ter 100 multiplicadores em paralelo, realizando todas as 100 multiplicações em apenas um ciclo de relógio.

3.4 Circuitos de Memória

Um dos aspetos fundamentais de uma arquitetura dedicada é a estrutura de memória do caminho de dados. Não basta ter uma unidade de cálculo otimizada com paralelismo e/ou *pipeline* para garantir o desempenho da arquitetura. Temos de ter em conta que os dados a serem processados são geralmente lidos de memórias e os resultados são igualmente escritos em memória. Quando se aumenta o paralelismo de um circuito ou a frequência de operação à custa de *pipeline* é necessário aumentar a taxa de transmissão de dados entre o circuito e a memória, ou seja, a largura de banda de acesso à memória.

A largura de banda de uma memória corresponde à taxa de transmissão de dados da memória. Por exemplo, uma largura de banda de 500 MBytes/s significa que é possível ler ou escrever 500 MBytes de dados por segundo.

Se considerarmos a taxa a que os dados podem ser consumidos pelo circuito e a largura de banda da memória disponível, é possível determinar qual dos sistemas, computação ou memória, está a limitar o funcionamento do circuito. Por exemplo, um circuito que consome 16 bytes por período de relógio a uma frequência de 100 MHz tem uma taxa de consumo de $16 \times 100 \times 10^6$ bytes/s = 1,6 GBytes/s. Se a memória tiver uma largura de banda de 500 MBytes/s, então o desempenho do sistema é limitado pelo acesso à memória. Neste caso, o sistema de computação não tira partido da capacidade de processamento total pois tem de esperar pelos dados. No caso de a taxa de consumo do sistema computacional ser inferior à largura de banda da memória, então o desempenho do sistema é limitado pelo circuito de computação. O projetista de hardware para aceleração de algoritmos depara-se, em geral, com a situação em que o desempenho do sistema é limitado pela largura de banda da memória.

Num sistema baseado num processador genérico, a arquitetura de memória e a arquitetura de computação são fixas. É constituída, tipicamente, por memória principal, memória *cache* e registos internos do processador. Face à largura de banda de acesso à memória principal,

o programador procura reutilizar dados guardados em registos internos e/ou em *cache* para reduzir os acessos à memória principal.

Num sistema hardware dedicado, o sistema de memória é ajustado da melhor forma ao algoritmo ou aplicação que se pretende implementar em hardware. Não existe, assim, ao contrário do que se verifica numa arquitetura baseada num processador genérico, uma arquitetura de memória fixa.

A arquitetura hardware tem acesso a memória externa e a memória interna. A memória externa tem uma largura de banda pré-definida, mas a memória interna pode ser ajustada às necessidades do sistema da aplicação. Uma ferramenta de síntese considera vários tipos de memória interna: registo, RAM (*Random Access Memory*), ROM (*Read Only Memory*) e FIFO (*First-In-First-Out*). Por exemplo, um vetor pode ser armazenado internamente em qualquer um destes tipos de memória, de acordo com as necessidades de memória e de largura de banda.

3.4.1 Registo

O registo é a implementação de memória mais rápida, sendo possível aceder de forma independente a vários registos, sem necessidade de lógica de endereçamento. O registo surge integrado com as unidades de cálculo do caminho de dados e é utilizado para armazenamento de variáveis ou de constantes. Como unidade independente, o registo pode ser configurado com qualquer número de bits.

3.4.2 Memória de Acesso Aleatório

A memória de acesso aleatório (RAM) permite o armazenamento de dados de forma mais eficiente do que utilizando registos, uma vez que a área ocupada por bit é bastante inferior comparativamente à área do registo. No entanto, não permite a leitura de todo o conteúdo em simultâneo.

Uma RAM tem um determinado número de posições ou espaços de armazenamento e em cada posição é possível armazenar uma palavra com um determinado número de bits. Os dados são escritos e lidos um de cada vez, sendo necessário identificar a posição que se pretende aceder através da entrada de endereço.

A RAM tem um tempo de acesso inferior ao da memória externa, mas maior do que o de um registo. É em geral utilizada para guardar um volume de dados que vai ser reutilizado pelas unidades de processamento, de modo análogo a uma *cache*.

A configuração mais comum de uma RAM apenas permite um acesso de cada vez. No entanto, a RAM pode ter mais do que um porto de acesso, passando a designar-se *RAM de múltiplo porto*. O caso mais comum é o de uma RAM de duplo porto, que permite dois acessos em simultâneo, que podem ser de escrita ou de leitura. Por exemplo, como descrito no capítulo 2, as RAM presentes nas FPGA, designadas *Block RAM (BRAM)*, podem ser configuradas como RAM de porto único ou como RAM de duplo porto. A vantagem na

utilização de RAM de duplo porto está na possibilidade de, por exemplo, escrever novos dados enquanto se leem os anteriores, ou ler dados de duas posições diferentes em simultâneo. Para tal, é preciso gerar dois endereços e dois conjuntos de sinais de controlo.

A largura de banda das memórias RAM depende do tamanho do barramento de dados das entradas/saídas e da frequência de operação. Por exemplo, uma RAM com um barramento de dados de 64 bits a operar a uma frequência de 100 MHz tem uma largura de banda de 64 x 100 × 10^6 = 6.4 Gbits/s ou 64/8 × 100 Mbytes/s = 800 Mbytes/s.

A estrutura de memória do circuito não é única e deve procurar garantir as necessidades de produção e consumo de dados da unidade de processamento. Consideremos que se pretende executar o produto interno de dois vetores com elementos de 32 bits a uma frequência de 100 MHz recorrendo a uma arquitetura com apenas um multiplicador e um somador em *pipeline*. O multiplicador necessita de dois operandos de 32 bits, que podem ser obtidos com diferentes configurações de memória (ver Figura 3-12)

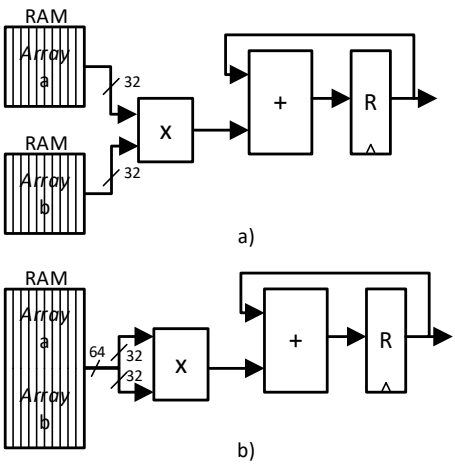


Figura 3-12 – Sistema de memória. a) Memórias separadas para cada um dos vetores e b) memória única para os dois vetores.

A arquitetura da Figura 3-12a utiliza duas memórias separadas, cada uma com um barramento de dados a 32 bits. Cada uma das memórias guarda um dos vetores de entrada. Na arquitetura da Figura 3-12b, ambos os vetores são guardados numa única memória com um barramento de 64 bits, em que em cada posição guarda dois elementos (um elemento do vetor `a` e um elemento do vetor `b`).

Ambas as arquiteturas se caracterizam por ter uma largura de banda e uma taxa de consumo de dados de 64/8 × 100 MHz = 800 Mbytes/s. Logo, o sistema de memória e o sistema de computação estão equilibrados.

No caso de a taxa de consumo da unidade de processamento aumentar, é necessário aumentar igualmente a largura de banda de acesso à memória para continuar a garantir um sistema equilibrado. Tratando-se de um circuito hardware dedicado, basta adicionar mais memórias em paralelo e distribuir os dados pelas memórias de modo a garantir as necessidades de dados da unidade de processamento. Para exemplificar este processo, consideremos o circuito de cálculo do produto vetorial com quatro multiplicadores em paralelo. Neste caso, para garantir a taxa de consumo de quatro elementos de ambos os vetores por ciclo de relógio, é necessário ler a mesma quantidade de elementos da memória em cada ciclo de relógio. Uma configuração possível do sistema de memória consiste em quatro RAM em paralelo com um barramento de dados de 64 bits (ver Figura 3-13).

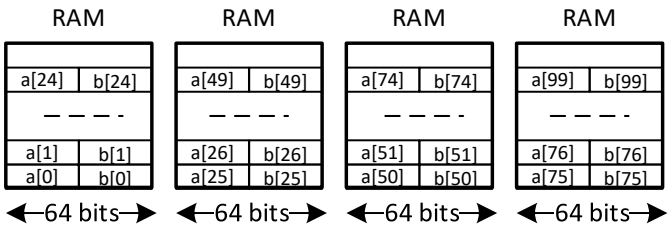


Figura 3-13 – Sistema de memória com quatro RAM em paralelo com barramento de dados de 64 bits por RAM

Os vetores foram partidos em quatro parcelas, sendo cada uma delas guardada numa memória RAM separada. Cada posição das memórias armazena um elemento de cada vetor com o mesmo índice. A largura de banda total do sistema de memória é de 4 × 64 bits por ciclo de relógio. Esta largura de banda corresponde à taxa de consumo do circuito de cálculo do produto vetorial, pelo que o sistema se encontra equilibrado.

A configuração de armazenamento de dados nas memórias não é única. Podia ter-se considerado, por exemplo, que as primeiras duas memórias guardam um vetor e as outras duas guardam o segundo vetor. O que é necessário garantir é que em cada ciclo de relógio se consigam ler quatro elementos de cada um dos vetores.

A análise efetuada na secção anterior aplica-se igualmente a memórias do tipo ROM. A única diferença é que a ROM é utilizada apenas para leitura, ou seja, para armazenar dados que não são modificados durante a execução do algoritmo.

3.4.3 Memória FIFO

A memória FIFO é uma memória de dois portos, um de leitura e outro de escrita, com acesso sequencial. Ter acesso sequencial significa que os dados são lidos pela mesma ordem com que são escritos, ao contrário da RAM e da ROM que permitem acesso não sequencial (aleatório) aos dados.

A vantagem da FIFO relativamente às memórias de acesso aleatório (RAM/ROM) é que não necessita de entradas de endereços para escrita e leitura, uma vez que os dados são acedidos de forma sequencial. No entanto, obriga a que os dados sejam lidos pela mesma ordem com que foram escritos e após cada leitura a FIFO endereça automaticamente o próximo dado para ser lido.

A sua utilização está, assim, condicionada ao armazenamento de dados que apenas sejam acedidos sequencialmente. O exemplo do produto vetorial permite a utilização de FIFO, uma vez que os vetores são acedidos sequencialmente. A estrutura paralela com memórias RAM da secção anterior pode ser substituída por FIFO (ver Figura 3-14).

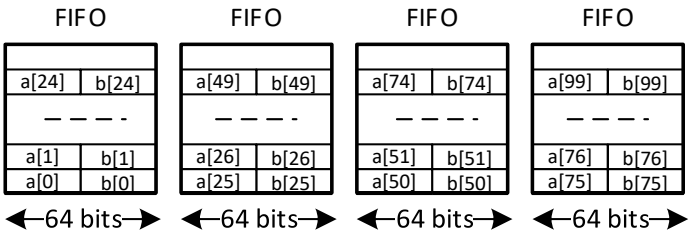


Figura 3-14 – Sistema de memória com quatro FIFO em paralelo com barramento de dados de 64 bits por FIFO

Repare-se que os vetores armazenados em FIFO são acedidos em sequência no algoritmo de cálculo do produto interno. Se tal não fosse possível, por restrições do algoritmo, então não era possível utilizar a FIFO.

3.5 Protocolos de Interface

Em geral, um circuito hardware dedicado é interligado com outros módulos ou integrado num sistema hardware/software. A definição das interfaces de acesso ao circuito dedicado é assim um passo fundamental no projeto destes circuitos, determinando a facilidade com que é integrado no sistema.

Nesta secção, aborda-se um dos protocolos mais utilizados na interligação de módulos hardware, o AXI (*Advanced eXtensible Interface*). O AXI é um protocolo que faz parte de uma arquitetura de barramentos da ARM, a AMBA (*Advanced Microcontroller Bus Architecture*). A norma AXI determina como os módulos devem trocar dados entre si.

3.5.1 Interface AXI

A última versão do AXI, AXI4, inclui três tipos de interface:

AXI4-Full – Comunicação baseada em mapeamento de memória para comunicações de elevada taxa de transmissão;

AXI4-Lite – Comunicação baseada em mapeamento de memória para comunicações de baixa taxa de transmissão;

AXI4-Stream – Comunicação ponto-a-ponto para elevada taxa de transmissão.

A especificação AXI assume que os intervenientes de uma comunicação são um mestre AXI e um escravo AXI. A interligação de vários mestres e/ou de vários escravos é feita através de um bloco de interligação que se encarrega de encaminhar as diversas transações entre os mestres e os escravos (ver Figura 3-15).

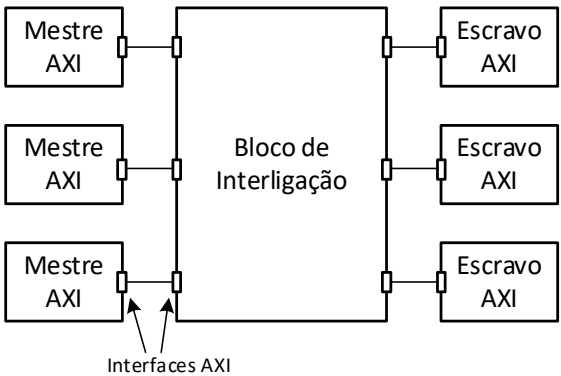


Figura 3-15 – Bloco de interligação que permite a interligação de várias entidades AXI

Cada uma das entidades AXI liga a uma interface AXI do bloco de interligação que se encarrega internamente do encaminhamento dos dados.

As interfaces baseadas em mapeamento de memória (*AIX4-Full* e *AXI4-Lite*) têm cinco canais separados:

- Canal de endereço de leitura;
- Canal de endereço de escrita;
- Canal de dados de leitura;
- Canal de dados de escrita;
- Canal de resposta à escrita;

Os cinco canais permitem o acesso a uma memória para leitura ou para escrita.

Uma leitura de dados é realizada com os canais de endereço de leitura e de dados de leitura (ver Figura 3-16).

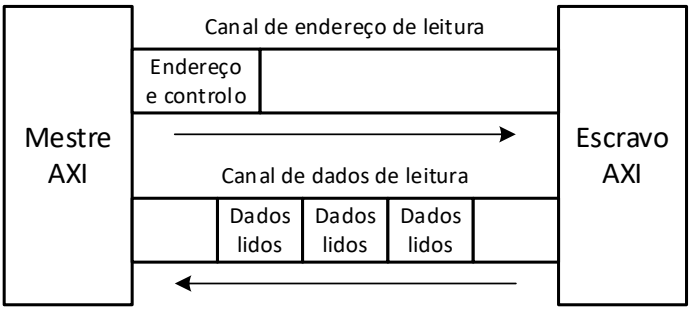


Figura 3-16 – Canais AXI envolvidos numa leitura de dados

O mestre envia o endereço, e o escravo devolve um ou mais dados a partir desse endereço.

Uma escrita de dados é realizada com os canais de endereço de escrita, de dados de escrita e de resposta de escrita (ver Figura 3-17).

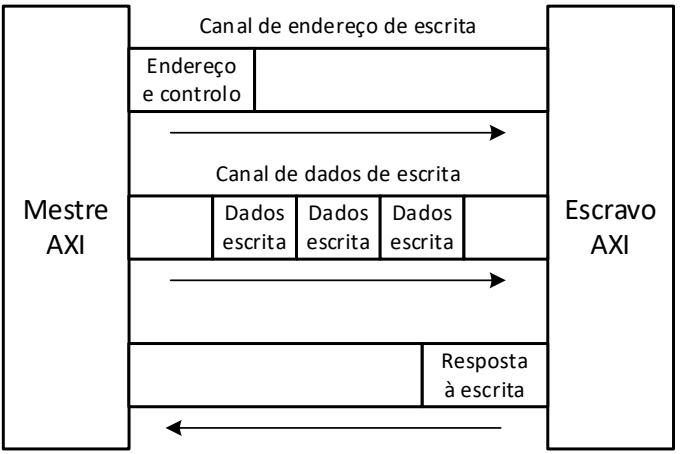


Figura 3-17 – Canais AXI envolvidos numa escrita de dados

O mestre envia o endereço de escrita e os dados a escrever, e o escravo devolve uma resposta após a escrita dos dados.

A principal característica das interfaces AXI é a existência de canais separados para endereços e para dados, em ambas as operações de escrita e de leitura.

Um canal é constituído por vários sinais de controlo, sendo a comunicação controlada pelos sinais `VALID` e `READY`. O sinal `VALID` é controlado por quem coloca dados no canal e indica que há dados válidos no canal. O sinal `READY` é controlado por quem recebe os dados e indica que está pronto para receber. A transmissão de uma palavra por qualquer um dos canais é designada *transferência*. Apenas ocorre uma transferência quando ocorre uma

transição no sinal de relógio e ambos os sinais `VALID` e `READY` estão ativos (ver Figura 3-18).

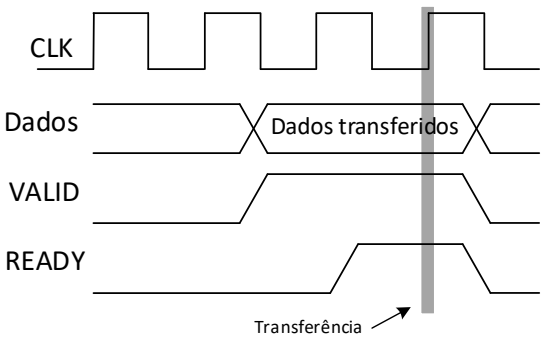


Figura 3-18 – Utilização dos sinais `VALID` e `READY` numa transmissão AXI4

A figura ilustra o momento da transferência que ocorre quando ambos os sinais estão ativos, indicando que há dados válidos e que o consumidor está pronto para os receber.

O *AXI4-Full* permite a comunicação em rajada, ou seja, fornecendo apenas o endereço inicial dos dados, acede até 256 palavras em sequência. A capacidade de comunicação em rajada permite uma comunicação com elevada taxa de transmissão.

Outras características do *AXI4-Full* é a possibilidade de as interfaces mestre e escravo operarem a frequências diferentes e de ser possível introduzir registos de *pipeline* no canal de comunicação entre as interfaces para melhorar os tempos de comunicação.

O *AXI4-Lite* é uma simplificação do *AXI4-Full*, sendo a mais relevante a não possibilidade de transmitir em modo rajada. Por esta razão, é, em geral, utilizado para envio de sinais de controlo, para leitura de registos de estado, etc. Tipicamente, é usado apenas para transferências que não exigem uma elevada taxa de transmissão e que ocorrem isoladamente.

A interface *AXI4-Stream* é utilizada para comunicações ponto-a-ponto entre dois blocos. O canal implementa uma FIFO que controla o fluxo de dados entre o produtor e o consumidor de dados. Neste tipo de interface apenas se utiliza o canal de dados de escrita (ver Figura 3-19).

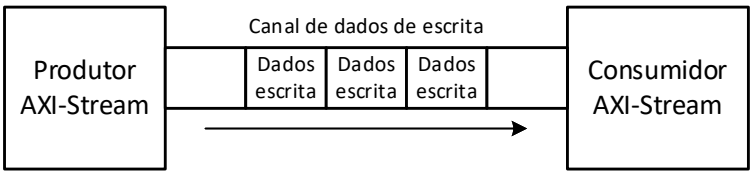


Figura 3-19 – Transação de escrita num canal AXI4-Stream

Os dados são enviados em sequência sem necessidade de endereço e não existe qualquer limite no tamanho da sequência.

No *AXI4-Stream* os sinais VALID e READY são designados TVALID e TREADY, respetivamente. Uma vez que a transmissão em rajada não tem limite, a interface inclui também o sinal TLAST, que indica o fim da sequência de dados (ver Figura 3-20).

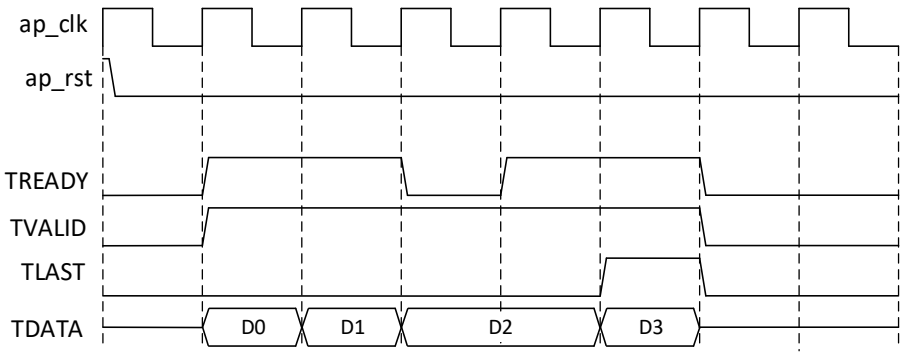


Figura 3-20 – Utilização dos sinais TVALID, TREADY e TLAST numa transmissão AXI4-Stream

No exemplo da figura, o sinal TLAST fica ativo durante a transmissão da última palavra da rajada de dados, D3.

O produtor dos dados controla os sinais TVALID, TLAST, e o consumidor dos dados controla o sinal TREADY. Quando o produtor tem dados válidos ativa o TVALID. O último dado enviado é acompanhado do TLAST. O consumidor de dados ativa o TREADY quando pode receber dados. Quando não pode receber, desativa o TREADY, como acontece no exemplo ilustrado durante a transmissão de D2. O *AXI4-Stream* inclui outros sinais, não representados no diagrama, que são opcionais. Os sinais TKEEP e TSTROBE permitem multiplexar a posição dos dados e os próprios dados no sinal TDATA. O sinal TUSER é utilizado para enviar dados adicionais do utilizador. Os sinais TID, TDEST são utilizados para identificar o conjunto de dados enviado e um identificador do destino, respetivamente.

A interface AXI4 foi abordada de forma introdutória, considerando apenas os pontos essenciais para perceber a descrição de interfaces AXI em HLS. Para uma leitura mais aprofundada do tema, poderá aceder à norma (*AMBA AXI3 and AXI4 Protocol Specification - Arm Developer*) e para uma descrição mais detalhada do AXI no âmbito dos dispositivos FPGA AMD-Xilinx poderá aceder ao *AXI Reference Guide* (UG761).

4 Introdução à Síntese de Alto Nível

O projeto de circuitos digitais baseado em síntese de alto nível (HLS – *High-Level Synthesis*) recebe como entrada uma descrição algorítmica do circuito e gera o hardware de um circuito digital que implementa o algoritmo. A descrição do algoritmo é feita com uma linguagem de software e o hardware do circuito gerado é descrito com uma linguagem de descrição de hardware. Neste livro, consideram-se as linguagens de programação C e C++ para a especificação do algoritmo a ser implementado em hardware. Nas secções seguintes, descrevem-se os fundamentos da síntese de alto nível, quais os passos envolvidos no projeto de hardware utilizando a síntese de alto nível e apresenta-se genericamente os mecanismos envolvidos na síntese de uma função. Estes mecanismos serão posteriormente descritos em detalhe na parte II do livro.

4.1 Introdução

A síntese de alto nível não deve ser vista simplesmente como um método totalmente automático de geração de hardware a partir de descrições software do circuito, similar a um compilador. É importante não esquecermos que, embora estejamos a utilizar linguagens de programação de alto nível para especificar um circuito, o objetivo final é a sua implementação em hardware. Uma descrição em C ou C++ menos cuidada pode resultar numa implementação hardware pouco otimizada e ineficiente. Assim, ao descrevermos um circuito hardware em C ou C++ devemos ter ideia do hardware que será gerado para determinada estrutura do código software. Neste sentido, sugere-se que se procure seguir as recomendações de descrição de hardware em C/C++, descritas na segunda parte deste livro. A síntese de alto nível atual deve ser vista como uma ferramenta de apoio ao projeto de hardware e não como um substituto do projetista de hardware.

O projeto de circuitos digitais a partir de linguagens de programação, como o C ou o C++, com recurso a ferramentas de síntese de alto nível tem diversas vantagens comparativamente ao projeto baseado em linguagens de descrição de hardware, como o VHDL ou o Verilog:

- Aumento da produtividade no projeto de hardware – A utilização de ferramentas de síntese de alto nível permite elevar o nível de abstração do projeto. Não só reduz a complexidade da descrição, como liberta o projetista para o essencial da descrição que é a funcionalidade pretendida. Este aspeto é particularmente notório no que diz respeito ao desenvolvimento do circuito de controlo, responsável por muitos dos problemas de projeto de hardware e pelo elevado tempo de depuração, que são tratados automaticamente pela ferramenta de HLS;
- Aproxima o projeto de hardware dos programadores de software disponibilizando um método de aceleração dos algoritmos através da sua execução em hardware – A implementação de um algoritmo ou parte de um algoritmo em hardware dedicado implica o projeto do hardware e a sua integração com o software que executa num processador. A síntese de alto nível eleva o projeto hardware/software para um nível de abstração mais próximo da programação de software;
- Validação e simulação da funcionalidade do sistema integrada com a descrição em software – As metodologias de síntese de alto nível permitem a simulação da descrição do circuito em software e a co-simulação do circuito resultante da síntese juntamente com o software onde será integrado. Isto torna o processo de verificação mais rápido e simultaneamente verifica a integração do software com o hardware utilizando o mesmo ambiente de simulação e depuração do software;
- Exploração eficiente do espaço de projeto – Permite de forma mais simples e rápida a exploração do espaço de projeto com a aplicação de diretivas de otimização do código e com recurso a estimativas de desempenho e de utilização de recursos geradas pela ferramenta de HLS.

De modo a obter os melhores resultados, devemos perceber quais os passos envolvidos na síntese de alto nível, como descrever o hardware usando linguagens C e C++ e quais as construções de programação que são sintetizáveis. Além disso, devemos ter presente qual o fluxo de projeto baseado em HLS e quais os principais métodos de otimização. Nas secções seguintes são abordados estes pontos.

4.2 Metodologia de Projeto de Hardware com Síntese de Alto Nível

As ferramentas de síntese de alto nível convertem uma função descrita com uma linguagem de programação de alto nível num módulo hardware ou núcleo IP (*Intellectual Property*). O módulo pode ser utilizado de modo independente ou posteriormente integrado num sistema hardware ou hardware/software.

A metodologia de projeto de hardware com base em síntese de alto nível integra um conjunto de ferramentas que permitem verificar a descrição da funcionalidade do circuito pretendido, bem como explorar o espaço de projeto e otimizar o circuito final de acordo com compromissos de desempenho e de recursos hardware. O projeto inicia com a especificação do circuito em C ou C++ e termina com a integração e o empacotamento da descrição RTL do circuito resultado da síntese de alto nível (ver metodologia de síntese de alto nível da ferramenta *Vitis HLS* na Figura 4-1). O projetista pode influenciar a síntese do circuito especificando restrições de desempenho e de área e aplicando diretivas que orientam a síntese (a serem descritas na Parte II).

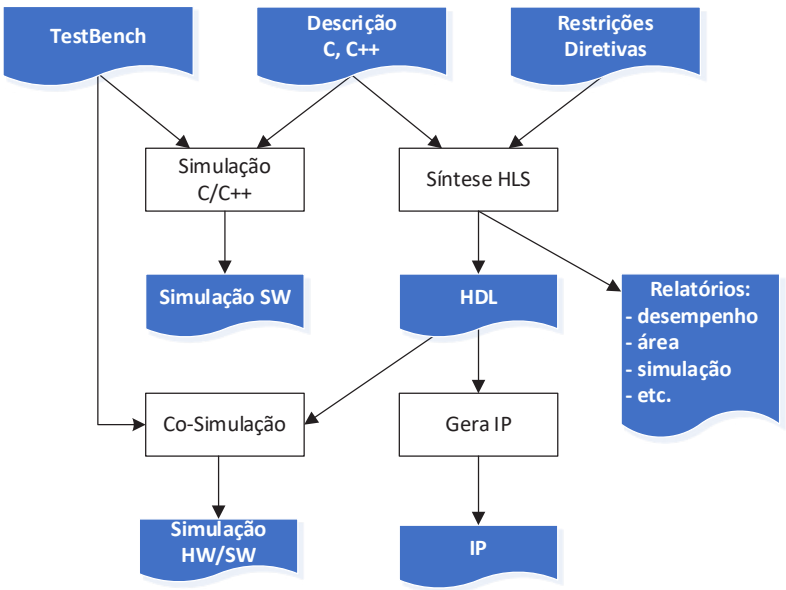


Figura 4-1 - Metodologia de síntese de alto nível

Inicialmente, são especificadas a função/algoritmo, em C ou C++ (*Descrição C, C++*), que se pretende implementar em hardware, e as restrições e diretivas (*Restrições e Diretivas*). A função pode ser simulada e depurada na linguagem nativa de modo a verificar a funcionalidade pretendida (*Simulação C/C++*). A função é depois sintetizada, ou seja, é gerada uma implementação descrita numa linguagem de descrição de hardware (*HDL*) a partir da descrição em C/C++. A síntese também gera um conjunto de relatórios com estimativas de desempenho e de área (*Relatórios*) a partir das quais é possível verificar se o circuito cumpre os requisitos de projeto. O circuito pode depois ser co-simulado (*Co-Simulação*) juntamente com uma descrição software de topo do sistema em que o módulo hardware vai ser integrado. O ficheiro com a descrição da sequência de testes a realizar na simulação e na co-simulação designa-se, geralmente, *Testbench*. Finalmente, é feito o

empacotamento do módulo hardware (*Gera IP*) de modo a ficar disponível para ser utilizado individualmente ou integrado com outros módulos hardware ou com um processador.

A informação de entrada da ferramenta HLS consiste assim no seguinte:

- Descrição em C ou C++ da função a sintetizar para uma descrição hardware. A função é traduzida para RTL e os seus argumentos são traduzidos em entradas e saídas de acordo com um determinado protocolo de comunicação;
- Descrição em C ou C++ de um ficheiro de teste (*testbench*). O ficheiro de teste é usado para a simulação e para a co-simulação do circuito;
- Restrições de projeto. As restrições especificam os objetivos de desempenho, em termos do período de relógio, taxa de produção, incerteza de relógio, etc. Muitos destes aspetos de desempenho dependem do dispositivo alvo, pelo que as restrições também incluem a especificação do dispositivo alvo do projeto;
- Diretivas de otimização. As diretivas são usadas para orientar a ferramenta de síntese na escolha da implementação para determinadas especificações, bem como para especificar determinadas técnicas de otimização do projeto de hardware. Por exemplo, serve para especificar o tipo de protocolo de comunicação a utilizar na implementação dos argumentos da função, entre outros.

O resultado da síntese de alto nível consiste numa descrição do circuito com uma linguagem de descrição de hardware (VHDL ou Verilog). Quando empacotado, o módulo RTL gerado pode ser integrado em bibliotecas e utilizado por outras ferramentas de projeto de hardware, como a síntese lógica da ferramenta *Vivado* da *AMD-Xilinx*. São também gerados relatórios do processo de síntese, da simulação e do empacotamento do módulo, bem como estimativas de desempenho e de ocupação de recursos hardware.

4.3 Passos da Síntese de Alto Nível

Para gerar um módulo hardware utilizando a síntese de alto nível, o projetista começa por descrever o circuito com uma função numa das linguagens de programação suportadas pela ferramenta de HLS. O código especificado pela função é depois traduzido (sintetizado) numa descrição RTL de acordo com a seguinte tradução de objetos:

- Argumentos e retorno da função: os argumentos da função e o valor de retorno, caso exista, são traduzidos em portos de entrada e de saída do módulo hardware. O protocolo de comunicação associado a um porto depende do tipo de argumento;
- Operações: as operações (lógicas, aritméticas, etc.) utilizadas na função são traduzidas em operadores hardware;
- Estruturas de controlo: as estruturas de controlo usadas na descrição de estruturas de repetição, de estruturas de condições, etc. são traduzidas em máquinas de estado para controlo do hardware;

- Funções: as funções chamadas dentro da função de topo são, cada uma delas, traduzidas em módulos hardware que depois são interligados entre si formando uma estrutura hierárquica de módulos hardware;
- Variáveis e vetores: as variáveis e os vetores declarados dentro da função são traduzidos em memória, que pode ser registos, RAM, ROM ou FIFO.

O processo de síntese da ferramenta HLS que traduz as diferentes estruturas de uma função em C ou C++ em estruturas hardware realiza três passos principais: (1) escalonamento, (2) mapeamento e (3) síntese da lógica de controlo.

O escalonamento determina a ordem com que as operações e as comunicações são realizadas e em que ciclo de relógio são processadas. Esta operação depende, naturalmente, da frequência de relógio, do atraso da cada operação e de diretivas de otimização. Uma operação mais lenta ou um relógio mais rápido pode levar a que a operação demore mais do que um ciclo de relógio para executar. As dependências de dados também determinam o escalonamento das operações e das transferências e, consequentemente, os tempos de atraso do componente hardware.

O mapeamento determina qual o recurso hardware a utilizar para cada uma das operações. A escolha pode ser influenciada pelo projetista através da utilização de diretivas HLS.

A síntese lógica de controlo implementa a máquina de estados que garante o funcionamento do circuito de acordo com o escalonamento e o mapeamento determinados nos passos anteriores.

Todo o processo de síntese baseia-se em estimativas de projeto relativas à ocupação de recursos hardware e aos tempos de execução. As estimativas são usadas durante os processos de escalonamento e de mapeamento, mas também podem ser utilizadas pelo projetista para estimar, numa fase inicial, se o projeto cumpre os requisitos necessários.

Para melhor percebermos o processo de síntese da ferramenta de HLS, consideremos um exemplo de síntese de um circuito simples que multiplica dois valores e soma com um terceiro. A função em C seria descrita de acordo com o Código 4-1.

```
int multiplyAdd (int a, int b, int c){

	int ma;

	ma = a * b + c;

	return ma;
}
```

Código 4-1 – Exemplo de uma função em C a ser traduzida para RTL

A ferramenta de síntese começa por determinar as dependências de dados entre as diversas atribuições da função. Estas dependências podem ser facilmente visualizadas com um grafo de fluxo de dados (ver Figura 4-2).

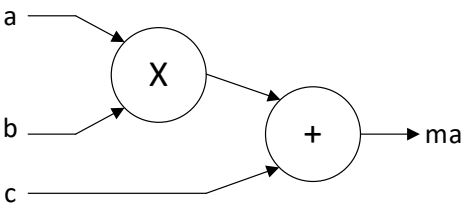


Figura 4-2 - Representação do grafo de fluxo de dados da função descrita no Código 4-1

Os dois nós representam as operações usadas na função: multiplicação e adição. A ligação entre os nós indica uma dependência de dados, ou seja, a soma só pode ser executada após a multiplicação. As dependências de dados determinam, assim, a ordem de execução das diversas operações.

De seguida, a ferramenta de síntese estabelece um mapeamento entre operações e operadores aritméticos a serem implementados em hardware. Os recursos são escolhidos a partir de uma biblioteca de componentes parametrizáveis e previamente caracterizados em termos de área e de desempenho. São possíveis diferentes implementações para o mesmo operador. O mesmo operador pode ser usado para implementar mais do que uma operação do mesmo tipo.

Por fim, é feito o escalonamento das operações em ciclos de relógio, ou seja, determina-se em que ciclo ou ciclos de relógio se dá a execução de cada uma das operações. No exemplo, existem duas operações que precisam de ser escalonadas: a multiplicação e a soma. Existem diversas soluções de implementação que dependem do número de portos de entrada, da frequência de relógio e dos tempos de atraso dos módulos de multiplicação e de soma alocados às operações. Consideremos uma primeira solução em que as entradas `a`, `b` e `c` estão disponíveis ao mesmo tempo e que a multiplicação seguida da soma necessita de dois ciclos de relógio para executar. Neste caso, teríamos a solução de escalonamento apresentada na Figura 4-3.

A multiplicação é executada no primeiro ciclo do sinal de relógio (CLK) e a soma no segundo. Os argumentos da função são implementados como portos de entrada de 32 bits (considerando que o tipo de dados `int` é de 32 bits). O retorno é implementado como um porto de saída também de 32 bits.

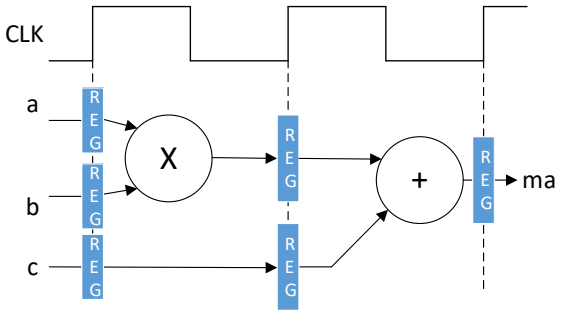


Figura 4-3 - Solução de escalonamento assumindo a disponibilidade das entradas a, b e c no mesmo ciclo de relógio

O controlo deste exemplo é relativamente simples. Os dados começam por ser registados à entrada e o resultado está disponível após dois ciclos de relógio.

Dependendo da frequência do relógio e dos tempos de atraso do multiplicador e do somador, a multiplicação e a soma poderiam ser executadas no mesmo ciclo de relógio. Neste caso, o circuito demoraria apenas um ciclo de relógio a executar.

Consideremos agora um exemplo em que se pretende um circuito que realize a multiplicação-soma do exemplo anterior sobre vetores de dados. A especificação em linguagem C pretendida está descrita no Código 4-2.

```
void multiplyAddArray (int a[8], int b[8], int c[8], ma[8]){

      for (i = 0; i < 8; i++)
            ma[i] = a[i] * b[i] + c[i];

}
```

Código 4-2 – Exemplo de uma função em C com uma estrutura de repetição (estrutura FOR) a ser traduzida para RTL

A operação considerada no primeiro exemplo é agora repetida oito vezes com valores de três vetores, a, b e c, com os mesmos índices. Sem considerar qualquer otimização, a operação realizada no corpo da estrutura de repetição, a × b + c, é escalonada como no exemplo anterior e depois executada oito vezes. Esta sequência de operações é garantida pelo circuito de controlo (ver Figura 4-4).

Os argumentos são vetores, pelo que são implementados com memórias fora da função. A ferramenta de síntese gera automaticamente os portos de entrada e de saída com os sinais necessários para aceder a estas memórias.

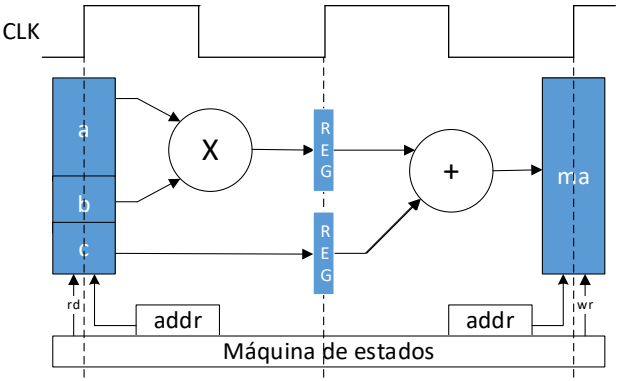


Figura 4-4 - Solução de escalonamento do exemplo do Código 2

Neste exemplo, a máquina de estados do circuito de controlo é responsável por contar o número de vezes que a operação é realizada e de gerar os endereços de acesso aos vetores.

O tempo de execução do algoritmo é determinado pelo tempo de execução da estrutura de repetição *for*. A latência de uma iteração da estrutura `FOR` é de 3 ciclos de relógio: guarda os dados de entrada em registos, guarda o resultado da multiplicação e o valor de `c` em registos intermédios e, por fim, regista o resultado da soma. A execução das 8 iterações da estrutura *for* demora, assim, 24 ciclos de relógio no total.

A execução descrita no parágrafo anterior não considera a técnica de *pipeline*, descrita na secção 3.3.1. No caso de se utilizar esta técnica, teremos um tempo de execução de 10 ciclos de relógio.

4.4 Tipos de Dados

As linguagens de programação têm um conjunto de tipos de dados pré-definidos com um número de bits de representação múltiplo de 8, nomeadamente, 8, 16, 32 e 64 bits. Este número de bits é uma consequência natural do facto de que, quando foram desenvolvidas, as linguagens de programação assumirem que o código seria executado num processador genérico. A unidade aritmética, os registos, o acesso à memória, etc., de um processador manipulam e armazenam, na sua maioria, dados representados com 8, 16, 32 ou 64 bits.

Quando se pensa numa implementação em hardware, não estamos necessariamente restringidos a estes tamanhos de dados. No projeto de um circuito em hardware é possível especificar qualquer tamanho em bits dos dados. A vantagem em considerar uma representação de dados com tamanho arbitrário resulta da redução do tamanho dos operadores e/ou da memória necessários para implementar o circuito.

Consideremos, como exemplo, a multiplicação de dois operandos cujos valores podem variar entre 0 e 1000. Quando descrito em software para ser executado num processador,

necessitamos de representar os operandos com 16 bits, uma vez que 8 bits não são suficientes para representar todo o domínio, [0, 1000], dos operandos. No caso de o circuito ser implementado em hardware, uma tradução direta da representação a 16 bits produziria um circuito com um multiplicador de 16×16 (ver Figura 4-5a).

Assumindo registos à entrada e à saída do multiplicador, teríamos dois registos de 16 bits, um registo de 32 bits e um multiplicador de 16×16.

No entanto, para representar o domínio dos operandos entre 0 e 1000 são necessários apenas de 10 bits. Neste caso, bastaria um multiplicador de 10×10. Consequentemente, teríamos um circuito hardware otimizado para o número de bits de representação estritamente necessário (ver Figura 4-5b).

A versão otimizada da implementação em hardware necessita de dois registos de 10 bits, um registo de 20 bits e um multiplicador de 10×10. O circuito otimizado ocupa menos área, é mais rápido e consome menos energia.

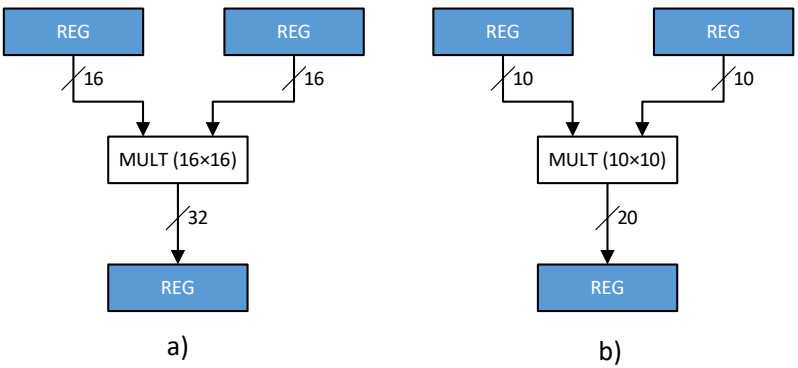


Figura 4-5 – Circuito de multiplicação. a) Solução não otimizada com implementação do tipo de dados nativo alinhado a 8 bits, b) Solução otimizada considerando precisão arbitrária.

Assumindo que os recursos hardware ocupados por um registo são proporcionais ao número de bits e que os ocupados pelo multiplicador aumentam quadraticamente com o número de bits, teríamos uma redução de 33% na área dos registos e de 60% na área do multiplicador.

A síntese de alto nível permite a otimização do tamanho dos operandos. Para tal, disponibiliza tipos de dados de precisão arbitrária e um conjunto de bibliotecas para a manipulação de dados com precisão arbitrária, ou seja, com um número de bits de representação diferente do número de bits dos tipos de dados nativos. Os tipos de dados de precisão arbitrária são sintetizados e simulados no ambiente de síntese de alto nível.

A precisão arbitrária pode ser aplicada a diferentes tipos de dados. Por exemplo, o *Vitis HLS* suporta o tipo de dados inteiro com precisão arbitrária em linguagem C, e o tipo de dados inteiro e de vírgula fixa com precisão arbitrária em linguagem C++.

Consideremos, como exemplo ilustrativo, uma representação em C de um multiplicador 10×10 (ver Código 4-3).

```
#include “ap_int.h”

ap_int<20> multiplica10x10 (ap_int<10> a, ap_int<10> b){

  ap_int<20> p;

  p = a * b;

  return p;

}
```

Código 4-3 – Exemplo de uma função em C com utilização de tipos de dados com precisão arbitrária. No exemplo, são considerados números inteiros com precisão arbitrária.

No exemplo, os operandos foram declarados como inteiros com sinal de 10 bits e o resultado é um inteiro de 20 bits. O tipo de dados inteiro com precisão arbitrária, bem como a operação de multiplicação sobre operandos de precisão arbitrária em C, necessita da biblioteca “`ap_int.h`”.

Considerando, como exemplo, uma multiplicação de dois números de 12 bits representados em vírgula fixa, com sinal, com oito bits para representar a parte inteira (incluindo o sinal) e 4 bits para representar a parte fracionária, teríamos uma descrição similar em C++, mas com a biblioteca de vírgula fixa (ver Código 4-4).

```
#include “ap_fixed.h”

ap_fixed<24,16> multiplica12x12 (ap_fixed<12,8> a, ap_fixed<12,8> b){

  ap_fixed<24,16> p;

  p = a * b;

  return p;

}
```

Código 4-4 – Exemplo de uma função em C++ com utilização de tipos de dados com precisão arbitrária. No exemplo, são considerados números representados em vírgula fixa com precisão arbitrária.

Os operandos `a` e `b` são representados em vírgula fixa com o tipo de dados `ap_fixed<X,Y>`, em que X é o número total de bits e Y é o número de bits utilizados para representar a parte inteira. O resultado é representado com 24 bits, sendo 16

reservados para a parte inteira. Para utilizar o tipo de dados em vírgula fixa com precisão arbitrária, bem como o operador aritmético de vírgula fixa, é necessário incluir a biblioteca "`ap_fixed.h`".

O tipo de dados em vírgula fixa permite também indicar o modo de arredondamento e o modo de excesso. O modo de arredondamento é útil para o caso de a precisão necessária para representar o resultado ser maior do que a que pode ser definida pela representação em vírgula fixa da variável destino do resultado. Os modos de arredondamento mais habituais são o arredondamento para zero e o arredondamento para infinito. O modo de excesso determina o que fazer caso o resultado de uma operação exceda o valor máximo ou mínimo que é possível representar com a variável utilizada para guardar o resultado. Por exemplo, um dos modos é a saturação, que determina que caso o número exceda o máximo representável, então considera-se como resultado esse valor máximo.

A escolha dos modos de arredondamento e de excesso dependem da aplicação. O modo de arredondamento permite limitar o número de bits do resultado ao longo de uma sequência de operações. Por exemplo, ao multiplicar dois números de 10 bits obtemos um resultado de 20. No caso de querermos reduzir o número de bits do resultado, podemos remover alguns bits da parte fracionária aplicando o arredondamento. Por outro lado, reduzir o número de bits da parte inteira por aplicação do modo de excesso pode alterar por completo o resultado, caso estejamos a remover bits significativos do resultado. Por esta razão, quando se projetam circuitos aritméticos de operandos inteiros ou representados com vírgula fixa, procura-se garantir o número de bits da parte inteira da saída do circuito seja suficiente para representar todo o domínio da parte inteira do resultado.

A utilização de tipos de dados com precisão arbitrária na síntese de alto nível será descrita com detalhe no capítulo 7.

4.5 Descrição e Síntese de Interfaces

Os argumentos de uma função e o retorno, caso existam, correspondem às entradas e às saídas do módulo hardware que implementa o corpo da função. No processo de síntese da função, os argumentos são sintetizados em portos e interfaces. A este processo designamos *síntese de interface*.

O tipo de porto necessário a cada argumento e ao retorno depende do tipo de dados de cada um. No caso de acesso a um valor, a função recebe diretamente o valor, enquanto no acesso a um vetor, a função recebe uma referência para o início do vetor. O tipo dos portos sintetizados em cada um dos casos é, por isso, necessariamente diferente, como ilustrado pelo processo de síntese da interface da função descrita no Código 4-5.

```
int acumula(int a, int b[10], int *acc){

  int i;

  for (i = 0; i < 10; i++)
    *acc = a * b[i] + *acc;

  return i;

}
```

Código 4-5 – Exemplo de uma função em C em que os argumentos são especificados de formas diferentes. O argumento `a` *é passado por valor, o argumento* `b` *é um vetor, e o argumento* `acc` *é um ponteiro ou referência. O retorno da função é uma passagem de valor calculado pela função*

Na função exemplo, o argumento `a` é passado por valor, o argumento `b` é passado como um vetor, ou seja, como uma referência para o início do vetor em memória, e o argumento `acc` é passado como ponteiro.

O argumento `a`, que é passado por valor, é traduzido numa entrada do bloco hardware. O argumento `b`, que corresponde a um vetor, é traduzido, por omissão, num porto de acesso a memória, pois é necessário aceder aos 10 valores do vetor. Neste caso, a ferramenta de síntese assume, por omissão, que os valores se encontram guardados numa memória. Consequentemente, é gerada uma interface com portos de dados, de endereço, de leitura e de escrita. O argumento `acc` é lido e escrito no corpo da função e, por isso, é sintetizado em dois portos independentes: um porto de entrada e um porto de saída. O retorno da função é um valor gerado pela função que é implementado com um outro porto de saída do bloco hardware.

A cada um dos argumentos é associado um protocolo de comunicação que depende do tipo do argumento. O conjunto dos protocolos que implementam cada um dos argumentos e o retorno designa-se *protocolos de interface ao nível do porto*.

Para além dos argumentos e do retorno da função, o módulo de hardware pode necessitar de sinais adicionais para controlo do bloco como um todo. Tipicamente, são usados sinais adicionais para indicar que pode iniciar a execução, que pode receber novos dados, que o módulo não está em execução ou que terminou a execução. Estes sinais de controlo operam independentemente dos portos associados aos argumentos e retorno da função. O protocolo associado à interface que agrega estes portos designa-se *protocolo de interface ao nível do bloco*.

No caso do circuito descrito pela função executar em múltiplos ciclos de relógio, é adicionado um terceiro conjunto de sinais que incluem o sinal de relógio (CLK) e o sinal de inicialização (*reset*).

Retornando ao exemplo da função descrita no Código 4-5, o módulo hardware teria o conjunto de interfaces e portos ilustrado na Figura 4-6.

Considerando as implementações por omissão, e o exemplo da função `acumula`, o porto correspondente ao argumento `a` passado por valor é simplesmente uma ligação, ou seja, não tem associado qualquer protocolo. Isto significa que durante a execução do módulo se assume que o valor de `a` permanece estável.

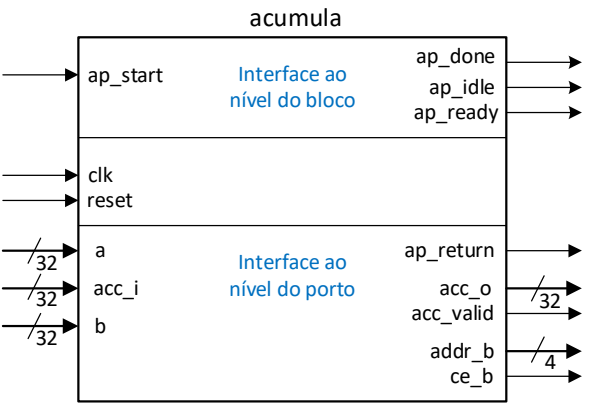


Figura 4-6 – Conjunto de interfaces e portos identificados pela ferramenta de síntese de interface a partir dos argumentos, do retorno da função e do corpo da função

Os portos de entrada e saída correspondentes ao argumento `acc`, passado por ponteiro, também são implementados como uma ligação. Neste caso, uma vez que o argumento é lido e escrito na função, a interface inclui um porto de entrada (`acc_i`) para a leitura e um porto de saída (`acc_o`) para a escrita. Na saída, em muitos casos é importante sinalizar quando é que a saída está pronta para ser lida. Para isso, a síntese de interface adiciona, por omissão, um porto de saída que indica quando os dados estão prontos a ser lidos. No exemplo, foi adicionada a saída `acc_valid`.

Para o argumento do tipo vetor `b`, foram criados portos de endereço (`addr_b`) e de ativação (`ce_b`, `chip_enable`). Neste caso, não é necessário um porto de controlo de escrita, uma vez que o vetor é apenas lido.

Para o retorno da função, é gerado um porto de saída (`ap_return`) que corresponde simplesmente a uma ligação. Para saber quando o valor de retorno é válido, temos de olhar para os portos da interface de controlo do bloco.

Existem três protocolos ao nível do bloco designados (no *Vitis HLS*) `ap_ctrl_none`, `ap_ctrl_hs`, `ap_ctrl_chain`, que serão detalhados na parte II. Neste exemplo, é implementado (por omissão) o protocolo `ap_ctrl_hs` que significa que o componente executa sequencialmente, ou seja, o componente tem de terminar uma execução antes de

começar a execução seguinte. O protocolo `ap_ctrl_none`, significa que não é utilizada uma interface ao nível do protocolo, ou seja, que não seriam gerados quaisquer portos de controlo do bloco. O protocolo `ap_ctrl_chain` é uma variação do protocolo `ap_ctrl_hs` que inclui um porto de entrada adicional, `ap_continue`, para controlar o reinício da execução.

As interfaces também podem ser implementadas de acordo com um protocolo AXI. Dependendo do tipo de argumentos, pode ser usada uma das três variantes do protocolo AXI, nomeadamente *AXI4-Stream*, *AXI4-Lite* e *AXI4-Full*. Os tipos de implementação disponíveis dependem de cada ferramenta HLS específica. A ferramenta *Vitis HLS* permite implementar a interface escravo do protocolo *AXI4-Lite*, a interface mestre do protocolo *AXI4-Full*, e as interfaces mestre e escravo do protocolo *AXI4-Stream*. A utilização de um protocolo AXI para as interfaces é uma opção de projeto e depende da utilização que se pretende dar ao bloco. Opcionalmente, pode considerar-se a implementação do bloco com ou sem interfaces AXI. A não utilização de interfaces AXI irá gerar um bloco hardware com uma interface nativa que não obedece a uma norma de interface. A utilização de uma interface AXI garante que o bloco pode ser interligado a outros blocos que implementem a mesma interface AXI. Torna-se assim um bloco facilmente integrável noutros projetos de hardware.

Apesar de toda a automatização da ferramenta de HLS associada à síntese da interface, muitas das opções de síntese da interface podem ser controladas pelo projetista. Existe um conjunto de diretivas associadas a síntese dos portos e das interfaces que permitem associar protocolos às interfaces. Os métodos e as opções de interface de síntese disponíveis ao projetista serão descritos em detalhe no capítulo 6.

4.6 Otimização do Circuito

O projeto de um circuito digital procura não só garantir que o circuito realiza a funcionalidade pretendida, mas também otimizar métricas de projeto ou satisfazer restrições relativas a essas métricas. As métricas mais comuns são o desempenho e a área.

Em geral, o espaço de projeto de um circuito, ou seja, o conjunto de soluções possíveis, é demasiado vasto para ser explorado exaustivamente pelas ferramentas de síntese. Por outro lado, não explorar ou comparar algumas destas soluções pode dar origem a uma solução final pouco otimizada que não cumpre os requisitos de projeto.

Assim, as ferramentas de HLS optam por uma solução intermédia em que o projetista orienta a ferramenta de HLS como forma de exploração do espaço de projeto de modo a gerar uma solução que cumpra os objetivos de desempenho e de área. Esta intervenção do projetista é feita através das designadas *diretivas*. Após descrever o circuito, o projetista especifica e associa diretivas às estruturas de programação, como estruturas de repetição ou de condição, às variáveis, aos portos/interfaces, etc.

Por exemplo, consideremos uma função para o cálculo do produto interno entre dois vetores (ver Código 4-6)

```
int prodInterno(int a[10], int b[10]){

  int i, pi, mul;

  pi = 0;
  for (i = 0; i < 10; i++)
      mul = a[i] * b[i];
      pi = pi + mul;

  return pi;

}
```

Código 4-6 – Exemplo de uma função em C para o cálculo do produto interno entre dois vetores de tamanho 10. Os argumentos são especificados como vetores de inteiros e o resultado é retornado como um inteiro

Sem otimizações, cada iteração da estrutura `for` é escalonada em três ciclos de relógio (ver Figura 4-7).

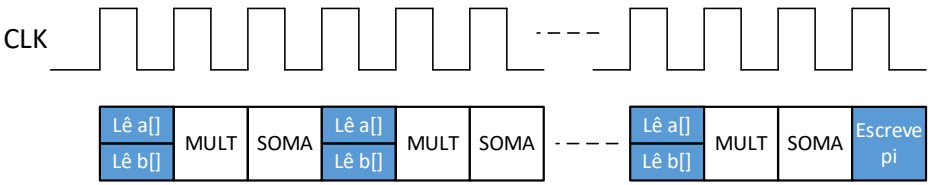


Figura 4-7 – Escalonamento da função `prodInterno` sem utilização de pipeline.

Por cada iteração é lido um elemento de cada um dos vetores e é feita uma multiplicação seguida de uma soma. O processo repete-se 10 vezes e no fim é escrito o resultado, num total de 31 ciclos de relógio. Uma nova iteração só pode iniciar após terminar a anterior, ou seja, após três ciclos de relógio.

Para otimizar a execução do módulo hardware pode considerar-se o método de *pipeline*. Para tal, indica-se à ferramenta de HLS que deve escalonar o circuito em *pipeline*, recorrendo à diretiva de *pipeline* (ver Figura 4-8).

A diretiva de *pipeline* permite iniciar uma nova iteração antes de a anterior terminar, possibilitando que as operações executem em sobreposição. No exemplo, consegue-se implementar o escalonamento com uma nova iteração a iniciar a cada ciclo de relógio, ou seja, cada sequência de leitura, multiplicação e soma pode ser iniciada a cada ciclo de relógio. O total de ciclos de relógio para executar a função é de 13 (ao décimo ciclo estamos a ler o último par de valores a que se segue uma multiplicação, uma soma e a escrita final).

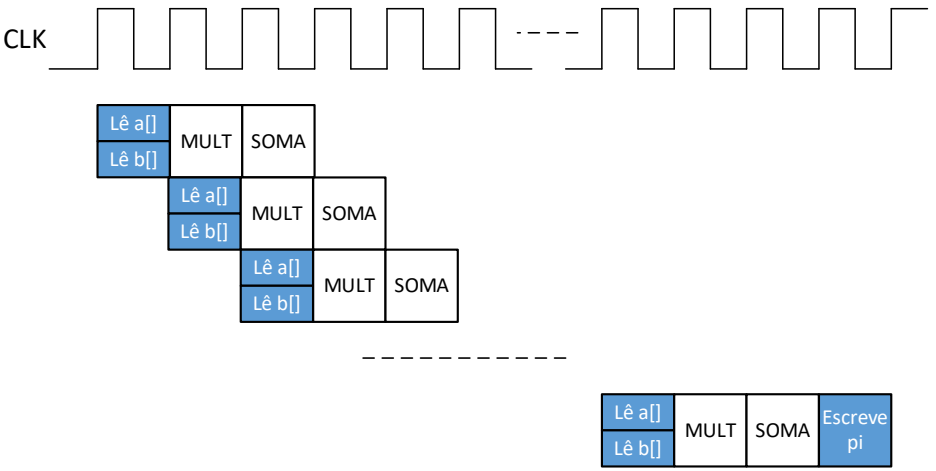


Figura 4-8 – Escalonamento da função `prodInterno` com utilização de pipeline.

O projetista consegue, assim, forçar a ferramenta de HLS a escalonar um determinado fluxo de dados numa estrutura em *pipeline*. Se por um lado ajuda a ferramenta a decidir como implementar o fluxo de dados, por outro permite ao projetista decidir sobre a implementação desse fluxo de dados.

Um outro exemplo de diretivas são as que incidem sobre a estrutura dos vetores e, consequentemente, das memórias utilizadas para os armazenar. Por vezes, a exploração do paralelismo ou do método de *pipeline* só é possível se se conseguir aceder a múltiplos dados em paralelo. Consideremos de novo o exemplo do produto interno. Na solução anterior, assumiu-se que era possível ler um elemento do vetor *a* e um elemento do vetor *b* em paralelo. Isto implica dois portos de acesso à memória.

Consideremos o exemplo anterior admitindo apenas um porto de leitura. Com esta limitação, só é possível ler um valor de cada vez (ver escalonamento na Figura 4-9).

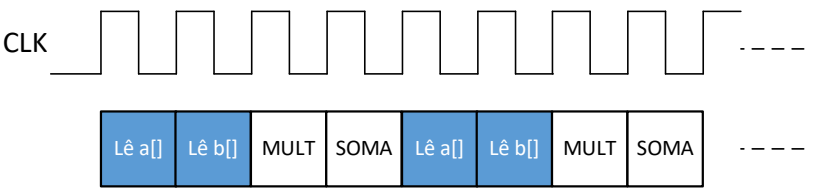


Figura 4-9 – Escalonamento da função `prodInterno` sem utilização de pipeline e com apenas um porto de acesso aos dados.

A leitura dos valores dos vetores tem de ser feita em série. Assim, cada iteração passa a demorar 4 ciclos de relógio, com um total de execução de 41 ciclos, incluindo o de escrita.

Mesmo que apliquemos a diretiva de *pipeline*, não se consegue iniciar uma nova iteração a cada ciclo de relógio devido à impossibilidade de ter duas leituras por ciclo (ver Figura 4-10).

Como se verifica através do escalonamento da função com *pipeline*, uma nova iteração inicia-se apenas a cada dois ciclos de relógio, já que não é possível ter duas leituras por ciclo de relógio. No total, temos 23 ciclos de execução da função, mais 10 ciclos do que o conseguido na primeira forma de *pipeline* com duas leituras em paralelo.

Para melhorar a latência do circuito, existe um conjunto de diretivas que permitem orientar a ferramenta de síntese a organizar os dados em memória. Por exemplo, armazenando os dados em várias memórias, em vez de apenas uma, consegue-se aceder a múltiplos valores em paralelo. Esta seria a solução para recuperar a solução de *pipeline* com a iniciação de uma nova iteração a cada ciclo de relógio. Também é possível um processo contrário, em que se juntam múltiplos vetores num único vetor de maior dimensão de modo a reduzir o número de memórias necessárias para o armazenamento dos elementos ao mesmo tempo que se garante a leitura de um elemento de cada vetor por ciclo de relógio.

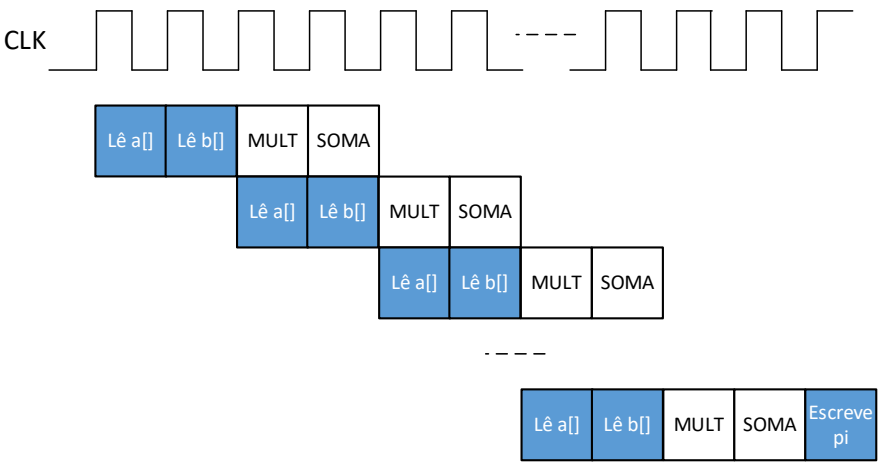


Figura 4-10 – Escalonamento da função `prodInterno` *com utilização de pipeline e com apenas um porto de acesso aos dados.*

Em muitos casos, as diretivas estão correlacionadas entre si. Por exemplo, ao desenrolar uma estrutura de repetição de modo a expor mais paralelismo, pode implicar que seja necessário acesso paralelo a múltiplos dados. Neste caso, convém aplicar uma diretiva de partição dos vetores que permita o acesso paralelo a múltiplos valores.

As diretivas de otimização serão abordadas em detalhe na segunda parte do livro enquadradas no assunto de cada um dos capítulos.

4.7 Descrição HLS com Múltiplas Funções

Os exemplos que abordámos até agora consideram um circuito descrito com apenas uma função. No entanto, um sistema digital é, em geral, formado por vários módulos que executam em paralelo ou em sequência. As ferramentas de HLS suportam a descrição de circuitos digitais usando múltiplas funções interligadas entre si.

Consideremos um primeiro exemplo ilustrativo muito simples formado por dois módulos que comunicam entre si, descritos com duas funções (ver Código 4-7).

```
void FUNC_0(int dadosIN[4], int dadosOUT[4]){

  int i;
  ...
  for (i = 0; i < 4; i++)
    Corpo da estrutura for da função FUNC_0
  ...

}

void FUNC_1(int dadosIN[4], int dadosOUT[4]){

  int i;
  ...
  for (i = 4; i > 0; i--)
    Corpo da estrutura for da função FUNC_1
  ...

}

void FUNC_TOPO(int dadosIN[4], int dadosOUT[4]){

  int dados[4], i;

  for (i = 0; i < 3){
    FUNC_0(dadosIN, dados);
    FUNC_1(dados, dadosOUT);
  }

}
```

Código 4-7 – Exemplo de uma descrição HLS com duas funções. A função FUNC_0 produz dados que são consumidos pela função FUNC_1.

No exemplo, o componente `FUNC_0` gera um vetor de dados que é lido pelo `FUNC_1`. Este processa os dados de entrada pela ordem inversa, pelo que tem de esperar por receber todos os dados antes de iniciar o processamento. Considerando um escalonamento dos blocos de forma sequencial, uma iteração da estrutura *for* do módulo de topo apenas inicia após a execução de ambos os módulos (ver Figura 4-11).

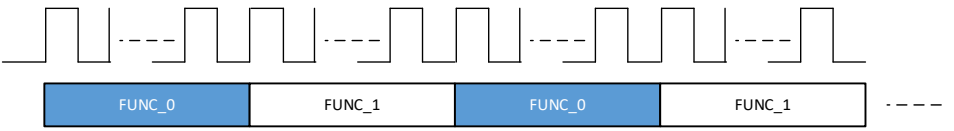


Figura 4-11 – Escalonamento do módulo de tipo, `FUNC_TOPO`, sem otimizar o fluxo de execução.

A estrutura *for* da função de topo descreve uma execução sequencial dos módulos, como se se tratasse de apenas um componente que recebe os dados de entrada (`DATA_IN`) e produz os dados de saída (`DATA_OUT`).

A descrição pode, em alternativa, ser implementada como uma hierarquia, em que as funções são sintetizadas individualmente e os componentes interligados com canais de comunicação. Neste caso, pode ser considerada uma implementação *pipeline* da estrutura *for* da função de topo, em que os componentes, `FUNC_0` e `FUNC_1`, executam em *pipeline*. Considerando que ambos os componentes têm o mesmo tempo de execução, obteríamos o escalonamento da Figura 4-12.

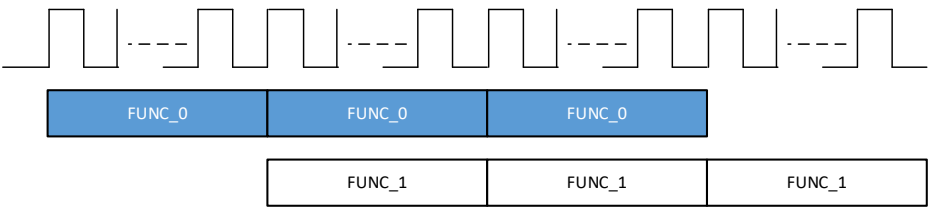


Figura 4-12 – Escalonamento da função `FUNC_TOPO` considerando uma implementação em hierarquia com recurso à otimização com pipeline em que ambas as funções têm o mesmo tempo de execução.

Neste caso, os dados produzidos na primeira execução de `FUNC_0` são consumidos pelo componente `FUNC_1` após a execução completa de `FUNC_0`. No entanto, a primeira execução de `FUNC_1` pode ocorrer em paralelo com a segunda de `FUNC_0`.

No caso genérico em que os componentes têm tempos de execução diferentes, aplica-se a mesma solução em *pipeline*, mas uma nova execução do módulo com menor tempo de execução só pode iniciar após o fim da execução do módulo mais lento, pois necessita dos dados que serão produzidos por este.

No exemplo da Figura 4-13, o módulo `FUNC_1` tem um menor tempo de execução, pelo que só volta a executar após `FUNC_0` terminar e produzir os dados de saída.

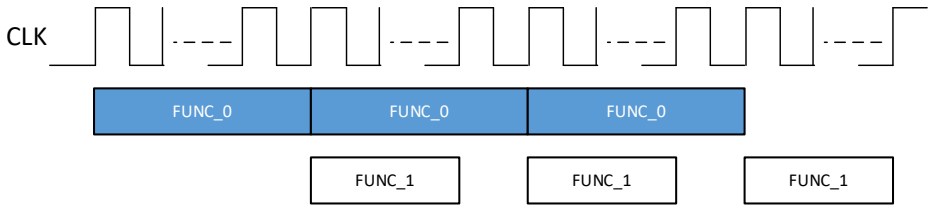


Figura 4-13 – Escalonamento da função `FUNC_TOPO` considerando uma implementação em hierarquia com recurso à otimização com pipeline em que ambas as funções não têm o mesmo tempo de execução.

Este tipo de execução requer uma duplicação de memória no canal de comunicação para guardar os resultados de saída de um módulo para serem utilizados na próxima iteração do módulo seguinte. Isto é, a memória do canal de comunicação contém os dados que estão a ser lidos pelo módulo subsequente e os dados que estão a ser produzidos pelo módulo anterior.

Na descrição de um circuito também podem ocorrer várias chamadas da mesma função dentro de uma iteração da função de topo. Por exemplo, consideremos uma função de multiplicação que é chamada múltiplas vezes por cada iteração da estrutura *for* da função de topo (ver Código 4-8)

```
int MULT(int a, int b){

return a * b;

}

void FUNC_TOPO(int x[4], int y[4], int resultado){

  int acc, i;

  acc = 0;
  for (i = 0; i < 3; i++){
    acc = acc + mult(x[i], x[i]) + mult(y[i], y[i]);
  }

}
```

Código 4-8 – Exemplo de uma descrição HLS com uma função chamada múltiplas vezes dentro da estrutura `for` da função de topo.

No exemplo, a função `MULT` é chamada duas vezes em cada iteração da estrutura *for* da função `FUNC_TOPO`. Na implementação da função em hardware há duas formas distintas de implementar o circuito. Uma das abordagens implementa um módulo para a execução da função `MULT` que depois é partilhado pelas várias execuções da multiplicação. No exemplo, a execução das multiplicações teria de ser escalonada em série, pois existe apenas um módulo de multiplicação. Em alternativa, pode considerar-se gerar um módulo por cada

chamada à função `MULT`. Neste caso, a ferramenta de síntese procede ao que se designa *inline* da função, ou seja, remove a hierarquia da função. Esta operação reduz o custo de partilha de um módulo hardware, permite otimizar a simplificação lógica entre funções em hierarquias diferentes possibilitando, em muitos casos, a aplicação de outras diretivas de otimização. A desvantagem reside no aumento de recursos hardware, pois a mesma função é implementada tantas vezes quanto o número de chamadas. De uma forma geral, as ferramentas de síntese consideram o processo de `inline`, quando as funções são relativamente pequenas.

A descrição detalhada do projeto hierárquico com a ferramenta de HLS será feita no capítulo 8.

4.8 Construções de Programação não Suportadas Pela Ferramenta HLS

A ferramenta de síntese de alto nível suporta quase todas as construções de programação das linguagens C e C++. Contudo, devemos ter em conta que todas as operações que resultam na chamada de funções do sistema operativo não podem ser utilizadas na descrição das funções a traduzir para hardware, pois não são sintetizáveis. Além disso, algumas construções da linguagem estão de certa forma limitadas na forma como podem ser utilizadas. No caso da descrição de ficheiros de teste, não existem limitações, pois estes ficheiros são usados apenas para testar o circuito e não para serem sintetizados.

A função mais comum na programação em C é o `printf()` ou `fprintf()`. Estas funções não são utilizadas na descrição de algoritmos, mas apenas para visualizar dados de entrada ou de saída numa fase de simulação do circuito. Traduzem-se em chamadas ao sistema, pelo que não podem ser utilizadas na descrição do circuito. Para facilitar a descrição de funções que tanto podem ser usadas para programação, como para serem sintetizadas, utilizam-se macros. Por exemplo, o *Vitis HLS* tem definida a macro `__SYNTHESIS__` para este efeito. Esta permite que o projetista utilize um único código para simulação e para síntese, bastando inserir a macro no código que orienta o pré-processador da ferramenta de síntese.

Consideremos, como exemplo, a função definida no Código 4-9.

```
int multiplyAdd (int a, int b, int c){
      int ma;

    #ifndef __SYNTHESIS__
      printf("Executa modulo multiplyAdd");
    #endif

      ma = a * b + c;
      return ma;
}
```

Código 4-9 – Exemplo de uma função em C com a utilização da macro `__SYNTHESIS__`.

No exemplo, a função `printf()` só é executada caso a macro `__SYNTHESIS__` não esteja definida. Isto permite que a função faça uma escrita na consola de saída quando usada em simulação e que a função `printf()` seja ignorada quando em modo de síntese, pois a macro está definida.

Um outro conjunto de funções usualmente utilizadas em programação são as de alocação de memória. Qualquer função de alocação recorre a funções do sistema e, como tal, não é sintetizável. Todas as operações de alocação dinâmica de memória têm de ser transformadas em representações equivalentes com a alocação estática da memória necessária.

Consideremos um outro exemplo de descrição de uma função que permite facilmente alterar a versão da função entre alocação dinâmica de memória e alocação estática (ver Código 4-10).

```
int exemploMalloc(int a[64], size) {

int i, zero;

#ifdef __SYNTHESIS__
   int _localMem[64];
   int *localMem = _localMem;

#else
   int *localMem = malloc (size * sizeof(int));
#endif

for (i = 0; i < size; i++)
       *(localMem+i) = a[i];

for (i = 0; i < size; i++)
       if (*(localMem+i) == 0)
              return 0

return -1;
}
```

Código 4-10 – Exemplo de descrição de uma função em C com possibilidade de alterar entre alocação dinâmica e alocação estática com a simples definição de uma macro.

A utilização de um ponteiro para a primeira posição de um valor definido estaticamente, caso a macro `__SYNTHESIS__` esteja definida, permite que o mesmo código seja utilizado sem alterações para simulação e para síntese.

O ponteiro é um tipo de dados da linguagem C que é parcialmente suportado pela ferramenta de síntese de alto nível. Algumas descrições com ponteiros não suportadas são:

- Vetores de ponteiros em que os elementos do vetor são ponteiros. A ferramenta de síntese requer que os ponteiros apontem para escalares ou para vetores de escalares;

- *Ponteiros para funções.*

Por fim, as funções recursivas também não são suportadas pela ferramenta de síntese de alto nível.

4.9 Verificação do Circuito

A verificação da funcionalidade do circuito e do cumprimento das restrições e objetivos de projeto é uma etapa fundamental no projeto de circuitos digitais. Recordemos o fluxo de projeto baseado em síntese de alto nível na Figura 4-14.

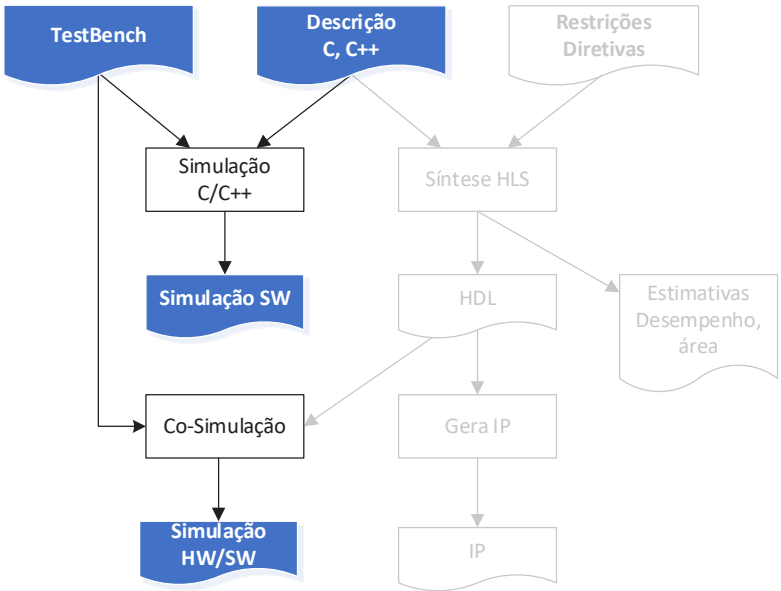


Figura 4-14 – Fluxo de verificação ao longo da metodologia de projeto baseada em síntese de alto nível.

A verificação do circuito decorre em paralelo com o fluxo de síntese. É constituída por duas formas de simulação: simulação da descrição C/C++ e simulação conjunta ou co-simulação do hardware com o software.

A primeira forma de verificação é utilizada para confirmar se a função descrita em C/C++ está correta, ou seja, de acordo com a funcionalidade pretendida pelo projetista. Tratando-se da verificação de uma função descrita em C/C++, o processo é bastante rápido quando comparado com uma simulação em hardware. Esta verificação pré-síntese é fundamental para melhorar a produtividade de projeto, pois é rápida e reduz o tempo de projeto.

Para verificar se uma função está bem descrita, basta chamar a função com a concretização dos valores dos argumentos e verificar o resultado da função. Executa-se a função para

vários valores dos argumentos, assumindo que o conjunto de testes é representativo do todo. De modo a automatizar o processo de verificação, ou seja, de modo a permitir a verificação automática dos resultados, e ao mesmo tempo facilitar a reverificação sempre que são feitas alterações às funções, aconselha-se o uso de uma bancada de teste (*testbench*).

A *testbench* é um programa em C/C++ que gera diferentes combinações dos argumentos da função, chama a função que está a ser verificada com esses argumentos e verifica automaticamente a correção dos resultados. O *testbench* pode conter construções não sintetizáveis, pois trata-se de uma descrição software que é usada apenas para verificação e não se pretende que seja sintetizada. Como tal, é boa prática separar a função a ser sintetizada do código de *testbench*. Consideremos de novo o exemplo do produto interno com dois vetores de dimensão 1000. Começamos por criar um ficheiro (ficheiro de cabeçalho com extensão `.h`) com as definições dos tamanhos dos vetores e da função a ser testada (`prodInterno.h`).

```
#ifndef _prodInterno_
#define _prodInterno_

int prodInterno(int a[], int b[]);

#define NUM_TEST_VECTORS 100
#define VECTOR_SIZE 1000

#endif
```

Código 4-11 – Ficheiro de definições - `prodInterno.h`.

Neste exemplo, o ficheiro de definições (ver Código 4-11) é bastante simples contendo apenas o protótipo da função a testar e o tamanho dos vetores. É comum utilizar este ficheiro para declarar constantes, tipos de dados, etc.

A função principal descreve o produto interno entre os dois vetores com um ciclo de repetição (ver Código 4-12).

```
#include "prodInterno.h"

int prodInterno(int a[VECTOR_SIZE], int b[VECTOR_SIZE]){

  int i, pi, mul;

  pi = 0;
  for (i = 0; i < VECTOR_SIZE; i++)
      mul = a[i] * b[i];
      pi = pi + mul;

  return pi;

}
```

Código 4-12 – Função de cálculo do produto interno entre dois vetores de tamanho 1000.

Para esta função, consideremos uma *testbench* que lê os dados dos vetores de um ficheiro e confirma o resultado comparando com o valor correto também obtido de um ficheiro. A *testbench* para o exemplo da função de produto interno é descrito no Código 4-13 .

```
#include “prodInterno.h”

void main() {

FILE *fp, *fp1;
int i, j, data, result, goldenResult;
int a[VECTOR_SIZE], b[VECTOR_SIZE];

fp = fopen(testDATA.dat,r);
fp1 = fopen(goldenDATA.dat,r);
for (i = 0; i < NUM_TEST_VECTORS; i++){
    for (j = 0; j < VECTOR_SIZE; j++){
       fscanf(fp, %d, &data);
       a[j] = data;
       fscanf(fp, %d, &data);
       b[j] = data;
       fscanf(fp1, %d, &goldenResult);
    }

    result = prodInterno(a, b);

    if(result != goldenResult)
       printf(“O resultado para %d vetores não está correto\n”, i);
}

fclose(fp);
fclose(fp1);
}
```

Código 4-13 – Ficheiro testbench da função `prodInterno()`.

O código lê um par de vetores de cada vez e o respetivo resultado esperado. Chama a função com estes vetores e compara o valor retornado pela função em teste com o resultado esperado. Caso não coincidam, é gerada uma mensagem de aviso.

O processo é repetido para vários pares de vetores. Sabemos que não é possível testar todos os casos, pelo que a escolha dos vetores de teste deve garantir a maior diversidade possível e incluir os casos considerados mais “difíceis” de tratar pela função.

A alteração da função a sintetizar não implica qualquer alteração ao ficheiro *testbench*, exceto se foram alterados os argumentos da função.

A co-simulação ou simulação hardware/software ocorre sobre o circuito gerado pela ferramenta de síntese. É uma forma de verificação pós-síntese em que é simulado o componente hardware sintetizado (descrição hardware), ao contrário da simulação descrita anteriormente em que foi simulada a descrição software. A simulação segue o fluxo descrito na Figura 4-15.

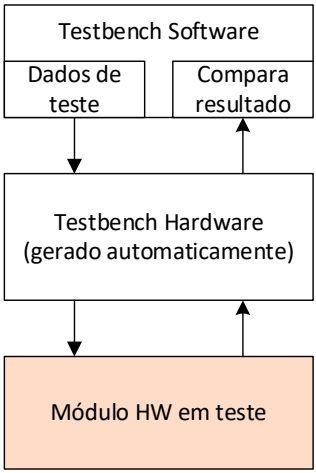


Figura 4-15 – Fluxo de co-simulação ou simulação hardware/software.

O mesmo *testbench* usado na simulação software é utilizado na co-simulação. Este gera os vetores de simulação que são enviados para um segundo *testbench* de hardware (descrição HDL) gerado automaticamente pela ferramenta de HLS. Este segundo *testbench* define os vetores de simulação de acordo com o protocolo associado aos portos da função sintetizada. Os vetores de teste são recebidos pelo simulador de hardware e aplicados ao módulo sintetizado. Em resposta, os resultados da simulação são enviados de volta até ao *testbench* de software para serem comparados com os resultados de referência.

Ao passo que a simulação software verifica se a funcionalidade está bem descrita em C/C++, a co-simulação verifica se a síntese produziu um circuito que cumpre a funcionalidade e os requisitos de projeto. É uma forma de simulação de hardware apoiada por um *testbench* em software com a grande vantagem de o *testbench* hardware ser gerado automaticamente.

PARTE II

Síntese de Alto Nível

5 Síntese de Estruturas de Repetição e de Condição

Um algoritmo em C/C++ é frequentemente descrito com estruturas de repetição e estruturas de condição. Qualquer uma destas construções pode ser reorganizada de modo a otimizar a implementação da função em hardware. Por exemplo, as estruturas de repetição podem ser desenroladas ou unidas e as de condição podem ser combinadas de modo a partilhar recursos. Sendo a estrutura de repetição uma execução repetida de um conjunto de operações, é também possível melhorar o seu tempo de execução com a técnica de *pipeline*.

As transformações que se podem aplicar a este tipo de estruturas de programação podem ser conseguidas alterando diretamente o código ou aplicando diretivas de síntese que, indiretamente, alteram a estrutura do código. É importante perceber de que forma podemos modificar o código ou aplicar diretivas de modo a otimizar a sua implementação, quais as condições em que podem ser aplicadas, quais as interdependências entre transformações e quais as implicações que uma transformação tem sobre a solução hardware.

Neste capítulo, serão abordados os mecanismos da ferramenta de síntese de alto nível que permitem transformar e otimizar as estruturas de repetição e de condição de modo a reduzir os recursos hardware necessários às suas implementações e/ou a melhorar os tempos de execução dos componentes hardware.

5.1 Síntese de Estruturas de Repetição

Uma estrutura de repetição permite descrever a execução repetitiva de um conjunto de operações. O número de vezes que o conjunto de operações é executado (número de iterações da estrutura de repetição) pode ser constante ou variável. Quando o número de repetições é constante diz-se que o limite é incondicional. Se o número de repetições é variável então dizemos que tem um limite condicional.

As estruturas de repetição em C/C++ são bem conhecidas. Contudo, para clarificar os termos utilizados ao longo do capítulo, segue-se uma breve descrição da sintaxe dos vários tipos.

Numa função C/C++ podemos encontrar três estruturas distintas de repetição: `for`, `while` e `do/while`. A estrutura `for` tem quatro campos de acordo com a seguinte sintaxe:

```
ETIQUETA: for (inicialização; condição; atualização) {
            corpo da estrutura;
}
```

O campo `inicialização` consiste numa ou mais atribuições de inicialização de uma ou mais variáveis. O campo `condição` é uma expressão que é calculada inicialmente e após cada iteração para determinar se executa ou não a próxima iteração. O campo atualização consiste numa ou mais expressões de atualização das variáveis utilizadas na estrutura `for`. Por fim, o `corpo da estrutura` é o conjunto de operações realizadas em cada iteração. O corpo da estrutura é executado enquanto se verificarem as condições especificadas no campo `condição`.

A estrutura `while` tem dois campos de acordo com a seguinte sintaxe:

```
ETIQUETA: while (condição) {
            corpo da estrutura;
}
```

Os campos `condição` e `corpo da estrutura` têm a mesma interpretação e função dos campos respetivos na estrutura `for`. O corpo da estrutura repete-se enquanto a `condição` for verdadeira. Uma estrutura `for` pode ser descrita com uma estrutura `while` e vice-versa. Para converter uma estrutura `for` numa `while` basta executar a inicialização antes de iniciar o `while` e passar as expressões de atualização para o corpo do `while` (após o corpo original da estrutura). Teríamos a seguinte estrutura:

```
inicialização;
ETIQUETA: while (condição) {
      corpo da estrutura;
      atualização;
}
```

A estrutura `do/while` tem os mesmos dois campos do `while`, mas o corpo da estrutura é executado pelo menos uma vez, ao passo que no `while` o corpo pode nunca ser executado. A sintaxe é a seguinte:

```
ETIQUETA: do {
            corpo da estrutura;
} while (condição);
```

NOTA: Para simplificar a linguagem, e sempre que não gerar ambiguidade, usaremos uma designação reduzida para fazer referência às estruturas de repetição. Por exemplo, em vez

de usar a expressão *estrutura de repetição* `for`, usaremos *estrutura* `for`, ou simplesmente `for`.

Uma estrutura de repetição pode surgir dentro de outra. Neste caso, dizemos que temos estruturas de repetição entrelaçadas. O caso mais comum é termos estruturas `for` entrelaçadas. Por exemplo, consideremos a escrita de dois `for` incondicional entrelaçados:

```
REPETEL1: for (i = 0; i < 3; i++) {
    REPETEL2: for (j = 0; j < 3; j++) {
          p[i,j] = a[i]*b[i+j*3];
    }
```

No exemplo, o *`for`* interno (`REPETEL2`) é executado 3 vezes por cada iteração do `for` externo (`REPETEL1`). No total, o corpo do `for` interno é executado 9 vezes.

Existe um conjunto de métricas de desempenho associadas às estruturas de repetição que permitem ao projetista identificar se os requisitos estão a ser cumpridos e ainda de que forma se pode melhorar o desempenho, nomeadamente:

Latência da função – Número de ciclos de relógio necessários para que a função calcule todos os valores de saída;

Latência da estrutura de repetição – Número de ciclos de relógio necessários para executar todas as iterações de uma estrutura de repetição;

Latência da iteração da estrutura – Número de ciclos de relógio necessários para completar uma iteração da estrutura de repetição;

Intervalo de iniciação (II) – Número de ciclos de relógio necessários até que a função ou a estrutura de repetição possa aceitar novos dados.

Em todos os casos, conhecendo o período de relógio, a latência pode ser expressa num intervalo de tempo.

5.1.1 Síntese de uma Estrutura de Repetição

Consideremos a função `acumula`, descrita no Código 5-1, com apenas um `for` incondicional para acumular os valores de um vetor.

```
void acumula (int a[4], int *dout){
  int acc = 0, i;

  repete: for (i = 0; i < 4; i++)
  #pragma HLS PIPELINE off
       acc = acc + a[i];

  *dout = acc;
}
```

Código 5-1 - Exemplo da função `acumula` que utiliza uma estrutura for incondicional para acumular os valores de um vetor, sem pipeline

Por omissão, o *Vitis HLS* aplica a técnica de *pipeline* às estruturas de repetição. A utilização ou não de *pipeline* pode ser controlada explicitamente pelo projetista através da diretiva `HLS PIPELINE`.

`#pragma HLS PIPELINE [II=<int>] [off] [rewind] style=<value>`

`II=<int>` - especifica o intervalo de iniciação da estrutura de repetição. Por omissão, II=1.

`off` – opção que impede a aplicação automática de *pipeline*

`rewind` – opção que permite *pipeline* contínuo entre execuções completas da estrutura de repetição.

`style` – especifica o tipo de *pipeline*:

- stp (valor por omissão): o *pipeline* só avança de houver dados;
- flp: permite fazer *flush* ao *pipeline*;
- frp: o *pipeline* avança mesmo sem dados e permite *flush*.

A função foi sintetizada com e sem *pipeline* para se identificar a diferença entre ambas as implementações. Não existindo *pipeline*, todas as operações são executadas sequencialmente em vários ciclos do sinal de relógio (`CLK`). Existe um primeiro estado de inicialização dos valores de `acc` e `i`, seguido das quatro iterações do `for` e de um ciclo de relógio final para retornar o valor de `acc` (ver escalonamento do exemplo na Figura 5-1).

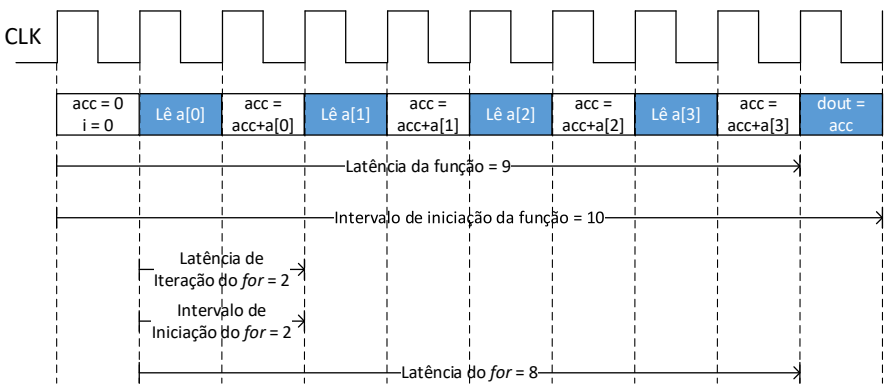


Figura 5-1 – Escalonamento do ciclo simples sem pipeline

Como indicado na figura, a latência da função é de 9 ciclos de relógio: 1 ciclo de relógio para inicializar o valor de `acc` e de `i` a 0, mais 8 ciclos de relógio para executar as iterações. Ao fim dos 9 ciclos de relógio, o valor de saída está calculado. O intervalo de iniciação da função é de 10 ciclos de relógio antes de se poder reiniciar a função com novos valores, pois inclui

um ciclo final para devolver o valor de *acc*. A latência de iteração do `for` é de 2 ciclos: um para ler o valor do vetor e outro para adicionar este valor ao `acc`. Não existindo ciclos de relógio entre as iterações do `for`, o intervalo de iniciação do `for` também é de 2. A latência do `for` é assim de 8 ciclos de relógio, ou seja, o número de iterações do *for*, 4, multiplicado pela latência de iteração do `for`, 2.

O diagrama temporal apenas apresenta as operações sobre os dados, incluindo a leitura e a escrita. No entanto, em paralelo ocorrem as operações de controlo, nomeadamente o incremento e o teste da variável de controlo da estrutura `for`. Estas operações ocorrem em paralelo com a leitura dos dados.

No caso de aplicarmos *pipeline* à estrutura de repetição, conseguimos reduzir o intervalo de iniciação do `for` (ver escalonamento na Figura 5-2)

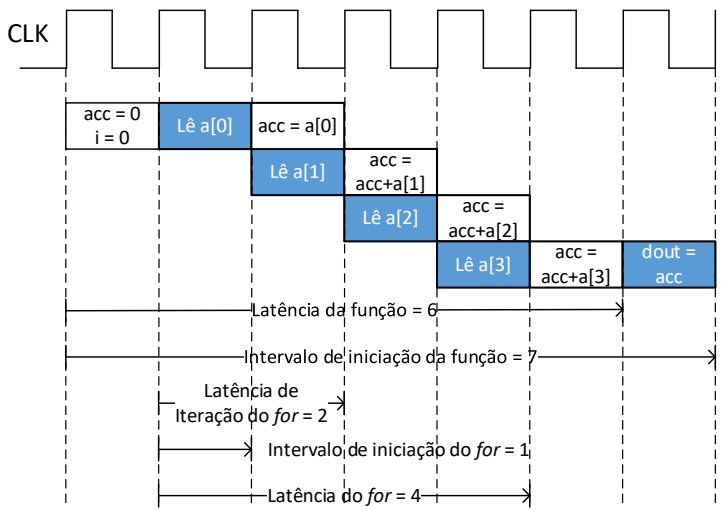


Figura 5-2 – Escalonamento do `for` com pipeline

Com a aplicação de *pipeline*, é possível fazer a acumulação do elemento do vetor em paralelo com a leitura do novo elemento. Consegue-se assim reduzir o intervalo de iniciação da função de 10 para 7 ciclos de relógio. A eficácia do método de *pipeline* é indicada pelo intervalo de iniciação da estrutura de repetição. No exemplo, esta métrica é 1, pelo que é possível iniciar uma iteração do `for` a cada ciclo de relógio. Quando se aplica *pipeline* a uma estrutura de repetição, procura-se sempre conseguir o melhor caso de um intervalo de iniciação igual a 1, mas nem sempre é possível, como veremos mais à frente. A latência de iteração do `for` mantém-se nos 2 ciclos de relógio, uma vez que as operações de leitura de um elemento do vetor e a sua acumulação são sequenciais devido à dependência de dados entre as duas.

A Figura 5-3 ilustra uma implementação da estrutura `for`.

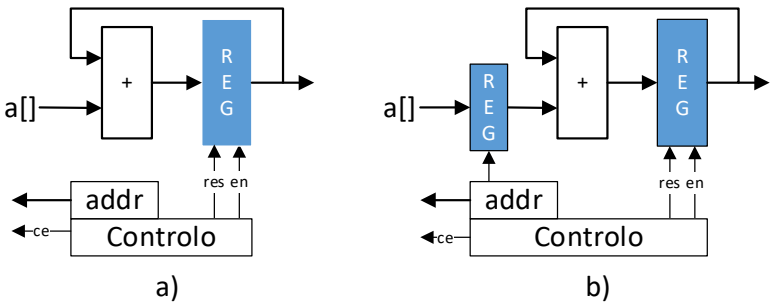


*Figura 5-3 – Implementação hardware d*a estrutura `for`: a) sem pipeline e b) *com pipeline*

O circuito utiliza um registo para implementar a acumulação, colocado inicialmente a 0 com uma entrada de inicialização (`res`). A diferença entre as duas implementações reside no registo de entrada. Na solução com *pipeline*, o registo de entrada guarda o valor lido para ser somado no ciclo de relógio seguinte.

5.1.2 Desenrolamento de Estruturas de Repetição Incondicionais

O desenrolamento de estruturas de repetição permite expor o paralelismo existente entre operações de iterações diferentes, possibilitando escalonar múltiplas iterações em paralelo. Quanto maior o fator de desenrolamento, maior será o número de operações que podem ser executadas por ciclo de relógio, assumindo que não existem dependências de dados que o impeça. O desenrolamento da estrutura pode ser feito manualmente ou recorrendo à diretiva `HLS UNROLL`.

Para ilustrar a aplicação do desenrolamento da estrutura de repetição, consideremos o exemplo do acumulador com desenrolamento manual (Código 5-2) e desenrolamento automático (Código 5-3).

```
#pragma HLS UNROLL factor=<N> skip_exit_check off=true

factor=<N> - especifica o número de vezes que o corpo da estrutura de
repetição é desenrolado. Se não for especificado, a estrutura é desenrolada
por completo.

skip_exit_check – opção considerada no caso de desenrolamento parcial. É
necessário quando o fator de desenrolamento não é múltiplo do número de
iterações.

off – desabilita o desenrolamento do ciclo
```

O exemplo com desenrolamento manual serve apenas para ilustrar o que a ferramenta de síntese de alto nível faz quando desenrola uma estrutura de repetição. O resultado de ambas as soluções é o mesmo, sendo que a utilização da diretiva é vantajosa, pois não obriga à alteração do código.

```
void acumulaManual (int a[4], int *dout){
  int acc = 0, i;

  repete: for (i = 0; i < 4; i+=2)
            acc = acc + a[i];
            acc = acc + a[i+1];
  *dout = acc;
}
```

Código 5-2 - D*esenrolamento manual de* um `for` com um fator de 2 *e com pipeline*

```
void acumulaAuto (int a[4], int *dout){
  int acc = 0, i;

  repete: for (i = 0; i < 4; i++)
  #pragma HLS UNROLL factor=2
            acc = acc + a[i];
  *dout = acc;
}
```

Código 5-3 - Exemplo com desenrolamento de 2, utilizando a diretiva de desenrolamento e com pipeline

Uma análise atenta ao código do exemplo permite verificar que existe uma dependência de dados entre as duas operações realizadas no corpo do `for`. Ambas acumulam sobre a variável `acc` de forma sequencial. Apesar desta dependência, vamos assumir que é possível executar as duas operações num único ciclo de relógio, ou seja, os dois elementos do vetor são somados e depois acumulados num único ciclo de relógio. Neste caso, o escalonamento da função está ilustrado na Figura 5-4.

Com o desenrolamento de 2, consegue-se reduzir o intervalo de iniciação da função para 5 ciclos de relógio e um intervalo de iniciação do `for` de 1. A latência da função reduziu para apenas 4 ciclos de relógio.

Existem alguns aspetos nesta implementação que convém realçar. Um dos aspetos, já referido anteriormente, é o de estarmos a executar duas somas por ciclo de relógio. Esta dupla soma resulta da dependência de dados entre operações dentro do `for`. A consequência é o aumento do caminho crítico do circuito e, consequentemente, a redução da frequência de relógio. Um outro aspeto importante está relacionado com o acesso aos elementos do vetor. No exemplo, estamos a aceder a dois elementos do vetor por cada ciclo de relógio. Caso não seja possível, as leituras teriam de ser sequenciadas, o que faria aumentar o intervalo de iniciação do `for`. No caso concreto do *Vitis HLS*, que tem como

alvo uma FPGA, as memórias internas têm duplo porto, o que permite dois acessos em paralelo.

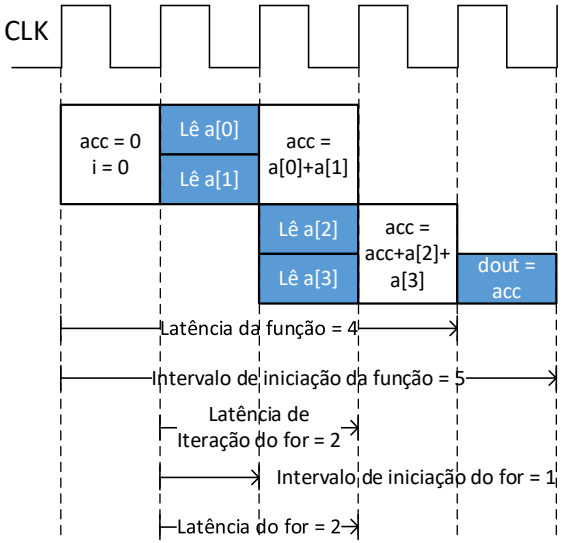


Figura 5-4 – Escalonamento do ciclo simples com desenrolamento de 2 e com pipeline

A implementação neste caso é similar ao circuito sem desenrolamento.

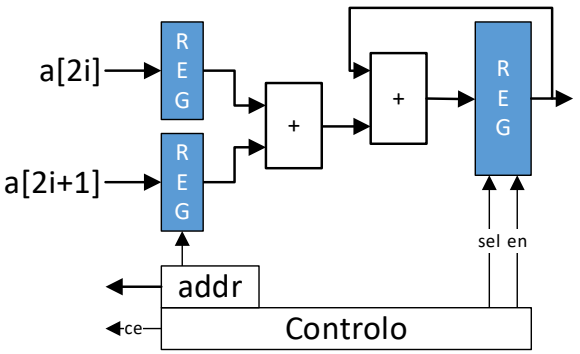


Figura 5-5 – Implementação hardware do ciclo `for` com desenrolamento de 2

Observa-se na Figura 5-5 a inclusão de um somador que adiciona dois elementos do vetor, cuja resultado é depois acumulado com outro somador no mesmo ciclo de relógio.

O desenrolar de uma estrutura de repetição incondicional pode ser levado até ao limite em que toda a estrutura é desenrolada, ou seja, todas as iterações são executadas em paralelo. No exemplo anterior do acumulador, o desenrolamento máximo é de 4, que corresponde

ao número de iterações da estrutura `for`. Nesta situação, o escalonamento ótimo lê todos os elementos do vetor num único ciclo de relógio e acumula todos os valores no ciclo de relógio seguinte. Para que isto seja possível, é necessária uma leitura de quatro elementos do vetor em paralelo.

Existem várias formas de reorganizar o vetor de modo a permitir o acesso a vários elementos em paralelo. Uma das formas consiste em concatenar vários elementos do vetor num único elemento com um tamanho em bits igual à soma do tamanho dos elementos concatenados. Por exemplo, um vetor com quatro elementos de 32 bits cada pode ser transformado num vetor de dois elementos com 64 bits cada. Isto permite que um acesso ao novo vetor corresponda a uma leitura de dois elementos em paralelo. Uma outra reorganização que permite a leitura de vários elementos em paralelo consiste em distribuir os elementos do vetor por várias memórias, aumentando assim o número de portos de acesso em proporção ao número de memórias. Qualquer uma destas formas de reorganização dos dados em memória pode ser conseguida com diretivas de síntese. O acesso aos dados, juntamente com as diretivas de reorganização dos dados em memória, são aspetos bastante importantes que serão abordados detalhadamente no capítulo 6.

Apenas para ilustrar a reorganização dos vetores em memória no contexto das estruturas de repetição, apliquemos a diretiva de concatenação (`HLS ARRAY_RESHAPE`) ao vetor de 4 elementos do exemplo (ver Código 5-4).

```
void acumula (int a[4], int *dout){
#pragma HLS ARRAY_RESHAPE variable=a factor=2 block

int acc = 0, i;

repete: for (i = 0; i < 4; i++)
#pragma HLS UNROLL factor=4
            acc = acc + a[i];

*dout = acc;
}
```

Código 5-4 - Exemplo com desenrolamento de 4, com concatenação dos elementos do vetor e com pipeline

A diretiva junta dois elementos do vetor num único com o dobro do tamanho. A estrutura `for` é completamente desenrolada, pois o fator de desenrolamento é igual ao número de iterações do `for` (ver escalonamento na Figura 5-6).

Este exemplo é sintetizado de forma diferente dos anteriores. Sendo o desenrolamento completo, a estrutura `for` desaparece e o circuito passa a ser apenas a leitura dos quatro elementos em paralelo seguido da sua soma. Uma vez que não necessita de acumulador, então não precisa do registo. Como tal, o retorno do valor de `acc` é feito no mesmo ciclo de relógio em que faz a soma. Passaríamos assim a uma latência do `for` de apenas 1 ciclo de relógio.

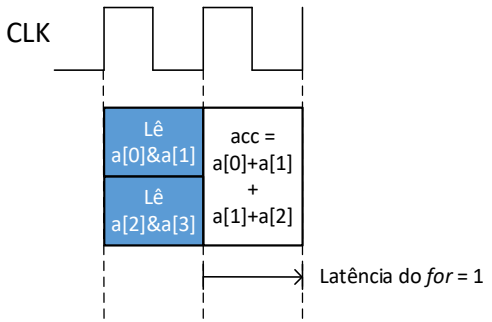


Figura 5-6 – Escalonamento do ciclo simples com desenrolamento de 4 e com pipeline

Em termos de implementação, necessitaríamos de somadores extra para adicionar os quatro elementos e um acesso em paralelo aos quatro elementos do vetor (ver Figura 5-7).

Neste caso, não é necessário acumulador, bastando uma árvore de somadores com dois níveis.

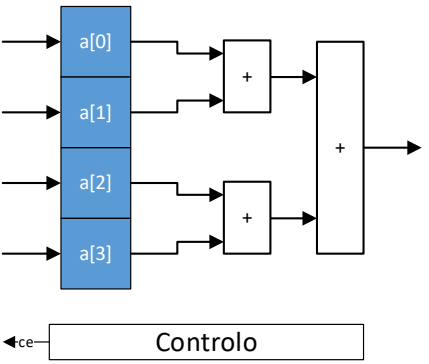


Figura 5-7 – Implementação hardware do ciclo simples com desenrolamento completo

Em todas as implementações que vimos até este ponto considerou-se não existirem restrições temporais ao caminho crítico. Por exemplo, no exercício anterior, a árvore de somadores determina o caminho crítico. No entanto, caso se pretendesse cumprir um determinado período de relógio, podia dar-se o caso de o caminho crítico ter um tempo superior ao período de relógio. Nesta situação, a ferramenta de síntese procura cumprir os requisitos temporais com a adição de registos intermédios que cortem o caminho crítico. Neste caso, podem surgir várias soluções. No circuito da Figura 5-7, poderiam colocar-se registos entre o primeiro nível de somadores e o segundo ou entre os dados de entrada e o primeiro somador. A solução gerada pelo *Vitis HLS* depende do valor da restrição temporal. Na Figura 5-8 apresenta-se o escalonamento gerado pela ferramenta de síntese com um período de relógio de 10 ns.

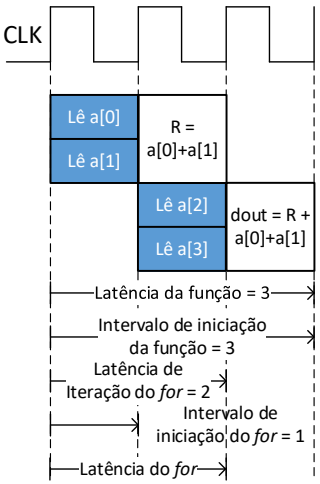


Figura 5-8 – Escalonamento da estrutura `for` com desenrolamento de 4, com pipeline e com uma restrição temporal de 10 ns

A ferramenta de síntese reduz a leitura a apenas dois elementos concatenados de cada vez e utiliza apenas um somador no primeiro nível. No primeiro ciclo de relógio lê o primeiro par de valores. No segundo ciclo lê o segundo par de valores e guarda a soma do par anterior em registo (R). No terceiro ciclo soma o segundo par com o registo que contém a soma do primeiro par de elementos. Uma vez que não existe acumulador, o valor final `dout` fica disponível no terceiro ciclo de relógio e pode ser enviado nesse mesmo ciclo.

A solução hardware da função com desenrolamento de quatro e que implementa o escalonamento da Figura 5-8 está ilustrada na Figura 5-9.

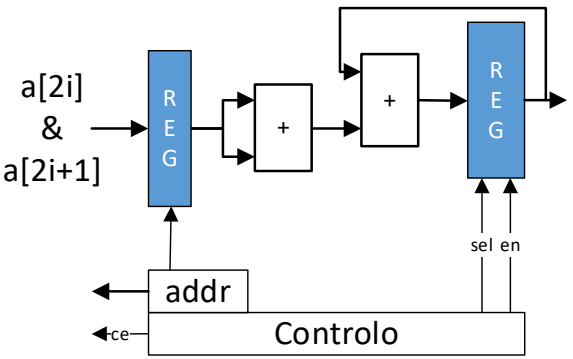


Figura 5-9 – Implementação hardware do ciclo simples com desenrolamento completo, pipeline e restrição temporal de 10 ns

Como se pode observar na Figura 5-9, os dois elementos lidos em paralelo são guardados num registo intermédio. Este registo reduz o caminho crítico do circuito.

O desenrolamento de estruturas de repetição em classes C++ está condicionado à variável de controlo da estrutura não ser um membro da classe. Se for este o caso, a ferramenta de síntese não consegue desenrolar a estrutura. Se pretendermos desenrolar, é necessário reescrever o código de modo a remover a variável de controlo da estrutura de repetição da lista de membros da classe C++.

5.1.3 Estruturas de Repetição Condicionais

Nas estruturas de repetição incondicionais o número de iterações é conhecido *a priori*, ou seja, o número de iterações é constante. A variável de controlo da estrutura de repetição e a condição de teste são iniciadas com constantes e o incremento também é feito com uma constante. Esta forma de especificar a estrutura de repetição é a ideal, pois permite determinar e otimizar o número de bits necessários para a variável de contagem do número de iterações, aplicar desenrolamento e determinar o número total de ciclos de relógio necessários à execução da estrutura de repetição.

Contudo, nem sempre é possível ter todos os parâmetros de controlo da estrutura de repetição com valores constantes. Por exemplo, o número de iterações pode depender de um valor de entrada. Consideremos o exemplo do acumulador com um número de iterações variável (ver Código 5-5).

```
void acumulaVar (int a[4], int limite, int *dout){

  int acc = 0, i;

  repete: for (i = 0; i < limite; i++)
             acc = acc + a[i];

  *dout = acc;
}
```

Código 5-5 - Exemplo da função de acumulação com um número de iterações variável

No exemplo, o número de iterações foi definido com a variável `limite`, que é um inteiro de 32 bits.

Um problema associado à utilização de limites variáveis é que a ferramenta de síntese não consegue determinar a latência da estrutura de repetição, uma vez que não sabe quantas iterações irá executar. Assim, o projetista não consegue determinar o desempenho do circuito.

No entanto, é possível indicar os limites máximo, mínimo ou médio do número de iterações com a diretiva `tripcount` para que a ferramenta determine a latência considerando este número de iterações. A diretiva não tem qualquer influência sobre a síntese do circuito,

apenas sobre o relatório das latências do circuito. Para a ferramenta de síntese, o limite do ciclo continua a ser variável.

```
#pragma HLS LOOP_TRIPCOUNT min=<int> max=<int> avg=<int>

min=<int> - especifica o número mínimo de iterações

max=<int> - especifica o número máximo de iterações

avg=<int> - especifica o número médio de iterações
```

A diretiva permite especificar um número mínimo, máximo ou médio de iterações relativo ao ciclo a que se aplica a diretiva. A diretiva pode combinar mais do que uma condição. Por exemplo, é possível especificar um mínimo e um máximo de iterações.

Se quisermos, por exemplo, indicar um número máximo de iterações para cálculo da latência no circuito de acumulação, teríamos a descrição do Código 5-6.

```
void acumulaVar (int a[4], int limite, int *dout){

int acc = 0, i;

repete: for (i = 0; i < limite; i++)
#pragma HLS LOOP_TRIPCOUNT max=4
            acc = acc + a[i];

*dout = acc;
}
```

Código 5-6 - Exemplo de aplicação da diretiva tripcount à função de acumulação

No exemplo descrito, definiu-se 4 como limite máximo. Desta forma, a ferramenta irá dar os valores de latência assumindo o valor máximo constante e igual a 4.

Em termos de síntese, a principal alteração que se verifica no hardware gerado quando se utilizam estruturas de repetição condicionais é sobre o contador de iterações e o comparador com o valor limite, que, no exemplo, passam a ter 32 bits. Além disso, é ainda necessário ler a entrada correspondente à variável *limite*.

Um outro problema das estruturas de repetição condicionais é que não podem ser desenroladas. No caso de estruturas `for` entrelaçadas não é possível aplicar *pipeline* às estruturas `for` que são exteriores à estrutura condicional (como veremos mais à frente, quando temos vários `for` entrelaçados só é possível aplicar *pipeline* a um `for` exterior desenrolando os `for` internos a este). O problema é que, sem saber o número de iterações, a ferramenta de síntese não consegue determinar o hardware necessário para garantir o desenrolamento pedido.

Existe uma forma de resolver esta limitação com a inclusão explícita de uma condição dentro da estrutura de repetição que verifica se a variável de controlo da iteração já atingiu o limite variável, que, no exemplo, pode ir até o máximo de 4 (ver Código 5-7).

```
void acumula (int a[4], int limite, int *dout){

int acc = 0, i;

repete: for (i = 0; i < 4; i++)
#pragma HLS UNROLL factor=2
       if (i < limite)
           acc = acc + a[i];

*dout = acc;
}
```

Código 5-7 - Exemplo de alteração do controlo do `for` condicional da função de acumulação de modo a permitir desenrolar a estrutura `for`

No exemplo, a acumulação só é feita se a variável de controlo do número de iterações, `i`, for inferior à condição limite dada pela variável de entrada `limite`. A estrutura de repetição `for` passou a ser incondicional, pois o limite da variável `i` é constante. Neste caso, já é possível desenrolar o `for`. Ao desenrolar com um fator de 2, obtém-se o escalonamento da Figura 5-10. A latência de iteração do `for` é igual a 3, pois é necessário fazer a comparação da variável de controlo do número de iterações com a condição limite. A latência da função passa a 5 ciclos de relógio.

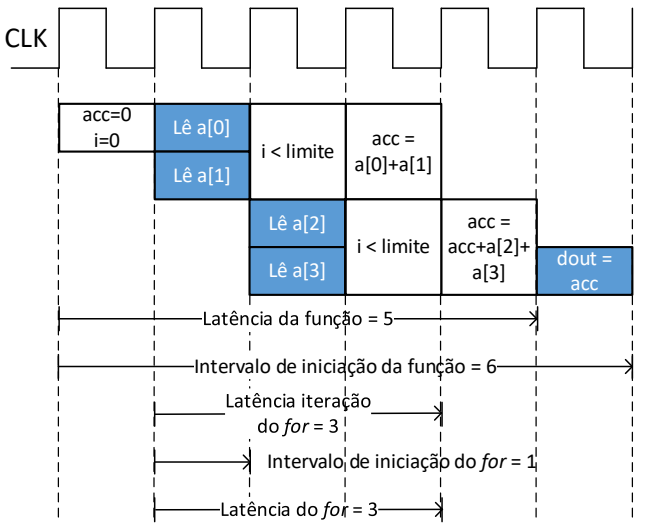


Figura 5-10 – Escalonamento da estrutura `for` condicional modificada com fator 2 de desenrolamento e com pipeline

O circuito de controlo pode ser mais otimizado ajustando o número de bits de representação da variável `limite`. No exemplo, é declarada como sendo do tipo inteiro, que é representado com 32 bits. Conhecendo o valor máximo que a variável pode tomar, sabemos quantos bits são necessários para a representar. Por exemplo, se o valor máximo fosse 1000, então apenas são necessários 10 bits para representar a variável. Os tipos de dados nativos do C/C++ não permitem ao projetista declarar variáveis do tipo inteiro com tamanho arbitrário, restringindo-se a tamanhos de 8, 16, 32 e 64 bits. Em todo o caso, para representar, por exemplo, variáveis com um valor máximo de 1000 bastariam 16 bits. Este pequeno detalhe já permitiria reduzir a área do circuito de controlo.

Sabendo que o projeto de hardware permite desenvolver circuitos em que os dados podem ter um qualquer número de bits, a HLS contém um conjunto de tipos de dados com precisão arbitrária e bibliotecas C/C++ que dão suporte a operações com tipos de dados de precisão arbitrária. Com este suporte, a variável de controlo seria declarada com o número de bits necessário e suficiente para o problema em causa. Os tipos de dados de precisão arbitrária serão abordados no Capítulo 7.

Para evitar que o projetista tenha de determinar manualmente o tamanho da variável de controlo de iterações, a ferramenta de síntese de alto nível, por omissão, otimiza o tamanho desta variável, mesmo que seja declarada com um tipo de dados nativo do C/C++.

5.1.4 Estruturas de Repetição Entrelaçadas

Como referido anteriormente, uma estrutura de repetição entrelaçada corresponde a ter uma estrutura de repetição dentro de outra. Consideremos o exemplo da acumulação dos elementos de um vetor, mas desta vez com uma matriz. Neste exemplo, pretende-se calcular a soma dos elementos de cada linha da matriz (ver Código 5-8).

```
void acumulaMatriz (int a[4][4], int *dout){
int acc, i, j;

repete_i: for (i = 0; i < 4; i++){
#pragma HLS PIPELINE off
            acc = 0;
  repete_j: for (j = 0; j < 4; j++){
               acc = acc + a[i][j];
            }

            *dout = acc;
}}
```

Código 5-8 - Exemplo de uma função de acumulação com estruturas de repetição entrelaçadas, sem pipeline e sem desenrolamento

A sequência de estruturas `for` permite percorrer todos os elementos da matriz. Não se pretende aplicar *pipeline*, pelo que se utilizou a diretiva `HLS pipeline off` na estrutura `for` exterior. No `for` interior não é preciso aplicar porque a ferramenta só faz *pipeline* a um `for` interior se os `for` que lhe são externos também tiverem *pipeline*.

O escalonamento da implementação das estruturas `for` entrelaçadas do exemplo pode ser visto na Figura 5-11.

A implementação de cada uma das estruturas `for` do exemplo é caracterizada pela latência do `for`, pela latência de iteração e pela latência de iniciação. Começando pelo `for` interior (`repete_j`), as latências são iguais às que se obteriam caso houvesse apenas este `for`. Assim, temos a latência de iteração do `for` igual a 2 (leitura seguida de acumulação) e a latência do `for` igual a 8 (4 iterações vezes a latência de iteração). No caso do ciclo exterior (`repete_j`), a latência de iteração do `for` é igual a 10 (incremento da variável `j`, seguido dos oito ciclos de relógio de execução do ciclo interior, mais um ciclo de relógio para escrita do resultado final). Logo, a latência de iteração do `for` exterior é igual a 40 (4 iterações vezes a latência de iteração do `for` interior. O intervalo de iniciação da função é igual a 40 ciclos de relógio, mais um ciclo para iniciação e um ciclo final.

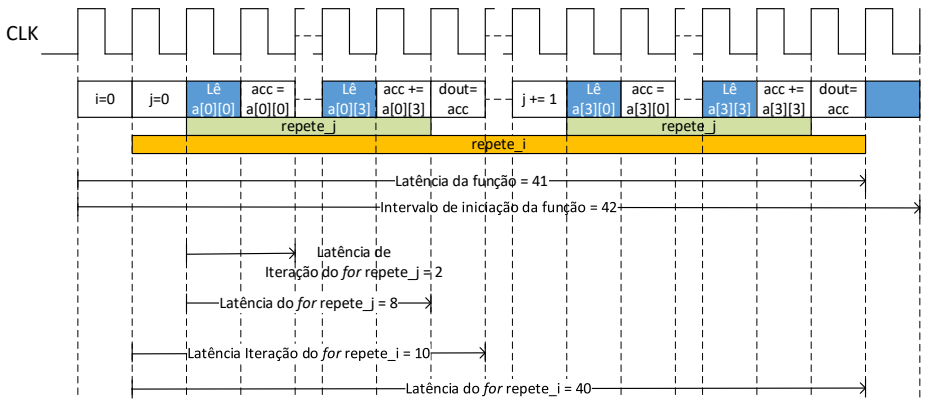


Figura 5-11 – Escalonamento das estruturas `for` entrelaçadas sem pipeline e sem desenrolamento

O circuito pode ser melhorado aplicando *pipeline*. No caso de se aplicar *pipeline* a ambos as estruturas `for`, verifica-se que, neste exemplo, as estruturas são unificadas numa única, ou seja, ficaria uma única estrutura `for` com 16 (4×4) iterações (ver Figura 5-12).

O escalonamento do circuito passa a ter 16 ciclos de relógio de latência correspondentes ao único `for`. No total, teríamos um tempo de execução de 19 ciclos de relógio. Contudo, podem existir dependências de dados ou de controlo que impeçam a unificação das estruturas de repetição.

As vantagens em unir duas ou mais estruturas de repetição numa só são a simplificação do circuito de controlo e a redução do ciclo de relógio gasto na transição entre estruturas.

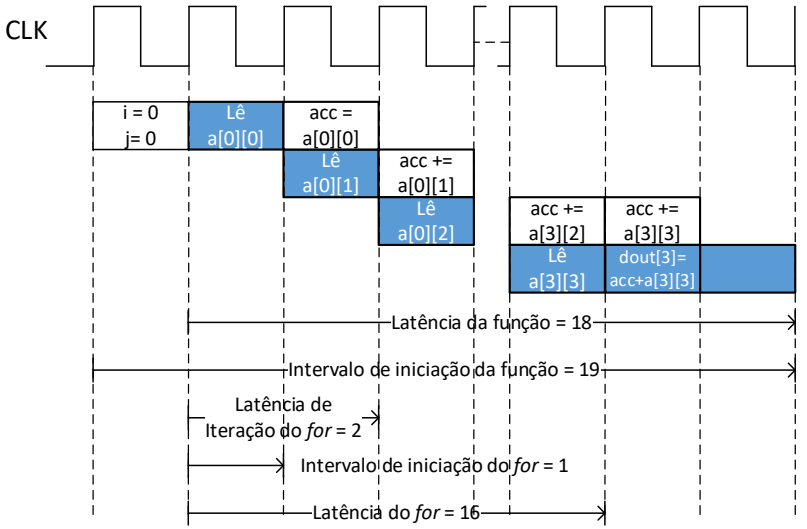


Figura 5-12 – Escalonamento das estruturas `for` entrelaçadas com pipeline e sem desenrolamento

A união das estruturas pode ser feita manualmente, com a reorganização do código, ou utilizando a diretiva `HLS loop_flatten`.

`#pragma HLS LOOP_FLATTEN off`

`off` – opção que previne a união das estruturas de repetição

Consideremos o exemplo das estruturas de repetição entrelaçadas sem *pipeline* e com união das estruturas (ver Código 5-9).

```
void acumulaMatriz (int a[4][4], int *dout){
#pragma HLS PIPELINE off

int acc, i, j;

repete_i: for (i = 0; i < 4; i++){
       acc = 0;
       repete_j: for (j = 0; j < 4; j++){
       #pragma HLS LOOP_FLATTEN
              acc = acc + a[i][j];
       }
       *dout = acc;
}}
```

Código 5-9 - Exemplo da função de acumulação com estruturas de repetição entrelaçadas, sem pipeline e com diretiva de união

A função é sintetizada com uma única estrutura `for` de 16 iterações. No total iremos ter 32 ciclos de relógio (16 iterações vezes dois ciclos de relógio por iteração), mais um ciclo de inicialização de `acc` e um ciclo de escrita em `dout`.

É possível unir estruturas de repetição entrelaçadas com a diretiva `HLS LOOP_FLATTEN` desde que as estruturas sejam perfeitas ou semi-perfeitas, de acordo com o seguinte:

- Estruturas de repetição entrelaçadas perfeitas:
  - Apenas a estrutura interior contém código dependente das variáveis de controlo;
  - Não existe lógica entre estruturas de repetição
  - Todos os limites de controlo das iterações são constantes;
- Estruturas de repetição entrelaçadas semi-perfeitas:
  - Apenas a estrutura interior contém código dependente das variáveis de controlo;
  - Não existe lógica entre estruturas de repetição
  - O limite de iteração da estrutura de repetição exterior pode ser uma variável.

Em todos os outros casos, se possível, deverá reorganizar-se o código de modo a cumprir uma das condições anteriores e assim poder aplicar a diretiva de união de estruturas.

As estruturas de repetição entrelaçadas também podem ser desenroladas com o objetivo de melhorar o desempenho à custa de mais recursos hardware. O desenrolamento máximo é determinado pelo produto do máximo do número de iterações de cada estrutura de repetição. No exemplo anterior, seria de 16. Este fator máximo de desenrolamento pode facilmente atingir valores elevados que não permitem o desenrolamento total das estruturas. Como tal, é necessário considerar um desenrolamento parcial. A regra é começar por aplicar o desenrolamento às estruturas interiores, aliado à técnica de *pipeline*.

Consideremos um desenrolamento de 4 do `for` interno com pipeline (ver Código 5-10).

```
void acumulaDesenrola (int a[4][4], volatile int *dout){
#pragma HLS ARRAY_PARTITION dim=2 factor=4 type=block variable=a

int acc, i, j;

repete_i: for (i = 0; i < 4; i++){
	acc = 0;
	repete_j: for (j = 0; j < 4; j++){
		#pragma HLS UNROLL factor=4
		#pragma HLS PIPELINE
		acc = acc + a[i][j];
	}
	*dout = acc;
}}
```

Código 5-10 - Exemplo de uma função de acumulação com ciclos entrelaçados com desenrolamento completo do `for` interior

No exemplo, os quatro elementos do vetor para cada valor diferente de `i` são lidos em paralelo. Para garantir um número de portos de acesso à memória suficiente para ler quatro elementos em paralelo, utilizou-se a diretiva de partição do vetor. As estruturas de repetição estão otimizadas com *pipeline*, o que permite que as execuções do `for` interno sejam escalonadas em *pipeline*. A diretiva de desenrolamento do `for` interior pode ser omitida mesmo quando se pretende desenrolar completamente. Isto porque para aplicar *pipeline* à estrutura de repetição exterior, o `for` interior é automaticamente desenrolado por completo. É preciso algum cuidado com este processo, pois o hardware aumenta proporcionalmente ao fator de desenrolamento. O escalonamento da solução encontra-se ilustrado na Figura 5-13.

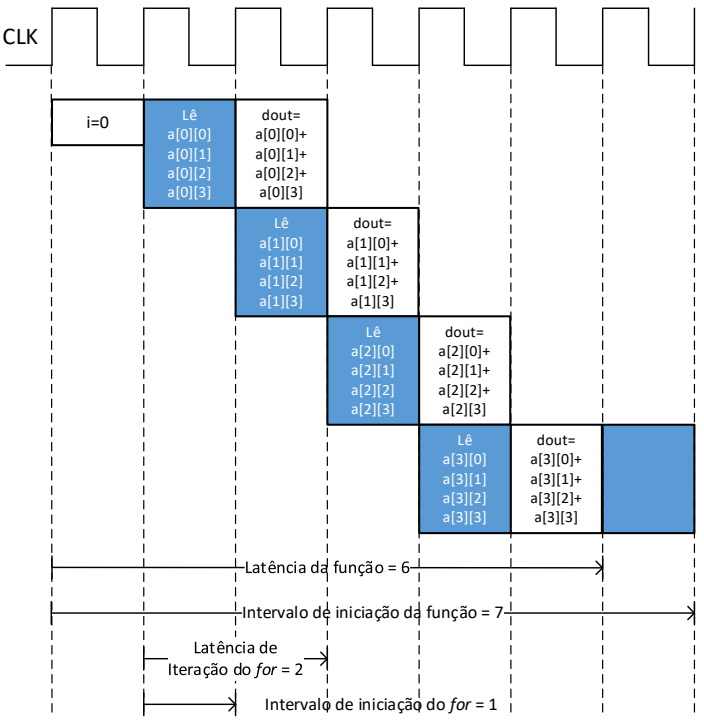


Figura 5-13 – Escalonamento das estruturas de repetição entrelaçadas com pipeline e desenrolamento completo do `for` interior

Como se pode observar no escalonamento da solução, a *pipeline* do `for` exterior tem uma latência de iteração de 2 ciclos de relógio: um ciclo de leitura de quatro elementos do vetor mais um ciclo de cálculo da soma dos elementos lidos no ciclo anterior. Uma vez que não é necessário acumulador, o valor da soma calculado no segundo ciclo de relógio da *pipeline* é disponibilizado de imediato na saída `dout`.

O processo pode ser levado ao extremo com o desenrolamento completo de ambos os ciclos (ver Código 5-11).

```
void acumulaDesenrola (int a[4][4], int dout[4]){
#pragma HLS ARRAY_PARTITION dim=1 factor=4 type=block variable=a
#pragma HLS ARRAY_PARTITION dim=2 factor=4 type=block variable=a
#pragma HLS ARRAY_PARTITION factor=2 type=block variable=dout

int acc, i, j;

repete_i: for (i = 0; i < 4; i++){
     #pragma HLS UNROLL factor=4

       acc = 0;
       repete_j: for (j = 0; j < 4; j++){
              #pragma HLS UNROLL factor=4

              acc = acc + a[i][j];
      }
       dout[i] = acc;
}}
```

Código 5-11 - Exemplo da função de acumulação com `for` entrelaçados, com pipeline e desenrolamento completo de ambos os `for`

O código inclui uma diretiva de desenrolamento em cada estrutura `for` e diretivas de partição dos vetores de modo a permitir múltiplos acessos em paralelo, quer ao vetor de entrada, quer ao vetor de saída.

A função demora apenas um ciclo de relógio a executar, caso não existam restrições temporais. No caso de existirem restrições temporais, a execução do circuito é dividida em vários ciclos de relógio de modo a cumprir os requisitos temporais.

5.1.5 Estruturas de Repetição Sequenciais

As múltiplas estruturas de repetição analisadas na secção anterior estavam entrelaçadas. Contudo, a função a sintetizar pode ter múltiplas estruturas de repetição em sequência. A ferramenta de síntese consegue juntar várias estruturas de repetição sequenciais de modo a executá-las em paralelo. Se a ferramenta não conseguir, o projetista pode ter de reescrever o código procurando juntar as estruturas de repetição manualmente.

Consideremos um exemplo simples que utiliza dois `for` independentes de acumulação de vetores também independentes (ver Código 5-12).

```
void acumula2FOR (int a[4], int b[4], int *dout0, int *dout1){

int acc0, acc1, i;

acc0 = 0;
acc1 = 0;
repete_0: for (i = 0; i < 4; i++){
       acc0 = acc0 + a[i];
```

```
}

repete_1: for (i = 0; i < 4; i++){
	acc1 = acc1 + b[i];
}

*dout0 = acc0;
*dout1 = acc1;
}
```

Código 5-12 - Exemplo de especificação de duas estruturas `for` sequenciais independentes

Como descrito no código do exemplo, os `for` operam sobre vetores e variáveis diferentes, pelo que não existe qualquer dependência de dados. Assim, as estruturas `for` podem ser executadas em paralelo. Além disso, trata-se de estruturas `for` simples com um processo de controlo de iterações idêntico, pelo que se pode utilizar o mesmo controlo de iteração para ambos. Nestas condições, podem fundir-se os dois `for`. Isto pode ser feito manualmente (ver Código 5-13).

```
void acumula2FOR (int a[4], int b[4], int *dout0, int *dout1){

int acc0, acc1, i;

acc0 = 0;
acc1 = 0;

repete_0: for (i = 0; i < 4; i++){
	acc0 = acc0 + a[i];
	acc1 = acc1 + b[i];
}

*dout0 = acc0;
*dout1 = acc1;
}
```

Código 5-13 - Junção manual de dois ciclos independentes

As duas estruturas `for` passam a ser descritas com apenas um `for` em que os corpos das estruturas originais passam a pertencer a um único `for`. O mesmo processo pode, naturalmente, ser aplicado a mais do que duas estruturas de repetição sequenciais.

A junção de dois ou mais ciclos pode ser conseguida automaticamente, sem ter de alterar o código manualmente, com a utilização da diretiva `HLS LOOP_MERGE`.

`#pragma HLS LOOP_MERGE force`

`force` – opção que força a junção das estruturas de repetição mesmo que a ferramenta de síntese gere um aviso

A ferramenta de síntese tenta juntar todos as estruturas de repetição que se encontram descritas no código ao mesmo nível em que é descrita a diretiva.

A utilização da diretiva no Código 5-12 é feita de acordo com o descrito no Código 5-14.

```
void acumula2FOR (int a[4], int b[4], int *dout0, int *dout1){
#pragma HLS LOOP_MERGE

int acc0, acc1, i;

(...) /* instruções iguais ao do Código 25 */
}
```

Código 5-14 - Aplicação da diretiva de junção de estruturas de repetição ao exemplo do Código 5-12

A junção pode ser aplicada dentro de uma estrutura de repetição cujo corpo contém várias estruturas de repetição sequenciais. Neste caso, a junção aplica-se apenas às estruturas interiores. Em ambos os casos, manual ou automático, o resultado é o mesmo. O escalonamento do componente sintetizado após a fusão das estruturas e com *pipeline* é ilustrado na Figura 5-14.

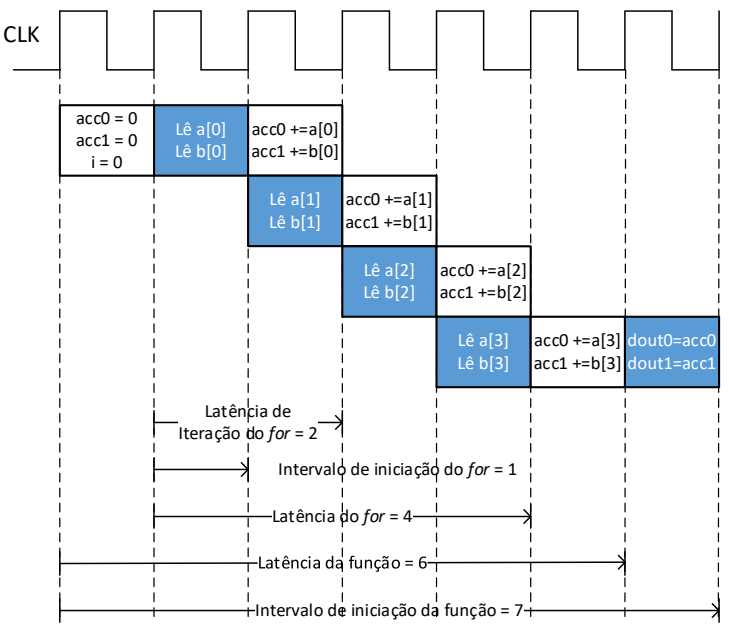


Figura 5-14 – Escalonamento da junção de duas estruturas `for` com pipeline

O escalonamento mostra que as operações de ambos os `for` executam ao mesmo tempo. No total, o componente necessita de 7 ciclos de relógio para executar e poder iniciar uma

nova execução. A implementação do componente hardware é feita com dois acumuladores (ver Figura 5-15).

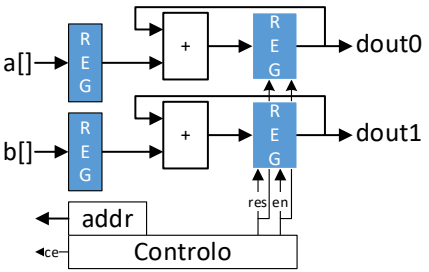


Figura 5-15 – Resultado de síntese da função após junção das estruturas `for`

O exemplo consiste em duas acumulações de dois vetores independentes, pelo que, ao juntar as estruturas `for`, são necessários dois acumuladores. A leitura dos elementos dos vetores `a` e `b` também são feitas em paralelo.

A junção de estruturas de repetição está sujeita às seguintes restrições:

- Se os limites das variáveis de controlo das iterações forem constantes, o valor máximo é utilizado como limite da estrutura de repetição única;
- Se os limites das variáveis de controlo das iterações não forem constantes, devem ter a mesma condição;
- A junção de estruturas de repetição com limites das variáveis de controlo constantes com estruturas de repetição com limites das variáveis de controlo não constantes não é possível;
- O código entre as estruturas de repetição sequenciais não pode ter efeitos colaterais (por exemplo, x = x + 1 não é permitido);
- As estruturas de repetição não podem ser juntas se contiverem leituras a memórias sequenciais, ou seja, tipo FIFO. A junção das estruturas pode alterar a leitura sequencial dos dados.

A junção das estruturas de repetição melhora os tempos de execução, pois executa ambas em paralelo. No entanto, aumenta os recursos necessários à sua implementação. Como se pode ver pela implementação do circuito, houve uma duplicação do hardware necessário ao armazenamento e processamento dos dados. Por outro lado, houve uma redução do hardware de controlo, uma vez que apenas é necessário controlar uma estrutura de repetição.

Vejamos o exemplo anterior sem junção de estruturas de repetição, mas com *pipeline*. O componente executa as operações das estruturas individualmente com *pipeline*, mas em sequência (ver escalonamento na Figura 5-16).

Pelo escalonamento, pode observar-se que as operações das estruturas `for` são executadas com *pipeline*, mas em sequência. Note-se que ocorre uma sobreposição da operação quando termina o primeiro `for` (`dout0 = acc0`) e inicia o segundo (`acc1 = 0; i1 = 0`). Estas operações são executadas fora do corpo dos `for` e não têm dependências entre si, pelo que a ferramenta de síntese as executa em paralelo. No total, temos 6 ciclos de relógio para executar as operações associadas a cada um dos `for` menos um ciclo de relógio devido à sobreposição das operações referidas atrás, um ciclo de enchimento da *pipeline* associado ao primeiro `for`, mais um ciclo final de escrita do resultado, num total de 13 ciclos de relógio.

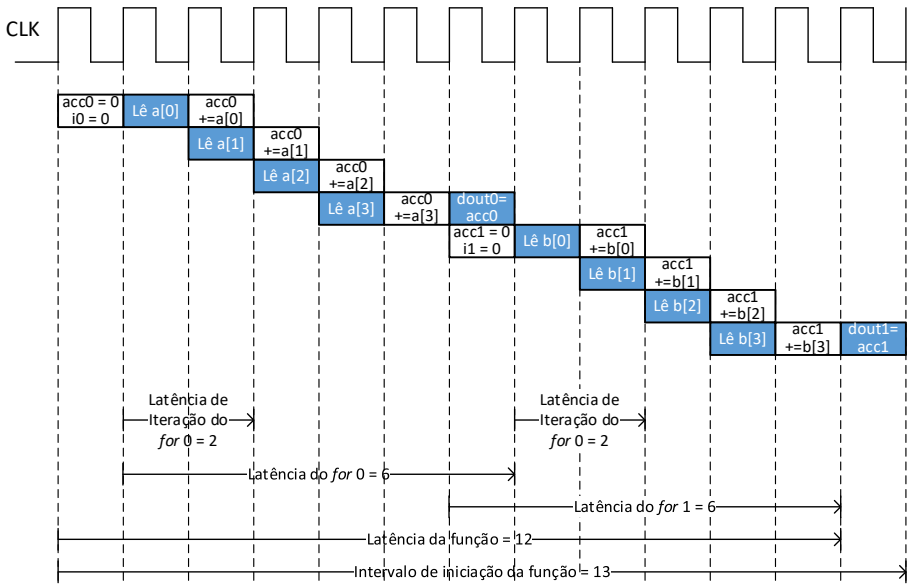


Figura 5-16 – Escalonamento das estruturas `for` com pipeline, mas sem serem juntas

Comparando as soluções com e sem junção de estruturas de repetição `for`, temos o compromisso esperado entre recursos utilizados e latência (ver Tabela 5-1).

	Sem junção	Com junção
Recursos	1 registos + 1 somador	2 registos + 2 somadores
Latência	13	7

Tabela 5-1 – Comparação entre as soluções com e sem junção de ciclos de repetição `for`. do exemplo de acumulação

Existe uma duplicação de recursos quando se juntam as estruturas, mas a latência reduz para aproximadamente metade.

5.2 Síntese de Estruturas de Condição

As estruturas de condição em C/C++ determinam a execução condicional de operações de forma exclusiva, ou seja, se executa as operações associadas a uma condição então não executa as operações associadas às restantes condições. Em termos de implementação hardware, a ferramenta de síntese procura partilhar operadores que sejam comuns a vários ramos da condição. Isto realça a importância em seguir boas práticas de codificação de modo a permitir à ferramenta identificar estas partilhas.

Numa função C/C++ podemos encontrar duas estruturas distintas de descrição de condições: `if-else` e `switch-case`.

A condição `if-else` tem os campos de teste da condição e o código associado a cada umas das condições:

```
ETIQUETA: if (condição A) {
                Corpo da condição A;
          }
          else if (condição B) {
                Corpo da condição B;
          }
          (...)
          else {
                Corpo da condição final;
          }
```

São permitidas múltiplas condições diferentes bastando encadear condições `if-else` sucessivas. A condição final não é indicada pois corresponde ao caso em que nenhuma das condições anteriores se verificou.

A condição `switch-case` tem um campo que determina o número da condição, seguido do código associado a cada umas das condições:

```
ETIQUETA: switch (expressão da condição) {
                case 0: Corpo da condição 0; break;
                case 1: Corpo da condição 1; break;
                case 2: Corpo da condição 2; break;
                case 3: Corpo da condição 3; break;
                default: Corpo da condição por omissão; break;
                (...)
```

Após o corpo de cada condição do `case` é necessário adicionar um `break` no caso de não se pretender continuar a testar as restantes condições.

Consideremos um exemplo de acumulação condicional (ver Código 5-15). O exemplo considera a acumulação dos elementos do vetor `a` ou do vetor `b`, dependendo do valor da variável `sel`. O exemplo descreve o corpo das condições com uma estrutura de repetição sobre o vetor `a` ou sobre o vetor `b`, caso a variável `sel` seja 0 ou 1, respetivamente.

```
void acumulaCond (int a[4], int b[4], bool sel, int *dout){

int acc, i;

acc = 0;
if (sel == 0)
    for (i = 0; i < 4; i++)
        acc += a[i];
else
    for (i = 0; i < 4; i++)
        acc += b[i];

*dout = acc;
}
```

Código 5-15 - Exemplo de uma função de acumulação condicional (versão 1)

Ambas as condições executam as mesmas operações, mas sobre vetores diferentes pelo que a ferramenta de síntese consegue partilhar os operadores. O mesmo resultado é obtido se considerarmos uma descrição com apenas uma estrutura de repetição partilhada.

```
void acumulaCond (int a[4], int b[4], bool sel, int *dout){
#pragma HLS PIPELINE off

int acc, i;

acc = 0;

for (i = 0; i < 4; i++)
    if (sel == 0)
        acc += a[i];
    else
        acc += b[i];

*dout = acc;
}
```

Código 5-16 - Exemplo de uma função de acumulação condicional (versão 2)

Na versão 2 do mesmo exemplo, descrita no Código 5-16, existe apenas uma estrutura de repetição `for`. O teste da condição é feito dentro do corpo do `for`, em que se escolhe o vetor cujo elemento será acumulado.

A implementação hardware apenas contém uma estrutura de acumulação, pois os operadores, de ambas as condições, são partilhados (ver implementação hardware na Figura 5-17).

A escolha do vetor é feita à entrada com um multiplexer cujo seletor é o sinal de entrada, `sel`. O circuito de acumulação é idêntico ao utilizado na acumulação dos elementos de um vetor, como introduzido em exemplos anteriores.

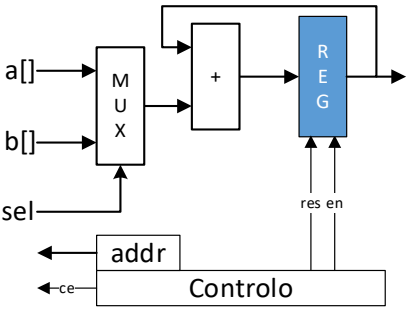


Figura 5-17 – Escalonamento dos ciclos sequenciais com pipeline, mas sem serem unidos

O escalonamento da versão 2 do circuito com *pipeline* é representado na Figura 5-18.

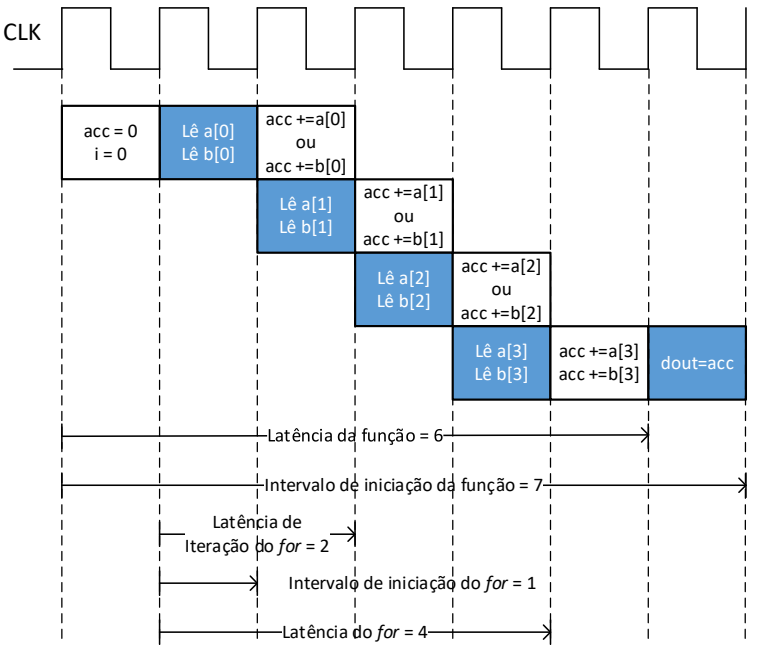


Figura 5-18 – Escalonamento da versão 2 do exemplo (Código 5-16) com condições considerando pipeline.

No primeiro ciclo de relógio do *pipeline* é lido em paralelo um elemento de cada vetor para o mesmo índice e no segundo ciclo acumula o valor do elemento do vetor `a` ou do vetor `b`, dependendo do valor da entrada `sel`.

6 Síntese de Interface

Uma função descrita em C/C++ recebe e envia dados através dos seus argumentos. Num circuito hardware, o envio e a receção de dados são feitos através de portos com base em protocolos de entrada e de saída. No processo de síntese de uma função de topo, os parâmetros da função são sintetizados em portos e é-lhes associado um protocolo. O *Vitis-HLS* designa este passo *Síntese de Interface*.

O modo como são descritos os parâmetros da função tem impacto nos resultados de síntese. Vimos no capítulo anterior que o acesso aos dados determina o resultado da síntese das estruturas de repetição. Em muitos casos, o acesso aos dados através de um porto único impede que sejam feitas leituras em paralelo e, consequentemente, pode tornar inconsequente o desenrolamento de uma estrutura de repetição ou reduzir a eficiência do método de *pipeline*.

Neste capítulo, descreve-se o funcionamento da síntese de interface, como são implementados os diferentes tipos de argumentos de uma função e como se pode otimizar o acesso aos dados.

6.1 O Funcionamento da Síntese de Interface

Para se perceber de forma genérica como funciona a síntese de interface, consideremos um exemplo simples de uma função com dois argumentos (ver Código 6-1).

```
int somaA (int a, int b[10]){

int res, i;

  for (i = 0; i < 10; i++)
       res = a + b[i];

return res;
}
```

Código 6-1 – Função utilizada como exemplo ilustrativo de aplicação da síntese de interface

A função tem um parâmetro com passagem por valor, `a`, um parâmetro com passagem por referência, `b`, neste caso, um vetor, e o retorno do valor da variável `res`.

Assumindo as configurações por omissão da síntese de interface, a ferramenta de síntese gera um bloco hardware com um conjunto de portos que implementam os argumentos e o retorno da função (ver Figura 6-1).

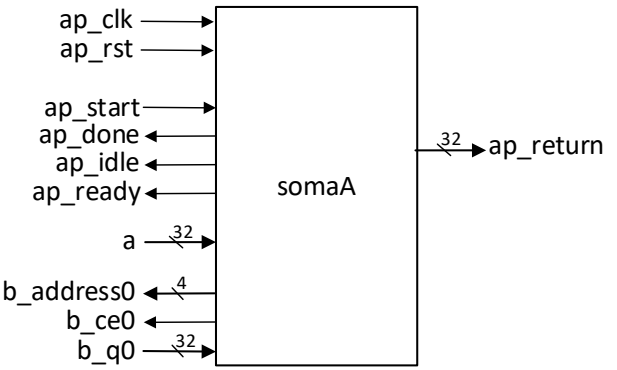


Figura 6-1 – Resultado da síntese de interface para a função do exemplo dado pelo Código 6-1

A interface inclui portos de sinal de relógio (`ap_clk`) e de inicialização (`ap_reset`). O sinal de relógio é adicionado automaticamente sempre que o componente necessita de mais do que um ciclo de relógio para completar a execução.

Para cada um dos argumentos e para o retorno da função são criados portos com um protocolo associado designado *Protocolo ao Nível do Porto*. Como se pode constatar na Figura 6-1, foi criada uma entrada de dados para o argumento passado por valor, `a`, um conjunto de portos de acesso a memória associado ao parâmetro passado por referência, `b`, para leitura do vetor, e um porto para enviar o valor de retorno (`ap_return`).

Por omissão, a interface também inclui sinais (`ap_start`, `ap_done`, `ap_idle`, `ap_ready`) associados ao designado *Protocolo ao Nível do Bloco*. Estes sinais controlam o acesso ao bloco independentemente de quaisquer protocolos associados aos portos individuais. Os sinais do protocolo de bloco determinam quando é que o bloco inicia a execução (`ap_start`), quando está preparado para poder receber novas entradas (`ap_ready`), se o bloco está inativo (`ap_idle`) ou se terminou a execução (`ap_done`).

O tipo de interface gerado pela ferramenta de síntese depende do tipo de argumento especificado na função, do tipo de interface considerado pela ferramenta por omissão e das diretivas HLS associadas às interfaces.

Nas secções seguintes, descrevem-se detalhadamente os protocolos de interface ao nível do porto e ao nível do bloco.

6.2 Protocolos de Interface ao Nível do Porto

A síntese de interface associa a cada tipo de argumento da função uma determinada interface ao nível do porto de entre um conjunto de possibilidades, a serem descritas nas secções seguintes.

6.2.1 Interface sem Protocolo

Um porto pode não ter qualquer protocolo associado, contendo apenas as linhas de dados. Aos escalares (parâmetros passados por valor) e aos argumentos passados por ponteiro ou por referência associados a um porto de entrada não é, por omissão, associado qualquer protocolo (`ap_none`). Isto significa que, para além do porto de dados, não têm mais nenhum sinal. O protocolo de interface de um porto pode ser forçado a não ter protocolo com a diretiva de interface:

```
#pragma HLS INTERFACE ap_ctrl_none port=<nome do porto>
```

Declarar um porto sem protocolo de interface tem a vantagem de simplificar a sua implementação. No entanto, quem envia os dados tem de garantir que os envia no tempo certo e que os mantém estáveis até serem lidos. Quem lê o resultado da função tem de garantir que o lê no ciclo de relógio certo.

6.2.2 Protocolo com Interface Válido/Recebido

Existem vários protocolos nativos que podem ser associados a um porto: `ap_hs`, `ap_valid`, `ap_ack`, `ap_ovld`. O mais completo de entre estes é o `ap_hs`, que inclui sinais de válido e de recebido. O `ap_vld` inclui apenas o sinal de válido e o `ap_ack` inclui apenas o sinal de recebido. O protocolo `ap_ovld` tem os mesmos sinais do `ap_vld`, mas é aplicado a portos de entrada/saída ou apenas de saída. O protocolo de interface de um porto é especificado com a diretiva de interface:

```
#pragma HLS INTERFACE mode=<nome do protocolo> port=<nome do porto>
```

O protocolo `ap_hs` pode ser aplicado a parâmetros passados por valor, por ponteiro ou referência, a vetores multidimensionais e a estruturas de dados sequenciais (`stream`), desde que os dados sejam acedidos sequencialmente. O acesso a memórias aleatórias não é possível com este protocolo.

Os sinais do protocolo incluem um porto de dados, um sinal de válido que indica a validade dos dados e um sinal de recebido que indica que os dados foram lidos. Este conjunto de sinais aplica-se igualmente, quer num porto de entrada, quer num porto de saída. Por exemplo, um módulo hardware com um porto de entrada e outro de saída, ambos com o protocolo `ap_hs`, tem uma interface como ilustrado na Figura 6-2.

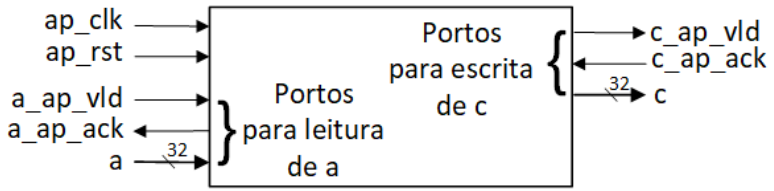


Figura 6-2 – Interface do módulo hardware como um porto de entrada e outro de saída, ambos com o protocolo `ap_hs`.

O protocolo baseado nos sinais válido e recebido determina que a leitura ou a escrita dos dados se dá quando ambos os sinais ficam ativos, como ilustrado na Figura 6-3.

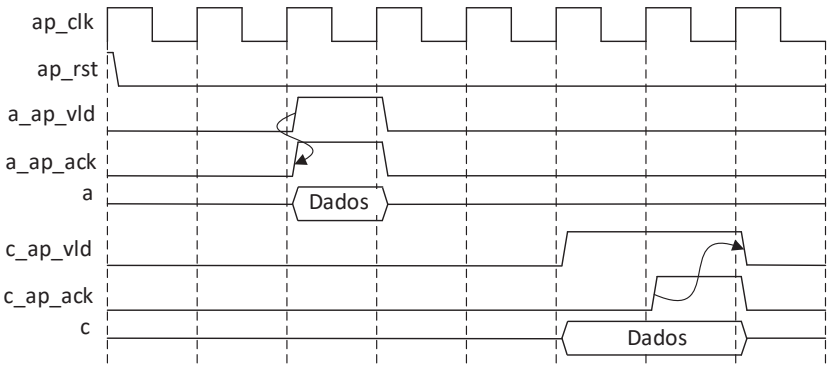


Figura 6-3 – Exemplo de evolução temporal dos sinais associados ao protocolo `ap_hs` para um porto de entrada e um porto de saída.

No porto de entrada (`a`), o módulo está à espera do sinal de válido (`a_ap_vld`) ativo. Com o sinal ativo, o módulo lê os dados (`a`) e ativa o sinal de recebido (`a_ap_ack`) indicando que o emissor dos dados já os pode retirar da entrada. No porto de saída, o módulo ativa o sinal de válido (`c_ap_vld`) assim que os dados (`c`) estiverem disponíveis para serem lidos. Estes permanecem na saída até que seja ativo o sinal de recebido (`c_ap_ack`). Nesta altura, o módulo retira os dados e desativa o sinal de válido.

Este tipo de protocolo garante que os dados são lidos e escritos, sem que o escalonamento do módulo hardware tenha de saber o momento exato em que vão ocorrer.

Os protocolos `ap_vld` e `ap_ack` são versões simplificadas do protocolo completo `ap_hs` e apenas se aplicam a ponteiros ou referências. O protocolo `ap_vld` não inclui o sinal de recebido. Quando utilizado num porto de entrada, o sinal de válido indica que os dados estão disponíveis. O módulo que envia os dados não sabe que foram lidos, pois não há sinal de recebido. Para que os dados não sejam perdidos, o módulo recetor guarda os dados

quando recebe o sinal de dados válidos. No caso de um porto de saída, o módulo de hardware ativa o sinal de válido assim que tiver os dados prontos, mas não recebe qualquer informação que indique se o módulo recetor já os leu. Este tipo de protocolo pode ser utilizado nos casos em que o recetor está sempre disponível para ler os dados e consegue fazê-lo quando estiverem disponíveis.

O protocolo `ap_ack` não inclui o sinal de válido. Quando utilizado num porto de entrada, o sinal de recebido indica que os dados foram lidos. O módulo que envia os dados não indica que os dados são válidos, pois não há sinal de válido. No caso de um porto de saída, o sinal de recebido indica que os dados foram lidos, mas o módulo que gera os dados não tem de indicar a sua validade. Este tipo de protocolo pode ser utilizado nos casos em que o recetor conhece o momento em que os dados estão disponíveis.

O protocolo `ap_ovld` é também uma simplificação do `ap_hs` que não utiliza o sinal de recebido. A diferença relativamente ao protocolo `ap_vld` é que nos portos de entrada assume o protocolo `ap_none`, ou seja também não tem sinal de válido. Este protocolo é utilizado em argumentos que podem ser de entrada/saída ou apenas de saída. No caso em que são de entrada/saída, o porto é desdobrado num porto de entrada e num de saída. Depois, por omissão, o porto de entrada fica com o protocolo `ap_none` e o porto de saída fica como `ap_vld`.

6.2.3 Protocolo com Interface de Memória

O protocolo para interface a memória inclui os sinais para acesso a uma memória que pode ser de acesso aleatório ou de acesso sequencial. No caso de ser acesso aleatório, utiliza-se o protocolo `ap_memory` ou `bram`, no caso de ser sequencial também se pode utilizar o protocolo `ap_fifo`.

Os protocolos `ap_memory` e `bram` são utilizados para acesso a estruturas de dados guardados em memória de acesso aleatório, como RAM e ROM, sendo apenas associados a vetores multidimensionais. Os protocolos incluem os mesmos sinais de acesso a memória (endereço, leitura e escrita). A diferença está em que uma interface `bram` é gerada como um único porto agrupado, ao passo que uma interface `ap_memory` é gerada com os sinais separados.

A Figura 6-4 ilustra duas operações de leitura e duas operações de escrita considerando uma memória de duplo porto e o protocolo `ap_memory`. A primeira leitura acede a um elemento de memória isolado e as duas leituras seguintes ocorrem em sequência. As escritas ocorrem também em sequência e em paralelo com as leituras, uma vez que a memória é de duplo porto.

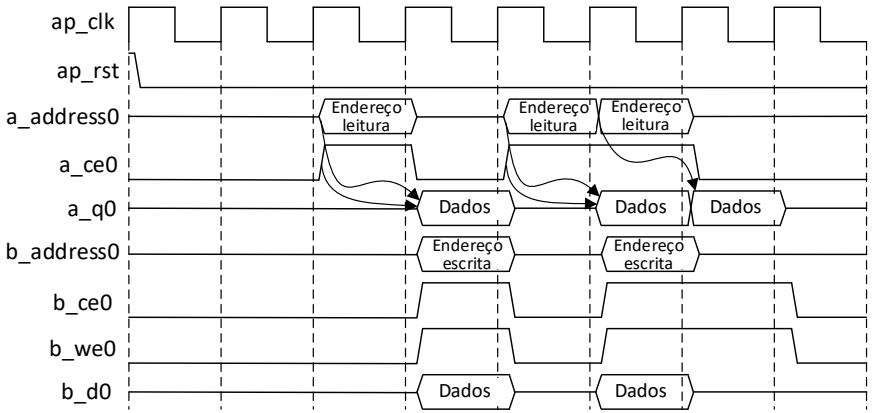


Figura 6-4 – Exemplo de evolução temporal dos sinais associados ao protocolo `ap_memory` para operações de leitura e de escrita através do mesmo porto.

Uma operação de leitura começa por colocar o endereço (`a_address0`) e ativar o porto da memória (`a_ce0`). No ciclo seguinte os dados (`a_q0`) ficam disponíveis. No caso de uma leitura em sequência, os acessos ocorrem em períodos de relógio seguidos. Para realizar uma escrita, colocam-se o endereço (`b_address1`), os dados (`b_d0`) e ativam-se os sinais de ativação do porto (`b_ce0`) e de escrita (`b_we0`). Os dados são escritos na memória ao fim do ciclo de relógio. As escritas também podem ser feitas em sequência, mantendo o sinal de escrita sempre ativo e alterando os endereços e dados a cada ciclo de relógio.

O protocolo `ap_fifo` pode ser associado a vetores multidimensionais, a ponteiros ou referências e para acessos sequências do tipo `stream`. No entanto, só permite acesso sequencial aos dados. De entre os protocolos para interface de memória, é o aconselhado nos casos em que o acesso à memória é feito de forma sequencial, pois simplifica a implementação da interface. O protocolo assume a ligação a uma FIFO com sinais de controlo `empty/full` e de escrita/leitura, para além dos sinais relativos aos dados de entrada e de saída. O sinal `empty` indica que a FIFO não contém dados e, como tal, não deve ser lida. O sinal `full` indica que a FIFO está completa e não pode receber mais dados. Neste caso, não devem ser feitas escritas na FIFO. Para realizar a leitura de um elemento da FIFO, deve ativar-se a entrada de leitura que faz avançar o endereço interno de leitura da FIFO. Para realizar uma escrita, deve ativar-se o sinal de escrita que faz avançar o endereço interno de escrita da FIFO.

O `ap_fifo` não pode ser associado a um argumento da função que seja simultaneamente lido e escrito, mas apenas a argumentos que sejam apenas de entrada ou de saída. Isto porque a FIFO tem sempre dois portos unidirecionais, um de entrada e outro de saída. A Figura 6-5 ilustra operações de acesso a uma FIFO para escrita e para leitura com o protocolo `ap_fifo`.

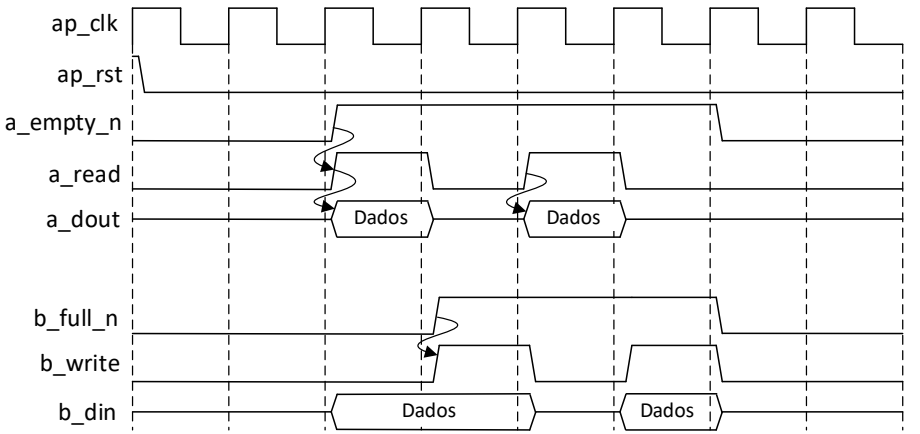


Figura 6-5 – Exemplo de evolução temporal dos sinais associados ao protocolo `ap_fifo` para operações de leitura e de escrita.

Os sinais de FIFO vazia (`a_empty_n`) e FIFO cheia (`b_full_n`) são ativos a 0. No exemplo de leitura da FIFO, só quando o sinal de FIFO vazia fica a 1, indicando que contém dados, é que se ativa o sinal de leitura (`a_read`). Os dados surgem no mesmo ciclo (`a_dout`). Enquanto a FIFO não estiver vazia, podem ser feitas leituras. No caso de escrita na FIFO, é preciso esperar que a FIFO deixe de estar cheia (`b_full_n`). Só nessa altura é que se pode ativar o sinal de escrita (`b_write`). Os dados podem manter-se à entrada da FIFO antes de serem escritos.

6.3 Protocolos de Interface ao Nível do Bloco

Enquanto os protocolos de interface ao nível do porto permitem controlar individualmente cada um dos portos do módulo hardware, um protocolo ao nível do bloco controla o módulo hardware como um todo. O *Vitis HLS* considera três protocolos ao nível do bloco distintos:

`ap_ctrl_chain` – Permite que o módulo inicie uma nova execução antes de a anterior terminar, ou seja, permite uma execução em *pipeline*;

`ap_ctrl_hs` – Apenas permite que o módulo inicie uma nova execução após terminar a anterior. Protocolo utilizado por omissão;

`ap_ctrl_none` – Permite que o módulo seja controlado pela disponibilidade de dados, ou seja, inicia a execução assim que os dados de entrada estiverem disponíveis. Corresponde a não ter um protocolo de controlo do bloco.

O protocolo de interface ao nível do bloco é associado à função ou ao retorno da função, mesmo que a função não retorne um valor. No caso de a função retornar um valor, o *Vitis*

HLS cria um porto de saída designado `ap_return`. O Código 6-2 ilustra a especificação do protocolo de bloco.

```
Void func(int a[4], int b[4]){
#pragma HLS INTERFACE mode=ap_ctrl_hs port=return

int i;

for (i = 0; i < 4; i ++)
       b[i] = a[i] + 1;

}
```

Código 6-2 – Exemplo de especificação de um protocolo ao nível do bloco

No exemplo, foi escolhido o protocolo *ap_ctrl_hs*. Repare-se que o protocolo foi especificado associado ao porto de retorno, apesar da função não ter retorno.

O protocolo `ap_ctrl_hs` inclui quatro sinais de controlo:

- `ap_start`: quando ativo indica que o módulo pode iniciar a operação;
- `ap_idle`: quando ativo indica que o módulo não está a executar nenhuma operação;
- `ap_ready`: quando ativo indica que o módulo pode executar. Só recomeça a execução após terminar a execução atual;
- `ap_done`: quando ativo indica que o módulo terminou a execução.

A Figura 6-6 ilustra o protocolo `ap_ctrl_hs` e a relação entre os sinais.

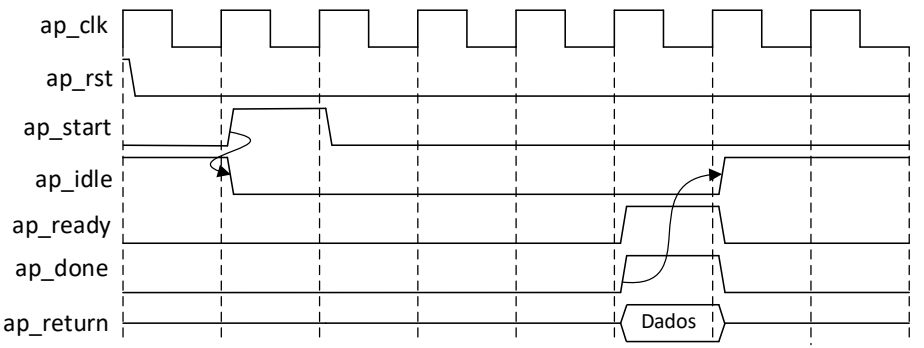


Figura 6-6 – Exemplo de evolução temporal dos sinais associados ao protocolo `ap_ctrl_hs` *para operações de leitura e de escrita.*

Inicialmente, o bloco espera pelo sinal `ap_start` para iniciar a execução. Assim que `ap_start` fica ativo, o sinal *ap_idle* fica inativo para indicar que o módulo está ocupado. No entanto, o módulo só inicia a execução quando o sinal *ap_ready* é ativado pelo módulo,

indicando que está pronto para iniciar a execução. Quando o módulo termina a execução, ativa o sinal `ap_done`.

O comportamento dos sinais varia com a utilização de *pipeline*. No caso de uma solução sem *pipeline*, os sinais `ap_ready` e `ap_done` ficam ativos ao mesmo tempo, ou seja, assim que o módulo termina a execução. No caso de um circuito com *pipeline*, o módulo pode iniciar uma nova execução, sem que a anterior tenha terminado, ou seja, o sinal `ap_ready` pode ficar ativo antes do módulo terminar a execução.

O protocolo `ap_ctrl_chain` tem um funcionamento similar ao protocolo `ap_ctrl_hs`, mas inclui um porto de entrada adicional (`ap_continue`) que indica se o bloco que recebe os dados está pronto para receber mais dados. Se o bloco que envia os dados deteta que o `ap_continue` está inativo, então mantém o resultado à saída, com o sinal `ap_done` ativo e aguarda. Só após receber `ap_continue` ativo é que desativa o sinal `ap_done` e inicia a próxima operação, se `ap_start` estiver ativo.

Quando não se pretende associar um protocolo ao nível de bloco, especifica-se o protocolo como `ap_ctrl_none`.

6.4 Descrição HLS com Protocolos de Interface

Nesta secção, aborda-se através de vários exemplos de que forma a síntese de interface lida com os vários tipos de argumentos, como se aplicam os protocolos de interface ao nível do porto e do bloco e qual o hardware gerado para cada caso. Consideram-se os vários tipos de argumento: escalar, vetor multidimensional, ponteiro e referência, e dados em *stream*.

6.4.1 Passagem de Parâmetros por Valor - Escalar

Um parâmetro passado por valor significa que a função recebe diretamente uma cópia da variável. Exemplifiquemos a passagem de parâmetro por valor com um caso simples em que se somam os argumentos de dois parâmetros de entrada passados por valor (ver Código 6-3).

```
int soma (int a, int b){

int res;

res = a + b;

return res;
}
```

Código 6-3 – Exemplo de uma função que soma dois parâmetros passados por valor

A ferramenta de síntese irá gerar um circuito hardware combinatório de soma com 32 bits (`int`). As entradas, `a` e `b`, e a saída, `res`, são simplesmente implementadas como fios.

No caso em que o parâmetro é acedido várias vezes durante a execução da função, o seu argumento é registado e depois lido sem possibilidade de ser modificado. Consideremos

um exemplo de acumulação dos elementos de um vetor, em que um parâmetro é utilizado repetidamente dentro do corpo de uma estrutura de repetição (ver Código 6-4).

```
int somaC (int a, int b[4]){

int acc = 0, i;

for (i = 0; i < 4; i++){
    acc = acc + b[i];
    if (acc < a)
        acc = a;
}

return acc;
}
```

Código 6-4 – Exemplo de uma função em que o parâmetro passado por valor é lido várias vezes

No exemplo, a variável `a`, passada por valor, é registada no início, antes de começar a execução do `for`. Assim, ao longo da execução do `for`, o valor de `a` é lido do registo, em vez de ser constantemente lido da interface. A implementação do vetor `b` será vista mais à frente.

O circuito correspondente à função descrita no Código 6-4 seria sintetizado de acordo com o diagrama da Figura 6-7.

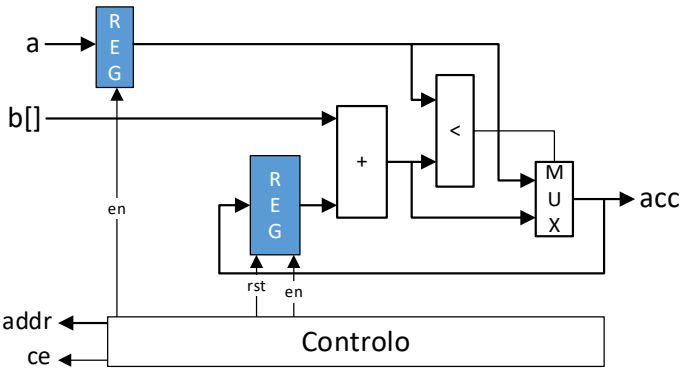


Figura 6-7 – Implementação hardware do ciclo descrito no Código 6-4 com o parâmetro 'a' passado por valor

O registo de acumulação é atualizado com o valor de `a` ou com a soma entre o seu valor e o elemento do vetor. Esta escolha é feita através do *multiplexer* à direita da figura, cujo seletor depende da comparação entre o resultado da soma e o valor de `a`, que é guardado no início da execução do circuito num registo.

Por omissão, os parâmetros passados por valor são implementados como fios. Isto pressupõe que os dados de entrada estão sempre disponíveis durante a execução da função, ou seja, não existe um protocolo de pedido e resposta (*handshaking*). Não cabe à função e ao módulo de hardware respetivo, após síntese, garantir que os dados estão disponíveis em qualquer altura durante a execução da função. Este tipo de entrada ou de saída está geralmente associado a sinais que se mantêm constantes ao longo de toda a execução do componente. Como exemplo, temos o acesso a parâmetros dentro de uma estrutura de repetição em *pipeline*, situação em que os dados são acedidos em todas as iterações.

O resultado da função quando devolvido através de um `return` (Código 6-3), também não tem, por omissão, um sinal associado que indique que o resultado está disponível. Neste caso, é difícil saber quando é que o resultado está correto e pode ser lido. Veremos mais â frente como resolver esta questão.

Como vimos anteriormente, o tipo de protocolo associado a cada um dos argumentos é determinado por omissão pela síntese de interface em função do tipo de argumento. No entanto, o projetista tem total controlo sobre o protocolo associado às interfaces e pode alterá-lo através de diretivas da ferramenta. Por exemplo, no caso da função anterior descrita no Código 6-4, podemos associar um protocolo específico.

```
int exemploSoma (int a, int b[4]){
#pragma HLS INTERFACE mode=ap_ctrl_none port=return
#pragma HLS INTERFACE mode=ap_hs port=a
#pragma HLS INTERFACE ap_memory port=b

int acc = 0, i;

for (i = 0; i < 4; i++){
    acc = acc + b[i];
    if (acc < a)
        acc = a;
}

return acc;
}
```

Código 6-5 – Exemplo da função descrita no Código 6-4, com descrição explícita dos protocolos ao nível do porto.

No exemplo ilustrado no Código 6-5, não foi associado qualquer protocolo ao porto de retorno (`ap_ctrl_none`). Ao parâmetro `a` foi associado o protocolo `ap_hs` e ao parâmetro `b` o protocolo `ap_memory`. Isto corresponde a ter a interface do bloco hardware como ilustrada na Figura 6-8.

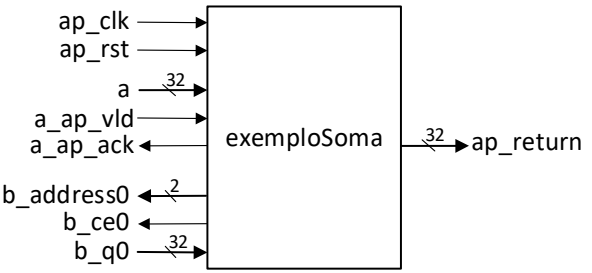


Figura 6-8 – Implementação hardware do ciclo descrito no Código 6-5 com o parâmetro 'a' passado por valor

No bloco da figura, observam-se todos os sinais de cada um dos protocolos associados aos parâmetros da função.

6.4.2 Passagem de Parâmetros por Referência

Um parâmetro da função passado por referência significa que a função recebe o endereço do parâmetro. Por exemplo, tratando-se de uma variável, a função recebe o endereço de memória onde a variável se encontra alojada.

Ambas as linguagens C e C++ suportam a passagem de parâmetros por referência. Na linguagem C, utiliza-se o ponteiro e o operador de indireção, `*`, para descrever a passagem de parâmetro por referência. O ponteiro é uma variável que contém o endereço de memória a que se pretende aceder. Na linguagem C++ utilizam-se os conceitos de ponteiro e de referência. A referência corresponde a associar um outro nome ao objeto apontado. Tal como um ponteiro, a referência contém o endereço a aceder, mas a referenciação é feita de forma automática sem necessidade de utilizar o operador `*`.

Os dados a que a referência diz respeito encontram-se guardados fora da função. Ao receber o argumento de uma referência, a função tem de aceder ao endereço respetivo para ler o valor guardado nessa posição.

Consideremos um exemplo de soma de duas variáveis: uma passada por ponteiro e outra por referência (ver Código 6-6).

```
int multAdd_ref (volatile int *a, volatile int &b){

int acc, i;

acc = 0;
for (i = 0; i < 4; i++)
       acc += *a * b;

return acc;
}
```

Código 6-6 – Exemplo de uma função em C++ que determina o produto interno entre dois vetores. O parâmetro `a` é passado por ponteiro e o `b` por referência

Os parâmetros são declarados como sendo do tipo *volatile* porque são lidos várias vezes durante a execução da função. Um parâmetro passado por referência deve ser declarado como *volatile* se for acedido múltiplas vezes durante uma execução da função. Caso não seja declarado como *volatile*, a ferramenta de síntese considera que o parâmetro é lido apenas uma vez e o seu argumento mantém-se constante ao longo da execução da função.

Os valores de `a` e de `b` são lidos a cada ciclo do sinal de relógio (CLK) e o seu produto é acumulado na variável `acc` (assumindo que um ciclo de relógio é suficiente para executar todas estas operações). O escalonamento é feito com um intervalo de iteração igual a 1 (ver escalonamento na Figura 6-9).

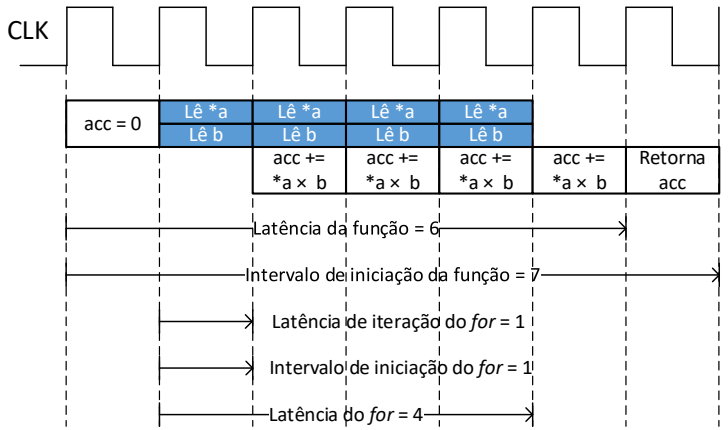


Figura 6-9 – Escalonamento da função `multAdd()` com argumentos passados por ponteiro e por referência

O escalonamento do circuito com *pipeline* executa alternadamente a leitura dos argumentos e a operação de multiplicação-acumulação. Os argumentos não são registados dentro da função. A solução assume que os dados de entrada estão disponíveis ciclo a ciclo. O resultado pode ser lido após o sexto ciclo de execução da função.

A implementação do circuito não considera registos nas entradas, como acontecia na passagem de parâmetros por valor. O circuito apenas necessita de um operador de multiplicação, outro de soma e um registo para guardar o valor da acumulação do resultado da multiplicação (ver Figura 6-10).

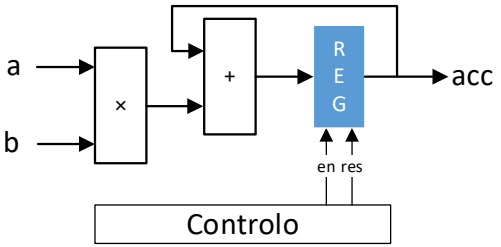


Figura 6-10 – Implementação hardware do ciclo descrito no Código 6-6 com o parâmetro `a` *passado por ponteiro e o* `b` *passado por referência*

Neste exemplo, a inicialização do registo de acumulação é feita com a entrada de *reset* (`res`) durante o primeiro ciclo de relógio. Os argumentos lidos são enviados diretamente para os operadores aritméticos e o resultado guardado no registo. O valor do registo é enviado diretamente para a saída do circuito para ser lido pelo circuito externo.

Por omissão, os argumentos passados por ponteiro ou por referência não têm qualquer protocolo de controlo associado. O circuito externo tem de garantir que os dados estão disponíveis de acordo com o escalonamento do circuito hardware sintetizado. O bloco é, por omissão, controlado pelo protocolo de bloco `ap_ctrl_hs`.

A memória a que o argumento da função faz referência pode ser de diversos tipos, dependendo do tipo de argumento. Por omissão, a interface de um argumento passado por referência ou ponteiro assume que o valor acedido está armazenado num registo. No caso de um ponteiro, os valores podem estar guardados em registo ou em memória sequencial. Isto pode ser alterado com diretivas de configuração da interface. Consideremos o mesmo exemplo, mas com a descrição explícita dos protocolos que queremos associar aos argumentos (ver Código 6-7).

```
int multAdd_Int (volatile int *a, volatile int &b){
#pragma HLS INTERFACE mode=ap_fifo port=a
#pragma HLS INTERFACE mode=ap_hs port=b

int acc, i;

acc = 0;
for (i = 0; i < 4; i++)
      acc += *a * b;

return acc;
}
```

Código 6-7 – Exemplo da função C++ que determina o produto interno entre dois vetores com indicação dos protocolos a associar a cada argumento

Neste exemplo, ao ponteiro associamos o protocolo de acesso a FIFO (`ap_fifo`) e à referência o protocolo `ap_hs`. O resultado será um bloco hardware com a interface descrita na Figura 6-11.

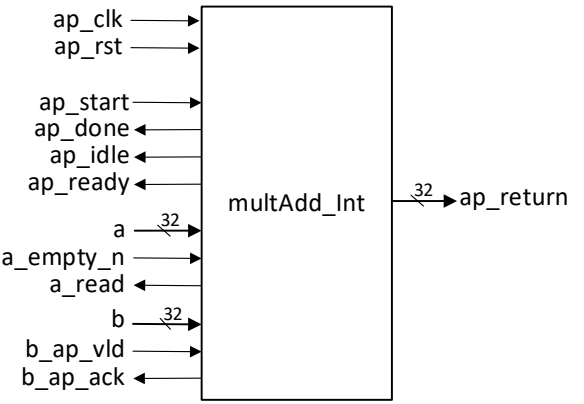


Figura 6-11 – Implementação hardware do ciclo descrito no Código 6-7 com o ponteiro `a` implementado com protocolo de acesso a FIFO e o `b` com um protocolo `ap_hs`

Os sinais do porto `a` incluem o barramento de dados, o sinal de controlo de FIFO vazia (`a_empty_n`) e o sinal de leitura da FIFO (`a_read`). Os sinais do porto `b` são os já conhecidos do protocolo `ap_hs`. Adicionalmente, inclui os sinais do protocolo de bloco e o retorno sem sinais de controlo.

O resultado da função pode ser devolvido com um *return*, cujo porto não tem, por omissão, um protocolo associado que permita saber quando é que o resultado está disponível. Em alternativa, o resultado pode ser devolvido com uma referência ou com um ponteiro. Recorrendo ao exemplo anterior, consideremos que o retorno é feito com o ponteiro `a` (ver Código 6-8).

```
void multAdd_Int (volatile int *a, volatile int &b){

int acc, i;

acc = 0;
for (i = 0; i < 4; i++)
       acc += *a * b;

a* = acc;
}
```

Código 6-8 – Exemplo da função C++ que determina o produto interno entre dois vetores com indicação dos protocolos a associar a cada argumento

O ponteiro `a` é utilizado como entrada e como saída (retorno). Não havendo descrição de diretivas, serão associados os protocolos por omissão a todos os portos. A interface do

bloco sintetizado irá ficar com o conjunto de sinais do protocolo ao nível do bloco, o ponteiro sintetizado com um porto de entrada sem protocolo de porto e um porto de saída com o protocolo `ap_valid`. O porto `b` também não tem qualquer protocolo de porto.

6.4.3 Passagem de Parâmetros com Vetor Multidimensional

Os vetores de qualquer dimensão são passados para uma função como referências. Por exemplo, um parâmetro da função declarado como vetor faz referência ao vetor guardado numa memória externa à função, em que a latência de acesso a um elemento do vetor é, por omissão, de um ciclo de relógio.

Consideremos, para exemplificar, o cálculo do produto interno entre dois vetores de dimensão 4, em que os vetores são recebidos como parâmetros da função (ver Código 6-9).

```
int multAddv2 (int a[4], int b[4]){
#pragma HLS INTERFACE mode=ap_ctrl_none port=return

int acc, i;
acc = 0;

for (i = 0; i < 4; i++){
      acc += a[i] + b[i];
}
return acc;
}
```

Código 6-9 – Exemplo de uma função em que os dois parâmetros são vetores. A função calcula o produto interno entre os dois vetores

Cada um dos parâmetros da função, `a` e `b`, é declarado como um vetor de 4 elementos. Neste caso, a síntese gera, por omissão, uma interface a memória aleatória, com portos de endereço de 2 bits para aceder à memória onde os vetores estão armazenados.

A síntese da descrição do Código 6-9 produziria um circuito em que a interface inclui os portos indicados na Figura 6-12.

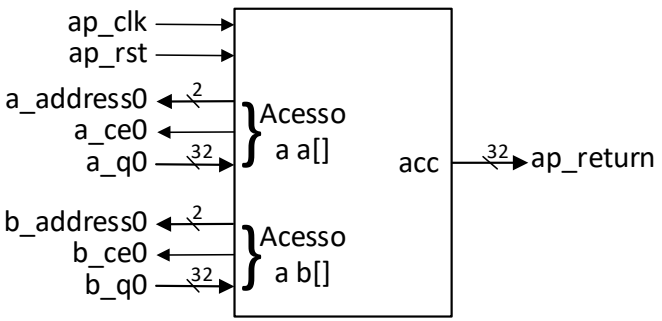


Figura 6-12 – Interface do módulo hardware gerado a partir do Código 6-9 com dois acessos a memória RAM

Sendo um circuito sequencial, foram adicionadas entradas de relógio (`ap_clk`) e de inicialização (`ap_rst`). Os portos de acesso aos vetores têm associados um endereço (`a_address0` e `b_address0`), um sinal de leitura dos dados (`a_ce0` e `b_ce0`) e os dados de entrada (`a_q0` e `b_q0`). O resultado surge na saída `ap_return`. Os dados são de 32 bits, pois trata-se de inteiros, e os sinais de endereço são de 2 bits, pois são vetores de quatro elementos.

O tamanho dos vetores pode ser omitido na declaração dos parâmetros (ver Código 6-10).

```
int multAddv3 (volatile int a[], volatile int b[]){
#pragma HLS INTERFACE mode=ap_ctrl_none port=return

int acc, i;

acc = 0;
for (i = 0; i < 4; i++){
       acc += a[i] * b[i];

return acc;
}
```

Código 6-10 – Exemplo de uma função que determina o produto interno entre dois vetores, em que os vetores são passados para a função como vetores sem dimensão

Se omitirmos o tamanho dos vetores na declaração dos parâmetros, a síntese de alto nível gera uma implementação idêntica à obtida caso se declarassem os parâmetros como ponteiros, como visto anteriormente. Neste caso, a interface não tem linhas de endereço.

Repare-se que os vetores foram declarados como voláteis, para que a ferramenta de síntese perceba que é necessário ler o vetor por cada acesso que ocorra na função. No caso de não se declarar o argumento como volátil, o HLS interpreta como um ponteiro que acede a um valor fixo. Além disso, uma vez que os ponteiros são associados por omissão a registos, a síntese do Código 6-10 gera apenas interfaces de acesso a registo, ou seja, unicamente com o barramento de dados.

6.4.4 Configuração da Memória da Interface

Vimos anteriormente que os vetores na interface são sintetizados com protocolos de acesso à memória. O *Vitis HLS* permite ao projetista influenciar a síntese da memória associada à interface através de diretivas da ferramenta. As configurações possíveis incluem:

- O tipo de memória da interface utilizado no armazenamento. A memória pode ser do tipo RAM ou FIFO. Por omissão, um vetor é armazenado numa memória RAM;
- O número de portas da RAM, caso o armazenado seja feito numa RAM. A RAM pode ser especificada como sendo de porto simples ou de porto duplo;
- A latência de acesso à RAM.

Estas configurações procuram melhorar o desempenho da solução reduzindo as limitações no acesso aos vetores. A eficácia das técnicas de *pipeline* e de desenrolamento estão em

muitos casos condicionadas pelo acesso aos dados. A reorganização dos dados em memória e o aumento dos portos de acesso aos dados são, assim, fundamentais para permitir o acesso paralelo a múltiplos dados da interface.

A memória associada a uma interface para armazenar, por exemplo, um vetor, pode ser RAM ou ROM, no caso de ser um vetor de constantes, ou FIFO. A utilização de uma FIFO como forma de armazenamento na interface evita que o módulo hardware a implementar tenha de gerar portos de endereços, ao contrário do que acontece quando se considera RAM. Contudo, a utilização de uma FIFO implica que os dados tenham de ser acedidos sequencialmente.

O projetista pode indicar explicitamente o tipo de memória da interface através da diretiva `HLS INTERFACE`.

`#pragma HLS INTERFACE mode=<modo> port=<nome> [opções]`

`mode` - especifica o protocolo da interface dos argumentos ou da função

`port` – especifica o nome do argumento ou o retorno da função

Opções:

`bundle` – permite agrupar argumentos nos mesmos portos

`channel` – permite agrupar múltiplos canais numa interface m_axi

`clock` – especifica o sinal de relógio a usar numa interface AXI4-Lite

`depth` – Indica o número máximo de elementos a considar num *testbench*

`interrupt` – Permite a gestão da entrada/saída por interrupção. Apenas nos protocolos ap_vld/ap_hs

`latency` – especifica a latência esperada em interfaces AXI ou ap_memory

`max_read_burst_length`, **`max_write_burst_length`** – especifica o número máximo de valores acedidos durante uma transmissão em rajada

`name` – nome do porto

`num_read_outstandingm num_write_outstanding` – especifica o número de pedidos num barramento AXI4 sem resposta

`offset` – controla o endereço de offset no AXI4

`register` – força o registo do sinal

`register_mode` – indica no *AXI4-Stream* que sinais são registados

`storage_type` – especifica o tipo de armazenamento numa interface ap_memory

Quando o projeto indica explicitamente que pretende utilizar uma FIFO para implementar a memória de uma interface, a ferramenta de síntese procura determinar se é possível utilizar uma FIFO de acordo com os acessos feitos ao vetor. Obtém-se um de três casos possíveis como resultado desta análise:

- A síntese determina que o acesso é sequencial e implementa a memória com FIFO;

- A síntese determina que o acesso não é sequencial e não implementa a memória com FIFO. O processo de síntese termina com um erro;
- A síntese não consegue determinar se o acesso é sequencial. Neste caso, segue a indicação do projetista e implementa a memória com uma FIFO.

Comecemos por considerar um caso em que a síntese consegue determinar que o acesso é sequencial (ver Código 6-11).

```
int multAddv4 (int a[4], int b[4]){
      int acc, i;
#pragma HLS INTERFACE mode=ap_fifo port=a
#pragma HLS INTERFACE mode=ap_fifo port=b
#pragma HLS INTERFACE ap_ctrl_none port=return

acc = 0;
for (i = 0; i < 4; i++)
      acc += a[i] * b[i];

return acc;
}
```

Código 6-11 – Exemplo de uma função com a interface dos vetores especificada como FIFO, em que a ferramenta de HLS consegue determinar que o acesso aos vetores é sequencial.

Através da diretiva `HLS INTERFACE` especificou-se a interface da memória como sendo do tipo FIFO. A função `multAddv4` inclui uma estrutura de repetição que acede aos elementos dos vetores `a` e `b`, em sequência. A ferramenta de síntese determina, desta forma, que a memória dos vetores da interface pode ser implementada com FIFO. A interface é sintetizada com sinais de acesso a FIFO (ver Figura 6-13).

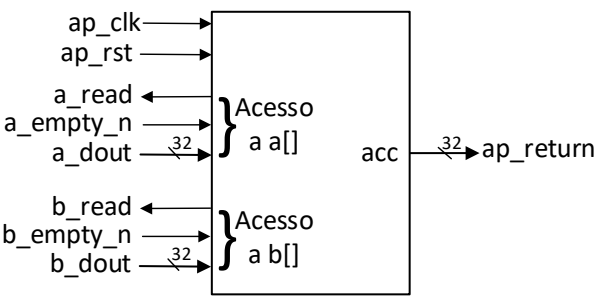


Figura 6-13 – Interface do módulo hardware gerado a partir do Código 6-11 com dois acessos a memória FIFO

A cada um dos vetores da interface são associados sinais de dados a 32 bits (`a_dout` e `b_dout`), de leitura da FIFO (`a_read` e `b_read`) e sinais que indicam se a FIFO está vazia ou não (`a_empty_n` e `b_empty_n`). Repare-se que não são gerados sinais de endereço, pois trata-se de acessos a FIFO.

Para exemplificar um caso em que a ferramenta de síntese não consegue determinar se o acesso ao vetor é sequencial ou aleatório, considere-se o mesmo exemplo, mas em que os índices dos vetores são dados por um vetor de entrada (ver Código 6-12).

```
int multAddv5 (int a[4], int b[4], int idx[4]){
	int acc, i;
#pragma HLS INTERFACE mode=ap_fifo port=a
#pragma HLS INTERFACE mode=ap_fifo port=b
#pragma HLS INTERFACE mode=ap_fifo port=idx
#pragma HLS INTERFACE ap_ctrl_none port=return

	acc = 0;
	for (i = 0; i < 4; i++)
		acc += a[idx[4]] * b[idx[4]];

	return acc;
}
```

Código 6-12 – Exemplo de uma função com a interface dos vetores especificada como FIFO, mas em que a ferramenta de síntese não consegue determinar se o acesso aos vetores é sequencial.

O exemplo considera que o acesso ao vetor `idx` é feito também através de uma FIFO, mas poderia ser através de uma RAM. Neste caso, a ferramenta de síntese não consegue determinar se o acesso aos vetores `a` e `b` é sequencial ou não, uma vez que durante a síntese não conhece os valores dos elementos do vetor `idx`. Se, por exemplo, `idx` = [0, 1, 2, 3], o acesso aos vetores é sequencial. Mas se, por exemplo, `idx` = [0, 3, 1, 2], o acesso não é sequencial. Como a ferramenta de síntese não consegue determinar em tempo de compilação se é sequencial ou não, segue a diretiva do projetista e mantém a FIFO. Contudo, gera um aviso a indicar que o resultado pode não ser válido se a leitura dos vetores não for sequencial.

Nos casos em que a síntese consegue determinar em tempo de compilação que um vetor não é acedido sequencialmente, gera um erro a indicar a situação e termina a síntese.

O pragma `HLS INTERFACE` também permite especificar o tipo de memória RAM associada a uma interface. Para isso, é utilizado o argumento `storage_type=<valor>` que se aplica apenas a memórias de acesso aleatório. Os tipos de memória podem ser: `ram_1p` (RAM porto simples), `ram_1wnr` (RAM com um porto de escrita e N portos de leitura), `ram_2p` (RAM com dois portos, um para leitura e outro para leitura e escrita), `ram_s2p` (RAM com um porto de leitura e outro de escrita), `ram_t2p` (RAM com dois portos de escrita e/ou de leitura), `rom_1p` (ROM com um porto simples), `rom_2p` (ROM com dois portos), e `rom_np` (ROM com múltiplos portos).

Também é possível especificar a latência de acesso à memória em interfaces AXI, que veremos mais à frente. Isto é feito através do argumento `latency=<valor>`. Esta informação é utilizada pela ferramenta de síntese para determinar quando deve iniciar o

acesso à memória. Se o valor não estiver de acordo com a verdadeira latência de acesso à memória, podem ocorrer dois casos durante o funcionamento do módulo hardware: (1) o número de ciclos necessários para aceder à memória é menor que o valor indicado, pelo que os dados na interface ficam disponíveis mais cedo e esperam até que sejam acedidos pelo módulo; (2) o número de ciclos necessários para aceder à memória é maior que o indicado, pelo que o módulo de hardware irá ficar a aguardar até que os dados estejam disponíveis na interface para serem usados.

6.4.5 Configuração da Estrutura da Memória Associada à Interface

Aquando da introdução da otimização com pipeline e desenrolamento no capítulo 5, alertou-se para a importância da configuração do acesso aos vetores de modo a impedir que o acesso limite a eficácia do método de *pipeline* e/ou do desenrolamento e, consequentemente, impeça a melhoria do tempo de execução do componente hardware.

Um parâmetro da função do tipo vetor é implementado, por omissão, com uma memória RAM de porto simples ou porto duplo, caso a ferramenta de síntese conclua que a utilização de um porto duplo melhora a latência e a eficácia da *pipeline* e do desenrolamento.

Vejamos um exemplo de uma função em que não se consegue obter um intervalo de iniciação da estrutura de repetição unitário devido à limitação no acesso ao vetor (ver Código 6-13).

```
int AddVector (int a[12]){

int acc, i;

acc = 0;
for (i = 0; i < 12; i += 4)
       acc += a[i] + a[i+1] + a[i+2] + a[i+3];

return acc;
}
```

Código 6-13 – Exemplo de uma função em que a memória de interface não tem portos suficientes que permita o acesso paralelo a quatro elementos do vetor, impedindo que se obtenha um intervalo de iniciação da estrutura de repetição igual a 1

Para obter um intervalo de iniciação (II) igual a 1, é necessário ler quatro valores do vetor em cada ciclo de relógio. Como a memória do vetor apenas dispõe de um máximo de dois portos, não é possível ler os quatro elementos ao mesmo tempo sem realizar alterações ao modo como o vetor está guardado na memória. O melhor intervalo de iniciação da estrutura `for` que se consegue com esta descrição é de 2 ciclos de relógio, pois o porto duplo da memória permite ler dois valores em paralelo.

Uma forma de conseguir ultrapassar a limitação no acesso aos dados consiste em dividir o vetor original em vetores mais pequenos e guardar cada um destes subvetores em memórias separadas. Por cada memória adicionada, estão a adicionar-se dois novos portos

de acesso ao vetor original. A síntese de alto nível dispõe da diretiva ARRAY_PARTITION que subdivide automaticamente o vetor.

#pragma HLS ARRAY_PARTITION variable=<nome> [type=<tipo>] factor=<int> dim=<int> off=true

variable=<nome> – especifica o nome da variável a que se quer aplicar a partição

type=<tipo> – especifica o tipo de partição: *cyclic*, *block*, *complete* (valor por omissão)

factor=<int> – especifica o número de subvetores a serem criados

dim=<int> – especifica a dimensão a que se pretende aplicar a partição no caso de estruturas de armazenamento multidimensionais

off – desabilita a partição da variável

Existem vários formas de partição dos vetores:

- Bloco (*block*) – o vetor original é dividido em blocos mais pequenos (o número de blocos é especificado com o parâmetro '*factor*' da diretiva. Se o número de elementos do vetor não for múltiplo do fator de partição, o último subvetor fica com menos elementos. Por exemplo, a partição de um vetor, a, de N elementos com o tipo bloco e um fator de 2 resultaria na seguinte divisão dos elementos:

subvetor 0	a[0]	a[1]	a[2]	…	a[N/2-2]	a[N/2-1]
subvetor 1	a[N/2]	a[N/2+1]	a[N/2+2]	…	a[N-2]	a[N-1]

  Como se pode observar, o subvetor 0 ficou com os primeiros N/2 elementos do vetor original e o subvetor 1 ficou com os últimos N/2 elementos;

- Cíclico (*cyclic*) – o vetor original também é, neste tipo, dividido em blocos mais pequenos, em que o número de blocos é especificado com o parâmetro '*factor*'. A diferença em relação ao tipo bloco é que os elementos do vetor são entrecruzados. Por exemplo, a partição do mesmo vetor, a, com o tipo cíclico e um fator de 2 resultaria na seguinte divisão dos elementos:

subvetor 0	a[0]	a[2]	a[4]	…	a[N-4]	a[N-2]
subvetor 1	a[1]	a[3]	a[5]	…	a[N-3]	a[N-1]

Os elementos do vetor original foram guardados alternadamente nos subvetores 0 (com os valores de índice par) e 1 (com os valores de índice ímpar);

- Completo (*complete*) – O vetor original é completamente separado em registos, sendo possível aceder a todos os elementos do vetor em paralelo. Este é o modo por omissão, mas tem um custo de hardware proporcional ao tamanho do vetor.

Se aplicarmos a partição do vetor ao exemplo anterior, é possível obter uma execução em *pipeline* com um intervalo de iniciação da estrutura de repetição igual a 1 (ver Código 6-14)

```
int AddVectorv2 (int a[12]){

int acc, i;

#pragma HLS ARRAY_PARTITION factor=2 type=cyclic variable=a

acc = 0;
for (i = 0; i < 12; i += 4)
      acc += a[i] + a[i+1] + a[i+2] + a[i+3];

return acc;
}
```

Código 6-14 – Exemplo de uma função com partição da memória de interface em dois blocos de modo a permitir o acesso paralelo a quatro elementos do vetor, possibilitando que se obtenha um intervalo de iniciação do `for` igual a 1.

O vetor original foi partido em dois subvetores com metade do tamanho e depois guardados em duas memórias separadas com uma partição cíclica. No conjunto, as duas memórias disponibilizam quatro portos de acesso, suficientes para garantir o acesso a quatro elementos do vetor em paralelo e, assim, obter um intervalo de iniciação do `for` igual a 1.

O processo de partição também pode ser aplicado a estruturas de dados com mais dimensões. O parâmetro `dim` da diretiva de partição especifica a dimensão da estrutura que se pretende particionar. O valor `dim=0` é reservado para especificar que se pretende aplicar o tipo e o fator de partição a todas as dimensões. Consideremos a estrutura multidimensional `a[6][4]`. Aplicando a diretiva de partição a esta estrutura com fator máximo produziria as seguintes subdivisões, de acordo com o valor da dimensão da diretiva:

```
#pragma HLS ARRAY_PARTITION ... dim=1

a[6][4] → a_0[4], a_1[4], a_2[4], a_3[4], a_4[4], a_5[4]

#pragma HLS ARRAY_PARTITION ... dim=2

a[6][4] → a_0[6], a_1[6], a_2[6], a_3[6]

#pragma HLS ARRAY_PARTITION ... dim=0

a[6][4] → a_00, a_10, a_20, a_30, a_40, a_50
          a_01, a_11, a_21, a_31, a_41, a_51
```

a_02, a_12, a_22, a_32, a_42, a_52

a_03, a_13, a_23, a_33, a_43, a_53

No último caso, a estrutura de dados é particionada por completo sendo separada em 6×4=24 elementos.

Um outro método de permitir o acesso a múltiplos elementos de um vetor consiste em concatenar elementos com o aumento do tamanho do barramento de dados, ou seja, redimensionar (*reshape*) o vetor. Ao contrário do método de partição da memória que obriga a aumentar o número de memórias, o redimensionamento reduz o número de memórias ao mesmo tempo que permite o acesso paralelo aos elementos do vetor. O redimensionamento do vetor pode ser realizado manualmente ou utilizando a diretiva `#pragma HLS ARRAY_RESHAPE`.

`#pragma HLS ARRAY_RESHAPE variable=<nome> [type=<tipo>] factor=<int> dim=<int> off=true`

`variable=<nome>` - especifica o nome da estrutura de dados a que se quer aplicar o redimensionamento.

`type=<tipo>` – especifica o tipo de redimensionamento: *cyclic*, *block*, *complete* (valor por omissão).

`factor=<int>` – especifica a subdivisão da estrutura de dados.

`dim=<int>` – especifica a dimensão de estruturas de dados multidimensionais a que se pretende aplicar a partição

`off` – desabilita o redimensionamento da variável

Os vários tipos de redimensionamento do vetor são equivalentes aos existentes na partição dos vetores:

- Bloco (*block*) – o vetor original é dividido em blocos mais pequenos (o número de blocos é especificado com o parâmetro '*factor*' da diretiva. Depois combina os blocos mais pequenos num vetor único com o tamanho dos elementos igual ao tamanho original multiplicado pelo fator de divisão. Por exemplo, o redimensionamento de um vetor, `a`, com N elementos com o tipo bloco e um fator de 2 resultaria no seguinte novo vetor:

novo vetor	a[N/2]:a[0]	a[N/2+1]:a[1]	…	a[N-2]:a[N/2-2]	a[N-1]:a[N/2-1]

  O novo vetor passa a ter dois elementos do vetor original em cada posição do novo vetor. Por exemplo, se o tamanho dos elementos for de 32 bits, o novo vetor tem elementos de 64 bits e metade das posições de memória;

- Cíclico (*cyclic*) – o vetor original é dividido em blocos mais pequenos, em que o número de blocos também é especificado com o parâmetro '*factor*'. Neste caso, os elementos do vetor são entrecruzados. Por exemplo, o dimensionamento do mesmo vetor, `a`, com o tipo cíclico e um fator de 2 resultaria no redimensionamento seguinte:

novo vetor	a[1]:a[0]	a[3]:a[2]	…	a[N-3]:a[N-4]	a[N-1]:a[N-2]

  Os elementos do novo vetor contêm elementos ordenados do vetor original;
- Completo (*complete*) – O vetor original é completamente redimensionado resultando num registo com tamanho necessário para conter todos os elementos do vetor concatenados. Este é o modo de redimensionamento por omissão.

O exemplo de soma anterior, poderia obter igualmente um intervalo de iniciação do `for` unitário por aplicação do redimensionamento (ver Código 6-15)

```
int AddVectorV3 (int a[12]){

int acc, i;

#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS ARRAY_RESHAPE factor=2 type=cyclic variable=a

	acc = 0;
	for (i = 0; i < 12; i += 4)
		acc += a[i] + a[i+1] + a[i+2] + a[i+3];

	return acc;
}
```

Código 6-15 – Exemplo de uma função com redimensionamento da memória de interface num único bloco com duplicação do tamanho dos elementos do vetor.

Neste exemplo, a interface do módulo hardware gerado teria apenas um acesso a uma memória de duplo porto, mas em que o barramento de dados tem o dobro do tamanho (ver Figura 6-14).

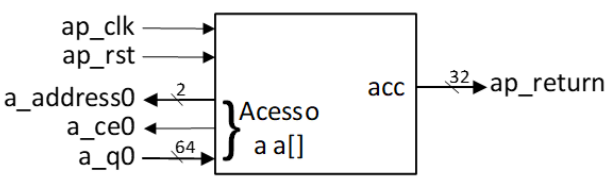


Figura 6-14 – Interface do módulo hardware gerado a partir do Código 6-15 com redimensionamento do vetor, passando a considerar elementos de 64 bits.

No bloco da Figura 6-14, a entrada de dados tem 64 bits, que permite enviar dois elementos do vetor em cada acesso.

6.5 Implementação da Interface com Protocolos AXI

O protocolo ao nível do porto associado aos argumentos da função também pode ser baseado numa interface AXI. O *Vitis HLS* tem as opções seguintes: *AXI4-Full* (`m_axi`), Escravo *AXI4-Lite* (`s_axilite`) ou *AXI4-Stream* (`axis`).

A interface *AXI4-Full* é aplicada a vetores multidimensionais e a ponteiros ou referências. A interface *AXI4-Lite* aplica-se a qualquer tipo de argumento, exceto a argumentos do tipo *stream*. É possível agrupar múltiplos argumentos na mesma interface *AXI4-Full* ou *AXI4-Lite*. Por fim, a interface tipo *AXI4-Stream* apenas se aplica a argumentos de entrada ou de saída com acesso sequencial.

6.5.1 Protocolo com Interface AXI4-Full

O protocolo *AXI4-Full* pode ser utilizado com argumentos que mapeiam em memória. Pode funcionar em modo de transferência individual, em que é feito o acesso a um elemento de memória por cada endereço, ou em modo de transferência em rajada (*burst*), em que é feito um acesso sequencial a múltiplos elementos em memória especificando unicamente o endereço inicial. O acesso em modo rajada permite taxas de transferência bastante elevadas pois, após o envio do primeiro endereço, os dados são transferidos em sequência, sem necessidade de enviar mais endereços.

Se a função fizer apenas uma leitura do argumento durante a execução, então é feita apenas uma transferência individual. A transferência em rajada ocorre quando se acede a uma sequência de dados durante a execução. O modo de transferência em rajada pode ser associado a um determinado porto de forma explícita ou implícita. A função C `memcpy` pode ser utilizada para configurar explicitamente o porto em modo de transferência em rajada. Por exemplo, consideremos uma função que adiciona a constante 10 aos elementos de um vetor (ver Código 6-16).

```
#include <string.h>

void doBurst (volatile int *data){
#pragma HLS INTERFACE mode=m_axi depth=100 port=data
#pragma HLS INTERFACE mode=s_axilite port=return

int i, mem[100];

memcpy(mem, (const int*)data, 100*sizeof(int));

REPETE: for(i=0; i < 100; i++){
       mem[i] = mem[i] + 10;
}

memcpy((int *)data, mem, 100*sizeof(int));
}
```

Código 6-16 – Exemplo de utilização da função `memcpy` para transferir dados numa interface AXI4-Fill em modo de transferência em rajada

O argumento da função é associado a uma interface do tipo *AXI4-Full* (*m_axi*) através da diretiva `#pragma HLS INTERFACE mode=m_axi depth=100 port=data`. Como descrito anteriormente, o parâmetro `depth` é utilizado para a co-simulação determinar o tamanho máximo da FIFO necessária para a simulação.

A função `memcpy` é sintetizada como uma transferência em rajada através de uma interface *AXI4-Full*. A primeira execução da função copia os dados de entrada para uma memória interna. De seguida, os elementos são lidos sequencialmente desta memória, adicionados a 10 e escritos de novo na memória interna na mesma posição. Por fim, os dados da memória interna são transferidos também em rajada pela mesma interface *AXI4-Full*. O total de ciclos de relógio de execução da função fica assim próximo de 300, ou seja, cerca de 100 ciclos de relógio em cada um dos passos de execução mencionados.

É possível obter o mesmo resultado descrevendo a transferência de dados com *pipeline* (ver Código 6-17).

```
void doBurst (volatile int *data){
#pragma HLS INTERFACE mode=m_axi depth=100 port=data

int i, mem[100];

LOOP0: for(i=0; i < 100; i++){
       mem[i] = data[i];
}

LOOP1: for(i=0; i < 100; i++){
       mem[i] = mem[i] + 10;
}

LOOP2: for(i=0; i < 100; i++){
       data[i] = mem[i];
}

}
```

Código 6-17 – Exemplo de descrição da transferência de dados em rajada com pipeline através de uma interface AXI4-Full

A descrição da transferência em rajada com uma estrutura de repetição deve seguir algumas regras: a estrutura de repetição deve ter *pipeline*, o acesso aos dados deve ser em sequência e não deve depender de condições, e, caso a estrutura de repetição faça parte de estruturas de repetição entrelaçadas, deve manter-se a separação das estruturas.

Os argumentos da função podem ser associados a interfaces distintas. Consideremos um exemplo em que são lidos dois vetores através de interfaces diferentes e o resultado é enviado por uma das interfaces (ver Código 6-18).

```
void dualBurst (volatile int *data0, volatile int *data1){
#pragma HLS INTERFACE mode=m_axi depth=100 port=data0
#pragma HLS INTERFACE mode=m_axi depth=100 port=data1 bundle=p2

int i, mem[100];

REPETE0: for(i=0; i < 100; i++){
	mem[i] = data0[i] * data1[i];
}

REPETE1: for(i=0; i < 100; i++){
	data0[i] = mem[i];
}

}
```

Código 6-18 – Transferência de dados em rajada através de duas interfaces AXI4-Full

A descrição associa uma interface *AXI4-Full* distinta a cada um dos argumentos da função. Para tal, utiliza o parâmetro `bundle` da diretiva HLS INTERFACE com um valor diferente na descrição da segunda interface para transferir `data1`. O parâmetro `bundle` da diretiva associada à interface `data0` é omitido, pelo que assume uma interface distinta.

Os diferentes argumentos da função também podem ser associados à mesma interface AXI. O exemplo anterior pode ser modificado para associar os dois argumentos à mesma interface AXI (`bundle`) (ver Código 6-19).

```
void dualBurst (volatile int *data0, volatile int *data1){
#pragma HLS INTERFACE mode=m_axi depth=100 port=data0 bundle=p2
#pragma HLS INTERFACE mode=m_axi depth=100 port=data1 bundle=p2

int i, mem[100];

REPETE0: for(i=0; i < 100; i++){
	mem[i] = data0[i];
}

REPETE1: for(i=0; i < 100; i++){
	data0[i] = mem[i] * data1[i];
}

}
```

Código 6-19 – Transferência de dados em rajada através de duas interfaces AXI-Full

Começa-se por especificar a transferência completa do vetor `data0` para memória interna (estrutura `REPETE0`). Depois, na estrutura `REPETE1`, os valores em memória interna são multiplicados pelos elementos do vetor `data1`, lido da memória externa, e os resultados da multiplicação são reescritos no vetor `data` (também em memória externa). Em ambos os casos, os dados são recebidos e enviados pela mesma interface *AXI4-Full*. Recorde-se que a interface *AXI4-Full* tem canais de escrita e de leitura independentes, pelo que permite leitura e escrita em simultâneo.

O *Vitis HLS* associa todos os portos cuja declaração não inclua o parâmetro `bundle` a uma única interface *AXI* com um nome pré-definido. Portos cuja declaração defina o nome do `bundle` são associados a interfaces diferentes da pré-definida. Portos com o mesmo nome de `bundle` são agrupados na mesma interface.

6.5.2 Protocolo com Interface AXI4-Lite

A interface *AXI4*-Lite escrava (`s_axilite`) é geralmente utilizada para a comunicação entre um processador e um bloco hardware. Permite, por exemplo, a escrita e a leitura em registos de dados ou de controlo do módulo hardware. A especificação dos argumentos com uma interface *AXI4-Lite* é similar à feita com uma interface *AXI4-Full* (Código 6-20).

```
void simpleT (int *a, int *b){
#pragma HLS INTERFACE mode=s_axilite port=a bundle=BUS1
#pragma HLS INTERFACE mode=s_axilite port=b bundle=BUS1
#pragma HLS INTERFACE mode=s_axilite port=return bundle=BUS1
#pragma HLS INTERFACE mode=ap_vld port=a

int i;

*b = *a + 10;

}
```

Código 6-20 – Descrição dos argumentos da função com interface AXI4-Lite

Neste exemplo, a transferência de dados e de sinais do protocolo de interface do bloco é feita através de uma única interface *AXI4-Lite*. Para além de associar os parâmetros à interface AXI, também se pode alterar os protocolos associados à interface ao nível do porto e ao nível do bloco. No exemplo, o porto `a` foi configurado com um protocolo `ap_valid`.

A síntese de interface gera um adaptador *AXI-Lite* juntamente com o bloco hardware (ver Figura 6-15).

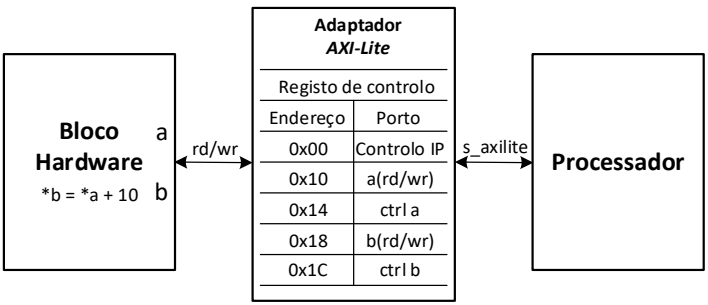


Figura 6-15 – Adaptador AXI-Lite.

O adaptador implementa o protocolo *AXI4-Lite* para comunicar com o mestre (p.ex., processador), comunica com o bloco hardware para permitir as operações de leitura e de escrita entre o bloco e o adaptador e implementa o mapa de endereços relativos a registos

da interface utilizados para guardar os argumentos da função e os sinais de controlo da interface ao nível do bloco. O acesso ao núcleo hardware através de uma interface *AXI4-Lite* faz-se lendo e escrevendo nos registos da interface.

No exemplo, o porto `a` é implementado com o protocolo `ap_vld`, como indicado na diretiva `INTERFACE`. O porto `b` é implementado, por omissão, com o protocolo `ap_vld`, uma vez que se trata de uma saída.

O mapa de endereços do adaptador relativo ao exemplo descrito no Código 6-20 é ilustrado na Figura 6-16.

```
-- 0x00 : Control signals
--        bit 0  - ap_start (Read/Write/SC)
--        bit 1  - ap_done (Read/COR)
--        bit 2  - ap_idle (Read)
--        bit 3  - ap_ready (Read/COR)
--        bit 7  - auto_restart (Read/Write)
--        others - reserved
-- 0x04 : Global Interrupt Enable Register
--        bit 0  - Global Interrupt Enable (Read/Write)
--        others - reserved
-- 0x08 : IP Interrupt Enable Register (Read/Write)
--        bit 0  - enable ap_done interrupt (Read/Write)
--        others - reserved
-- 0x0c : IP Interrupt Status Register (Read/TOW)
--        bit 0  - ap_done (COR/TOW)
--        others - reserved
-- 0x10 : Data signal of a
--        bit 31~0 - a[31:0] (Read/Write)
-- 0x14 : Control signal of a
--        bit 0  - a_ap_vld (Read/COR)
--        others - reserved
-- 0x18 : Data signal of b
--        bit 31~0 - b[31:0] (Read)
-- 0x1c : Control signal of b
--        bit 0  - b_ap_vld (Read/COR)
--        others - reserved
```

Figura 6-16 – Mapa de endereços associados aos registos da interface AXI4-Lite do exemplo descrito no Código 6-20.

O endereço `0x00` contém os sinais de controlo do protocolo `ap_ctrl_hs`. Cada um dos sinais está associado a um bit do registo de controlo. Seguem-se os endereços `0x04`, `0x08`, `0x0C` para acesso a registos de controlo da interrupção. O argumento `a` encontra-se mapeado no registo com endereço `0x10` e o sinal de válido no endereço `0x14`. O argumento `b` encontra-se mapeado no registo com endereço `0x18`. O sinal de válido associado ao porto `b` encontra-se mapeado no endereço `0x1c`.

Numa situação normal de utilização do módulo hardware, começa-se por enviar o valor do argumento `a` seguido do sinal de `ap_start`. A ativação deste sinal faz-se colocando o bit

de menor peso do registo no endereço 0x00 a '1'. O fim da operação pode ser confirmado pela leitura do sinal `ap_done`, o segundo bit de menor peso do registo com endereço 0x00.

A comunicação entre o bloco hardware e o adaptador *AXI4-Lite* faz-se com os protocolos ao nível do porto. A atribuição faz-se de acordo com as regras vistas anteriormente para os diferentes tipos de argumentos. A única exceção reside no mapeamento dos vetores que, por omissão, são associados ao protocolo `ap_memory`.

A atribuição de portos diferentes à mesma interface também é feita com o parâmetro `bundle` da diretiva `HLS INTERFACE`. As regras de agrupamento de portos em interfaces *AXI4 Lite* são idênticas às aplicadas nas interfaces *AXI4-Full*. Todos os portos que não tenham o parâmetro `bundle` definido são associados a uma única interface *AXI* com um nome pré-definido. Os portos que incluam a descrição do nome do `bundle` são associados a interfaces diferentes da pré-definida. Portos com o mesmo nome de `bundle` são agrupados na mesma interface. O exemplo no Código 6-21 ilustra uma função em que os argumentos são associados a interfaces *AXI4-Lite* diferentes.

```
void multipleAXILite (int *a, int *b, int *c, int *d){
#pragma HLS INTERFACE mode=s_axilite port=a
#pragma HLS INTERFACE mode=s_axilite port=b
#pragma HLS INTERFACE mode=s_axilite port=c bundle=BUS1
#pragma HLS INTERFACE mode=a_axilite port=d bundle=BUS1

 (...)
}
```

Código 6-21 – Atribuição dos argumentos da função a diferentes interfaces AXI4-Lite

Os argumentos `a` e `b` são atribuídos a uma interface comum, e os argumentos `c` e `d` são atribuídos também a uma interface comum, mas diferente da utilizada por `a` e `b`. A distinção é feita pelo `bundle`, como descrito anteriormente.

6.5.3 Protocolo com Interface AXI4-Stream

O protocolo *AXI4-Stream* aplica-se a qualquer tipo de argumento de entrada ou de saída. A transferência de dados é feita em sequência de forma unidirecional. Como tal, não são suportados argumentos de entrada/saída. Numa transferência de dados com *AXI4-Stream* existe apenas um produtor que envia os dados e apenas um consumidor que recebe os dados. Como descrito anteriormente, para além do sinal de dados válidos (`TVALID`) e o sinal de pronto a receber (`TREADY`), o *AXI4-Stream* inclui o sinal que indica o fim da sequência (`TLAST`).

O mapeamento de um argumento da função numa interface de comunicação de dados sequencial (*streaming*) pode ser especificada de várias formas. Similar ao que foi feito com as interfaces AXI anteriores, podemos simplesmente declarar os argumentos como sendo *AXI4-Stream* (`axis`) (ver Código 6-22).

```
void argSerie (int a[100], int b[100]){
#pragma HLS INTERFACE mode=ap_ctrl_none port=return
#pragma HLS INTERFACE mode=axis port=a
#pragma HLS INTERFACE mode=axis port=b

int i;

REPETE:for (i= 0; i < 100; i++){
       b[i] = a[i] + 10;
        }
}
```

Código 6-22 – Declaração dos argumentos da função com a diretiva `HLS INTERFACE`

O HLS cria duas interfaces *AXI4-Stream*, uma de entrada e outra de saída, com os sinais `TDATA`, `TVALID` e `TREADY`.

Contudo, esta forma de declaração das interfaces do tipo *AXI4-Stream* não é aconselhada porque o projetista pode inadvertidamente aceder ao vetor de forma não sequencial, quando o acesso por *AXI4-Stream* obriga a um acesso sequencial.

Quando se pretende uma interface para comunicação de dados em sequência (*stream*), devemos utilizar a estrutura `hls::stream`. Um argumento da função declarado com este tipo de dados é implementado, por omissão, com uma interface FIFO (`ap_fifo`), mas também pode ser implemento com uma interface `ap_hs` ou com uma interface *AXI4-Stream* (`axis`).

Consideremos, como exemplo, uma função que adiciona o valor 10 aos elementos de um vetor (ver Código 6-23).

```
#include <ap_int.h>
#include "hls_stream.h"

void argSerie(hls::stream<int> &a, hls::stream<int> &b){

int i, tmp;

REPETE:for (i= 0; i < 4; i++){
       a.read(tmp);
       tmp += 10;
       b.write(tmp);
}}
```

Código 6-23 – Declaração dos argumentos da função com a estrutura de dados hls::stream

O corpo da estrutura de repetição inclui uma leitura, um cálculo e uma escrita. Por omissão, é criada uma interface FIFO de entrada (`ap_fifo`) e uma outra interface FIFO de saída (`ap_fifo`). O tipo de protocolo da interface pode ser alterado para *AXI4-Stream* com a diretiva `HLS INTERFACE` (ver Código 6-24).

```
#include <ap_int.h>
#include "hls_stream.h"

void argSerie(hls::stream<int> &a, hls::stream<int> &b){
#pragma HLS INTERFACE mode=axis port=a
#pragma HLS INTERFACE mode=axis port=b

(...)
}
```

Código 6-24 – Declaração dos argumentos da função com a estrutura de dados hls::stream e com a interface declarada como AXI4-Stream

No exemplo, ambos os argumentos foram associados a interfaces *AXI4-Stream* (`axis`). Neste caso, a ferramenta de síntese gera uma interface com os sinais `TDATA`, `TVALID` e `TREADY`.

No caso de ser necessário incluir outros sinais do protocolo *AXI4-Stream*, deve usar-se o tipo de dados `hls::axis` definido em `ap_axi_sdata.h`.

`hls::axis<T, WUser, WId, WDest>`

`T` – tipo de dados.

`WUser` – tamanho em bits do sinal TUSER

`WId` – tamanho em bits do sinal TID

`WDest` – tamanho em bits do sinal TDEST

Os parâmetros determinam a utilização dos sinais `TUSER`, `TID`, `TDEST`. Quando a zero, estes sinais não são incluídos na interface. Os restantes sinais, `TVALID`, `TREADY`, `TKEEP`, `TSTRB` e `TLAST` são adicionados automaticamente.

No caso em que os dados (`T`) são do tipo inteiro, existem dois tipos de dados pré-definidos

- Inteiros com sinal:
  `hls::axis<ap_int<WData>, WUser, WId, WDest>`

  ou

  `ap_axis<Wdata, WUser, WId, WDest>`

- Inteiros sem sinal:
  `hls::axis<ap_uint<WData>, WUser, WId, WDest>`

  ou

  `ap_axiu<Wdata, WUser, WId, WDest>`

Na implementação da interface *AXI4-Stream* todos os sinais são registados por omissão. No entanto, é possível forçar a que apenas alguns ou nenhum dos sinais seja registado com o parâmetro `register_mode` da diretiva `HLS INTERFACE`. As opções são: `Off` (não registados), `Forward` (`TDATA` e `TVALID` registados) e `Reverse` (`TREADY` registado).

Consideremos um exemplo de uma função em que os argumentos são transferidos através de uma interface *AXI4-Stream* especificada com o tipo de dados `ap_axis` (ver Código 6-25).

```
#include <ap_int.h>
#include "ap_axi_sdata.h"
#include "hls_stream.h"

typedef ap_axis<32, 0, 0, 0> dataType;

void axi4STR(hls::stream<dataType> &a, hls::stream<dataType> &b){
#pragma HLS INTERFACE mode=ap_ctrl_none port=return
#pragma HLS INTERFACE mode=axis port=a
#pragma HLS INTERFACE mode=axis port=b

int i;
dataType tmp;

for (;;){
      a.read(tmp);
      tmp.data += 10;
      tmp.keep = -1;
      tmp.strb = -1;
      b.write(tmp);
      if (tmp.last == 1)
            break;
}
}
```

Código 6-25 – Declaração dos argumentos da função com a estrutura de dados hls::axis e utilização dos sinais da estrutura.

Inicialmente, foi definida uma estrutura (`dataType`) com os sinais `TUSER`, `TID` e `TDEST` a 0, ou seja, inativos. No corpo da função, definiu-se uma estrutura de repetição sem limite controlada pelo sinal `TLAST`. Os dados da interface são lidos até que se receba o sinal `TLAST` a indicar que se trata do último elemento da sequência de dados. Os sinais `TKEEP` e `TSTRB` tomam o valor -1, indicando que todos os bytes devem ser lidos.

O acesso e o processamento dos dados devem ser feitos com variáveis temporárias. Ler o elemento para a variável temporária, processar sobre a variável e escrever a variável na saída.

6.6 Otimização do Acesso aos Dados com Memória Interna

O tempo de acesso aos elementos de um vetor guardado em memória influencia o tempo de execução de uma função, pois determina quantos elementos se conseguem ler por ciclo de relógio. Como foi descrito nas secções anteriores, o acesso pode ser aleatório ou sequencial, dependendo se se trata de uma memória de acesso aleatório (RAM ou ROM) ou uma memória de acesso sequencial (FIFO). Um acesso aleatório a uma estrutura de dados implica que os dados têm de ser armazenados numa memória de acesso aleatório, enquanto um acesso sequencial pode ser feito a uma memória de acesso aleatório ou sequencial. Em ambos os casos, quando um vetor é definido nos argumentos da função, a ferramenta de síntese assume que a memória é externa à função e gera os sinais de interface necessários para aceder à memória.

O acesso a uma memória externa está sempre dependente da sua disponibilidade, caso seja uma memória partilhada, e da velocidade de acesso à mesma. A função pode ter de suspender a execução se estiver à espera de um acesso à memória.

Aquando do projeto de um módulo que acede a estruturas de dados multidimensionais, pode considerar-se que é feito um acesso externo à memória sempre que são necessários dados. Em alternativa, o projetista pode optar por criar uma memória interna para armazenar temporariamente os dados. Neste caso, os dados são lidos da memória externa e guardados na memória interna, e posteriormente acedidos pela função. Esta solução tem diversas vantagens:

- A leitura da memória externa pode ser lida de forma aleatória ou sequencial;
- Os dados são guardados em memória interna com a configuração que melhor se adequa aos acessos realizados pela função;
- Quando armazenados em memória interna, os dados não precisam de ser lidos repetidamente da memória externa sempre que necessários;
- A memória interna está sempre disponível e o acesso é feito à velocidade do núcleo *hardware*.

Naturalmente que a utilização de uma ou mais memórias internas acarreta um custo em termos de recursos utilizados para implementar a memória e o circuito de controlo da mesma. Cabe ao projetista optar pela melhor solução, tendo em conta o compromisso entre o desempenho, o custo e o padrão de acessos à memória.

Para gerar um módulo hardware com memória interna basta declarar a estrutura de dados dentro da função. Consideremos, como exemplo, a soma de todos os elementos de um vetor, em que o vetor é inicialmente guardado em memória interna (ver Código 6-26).

```
#include "ap_axi_sdata.h"
#include "hls_stream.h"

typedef ap_axis<32, 0, 0, 0> trans_pkt;

int ioIntMem(hls::stream<trans_pkt> &strm_in){
#pragma HLS INTERFACE mode=axis port=strm_in

int i, j, acc;

static int lmem[100];
trans_pkt tmp;

DADOS:for (i=0; i<100; i++) {
	tmp = strm_in.read();
	lmem[i] = tmp.data;
	if (tmp.last == 1)
		break;
}

acc = 0;
SOMA:for(j=0; j < 100; j++){
	acc += lmem[j];
	if (i == j)
		break;
}

return acc;
}
```

Código 6-26 – Exemplo de utilização de memória interna para guardar um vetor lido através de uma interface AXI4-Stream.

A implementação utiliza uma memória interna no módulo hardware para armazenar o vetor. A primeira estrutura de repetição (`DADOS`) carrega o vetor em memória interna. De seguida, na segunda estrutura de repetição (`SOMA`), os elementos do vetor que estão guardados em memória interna são acumulados.

A declaração do vetor como estático garante que o HLS implementa o vetor como uma memória. Além disso, também determina o resultado da inicialização do vetor. Se se declarar um vetor ou outra estrutura multidimensional com valores iniciais sem utilizar o tipo estático, a estrutura em memória é inicializada com estes valores iniciais sempre que a função for executada. Este processo implica a escrita na memória interna. No caso de o vetor ser declarado como estático, apenas é inicializado na primeira execução da função. No caso da síntese em FPGA, a inicialização da memória é integrada com o ficheiro de configuração da FPGA, pelo que não é necessário copiar os valores para a memória interna na primeira execução, pois estes já lá se encontram.

Assim, recomenda-se que as estruturas de dados internas a serem implementadas com memória sejam declaradas como estáticas. Esta opção garante que a estrutura de dados é

sintetizada como memória. Além disso, também garante que no caso da inicialização da estrutura, esta só acontece uma vez.

As diretivas de partição e de redimensionamento utilizadas anteriormente para organizar as estruturas de dados associadas a uma interface podem ser aplicadas igualmente a estruturas guardadas em memória interna. No exemplo anterior, vários elementos do vetor podem ser adicionados em paralelo. Para tal, é preciso garantir o acesso paralelo a vários destes elementos. Por exemplo, caso se pretenda adicionar quatro elementos por cada ciclo de relógio, é necessário ler quatro valores em paralelo. Considerando uma memória de duplo porto, em cada porto terão de ser lidos dois elementos em paralelo (ver Código 6-27).

```
#include "ap_axi_sdata.h"
#include "hls_stream.h"

typedef ap_axis<32, 0, 0, 0> trans_pkt;

int ioEx(hls::stream<trans_pkt> &strm_in){
#pragma HLS INTERFACE mode=axis port=strm_in

int i, j, acc;

static int lmem[100];
#pragma HLS ARRAY_RESHAPE dim=1 factor=2 type=cyclic variable=lmem

trans_pkt tmp;

DADOS:for (i=0; i<100; i++) {
	tmp = strm_in.read();
	lmem[i] = tmp.data;
	if (tmp.last == 1)
		break;
}

acc = 0;
SOMA:for(j=0; j < 100; j +=4){
	acc += lmem[j] + lmem[j+1] + lmem[j+2] + lmem[j+3];
	if (i == j)
		break;
}

return acc;
}
```

Código 6-27 – Aplicação da diretiva de partição a um vetor armazenado em memória interna.

A diretiva redimensiona o vetor com uma estrutura cíclica e um fator de 2, ou seja, os elementos do vetor são armazenados em pares na forma a[1]:a[0], a[3]:a[2], etc., em que cada elemento do novo vetor passa a ter 64 bits. A memória é configurada com duplo porto, tendo cada porto um tamanho de dados de 64 bits. Assim, conseguem-se ler quatro elementos de 32 bits, um total de 128 bits, em cada ciclo de relógio. A Figura 6-17 ilustra o

circuito que resulta da síntese da função, considerando uma interface de entrada *AXI4-Stream* com 32 bits de dados.

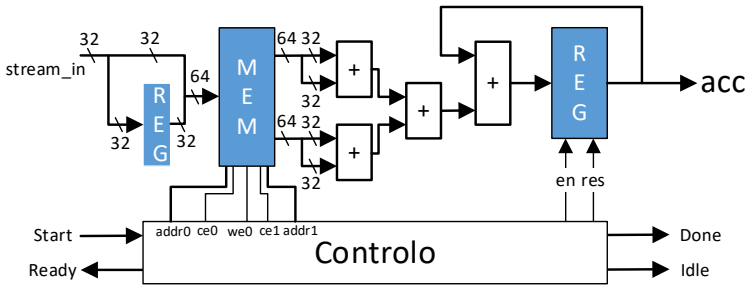


Figura 6-17 – Resultado da síntese do Código 6-27 com a utilização de uma memória interna de duplo porto com 64 bits de dados.

Os elementos do vetor são recebidos através de uma interface *AXI4-Stream* de 32 bits de dados. Os dados são guardados aos pares na memória interna (`MEM`). O primeiro elemento de cada par é guardado num registo e o segundo é concatenado com o valor guardado no registo para formar os 64 bits.

A partição ou o redimensionamento automático da memória para resolver as restrições no acesso aos dados em paralelo nem sempre conduz à melhor solução. Por vezes, é preciso reestruturar o código e considerar uma arquitetura de memória adequada para obter uma solução melhor. Consideremos, como exemplo, um filtro de 4 elementos a uma dimensão. O filtro é aplicado sequencialmente aos elementos de um vetor, sendo multiplicados e acumulados (ver Código 6-28).

```
#include "ap_axi_sdata.h"
#include "hls_stream.h"

typedef ap_axis<32, 0, 0, 0> trans_pkt;

void acumula(hls::stream<trans_pkt> &strm_in,
             hls::stream<trans_pkt> &strm_out){
#pragma HLS INTERFACE mode=axis port=strm_in
#pragma HLS INTERFACE mode=axis port=strm_out

trans_pkt tmp;

int i, j, acc, b;
const int coef[4] = {1, 2, 3, 4};
static int lmem[1000];
#pragma HLS ARRAY_PARTITION dim=1 factor=4 type=cyclic variable=lmem

DADOS:for (i=0; i<100; i++) {
       strm_in.read(tmp);
       lmem[i] = tmp.data;
```

```
        if (tmp.last == 1)
              break;
}

ACUMULA:for(j=0; j < 97; j++){
        b = lmem[j]*coef[0] + lmem[j+1]*coef[1] +
            lmem[j+2]*coef[2] + lmem[j+3]*coef[3];
        tmp.data = b;
        tmp.keep = -1;
        tmp.strb = -1;
        if (j == 96)
            tmp.last = 1;
        strm_out.write(tmp);

}}
```

Código 6-28 – Aplicação da diretiva de partição no cálculo de um filtro.

A síntese gera um circuito com duas memórias de duplo porto, em que cada uma contém uma parte do vetor. Sendo as memórias de duplo porto, é possível aceder a quatro valores em simultâneo, o que permite o cálculo do filtro num ciclo de relógio.

No entanto, uma análise mais cuidada ao padrão de acesso aos dados do vetor permite verificar que apenas é necessário ler um novo elemento do vetor a cada ciclo de relógio. Os outros três elementos foram considerados nos ciclos anteriores e podem ser reutilizados. Para isso, basta guardá-los temporariamente em três registos. Estes três registos deslocam os valores entre si a cada ciclo de relógio. Assim, é possível reduzir a solução a apenas uma memória e um registo de deslocamento. A arquitetura simplificada do registo de deslocamento juntamente com o cálculo do filtro é ilustrada na Figura 6-18.

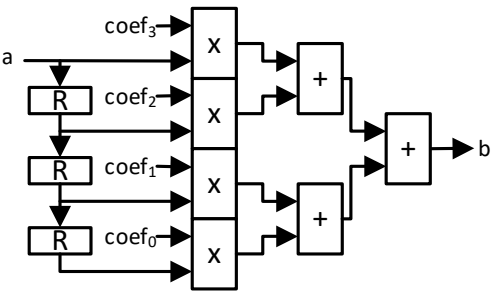


Figura 6-18 – Módulo de registo e de cálculo do filtro a uma dimensão.

Como se pode observar pela Figura 6-18, os registos transferem os dados entre si. Em cada ciclo de relógio, são lidos os três registos e um novo elemento do vetor que depois são enviados para os multiplicadores e para os registos em sequência. A descrição da solução pode ser feita de acordo com o Código 6-29.

```
#include "ap_axi_sdata.h"
#include "hls_stream.h"

typedef ap_axis<32, 0, 0, 0> trans_pkt;

void acumula(hls::stream<trans_pkt> &strm_in,
             hls::stream<trans_pkt> &strm_out){
#pragma HLS INTERFACE mode=axis port=strm_in
#pragma HLS INTERFACE mode=axis port=strm_out

trans_pkt tmp;

int i, j, acc, b;
const int coef[4] = {1, 2, 3, 4};
int r0, r1, r2;
static int lmem[100];

DADOS:for (i=0; i<100; i++) {
       strm_in.read(tmp);
       lmem[i] = tmp.data;
       if (tmp.last == 1)
              break;
}

r0 = lmem[0];
r1 = lmem[1];
r2 = lmem[2];

ACUMULA:for(j=3; j < 97; j++){
       b = r0*coef[0] + r1*coef[1] + r2*coef[2] + lmem[j]*coef[3];
       r0 = r1;
       r1 = r2;
       r2 = lmem[j];
       tmp.data = b;
       tmp.keep = -1;
       tmp.strb = -1;
       if (j == 96)
              tmp.last = 1;
       strm_out.write(tmp);
}}
```

Código 6-29 – Descrição do filtro a uma dimensão com um registo de deslocamento.

Inicialmente, os registos, r0, r1 e r2, são inicializados com os primeiros valores do vetor. A estrutura de repetição inicia-se com a leitura do quarto elemento seguido do cálculo.

O exemplo ilustra a vantagem em se analisar inicialmente o fluxo de dados e procurar descrever a função de modo a facilitar esse fluxo de dados. O projetista consegue, deste modo, orientar a ferramenta de síntese de modo a gerar uma solução melhor, ou seja, com utilização de menos recursos e com um melhor tempo de execução.

7 Tipos de Dados

O Vitis HLS suporta os tipos de dados nativos do C/C++, nomeadamente os inteiros (com e sem sinal) de 8, 16, 32 e 64 bits, e os reais de vírgula flutuante de 32 e 64 bits (*floats* e *doubles*). Todos estes tipos de dados nativos têm dimensões que são obrigatoriamente múltiplas de 8 bits e potências de 2. No entanto, os barramentos e os operadores em hardware podem ter dimensões arbitrárias e o projetista deve escolher, para cada operador, a dimensão mais adequada à aplicação específica a executar, tendo em consideração que a redução do número de bits ao mínimo necessário resulta em circuitos mais pequenos e mais rápidos.

O Vitis HLS permite definir tipos de dados de precisão arbitrária em C++, para representar inteiros e reais de vírgula fixa. Os inteiros de precisão arbitrária, `ap_int`, permitem representar inteiros com um número de bits específico, entre 1 e 1024 bits, e os reais de vírgula fixa, `ap_fixed`, permitem representar reais com um número pré-definido de bits para as partes inteira e fracionária.

Neste capítulo são introduzidos os vários tipos de dados disponibilizados pela ferramenta de síntese e são analisados os métodos de especificação e modelação utilizando tipos de dados de precisão arbitrária que permitem obter circuitos com barramentos e operadores mais pequenos e validar a sua funcionalidade por simulação em C/C++.

7.1 Inteiros Padrão

A síntese de alto nível suporta a definição dos tipos de inteiros padrão C/C++, `char` (8-bit), `short` (16-bit), `int` (32-bit), `long` (32-bit ou 64-bit, conforme o sistema operativo em que o HLS é executado) e `long long` (64-bit). Suporta também a definição dos inteiros de dimensão exata, `intN_t` (para N = 8, 16, 32 ou 64, conforme definido em `stdint.h`). Todos os tipos de inteiros podem ser definidos com sinal (*signed*) ou sem sinal (*unsigned*).

Somador de Inteiros		
Tipo e nº de bits dos Operandos	LUTs	Tempo de atraso (ns)
char (8-bit)	8	3.0
short (16-bit)	16	3.2
int (32-bit)	32	3.7
long long (64-bit)	64	4.6

Tabela 7-1 – Recursos ocupados e tempo de execução de somadores combinatórios (operandos inteiros).

A Tabela 7-1 apresenta os resultados obtidos com a síntese de somadores combinatórios com operandos inteiros e usando os tipos nativos do C/C++. Como esperado, os somadores são maiores e mais lentos quando os operandos têm um maior número de bits.

A utilização dos tipos nativos de C/C++ permite que o circuito gerado tenha exatamente o mesmo comportamento que a execução em *software*. No entanto, os operandos ficam limitados a ter um número de bits múltiplo de 8 e simultaneamente potência de 2. O resultado da operação aritmética com dois operandos do mesmo tipo tem também o mesmo tipo, o que implica, por exemplo, que o resultado de uma multiplicação de 2 inteiros de N-bit seja um inteiro também de N-bit (só são gerados os N bits menos significativos do resultado).

Os inteiros de precisão arbitrária, *ap_int*, (descritos na secção 9.3) permitem representar inteiros com um número de bits específico e obter diretamente operadores que geram todos os bits significativos do resultado. Por isso, são mais adequados para sintetizar componentes de hardware dedicados.

7.2 Reais Padrão – Vírgula Flutuante

O *Vitis HLS* suporta a definição dos tipos padrão C/C++ para representação de reais em vírgula flutuante, `float` (precisão simples, 32-bit) e `double` (precisão dupla, 64-bit). Suporta também a definição do tipo `half` (meia-precisão, 16-bit), definido na biblioteca *`hls_math`*. Estes tipos de dados são sintetizados de acordo com a norma IEEE-754, que define os números de bits do expoente e da fração utilizados na representação de vírgula flutuante (conforme indicado na Tabela 7-2).

Tipo	Nº bits		
	Palavra	Expoente	Fração
half	16	5	11
float	32	8	24
double	64	11	53

Tabela 7-2 - Tipos padrão de vírgula flutuante (norma IEEE-754).

Os operadores aritméticos são gerados utilizando a biblioteca matemática do Vitis HLS (`hls_math`), que permite sintetizar as funções matemáticas padrão do C++ (`cmath.h`) e

do C (`math.h`) em vírgula flutuante (*half*, float e double). Esta biblioteca matemática é detalhada na seção 7.5.

Somador de Reais		
Tipo e nº de bits dos Operandos	LUTs	Tempo de atraso (ns)
half (16-bit)	165	16
float (32-bit)	338	20
double (64-bit)	662	27

Tabela 7-3 - Recursos ocupados e tempo de execução de somadores combinatórios (operandos reais de vírgula flutuante).

A Tabela 7-3 apresenta os resultados obtidos de síntese de somadores combinatórios com operandos reais representados em vírgula flutuante. Os operadores de vírgula flutuante são bastante mais complexos que os operadores de inteiros e, portanto, consomem bastante mais recursos e são bastante mais lentos. Por exemplo, comparando as Tabelas 7-3 e 7-5, um somador de vírgula flutuante de 64-bit ocupa 10× mais recursos e é 6× mais lento que um somador de inteiros de 64-bit. Por isso, e sempre que os requisitos de precisão o permitam, é preferível realizar as operações com reais usando operadores de vírgula fixa (conforme descrito na secção 7.4), que têm uma eficiência próxima da dos operadores de inteiros.

É necessário cuidado quando se misturam tipos de vírgula flutuante diferentes porque poderá ser gerado um operador de maior precisão que o necessário e porque cada conversão entre tipos requer um componente hardware adicional.

```
#include <hls_math.h>

#define DOIS_PI 6.2831853

void circunferencia(float raio, float *outP) {
  *outP = raio * DOIS_PI;
}
```

Código 7-1 – Exemplo com multiplicação `double` *e conversores* `float` *de/para* `double`*.*

```
#include <hls_math.h>

#define DOIS_PI 6.2831853F

void circunferencia(float raio, float *outP) {
  *outP = raio * DOIS_PI;
}
```

Código 7-2 – Exemplo com multiplicação `float` *(sem conversores).*

Por exemplo, o Código 7-1 requer uma multiplicação em precisão dupla (porque a constante `DOIS_PI` está definida implicitamente como `double`) e, portanto, o circuito resultante

necessita de um conversor `float` para `double` (para o operando `float`), de um multiplicador-`double` e de um conversor `double` para `float` para o resultado. No Código 7-2, a constante `DOIS_PI` é definida como `float` e, como ambos os operandos são `float`, assim é inferido apenas um multiplicador de precisão simples e não são necessários conversores.

	LUT	DSP	FF
Circuito 3			
conversor float-para-double	67	-	-
multiplicador-double	71	11	149
conversor double-para float	44	-	-
Circuito 4			
multiplicador-float	63	3	32

Tabela 7-4 - Recursos ocupados pelos multiplicadores e pelos conversores de vírgula flutuante.

A Tabela 7-4 mostra o impacto da utilização dos conversores e do multiplicador de precisão dupla no consumo de recursos nas unidades de dados. Os 2 conversores ocupam apenas cerca de uma centena de LUT, mas o multiplicador de 64-bit necessita de mais do triplo dos blocos DSP consumidos pelo multiplicador de 32-bit.

7.3 Inteiros de precisão arbitrária

Os inteiros de precisão arbitrária, definidos na biblioteca *`ap_int`*, permitem que as variáveis e os operadores sejam especificados com qualquer número de bits no código C++. Por exemplo, 5-bit, 17-bit, 33-bit, até 1024 bits.

No Código 7-3, `ap_int<17>` representa um inteiro de 17 bits, com sinal, e `ap_uint<53>` representa um inteiro de 53 bits, sem sinal. O tamanho dos operadores é diretamente ajustado, de acordo com o tipo dos operandos: a multiplicação de 2 operandos de 17 bits especifica um multiplicador de 17 bits com uma saída de 34 bits e a soma de 2 operandos de 53 bits especifica um somador de 53 bits com uma saída de 54 bits (sinal de transporte e resultado. Recordar que a soma de dois operandos de N bits produz um resultado de N+1 bits). Em ambos os casos, é possível obter todos os bits necessários para representar corretamente o resultado.

Os tipos de dados de precisão arbitrária permitem, assim, gerar hardware mais eficiente: se os operandos têm 17 bits, define-se especificamente um multiplicador de 17 bits, portanto mais rápido e mais pequeno que o multiplicador de 32 bits que seria gerado se se usasse na especificação o tipo nativo *int*.

```
#include "ap_int.h"

ap_int<17> a, b;    // inteiros de 17 bits signed
ap_int<34> m;       // inteiro de 34 bits signed
ap_uint<53> x, y;   // inteiros de 53 bits unsigned
ap_uint<54> s;      // inteiro de 54 bits unsigned

m = a * b;          // multiplicador de 17 bits signed
s = x + y;          // somador de 53 bits unsigned
```

Código 7-3 – Inteiros de precisão arbitrária.

Os tipos de dados de precisão arbitrária permitem também fazer a simulação/análise C++ utilizando larguras de bits exatas e, por isso, validar a funcionalidade (e precisão) dos algoritmos antes da síntese.

O Código 7-4 especifica um produto vetorial utilizando um multiplicador-acumulador. Os tipos dos sinais estão pré-definidos no cabeçalho, o que torna a especificação mais clara e facilita, quando necessário, refinar as dimensões dos sinais para encontrar o tamanho mais adequado (avaliando a precisão geral do modelo). Neste exemplo, os elementos dos vetores são representados como inteiros de 8-bit e o multiplicador terá, portanto, uma saída de 16 bit. A acumulação final corresponde a 1024 somas de produtos e, por isso, o registo acumulador tem 26 bit para garantir o somatório de 1024 valores de 16 bit cada.

```
#include "ap_int.h"

typedef ap_int<8>  vec_t;
typedef ap_int<16> mul_t;
typedef ap_int<26> acum_t;
typedef ap_int<11> iter_t;
typedef ap_int<26> prodv_t;

#define VSIZE 1024

void vec_prod(vec_t a[VSIZE], vec_t b[VSIZE], prodv_t *prodv) {
  static acum_t reg_acum=0;

  for (iter_t i=0; i<VSIZE; i++) {
    mul_t mul = a[i] * b[i];
    if (i==0) reg_acum = mul;
    else reg_acum += mul;
  }

  *prodv = reg_acum;
}
```

Código 7-4 – Produto vetorial usando inteiros de precisão arbitrária.

O resultado da síntese da função de produto vetorial seria um circuito com operadores ajustados às dimensões dos operandos (ver Figura 7-1).

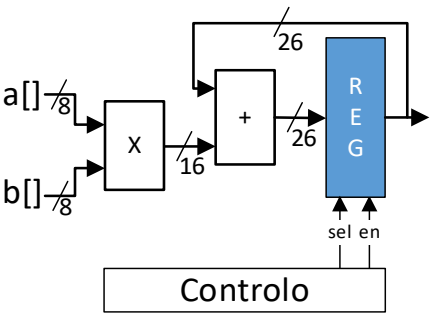


Figura 7-1 – Circuito sintetizado para o exemplo do produto vetorial usando inteiros de precisão arbitrária.

A HLS disponibiliza diversos métodos para realizar operações a nível dos bits de um valor representado como um inteiro de precisão arbitrária, `ap_int` (ou como um real de vírgula fixa, `ap_fixed`). O Código 7-5 exemplifica o método *`range`*, que permite aceder a um intervalo de bits e que também pode ser usado simplesmente através do operador `()`. O intervalo selecionado é definido pelos bits mais significativo e menos significativo (o bit menos significativo do valor fonte corresponde à posição 0, ou seja, a posição do bit mais à direita).

```
#include "ap_int.h"

ap_uint<10> Valor1 = 0x35F;
ap_uint<4> V0;
ap_uint<6> V1;
ap_uint<8> Valor2;

V0 = Valor1.range(3,0);  // resulta em V0 = 0xF
V1 = Valor1(9,4);        // resulta em V1 = 0x35

Valor2(3,0) = V1(3,0);
Valor2(7,4) = V0;        // resultam em Valor2 = 0xF5
```

Código 7-5 – Seleção de Bits por intervalo.

O operador *`range`* permite, por exemplo, empacotar/desempacotar vários sinais em/de um sinal comum, p.ex. um barramento. O Código 7-6 otimiza a especificação do produto vetorial (código 6) para utilizar barramentos de entrada e saída de 32-bit. Através do barramento de 32-bit, é possível aceder simultaneamente a 4 elementos de 8-bit, de cada vetor, e realizar 4 multiplicações em paralelo. O componente terá, portanto, um desempenho 4× superior (leitura dos valores de entrada 4× mais rápida e tempo de cálculo também 4× mais rápido).

```
#include "ap_int.h"

typedef ap_int<32> bus_t;
typedef ap_int<8> vec_t;
typedef ap_int<16> mul_t;
typedef ap_int<26> acum_t;
typedef ap_int<11> iter_t;
typedef ap_int<26> prodv_t;
typedef ap_int<18> soma_t;
#define VSIZE 1024

void vec_prod4(bus_t a[VSIZE/4], bus_t b[VSIZE/4], prodv_t *prodv) {
  static acum_t reg_acum=0;

  PRODVET:for (iter_t i=0; i<VSIZE/4; i++) {
        vec_t a0, a1, a2, a3;
        vec_t b0, b1, b2, b3;

        a0 = a[i].range(7,0);    b0 = b[i].range(7,0);
        a1 = a[i].range(15,8);   b1 = b[i].range(15,8);
        a2 = a[i].range(23,16);  b2 = b[i].range(23,16);
        a3 = a[i].range(31,24);  b3 = b[i].range(31,24);

        mul_t mul0 = a0 * b0;
        mul_t mul1 = a1 * b1;
        mul_t mul2 = a2 * b2;
        mul_t mul3 = a3 * b3;

       soma_t soma = mul0 + mul1 + mul2 + mul3;
        if (i==0) reg_acum = soma;
        else reg_acum += soma;
  }

  *prodv = reg_acum;
}
```

Código 7-6 – Produto vetorial com 4 elementos do vetor por acesso.

A Tabela 7-5 mostra as estimativas de ocupação de recursos do multiplicador vetorial, em que se nota a utilização de 4 blocos DSP para efetuar as 4 multiplicações em simultâneo. Indicam-se também a latência total, a latência por iteração, o intervalo de iniciação por ciclo (II) e o período máximo de relógio estimado. Note-se que, sendo o circuito implementado com o método de *pipeline*, a latência total é praticamente igual ao nº de iterações.

LUT	DSP	FF	Latência	Latência por iteração	Nº de iterações	II	Período Máximo de Relógio
117	4	70	261	5	256	1	5,2 ns

Tabela 7-5 - Recursos ocupados e estimativas temporais do multiplicador vetorial de inteiros de 8 bits.

7.4 Reais de Vírgula Fixa

Os tipos de vírgula fixa, *ap_fixed*, definidos na biblioteca *ap_fixed.h*, permitem representar números reais com um número arbitrário de bits, W, em que o número de bits dedicados à representação da parte inteira é definido por I, ficando o número de bits para representar a parte fracionária definido implicitamente como F=W-I. O uso de tipos de vírgula fixa garante o alinhamento correto das vírgulas dos operandos e do resultado, a extensão automática dos operandos à direita e/ou à esquerda, se necessário, e o ajuste do tamanho dos operadores.

No Código 7-7, `ap_ufixed<10,5>` representa um real de 10 bits, sem sinal, com 5 bits na parte inteira e 5 bits na parte fracionária, e `ap_fixed<12,11>` representa um real de 12 bits, com sinal, com 11 bits na parte inteira (incluindo o bit de sinal) e 1 bit na parte fracionária. A soma destes 2 operandos especifica um somador de 16 bits, com um resultado de 17 bits em que a parte inteira tem 12 bits (igual à parte inteira maior entre os operandos, 11, mais 1 para o transporte) e a parte fracionária tem 5 bits (igual à parte fracionária maior entre os operandos). É, assim, garantido o alinhamento da vírgula e a extensão dos operandos à esquerda e à direita, conforme indicado no exemplo.

```
#include "ap_fixed.h"

ap_ufixed<10,5> Var1 = 22.96875;     //  10110.11111
ap_fixed<12,11> Var2 = 512.5;        //  01000000000.1

ap_fixed<17,12> Res1 = Var1 + Var2;

//                          00000010110.11111
//                        + 01000000000.10000
// Resultado é 535.46875 : 001000010111.01111
```

Código 7-7 – Soma de reais de vírgula fixa.

Além da dimensão das partes inteiras e fracionárias, é possível definir como realizar a quantização e a gestão de resultados fora do domínio de representação (*overflows)*, quando for realizada conversão entre formatos. Estas operações são automaticamente concretizadas no circuito sintetizado.

```
ap_[u]fixed<int W, int I, ap_q_mode Q, ap_o_mode O>;

W - número de bits da palavra
I - número de bits da parte inteira
Q – modo de quantização (opcional)
O – modo de gestão de overflow (opcional)
```

O modo de quantização define como fazer o arredondamento quando é realizada uma conversão para um tipo com um número de bits fracionários menor. O Código 7-8

exemplifica os 2 principais modos de quantização: truncagem, *AP_TRN*, e arredondamento para +∞, *AP_RND*. Por omissão, é realizada a truncagem.

```
#include "ap_fixed.h"

ap_fixed<3, 2, AP_TRN> a4 = 1.25;  // 01.01 → 01.0
// Truncado para 1.0

ap_fixed<3, 2, AP_TRN> b4 = -1.25; // 10.11 → 10.1
// Truncado para -1.5

ap_fixed<3, 2, AP_RND> x4 = 1.25;  // 01.01 → 01.1
// Arredondado para 1.5

ap_fixed<3, 2, AP_RND> y4 = -1.25; // 10.11 → 11.0
// Arredondado para -1.0
```

Código 7-8 – Arredondamento ou truncagem de reais representados em vírgula fixa.

O modo de gestão de *overflow* define o que fazer quando é realizada uma conversão para um tipo com um número de bits inteiros menor. O Código 7-9 exemplifica os 2 principais modos de gestão de *overflow*: *AP_WRAP*, truncagem de todos os bits mais significativos fora do intervalo, e AP_SAT, saturação (em caso de *overflow* positivo, satura no valor máximo, em caso de *overflow* negativo, satura no valor mínimo). Por omissão, é realizada a truncagem, por ser mais simples de implementar em hardware.

```
#include "ap_fixed.h"

ap_fixed<5, 4, AP_RND, AP_WRAP> a4 = 19.0;
// 010011.0 → 0011.0 = 3, truncados os 2 bits à esquerda

ap_fixed<5, 4, AP_RND, AP_WRAP> b4 = -19.0;
// 101101.0 → 1101.0 = -3, truncados os 2 bits à esquerda

ap_fixed<5, 4, AP_RND, AP_SAT> x4 = 19.0;
// 010011.0 → 0111.1 = 7.5, saturado para o máximo representável

ap_fixed<5, 4, AP_RND, AP_SAT> y4 = -19.0;
// 101101 → 1000.0 = -8, saturado para o mínimo representável
```

Código 7-9 – Opções de overflow em reais de vírgula fixa.

A utilização da representação de vírgula fixa permite realizar as operações com aritmética inteira, portanto bastante mais eficientemente que usando aritmética de vírgula flutuante.

A Tabela 7-6 apresenta os resultados obtidos com a síntese de somadores combinatórios com operandos reais representados em vírgula fixa de 16, 32 e 64 bits. Estes resultados mostram que os somadores de vírgula fixa são aproximadamente 10× mais pequenos e 7× mais rápidos que os somadores de vírgula flutuante com operandos de dimensão similar.

Somador de Reais		
Tipo e nº de bits dos Operandos	LUTs	Tempo de atraso (ns)
ap_fixed<16,8>	17	2,4
ap_fixed<32,16>	33	2,8
ap_fixed<64,32>	65	3,8

Tabela 7-6 - Recursos ocupados e tempo de execução de somadores combinatórios (operandos reais de vírgula fixa).

O Código 7-10 especifica um produto vetorial em que os elementos dos vetores são números reais representados em vírgula fixa de 8 bits. A especificação do circuito é praticamente igual à do produto vetorial de inteiros de 8 bits (código 8), usando 4 multiplicadores em paralelo e barramentos de entrada/saída de 32 bits. Neste exemplo, os elementos dos vetores têm 2 bits de parte inteira, `IIN`, e 6 bits de parte fracionária, `FIN`. Os tipos dos resultados das operações aritméticas são definidos em função das dimensões dos elementos vetores. Por exemplo, os resultados das multiplicações têm dimensão igual à soma das dimensões dos operandos. Como o barramento de saída está definido como um inteiro de 32 bits, a escrita do resultado utiliza o operador *range* para fazer a atribuição direta do intervalo de bits correspondente ao valor de vírgula fixa.

```
#include "ap_int.h"
#include "ap_fixed.h"

#define IIN 2
#define FIN 6
#define WIN FIN+IIN
#define WOUT WIN+WIN+10
#define IOUT IIN+IIN+10

typedef ap_int<32> bus_t;
typedef ap_int<11> iter_t;

typedef ap_fixed<WIN,IIN> vec_t;
typedef ap_fixed<WIN+WIN,IIN+IIN> mul_t;
typedef ap_fixed<WIN+WIN+10,IIN+IIN+10> acum_t;
typedef ap_fixed<WIN+WIN+2,IIN+IIN+2> soma_t;
typedef ap_fixed<WOUT,IOUT> prodv_t;

#define VSIZE 1024

void vec_prod4(bus_t a[VSIZE/4], bus_t b[VSIZE/4], bus_t *prodv) {
  static acum_t reg_acum=0;

  for (iter_t i=0; i<VSIZE/4; i++) {
        vec_t a0, a1, a2, a3;
        vec_t b0, b1, b2, b3;
        a0(7,0) = a[i].range(7,0);    b0(7,0) = b[i].range(7,0);
        a1(7,0) = a[i].range(15,8);   b1(7,0) = b[i].range(15,8);
        a2(7,0) = a[i].range(23,16);  b2(7,0) = b[i].range(23,16);
```

```
        a3(7,0) = a[i].range(31,24);  b3(7,0) = b[i].range(31,24);

        mul_t mul0 = a0 * b0;
        mul_t mul1 = a1 * b1;
        mul_t mul2 = a2 * b2;
        mul_t mul3 = a3 * b3;

      soma_t soma = mul0 + mul1 + mul2 + mul3;
        if (i==0) reg_acum = soma;
        else reg_acum += soma;
  }

  bus_t aux;
  aux(WOUT-1,0) = reg_acum(WOUT-1,0);
  *prodv = aux;
}
```

Código 7-10 – Produto vetorial com elementos de vírgula fixa.

Neste exemplo, os tipos dos operandos dos somadores são iguais pelo que não é necessária extensão à direita ou à esquerda e, portanto, o circuito resultante é praticamente igual ao circuito para realizar o produto vetorial de inteiros. A Tabela 7-7 mostra as estimativas de ocupação de recursos do multiplicador vetorial que são, como esperado, exatamente iguais às do multiplicador vetorial de inteiros apresentados na Tabela 7-5.

LUT	DSP	FF	Latência	Latência por iteração	Nº de iterações	II	Período Máximo de Relógio
117	4	70	261	5	256	1	5,2 ns

Tabela 7-7 - Recursos ocupados e estimativas temporais pelo/do multiplicador vetorial de vírgula fixa.

Sempre que possível, os componentes *hardware* aritméticos em que os operandos sejam números reais devem ser concretizados usando vírgula fixa. No entanto, o intervalo de representação e a resolução proporcionados pela representação em vírgula fixa são inferiores aos disponibilizados pela representação em vírgula flutuante (para dimensões de palavra similares), pelo que em algumas aplicações específicas pode ser necessário recorrer a representações em vírgula flutuante para conseguir garantir a precisão necessária.

7.5 Biblioteca Aritmética hls_math

A biblioteca *`hls_math.h`* suporta a síntese e simulação de funções matemáticas das bibliotecas C++ (*`cmath`*) em vírgula flutuante (precisão simples, dupla precisão e meia precisão) e em vírgula fixa. O conjunto de funções aritméticas disponível nesta biblioteca para síntese e modelação inclui funções diversas, p.ex. trigonométricas, exponenciais, logarítmicas, potências e raízes quadradas.

O Código 7-11 especifica um componente para realizar a função `seno(din × π)`, disponível na biblioteca como *`sinpi(din)`*. A função está definida para valores

representados em vírgula flutuante ou em vírgula fixa, e é diretamente sintetizada de acordo com o tipo do argumento (o resultado tem o mesmo tipo da entrada).

```
#include "hls_math.h"
#include "ap_fixed.h"

typedef ap_fixed<16,3> data_t;
// typedef ap_fixed<32,3> data_t;
// typedef half data_t;
// typedef float data_t;

void my_component(data_t din, data_t *dout)
{
	data_t res = hls::sinpi(din);
	*dout = res;
}
```

Código 7-11 – Especificação de componente para calcular o seno.

Neste exemplo, é possível comparar diretamente os resultados de implementação porque a biblioteca `hls_math` usa o mesmo algoritmo nos vários casos. A Tabela 7-8 apresenta os resultados obtidos com a síntese do componente para operandos reais representados em vírgula flutuante e fixa de 16 e 32 bits. Conforme esperado, os componentes de vírgula fixa são mais pequenos e mais rápidos que os componentes de vírgula flutuante com operandos da mesma dimensão.

Dimensão e Tipo dos Operandos		DSP	FF	LUTs	Latência (ciclos)	Erro máximo
16 *bits*	`half`	1	169	508	12	1.7e-3
	`ap_fixed<16,3>`	1	99	213	7	5.0e-4
32 *bits*	`float`	8	684	1271	28	2.2e-7
	`ap_fixed<32,3>`	9	262	606	9	7.7e-9

Tabela 7-8 - Recursos ocupados, desempenho e precisão do componente `sinpi` *com operandos reais de vírgula flutuante e fixa.*

A tabela mostra também a precisão (erro máximo) dos resultados obtidos com a simulação dos componentes para valores de entrada $din \in [-2; 2]$ (resultados desde seno(-2π) até seno(2π)). Neste exemplo específico, os componentes de vírgula fixa conseguem precisões melhores porque, para o intervalo de valores de entrada utilizado, permitem usar mais bits fracionários.

8 Projeto Modular com Fluxo de Dados

O desenvolvimento de hardware com síntese de alto nível não tem de reduzir-se à especificação da funcionalidade com apenas uma função. Se pensarmos num programa de software, este é, em geral, descrito com várias funções interligadas que executam segundo uma determinada sequência. Esta abordagem permite descrever os programas de forma modular, com o objetivo de subdividir o problema em funções ou troços de programa mais pequenos interligados ou não com dependências de dados e/ou de controlo. Além disso, podemos ter uma função que é reutilizada em outro projeto, reduzindo o esforço e o tempo de projeto. A síntese de alto nível também permite que uma determinada funcionalidade ou algoritmo a ser implementado em hardware possa ser descrito como uma interligação de várias funções integradas dentro de uma função de topo.

Uma outra analogia com o software, é a execução de vários processos em concorrência. Existe um processo de topo que inicia outros processos que podem executar em concorrência ou em paralelo, se o processador o permitir. Os resultados de cada um dos processos é depois enviado para o processo principal que agrega as saídas e termina a execução. Enquanto num processador estes processos são executados tipicamente em sequência, os processos implementados em hardware podem ser executados em paralelo. Neste caso, cada processo é descrito em HLS como uma função, cabendo à função de topo agregar as chamadas às diferentes funções e garantir a transmissão e recolha ordenada e sincronizada de dados entre as funções.

Uma descrição HLS otimizada com funções interligadas permite explorar o paralelismo para melhorar o desempenho das soluções.

Neste capítulo, abordam-se os mecanismos para a descrição de circuitos com a interligação de várias funções e as diretivas que permitem otimizar estas descrições. Descrevem-se ainda os mecanismos de descrição da comunicação e da sincronização entre módulos hardware.

8.1 O Paradigma de Fluxo de Dados

O paradigma de fluxo de dados considera um conjunto de entidades que produzem e consomem dados. O caso mais simples corresponde a ter uma entidade que produz dados e outra entidade que consome esses dados. O processo de controlo da comunicação entre o produtor e o consumidor de dados tem de garantir que não se perdem dados e que estes são enviados e recebidos na ordem certa. O método mais comum de interligação de duas entidades considera uma memória que armazena os dados enviados pelo produtor, que depois são lidos da memória pelo consumidor. Para além do armazenamento dos dados, é ainda necessário garantir um protocolo de controlo de acessos aos dados. Se a memória está vazia, o consumidor tem de esperar até que surjam dados; se estiver cheia, o produtor tem de esperar até que haja espaço na memória para receber novos dados.

Em hardware, a memória do canal de comunicação é implementada recorrendo a uma FIFO, a uma memória de acesso aleatório (RAM) a funcionar em modo alternado ou simplesmente a um registo. A estrutura FIFO garante não só o armazenamento, como também a sincronização automática entre o produtor e o consumidor, com recurso aos sinais de indicação de FIFO vazia e de FIFO cheia. A FIFO só pode ser utilizada se se pretender que a leitura dos dados seja feita pela mesma ordem com que foram escritos. Se tal não se verificar, a memória do canal tem de ser implementada com uma memória de acesso aleatório, com funcionamento em modo alternado.

A memória em modo alternado é implementada com duas ou mais memórias de acesso aleatório que são alternadamente escritas e lidas (memória em *ping-pong* - *PIPO*). Num momento, o produtor escreve numa memória RAM e o consumidor lê de outra. No momento seguinte, a leitura e a escrita fazem-se sobre as outras memórias (Figura 8-1).

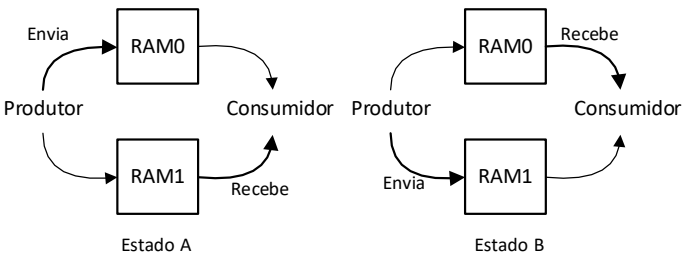


Figura 8-1 – Funcionamento da memória de canal com memórias RAM acedidas em modo alternado.

Como se pode observar na Figura 8-1, no estado A o produtor envia para a RAM0 e o consumidor lê da RAM1. No estado B, o produtor envia para a RAM1 e o consumidor lê da RAM0. O processo repete-se, com o sistema a alternar entre o estado A e o estado B.

A troca de memória só se dá quando ambos terminam a leitura ou a escrita das memórias respetivas. Deste modo, garante-se a sincronização do consumo e da produção de dados.

O método de memórias alternadas pode ser implementado com duas ou mais memórias RAM.

A vantagem deste tipo de memória de canal é que a leitura e a escrita podem ser feitas em modo aleatório, ao contrário da FIFO que apenas permite acesso sequencial. Por outro lado, necessita de pelos menos duas memórias a funcionar em modo alternado. Sempre que possível, deve ser utilizada uma memória FIFO, pois não só utiliza menos recursos de memória, como também garante a sincronização através dos sinais de estado da FIFO.

Uma característica importante em ambas as implementações do canal de comunicação é a de permitir sobrepor a comunicação com a execução em iterações diferentes do fluxo de dados, ou seja, o produtor pode começar a enviar dados na nova iteração, enquanto o consumidor acede aos dados da iteração anterior. Este funcionamento é implementado com ambos os mecanismos de memória do canal de comunicação. No caso da FIFO, o produtor pode continuar a enviar dados enquanto o consumidor acede aos dados pela ordem que foram escritos na FIFO. No mecanismo de memória alternada, enquanto o consumidor lê os dados escritos na iteração anterior, o produtor inicia o envio de novos dados para a memória seguinte em modo alternado.

A descrição da funcionalidade de um componente hardware de acordo com o modelo de fluxo de dados consiste em subdividir o problema em módulos independentes que trocam dados através de canais com memória. A comunicação dos dados e a sincronização entre os módulos são realizadas através dos canais de comunicação.

O desempenho do sistema como um todo depende do desempenho de cada um dos módulos de processamento. Produzir dados mais depressa do que podem ser consumidos ou desenvolver um circuito capaz de consumir dados mais depressa do que são produzidos leva a que o fluxo de dados tenha de ser suspenso, ou seja, os componentes mas rápidos do fluxo de dados têm períodos de suspensão da sua execução. A descrição de uma funcionalidade num paradigma de fluxo de dados deve procurar garantir uma execução balanceada de todas as entidades de processamento de modo a gerar a solução mais rápida com a menor quantidade de recursos possível. Isto é particularmente importante em aplicações de processamento em tempo real, como processamento de vídeo, em que as taxas de processamento têm de garantir o processamento em tempo real dos algoritmos e não mais do que isso.

A ferramenta de síntese de alto nível traduz uma função num módulo hardware com entradas e saídas. No caso de um circuito descrito por múltiplas tarefas em fluxo de dados, cada uma das tarefas é traduzida num módulo hardware e a passagem de dados entre tarefas é traduzida num canal de comunicação com uma determinada memória e um determinado mecanismo de sincronização. Os dados produzidos por uma tarefa podem ser consumidos por mais do que uma tarefa subsequente, e uma tarefa pode consumir dados de mais do que uma tarefa.

Consideremos, como exemplo, a descrição de um fluxo de dados com várias tarefas que trocam dados entre si (ver Código 8-1).

```
// Descrição dos circuitos em funções para utilizar num fluxo de
// dados descrito numa função de topo
//
void F1(int in[100], int out[100]){...}
void F2(int in[100], int out0[100], int out1[100]){...}
void F3(int in[100], int out[100]){...}
void F4(int in[100], int out[100]){...}
void F5(int in0[100], int in1[100], int out[100]){...}

// Descrição do fluxo de dados
//
void topF(int a[4][100], int b[4][100]){
      int d0[100], d1[100], d2[100], d3[100], d4[100];
      int i;

      for (i = 0; i < 4; i++){
            F1(a[i], d0);
            F2(d0, d1, d2);
            F3(d1, d3);
            F4(d2, d4);
            F5(d3, d4, b[i]);
      }
}
```

Código 8-1 – Exemplo de descrição de um circuito com interligação de várias funções segundo um paradigma de fluxo de dados

Em cada iteração da estrutura de repetição da função de topo (`topF`) existe uma sequência de chamadas a várias funções. No exemplo, consideraram-se cinco funções: `F1`, `F2`, `F3`, `F4` e `F5`. A saída de uma função é entrada de outra função atá à função F5 que gera os dados de saída da função de topo.

As funções podem ser executadas sequencialmente, respeitando as dependências de dados e de controlo (ver Figura 8-2).

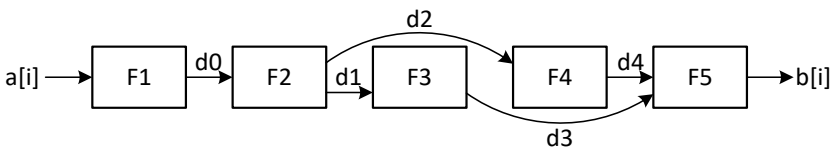


Figura 8-2 – Execução do fluxo de dados do exemplo do Código 8-1 em série.

Neste caso, o circuito de controlo apenas permite a execução de uma função de cada vez. Uma vez que a sequência se encontra dentro de uma estrutura de repetição, é executada múltiplas vezes. O tempo de execução total da função de topo, `topF`, é igual à soma dos tempos das execuções de cada uma das funções multiplicada pelo número de iterações da

estrutura `for`, assumindo que os dados de entrada estão sempre disponíveis, ou seja, que não existe nenhuma interrupção na execução da função inicial, `F1`.

Pela análise do fluxo de dados da Figura 8-2, concluímos que as funções `F3` e `F4` podem ser executadas em paralelo, uma vez que não existem dependências de dados entre elas (ver Figura 8-3).

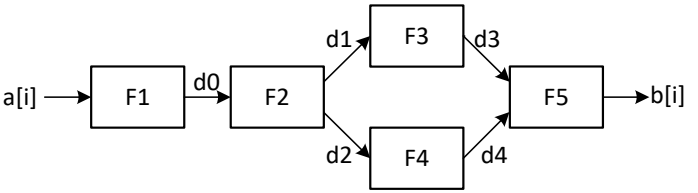


Figura 8-3 – Execução do fluxo de dados do exemplo do Código 8-1 permitindo a execução paralela de funções que não tenham dependências de dados.

Neste fluxo de dados, o tempo de execução total da função de topo, `topF`, é igual à soma dos tempos de execução das funções `F1`, `F2` e `F5` mais o máximo dos tempos de execução das funções em paralelo, `F3` e `F4`. Nesta solução, não adianta que uma das funções `F3` ou `F4` seja mais rápida que a outra, uma vez que a função `F5` só inicia após ambas terminarem. No caso de, por exemplo, `F3` executar mais rápido que `F4`, pode reduzir-se o tempo de execução de `F3`, poupando recursos, ou melhorando o tempo de execução de `F4`, se possível, com mais recursos.

A função de topo, mais uma vez, tem um tempo de execução igual ao número de iterações do `for` multiplicado pelo tempo de execução da sequência de funções. Este tempo é melhor que o anterior, pois as funções F3 e F4 executam em paralelo, mas ainda pode ser melhorado com a utilização de *pipeline* ao nível do fluxo de dados.

Considerando que os módulos que implementam cada uma das funções são partilhados na execução das várias iterações e que não existe dependências de dados entre iterações diferentes, pode iniciar-se uma nova iteração sem que a anterior tenha terminado com recurso à técnica de *pipeline* (ver Figura 8-4).

Com uma implementação em *pipeline*, o tempo de execução das quatro iterações é determinado pelas dependências de dados e de controlo. Qualquer função pode iniciar a execução a partir do momento em que os dados de entrada estão disponíveis e desde que o módulo que implementa a função não esteja a ser utilizado numa iteração anterior. O tempo de execução total diminui porque uma iteração inicia sem que a anterior tenha terminado.

Nesta configuração, é necessário algum cuidado com as memórias de comunicação. Como se pode observar pela figura, a função `F1`, por exemplo, produz dados, `d0`, em todas as iterações (`T1`, `T2`, `T3`, etc.) que são guardados na memória de canal. Os dados produzidos

por `F1` numa determinada iteração são consumidos por F2 na iteração seguinte. Ao mesmo tempo, `F1` já está a produzir novos dados. Se os dados forem consumidos ao mesmo ritmo a que são produzidos, não há problema de a memória de canal entre `F1` e `F2` encher, mas se forem produzidos mais rápido do que são consumidos, é necessário aumentar o tamanho da memória de canal.

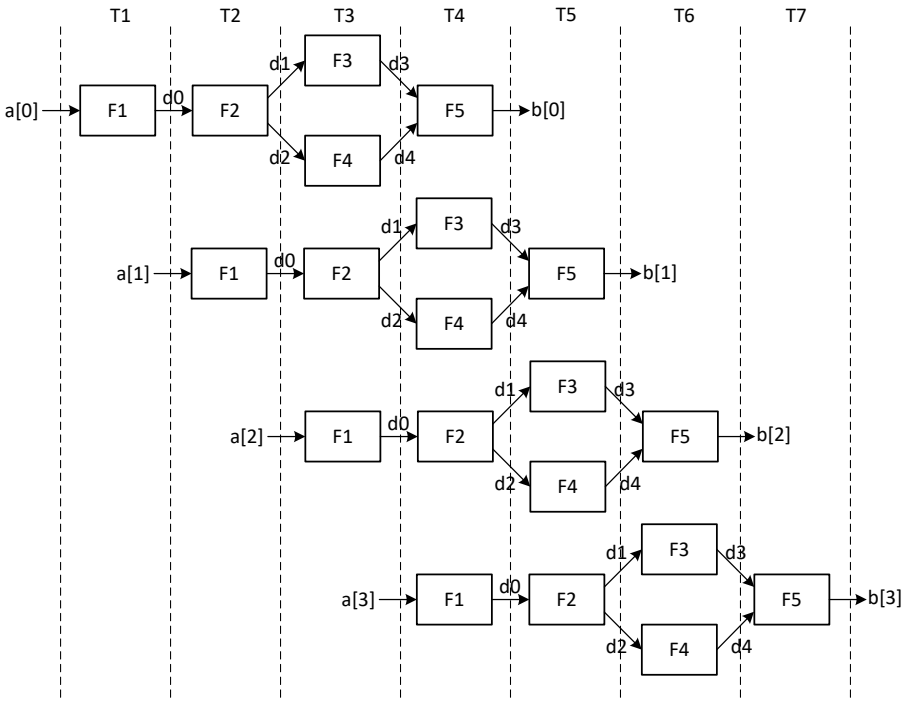


Figura 8-4 – Execução do fluxo de dados do exemplo do Código 8-1 com pipeline ao nível dos componentes que implementam cada uma das funções.

O tempo de execução pode continuar a ser melhorado com a utilização de *pipeline* dentro dos componentes e com inicialização da sua execução à medida que os dados estão disponíveis. Isto significa que uma segunda execução do mesmo componente pode ter início antes de a execução anterior terminar. Por outro lado, as funções também podem iniciar assim que houver dados disponíveis na(s) memórias de entrada, ou seja, podemos ter sobreposição entre funções.

Como foi apresentado nos capítulos anteriores, uma outra forma de melhorar a latência das estruturas de repetição, para além da técnica de *pipeline*, consiste em desenrolá-las. Neste caso, ao desenrolar a estrutura teremos vários componentes da mesma função a executar em paralelo (ver Figura 8-5).

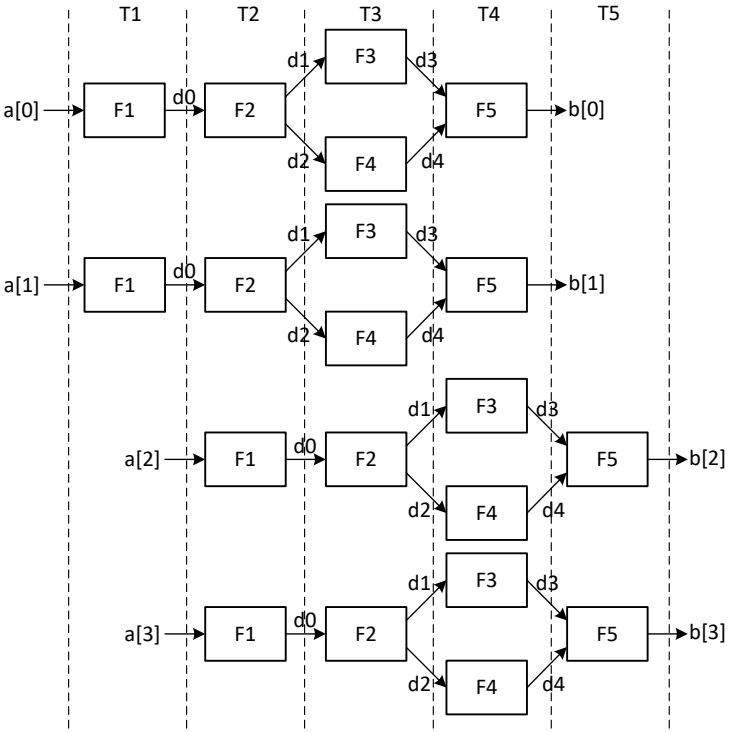


Figura 8-5 – Execução do fluxo de dados do exemplo do Código 8-1 com desenrolamento da estrutura `for` com um fator de 2.

No exemplo da Figura 8-5, a estrutura `for` foi desenrolada com um fator de 2. A solução implica, naturalmente, uma duplicação dos recursos, uma vez que a mesma função é executada duas vezes em paralelo. Não só são necessárias duas implementações de cada função, como também as memórias de comunicação.

O modelo de fluxo de dados permite organizar uma função em módulos de forma sequencial, paralela e/ou hierárquica, em que funções são chamadas dentro de outras funções. Os módulos são interligados com memórias para comunicação de dados e sincronização entre módulos. Nas secções seguintes, introduzem-se os métodos HLS de descrição e de otimização de uma função com o paradigma de fluxo de dados.

8.2 O Modelo Produtor-Consumidor em HLS

O paradigma de fluxo de dados baseia-se no modelo simples de produtor-consumidor que consiste numa função que produz dados e outra que consome esses dados. A utilização de uma memória entre as funções permite a comunicação em fluxo de dados com sobreposição, ou seja, os dados podem ser consumidos e produzidos em paralelo sem que uma tarefa tenha de esperar pelo fim da outra para dar início à sua execução.

Consideremos a descrição de duas funções em sequência que são chamadas dentro de uma função de topo, de acordo com o modelo produtor-consumidor (ver Código 8-2).

```
// Função de topo
void topF(int dataIn[100], int dataOut[100]);

// Protótipos das subfunções da função de topo
void fA(int fAin[100], int fAout[100]);
void fB(int fBin[100], int fBout[100]);

void topF(int dataIn[100], int dataOut[100]) {

// Diretiva para indicar à ferramenta de HLS que se pretende
// implementar as funções em fluxo de dados
#pragma HLS DATAFLOW

int buf1[100];

// Chamada das funções com comunicação dos dados buf1[]
fA(dataIn, buf1);
fB(buf1, dataOut);
}

//Função fA
void fA(int *in, int *out1)
{
      Loop0:for (int i = 0; i < 100; i++){
      // Diretiva que permite pipeline entre iterações diferentes
      #pragma HLS PIPELINE rewind

        int t = in[i] * 2;
        out1[i] = t;
      }
}

//Função fB
void fB(int *in, int *out)
{
      Loop0:for (int i = 0; i < 100; i++) {
      #pragma HLS PIPELINE rewind

        out[i] = in[i] + 1;
      }
}
```

Código 8-2 – Descrição em HLS de um circuito em fluxo de dados com uma função produtora de dados e outra consumidora.

A função `fA` consiste numa estrutura `for` que multiplica por dois os elementos de um vetor. A função `fB` incrementa uma unidade a todos os elementos. A função `topF` descreve a estrutura em fluxo de dados com a chamada sequencial de cada uma das funções, em que a saída da função `fA` é a entrada da função `fB`. A diretiva `HLS DATAFLOW`, que surge na função de topo, indica à ferramenta de HLS para gerar uma solução com *pipeline* das funções, ou seja, a função `fA` pode começar a executar sobre um novo vetor enquanto a

função `fB` executa sobre o vetor produzido anteriormente pela função `fA`. Com a especificação desta diretiva, o HLS cria automaticamente um canal de comunicação com uma memória entre as funções. O fluxo de dados pode ser visualizado através de um grafo de fluxo de dados (ver Figura 8-6).

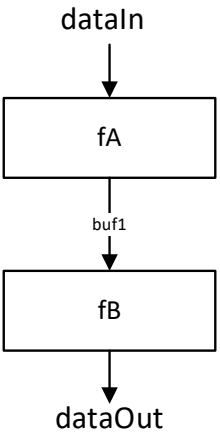


Figura 8-6 – Visualização do fluxo de dados do exemplo do Código 8-2.

O diagrama de fluxo de dados representa as funções com retângulos e as dependências de dados entre as funções com setas. O grafo de fluxo de dados não representa as dependências de controlo entre as diversas funções. As funções são sintetizadas em módulos hardware e as dependências de dados são mapeadas em canais de comunicação entre os módulos. Os canais de comunicação, como explicado anteriormente, são implementados com memórias para guardar temporariamente os dados transferidos entre os módulos.

De acordo com a descrição das funções, os dados são produzidos e consumidos pela mesmo ordem. Assim, a memória do canal de comunicação pode ser implementada com FIFO ou com memória RAM em modo alternado. Para o exemplo do produtor-consumidor, uma memória de canal implementada com uma memória alternada consiste em duas memórias RAM (ver Figura 8-7).

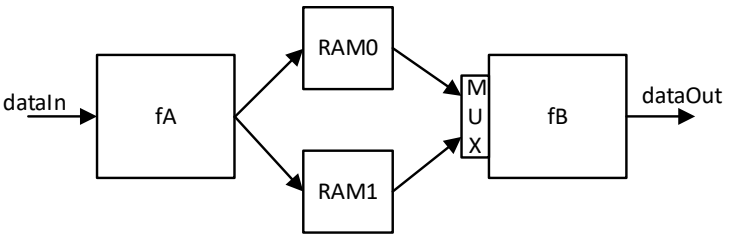


Figura 8-7 – Implementação da memória alternada no canal de comunicações.

Na primeira execução, a função `fA` executa e escreve o vetor de saída na memória RAM0. Considerando o funcionamento em *pipeline*, a função irá executar durante 101 ciclos de relógio (é necessário um ciclo de relógio para iniciar o *pipeline*), uma vez que executa um `for` de 100 iterações. Após concluir, a função `fB` inicia a leitura e o processamento dos dados escritos na RAM0, enquanto a função `fA` começa a execução de uma nova iteração com a escrita do vetor de saída na memória RAM1. A função `fB` irá igualmente executar em 101 ciclos de relógio. O processo repete-se com a escrita e a leitura das memórias RAM0 e RAM1 em modo alternado. Deste modo, ambas as funções estão sempre a executar, exceto na primeira execução da função `fA`, em que a função `fB` está à espera de dados, e na última execução da função `fB`, em que a função `fA` já não tem mais dados para processar.

Uma vez que as memórias BRAM da FPGA têm duplo porto, a implementação com duas memórias em modo alternado é feita apenas com uma memória de duplo porto.

```
// Função de topo
void topF(int dataIn[100], int dataOut[100]);

int main()
{
  int entrada[100];
  int saida[100];

  // Inicializa vetor de teste
  for (int i = 0; i < 100; i++){
    entrada[i] = i;
  }

  for (int i = 0; i < 4; i++){
      topF(entrada, saida);

      for (int i = 0; i < 10; i++)
        if (saida[i] != i*2+1)
          return 1;

   }

  return 0;
}
```

Código 8-3 – Ficheiro de teste do exemplo do Código 8-2.

A simulação da função de topo permite-nos verificar a execução do fluxo de dados em *pipeline*. Consideremos a co-simulação do exemplo através do ficheiro de teste dado no Código 8-3.

O ficheiro de simulação começa por inicializar um vetor (`entrada`) com 100 valores. Este vetor é de seguida aplicado à função de topo (`topF`) que gera um vetor de saída (`saida`). O vetor de saída é depois comparado com as mesmas operações calculadas em software. O processo repete-se quatro vezes, ou seja, a simulação executa a função quatro vezes e testa

a saída após cada chamada da função. O exemplo está de certa forma simplificado, pois o vetor de entrada permanece sempre igual.

A Figura 8-8 ilustra o escalonamento obtido na co-simulação do exemplo do produtor-consumidor considerando uma memória de canal com RAM em modo alternado.

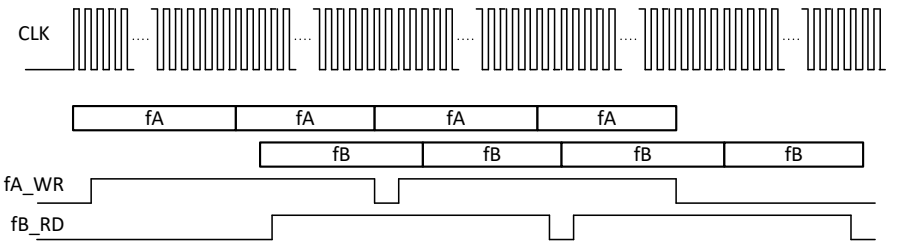


Figura 8-8 – Escalonamento das funções com uma memória de canal alternada.

Através da figura, é possível observar a execução do fluxo de dados em *pipeline*, como descrito anteriormente. Na primeira iteração, a função `fA` produz os primeiros dados e escreve na memória de canal (`fA_WR`). Enquanto isso, a função `fB` encontra-se à espera de que surjam os primeiros dados. Nas iterações seguintes, as funções executam em paralelo operando sobre os dados do canal de forma alternada de acordo com o funcionamento da memória de canal com RAM em modo alternado. No fim da última iteração, apenas executa a função `fB`.

A figura inclui também os sinais de escrita na memória do canal da função `fA` (`fA_WR`), o sinal de leitura do canal da `fB` (`fB_RD`) e o sinal de escrita na saída da função `fB` (`fB_WR`), Através da análise destes sinais confirma-se o desfasamento entre a leitura e a escrita da memória do canal, como descrito anteriormente. Verifica-se ainda uma pausa de 2 períodos do sinal de relógio (CLK) na escrita de dados da função `fA` a cada duas iterações. Este atraso deve-se ao facto de que após escrever na memória de canal RAM0 e RAM1 (modo alternado), a função `fA` tem de esperar que a função `fB` leia todo o conteúdo da RAM0 antes de iniciar uma nova iteração.

O intervalo de iteração é determinado pela função com maior tempo de execução. No exemplo, ambas as funções executam o mesmo número de ciclos. A latência da iteração é determinada pela soma das latências de ambas as funções. Devido aos dois ciclos de atraso a cada duas iterações, mais dois ciclos iniciais para a função `fA` começar a escrever dados, a latência total é dada por: 100 * 5 + 2 + 2*2 = 506 ciclos de relógio. Genericamente, para um determinado número de iterações, NI, a latência total é dada por 100 * 5 + 2 + 2*NI

A utilização de memória alternada para implementar a memória do canal entre as funções permite que os vetores possam ser consumidos com uma ordem diferente da que foram escritos pelo produtor de dados. A solução funciona para qualquer sequência de

escrita/leitura, mas implica a utilização de memórias a funcionarem em modo alternado. Esta solução pode ser simplificada, como já referido, nos casos em que a leitura da memória do canal é feita pela mesma ordem com que é escrita, com a utilização de uma memória tipo FIFO.

Em geral, as ferramentas de HLS conseguem determinar se é possível ou não utilizar uma FIFO pela análise da sequência de leitura e de escrita dos dados na memória. Se a sequência de leitura for a mesma da sequência de escrita então é possível utilizar uma FIFO. Caso contrário, é utilizada uma memória em modo alternado.

O projetista pode indicar qual o tipo de buffer através da diretiva `HLS STREAM`.

`#pragma HLS STREAM variable=<nome> type=<tipo> depth=<int>`

`variable=<nome>` - especifica o nome da variável de armazenamento

`type=<tipo>` – especifica o tipo de *buffer*: *FIFO*, *PIPO*, *shared* (sincronizado), *unsync* (não sincronizado):

FIFO – *buffer* tipo FIFO com tamanho *depth*

PIPO – *buffer* em modo alternado. O número de memórias é definido por *depth*

shared – similar ao PIPO, mas sem duplicação do vetor

unsync – similar ao PIPO, mas sem sincronização

`depth=<int>` – Profundidade da FIFO ou número de bancos do PIPO

O tipo de memória especifica a implementação a considerar para a memória do canal. Quando se indica a memória tipo FIFO, a ferramenta de HLS verifica se a utilização de uma FIFO garante a correta execução da função, ou seja, se os dados podem ser lidos sequencialmente. Se determinar que não é possível, então não sintetiza a memória e gera um erro. Em determinadas especificações, a ferramenta não consegue determinar a sequência de leitura dos dados. Nestes casos, é gerado apenas um aviso e a FIFO é sintetizada. A profundidade da FIFO é, por omissão, o tamanho do vetor a transmitir. Geralmente, é possível reduzir este tamanho sem alterar o desempenho do fluxo de dados, reduzindo assim os recursos hardware necessários para implementar a FIFO. A diretiva `HLS STREAM` permite que o projetista indique o tamanho da FIFO através do parâmetro `depth`. O mesmo parâmetro é utilizado para especificar o número de bancos de memória no caso de implementação da memória do canal com RAM alternada.

Para ilustrar a utilização de FIFO como memória de canal, consideremos o exemplo anterior, mas implementado com FIFO (ver Código 8-4).

```
(...)
void topF(int dataIn[100], int dataOut[100]) {
#pragma HLS DATAFLOW

int buf1[100];
// Especificação da memória do canal como sendo do tipo fifo com uma
// profundidade de 1
#pragma HLS STREAM variable=buf1 type=fifo depth=1

fA(dataIn, buf1);
fB(buf1, dataOut);
}

// A descrição das funções não depende do tipo de memória.
void fA(int *in, int *out1){
      (...)
}

void fB(data_t *in, data_t *out){
      (...)
}
```

Código 8-4 – Descrição em HLS de um circuito em fluxo de dados com uma função produtora de dados e outra consumidora em que a memória do canal de comunicação é implementada com FIFO.

No exemplo, considerou-se uma memória de canal implementada com uma FIFO de profundidade 1. Quando se utiliza uma FIFO, o bloco consumidor pode começar a executar assim que aparece o primeiro elemento na FIFO, não precisando de esperar que todo o vetor seja escrito na memória. Uma vez que ambos os blocos funcionam em *pipeline* ao mesmo ritmo, a profundidade da FIFO pode reduzir-se a apenas um, podendo até ser completamente eliminada quando a sincronização está garantida e o valor é registado à saída da função produtora.

A co-simulação do exemplo com FIFO, utilizando o mesmo ficheiro de simulação, produz o escalonamento ilustrado na Figura 8-9

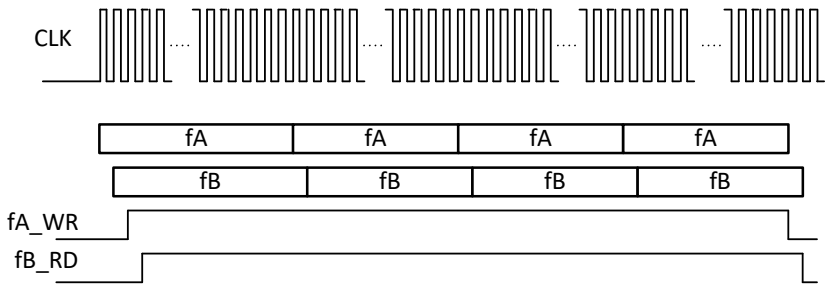


Figura 8-9 – Escalonamento das funções com uma memória FIFO.

Como se pode observar pelo escalonamento, a função `fB` inicia a execução assim que surge o primeiro valor na FIFO produzido e escrito pela função `fA`. A latência de iteração do ciclo é aproximadamente igual ao intervalo de iteração. A latência total da função de topo é igual a quatro vezes o intervalo de iteração, ou seja, 400 ciclos de relógio, mais 3 ciclos iniciais até a função `fB` começar a consumir os dados da FIFO. Assim, com a utilização de FIFO na memória do canal, a função de topo tem uma latência de 403 ciclos de relógio. Este valor contrasta com os 506 ciclos de relógio conseguidos com a memória de canal implementada com RAM em modo alternado. No entanto, relembre-se que a utilização da FIFO como memória de canal está condicionada à leitura sequencial dos dados.

Os exemplos de fluxo de dados anteriores descritos em HLS descrevem as tarefas como funções. No entanto, uma descrição de fluxo de dados não tem de ser feita necessariamente com a subdivisão do código em funções. O mesmo fluxo de dados pode ser obtido com a descrição do corpo das várias funções na mesma função de topo. Consideremos, como exemplo, a descrição de fluxo de dados do Código 8-2, em que o produtor e o consumidor são descritos como funções, mas desta vez considerando uma descrição sem recurso a sub-funções (ver Código 8-5).

```
// Função de topo
void topF(int dataIn[100], int dataOut[100]);
#pragma HLS DATAFLOW

int buf1[100];

// Corpo da função fA
      fA: for (int i = 0; i < 100; i++){
      #pragma HLS PIPELINE rewind
        buf1 = dataIn[i] * 2;
      }

// Corpo da função fB

      fB: for (int i = 0; i < 100; i++) {
      #pragma HLS PIPELINE rewind
        dataOut[i] = buf1[i] + 1;
      }
}
```

Código 8-5 – Descrição em HLS fluxo de dados do Código 8-2, mas sem uma estrutura hierárquica, ou seja, sem a subdivisão da descrição de topo em subfunções.

Ao indicar a diretiva de fluxo de dados (`HLS DATAFLOW`), a ferramenta de HLS associa cada uma das estruturas de repetição a uma tarefa e determina a dependência de dados entre as tarefas. Deste modo, gera uma implementação em fluxo de dados com duas tarefas, igual ao que foi gerado quando as tarefas foram descritas com subfunções. Esta descrição é mais compacta, mas não identifica explicitamente os módulos do fluxo de dados.

8.3 Fluxo de Dados com Múltiplas Funções

Um fluxo de dados pode ter múltiplas funções em sequência e/ou em paralelo. Quando as funções executam em paralelo ou quando os dados produzidos por uma função são consumidos por funções diferentes, é preciso algum cuidado na síntese das memórias de canal de modo a reduzir o impacto que estas têm sobre o tempo de execução do fluxo de dados.

Consideremos um exemplo de descrição de fluxo de dados com três funções em que os dados produzidos por uma das funções são consumidos por funções diferentes (ver Código 8-6).

```
#define N 100

// Funções do fluxo de dados
void fA(int in[N], int out1[N], int out2[N]);
void fB(int in[N], int out[N]);
void fC(int in1[N], int in2[N], int out[N]);

// Função de topo
void topF(int vecIn[N], int vecOut[N]){

// Declaração dos canais
int cAB[N], cAC[N], cBC[N];

// Descrição do fluxo de dados
#pragma HLS DATAFLOW
  fA(vecIn, cAB, cAC);
  fB(cAB, cBC);
  fC(cAC, cBC, vecOut);
}

// Função fA com uma entrada e duas saídas
void fA(int *in, int *out1, int *out2){

  Rep0:for (int i = 0; i < N; i++){
  #pragma HLS PIPELINE rewind

    int t1 = in[i] * 2;
    int t2 = in[i] * 4;
    out1[i] = t1;
    out2[i] = t2;
  }
}

void fB(int *in, int *out){

  Rep0:for (int i = 0; i < N; i++){
  #pragma HLS PIPELINE rewind
    out[i] = in[i] + 1;
  }
}

void fC(int *in1, int *in2, int *out){
```

```
  Rep0: for (int i = 0; i < N; i++){
  #pragma HLS PIPELINE rewind
    out[i] = in1[i] + in2[i];

  }
}
```

Código 8-6 – Descrição em HLS de um circuito em fluxo de dados com três funções.

No exemplo, a função `fA` gera dois vetores de saída (canais `cAB` e `cAC`). Um dos vetores é consumido pela função `fB` e o outro é consumido pela função `fC` juntamente com o vetor de saída da função `fB` (canal `cBC`).

A interligação entre as três funções, ou seja, o fluxo de dados, pode ser facilmente visualizada com um diagrama de fluxo de dados (ver Figura 8-10).

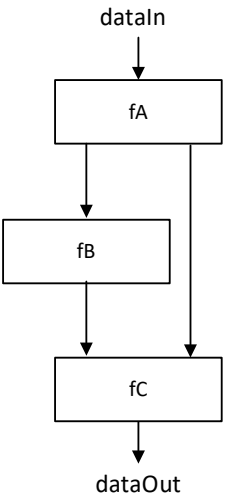


Figura 8-10 – Visualização do fluxo de dados do exemplo com três funções.

Os dois vetores produzidos pela função `fA` são consumidos pelas funções destino em momentos diferentes. Neste caso, como veremos, é preciso especial cuidado na síntese das memórias dos canais de comunicação para evitar tempos de espera.

Consideremos a síntese do fluxo de dados com memórias alternadas duplas em todos os canais de comunicação. Neste caso, o escalonamento obtido encontra-se ilustrado na *Figura 8-11*.

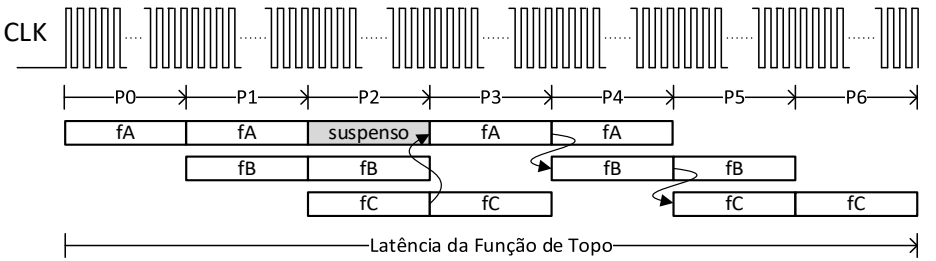


Figura 8-11 – Escalonamento das funções considerando que as memórias dos canais são implementadas com RAM em modo alternado.

No escalonamento da figura, as várias iterações do fluxo de dados estão identificadas por P0, P1, etc. Consideremos que o canal de comunicação entre funções `X` e `Y` é identificado por `cXY` e que cada canal tem duas memórias alternadas: RAM0 e RAM1. Em cada uma das iterações de execução da função de topo ocorre o seguinte:

- P0: a função `fA` escreve na RAM0 do canal c`AB` e na RAM0 do canal c`AC`;
- P1: a função `fA` escreve na RAM1 do canal c`AB` e na RAM1 do canal c`AC`. Ao mesmo tempo, `fB` consome os dados da RAM0 do canal c`AB`;
- P2: a função `fB` consome os dados da RAM1 do canal c`AB`. A função `fC` consome os dados da RAM0 do canal `cBC` e os dados da RAM0 do canal `cAC`.
  Neste momento, a função `fA` não pode iniciar a terceira iteração porque as RAM0 e RAM1 do canal `cAC` estão ocupadas, pois só nesta iteração é que a função `fC` começou a consumir os primeiros dados;
- P3: A função `fA` já pode iniciar a terceira iteração, mas a função `fB` tem de esperar pelo vetor gerado por `fA`;
- P4: A função `fB` já pode executar a terceira iteração, mas a função `fC` tem de esperar pelo vetor gerado por `fB`;
- O processo repete-se com as funções a suspenderem a execução a cada duas iterações do fluxo de dados.

Consegue-se evitar a suspensão da função `fA` utilizando três memórias alternadas na implementação da memória do canal entre as funções `fA` e a `fC`. Esta configuração é feita com a diretiva `HLS STREAM` através do parâmetro *`depth`* (ver Código 8-7).

No exemplo, o vetor enviado através do canal c`AC` entre as funções `fA` e a `fC` (`c2`) é configurado com uma profundidade de 3. Neste caso, a função `fA` não suspende durante a execução da primeira iteração da função `fC`, pois o canal tem uma terceira memória (ver escalonamento do fluxo de dados do Código 8-7 na Figura 8-12).

`(...)`

```
void topF(int vecIn[N], int vecOut[N]){

int c1[N], c2[N], c3[N];
// Especificação do tipo de memória de canal e o número de RAM
// a funcionar em modo alternado
#pragma HLS STREAM variable=c2 type=pipo depth=3

#pragma HLS DATAFLOW
  fA(vecIn, c1, c2);
  fB(c1, c3);
  fC(c2, c3, vecOut);
}

(...)

}
```

Código 8-7 – Configuração da memória do canal entre a função fA e a função fC com três memórias alternadas.

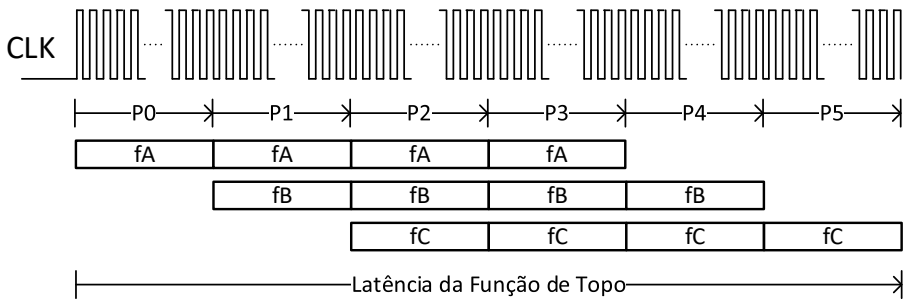


Figura 8-12 – Escalonamento do fluxo de dados do Código 8-7.

Através do escalonamento do fluxo de dados, verifica-se que as funções nunca suspendem. Assim, o fluxo de dados funciona sem interrupção, mas com um custo extra de memória no canal `cAC` entre as funções `fA` e `fC`.

No exemplo do fluxo de dados com três funções, considerado anteriormente, o consumo dos dados é feito pela mesma ordem com que são produzidos. Assim, as memórias dos canais podem ser implementadas com FIFO. Como vimos no exemplo do produtor-consumidor, a utilização de FIFO reduz o custo das memórias de canal e reduz a latência do circuito. Consideremos o mesmo exemplo do Código 8-6, mas com memórias tipo FIFO (ver Código 8-8).

Com esta nova especificação HLS, todos os canais de comunicação são implementados com FIFO. Para evitar que os processos suspendam, é necessário garantir que as FIFO não enchem.

```
#define N 100

(...)

void diamond(int vecIn[N], int vecOut[N]){

int c1[N], c2[N], c3[N];
//Especificação dos três canais com memórias tipo FIFO
#pragma HLS STREAM variable=c1 type=fifo
#pragma HLS STREAM variable=c2 type=fifo
#pragma HLS STREAM variable=c3 type=fifo

#pragma HLS DATAFLOW
  fA(vecIn, cAB, cAC);
  fB(cAB, cBC);
  fC(cAC, cBC, vecOut);
}

(...)
}
```

Código 8-8 – Descrição em HLS de um circuito em fluxo de dados com três funções utilizando memórias FIFO para implementar os canais de comunicação entre funções.

A diretiva `HLS STREAM` especifica, por omissão, uma FIFO com o tamanho dos vetores. Contudo, em geral, a profundidade da FIFO que garante a não suspensão da execução das funções é menor. Este valor pode ser determinado pelo projetista, através da análise das latências, ou pela ferramenta de síntese. A ferramenta de HLS determina, em geral, a profundidade da FIFO por simulação ou por análise. No exemplo do Código 8-8, o HLS determina uma profundidade de 3 para as FIFO dos canais `cAB` e `cBC` e 6 para a FIFO do canal `fAC`.

Repare-se que a soma das profundidades das FIFO dos canais `cAB` e `cBC` é igual à profundidade da FIFO do canal `fAC`. Esta relação está de acordo com a convergência dos dois caminhos de dados entre a função `fA` e a função `fC`. Sendo que a função `fB` consegue ler da FIFO de entrada, processar e escrever na FIFO de saída num ciclo de relógio, o balanceamento dos dois caminhos de dados em paralelo é garantido com a relação de profundidades de FIFO indicadas.

O balanceamento da latência entre funções é um aspeto bastante importante na implementação e otimização dos fluxos de dados. A propagação dos dados ao longo das diversas funções sofre atrasos diferentes se as funções tiverem latências diferentes ou se os caminhos percorridos pelos dados tiverem latências diferentes. Em determinado ponto é necessário fazer a convergência dos dados de modo a garantir que a implementação está de acordo com a especificação inicial em C/C++. Esta convergência é feita à custa de atrasos nos caminhos mais rápidos, de modo a garantir a mesma latência de dados que percorrem caminhos diferentes e que convergem numa determinada função.

O exemplo anterior é um caso em que é necessária a convergência dos vetores na função `fC`, uma vez que têm de ser consumidos ordenadamente em simultâneo. A diferença entre as profundidades das FIFO realiza este balanceamento e evita, assim, que as funções tenham de estar constantemente a suspender à espera de dados.

Existe um outro aspeto de implementação que mostra a vantagem na utilização de FIFO em vez de RAM em modo alternado. A memória dos canais com FIFO com profundidades pequenas pode ser implementada com registos, ao contrário da implementação com memórias RAM alternadas que teriam um custo elevado se fossem implementadas com registos. Além disso, caso se aumentasse o tamanho dos vetores, a implementação com memórias alternadas iria exigir mais memória, em proporção, enquanto a implementação com FIFO não se alteraria.

8.3.1 Violação do Modelo Produtor-Consumidor Único

Ao longo desta secção, qualquer comunicação entre duas tarefas está de acordo com o modelo de produtor-consumidor único. Este modelo determina que um conjunto de dados produzidos por uma tarefa só pode ser consumido por uma outra tarefa. Uma tarefa que produz dados para mais do que uma tarefa consumidora deve considerar uma saída para cada tarefa. Consideremos dois casos distintos de geração de dados para múltiplos consumidores (ver Figura 8-13).

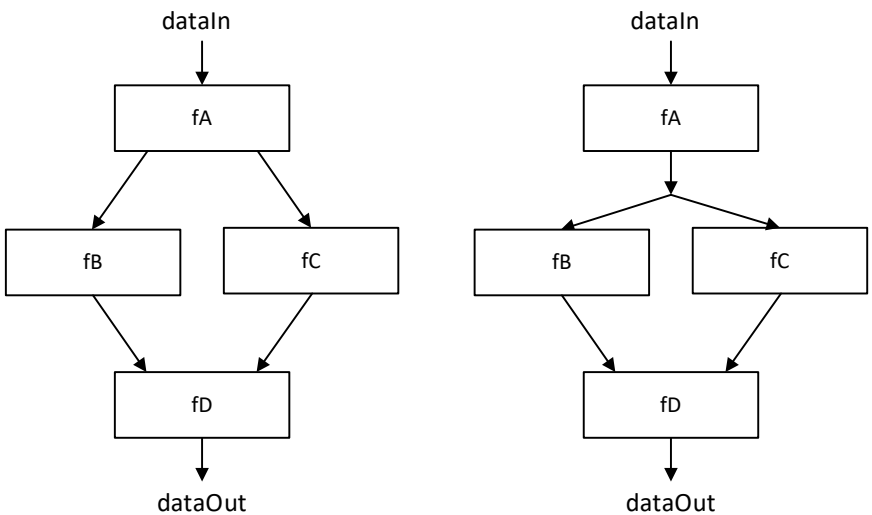


Figura 8-13 – Dois cenários de ligação de um produtor com múltiplos consumidores. O fluxo de dados da esquerda não respeita o modelo produtor-consumidor único, mas o da direita respeita.

No caso do fluxo da esquerda, o produtor gera duas saídas diferentes, cada uma com dados separados. Neste caso, verifica-se o modelo produtor-consumidor único pois as funções

consumidoras leem dados de canais diferentes. No fluxo da direita, é gerado apenas um conjunto de dados que é consumido igualmente por duas tarefas. Neste caso, não se verifica o modelo produtor-consumidor único, uma vez que os mesmos dados são lidos por dois consumidores diferentes. A dificuldade em considerar este modelo está no modo como se sincroniza a leitura dos dados. Por exemplo, se se tratasse de uma FIFO, ambas as funções destino teriam de ler o valor da FIFO ao mesmo tempo, obrigando a uma sincronização entre funções diferentes.

Nos casos em que não se verifica o modelo produtor-consumidor único, a ferramenta de síntese, em geral, gera um erro indicando que o modelo não está a ser cumprido. Consideremos o exemplo do Código 8-9 que não cumpre os requisitos de unicidade do modelo.

```
void dF(int in0[8], int in1[8], int out1[8], int out2[8]) {
#pragma HLS DATAFLOW

int aux[8];

  L1: for(int i = 0; i < 8; i++) {
      aux[i] = in0[i] - in1[i];
  }

  L2: for(int i = 0; i < 8; i++) {
      out1[i] = aux[i] * 2;
  }

  L3: for(int i = 0; i < 8; i++) {
      out2[i] = aux[i] * 4;
  }
}
```

Código 8-9 – Exemplo de uma descrição em fluxo de dados que não cumpre o modelo do produtor-consumidor.

No exemplo, são descritas três estruturas de repetição `for` que são sintetizadas como um fluxo de dados, ou seja, cada uma das estruturas é sintetizada como um módulo separado que comunica os dados entre si através de canais com memória. A variável auxiliar (`aux`) é produzida no ciclo `L1`, mas depois é consumida nos dois ciclos `L2` e `L3`, por dois módulos diferentes. A ferramenta de síntese gera um erro de síntese com a indicação de que não está a ser cumprido o modelo do produtor-consumidor único.

O código pode, no entanto, ser reescrito de modo a cumprir o modelo (ver Código 8-10).

A solução consiste em criar uma função (`separa`) que gera dois canais de comunicação a partir de uma entrada. A função recebe uma entrada e gera duas saídas com o mesmo conteúdo. Os dois módulos seguintes recebem cada um uma das saídas produzidas pelo módulo `separa`. Deste modo, todas as comunicações entre módulos são de produtor-consumidor único.

```
void separa(int in[8], int out1[8], int out2[8]) {

L1: for(int i=1; i<8; i++) {
  out1[i] = in[i];
  out2[i] = in[i];
  }
}

void diamond(int in0[8], int in1[8], int out1[8], int out2[8]) {
#pragma HLS DATAFLOW

int aux1[8], aux2[8], aux3[8];

  L1: for(int i = 0; i < 8; i++) {
    aux1[i] = in0[i] - in1[i];
  }

  separa(aux1, aux2, aux3);

  L2: for(int i = 0; i < 8; i++) {
    out1[i] = aux2[i] * 2;
  }

  L3: for(int i = 0; i < 8; i++) {
    out2[i] = aux3[i] * 4;
  }
}
```

Código 8-10 – Nova descrição em fluxo de dados do exemplo do Código 8-9 e que cumpre o modelo do produtor-consumidor único.

8.4 Fluxo de Dados com Bloqueamento

Uma especificação em fluxo de dados pode ter subjacente uma situação de bloqueio (*deadlock*) caso as FIFO não sejam bem dimensionadas. O bloqueio ocorre quando todas as funções do fluxo de dados se encontram suspensas apesar de haver dados de entrada da função de topo disponíveis e os dados de saída poderem ser consumidos.

Para exemplificar, consideremos o fluxo de dados apresentado na Figura 8-14.

No fluxo de dados representado na figura, os dados de saída da função `fA` percorrem dois caminhos diferentes: um caminho através da função `fB` e um outro através da função `fC`. Todas as funções têm *pipeline* com uma latência de 1, exceto a função `fA` com uma latência de 3. Um caminho começa em `fA` e passa por `fB` e o outro começa igualmente em `fA` e passa por `fC`. Admitindo que todas as FIFO dos canais de comunicação têm uma profundidade de um, a latência dos dois caminhos é diferente.

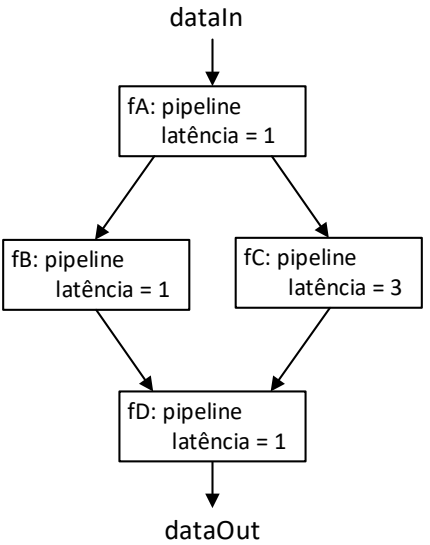


Figura 8-14 – Fluxo de dados com caminhos de dados não balanceados que provocam uma situação de bloqueamento.

Após os primeiros dois ciclos de relógio, a função `fB` produziu e escreveu dados no canal de saída com ligação à função `fD`. No entanto, a função `fD` só irá consumir estes dados quando a FIFO do canal `fCD` também tiver dados disponíveis. Neste momento surge um problema porque a função `fC` tem uma latência maior e os dados produzidos por esta função ainda não estão disponíveis. Como a função `fB` não consegue receber nem enviar dados enquanto a FIFO de saída não estiver vazia, a função `fA` também não pode enviar dados, nem para `fB`, nem para `fC`. Logo a função `fC` fica suspensa à espera de mais dados que façam avançar o *pipeline*. A execução do fluxo de dados encontra-se numa situação de bloqueio.

O problema pode ser resolvido com o balanceamento da latência entre os dois caminhos do fluxo de dados. Para tal, adiciona-se uma FIFO no caminho de menor latência com uma profundidade suficiente para equilibrar as latências dos dois caminhos paralelos. Outra solução consiste em permitir o vazamento (*flush*) do *pipeline* mesmo quando não existem dados à entrada, guardando-os localmente.

Aplicando a solução baseada na adição de atraso com uma FIFO, a implementação do fluxo de dados da Figura 8-14 incluiria uma FIFO de profundidade 3 (ver Figura 8-15).

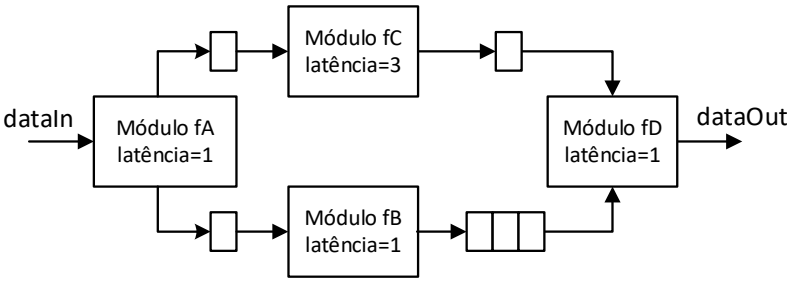


Figura 8-15 – Balanceamento de dois caminhos paralelos da implementação do fluxo de dados com uma FIFO.

Para determinar a profundidade das FIFO é necessário realizar o escalonamento do todo o circuito para obter a latência de cada uma das funções. Como base nesta informação, determina-se a profundidade das FIFO de modo a balancear todos os caminhos que convergem para uma determinada função.

O escalonamento do fluxo de dados balanceado está representado na Figura 8-16.

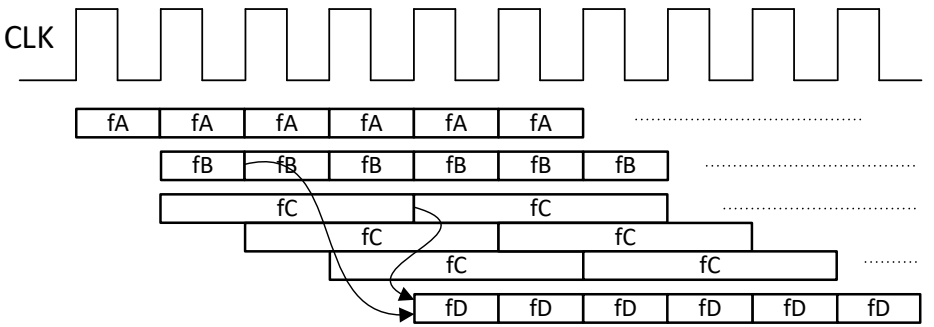


Figura 8-16 – Escalonamento do fluxo de dados da Figura 8-15 com caminhos de dados balanceados.

A função `fD` depende dos dados de `fB` e de `fC`. Como tal, só inicia a primeira execução após as primeiras execuções das funções `fB` e de `fC` terem gerado os primeiros dados. A FIFO à saída da função `fB`, com a profundidade adequada, permite que o módulo possa continuar a execução sem ter de se suspender à espera que `fC` termine.

8.5 Fluxo de Dados com Realimentação

A realimentação ocorre quando a saída de uma determinada função é consumida por uma função anterior no fluxo de dados. Em geral, não se recomenda a utilização de realimentação dentro de uma região de fluxo de dados. No entanto, o HLS consegue gerar soluções de fluxo de dados com realimentação, desde que descritas com os cuidados

necessários de modo a evitar situações de bloqueio. Em geral, o problema surge na tarefa que recebe uma entrada realimentada. Como os dados realimentados são gerados por uma tarefa posterior, no início da execução do fluxo de dados ainda não existe dados disponíveis no canal de realimentação. Logo, a tarefa que recebe este canal não pode iniciar a execução.

Para visualizar melhor o problema associado à realimentação descrito anteriormente, consideremos um exemplo de fluxo de dados com apenas dois módulos e com um caminho realimentado do segundo módulo para o primeiro (ver Figura 8-17).

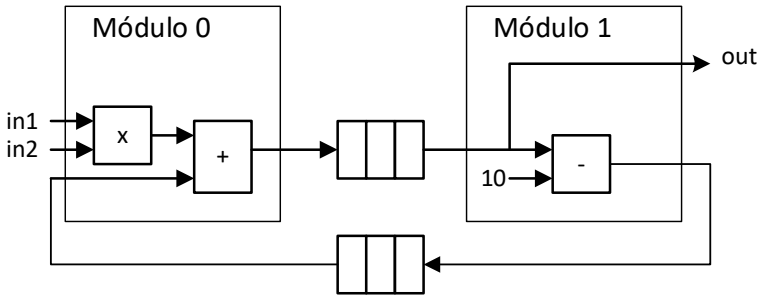


Figura 8-17 – Interligação de dois módulos com um modelo de fluxo de dados em que existe uma realimentação do módulo 1 para o módulo 0.

No exemplo da Figura 8-17, o módulo 0 necessita de dados das entradas `in1` e `in2`, e uma terceira entrada que provém do módulo 1. O problema é que o módulo 1 só produz dados após a execução do módulo 0 ter gerado e enviado um valor para a FIFO que se encontra no seu canal de saída. Contudo, o módulo 0 só produz dados quando tiver dados em todas as suas entradas, sendo que uma delas depende da execução do módulo 1. Nesta situação, o circuito completo não executa porque ambos os módulos se encontram suspensos à espera de dados e as FIFO permanecem vazias. Nestes casos, a ferramenta de HLS gera um aviso e pode não gerar a síntese do fluxo de dados.

Para a resolução deste problema, deveremos considerar acessos bloqueantes e acessos não-bloqueantes. Um acesso bloqueante a uma FIFO apenas fica resolvido quando se consegue ler um valor da FIFO do canal ou escrever na FIFO, dependendo do tipo de acesso, ou seja, uma tarefa que realiza um acesso bloqueante só avança quando o acesso ficar resolvido. No caso dos acessos não-bloqueantes, a tarefa não fica bloqueada numa leitura ou escrita.

8.5.1 Fluxo de Dados com Realimentação e Acessos Bloqueantes

A solução para resolver a situação de bloqueio existente num fluxo de dados com realimentação e acessos bloqueantes consiste em considerar um valor inicial na entrada de realimentação. O Código 8-11 descreve o fluxo de dados da Figura 8-17 com inicialização do valor de realimentação, assumindo acessos bloqueantes.

```
#include "ap_axi_sdata.h"
#include "hls_stream.h"

void modulo0(hls::stream<int> &in1, hls::stream<int>&in2,
             hls::stream<int>&inFeed, hls::stream<int>&out) {

  int fromM1, tmp;
  ap_uint<4> i;

  // variável que identifica o estado da execução
  ap_uint<1> conta = 1;

  for (i = 0; i < 10; i++){
    if (conta){
      fromM1 = 1; // Valor inicial da entrada
      conta--;
    }
    else
      fromM1 = inFeed.read(); //Valor realimentado

    tmp = in1.read() * in2.read() + fromM1;
    out.write(tmp);
  }
}

void modulo1(hls::stream<int> &in, hls::stream<int> &out,
             hls::stream<int> &outFeed) {

  int tmp, tmp1;
  ap_uint<4> i;

  for (i = 0; i < 10; i++){
  tmp = in.read();
  tmp1 = tmp - 10;

  outFeed.write(tmp1);
  out.write(tmp);
  }
}

void topF(hls::stream<int> &in1, hls::stream<int>&in2,
             hls::stream<int> &out){
#pragma HLS DATAFLOW
  hls::stream<int> forward, backward;

        modulo0(in1, in2, backward, forward);
        modulo1(forward, out, backward);
}
```

Código 8-11 – Descrição do fluxo de dados da Figura 8-17 com inicialização do valor de realimentação.

No exemplo, considerou-se uma variável (`conta`) no módulo 0 que determina quando é que o módulo deve começar a ler da entrada de realimentação. A variável é iniciada a 1, pelo que no primeiro ciclo é lida uma constante (`=1`) e nos ciclos seguintes é lido o valor da entrada de realimentação.

Nesta situação, existe a possibilidade de suspensão dos processos. Consideremos que ambos os módulos executam num ciclo de relógio. No primeiro ciclo de relógio, o módulo 0 gera o primeiro valor que é guardado na FIFO do canal de saída. No ciclo de relógio seguinte, o módulo 0 fica suspenso, uma vez que ainda não tem dados na entrada de realimentação. O módulo 1 executa e produz o primeiro valor que é guardado na FIFO do canal de realimentação. Com este valor, o módulo 0 já pode executar de novo, enquanto o módulo 1 tem de suspender pois a sua FIFO de entrada está vazia (ver escalonamento na Figura 8-18).

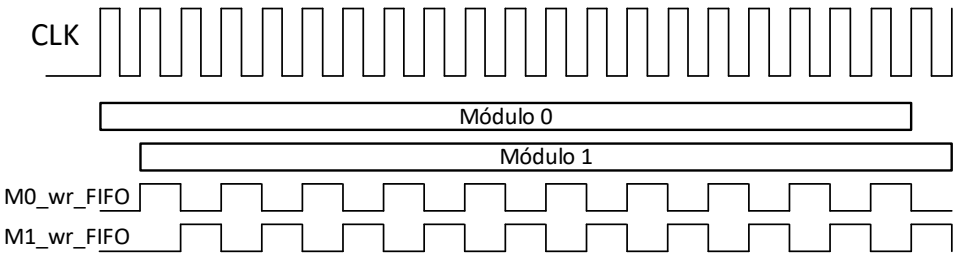


Figura 8-18 – Escalonamento do fluxo de dados com realimentação do Código 8-11 considerando apenas um valor inicial no canal de realimentação.

Através do escalonamento representado na figura, verifica-se o comportamento descrito, em que os módulos suspendem alternadamente durante um ciclo à espera de que o módulo que o antecede escreva na FIFO do canal. O tempo de suspensão agrava-se à medida que for aumentando o tempo de latência de geração dos valores de realimentação. Por exemplo, se a latência do fluxo de dados até gerar um valor no canal de realimentação fosse de 10, os módulos ficariam 10 períodos em suspensão até que surgisse um valor no canal de realimentação.

Este resultado é consequência de a latência total do fluxo de dados ser maior do que o número de valores iniciais da realimentação, definido no exemplo pela variável `conta`. Se igualarmos esta variável à latência total, então não teremos tempos de espera na execução dos módulos. No exemplo, assumindo uma latência de 1 em ambos os módulos, igualando a variável `conta` à latência total igual a 2, teremos uma solução sem tempos de espera (ver Figura 8-19).

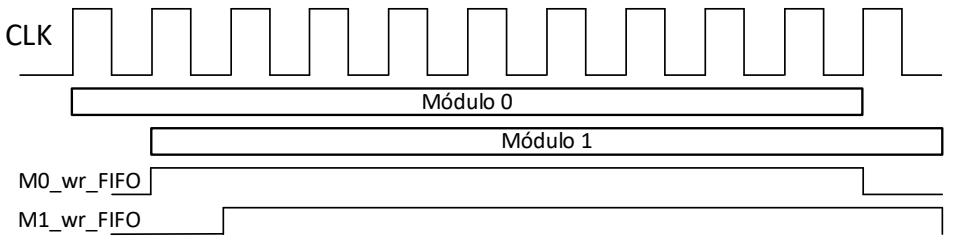


Figura 8-19 – Escalonamento do fluxo de dados com realimentação do Código 8-11 considerando um valor inicial no canal de realimentação igual à latência do fluxo de dados.

Através da representação do escalonamento é possível verificar que os módulos escrevem nas respetivas FIFO em todos os ciclos de relógio e que o número de ciclos de execução do fluxo de dados reduz a metade.

Genericamente, para garantir que não ocorrem tempos de suspensão, o valor da variável de contagem dos valores iniciais deverá ser igual à latência total do fluxo de dados.

8.5.2 Fluxo de Dados com Realimentação e Acessos Não-Bloqueantes

Um acesso não-bloqueante permite que uma tarefa continue a sua execução mesmo que seja tentada uma leitura de uma FIFO vazia ou uma escrita de uma FIFO cheia. Os métodos HLS de leitura e de escrita não-bloqueantes devolvem um valor que indica o sucesso do acesso, ou seja, verdadeiro (*true*) se houve sucesso no acesso e falso (*false*) se não foi possível realizar a operação.

O Código 8-12 modifica o Código 8-11 para considerar a utilização de métodos de acesso não bloqueantes.

```
#include "ap_axi_sdata.h"
#include "hls_stream.h"

void modulo0(hls::stream<int> &in1, hls::stream<int>&in2,
             hls::stream<int>&inFeed, hls::stream<int>&out) {

int fromM1, tmp;
ap_uint<4> i;

for (i = 0; i < 10; i++){
  if (inFeed.read_nb(fromM1) == 0)
      tmp = in1.read() * in2.read() + 1;
  else
      tmp = in1.read() * in2.read() + fromM1;

  out.write_nb(tmp);
}
}
```

```
void modulo1(hls::stream<int> &in, hls::stream<int> &out,
             hls::stream<int> &outFeed) {

int tmp, tmp1;
ap_uint<4> i;

for (i = 0; i < 10; i++){
  tmp = in.read();
  tmp1 = tmp - 10;

  outFeed.write_nb(tmp1);
  out.write_nb(tmp);
}
}

void topF(hls::stream<int> &in1, hls::stream<int>&in2,
             hls::stream<int> &out){
#pragma HLS DATAFLOW

hls::stream<int> forward, backward;

      modulo0(in1, in2, backward, forward);
      modulo1(forward, out, backward);
}
```

Código 8-12 – Descrição do fluxo de dados da Figura 8-17 com utilização de métodos de acesso não-bloqueantes.

O módulo 0 lê a FIFO do canal de realimentação e determina a operação a realizar em função do estado da FIFO. Caso não contenha dados, a função usa o valor 1 na entrada de realimentação. Caso haja dados, o valor é lido e considerado na operação do módulo. Em qualquer dos casos, o módulo nunca bloqueia à espera de dados na FIFO do canal de entrada.

Considerando ambos os tipos de acesso, não-bloqueante e bloqueante, cabe ao projetista determinar o que é necessário em cada caso e que a descrição HLS cumpre a funcionalidade desejada.

8.6 Limitações do Fluxo de Dados

A ferramenta de HLS otimiza a execução das tarefas em fluxo de dados com a utilização de *pipeline* e de memórias nos canais de comunicação. Realiza também o balanceamento dos caminhos em fluxos de dados com múltiplas funções e com caminhos de execução paralelos. Vimos nas secções anteriores algumas descrições em fluxo de dados que, embora sejam suportadas, devem obedecer a algumas regras de modo a não reduzir o desempenho do fluxo de dados e a não criar situações de bloqueio. Algumas das descrições que podem limitar a otimização do fluxo de dados são as seguintes:

- Leitura ou escrita de dados na interface do fluxo de dados a meio do fluxo – A leitura de dados da função de topo deve ser feita no início do fluxo de dados e as escritas devem ser feitas no fim do fluxo de dados. A leitura e/ou escrita de dados em

funções no meio do fluxo de dados leva a que os componentes hardware respetivos fiquem dependentes da disponibilidade das interfaces para a comunicação dos dados. Isto pode levar à necessidade de executar os componentes em série, com impacto sobre o desempenho do sistema;

- Comunicações entre tarefas que não cumprem o modelo de produtor/consumidor único, ou seja, os dados produzidos por uma tarefa são consumidos por mais de uma tarefa. Em geral, é possível reescrever o código de modo a cumprir o modelo. A solução consiste em criar cópias da saída do produtor, dando origem a múltiplos canais de comunicação a partir de uma entrada. Os módulos seguintes recebem separadamente um destes canais. Deste modo, todas as comunicações entre módulos são de produtor-consumidor único. O problema pode ser resolvido com o envio dos dados através de saídas independentes da tarefa produtora ou, caso não se queira alterar a tarefa, com a utilização de uma tarefa adicional que gera várias saídas com cópias da entrada;
- Caminhos do fluxo de dados não balanceados – um fluxo de dados com vários caminhos em paralelo com latências diferentes leva a períodos de suspensão das tarefas e, em alguns casos, pode gerar situações de bloqueio. O fluxo de execução pode ser otimizado através do balanceamento dos caminhos paralelos. O processo consiste em determinar o tamanho das memórias dos canais de comunicação de modo a compensar a diferença de latências entre os caminhos paralelos do fluxo de dados. Esta situação aplica-se igualmente a memórias de canal sequenciais ou alternadas de acesso aleatório;
- Fluxo de dados com realimentação – Apesar de não se recomendar caminhos de realimentação em fluxos de dados, é possível sintetizá-los desde que se sigam algumas recomendações para evitar situações de suspensão dos módulos ou até mesmo de bloqueio. Vimos na secção anterior que a utilização de valores iniciais no canal de realimentação evita que o módulo com entradas realimentadas do fluxo de dados fique suspenso à espera de dados no canal de realimentação. O número de valores iniciais a considerar depende da latência total do caminho direto entre o módulo realimentado e a tarefa que gera os dados de realimentação. Em alternativa, podem ser usados acessos não-bloqueantes que evitam que a tarefa suspenda a sua execução quando o estado da FIFO não permite acessos.

PARTE III

Exemplos de Aplicação da Síntese de Alto Nível

9 Exemplos

9.1 Multiplicação de Matriz por Vetor

A multiplicação de uma matriz por um vetor é uma operação linear que produz como resultado um vetor. Consideremos a multiplicação entre uma matriz A e um vetor x, em que o número de colunas de A é igual ao número de elementos do vetor x. Sendo A uma matriz m × n, o produto matricial Ax é igual a um vetor b com m elementos. Formalmente, o produto matricial é determinado do seguinte modo:

$$\mathrm{Ax} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} = \mathrm{b}$$

O produto matricial obtém-se, assim, através da multiplicação de vetores. Um elemento de b é determinado pelo produto de uma linha de A, um vetor, com o vetor x. O cálculo do produto de vetores é obtido através da acumulação do produto escalar entre os elementos dos vetores. Formalmente, o produto entre a linha $\mathrm{A_k}$ e o vetor x é dado por:

$$\mathrm{A_k} \times \mathrm{x} = \sum_{i=1}^{n} a_{ki} \times x_i \qquad (1)$$

Para o cálculo do produto de uma matriz com um vetor em hardware, vamos considerar a seguinte sequência de operações:

- Leitura do vetor x para memória interna;
- Leitura do vetor da primeira linha da matriz A:
  - Por cada elemento do vetor lido, multiplica pelo elemento de x com o mesmo índice, de acordo com a equação 1 e acumula;
  - Após completar o cálculo da equação 1, escreve o resultado, que corresponde a um elemento de b.
- Repetição do processo para as restantes linhas de A.

Na descrição do algoritmo em C/C++ (ver Código 9-1), consideramos que os dados são recebidos e enviados por interfaces *AXI4-Stream*, que o vetor x tem um tamanho máximo dado por `MEM_SIZE` e que os dados são representados com inteiros de 32 bits.

```
#include <ap_int.h>
#include <hls_stream.h>
#include <ap_axi_sdata.h>

#define MEM_SIZE 512

typedef hls::axis<ap_int<32>, 0, 0, 0> strmio_t;

void axis_gemv( hls::stream<strmio_t> &strm_in, hls::stream<strmio_t> &strm_out)
{
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=strm_in
#pragma HLS INTERFACE axis port=strm_out

strmio_t tmp, tmpa;
static ap_int<64> mult;
static ap_int<73> acc;
static ap_uint<9> last_velem;
static ap_int<32> localmem[MEM_SIZE];
int i;

LE_VETOR: for (i=0 ; i<MEM_SIZE; i++) {
       tmp = strm_in.read();
       localmem[i] = tmp.data;
       if (tmp.last == 1) break ;
      }
last_velem = i;

PROC_LINHAS: for (i = 0; ; ) {
#pragma HLS LOOP_TRIPCOUNT max=512*512

tmp = strm_in.read();
mult = tmp.data * localmem[i];

if (i == 0) acc = mult;
else acc += mult;

if (i == last_velem) {
       i = 0;
       tmpa.last = tmp.last;
       tmpa.data = (ap_int<32>)acc;
       tmpa.keep = 0xF;
       tmpa.strb = 0xF;
       strm_out.write(tmpa);
       if (tmp.last == 1) break ;
}
else
       i++;
}}
```

Código 9-1 – Função de cálculo do produto entre uma matriz e um vetor

O código começa por definir as variáveis que são mapeadas em memória:

- `mult`: guarda o valor do produto entre um elemento de uma linha da matriz `A` e um elemento de `x`. Sendo os elementos de 32 bits, a variável tem 64 bits para armazenar o produto;
- `acc`: acumula o valor dos produtos entre os elementos de uma linha da matriz `A` e um elemento de `x`. Sendo o produto de 64 bits, a variável de acumulação inclui bits extra suficientes para acumular até `MEM_SIZE` produtos. No exemplo, considera-se que `MEM_SIZE` = 512 (2^9), pelo que o acumulador tem 64 + 9 = 73 bits;
- `last_velem`: guarda o tamanho do vetor `x`. A implementação suporta vetores com qualquer tamanho até o máximo definido por `MEM_SIZE`. O valor desta variável é determinado após a leitura do vetor `x`;
- `local_mem`: guarda o vetor `x`. Define uma memória interna com `MEM_SIZE` posições de 32 bits para guardar o vetor `x`. A implementação suporta vetores com qualquer tamanho até o máximo definido por `MEM_SIZE`.

De seguida, inicia a estrutura de repetição (`LE_VETOR`) de leitura do vetor. Os elementos são lidos um a um pela interface `AXI4-Stream`. Após receber o último elemento (por deteção do sinal `last`), guarda o valor da variável de controlo da estrutura `FOR` (`i`) na variável `last_velem`, que corresponde ao tamanho do vetor lido.

Após a leitura e o armazenamento do vetor `x`, dá-se início ao passo de leitura da matriz `A` e de cálculo do produto de vetores (PROC_LINHAS). Em cada iteração do `FOR`, é lido um elemento de `A`, que é multiplicado pelo elemento respetivo do vetor `x` e acumulado. O primeiro produto é registado diretamente no acumulador, enquanto os restantes são somados ao valor presente no acumulador. Após ler o elemento de índice `last_velem`, que corresponde ao último elemento da linha `A`, envia o resultado do produto vetorial por `AXI4-Stream`, reinicia a variável de controlo da estrutura de repetição (`i`), e passa à próxima linha de `A`. O processo termina após receber o último elemento de `A`. Para saber quando termina, faz-se o teste do sinal `TLAST` do *AXI4-Stream* (`tmp.last == 1`).

A latência do circuito de multiplicação de matriz por vetor depende do tamanho do vetor `x`, `n`, do número de linhas da matriz `A`, `m`, e do número de elementos lidos da interface por ciclo de relógio. No exemplo, é lido um elemento de cada vez, pelo que a latência teórica, `latC`, do circuito é determinada do seguinte modo:

$$\texttt{latC} = n + n \times m$$

O primeiro termo diz respeito à leitura do vetor e o segundo diz respeito à leitura e multiplicação da matriz pelo vetor. Por exemplo, o cálculo do produto matricial entre uma matriz de tamanho 512 × 512 por um vetor de tamanho 512 demora aproximadamente 512 + 512^2 = ciclos de relógio. O tempo de execução do algoritmo aumenta quadraticamente com o tamanho do vetor.

9.1.1 Execução Paralela da Multiplicação de Matriz por Vetor

O algoritmo pode ser acelerado com a leitura de dados em paralelo. Consideremos agora a leitura de dois elementos de cada vez (ver Código 9-2).

```
#include <ap_int.h>
#include <hls_stream.h>
#include "ap_axi_sdata.h"

#define MEM_SIZE 512

typedef hls::axis<ap_int<64>, 0, 0, 0> strmio_t;

void axis_gemv2(hls::stream<strmio_t> &strm_in, hls::stream<strmio_t> &strm_out)
{
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=strm_in
#pragma HLS INTERFACE axis port=strm_out

strmio_t tmp, tmpa;
static ap_int<64> mult1, mult2;
static ap_int<73> acc;
static ap_uint<9> last_velem;
static ap_int<64> localmem[MEM_SIZE];
int i;

for (i=0 ; i<MEM_SIZE; i++) {
	tmp = strm_in.read();
	localmem[i] = tmp.data;
	if (tmp.last == 1) break ;
}
last_velem = i;

for (i = 0; ;) {
#pragma HLS LOOP_TRIPCOUNT max=512*512/2

tmp = strm_in.read();
mult1 = tmp.data.range(31,0) * localmem[i].range(31,0);
mult2 = tmp.data.range(63,32) * localmem[i].range(63,32);

if (i == 0) acc = mult1 + mult2;
else acc += mult1 + mult2;

if (i == last_velem) {
	i = 0;
	tmpa.last = tmp.last;
	tmpa.data = (ap_int<64>)acc;
	tmpa.keep = 0xF;    tmpa.strb = 0xF;
	strm_out.write(tmpa);
	if (tmp.last == 1) break ;}
else
	i++;
}}
```

Código 9-2 – Função HLS de cálculo do produto entre uma matriz e um vetor com leitura em paralelo de dois elementos

Para simplificar, assume-se que o número de elementos é par. Neste caso, são lidos e armazenados em memória interna dois elementos do vetor de cada vez. No cálculo do produto de vetores, são executadas duas multiplicações em paralelo. Deste modo, a latência reduz para aproximadamente metade da obtida com a primeira versão HLS do algoritmo. O método é naturalmente escalável à medida que se aumenta o número de elementos lidos em paralelo.

9.1.2 Resultados de Implementação

A Tabela 9-1 apresenta os recursos utilizados após síntese dos dois circuitos e as latências totais para um vetor de tamanho 512.

Versão	Recursos				Latência (ciclos)
	LUT	FF	BRAM	DSP	
1	235	259	1	4	262 667
2	527	552	2	8	131 596

Tabela 9-1 - Utilização de recursos e latência total das duas versões do circuito de implementação da multiplicação de uma matriz por um vetor com 512 elementos.

Como esperado, a latência reduz aproximadamente para metade na implementação paralela (versão 2). Em termos de recursos, como seria de prever, a versão 2 necessita de mais recursos. Em particular, tem mais quatro DSP, mais um bloco de memória RAM (BRAM) e mais LUT, pois precisa de mais um multiplicador de 32×32 bits, mais um somador e mais uma BRAM para ter acesso a 64 bits.

9.2 Multiplicação de Matrizes

A multiplicação de matrizes é uma operação que gera uma matriz a partir de duas matrizes. O produto matricial entre uma matriz `A` com dimensões `n` × `m` e uma matriz `B` com dimensões `m` × `p` resulta numa matriz `C` com dimensões `n` × `p` da seguinte forma:

$$\mathtt{AB} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \dots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mp} \end{bmatrix} = \begin{bmatrix} (ab)_{11} & (ab)_{12} & \dots & (ab)_{1p} \\ (ab)_{21} & (ab)_{22} & \dots & (ab)_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ (ab)_{n1} & (ab)_{n2} & \cdots & (ab)_{np} \end{bmatrix} = \mathtt{C}$$

em que

$$(ab)_{ij} = \sum_{k=1}^{m} a_{ik} \times b_{kj}$$

O produto matricial obtém-se através da multiplicação de vetores. Cada um dos elementos da matriz C é determinado como o produto entre uma linha de A e uma coluna de B. No total, são necessários $n \times p$ produtos de vetores, em que cada produto de vetor necessita de m^2 multiplicações-acumulações.

Uma linha de A ou uma coluna de B são reutilizadas várias vezes no cálculo do produto matricial. Uma linha de A tem de ser multiplicada por todas as colunas de B, assim como uma coluna de B tem de ser multiplicada por todas as linhas de A. Para evitar aceder pela interface à mesma linha ou à mesma coluna várias vezes, podemos guardar dados em memória interna e depois reutilizá-los.

Uma abordagem possível será ler e guardar em memória interna ambas as matrizes e depois ler uma ou mais linhas ou colunas simultaneamente de modo a explorar o paralelismo disponível. Esta solução está condicionada aos recursos de memória interna do dispositivo disponíveis para armazenar as matrizes. Por exemplo, considerando matrizes de dimensão 128×128, são necessárias 2×2^{14} posições de memória. Se aumentarmos a dimensão das matrizes para 512×512, serão necessárias 2×2^{18} posições de memória. Neste caso, assumindo dados a 32 bits, seriam necessários 2 MBytes de memória. Com esta solução, as matrizes são lidas apenas uma vez e são lidas várias linhas e várias colunas em paralelo expondo mais paralelismo. Contudo, requer bastante memória interna, que aumenta quadraticamente com o tamanho das matrizes.

Em alternativa, podemos armazenar apenas a matriz B ou algumas colunas da matriz B e depois ler linha a linha a matriz A. Por cada linha da matriz A lida, calcula-se o produto vetorial da linha A com todas as colunas da matriz B armazenadas internamente. Nesta solução, a matriz B é lida uma vez e a matriz A pode ter de ser lida várias vezes. Se a matriz B for guardada por completo, então a matriz A só é lida uma vez. Caso contrário, terá de ser lida por completo por cada conjunto de colunas da matriz B armazenadas. Por exemplo, se partirmos a matriz B em dois conjuntos de colunas, o primeiro conjunto é lido, armazenado e multiplicado pela matriz A. De seguida, é lido o segundo conjunto de colunas da matriz B e é novamente necessário ler a matriz A para ser multiplicada pelo segundo conjunto de colunas de B.

Como solução do produto de matrizes, vamos considerar a solução em que a matriz B é guardada por completo na memória interna. Após o armazenamento interno da matriz B, a matriz A é lida linha a linha. Cada uma das linhas é multiplicada em paralelo por todas as colunas de B. Com esta abordagem, ambas as matrizes são lidas apenas uma vez, mas só é possível ler uma linha de A de cada vez.

O cálculo do produto matricial em hardware é assim determinado com a seguinte sequência de operações:

- Leitura da matriz B para memória interna;

- Para cada linha da matriz `A`:
  - Leitura de um elemento, multiplicação pelos elementos respetivos de todas as colunas de `B` e acumulação de cada um dos produtos num acumulador distinto, ou seja, temos um acumulador por cada coluna de `B`;
  - Após terminar os produtos vetoriais entre as colunas de `B` e a linha da matriz `A`, envia-se pela interface de saída os resultados dos acumuladores (que correspondem a uma linha da matriz `C`) e prossegue-se para a próxima linha da matriz `A`.

Nesta implementação, o fator de paralelismo é igual ao número de colunas de `B`. De forma idêntica ao que foi assumido no produto entre matriz e vetor, vamos considerar que os dados são enviados por interfaces *AXI4-Stream*. Para facilitar a compreensão da implementação, considera-se que as matrizes têm um tamanho fixo e que os dados são representados como inteiros de 32 bits.

A descrição do algoritmo está dividida em três funções: uma função de leitura da matriz `B`, uma função de leitura de `A` e cálculo do produto vetorial com as colunas de `B` e uma função de escrita dos valores finais dos acumuladores.

O Código 9-3 define o tamanho das matrizes e o tipo de dados utilizado nas interfaces *AXI4-Stream* no ficheiro de cabeçalho `axis_gemm.h`.

```
#include <ap_int.h>
#include <hls_stream.h>
#include <ap_axi_sdata.h>

#define A_ROWS 20
#define A_COLS 64
#define B_COLS 16
#define B_ROWS A_COLS
#define C_ROWS A_ROWS
#define C_COLS B_COLS

typedef hls::axis<ap_int<32>, 0, 0, 0> strmio_t;
```

Código 9-3 – Definições utilizadas na função de cálculo do produto matricial (`axis_gemm.h`)

A função de leitura de `B` é descrita no Código 9-4.

```
void Read_MatB(hls::stream<strmio_t> &strm_in, ap_int<32> mem[B_ROWS][B_COLS])
{
	strmio_t tmp;
	LE_LINHA_B: for (int i=0 ; i<B_ROWS; i++) {
		LE_COLUNA_B: for (int j=0 ; j<B_COLS; j++) {
			tmp = strm_in.read();
			mem[i][j] = tmp.data;
		}
	}
}
```

Código 9-4 – Função leitura da matriz `B` utilizada no cálculo do produto matricial

A função lê os elementos da matriz B linha a linha através da interface *AXI4-Stream* e guarda-os em memória interna. A memória interna é definida com uma matriz de dimensões B_ROWS × B_COLS.

A função de leitura das linhas de A e o cálculo do produto vetorial de cada linha com as colunas de B é descrita no Código 9-5.

```
void Read_Process_Arow(hls::stream<strmio_t> &strm_in,
                       ap_int<32> mem[B_ROWS][B_COLS],
                       ap_int<32> acc[B_COLS])
{
#pragma HLS ARRAY_PARTITION dim=1 factor=16 type=cyclic variable=acc
#pragma HLS ARRAY_PARTITION dim=2 factor=16 type=cyclic variable=mem

strmio_t tmp;
ap_int<32> op;
ap_int<32> mult;

LE_A: for (ap_uint<7> j = 0; j < (A_COLS); j++) {
      tmp = strm_in.read();
      op = tmp.data;
      MULT_B: for (int p = 0; p < B_COLS; p++) {
            mult = op * mem[j][p];
            if (j == 0) acc[p] = mult;
            else acc[p] += mult;
      }
}}
```

Código 9-5 – Função de cálculo do produto vetorial de uma linha de A com as colunas de B

Para permitir aplicar *pipeline* ao circuito, a estrutura MULT_B é desenrolada por completo. Cada elemento lido da linha de A (LE_A), que é lida sequencialmente, é multiplicado por cada um dos elementos das colunas de B (MULT_B). À estrutura FOR exterior (LE_A) é aplicado *pipeline* (por omissão), pelo que a estrutura FOR interna (MULT_B) é desenrolada por completo. Isto significa que são executadas B_COLS multiplicações-acumulações em paralelo. Após percorrer todas as colunas de B, os valores dos acumuladores são enviados pela interface *AXI4-Stream* de acordo com o Código 9-6.

```
void Write_MatC(hls::stream<strmio_t> &strm_out, ap_int<32> acc[B_COLS], int i)
{
strmio_t tmpa;

ciclo_writeC: for (ap_uint<5> k = 0; k < B_COLS; k++) {
#pragma HLS PIPELINE
      tmpa.keep = 0xF;
      tmpa.strb = 0xF;
      tmpa.data = (ap_int<32>)acc[k];
      tmpa.last = (k == (B_COLS-1) && i == (A_ROWS -1)) ? (ap_uint<1>)1 :
(ap_uint<1>)0;
      strm_out.write(tmpa);
}}
```

Código 9-6 – Função de escrita dos elementos calculadas da matriz C

A função envia todos os valores de uma linha da matriz C, gerando o sinal de LAST no envio do último elemento da linha da matriz.

A descrição da função de topo da multiplicação de matrizes é descrita no Código 9-7.

```
#include "axis_gemm.h"

void Read_MatB(hls::stream<strmio_t> &strm_in, ap_int<32> mem[B_ROWS][B_COLS])
      {...}

void Read_Process_Arow(hls::stream<strmio_t> &strm_in,
                       ap_int<32> mem[B_ROWS][B_COLS], ap_int<32> acc[B_COLS])
      {...}

void Write_MatC(hls::stream<strmio_t> &strm_out, ap_int<32> acc[B_COLS], int i)
      {...}

void axis_gemm(
      hls::stream<strmio_t> &strm_in,
      hls::stream<strmio_t> &strm_out
      )
{
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=strm_in
#pragma HLS INTERFACE axis port=strm_out

ap_int<32> localmem[B_ROWS][B_COLS];

Read_MatB(strm_in, localmem);

CALCULA_C: for (int i = 0; i < (A_ROWS); i++) {
       ap_int<32> acc[B_COLS];
       #pragma HLS DATAFLOW
       Read_Process_Arow(strm_in, localmem, acc);
       Write_MatC(strm_out, acc, i);
}}
```

Código 9-7 – Função de cálculo do produto matricial

A função descreve a sequência de passos do algoritmo de multiplicação descrito anteriormente. Começa por ler e guardar em memória interna a matriz B. De seguida, inicia a estrutura de repetição de cálculo de uma linha da matriz C (CALCULA_C) em que iterativamente calcula uma linha da matriz C como o produto de uma linha da matriz A (Read_Process_Arow) com todas as colunas de B. Por fim, envia o vetor resultado da matriz C (Write_MatC). As duas funções do corpo do ciclo operam em fluxo de dados (pragma #HLS dataflow).

A latência teórica do circuito de multiplicação de matrizes, $GEMM_{lat}$, depende das dimensões das matrizes A ($n \times m$) e B ($m \times p$) sendo determinada do seguinte modo:

$$GEMM_{lat} = m \times p + n \times m$$

O primeiro fator corresponde à leitura e armazenamento da matriz `B` e o segundo termo corresponde ao cálculo dos produtos vetoriais. Na prática, teremos mais ciclos na execução da função devido ao intervalo de iniciação da estrutura de repetição interior da função `Read_Process_Arow`.

9.2.1 Execução Paralela da Multiplicação de Matrizes

De forma similar ao exemplo da multiplicação de uma matriz por um vetor, o algoritmo pode ser acelerado com a leitura de dados em paralelo. Na versão anterior, é lido um elemento de cada vez. Se considerarmos, por exemplo, a leitura de dois elementos de cada vez, a latência de leitura da matriz `B` e o cálculo dos produtos vetoriais reduz para aproximadamente metade.

9.2.2 Resultados de Implementação

A Tabela 9-2 apresenta os recursos utilizados após síntese dos dois circuitos: a versão 1 corresponde ao circuito com a leitura de apenas um elemento por cada acesso e a versão 2 corresponde ao circuito com leitura de dois elementos por cada acesso. As latências totais correspondem a matrizes com dimensões `A` (20 × 64) e `B` (64 × 16).

Versão	Recursos				Latência (ciclos)
	LUT	FF	BRAM	DSP	
1	2975	3169	16	48	2448
2	6902	4880	32	96	1296

Tabela 9-2 - Utilização de recursos e latência total das duas versões do circuito de implementação da multiplicação de matrizes: `A (20×64), B (64×16).`

A latência reduz aproximadamente para metade quando se considera a leitura de dois elementos por cada acesso à memória. Nesta implementação, o número de multiplicadores duplica, pelo que o número de DSP aumenta proporcionalmente.

A solução apresentada implica um fator de paralelismo igual ao número de colunas da matriz `B`. Uma matriz com 16 colunas necessita de 48 DSP. Ao passarmos para 1024 colunas, são necessários 3072 DSP. Para evitar este aumento proporcional de DSP, devem ser seguidas outras implementações da multiplicação de matrizes. Por exemplo, pode optar-se por separar a matriz `B` por grupos de colunas, como referido anteriormente, e multiplicar `A` por cada um dos grupos de colunas de cada vez. Neste caso, o paralelismo fica reduzido ao número de colunas de cada grupo, mas apenas necessita de um número de DSP proporcional ao número de colunas por grupo.

9.3 Histograma

Um histograma é utilizado para visualizar as frequências de ocorrência de dados por intervalos de valores, designados *bins*. Para exemplificar, consideremos o conjunto de

dados H = {10, 20, 25, 30, 33, 38, 40, 45, 50, 55, 56, 57, 60, 65, 66, 68, 69]. Pretende-se determinar o histograma do conjunto H em intervalos de 10. A distribuição dos dados pelos vários intervalos resulta no histograma da Figura 9-1.

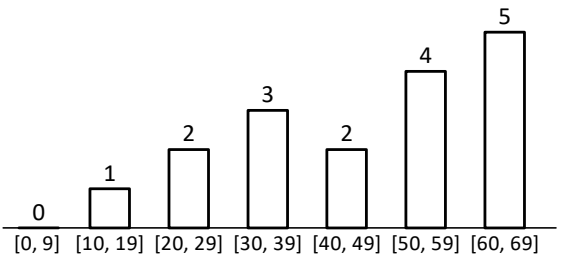


Figura 9-1 – Exemplo de um histograma com intervalos de 10 unidades

Cada um dos intervalos do histograma indica o número de elementos de H cujos valores se encontram nesse intervalo. Por exemplo, no intervalo [50, 59] existem quatro elementos do conjunto H, nomeadamente {50, 55, 56, 57].

As aplicações dos histogramas são variadas, como classificar um conjunto de pessoas por faixas etárias ou calcular a frequência de valores de pixels numa imagem em escala cinza.

Conceptualmente, o histograma obtém-se percorrendo os dados de entrada um a um e incrementando o *bin* respetivo. Consideremos o exemplo de um conjunto de dados com 16384 elementos que tomam valores entre [0,255]. Pretende-se determinar o histograma deste conjunto de dados assumindo intervalos unitários, ou seja, com 256 *bins*.

Para a descrição do circuito de geração deste histograma, vamos assumir que os dados são enviados por interfaces *AXI4-Stream*. Começou-se por definir as constantes e os tipos de dados a utilizar na função de descrição do circuito num ficheiro de cabeçalho (`axis_histogram.h`) de acordo com o Código 9-8.

```
#include <ap_int.h>
#include <hls_stream.h>
#include <ap_axi_sdata.h>

#define BIN_SIZE    256
#define DATA_SIZE   16384

typedef hls::axis<ap_uint<8>, 0, 0, 0> strmio8_t;
typedef hls::axis<ap_uint<32>, 0, 0, 0> strmio32_t;
```

Código 9-8 – Definições de constantes e de tipos de dados úteis à descrição da função de geração do histograma (`axis_histogram.h`)

Definiu-se o número de *bins* (`BIN_SIZE`) e o tamanho dos dados (`DATA_SIZE`) e dois tipos de interfaces *AXIS_Stream*, um com dados a 8 bits e outro com dados a 32 bits.

O circuito de geração do histograma pode ser descrito de acordo com o Código 9-9.

```
#include "axis_histogram.h"

void histogramAXIS1p(hls::stream<strmio8_t> &dataIn,
                     hls::stream<strmio32_t> &histogram){
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=dataIn
#pragma HLS INTERFACE axis port=histogram

ap_uint<32> hm[BIN_SIZE];
strmio32_t dados32;
strmio8_t dados8;

HIST_INICIO: for(int i = 0; i < BIN_SIZE; i++)
      hm[i] = 0;

HIST_PROC:for(int i = 0; i < DATA_SIZE; i++){
#pragma HLS PIPELINE
      dados8 = dataIn.read();
      hm[(unsigned int)dados8.data]++;
}

HIST_FIM: for(int i = 0; i < BIN_SIZE; i++){
#pragma HLS PIPELINE
      dados32.data = hm[i];
      dados32.last = (i == BIN_SIZE-1) ? 1 : 0;
      dados32.keep=0xF;
      dados32.strb=0xF;
      histogram.write(dados32);
}}
```

Código 9-9 – Função HLS de cálculo do histograma

O vetor onde será guardado o histograma (`hm`) é inicializado na primeira estrutura de repetição `FOR` (`HIST_INICIO`) com todos os elementos a 0, ou seja, com todos os *bin* a zero. Na segunda estrutura de repetição (`HIST_PROC`), percorrem-se os dados de entrada um a um e incrementa-se o *bin* respetivo do histograma. Na última estrutura `FOR` (`HIST_FIM`), envia-se o histograma pela interface *AXI4-Stream* de saída.

A Figura 9-2 apresenta o escalonamento dos vários ciclos, com maior detalhe no escalonamento do ciclo de processamento.

O ciclo de processamento executa com um intervalo de iteração de um período do sinal de relógio (CLK). A escrita de um valor e a leitura do próximo são executados no mesmo período de relógio. O número total de períodos de relógio do ciclo de processamento (latência total = 16386) é assim aproximadamente igual ao número de dados a processar (16384). A latência total do algoritmo é de 16909.

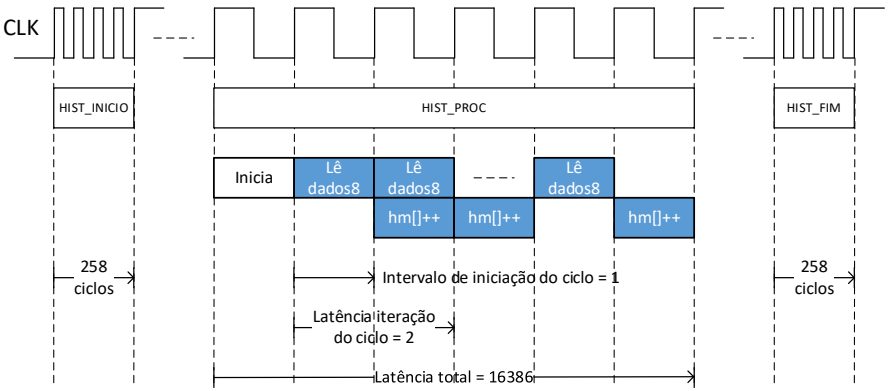


Figura 9-2 – Escalonamento da função de cálculo do histograma

9.3.1 Execução Paralela de Cálculo do Histograma

O algoritmo pode ser paralelizado com a separação dos dados de entrada em grupos que são processados separadamente e no final são acumulados entre si. Para isso, é necessário receber dados de entrada também em paralelo. A versão anterior do algoritmo considera a interface de entrada a receber apenas um elemento de 8 bits por cada acesso. A versão paralelizada descrita no Código 9-10 recebe 4 elementos empacotados numa interface de 32 bits e calcula 4 histogramas parciais em simultâneo.

```
#include "axis_histogram.h"

void histogramAXIS4p(hls::stream<strmio32_t> &dataIn,
                     hls::stream<strmio32_t> &histogram){
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=dataIn
#pragma HLS INTERFACE axis port=histogram

ap_uint<32> hm1[BIN_SIZE], hm2[BIN_SIZE];
ap_uint<32> hm3[BIN_SIZE], hm4[BIN_SIZE];
ap_uint<32> histAcc[BIN_SIZE];
strmio32_t dados32;

HIST_INICIO: for(int i = 0; i < BIN_SIZE; i++){
       hm1[i] = 0;
       hm2[i] = 0;
       hm3[i] = 0;
       hm4[i] = 0;
}

HIST_PROC:for(int i = 0; i < DATA_SIZE/4; i++) {
#pragma HLS PIPELINE

       dados32 = dataIn.read();

       hm1[(unsigned int)dados32.data.range(7,0)]++;
```

```
        hm2[(unsigned int)dados32.data.range(15,8)]++;
        hm3[(unsigned int)dados32.data.range(23,16)]++;
        hm4[(unsigned int)dados32.data.range(31,24)]++;
}

HIST_FIM: for(int i = 0; i < BIN_SIZE; i++){
#pragma HLS PIPELINE
        histAcc[i] = hm1[i] + hm2[i] + hm3[i] + hm4[i];
        dados32.data = histAcc[i];
        dados32.last = (i == HIST_SIZE-1) ? 1 : 0;
        dados32.keep=0xF;
        dados32.strb=0xF;
        histogram.write(dados32);
}
}
```

Código 9-10 – Função HLS de cálculo do histograma de um conjunto de dados

Após o processamento dos quatro blocos de dados em quatro histogramas parciais, o `FOR` final (`HIST_FIM`) acumula os *bins* respetivos dos quatro histogramas num único histograma e envia o valor pela interface de saída. A execução do algoritmo tem uma latência total de 4621 ciclos de relógio (ver Figura 9-3).

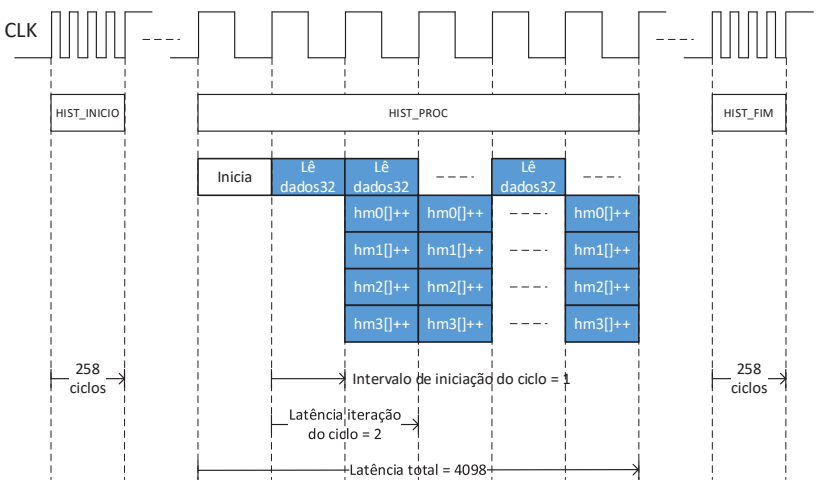


Figura 9-3 – Escalonamento da função de cálculo do histograma com um fator de paralelismo de 4.

O escalonamento evidencia o paralelismo de quatro com apenas uma leitura de dados (4 valores compactados em 32 bits) por ciclo de relógio. O fator de paralelização pode, naturalmente, ser aumentado, com o consequente aumento de recursos hardware necessários e limitado pelo número de bits de dados da interface.

9.3.2 Resultados de Implementação

A Tabela 9-3 apresenta os recursos utilizados após síntese dos circuitos e as latências totais.

Fator de Paralelismo	Recursos			Latência (ciclos)
	LUT	FF	BRAM	
1	236	198	2	16909
4	749	391	8	4621

Tabela 9-3 - Utilização de recursos e latência total do circuito de implementação do histograma com fatores de paralelização igual a um e a quatro para um conjunto de dados com 16384 elementos.

Em termos de latência, a solução é escalável, pois a latência é inversamente proporcional ao fator de paralelismo. O aumento dos recursos hardware é consequência de ser necessário armazenar mais vetores e realizar mais somas e incrementos.

9.4 Equalização com Histograma

Quando aplicado a uma imagem, o histograma determina a distribuição de tons da imagem. Considerando uma imagem em escala de cinzas, o histograma da imagem indica o número de pixels de cada um dos tons de cinza entre 0 e 255. Consideremos a imagem exemplo representada na Figura 9-4, juntamente com o seu histograma.

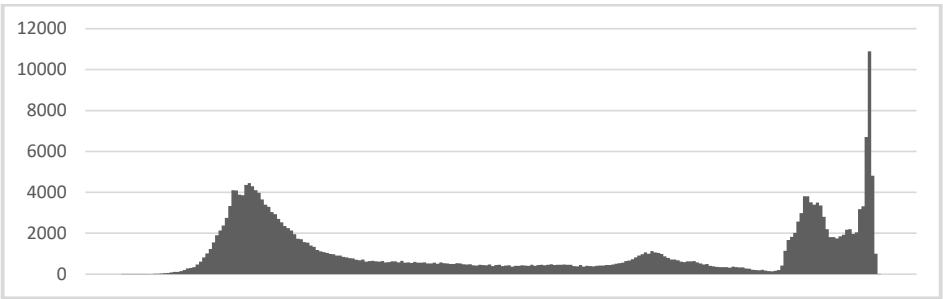


Figura 9-4 – Exemplo de uma imagem em escala de cinzas e o respetivo histograma

A equalização do histograma permite melhorar o contraste das imagens. A equalização da imagem espalha a distribuição dos pixels fazendo com que os pixels escuros pareçam mais escuros e os pixels claros pareçam mais claros.

Consideremos o número de níveis cinza da nova imagem, `L`, o número total de pixels da imagem, `A`, e o histograma da imagem, `H(x)`. A função de equalização, `feq(pixval)`, é a seguinte:

$$\text{feq (pixval)} = L \times \frac{1}{A} \times \sum_{i=0}^{pixel} H(i) \quad (1)$$

Considerando a função de equalização e a função HLS de obtenção do histograma descrita na secção anterior, chegamos à descrição do circuito de equalização com histograma.

O Código 9-11 define os tipos de dados e constantes utilizadas na função (`axis_histogramEq.h`).

```
#include <ap_int.h>
#include <hls_stream.h>
#include <ap_axi_sdata.h>

#define BIN_SIZE    256
#define IMAGEH      512
#define IMAGEV      512
#define AREA        IMAGEH * IMAGEV
#define GREYLEVELS  256
#define CONST       1 // GREYLEVELS / AREA * 1024

#define DATA_SIZE   IMAGEH * IMAGEV

typedef hls::axis<ap_uint<8>, 0, 0, 0> strmio8_t;
typedef hls::axis<ap_uint<32>, 0, 0, 0> strmio32_t;
```

Código 9-11 – Definições de constantes e de tipos de dados úteis à descrição da função de cálculo da equalização da imagem com base no seu histograma

Define-se o número de *bins* (`BIN_SIZE`) e o tamanho dos dados (`DATA_SIZE`) em função do tamanho da imagem nas dimensões horizontal (`IMAGEH`) e vertical (`IMAGEV`). Definem-

se ainda dois tipos de dados *AXI4_Stream*, um com dados a 8 bits e outro com dados a 32 bits. O algoritmo de equalização com histograma pode ser descrito de acordo com o Código 9-12.

```
#include "axis_histogramEq.h"

void histogramEqAXIS1p(hls::stream<strmio8_t> &imageIn,
                       hls::stream<strmio8_t> &imageOut){

#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=imageIn
#pragma HLS INTERFACE axis port=imageOut

ap_uint<19> hm[BIN_SIZE];
ap_uint<19> histogram[BIN_SIZE];
ap_uint<18> sumHist[BIN_SIZE];
strmio8_t tmp8;
ap_uint<19> value;
ap_uint<8> k0, i0;

HIST_INIT: for (ap_uint<9> i = 0; i < BIN_SIZE; i++){
      hm[i] = 0;
}


HIST_PROC: for (ap_uint<19> i = 0; i < DATA_SIZE; i++) {
#pragma HLS PIPELINE
      tmp8 = imageIn.read();
      hm[(unsigned char)tmp8.data]++;
}

value = 0;
HIST_EQ: for(ap_uint<9> i = 0; i < BIN_SIZE; i++){
#pragma HLS PIPELINE
      value = value + hm[i];
      sumHist[i] = value;
}

HIST_END: for(ap_uint<19> i = 0; i < DATA_SIZE; i++){
#pragma HLS PIPELINE

      tmp8 = imageIn.read();
      tmp8.data = (CONST * sumHist[(unsigned char)tmp8.data]) >> 11;
      tmp8.last = (i == DATA_SIZE-1) ? 1 : 0;
      tmp8.keep=0x1;
      tmp8.strb=0x1;
      imageOut.write(tmp8);

}}
```

Código 9-12 – Função de cálculo da equalização da imagem com base no seu histograma

As duas primeiras estruturas de repetição dizem respeito ao cálculo do histograma. A estrutura `FOR` seguinte (`HIST_EQ`) determina o somatório da equação 1. A última

estrutura (`HIST_END`) aplica o histograma acumulado aos pixels da imagem de entrada para obter cada um dos pixels da imagem de saída.

A função simplifica o cálculo da equação $L \times \frac{1}{A} \times \sum_{i=0}^{pixel} H(i)$ ao considerar a constante, CONST = $L \times \frac{1}{A} \times 1024 = 1$. Para recuperar o valor correto, $L \times \frac{1}{A}$, basta realizar um deslocamento de 10 posições no valor final. Evita-se assim a necessidade de utilizar uma divisão. Nos casos em que esta constante não é um valor inteiro, pode considerar-se uma aproximação ao inteiro mais próximo.

O circuito pode ser paralelizado de forma análoga ao que foi feito no cálculo do histograma.

9.4.1 Resultados de Implementação

A Tabela 9-4 apresenta os recursos necessários para a implementação do circuito resultado da síntese.

Fator de Paralelismo	Recursos			Latência (ciclos)
	LUT	FF	BRAM	
1	286	191	3	524816
4	631	391	11	131600

Tabela 9-4 – Utilização de recursos e latência total do circuito de equalização do histograma com fatores de paralelismo 1 e 4 para uma imagem de 512 × 512.

Aplicando o módulo hardware à imagem obtém-se a imagem equalizada (ver Figura 9-5).

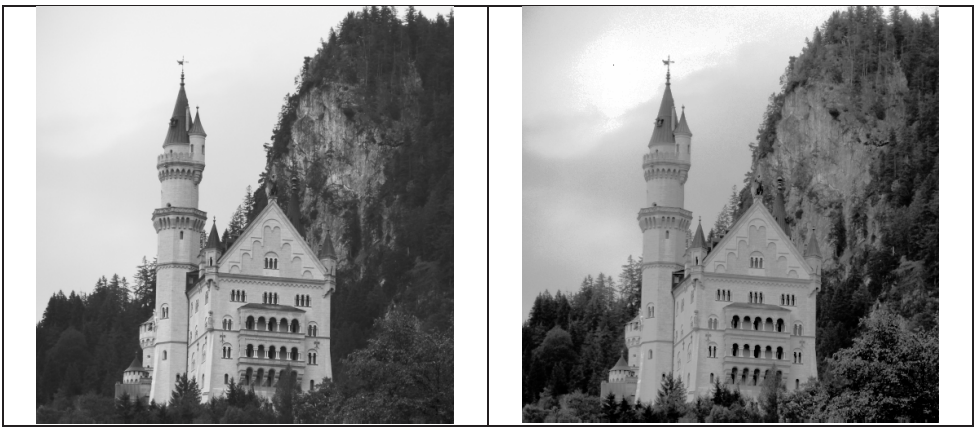


Figura 9-5 – Imagem equalizada (direita) após execução do módulo de equalização à imagem original (esquerda)

Com o paralelismo indicado consegue-se reduzir a latência em aproximadamente quatro vezes. É possível aumentar ainda mais o paralelismo, mas a sua eficácia está dependente do acesso aos dados através da interface *AXI4-Stream*. O aumento do paralelismo implica o aumento proporcional da largura de banda de acesso aos dados externos. Enquanto tal se

verificar, a latência reduz proporcionalmente ao fator de paralelismo. Quando se atinge o limite da taxa de transferência de dados, já não é vantajoso aumentar o fator de paralelismo.

9.5 Filtro de Resposta de Impulso Finito

Os filtros de resposta de impulso finito (FIR- *Finite Impulse Response*) são bastante utilizados em aplicações de processamento de sinal. Por exemplo, na implementação de filtros passa baixo para remover frequências altas não desejadas ou na realização de filtros passa banda para captar uma determinada gama de frequências. Também são bastante utilizados na restauração de sinal, em que, por exemplo, são utilizados para remover ruído.

A resposta de um filtro FIR é finita, ou seja, vai a zero passado um determinado tempo. A resposta ao impulso de um filtro de ordem `N` gera exatamente `N+1` pontos no tempo e depois vai a zero. Os filtros FIR podem ser contínuos no tempo ou discretos. Nesta secção, consideramos apenas o caso dos filtros discretos.

Um filtro FIR é descrito pela sua função de resposta ao filtro, `h[n]`, com `n=0,1,…,M-1`, em que `M` é a ordem do filtro. O sinal de saída, `y`, de um filtro FIR de ordem M é calculado através da convolução entre os coeficientes do filtro, `h`, com o sinal de entrada `x`, descrita pela equação:

$$y[i] = \sum_{j=0}^{M-1} h[j] \cdot x[i-j] \quad (1)$$

Como se pode constatar através da equação, cada elemento do sinal de saída é calculado à custa de `M` (número de coeficientes) multiplicações entre os coeficientes e os pontos do sinal de entrada.

O cálculo da convolução pode ser feito através do deslocamento do sinal de entrada ao longo de registos que são multiplicados pelos coeficientes (ver exemplo com um filtro FIR de 5 coeficientes na Figura 9-6).

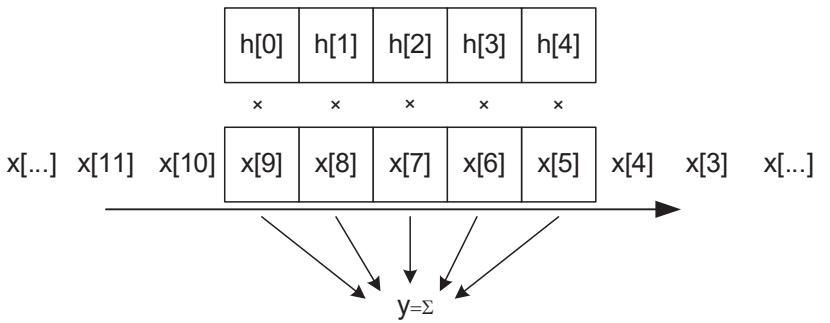


Figura 9-6 – Método de convolução entre os coeficientes e o sinal de entrada com deslocamento do sinal de entrada.

A figura identifica duas janelas de valores: uma de coeficientes e outra de elementos de entrada. Em cada passo, é calculado o produto vetorial entre os valores das duas janelas. Após o cálculo, os elementos de entrada são deslocados uma casa para a direita e procede-se a um novo cálculo. Inicialmente, consideram-se zeros na janela de elementos de entrada, ou seja, x[-1] = x[-2] = x[-3] = x[-4] = 0, de modo a calcular os primeiros valores de `y`.

Assim, para o cálculo do filtro FIR em hardware, vamos considerar a seguinte sequência de operações:

- Leitura dos coeficientes e armazenamento em registos;
- Inicialização da janela de entradas a 0;
- Inicialização do ciclo de cálculo da resposta:
    - Deslocam-se os elementos de entrada e calcula-se o produto vetorial entre os coeficientes e os valores de entrada, de acordo com a equação 1;
    - Escreve-se o resultado na interface de saída

De forma similar ao que foi considerado nos exemplos anteriores, os dados são enviados por interfaces *AXI4-Stream*. O tamanho do filtro (número de coeficientes) é fixo, os dados de entrada são representados em vírgula fixa com 32 bits (2 bits na parte inteira) e os coeficientes são representados também em vírgula fixa com 32 bits (1 bit na parte inteira).

Os parâmetros da arquitetura são definidos de acordo com o ficheiro cabeçalho `axis_fir.h` descrito no Código 9-13.

```
#include <ap_int.h>
#include <ap_fixed.h>
#include <hls_stream.h>
#include <ap_axi_sdata.h>

#define SIGNAL_POINTS 320
#define NUM_TAPS 29

typedef ap_fixed<32,2> data_t;
typedef ap_fixed<32,1> coef_t;
typedef hls::axis<data_t, 0, 0, 0> strmio_t;
```

Código 9-13 – Definição dos parâmetros do filtro e da função de descrição do filtro FIR (`axis_fir.h`)

O exemplo considera um filtro com 29 coeficientes e um sinal de entrada discreto com 320 pontos. A descrição do filtro FIR encontra-se no Código 9-14.

```
#include "axis_fir.h"

void axis_fir ( hls::stream<strmio_t> &strm_in, hls::stream<strmio_t> &strm_out )
{
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=strm_in
#pragma HLS INTERFACE axis port=strm_out
```

```
strmio_t tmpi, tmpo;
static data_t hwin[NUM_TAPS];
static coef_t coeffs[NUM_TAPS];
#pragma HLS ARRAY_PARTITION dim=1 type=complete variable=coeffs
#pragma HLS ARRAY_PARTITION dim=1 type=complete variable=hwin

ap_int<32> i, k;
data_t in_val, out_val;

LE_COEFS: for (i=0 ; i<NUM_TAPS; i++) {
	tmpi = strm_in.read();
	coeffs[i](31,0) = tmpi.data.range(31,0);
}

INIT_RD: for (i=0; i<NUM_TAPS; i++) {
#pragma HLS UNROLL
	hwin[i] = 0;
}

FIR_PROC: for (i=0; i< SIGNAL_POINTS; i++) {
	tmpi = strm_in.read();
	in_val = tmpi.data.range(31,0);

	SR: for(k = NUM_TAPS-1; k >= 0; k--) {
		hwin[k] = (k == 0) ? in_val : hwin[k - 1];
	}

	out_val = 0;
	HCONV: for(k = 0; k < NUM_TAPS; k++) {
		out_val += hwin[k] * coeffs[k];
	}

	tmpo.data = out_val;
	tmpo.last = (i == SIGNAL_POINTS -1) ? 1 : 0;
	tmpo.keep = 0xF;
	tmpo.strb = 0xF;
	strm_out.write(tmpo);
}}
```

Código 9-14 – Função de descrição do filtro FIR

A primeira estrutura `FOR` (`LE_COEFS`) lê os coeficientes do filtro pela interface *AXI4-Stream* e armazena-os em memória local. O vetor de armazenamento dos filtros é desenrolado por completo, sendo implementado com registos, para permitir o acesso em paralelo a todos os coeficientes.

A estrutura `FOR` seguinte (`INIT_RD`) inicializa a janela de valores de entrada a zero. O cálculo da convolução é feito dentro do `FOR` (`FIR_PROC`) para todos os elementos de entrada. A estrutura de repetição interior (`SR`) faz o deslocamento dos valores de entrada dentro da janela respetiva, e a estrutura seguinte (`HConv`) calcula o produto interno entre os valores das janelas de coeficientes e de elementos de entrada. Esta estrutura `FOR` é

completamente desenrolada, pois encontra-se dentro da *pipeline*, o que permite explorar o paralelismo disponível no cálculo do produto interno.

Por fim, à medida que os valores da resposta de impulso são calculados, são enviados pela interface *AXI4-Stream*.

O algoritmo paraleliza o cálculo do somatório (produto interno) com um fator de paralelismo igual ao número de coeficientes do filtro. Deste modo, a latência do circuito depende do tamanho do vetor `x`, `SIGNAL_POINTS`, e do número de coeficientes, `NUM_TAPS`. Não considerando os ciclos de enchimento da *pipeline*, a latência teórica, `latY`, do circuito FIR é determinada do seguinte modo:

$$\text{latY} = NUM\_TAPS + SIGNAL\_POINTS$$

O primeiro termo diz respeito à leitura dos coeficientes e o segundo diz respeito ao cálculo da resposta. No exemplo descrito, a latência seria aproximadamente 29 + 320 = 359 ciclos.

9.5.1 Execução Paralela de Cálculo do Filtro FIR

O tempo de execução do algoritmo pode ser melhorado com o cálculo de várias convoluções em paralelo. Para tal, é necessário ler vários elementos do vetor de entrada em simultâneo. Consideremos, como exemplo, o cálculo paralelo de duas convoluções. A interface *AXI4-Stream* passa a 64 bits, permitindo a leitura de dois coeficientes ou dados em simultâneo (ver Código 9-15), assumindo que estão disponíveis.

```
void axis_fir ( hls::stream<strmio_t> &strm_in, hls::stream<strmio_t> &strm_out )
{
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=strm_in
#pragma HLS INTERFACE axis port=strm_out

strmio_t tmpi, tmpo;
static data_t hwin[NUM_TAPS +1];
static coef_t coeffs[NUM_TAPS];
#pragma HLS ARRAY_PARTITION dim=1 type=complete variable=coeffs
#pragma HLS ARRAY_PARTITION dim=1 type=complete variable=hwin

ap_int<32> i, k;
data_t in_val, in_val1, out_val, out_val1;

LE_COEFS: for (i=0 ; i<NUM_TAPS; i+=2) {
       tmpi = strm_in.read();
       coeffs[i].range(31,0) = tmpi.data.range(31,0);
       coeffs[i+1].range(31,0) = tmpi.data.range(63,32);
}

INIT_RD: for (i=0; i<NUM_TAPS -1; i++) {
#pragma HLS UNROLL
       hwin[i] = 0;
}
```

```
FIR_PROC: for (i=0; i<SIGNAL_POINTS /2; i++) {

	tmpi = strm_in.read();
	in_val.range(31,0) = tmpi.data.range(31,0);
	in_val1.range(31,0) = tmpi.data.range(63,32);

	SR: for(k = NUM_TAPS; k >= 0; k--) {
		hwin[k] = (k == 1) ? in_val : (k == 0) ? in_val1 : hwin[k - 2];
	}

	out_val = 0;
	out_val1 = 0;
	HConv: for(k = 0; k < NUM_TAPS; k++) {
		out_val += hwin[k] * coeffs[k];
		out_val1 += hwin[k+1] * coeffs[k];
	}

	tmpo.data.range(31,0) = out_val1.range(31,0);
	tmpo.data.range(63,32) = out_val.range(31,0);
	tmpo.last = (i == SIGNAL_POINTS/2 -1) ? 1 : 0;
	tmpo.keep = 0xF;
	tmpo.strb = 0xF;
	strm_out.write(tmpo);
}}
```

Código 9-15 – Função HLS de descrição do filtro FIR com cálculo de duas convoluções em paralelo

A estrutura de repetição de leitura dos coeficientes lê dois valores em simultâneo pela interface *AXI4-Stream* de 64 bits. A sua latência é, assim, reduzida a metade.

O número de iterações do ciclo de cálculo das convoluções (`loop_fir`) também reduz a metade, pois são executadas duas convoluções por cada iteração. O ciclo de deslocamento dos registos (`SR`) desloca os valores da janela de elementos dos dados de entrada de dois em dois devido a estarmos a considerar duas convoluções em paralelo. O ciclo de cálculo dos produtos vetoriais (`HConv`) considera um produto entre os coeficientes e os elementos da janela de dados com índice entre [0, 28] e um outro produto com os mesmos coeficientes, mas com os elementos da janela de dados com índice entre [1, 29]. Os dois resultados dos produtos vetoriais são compactados em 64 bits e depois enviados em simultâneo pela interface *AXI4-Stream*.

9.5.2 Resultados de Implementação

A Tabela 9-5 apresenta os recursos utilizados após síntese dos circuitos e as latências totais para o filtro FIR de 29 coeficientes e 320 pontos de entrada. A versão 1 corresponde ao circuito com o cálculo de apenas um convolução de cada vez e a versão 2 corresponde ao circuito com o cálculo de duas convoluções em paralelo.

Versão	Recursos				Latência (ciclos)
	LUT	FF	BRAM	DSP	
1	2530	1596	0	116	363
2	4705	2168	0	232	189

Tabela 9-5 – Utilização de recursos e latência total do circuito de implementação do filtro FIR com 29 coeficientes e um vetor de entrada de 320 elementos.

Os resultados comprovam que a latência reduz a aproximadamente metade quando se considera o cálculo em paralelo de duas convoluções. Em troca, foi necessário o dobro de unidades DSP, bem como um aumento de 87% nas LUT e 36% de FF.

9.6 Convolução 2D

A convolução 2D é uma operação muito utilizada no processamento de imagens. A operação consiste em percorrer uma imagem 2D com um filtro de pequenas dimensões, comparativamente à imagem. Os valores dos pixels são multiplicados pelos coeficientes do filtro e os produtos são acumulados. Para exemplificar a operação, consideremos uma imagem de 6×6 e um filtro de 3×3 (ver Figura 9-7).

3	5	0	1	4	4
1	2	5	3	4	1
0	5	1	5	2	2
1	2	4	3	3	1
3	2	4	1	0	0
4	2	2	1	1	0

a) Imagem 6x6

1	2	1
2	1	2
1	2	1

b) Filtro 3x3

Figura 9-7 – Exemplo de uma imagem de 6×6 e um filtro de 3×3 com os quais se pretende aplicar a operação de convolução.

A Figura 9-7a ilustra a imagem de entrada, em que os valores de cada um dos quadrados correspondem aos valores dos pixels da imagem. A Figura 9-13b ilustra o filtro que se pretende aplicar à imagem com a operação de convolução. Quer a imagem, quer o filtro, são representados com matrizes do tamanho respetivo.

Para realizar a convolução, é necessário aplicar o filtro sucessivamente sobre a imagem, deslocando-se após o cálculo de cada valor da saída. A convolução começa geralmente pelo ponto em cima do lado esquerdo. Comecemos por este ponto, como ilustrado na Figura 9-8.

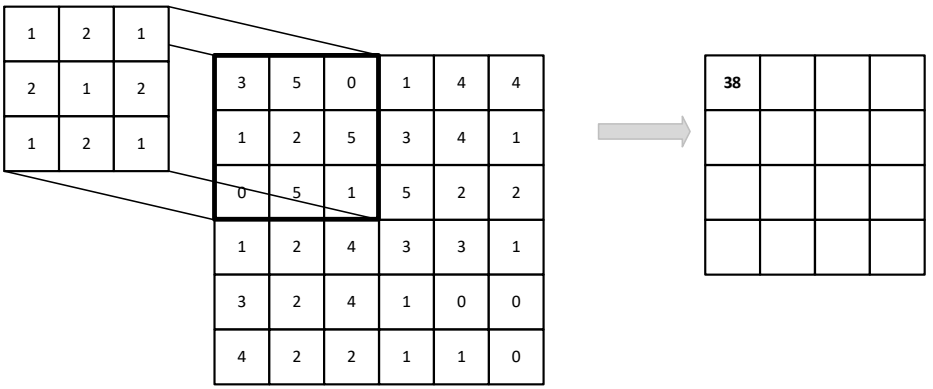


Figura 9-8 – Primeiro passo da convolução entre a imagem e o filtro do exemplo.

Cada um dos coeficientes do filtro é multiplicado pelo pixel respetivo da imagem. O valor final da operação, `Conv(0,0)`, resulta da acumulação de todos os produtos do seguinte modo:

$$\text{Conv}(0,0) = 1\times3 + 2\times5 + 1\times0 + 2\times1 + 1\times2 + 2\times5 + 1\times0 + 2\times5 + 1\times1 = 38$$

O valor surge na matriz de saída ilustrada à direita na figura. De seguida, desloca-se o filtro um pixel à direita e aplica-se a mesma operação (ver Figura 9-9).

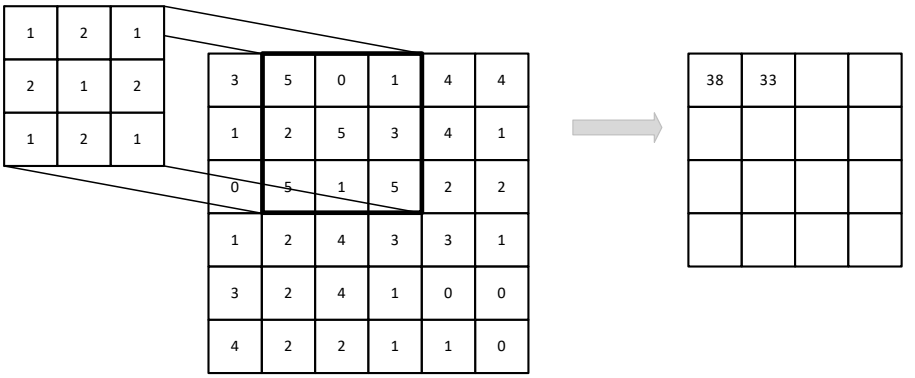


Figura 9-9 – Segundo passo da convolução entre a imagem e o filtro do exemplo.

O processo repete-se até percorrer toda a imagem de entrada. O resultado é uma matriz de 4×4 (ver o resultado da matriz exemplo na Figura 9-10).

38	33	46	36
26	49	39	38
34	37	36	23
35	32	27	15

Figura 9-10 – Resultado da convolução entre a imagem 6×6 e o filtro 3×3 do exemplo.

Teoricamente, um ponto do mapa de saída da convolução, `c[x,y]`, entre um filtro, `k[x,y]`, e uma imagem, `im[x,y]`, é calculado de acordo com a seguinte expressão:

$$c[x,y] = \sum_{dy}\sum_{dx} k[dx,dy]\, im[x+dx, y+dy]$$

em que `dx` e `dy` percorrem as dimensões do filtro em `X` e em `Y`.

Um algoritmo simples de implementação da convolução aplica iterativamente a equação de cálculo de um ponto do mapa de saída de convolução a todos os pontos da imagem (ver Código 9-16).

```
int im[xs][ys]; //Imagem de entrada
int k[xk][yk];  //Filtro
int m[xm][ym];        // Mapa de saída

for (y = 0; y < ys; y++){
    for (x = 0; x < xs; x++){
        acc = 0;
        for (dy = 0; dy < yk; dy++){
            for (dx = 0; dx < xk; dx++){
                acc += im[x+dx][y+dy] * k[dx][dy];
            }
        }
        m[x,y] = acc;
    }
}
```

Código 9-16 – Algoritmo de convolução 2D

Quando aplicamos o algoritmo de convolução a uma imagem, verifica-se que a imagem ou mapa de saída tem uma dimensão menor. Esta redução deve-se ao facto de que a aplicação do filtro às últimas linhas ou às últimas colunas não é possível por falta de valores de imagem dentro da janela do filtro. Por exemplo, consideremos uma imagem de 6×6 e um filtro de 3×3. Ao aplicarmos o filtro às duas últimas colunas da imagem, concluímos que não é possível aplicar a convolução pois são necessárias tantas colunas quanto a dimensão do

filtro. O mesmo acontece com as duas últimas linhas. Neste caso, a imagem de saída é apenas de 4×4. No caso de se querer uma imagem de saída com a dimensão igual à da imagem de entrada aplica-se a técnica de *padding*. Esta consiste em aumentar a imagem de entrada com novos valores (por exemplo, zeros) à sua volta. No exemplo considerado, bastaria acrescentar zeros a toda a volta da imagem.

O algoritmo apresentado no Código 9-16, sem considerar a técnica de *padding*, pode ser facilmente sintetizado com *pipeline*, permitindo uma multiplicação-acumulação por ciclo de relógio. Podemos acelerar a operação de convolação de duas formas: 1) paralelizando as operações para o cálculo de um ponto do mapa de saída e/ou 2), aumentando o número de pontos do mapa de saída calculados em paralelo.

Para aumentar o número de operações em paralelo é preciso aumentar o número de leituras em paralelo da imagem e do filtro. O problema é que os acessos à imagem não são sequenciais, obrigando a ter vários portos para garantir o acesso aos pixels em posições diferentes. Esta questão é resolvida com uma técnica baseada em registos de deslocamento. A ideia é manter uma janela de valores da imagem em registos de deslocamento que vão sendo deslocados a cada convolução.

Por exemplo, se o filtro tiver uma dimensão de 3×3, as primeiras duas linhas da imagem e os três primeiros pixels da terceira linha são inicialmente guardados em registos de deslocamento. Após o cálculo da primeira convolução, os elementos do registo são deslocados uma casa para a direita. A Figura 9-11 apresenta a solução baseada em registos de deslocamento para o exemplo com imagens de 6×6 e filtro de 3×3.

O filtro não altera durante o cálculo da convolução com a imagem. O deslocamento do filtro ao longo da imagem é conseguido mantendo o filtro inalterado e deslocando as linhas da imagem. A multiplicação e a acumulação do produto entre os coeficientes do filtro e os pixels da imagem são feitas em paralelo. No exemplo da figura, com um filtro de 3×3, a convolução pode ser calculada em paralelo com 9 multiplicadores e somadores. Assim, com um único ciclo de relógio, é calculado um elemento do mapa de saída. Após o cálculo, os elementos do registo são deslocados para a direita.

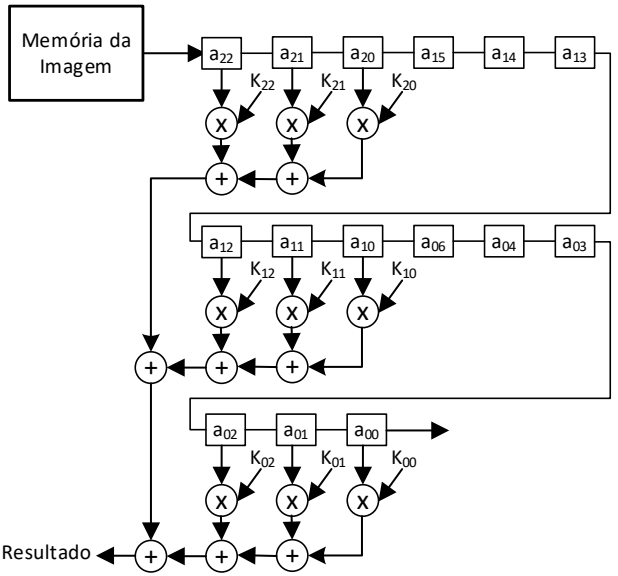


Figura 9-11 – Cálculo da convolução com registos de deslocamento. Primeiro pixel de saída.

Ao fazer o deslocamento, obtém-se a próxima janela de pixels (ver Figura 9-12).

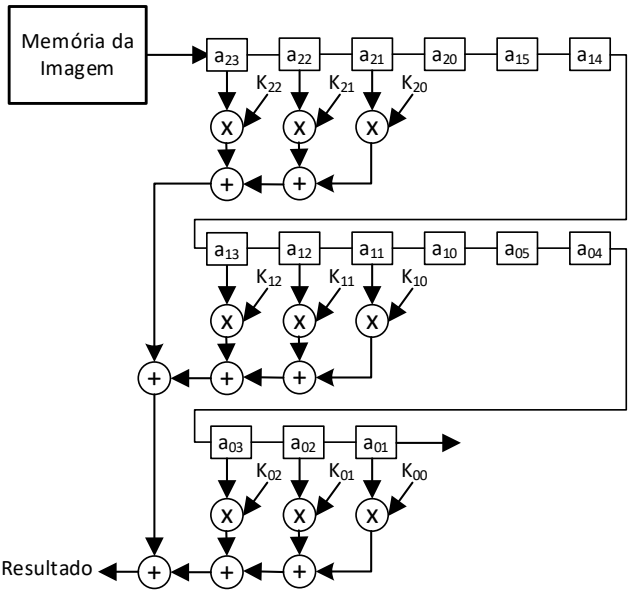


Figura 9-12 – Cálculo da convolução com registos de deslocamento. Cálculo do segundo pixel de saída.

A descrição do circuito segue o método de cálculo da convolução apresentado anteriormente. Como exemplo de aplicação, vamos considerar imagens de 32×32, um filtro de 5×5 e a não utilização de *padding*. Assume-se que os dados são transferidos por interfaces *AXI4-Stream*. O filtro é lido apenas uma vez e mantido constante durante toda a execução da função. Os pixels da imagem e os coeficientes do filtro são representados com inteiros de 8 bits.

Os parâmetros da função são definidos no ficheiro cabeçalho (`axis_conv2D.h`) de acordo com o Código 9-17.

```
#include <ap_int.h>
#include <hls_stream.h>
#include <ap_axi_sdata.h>

#define K_SIZE 5
#define IHEIGHT 32
#define IWIDTH 32

#define PADDING 0

#define OHEIGHT (IHEIGHT-(K_SIZE-1)+PADDING*(K_SIZE-1))
#define OWIDTH  (IWIDTH-(K_SIZE-1)+PADDING*(K_SIZE-1))

#define SREG_SIZE (K_SIZE+IWIDTH*(K_SIZE-1))

typedef hls::axis<ap_int<8>, 0, 0, 0> strmio_t;

typedef ap_int<8> weigth_t;
typedef ap_int<8> imap_t;
```

Código 9-17 – Definição dos parâmetros da função de descrição da convolução 2D (`axis_conv2D.h`)

A descrição da convolução 2D encontra-se descrita no Código 9-18.

```
#include "axis_conv2D.h"

void axis_conv2Dv2 ( hls::stream<strmio_t> &strm_in,
                     hls::stream<strmio_t> &strm_out )
{
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=strm_in
#pragma HLS INTERFACE axis port=strm_out

strmio_t tmp1, tmp2;
int i, j, pos, row, col;
imap_t hwin[SREG_SIZE], in_val;
ap_int<16> out_val;
static weigth_t kernel[K_SIZE*K_SIZE];
#pragma HLS ARRAY_PARTITION dim=1 type=complete variable=hwin
#pragma HLS ARRAY_PARTITION dim=1 type=complete variable=kernel

LE_FILTRO: for (i=0; i<K_SIZE*K_SIZE; i++) {
```

```
        tmp1 = strm_in.read();
        kernel[i] = (weigth_t)tmp1.data;
        if (tmp1.last == (ap_uint<1>)1) break;
}

PROC_CONV: for(pos = 0, row = 0; row < IHEIGHT; row++) {
        for(col = 0; col < IWIDTH; pos++, col++) {
                tmp1 = strm_in.read();
                in_val = tmp1.data;

                SR: for(i = 0; i < SREG_SIZE; i++) {
                        hwin[i] = (i == SREG_SIZE-1) ? in_val : hwin[i+1];
                }

                if (pos >= SREG_SIZE-1 && col >= K_SIZE-1) {
                        out_val = 0;
                        for(i = 0; i < K_SIZE; i++) {
                                for(j = 0; j < K_SIZE; j++) {
                                    out_val += hwin[i*IWIDTH+j] * kernel[i*K_SIZE+j];
                                }
                        }
                        tmp2.data = out_val.range(7,0);
                        tmp2.last = (pos == IHEIGHT*IWIDTH -1) ? 1 : 0;
                        strm_out.write(tmp2);
                }
        }
}}
```

Código 9-18 – Função HLS de descrição da convolução 2D

Os vetores `hwin` e `kernel` são utilizados para guardar os pixels e o filtro, respetivamente. São particionados por completo (`#pragma HLS ARRAY PARTITION`) para aceder em paralelo a todos os seus elementos.

A primeira estrutura de repetição `FOR` (`LE_FILTRO`) lê os coeficientes do filtro pela interface *AXI4-Stream* e armazena-os em memória local. A segunda estrutura `FOR` (`PROC_CONV`) calcula todas as convoluções uma a uma por linha e por coluna.

Dentro do `PROC_CONV` encontra-se o `FOR` que descreve o registo de deslocamento (`SR`) com um tamanho suficiente para armazenar as primeiras quatro linhas da imagem mais 5 pixels da quinta linha, uma vez que o filtro tem dimensão 5×5. Apenas se inicia o cálculo da primeira convolução após preencher o registo de deslocamento. A partir daqui, após ler um pixel é calculada uma convolução. Para permitir uma convolução por ciclo de relógio, sendo um filtro de dimensão 5×5, são utilizados 25 multiplicadores em paralelo.

O algoritmo irá demorar um número de ciclos de relógio aproximadamente igual à dimensão da imagem de entrada, uma vez que é necessário ler todos os pixels em sequência. No entanto, a imagem de saída é menor, como explicado anteriormente. Para o exemplo considerado, a imagem de saída terá uma dimensão de 28×28. Como tal, o número de escritas *AXI4-Stream* é menor que o número de leituras. O cálculo da convolução e o

respetivo envio está condicionado à verificação da condição `col >= K_SIZE-1`, ou seja, apenas calcula a convolução se existirem elementos suficientes no registo de deslocamento.

O algoritmo paraleliza o cálculo da convolução com um fator de paralelismo igual ao número de coeficientes do filtro. Deste modo, a latência do circuito é dada aproximadamente pelo tamanho da imagem de entrada mais o tamanho do registo de deslocamento, uma vez que é necessário preencher o registo antes de iniciar o cálculo.

9.6.1 Execução Paralela de Cálculo da Convolução 2D

Similar ao que foi feito no circuito FIR, também aqui é possível calcular várias convoluções em paralelo. Consideremos, como exemplo, o cálculo paralelo de quatro. Isto implica que tenham de ser lidos e deslocados quatro pixels de cada vez, seguindo a ordem com que são calculados os elementos do mapa de saída. A interface *AXI4-Stream* passa a transmitir 32 bits por transação, permitindo a leitura de quatro pixels ou quatro coeficientes em simultâneo (ver Código 9-19).

```
#include "axis_conv2D.h"

void axis_conv2Dpar (hls::stream<strmio_t> &strm_in,
                     hls::stream<strmio_t> &strm_out )
{
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=strm_in
#pragma HLS INTERFACE axis port=strm_out

strmio_t tmp1, tmp2;
int i, j, pos, row, col;
ap_int<16> out_val1, out_val2, out_val3, out_val4;

imap_t hwin[SREG_SIZE+3], in_val1, in_val2, in_val3, in_val4;
static weigth_t kernel[K_SIZE*K_SIZE];
#pragma HLS ARRAY_PARTITION dim=1 type=complete variable=hwin
#pragma HLS ARRAY_PARTITION dim=1 type=complete variable=kernel

LE_COEF: for (i=0; i<K_SIZE*K_SIZE+3; i+=4) {
       tmp1 = strm_in.read();
       kernel[i] = (weigth_t)tmp1.data.range(7,0);
       kernel[i+1] = (weigth_t)tmp1.data.range(15,8);
       kernel[i+2] = (weigth_t)tmp1.data.range(23,16);
       kernel[i+3] = (weigth_t)tmp1.data.range(31,24);
       if (tmp1.last == (ap_uint<1>)1) break;
}

PROC_CONV: for(pos = 0, row = 0; row < IHEIGHT; row++) {
    for(col = 0; col < IWIDTH; pos+=4, col+=4) {
#pragma HLS PIPELINE;
       tmp1 = strm_in.read();
       in_val1 = tmp1.data.range(7,0);
       in_val2 = tmp1.data.range(15,8);
```

```
        in_val3 = tmp1.data.range(23,16);
        in_val4 = tmp1.data.range(31,24);

        SR: for(i = 0; i < SREG_SIZE; i+=4) {
               hwin[i] = (i == SREG_SIZE-4) ? in_val1 : hwin[i+4];
               hwin[i+1] = (i == SREG_SIZE-4) ? in_val2 : hwin[i+5];
               hwin[i+2] = (i == SREG_SIZE-4) ? in_val3 : hwin[i+6];
               hwin[i+3] = (i == SREG_SIZE-4) ? in_val4 : hwin[i+7];
        }

        if (pos >= SREG_SIZE-4 && col >= K_SIZE-1) {
               out_val1 = 0; out_val2 = 0; out_val3 = 0; out_val4 = 0;

               HConv: for(i = 0; i < K_SIZE; i++) {
                         for(j = 0; j < K_SIZE; j++) {
                            out_val1 += hwin[i*IWIDTH+j] * kernel[i*K_SIZE+j];
                            out_val2 += hwin[i*IWIDTH+j+1] * kernel[i*K_SIZE+j];
                            out_val3 += hwin[i*IWIDTH+j+2] * kernel[i*K_SIZE+j];
                            out_val4 += hwin[i*IWIDTH+j+3] * kernel[i*K_SIZE+j];
                         }
                      }

               tmp2.data.range(7,0) = out_val1.range(7,0);
               tmp2.data.range(15,8) = out_val2.range(7,0);
               tmp2.data.range(23,16) = out_val3.range(7,0);
               tmp2.data.range(31,24) = out_val4.range(7,0);
               tmp2.last = (pos == IHEIGHT*IWIDTH -1) ? 1 : 0;
               strm_out.write(tmp2);
        }
      }
}}
```

Código 9-19 – Função HLS de descrição da convolução 2D com cálculo de quatro convoluções em paralelo

Os elementos do filtro são lidos de quatro em quatro por cada transação *AXI4-Stream*. Como o filtro tem 25 valores, não é múltiplo de quatro, adicionaram-se zeros aos últimos elementos do filtro até perfazer 28.

O ciclo de controlo do número de colunas avança igualmente de quatro em quatro. Em cada acesso aos dados, são lidos quatro novos pixels e o deslocamento é feito de quatro em quatro. No ciclo de cálculo da convolução são executadas quatro convoluções em paralelo.

9.6.2 Resultados de Implementação

A Tabela 9-6 apresenta os recursos utilizados após síntese e as latências totais para as duas versões do circuito. A versão 1 corresponde ao primeiro circuito proposto, em que é calculado apenas um pixel de cada vez. A versão 2 corresponde ao circuito com cálculo paralelo de quatro pixels.

Versão	Recursos			Latência (ciclos)
	LUT	FF	DSP	
1	263	315	25	1062
2	629	601	100	276

Tabela 9-6 – Utilização de recursos e latência total do circuito de implementação da convolução 2D com imagens de 32×32 e filtros de 5×5.

Os resultados apresentados na tabela comprovam que a latência é proporcional ao tamanho da imagem e do filtro, e inversamente proporcional ao número de cálculos realizados em paralelo. Os recursos também aumentam com a execução paralela: os DSP e SRL (número de LUT configuradas como registo de deslocamento) aumentam aproximadamente quatro vezes, e as LUT e os FF aumentam próximo do dobro.

9.7 Multiplicação de Matriz Esparsa por Vetor Denso

Uma matriz esparsa tem a maioria dos seus elementos igual a zero. A matriz esparsa caracteriza-se pela densidade de valores diferentes de zero, dada pela percentagem de valores diferentes de zero do total de elementos da matriz. A multiplicação de uma matriz esparsa por um vetor, geralmente denso, é uma operação comum em várias aplicações científicas e de processamento de dados. Uma matriz esparsa pode ser representada como uma matriz densa e multiplicada por um vetor utilizando um algoritmo de multiplicação de matriz por vetor. Contudo, sendo que a maioria das entradas da matriz esparsa são zero, evitar as multiplicações por zero reduz bastante a complexidade do algoritmo. Por exemplo, a multiplicação de uma matriz quadrada de 500×500 por um vetor de 500 realiza 500×500 = 250000 multiplicações. Por outro lado, considerando uma densidade de 0,5%, são necessárias apenas 500×500×0,5% = 1250 multiplicações.

9.7.1 Representação da Matriz Esparsa

Existem vários formatos para o armazenamento de matrizes esparsas que evitam guardar os elementos a zero e permitem o acesso organizado aos elementos diferentes de zero. Neste exemplo, adota-se o formato de representação CRS (*Compressed Raw Storage*) com compressão ao longo das linhas. A representação CRS consiste em três vetores:

- Um vetor (*Elementos*) para armazenar os elementos da matriz diferentes de zero;
- Um vetor (*Índice da coluna*) para guardar o índice da coluna em que se encontra cada um dos elementos diferentes de zero;
- Um vetor (*Ponteiro de linha*) para guardar o índice do vetor *Elementos* relativo ao primeiro elemento de cada linha.

Consideremos o exemplo da Figura 9-13, em que se ilustra a representação CRS de uma matriz com dimensão 6×6.

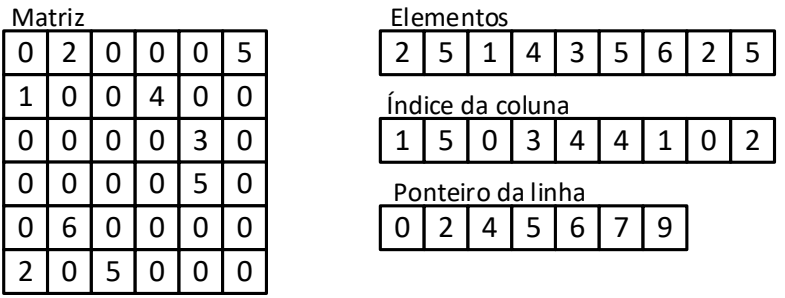


Figura 9-13 – Exemplo de representação de uma matriz no formato CRS.

Como ilustrado no exemplo, o vetor *Elementos* (`Elem`) contém todos os elementos diferentes de zero da matriz que são guardados percorrendo a matriz da esquerda para a direita e de cima para baixo. O vetor *Índice da coluna* (`iCol`) contém uma entrada para cada valor do vetor *Elementos* com o índice da coluna da matriz onde se encontra esse elemento. O vetor *Ponteiro da linha* (`pLin`) tem uma dimensão de 6+1 e contém o índice do vetor *Elementos* dos primeiros elementos de cada uma das linhas da matriz. Por exemplo, na linha 3 o primeiro elemento é o 5 que corresponde ao índice 5 do vetor *Elementos*.

O número de elementos da linha é dado por `pLin[l] – pLin[l+1]`. Para obter os elementos da linha e respetivas colunas basta aceder ao vetor dos elementos entre estes dois índices. Por exemplo, o primeiro elemento é `elem[pLin[i]]` que se encontra na coluna `pLin[i]`. Desta forma, conseguem-se recuperar todos os elementos da matriz sem ter de reconstruir a matriz original.

9.7.2 Algoritmo de Multiplicação de Matriz Esparsa por Vetor Denso

O algoritmo de multiplicação de matriz esparsa por vetor denso percorre os elementos de uma linha, multiplica pelo elemento do vetor respetivo e acumula. O processo repete-se para todas as linhas (ver Código 9-20).

```
#define ROWS 512          // número de elementos do vetor
#define MAX_ELEM 2000     // número máximo de elementos não zero da matriz

void spmv(int pLin[ROWS], int iCol[MAX_ELEM], int Ax[MAX_ELEM],
           int elem[ROWS], int p[ROWS]) {

unsigned int row, row_start, row_end, i;

LINHAS: for(row = 0; row < ROWS; row++){
       sum = 0;
       row_start = pLin[row];
       row_end   = pLin[row+1];

       ELEM_LINHA: for (i = row_start; i < row_end; i++){
              sum += Ax[i] * elem[iCol[i]];
```

```
        }
        p[row] = sum;
}}
```

Código 9-20 – Algoritmo de multiplicação de matriz esparsa por vetor denso

A estrutura de repetição exterior percorre todas as linhas e a estrutura de repetição interior percorre todos os elementos de uma linha. Os elementos do vetor a multiplicar são determinados pelos índices das colunas do vetor `iCol`.

Na descrição do algoritmo para posterior síntese, considera-se o ficheiro de cabeçalho (`axis_spmv.h`) onde se define o número de linhas da matriz (`ROWS`) e o número máximo de elementos da matriz esparsa (`MAX_ELEM`). (ver Código 9-21).

```
#include <ap_int.h>
#include <hls_stream.h>
#include <ap_axi_sdata.h>

#define ROWS 512
#define MAX_ELEM 1536

typedef hls::axis<ap_int<32>, 0, 0, 0> strmio32_t;
```

Código 9-21 – Código do ficheiro de cabeçalho (`axis_spmv.h`) do algoritmo de multiplicação de matriz esparsa por vetor denso

Na descrição do algoritmo, consideremos que o vetor denso é guardado em memória interna, juntamente com os vetores auxiliares `iCol` (índice da coluna) e `pLin` (ponteiro da linha) de representação da matriz esparsa. Os elementos da matriz esparsa são lidos e processados de imediato, pelo que não precisam de ser guardados em memória (ver Código 9-22).

```
#include "axis_spmv.h"

void AXISspmv(hls::stream<strmio32_t> &str_in,
              hls::stream<strmio32_t> &str_out){
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=str_in
#pragma HLS INTERFACE axis port=str_out

int elem[ROWS];
ap_uint<10> iCol[MAX_ELEM];
ap_int<10> pLin[ROWS+1];
strmio32_t tmp32;
int soma;

LE_PLIN: for (ap_uint<10> k = 0; k <= ROWS; k++) {
#pragma HLS PIPELINE
       tmp32 = str_in.read();
       pLin[k] = (int)tmp32.data;
}

LE_ICOL: for (ap_uint<12> k = 0; k < MAX_ELEM; k++) {
```

```
#pragma HLS PIPELINE
	tmp32 = str_in.read();
	iCol[k] = (int)tmp32.data;
	if (tmp32.last == 1){
		break;
	}
}

LE_ELEM: for (ap_uint<10> j = 0; j < ROWS; j++) {
#pragma HLS PIPELINE
	tmp32 = str_in.read();
	elem[j] = (int)tmp32.data;
}

MULT_CICLO: for(ap_uint<10> row = 0; row < ROWS; row++){
		soma = 0;
		CICLO1: for (ap_uint<10> i = pLin[row]; i < pLin[row+1]; i++){
		#pragma HLS PIPELINE
			tmp32 = str_in.read();
			soma += (int)tmp32.data * elem[iCol[i]];
		}
		tmp32.data = ap_int<32>(soma);
		tmp32.last = (row == ROWS-1) ? 1 : 0;
		tmp32.keep=0xF;
		tmp32.strb=0xF;
		str_out.write(tmp32);
	}
}
```

Código 9-22 – Função de descrição do cálculo da multiplicação de uma matriz esparsa por um vetor denso.

As três primeiras estruturas de repetição (`LE_PLIN`, `LE_ICOL`, `LE_ELEM`) leem os dados da matriz e o vetor denso por uma interface `AXI4-Stream` e guardam-nos em memória interna. A última estrutura (`MULT_CICLO`) calcula e acumula o produto entre os elementos não zero da matriz com os elementos respetivos do vetor para todas as linhas. O resultado é um vetor que é enviado pela interface `AXI4-Stream`.

Uma vez que apenas se lê um valor de cada vez pela interface `AXI4-Stream`, o algoritmo tem um paralelismo limitado. Poder-se-ia pensar em desenrolar o `FOR CICLO1`, interno ao `MULT_CICLO`, mas como os limites da estrutura de repetição não são constantes, o HLS não o consegue desenrolar.

9.7.3 Resultados de Implementação

A Tabela 9-7 apresenta os recursos necessários para a implementação do circuito resultado da síntese de alto nível da função, bem como o número de ciclos de relógio de execução.

Nº Linhas/ Nº Máximo de Elementos	Recursos				Latência (ciclos)
	LUT	FF	BRAM	DSP	
512/1536	387	447	4	6	10 769
512/3072	393	448	5	6	13 329

Tabela 9-7 – Utilização de recursos e latência total do circuito do circuito de multiplicação de matriz esparsa por vetor.

Quando variamos o número de elementos da matriz, os recursos de memória aumentam, pois é necessário guardar mais elementos da matriz. A latência também aumenta pois é necessário ler mais elementos da matriz e executar mais operações.

9.8 Operação de Ordenação

Existem diversos algoritmos de ordenação de dados. Nesta secção, considera-se um algoritmo básico de ordenação que permite explorar bastante paralelismo – ordenação por inserção (*insertion sort*).

9.8.1 Ordenação por Inserção

O algoritmo por inserção é um algoritmo iterativo que considera um elemento de cada vez. Em cada iteração, coloca um elemento na ordem certa. Para exemplificar o funcionamento do algoritmo, considere-se a seguinte lista de elementos:

{4, 3, 1, 2}

Começando pelo primeiro elemento da lista, define-se uma sublista com apenas um elemento. Este sublista, tendo apenas um elemento, já se encontra, naturalmente, ordenada.

{**4**}, {3, 1, 2}

De seguida, escolhe-se o próximo elemento da lista de valores não ordenados e insere-se na sublista ordenada.

{**3**, **4**}, {1, 2}

A sublista ordenada passa a ter os elementos 3 e 4 na ordem certa. Para tal, colocou-se o elemento 3 no início da sublista e deslocou-se o elemento 4 para a direita. O procedimento repete-se para os restantes elementos não ordenados. Insere-se o elemento 1

{**1**, **3**, **4**}, {2}

seguido do elemento 2

{**1**, **2**, **3**, **4**}

O algoritmo é descrito formalmente pelo Código 9-23.

```
void isort(int X[SIZE]){

    C1: for(int i = 1; i < SIZE; i++){
          int elem = X[i];
          int j = i;

        C2: while(j > 0 && X[j-1] > elem) {
            X[j] = X[j-1];
            j--;
        }
        X[j] = elem;
    }
}
```

Código 9-23 – Algoritmo de ordenação por inserção

O algoritmo considera `X[0]` como sublista inicial ordenada. A estrutura de repetição `C1` percorre todos os elementos a partir do segundo elemento da lista e a `C2` percorre a sublista ordenada para inserir o novo elemento de forma ordenada. Assim, o `FOR` exterior percorre todos os elementos a ordenar desde o `X[1]` até ao `X[SIZE-1]`, em que `SIZE` corresponde ao número de elementos a ordenar. Para cada elemento a inserir, o `FOR` interior percorre a sublista já ordenada, cujo tamanho é dado pela variável de iteração do `FOR` exterior (`i`), e desloca todos os elementos maiores que o novo elemento a inserir de modo a libertar a posição onde será inserido o novo elemento. O `FOR` interior executa enquanto não chegar ao fim da sublista ordenada (`j>0`) e enquanto os elementos forem maiores que o elemento a inserir. Ao sair do `FOR` interior, o elemento a inserir é guardado na lista na posição com índice determinado pela variável de controlo do `FOR` interior (`j`).

O número de iterações do algoritmo depende naturalmente da ordenação inicial dos elementos, mas em média demora $SIZE^2/4$ passos[1]. Se sintetizada em hardware, assumindo uma comparação por ciclo de relógio, teríamos uma latência média de $SIZE^2/4$.

Por ter complexidade quadrática, o algoritmo de ordenação por inserção é utilizado geralmente em listas mais pequenas, servindo muitas vezes de base a algoritmos de ordenação mais complexos que combinam hierarquicamente as várias listas preordenadas.

Com listas pequenas, é possível executar todas as comparações em paralelo, passando a latência a depender linearmente do número de elementos a ordenar.

9.8.2 Paralelização do Algoritmo de Ordenação por Inserção

O código do algoritmo descrito anteriormente apresenta algumas limitações quando se pretende otimizar a implementação hardware. A existência de uma estrutura de repetição

[1] Thomas Cormen, Charles Leiserson, Ronald Rivest, "Introduction to Algorithms", MIT Press Ltd, ISBN:9780262033848

interior sem um limite conhecido e constante impede a aplicação de *pipeline* ao corpo da estrutura de repetição exterior. Assim, é necessário reorganizar o código de modo a permitir aplicar de forma eficiente mecanismos de otimização como o *pipeline* e o desenrolamento, para se obter uma implementação hardware otimizada.

É possível implementar o algoritmo de ordenação por inserção, em hardware, de modo a ordenar um novo elemento a cada ciclo de relógio. Para tal, é necessário usar tantos comparadores quanto o número de elementos da lista a ordenar e poder deslocar todos os elementos da lista para permitir inserir o novo elemento na posição certa. Este comportamento pode ser conseguido com um vetor linear de elementos de comparação e inserção, que designaremos por células (ver Figura 9-14).

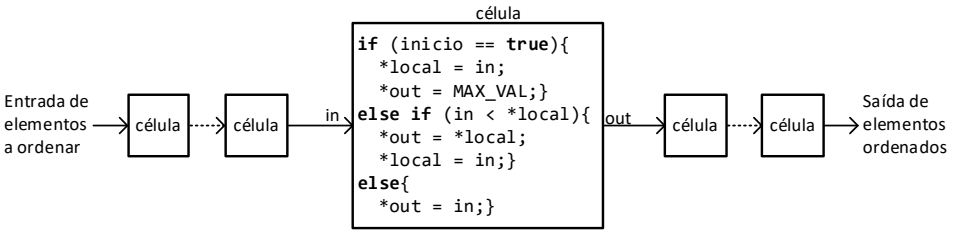


Figura 9-14 – Estrutura linear de elementos de comparação e de inserção para implementação em fluxo de dados do algoritmo de ordenação por inserção.

Todas as células têm a mesma funcionalidade. A célula do meio da figura contém o código da funcionalidade das células, assumindo uma ordenação crescente. Inicialmente (`inicio == True`), a célula guarda o valor de entrada localmente e envia para a célula seguinte o valor máximo possível dos elementos (`MAX_VAL`). Isto permite inicializar todas as células com o valor máximo. Depois, quando a célula recebe um novo valor, compara-o com o valor guardado localmente. Se for menor (`in < *local`), o valor guardado localmente é enviado para a célula seguinte e o novo elemento substitui o valor guardado localmente. Este caso corresponde a inserir o elemento ordenado e a deslocar o valor local para a célula seguinte. Se o valor for maior ou igual, simplesmente envia o valor recebido para a célula seguinte sem alterar o valor local.

Um método similar pode ser considerado para ordenador de forma descendente. Neste caso, a substituição do valor local faz-se quando o novo elemento é maior que o guardado localmente e as células são inicializadas com o menor valor possível.

Após ter movido todos os elementos através da estrutura de células, o maior elemento encontra-se na célula mais à direita. A partir daqui, podem ser lidos de forma ordenada a partir da saída da última célula. No caso de se querer ler a partir do menor elemento, então teria de se ordenar de forma inversa da feita anteriormente, ou seja, de forma decrescente.

Para a especificação do componente hardware usando este método de ordenação, vamos considerar uma função para descrever o subcomponente célula e uma outra função para o componente de topo que instancia tantas células quanto o número de elementos a ordenar. Os elementos a ordenar e os elementos ordenados são transferidos de/para o componente de topo por interfaces *AXI4-Stream*. O número de elementos a ordenar, ou seja, o número de células é um parâmetro fixo da arquitetura. Considera-se ainda que os elementos a ordenar são representados como números inteiros de 32 bits.

Os parâmetros da arquitetura são definidos no ficheiro de cabeçalho (`axis_isort.h`) de acordo com o Código 9-24.

```
#include <ap_int.h>
#include <hls_stream.h>
#include <ap_axi_sdata.h>

typedef hls::axis<ap_int<32>, 0, 0, 0> strmio32_t;

#define SIZE 64
#define MIN_VAL (-1000)
#define MAX_VAL 1000
```

Código 9-24 – Definição dos parâmetros de definição da arquitetura de implementação do algoritmo de ordenação (`axis_isort.h`).

O parâmetro `SIZE` especifica o número de células, e os parâmetros `MIN_VAL` e `MAX_VAL` indicam os valores mínimo e máximo, entre os quais se encontram os valores dos elementos a ordenar. Neste exemplo, consideram-se como valores mínimo e máximo, -1000 e 1000, respetivamente.

A descrição da função da célula (ver Código 9-25) segue o algoritmo indicado na Figura 9-14.

```
void cell(ap_int<32> in, ap_int<32> *local, ap_int<32> *out, bool i)
{
#pragma HLS INLINE
	if (i == true) { *local = in; *out = MAX_VAL; }
	else if (in < *local) { *out = *local; *local = in; }
	else { *out = in; }
}
```

Código 9-25 – Função HLS da célula de ordenação

A função tem como argumentos os elementos de entrada e de saída e um *booleano* a indicar se se trata do primeiro elemento a ordenar. Isto permite inicializar o registo local da primeira célula com o primeiro valor da lista e as restantes células com o valor máximo. Adicionalmente, apesar do valor guardado localmente ser interno a cada célula, o vetor das variáveis locais das células é declarado fora da função da célula. Este procedimento facilita a geração das células dentro de uma estrutura de repetição.

Por fim, a função de geração da estrutura de células interligadas em sequência é descrita de acordo com o *Código 9-26*.

```
#include “axis_isort.h”

void cell(ap_int<32> in, ap_int<32> *local, ap_int<32> *out, bool i){
	(...)
}

void axis_isort( hls::stream<strmio32_t> &strm_in,
                 hls::stream<strmio32_t> &strm_out )
{
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=strm_in
#pragma HLS INTERFACE axis port=strm_out

ap_int<32> regs[SIZE];
#pragma HLS ARRAY_PARTITION variable=regs type=complete
strmio32_t tmp;

bool first = true;
LE_ELEM: for (int m = 0; m < 2*SIZE; m++) {
#pragma HLS PIPELINE II=1
	ap_int<32> val[SIZE+1];

	if (m < SIZE){
		tmp = strm_in.read();
		val[0] = tmp.data;
	}
	else{
		val[0] = MIN_VAL;
		tmp.keep = 0xF;
		tmp.strb = 0xF;
		tmp.data = regs[SIZE-1];
		tmp.last = (m == (2*SIZE -1)) ? (ap_uint<1>)1 : (ap_uint<1>)0;
		strm_out.write(tmp);
	}
	GERA_CELL: for (int k=0; k<SIZE; k++) {
		cell(val[k], &(regs[k]), &(val[k+1]), first);
	}
	if (first == true) first = false;
}}
```

Código 9-26 – Função de descrição da estrutura de células de ordenação em sequência

A função tem um `FOR` exterior (`LE_ELEM`) que controla o número de elementos lidos e escritos e um interior (`GERA_CELL`) que gera a sequência de células interligadas. A estrutura `FOR` exterior consiste em `2×SIZE` iterações, sendo que as primeiras `SIZE` iterações colocam os elementos na ordem certa e as segundas transmitem a lista ordenada.

Durante a fase de ordenação do `FOR` exterior, é feita a leitura de um elemento, seguida da sua ordenação com o `FOR GERA_CELL`. O ciclo interno é desenrolado por completo para gerar a interligação de `SIZE` células, em que a saída de cada célula é a entrada da seguinte.

Na fase de envio de dados do `FOR` exterior, são introduzidos novos valores na lista com o valor mínimo. Isto força o deslocamento dos valores locais das células, valores ordenados,

que vão saindo pela célula mais à direita. O valor da célula mais à direita é enviado pela interface *AXI4-Stream*.

Como descrito anteriormente, a implementação do algoritmo de ordenação com N células permite realizar N comparações em paralelo. Cada elemento a ordenar entra, assim, diretamente na posição certa. No fim, os elementos são lidos um a um. Considerando que a leitura e a escrita são feitas num ciclo de relógio, a latência total teórica do circuito, `latS`, para N células é dada por:

$$\texttt{latS} = 2 \times N$$

9.8.3 Resultados de Implementação

A Tabela 9-8 apresenta os recursos utilizados após síntese e as latências totais para várias implementações do circuito de ordenação com diferentes quantidades de células, em função do número de elementos a ordenar.

Elementos	Recursos			Latência (ciclos)
	LUT	FF	SRL	
8	599	446	0	19
16	1189	742	0	36
32	2409	1366	2	71
64	4823	2577	2	140
128	8263	5008	2	279
256	15965	9901	4	556

Tabela 9-8 – Utilização de recursos e latência total do circuito de ordenação para diferentes números de elementos e com um período de relógio de 10 ns.

Os resultados confirmam o esperado. A latência varia de acordo com a equação `latS1` e os recursos aumentam proporcionalmente com o número de células.

9.9 Rede Neuronal Convolucional

As redes neuronais convolucionais (CNN – *Convolutional Neural Network*) são um algoritmo recente da área de inteligência artificial bastante utilizado a para análise e a classificação de imagens, deteção de objetos, segmentação de imagens, etc. As CNN são uma classe particular das redes neuronais. As redes neuronais são formadas por camadas de neurónios: uma camada de entrada, uma camada de saída e várias camadas escondidas (ver Figura 9-15).

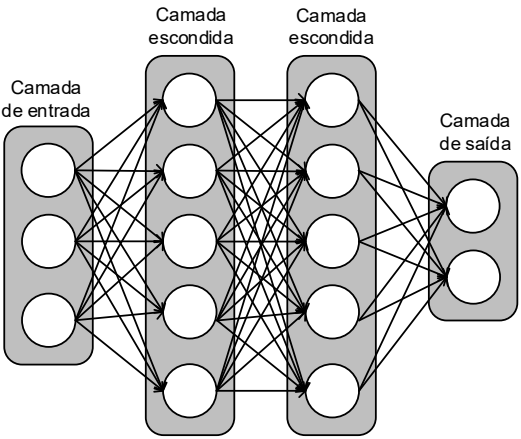


Figura 9-15 – Rede Neuronal.

Neurónios (nós) de uma camada ligam aos neurónios das camadas anteriores e posteriores, exceto os das camadas de entrada e de saída, que apenas ligam aos posteriores e aos anteriores, respetivamente. Um neurónio recebe *n* entradas de *n* neurónios da camada anterior {x1, x2, x3,…, xn}. Cada ligação entre neurónios tem um peso {p_1, p_2, p_3,…, p_n} (ver Figura 9-16).

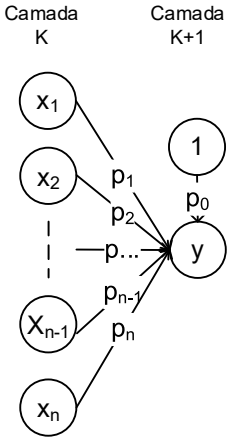


Figura 9-16 – Ligações de um neurónio.

A saída do neurónio é definida como:

$$y = f\left(p_0 + \sum_{k=1}^{n} p_k x_k\right)$$

Em que `f` é uma função de ativação não linear e p_0 é um valor constante adicionado ao resultado da convolução com o filtro (*bias*).

A rede neuronal convolucional considera uma variação das camadas de neurónios de modo a reduzir o processamento no cálculo dos neurónios da camada seguinte. Uma rede neuronal convolucional inclui as designadas *camadas convolucionais*. Esta camada considera a informação espacial entre pixels, ou seja, a saída de um neurónio é o resultado da convolução entre um pequeno subconjunto da imagem ou mapa de entrada e um *kernel* com os pesos. Por exemplo, se consideramos um *kernel* de 5×5, um neurónio recebe precisamente 5×5 pixels do mapa de entrada. No conjunto, uma camada convolucional recebe vários mapas de entrada e gera vários mapas de saída. Cada um dos mapas de saída resulta da aplicação de um filtro 3D (formado por vários *kernels*) ao conjunto dos mapas de entrada (ver Figura 9-17).

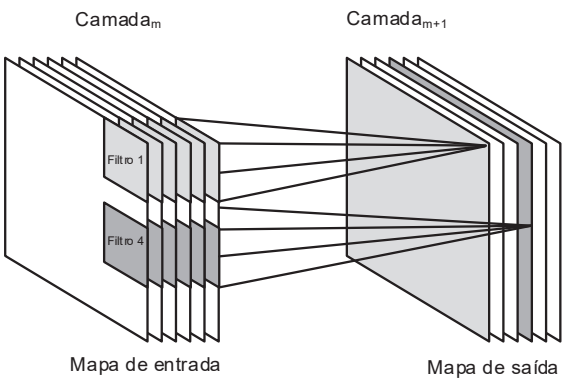


Figura 9-17 – Camada convolucional.

Como se pode observar pela figura, cada mapa ou canal de saída é obtido através da convolução de um filtro 3D com os mapas de entrada. A convolução com um filtro 3D pode ser implementada por aplicação da convolução 2D, apresentada na secção 6, a cada um dos mapas seguida da acumulação do resultado das convoluções.

Uma rede neuronal convolucional pode ter várias camadas convolucionais interligadas entre si, em que os mapas de saída de uma são os mapas de entrada de outra. Estas camadas são utilizadas para extrair características da imagem de entrada.

Para além das camadas convolucionais, as CNN podem incluir outro tipo de camadas, como as camadas de redução e as camadas densas ou totalmente ligadas.

A camada de redução permite reduzir o tamanho dos mapas por aplicação de uma janela 2D que reduz o número de pontos do mapa.

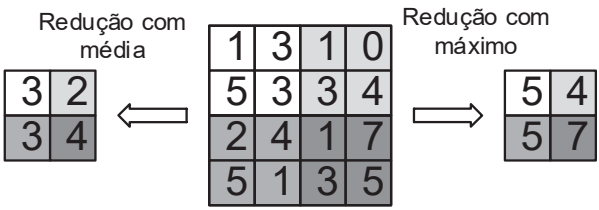


Figura 9-18 – Camada de redução.

O exemplo da Figura 9-18 mostra dois exemplos de redução, em que é determinado o valor médio (redução com média) ou o máximo (redução com máximo) para cada janela 2×2 do mapa. Assim, o mapa de entrada de 4×4 é reduzido a um mapa de 2×2. A redução dos mapas de entrada conduz a uma redução do número de operações necessários para o cálculo da rede neuronal e permite também obter invariância de translação na imagem.

As camadas densas ou totalmente ligadas são idênticas às camadas tradicionais das redes neuronais, em que todos os neurónios estão ligados a todos os neurónios da camada anterior. São utilizadas, tipicamente, para classificar as informações extraídas pelas camadas convolucionais.

9.9.1 CNN para Classificação de Dígitos

Para exemplificar a descrição de uma rede CNN, vamos considerar uma rede simples para classificação de dígitos com imagens de 28×28. A rede recebe a imagem e gera à saída a classe a que pertence o dígito presente na imagem.

A Figura 9-19 ilustra quatro imagens de exemplos de dígitos a serem classificados pela rede CNN ilustrada na Figura 9-20.

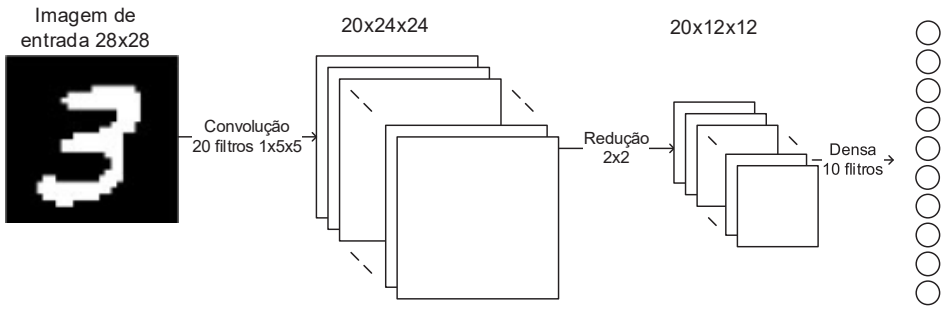


Figura 9-20 – Rede CNN ilustrativa para classificação dos dígitos.

A rede tem uma camada convolucional, uma camada de redução e uma camada totalmente ligada. Na camada convolucional, são realizadas convoluções com 20 filtros 1x5x5 adicionadas do valor de *bias*. O resultado da convolução de um filtro por uma imagem de 28x28 é um mapa de 24x24, uma vez que não se considera *padding*. Na camada totalmente ligada são realizados produtos internos com conjuntos de 2880 (20x12x12) pesos e é somada a constante *bias*.

A rede foi previamente treinada e os valores obtidos para os pesos de todas as camadas foram convertidos de vírgula flutuante para vírgula fixa. Os dados da rede têm os seguintes formatos:

- Os pixels da imagem estão representados a 8 bits entre [0, 255] e estão guardados por linhas;
- Os pesos variam entre]-1, 1[e estão representados com 8 bits em vírgula fixa com um bit para a parte inteira.

Os pesos estão guardados em ficheiros por filtros de acordo com os formatos seguintes:

- Filtros da camada convolucional: constante 0, filtro 0, constante 1, filtro 1, etc. Após cada conjunto (constante, filtro) com 25 bytes, é feito um ajuste para obter um total de bytes múltiplo de oito (64 bits) acrescentando zeros. Isto facilita a leitura dos dados através de uma interface a 64 bits;
- Os pesos da camada densa: constante 0, constante 1, …, filtro 0, filtro 1, etc.

9.9.2 Execução Paralela das Camadas da Rede

Os algoritmos de cálculo das camadas convolucional e densa exibem bastantes operações que podem ser calculadas em paralelo. Na camada convolucional podem ser exploradas as seguintes formas de paralelismo:

- Paralelismo entre filtros: o cálculo dos mapas de saída para cada um dos filtros pode ser feito em paralelo. O cálculo de cada mapa é independente, pois corresponde à aplicação de um filtro diferente à mesma imagem de entrada;

- Paralelismo por filtro: o cálculo de vários elementos do mesmo mapa de saída é feito em paralelo;
- Paralelismo do filtro: várias ou todas as multiplicações necessárias ao cálculo de uma convolução com um filtro podem ser feitas em paralelo.

No caso da camada densa, pode considerar-se o paralelismo entre filtros e o paralelismo do filtro.

O fator de paralelismo para cada uma das formas de paralelismo estabelece um compromisso entre os recursos hardware necessários e a latência. A escolha irá assim depender dos recursos disponíveis e da latência que se pretende para o circuito.

Para analisar estes aspetos, a especificação do componente CNN, apresentada na secção seguinte, irá permitir a configuração dos fatores de paralelismo entre filtros, além de considerar o paralelismo do filtro.

9.9.3 Especificação do Componente CNN

O componente CNN será especificado de acordo com o diagrama de blocos ilustrado na Figura 9-21.

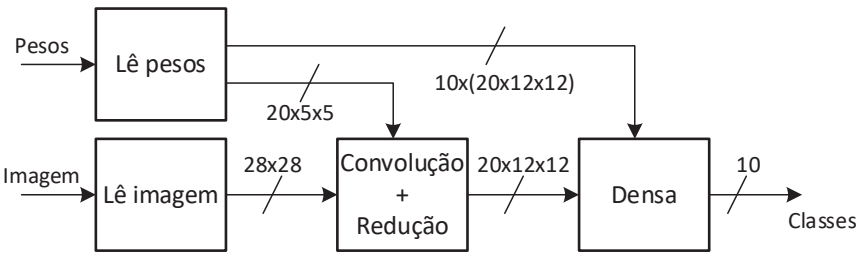


Figura 9-21 – Diagrama de blocos da implementação da rede CNN.

Os primeiros dois blocos leem os pesos e a imagem, depois é feito o cálculo da camada convolucional seguido ca camada de redução e, por fim, da camada densa. A camada convolucional e a de redução são implementadas com um único bloco. A figura indica ainda o volume de dados transferido entre cada um dos blocos. Por exemplo, o bloco *Lê pesos* transfere 20 filtros de 5×5 para o bloco *Convolução+Redução* e 10 filtros de 20×12×12 organizados como um vetor único de 2880 elementos. A especificação dos vários blocos é descrita nas subsecções seguintes.

Para a descrição do componente CNN começou-se por elaborar o ficheiro de cabeçalho (`cnn.h`) com definições das características da rede (ver Código 9-27).

```
#include <ap_int.h>
#include <ap_fixed.h>
#include <hls_stream.h>
#include <ap_axi_sdata.h>
```

```
#define W_BIT_WIDTH 8
#define I_BIT_WIDTH 8

#define K_SIZE 5

#define IWIDTH     28
#define IHEIGHT    28
#define OHEIGHT IHEIGHT-(K_SIZE-1)
#define OWIDTH  IWIDTH-(K_SIZE-1)
#define SREG_SIZE (K_SIZE+IWIDTH*(K_SIZE-1))
#define PARCONV     1

#define NUM_FILTERS 20
#define NUM_CLASSES 10

typedef hls::axis<ap_uint<8>, 0, 0, 0> strmio8_t;
typedef hls::axis<ap_uint<64>, 0, 0, 0> strmio_t;

typedef ap_fixed<1,W_BIT_WIDTH> weigth_t;
typedef ap_ufixed<1,I_BIT_WIDTH> imap_t;
```

Código 9-27 – Descrição das características da rede no ficheiro de cabeçalho (`cnn.h`).

O cabeçalho inclui a definição do tamanho da imagem de entrada e de saída, do filtro da camada convolucional, o número de filtros da camada convolucional e da camada densa, o tipo de dados das interfaces *AXI4-Stream* e o tipo de dados das ativações e dos pesos.

9.9.3.1 Lê Imagem

A imagem é lida e guardada em memória interna. Assume-se que o dispositivo tem memória suficiente para guardar toda a imagem. Caso contrário, teria de ser processada por partes. Considera-se uma interface *AXI4-Stream* de 64 bits, pelo que é possível ler 8 pixels por cada transferência (ver Código 9-28).

```
void ReadImage(ap_ufixed<8,0> image[IHEIGHT*IWIDTH],
               hls::stream<strmio_t> &strm_in) {

#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=strm_in

strmio_t tmp;

LE_IMAGEM: for (int i = 0; i < IHEIGHT*IWIDTH/8; i++) {
      tmp = strm_in.read();
      for(int k = 0; k < 8; k++)
            image[i*8+k].range(7,0) = tmp.data.range((k+1)*8-1,k*8);
}}
```

Código 9-28 – Funções HLS de leitura da imagem no exemplo da CNN

Os dados são lidos em grupos de 64 bits e guardados na memória interna organizados de modo a permitir o acesso a cada byte da imagem.

9.9.3.2 Lê Pesos

Os pesos também são lidos e guardados em memória interna. A leitura dos pesos tem de ter em conta os formatos com que são guardados nos ficheiros, como descrito anteriormente. Considera-se igualmente uma interface *AXI4-Stream* de 64 bits, pelo que é possível ler 8 pesos por cada transferência (ver Código 9-29).

```
void ReadWeights(ap_fixed<8,1>  wConv[NUM_FILTERS][K_SIZE*K_SIZE+1],
                 ap_uint<64>  wFC[NUM_CLASSES][NUM_FILTERS*OHEIGHT*OWIDTH/4/8],
                 ap_fixed<8,1>  biasFC[NUM_CLASSES],
                 hls::stream<strmio_t> &strm_in) {

#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=strm_in
#pragma HLS ARRAY_PARTITION variable=wConv type=complete dim=2

strmio_t tmp;

LE_FILTROS_CONV: for (int i = 0; i < NUM_FILTERS; i++) {
       LE_PESOS_C: for (int j = 0; j < (K_SIZE*K_SIZE+7)/8; j++) {
              tmp = strm_in.read();
              if (j == (K_SIZE*K_SIZE+7)/8 - 1){
                     wConv[i][j*8].range(7,0) = tmp.data.range(7,0);
                     wConv[i][j*8+1].range(7,0) = tmp.data.range(15,8);
              }
              else
                  RDConvin: for(int k = 0; k < 8; k++)
                     wConv[i][j*8+k].range(7,0) = tmp.data.range((k+1)*8-1,k*8);
       }}

tmp = strm_in.read();
bias0: for(int k = 0; k < 5; k++)
       biasFC[k].range(7,0) = tmp.data.range((k+1)*8-1+24,k*8+24);
tmp = strm_in.read();
bias1: for(int k = 0; k < 5; k++)
       biasFC[k+5].range(7,0) = tmp.data.range((k+1)*8-1+24,k*8+24);

LE_FILTROS_FC: for (int i = 0; i < NUM_CLASSES; i++) {
       LE_PESOS_FC: for (int j = 0; j < (NUM_FILTERS*OHEIGHT*OWIDTH/4)/8; j++) {
              tmp = strm_in.read();
              wFC[i][j].range(63,0) = tmp.data.range(63,0);
       }
}}
```

Código 9-29 – Funções HLS de leitura dos pesos no exemplo da CNN

A função começa por ler os pesos convolucionais. São lidos em blocos de oito e guardados um a um para permitir o acesso a um peso de cada vez. No caso dos pesos da camada densa, são lidas inicialmente as constantes dos filtros e seguidamente são lidos os pesos. Neste caso, são guardados em grupos de oito pesos.

9.9.3.3 Convolução e Redução

A convolução da imagem com os filtros segue a implementação da convolução 2D descrita na secção 9.6 (ver Código 9-30).

A função lê os pixels da imagem através de um registo de deslocamento (`SR`) de modo a implementar o algoritmo de convolução. Para cada janela de pesos são aplicados `PARCONV` filtros, em que `PARCONV` especifica o fator de paralelismo entre filtros. Para isso, o ciclo `FILTROS` vai ser completamente desenrolado. O desenrolamento é feito de forma automática resultado da diretiva de *pipeline* no ciclo exterior.

```
void conv2DwPool (  ap_ufixed<8,0>  image[IHEIGHT*IWIDTH],
                    ap_fixed<8,1>   weights[NUM_FILTERS][K_SIZE*K_SIZE+1],
                    ap_fixed<8,4>   outMap[NUM_FILTERS][OHEIGHT*OWIDTH/4])
{
#pragma HLS ARRAY_PARTITION variable=weights type=complete dim=2
#pragma HLS ARRAY_PARTITION variable=weights type=cyclic factor=t_parconv dim=1
#pragma HLS ARRAY_PARTITION variable=outMap type=cyclic factor=t_parconv dim=1

strmio_t tmp1, tmp2;
ap_ufixed<8,0> hwin[SREG_SIZE], in_val;
ap_fixed<21,6> out_val;
int i, j, k, l;
int pos, index, row, col;
ap_fixed<8,4> maxP[NUM_FILTERS][OWIDTH/2];

#pragma HLS ARRAY_PARTITION variable=hwin type=complete
#pragma HLS ARRAY_PARTITION variable=maxP type=cyclic factor=t_parconv dim=1
#pragma HLS DEPENDENCE variable = maxP inter false
#pragma HLS DEPENDENCE variable = outMap inter false

for (int m = 0; m < NUM_FILTERS/PARCONV; m++){
index = 0;
HEIGHT: for(pos = 0, row = 0; row < IHEIGHT; row++) {
    WIDTH: for(col = 0; col < IWIDTH; pos++, col++) {
    #pragma HLS PIPELINE

       in_val = image[row*IWIDTH + col];

       SR: for(i = 0; i < SREG_SIZE; i++)
           hwin[i] = (i == SREG_SIZE-1) ? in_val : hwin[i+1];

       FILTROS: for (k = 0; k < PARCONV; k++){
          if (pos >= SREG_SIZE-1 && col >= K_SIZE-1) {
              HCONV: for(i = 0; i < K_SIZE; i++) {
                  VCONV: for(j = 0; j < K_SIZE; j++) {
                       if (i == 0 && j == 0)
                            out_val = weights[k+m*PARCONV][0] + hwin[i*IWIDTH+j] *
                            weights[k+m*PARCONV][i*K_SIZE+j+1];
                       else
                            out_val = out_val + hwin[i*IWIDTH+j] *
                            weights[k+m*10][i*K_SIZE+j+1];
                    }
             }
```

```
                if (col[0] == 0 && row[0] == 0){
                    maxP[k+m*PARCONV][col.range(4,1)] = (ap_fixed<8,4>)out_val;
                }
                else{
                    if            (maxP[k+m*PARCONV][col.range(4,1)]           <
(ap_fixed<8,4>)out_val)
                      maxP[k+m*PARCONV][col.range(4,1)] = (ap_fixed<8,4>)out_val;
                }
                if (col[0] == 1 && row[0] == 1){
                    outMap[k+m*PARCONV][index]                                  =
maxP[k+m*PARCONV][col.range(4,1)];
                    if (k == (PARCONV - 1))
                        index = index + 1;
                }
            }
          }
        }
      }
}}}
```

Código 9-30 – Função do cálculo das camadas convolucional e de redução no exemplo da CNN

A função integra o cálculo da camada de redução juntamente com a camada convolucional. Esta opção de projeto evita a utilização de memória para guardar a saída do cálculo convolucional, guardando apenas o mapa reduzido após a camada de redução. Além disso, o cálculo da redução fica integrado na *pipeline* do circuito de cálculo da camada convolucional, melhorando a latência final. Para implementar a camada de redução, é utilizado um vetor (`maxP`) que guarda os valores máximos das janelas de redução à medida que se vão calculando as convoluções. Após concluir o cálculo do máximo dos quatro valores da janela, o resultado é escrito no mapa de saída.

Assim, após o cálculo da convolução 5×5, o resultado é comparado com os elementos da janela de redução de modo a calcular o valor máximo. O valor máximo da janela de redução é finalmente guardado em memória para ser lido pela camada densa.

O processo repete-se `NUM_FILTERS/PARCONV` vezes. A variação do parâmetro `PARCONV` determina o fator de paralelismo. Por exemplo, com `PARCONV` = 2, são calculados de cada vez dois mapas de saída em paralelo. Para 20 filtros no total, o processo repete-se 10 vezes. É importante observar que para permitir a execução paralela é necessário um número de portos suficientes para acesso aos vetores. Para isso, é feita a partição dos vetores com um fator proporcional (`t_parconv`).

Além do paralelismo entre filtros, a função também considera o paralelismo do filtro, em que as 25 multiplicações da convolução são calculadas em paralelo.

9.9.4 Resultados de Implementação das Camadas

A Tabela 9-9 apresenta os resultados de ocupação de recursos e de latência do módulo convolucional para diferentes fatores de paralelismo (PARCONV).

PARCONV	Recursos				Latência (ciclos)
	LUT	FF	BRAM	DSP	
1	446	349	0	25	15,686
2	553	348	0	50	7846
5	1018	1067	0	139	3145
10	1817	1605	0	258	1577
20	3503	4750	0	500	791

Tabela 9-9 – Utilização de recursos e latência total do circuito do módulo de cálculo da convolução para diferentes fatores de desenrolamento do ciclo de filtros.

Como esperado, a quantidade de recursos aumenta com o fator de paralelismo, em particular, o número de multiplicadores aumenta proporcionalmente. A latência diminui com o fator de paralelismo, mostrando que a solução é escalável com este fator.

9.9.4.1 Camada Densa

O cálculo da camada densa consiste no produto interno entre todos os elementos dos mapas de saída com 10 filtros diferentes. O resultado é um vetor de 10 elementos (ver Código 9-31).

```
void FClayer(ap_fixed<8,4> map[NUM_FILTERS][OHEIGHT*OWIDTH/4],
             ap_uint<64> wFC[NUM_CLASSES][NUM_FILTERS*OHEIGHT*OWIDTH/4/8],
             ap_fixed<8,1> bias[NUM_CLASSES],
             hls::stream<strmio_t> &strm_out) {

#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=strm_out
#pragma HLS ARRAY_PARTITION variable=wFC type=cyclic factor=1 dim=1
#pragma HLS ARRAY_PARTITION variable=map type=cyclic factor=8 dim=2
#pragma HLS ARRAY_PARTITION type=cyclic variable=bias factor= 1

strmio_t tmp;
ap_uint<64> auxW;
ap_fixed<22,10> acc[NUM_CLASSES];
ap_fixed<8,1> aux;
ap_fixed<8,4> aux1;
#pragma HLS ARRAY_PARTITION type=cyclic variable=acc factor= 1

LOADBIAS: for (int i = 0; i < NUM_CLASSES; i++)
      acc[i] = bias[i];

L_FILTERS: for (int j = 0; j < NUM_FILTERS; j++){
      L_W: for (int k = 0; k < (OHEIGHT*OWIDTH/4/8); k++){
            FCLAYER:for (int i = 0; i < NUM_CLASSES; i++) {
             #pragma HLS UNROLL factor = 1
                  auxW = coeffsFC[i][j*(OHEIGHT*OWIDTH/4)/8+k];
```

```
                    for (int l = 0; l<8; l++){
                    #pragma HLS UNROLL
                           aux.range(7,0) = auxW.range((l+1)*8-1,l*8);
                           aux1.range(7,0) = map[j][k*8+l].range(7, 0);
                           acc[i] += aux1 * aux;
                    }
             }
       }
}

SAVE:for (int i = 0; i < NUM_CLASSES; i++) {
       tmp.data.range(21,0) = acc[i].range(21,0);
       if (i==(NUM_CLASSES-1)) tmp.last = (ap_int<1>)1;
       else tmp.last = (ap_int<1>)0;
       strm_out.write(tmp);
}
}
```

Código 9-31 – Função HLS do cálculo da camada densa no exemplo da CNN

A função começa por inicializar os 10 acumuladores dos produtos internos com os valores das constantes dos filtros (`LOADBIAS`). De seguida, calcula o produto interno usando as duas formas de paralelismo da camada densa. Para cada filtro são calculadas 8 multiplicações em paralelo e são calculados vários filtros em paralelo (`t_parfc`). Por fim, o vetor final é enviado pela interface *AXI4-Stream* (`SAVE`).

A Tabela 9-10 apresenta os resultados de ocupação de recursos e de latência do módulo da camada densa para diferentes fatores de paralelismo (`PARCONV`).

PARFC	Recursos				Latência (ciclos)
	LUT	FF	BRAM	DSP	
1	353	268	0	8	3639
2	555	337	0	15	1834
5	962	660	0	36	751
10	967	435	0	80	381

Tabela 9-10 – Utilização de recursos e latência total do circuito do módulo de cálculo da camada densa para diferentes fatores de desenrolamento do ciclo de classes.

O número de DSP aumenta e a latência diminui proporcionalmente com o fator de paralelismo.

9.9.4.2 Função de Topo do Componente CNN

A função de topo descreve a arquitetura da Figura 9-21 com a interligação dos módulos descritos anteriormente (ver Código 9-32).

```
#include “cnn.h”

void ReadWeights(...) {...}
void ReadImage(...) {...}
void FCLayer(...) {...}
void Conv2DwPool(...) {...}

void cnn (hls::stream<strmio_t> &strm_in,
          hls::stream<strmio_t> &strm_out) {

#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=strm_in
#pragma HLS INTERFACE axis port=strm_out

ap_fixed<8,1> wConv[NUM_FILTERS][K_SIZE*K_SIZE+1];
ap_uint<64> wFC[NUM_CLASSES][NUM_FILTERS*OHEIGHT*OWIDTH/4/8];
ap_fixed<8,1> biasFC[NUM_CLASSES];
ap_ufixed<8,0> image[IHEIGHT*IWIDTH];
ap_fixed<8,4> outMap[NUM_FILTERS][OHEIGHT*OWIDTH/4];

ReadWeights(wConv, wFC, biasFC, strm_in);

ReadImage(image, strm_in);

conv2DwPool(image, wConv, outMap);

FClayer(outMap, wFC, biasFC, strm_out);
}
```

Código 9-32 – Função HLS de topo que executa o modelo da CNN

Começa por ler os pesos, seguido da leitura da imagem. Depois, calcula a camada convolucional e a de redução seguida da camada densa. A implementação pode ser otimizada adicionalmente se considerarmos que os pesos são sempre os mesmos para todas as imagens. Assim, os pesos seriam carregados na inicialização do circuito e manter-se-iam constantes durante todo o funcionamento do circuito. Depois, por cada processo de classificação, seria apenas necessário carregar a imagem e executar as camadas.

9.9.5 Resultados de Implementação do Circuito Completo

Implementaram-se dois circuitos com fatores de paralelismo entre filtros diferentes: uma com fatores de paralelismo unitários para ambas as camadas e outra com fatores de paralelismo máximo. A Tabela 9-11 apresenta a latência das duas soluções e a Tabela 9-12 apresenta os recursos utilizados.

As duas tabelas mostram os casos extremos de compromisso entre desempenho e ocupação de recursos hardware. Existem, naturalmente, soluções intermédias quando se adotam outros fatores intermédios de paralelismo. A melhor solução depende sempre dos requisitos específicos de cada projeto.

Módulo	Latência (ciclos)	
	Paralelismo unitário	Paralelismo máximo
Read Weights	3690	3690
Read Images	392	392
Conv2D	15686	790
FCLayer	3638	380
Total	**23406**	**5252**

Tabela 9-11 – Latência total do circuito para dois fatores de paralelismo entre filtros.

Fator de Paralelismo	Recursos			
	LUT	FF	BRAM	DSP
Unitário	2386	1160	1	34
Máximo	12650	10015	21	580

Tabela 9-12 – Utilização de recursos do circuito para dois fatores de paralelismo entre filtros.

9.10 Algoritmo de Agrupamento – *K-Means*

A análise de dados com o objetivo de extrair informação relevante é um processo fundamental em diversas áreas, como a financeira, a saúde, etc. Tipicamente, a partir de um conjunto de dados, procura-se agrupar os dados com base nas características dos próprios dados.

O *K-Means* é um algoritmo de agrupamento bastante utilizado, não só pela sua simplicidade, mas também pelos bons resultados, que permite encontrar padrões e semelhanças entre dados sem supervisão.

O algoritmo distribui os elementos do conjunto de dados por grupos com base em métricas de semelhança. Considerando K grupos, o algoritmo gera K grupos e distribui cada um dos elementos do conjunto de dados pelo grupo que melhor representa o elemento. A representatividade de um grupo ou o centro é designado *centroide*. O centroide de um grupo é determinado pelos elementos que pertencem a esse grupo.

O algoritmo começa com um conjunto de K centroides e distribui os dados pelos centroides formando K grupos de elementos. Para tal, atribui cada um dos elementos ao centroide mais próximo. De seguida, recalcula os centroides com base nos elementos que foram atribuídos ao seu grupo. Isto pode levar a que determinados elementos passem a estar mais próximos de outros centroides. Como tal, reinicia-se outra iteração com nova redistribuição dos elementos e o recálculo dos centroides.

Os valores iniciais dos centroides podem ser determinados de várias formas, sendo o método de atribuição aleatória bastante utilizado. Nesta abordagem, os centroides iniciais são escolhidos aleatoriamente, coincidentes ou não com elementos do grupo de dados.

A similaridade de um ponto relativamente a um centroide é determinada pela distância entre os dois. O algoritmo termina ao fim de um determinado número de iterações ou quando não se verificam alterações nos grupos de uma iteração para outra.

O Código 9-33 descreve genericamente os principais passos do algoritmo *K-Means*.

```
K-Means()
       inicializa os K centroides, c[k]
       enquanto c(n)[k] ≠ c(n+1)[k]:
         para cada elemento:
           calcula a distância a todos os centroides
         atribui cada elemento ao centroide com menor distância
         calcula os novos centroides
       armazena os centros finais
```

Código 9-33 – Passos principais do algoritmo K-Means

O algoritmo tem assim diversos passos:

- Inicialização dos centros ou centroides – de entre os vários métodos, o de inicialização aleatória é bastante utilizado pela sua simplicidade. Os centros podem ser inicializados aleatoriamente ou igualados às posições de elementos dos dados escolhidos aleatoriamente;
- Cálculo iterativo dos centros – faz a distribuição dos pontos pelos grupos, com base na proximidade entre os elementos e os centros de cada grupo, recalcula os centros tendo em conta os elementos que fazem parte do seu grupo e armazena o centro a que pertence cada elemento;
- Armazena os centros finais – após atingir a condição que determina a paragem do processo iterativo, os centros finais são armazenados.

O cálculo iterativo dos centros realiza os seguintes passos:

1. Carrega os pontos – lê pontos da memória;
2. Atribui os pontos aos centros – atribui cada um dos pontos ao centro mais próximo. Para tal, determina a distância de cada ponto a todos os centros e atribui o ponto ao centro que lhe fica mais perto;
3. Recalcula os centros – recalcula as coordenadas dos centros com base nos elementos do seu grupo. Determina a média dos pontos do grupo para todas as dimensões;
4. Guarda os centros por elemento – para cada elemento armazena o valor do centro a que pertence.

A descrição do algoritmo em HLS será assim subdividida de acordo os passos descritos anteriormente (ver Figura 9-22).

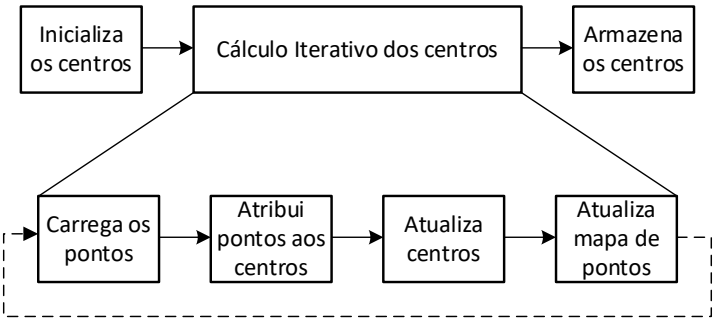


Figura 9-22 – Fluxo de processamento do algoritmo K-Means a considerar na descrição HLS.

A distância, `d(px, cy)`, entre um ponto, `px`, e o centro do grupo ou centroide, `cy`, pode ser calculada de diferentes formas. Uma das mais utilizadas considera a distância euclidiana definida na equação 1, em que se acumula a diferença entre o ponto e o centro para as várias dimensões, `D`, dos elementos.

$$d(p_x, c_y) = \sqrt{\sum_{i=1}^{D}(p_x(i) - c_y(i))^2} \quad (1)$$

A implementação da equação 1 implica o cálculo da raiz quadrada. Sendo utilizada para comparar distâncias, considera-se geralmente o quadrado da distância Euclidiana, $d(p_y, c_y)^2$ que garante os mesmos resultados de comparação.

A utilização da distância euclidiana ao quadrado tem a vantagem de evitar a complexidade associada ao cálculo da raiz quadrada necessária ao cálculo da distância euclidiana, que se obtém aplicando a raiz quadrada à equação 1.

Os novos valores dos centros são determinados com base nos pontos associados ao grupo respetivo. Considerando S pontos associados a um grupo, para cada dimensão é calculada a média dos S pontos na dimensão respetiva de acordo com a equação 2.

$$c_y(i) = \frac{1}{S}\sum_{i=1}^{S}(p_x(i) - c_y(i))^2 \quad Para\ i = 1, 2, \dots, D \quad (2)$$

A equação 2 aplica-se a todos os centroides e a todas as dimensões.

9.10.1 Execução Paralela do Algoritmo K-Means

Em cada iteração do algoritmo, é necessário calcular a distância de todos os pontos a cada um dos centroides para cada uma das dimensões. Assim, o número de operações torna-se bastante elevado quando tempos muitos pontos, centroides e dimensões. Por exemplo, a execução de 100 iterações de agrupamento de 100,000 pontos com 10 dimensões em 10 centroides necessita de calcular 1,000,000,000 de distâncias entre pontos e centros.

Apesar da complexidade do algoritmo, o cálculo das distâncias entre os pontos e os centroides pode ser facilmente paralelizado, pois são operações independentes, permitindo reduzir o tempo de computação. Existem diversas formas de explorar o algoritmo em paralelo:

- Na equação 1, vários ou todos os termos (em cada dimensão) da distância podem ser calculados em paralelo. O máximo do fator de paralelismo possível é igual ao número de dimensões;
- O cálculo da distância entre um ponto e vários ou todos os centroides também pode ser feito em paralelo, permitindo um fator de paralelismo máximo igual ao número de centroides;
- Também é possível calcular a distância em paralelo de um centroide a vários ou a todos os pontos. Neste caso, o fator de paralelismo pode ser bastante mais elevado, pois corresponde ao número de pontos calculados em paralelo.

Sabendo antecipadamente o número de dimensões, de pontos e de centroides, é possível projetar um circuito paralelo otimizado. Apesar de todos os cálculos poderem ser feitos em paralelo, temos de ter em conta que é necessário ler os pontos de uma memória, pelo que a largura de banda de acesso à memória determina o fator de paralelismo que pode ser explorado de forma mais eficiente.

Na prática, o número de dimensões, o número de centroides e o número de pontos variam de acordo com o conjunto de dados que se pretende agrupar. Por isso, é preciso algum cuidado na escolha da melhor forma de paralelização.

De uma maneira geral, a melhor opção será paralelizar pelo número de elementos ou pontos do conjunto de dados, pois o seu número é habitualmente muito maior que o número de centroides e de dimensões. Para melhor percebermos esta abordagem, consideremos que se pretende paralelizar o cálculo das dimensões. A questão que se coloca é a de saber qual o fator de paralelismo que devemos considerar aquando da implementação do acelerador. Se escolhermos um fator de paralelismo superior ao número de dimensões, estamos a contribuir para a ineficiência do acelerador, pois há recursos disponíveis que não são utilizados. Por outro lado, se escolhermos um fator inferior, reduzimos o cálculo em paralelo com o consequente aumento do tempo de computação. Por exemplo, um conjunto de dados em que a dimensão dos elementos é igual a 10 e o fator de paralelismo é igual a 16, leva a que apenas seja possível utilizar 10 das 16 unidades de cálculo em paralelo. Se, por outro lado, tivéssemos um fator de paralelismo igual a 8, seria necessário executar o cálculo em dois passos, sendo que no segundo apenas são utilizadas duas das oito unidades de cálculo em paralelo. Neste caso, a cada dois ciclos de cálculo desperdiça-se a disponibilidade de seis unidades de computação, que ficam paradas.

A mesma ideia aplica-se à paralelização no cálculo da distância de um ponto a todos os centroides, considerando a relação entre paralelismo e número de centros.

No caso de aplicarmos paralelismo no cálculo da distância de um centroide a todos os pontos, verifica-se que, devido ao elevado número de pontos, a eficiência de utilização das unidades de cálculo em paralelo é bastante elevada. Por exemplo, se tivermos 5,000 pontos e um fator de paralelismo de 16, o cálculo das distâncias para o total de pontos é feito em 313 iterações. Nas primeiras 312 iterações, todas as 16 unidades de cálculo são utilizadas em paralelo e apenas na última é que são utilizadas apenas oito unidades de cálculo. Esta forma de exploração do paralelismo é assim a mais eficiente.

As várias formas de paralelismo podem ser exploradas em simultâneo. Por exemplo, calcular em paralelo as distâncias de N pontos a M centroides. Neste caso, o fator de paralelismo é dado por N × M. Naturalmente, que nas iterações finais irá existir alguma ineficiência, exceto se o número de pontos e de centroides em paralelo coincidir com o fator de paralelismo.

No exemplo seguinte, é considerado apenas o paralelismo no cálculo das distâncias entre um conjunto de pontos ao centroide.

9.10.2 Descrição do Componente K-Means Paralelo

O algoritmo *K-Means* é descrito de acordo com o diagrama da Figura 9-22 ilustrada na secção anterior. O primeiro e o último módulo são executados apenas uma vez, enquanto os centroides são recalculados várias vezes de acordo com o algoritmo. O cálculo iterativo dos centroides é feito em paralelo, como indicado na secção anterior, ou seja, com o cálculo paralelo da distância entre N pontos e cada um dos centroides.

Na descrição dos vários componentes, considera-se um ficheiro de cabeçalho com as definições de vários valores e macros, incluindo a especificação do número máximo de pontos, de centros e de dimensões suportado pela implementação. Este valor pode ser alterado em tempo de compilação (ver Código 9-34).

```
#include <ap_int.h>
#include <ap_axi_sdata.h>
#include <hls_stream.h>

#define MAXPOINTS 100000
#define MAXCENTERS 10
#define MAXDIM 2
#define PARPOINTS 64// Pontos processados em paralelo
#define CDSIZE (MAXCENTERS * MAXDIM)
#define MAX_VALUE 0xFFFFFFFF

// valores utilizados na simulação
#define NUMPOINTS 100
#define NUMCENTERS 10
#define NUMDIMS 2

typedef ap_uint<7> index_t;
typedef ap_axis<64, 0, 0, 0> data_t;
```

```
struct point {
    index_t index;
    unsigned long min_dist;
    unsigned long dist;
};

const unsigned int t_ndims = MAXDIM;
const unsigned int t_ncenters = MAXCENTERS;
const unsigned int t_par_points = PARPOINTS;
const unsigned int t_num_iter = MAXPOINTS / PARPOINTS;
const unsigned int t_st_members = PARPOINTS/BUS_SIZE;
```

Código 9-34 – Definição de valores de implementação e macros no ficheiro de cabeçalho (`axis_kmeans.h`)

Cada ponto é descrito com uma estrutura (`point`) que inclui o centro associado (`index`), a distância do ponto a um determinado centro (`dist`) e a distância mínima a um centro (`min_dist`). As constantes que surgem no código são utilizadas para definir limites na diretiva `HLS_TRIP_COUNT` de modo a obtermos estimativas de latência.

Na descrição do componente, assume-se que os pontos são recebidos um a um com os valores de todas as dimensões, ou seja, recebe o valor de todas as dimensões do ponto 1, depois todas as dimensões do ponto 2, etc. Os pontos são depois guardados por dimensão, ou seja, guardam-se os valores da primeira dimensão de todos os pontos, seguido dos valores da segunda dimensão de todos os pontos, etc. Esta organização permite a exploração do paralelismo pelos pontos.

Os centros estão organizados em memória centro a centro, ou seja, guardam-se os valores de todas as dimensões do primeiro centro, seguido dos valores de todas as dimensões do segundo centro, etc.

A especificação dos vários blocos é descrita nas secções seguintes.

9.10.2.1 Inicializa Centros

O módulo de inicialização inclui a função `init_centers` que inicializa os centros e a função `rst_new_centers` que inicializa a 0 o vetor que armazena os novos centros, após cada iteração do *K-Means*, e o contador que armazena o número de pontos atribuídos a cada centro, que também é atualizado após cada iteração (ver Código 9-35).

```
void init_centers(unsigned int l_centers[CDSIZE], hls::stream<data_t> &centers,
ap_uint<7> ncenters, ap_uint<7> ndims) {
       ap_uint<14> nreads = (ncenters * ndims);
    data_t tmp;

    LOAD_CENTERS: for (ap_uint<14> i = 0; i < nreads; i +=2) {
      tmp = centers.read();
      l_centers[i] = (unsigned int)tmp.data.range(31,0);
      l_centers[i+1] = (unsigned int)tmp.data.range(63,32);
    }
}
```

```
void rst_new_centers(
       unsigned int new_centers[PARPOINTS][MAXCENTERS][MAXDIM],
       unsigned int ncenters_cnt[PARPOINTS][MAXCENTERS],
       ap_uint<7> ncenters, ap_uint<7> ndims) {
       #pragma HLS ARRAY_PARTITION factor = par_points variable=new_centers
       cyclic dim=1
           init_centers1: for (ap_uint<7> j = 0; j < ncenters; j++){
              init_centers2: for (ap_uint<7> k = 0; k < ndims; k++){
                     init_centers3: for (ap_uint<7> i = 0; i < PARPOINTS; i++){
                     #pragma HLS UNROLL factor=t_par_points
                            ncenters_cnt[i][j] = 0;
                            new_centers[i][j][k] = 0;
                     }
              }
           }
}
```

Código 9-35 – Funções de inicialização dos centros e dos vetores de armazenamento temporário dos novos centros

No exemplo, os centros são inicializados com os primeiros pontos do conjunto total de pontos a serem agrupados (`LOAD_CENTERS`).

9.10.2.2 Cálculo Iterativo dos Centros

Os vários passos do cálculo iterativo dos centros são encadeados de acordo com uma estrutura de fluxo de dados procurando reduzir o tempo de computação de cada iteração. Uma iteração só inicia após a iteração anterior terminar, uma vez que existe uma dependência em relação aos centros (ver Figura 9-23).

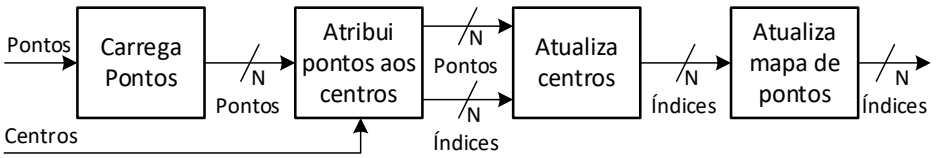


Figura 9-23 – Processamento em fluxo de dados do passo iterativo do K-Means a considerar na descrição HLS.

Os pontos são lidos e processados em blocos de N pontos. Como os blocos são processados em fluxo de dados, cada conjunto de N pontos é processado enquanto o sistema lê os próximos N pontos. Para cada N pontos são gerados N índices que correspondem à identificação dos centros a que foram atribuídos.

Nas subsecções seguintes, são apresentadas as descrições de cada um dos blocos da Figura 9-23.

9.10.2.3 Carrega Pontos

Vamos considerar que as coordenadas dos pontos são representadas com inteiros de 32 bits e que são lidas através de uma interface *AXI4-Stream* de 64 bits. Consegue-se, assim,

ler dois valores por cada leitura de uma palavra de 64 bits. Como referido anteriormente, os pontos são guardados em memória local em grupos de `N` pontos (fator de paralelismo) formando palavras de `N` × 32 bits por cada uma das dimensões (ver Figura 9-24).

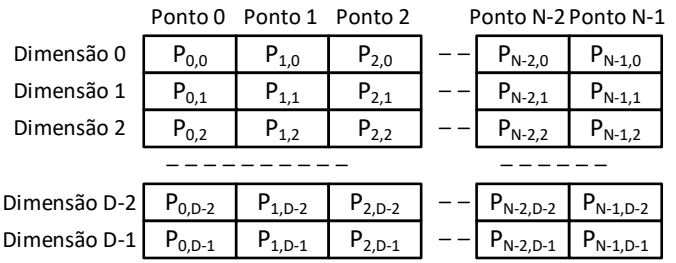


Figura 9-24 – Armazenamento local dos N pontos por dimensão.

Esta organização do armazenamento de pontos permite que seja lida uma determinada dimensão de todos os `N` pontos simultaneamente. O Código 9-36 descreve o módulo para carregamento dos pontos.

```
void load_data(ap_uint<par_points*32> dimPoints[MAXDIM],
       hls::stream<data_t> &data, ap_uint<7> ndims, ap_uint<11> nreads) {

ap_uint<7> f = 0;
ap_uint<7> p = 0;
ap_uint<par_points*32> tmp[MAXDIM];
data_t tmpStr;

LE_DADOS: for (ap_uint<11> i = 0; i < nreads; i+=2, f+=2) {
    if (f == ndims) {
       f = 0;
       p++;
    }

    tmpStr = data.read();
    tmp[f].range(32*p + 31, 32*p) = (unsigned int)tmpStr.data.range(31,0);
    tmp[f+1].range(32*p + 31, 32*p) =
                  (unsigned int)tmpStr.data.range(63,32);
}

for (ap_uint<7> j = 0; j < ndims; j++){
      dimPoints[j] = tmp[j];
}}
```

Código 9-36 – Função do bloco de carregamento de um conjunto de N pontos

A função recebe o número de dimensões dos pontos (`ndims`), o número de leituras (`nreads`) a realizar e lê os N pontos (`par_points`) através do barramento de 64 bits.

9.10.2.4 Atribuição de Pontos aos Centros

A atribuição de pontos é feita igualmente em blocos de `N` pontos. Para cada centro, são determinadas (em paralelo) as distâncias de todos os `N` pontos a esse centro. Quando a

distância de um ponto ao centro é inferior à calculada para os centros anteriores, atualiza-se o centro a associar a esse ponto (ver Código 9-37).

```
void assign_points (ap_uint<7> index_str[PARPOINTS],
                    unsigned int dataP[PARPOINTS][MAXDIM],
                    ap_uint<par_points*32> data[MAXDIM],
                    unsigned int centers[CDSIZE],
                    ap_uint<7> ncenters,
                    ap_uint<7> ndims) {

point pt[PARPOINTS];
#pragma HLS ARRAY_PARTITION factor = par_points variable=pt cyclic dim=1
#pragma HLS ARRAY_PARTITION factor = par_points variable=index_str cyclic
#pragma HLS ARRAY_PARTITION factor = par_points variable=data cyclic dim=1

INIT: for (ap_uint<7> p = 0; p < PARPOINTS; p++) {
#pragma HLS UNROLL
       pt[p].index = 0;
       pt[p].min_dist = MAX_VALUE;
       pt[p].dist = 0;
}

CAL_PONTOS1: for (ap_uint<7> c = 0; c < ncenters; c++) {
       CAL_PONTOS2: for (ap_uint<7> f = 0; f < ndims; f++) {
              CAL_PONTOS3: for (ap_uint<7> p = 0; p < PARPOINTS; p++) {
              #pragma HLS UNROLL
                     int tmp;
                     tmp = dataP[f].range(p*32+31, p*32);
                     int diff = (tmp - centers[c*ndims+f]);
                     pt[p].dist += diff * diff;
                     if (c == (ncenters - 1)) data_str[p][f] = tmp;
              }
       }

       ATUALIZA: for (ap_uint<7> p = 0; p < PARPOINTS; p++) {
       #pragma HLS UNROLL factor = par_points
              if (pt[p].dist < pt[p].min_dist){
                     pt[p].min_dist = pt[p].dist;
                     pt[p].index = c;
                     index_str[p] = c;
              }
              pt[p].dist = 0;
       }
}}
```

Código 9-37 – Função HLS do bloco de atribuição de N pontos aos centros mais próximos

A função recebe o número de centros (`ncenters`) e o número de dimensões dos pontos (`ndims`), e dois vetores: um de pontos e outro de centros. Como saídas, gera dois vetores, um de atribuição dos N pontos aos centros mais próximos (`index_str`), e um outro (`dataP`) que é uma cópia dos dados de entrada organizada em matriz. Os dados são, assim, processados pela função e encaminhados para a função seguinte. O encaminhamento de

dados através da função garante o requisito de um produtor/um consumidor do modelo de fluxo de dados.

O primeiro `FOR` (`INIT`) inicializa os índices e as distâncias dos pontos. De seguida, temos três estruturas `FOR` encadeadas para o cálculo das distâncias mínimas aos centros. Para cada centro (`CAL_PONTOS1`) calcula as distâncias dos N pontos ao centro (`CAL_PONTOS2` e `CAL_PONTOS3`). Calculadas estas distâncias, determina, para cada ponto, se a distância é menor relativamente ao centro anterior. Se for, então atualiza o vetor de atribuição de pontos. O processo termina quando se esgotarem os centros.

O paralelismo é realizado com um desenrolamento completo das estruturas `FOR` `CAL_PONTOS3` e `ATUALIZA`. Deste modo, são calculadas N distâncias por cada dimensão em paralelo.

9.10.2.5 Atualização dos Centros

A atualização dos centros consiste em acumular em cada centro as distâncias dos pontos que pertencem a esse centro e atualizar o número de pontos que pertencem a um centro. Estes dados são utilizados posteriormente para recalcular a posição dos centros (ver Código 9-38).

```
void update_centers(unsigned int new_centers[PARPOINTS][MAXCENTERS][MAXDIM],
       unsigned int ncenters_cnt[PARPOINTS][MAXCENTERS],
       ap_uint<7> indexOut[PARPOINTS],
       ap_uint<7> index[PARPOINTS],
       unsigned int data[PARPOINTS][MAXDIM],
       unsigned int npoints,
       ap_uint<7> ndims) {

data_t tmp;
ap_uint<7> it;

#pragma HLS ARRAY_PARTITION factor = par_points variable=data cyclic dim=1
#pragma HLS DEPENDENCE variable = new_centers inter false
#pragma HLS DEPENDENCE variable = ncenters_cnt inter false

calcenters_2: for (ap_uint<7> f = 0; f < ndims; f++) {
       calcenters_1: for (ap_uint<7> p = 0; p < PARPOINTS; p++) {
       #pragma HLS UNROLL
              it = index[p];
              if (p < npoints){
                     new_centers[p][it][f] += data[p][f];
                     if (f == ndims-1){
                            ncenters_cnt[p][index[p]]++;
                            indexOut[p] = index[p];
                            }
}}}}
```

Código 9-38 – Função HLS do bloco de atualização dos centros

A função recebe o número de pontos (`npoints`), o número de dimensões dos pontos (`ndims`) e dois vetores: um de pontos (`data`) e um de índices (`index`). O de índices é

utilizado pela função e encaminhado para o próximo bloco. Recebe ainda e altera dois outros vetores: o de contador de pontos dos centros (`ncenters_cnt`) e o de armazenamento dos novos centros (`new_centers`). O vetor de novos centros é utilizado para acumular as distâncias dos pontos aos centros respetivos e o vetor de contador de pontos vai sendo atualizado com o número de pontos atribuídos a cada centro.

Um aspeto a realçar na implementação destes vetores é o facto de se ter considerado um acumulador por cada ponto do subconjunto de N pontos processados em cada iteração, em vez de usar um acumulador por cada centro. Para identificar a diferença destas duas possíveis implementações, comecemos por considerar o caso em que todos os N pontos são atribuídos a centros diferentes. Neste caso, em ambas as soluções, todas as acumulações podem ocorrer no mesmo ciclo de relógio. No entanto, se todos os pontos forem atribuídos ao mesmo centro, teremos desempenhos diferentes entre as duas implementações. No caso de termos acumuladores nos centros, os pontos têm de ser acumulados em sequência ao mesmo centro, obtendo um tempo de acumulação proporcional ao número de pontos a somar, não tirando partido do fator N de paralelismo. Por outro lado, se considerarmos um acumulador por cada ponto do subconjunto de N pontos, as acumulações podem ser feitas em paralelo. Neste caso, a implementação mantém a escalabilidade com o número N de pontos.

Neste exemplo, consideram-se N acumuladores, mas este paralelismo pode ser otimizado considerando apenas os acumuladores necessários para garantir um tempo de execução próximo do tempo de execução dos outros blocos do fluxo de dados.

9.10.2.6 Envio do Mapa de Pontos

O envio do mapa de pontos consiste apenas em guardar em memória o centro a que pertence cada ponto (ver Código 9-40).

```
void store_memberships(hls::stream<data_t> &membership,
                       ap_uint<7> index[PARPOINTS],
                       ap_uint<7> nmem_writes, ap_uint<1> last) {

#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=membership

data_t tmp;

st_members: for (ap_uint<7> i = 0; i < nmem_writes; i +=2) {
#pragma HLS PIPELINE
      tmp.data.range(31,0) = index[i];
      tmp.data.range(63,32) = index[i+1];
      tmp.last = last;
      tmp.keep=0xFF;
      tmp.strb=0xFF;
      membership.write(tmp);
}}
```

Código 9-39 – Função do bloco de atualização do mapa de pontos

A função recebe o índice do centro a que pertence cada ponto (`index[PARPOINTS]`) e envia-o por uma interface *AXI4-Stream*. Também aqui se aproveita o tamanho da interface para enviar dois valores por cada transferência.

9.10.2.7 Armazena Centros

O módulo final calcula os novos centros e envia pela interface *AXI4-Stream*. Para tal, para cada centro soma os resultados dos vários acumuladores e divide-os pelo número de pontos atribuídos ao centro respetivo (ver Código 9-40).

```
void store_centers(hls::stream<data_t> &new_centers,
                   unsigned int new_centersIn[PARPOINTS][MAXCENTERS][MAXDIM],
                   unsigned int ncenters_cnt[PARPOINTS][MAXCENTERS],
                   ap_uint<7> ncenters, ap_uint<7> ndims) {

data_t tmp;
unsigned int accenters[MAXCENTERS][MAXDIM];
unsigned int accenter_cnt[MAXCENTERS];

GUARDA_C1: for (ap_uint<7> i = 0; i < ncenters; i++) {
    GUARDA_C2: for (ap_uint<7> j = 0; j < ndims; j++) {
        GUARDA_C3: for (ap_uint<7> l = 0; l < PARPOINTS; l++) {
           if (l == 0) accenters[i][j] = new_centersIn[l][i][j];
           else accenters[i][j] += new_centersIn[l][i][j];
           if (j == 0){
              if (l == 0) accenter_cnt[i] = ncenters_cnt[l][i];
                  else accenter_cnt[i] += ncenters_cnt[l][i];
              }
       }
      if (i%2==1){
          tmp.data.range(31,0) = accenters[i][j]/accenter_cnt[i];
          tmp.data.range(63,32) = accenters[i][j+1]/accenter_cnt[i];
          new_centers.write(tmp);
          tmp.last = (i == ncenters-1 && j == ndims-1) ? 1 : 0;
          tmp.keep=0xFF;
          tmp.strb=0xFF;
      }
   }
}}
```

Código 9-40 – Função do bloco de cálculo e envio dos novos centros

A função soma para cada centro e para cada dimensão os `PARPOINTS` valores acumulados e divide pelo número de centros. Os resultados são enviados através da interface *AXI4-Stream*.

9.10.2.8 Função de Topo do Algoritmo K-Means

A função de topo implementa uma iteração do algoritmo *K-Means*. Para executar todas as iterações do algoritmo terá de ser executada várias vezes, em que no início do fluxo são carregados os novos centros calculados na iteração anterior (ver Código 9-41).

```
#include <axis_kmeans.h>

void init_centers(...) {...}

void load_data(...) {...}

void assign_points(...) {...}

void update_centers(...) {...}

void store_memberships (...) {...}

void store_centers(...) {...}

void kmeans( hls::stream<data_t> &dataIn,
             hls::stream<data_t> &dataOut,
             int npoints, ap_uint<7> ncenters, ap_uint<7> ndims) {

#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=dataIn
#pragma HLS INTERFACE axis port=dataOut
#pragma HLS INTERFACE s_axilite port = npoints bundle = control
#pragma HLS INTERFACE s_axilite port = ncenters bundle = control
#pragma HLS INTERFACE s_axilite port = ndims bundle = control

ap_uint<7> max_n_points = PARPOINTS;
ap_uint<7> min_n_points = npoints - (npoints / PARPOINTS) * PARPOINTS;
ap_uint<11> max_d_count = ndims * max_n_points;
ap_uint<11> min_d_count = ndims * min_n_points;

unsigned num_bursts = (npoints + PARPOINTS - 1) / PARPOINTS;
ap_uint<1> last;

unsigned int centers[CDSIZE];
ap_uint<t_par_points*32> l_data[MAXDIM];
unsigned int new_centers[PARPOINTS][MAXCENTERS][MAXDIM];
#pragma HLS ARRAY_PARTITION variable=new_centers cyclic factor=t_par_points dim=1
unsigned ncenters_cnt[PARPOINTS][MAXCENTERS];
#pragma HLS ARRAY_PARTITION variable=ncenters_cnt cyclic factor = t_par_points
dim=1
unsigned int dataFwd[PARPOINTS][MAXDIM];
#pragma HLS ARRAY_PARTITION factor=t_par_points variable=dataForward cyclic dim=1
ap_uint<7> index[PARPOINTS];
#pragma HLS ARRAY_PARTITION factor=t_par_points variable=index cyclic dim=1
ap_uint<7> indexOut[PARPOINTS];
#pragma HLS ARRAY_PARTITION variable=indexOut cyclic factor = t_par_points dim=1

init_centers(centers, dataIn, ncenters, ndims);
rst_new_centers(new_centers, ncenters_cnt, ncenters, ndims);

PROCYCLE: for (int i = 0; i < num_bursts; i++) {
#pragma HLS DATAFLOW

    ap_uint<11> data_count=(i==(num_bursts-1)) ? min_d_count: max_d_count;
    ap_uint<7> store_cnt = (i == (num_bursts-1)) ? min_n_points : max_n_points;
```

```
    last = (i == num_bursts-1? 1: 0);

    load_data(l_data, dataIn, ndims, data_count);
    assignPoints(index, dataForward, l_data, centers, ncenters, ndims);
    update_centers(new_centers, ncenters_cnt, indexOut, index, dataFwd, npoints,
                   ndims);
    store_memberships(dataOut, indexOut, store_cnt, last);
}

store_centers(dataOut, new_centers, ncenters_cnt, ncenters, ndims);
}
```

Código 9-41 – Função de topo que executa uma iteração do algoritmo K-Means

O bloco iterativo funciona em fluxo de dados (`HLS DATAFLOW`) e o número de vezes que é executado (`num_bursts`) depende do número de pontos (`npoints`) e do fator de paralelismo (`PARPOINTS`) sendo igual a $\left\lceil \frac{npoints}{PARPOINTS} \right\rceil$. Na especificação houve o cuidado de determinar o número de pontos na última iteração, uma vez que o número total de pontos pode não ser múltiplo do fator de paralelismo.

9.10.3 Resultados de Implementação

A Tabela 9-13 apresenta os recursos necessários para a implementação do circuito resultado da síntese da função, bem como o número de ciclos de relógio de execução. Consideram-se diferentes fatores de paralelismo: 16, 32 e 64. Em todos os casos, a latência é calculada para 100,000 pontos, cada um com 2 dimensões e 10 centros.

Fator de paralelismo	Recursos				Latência (ciclos)
	LUT	FF	BRAM	DSP	
16	15129	13192	34	52	188074
32	29320	21247	66	100	94352
64	55191	37759	130	196	67807

Tabela 9-13 – Utilização de recursos e latência total do circuito do circuito de execução do algoritmo K-Means.

Como seria de esperar, o aumento do fator de paralelismo melhora a latência, mas os recursos necessários também aumentam de forma aproximadamente linear em função do fator de paralelismo.


POLITÉCNICO
DE LISBOA
POLYTECHNIC
UNIVERSITY
OF LISBON

Mário Véstias doutorou-se em Engenharia Eletrotécnica e de Computadores (ECE) pelo Instituto Superior Técnico (IST), Universidade de Lisboa, em 2002. É atualmente Professor Coordenador no Departamento de Engenharia Eletrónica e Telecomunicações e de Computadores do Instituto Superior de Engenharia de Lisboa, onde leciona e é responsável por disciplinas na área do projeto de hardware. É investigador no INESC INOV onde realiza investigação nas áreas de arquitetura de computadores, computação embebida de elevado desempenho e computação reconfigurável.

Paulo Flores doutorou-se em Engenharia Eletrotécnica e de Computadores (ECE) pelo Instituto Superior Técnico (IST), Universidade de Lisboa, em 2001. É atualmente Professor Associado no Departamento de Engenharia Eletrotécnica e de Computadores do Instituto Superior Técnico, onde leciona e é responsável por disciplinas nas áreas dos algoritmos e estruturas de dados e do projeto de sistemas digitais. É investigador no INESC-ID onde realiza investigação nas áreas de arquitetura de computadores, sistemas embebidos e computação reconfigurável recorrendo a FPGA.

Horácio C. Neto doutorou-se em Engenharia Eletrotécnica e de Computadores (ECE) pelo Instituto Superior Técnico (IST), Universidade de Lisboa, em 1992. É atualmente Professor Associado no Departamento de Engenharia Eletrotécnica e de Computadores do Instituto Superior Técnico, onde leciona e é responsável por disciplinas na área do projeto de sistemas digitais. É investigador no INESC INOV onde realiza investigação nas áreas de projeto de sistemas digitais e arquiteturas de computadores, com ênfase em computação reconfigurável.

O progresso da tecnologia ditou o aumento da complexidade dos sistemas digitais. Em consequência, tornou-se necessário utilizar descrições mais abstratas dos circuitos e recorrer a novas ferramentas de síntese para o seu projeto e desenvolvimento. Esta abordagem contribui para o aumento da produtividade no projeto de sistemas digitais e permite uma exploração mais ampla do espaço de projeto.

O livro aborda os diferentes aspetos da síntese de alto nível de sistemas digitais em FPGA (*Field Programmable Gate Array*) e está dividido em três partes. Começa por abordar os conceitos fundamentais à compreensão do projeto de hardware digital. Depois apresenta as estruturas de programação para a síntese de alto nível, nomeadamente, a síntese de estruturas de repetição, a síntese de interfaces, a síntese de tipos de dados com precisão arbitrária e a síntese modular. Por fim, apresenta um conjunto de exemplos ilustrativos da aplicação da síntese de alto nível.

O livro destina-se a estudantes e a profissionais na área de projeto de hardware. No âmbito do ensino pode ser utilizado como bibliografia de unidades curriculares que abordem o projeto de sistemas digitais com síntese de alto nível. Em contexto profissional, pode ser utilizado como bibliografia de formação e de consulta.

9 789893 515853

POLITÉCNICO DE LISBOA | POLYTECHNIC UNIVERSITY OF LISBON