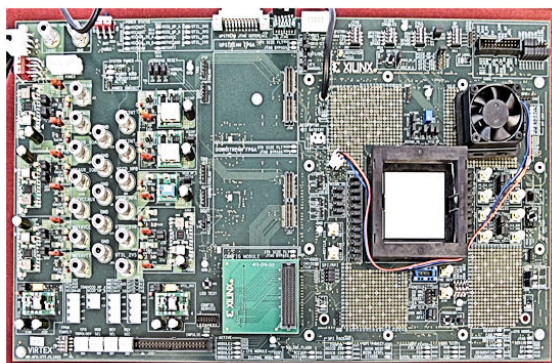




TÉCNICO LISBOA



Compilador para a Arquitectura Reconfigurável Versat

Rui Manuel Alves Santiago

Dissertação para obtenção do Grau de Mestre em

Engenharia Electrotécnica e de Computadores

Orientador(es): Prof. José João Henriques Teixeira de Sousa

Júri

Presidente: Prof. Gonçalo Nuno Gomes Tavares

Orientador: Prof. José João Henriques Teixeira de Sousa

Vogal: Prof. Paulo Ferreira Godinho Flores

Maio de 2016

Agradecimentos

Quero agradecer o apoio do Professor José de Sousa, pela boa orientação dada durante a dissertação toda, tanto a nível de escrita como de implementação prática do trabalho.

Quero agradecer o apoio do João Lopes, pela ajuda que deu na compreensão da arquitectura e na geração e depuração do resultado final do compilador.

Quero agradecer o apoio do Luis Valente, pela colaboração que deu para o compilador do Versat durante o estágio de Verão, no qual se iniciou a implementação do compilador.

Quero agradecer o apoio da Raísa Paes, por me ter ajudado a ler o relatório e por me ter apoiado nos momentos mais complicados que decorreram durante o trabalho.

Quero agradecer à minha família por ter acreditado em mim desde a entrada na faculdade.

Resumo

Existem sequências de operações matemáticas na área de processamento de sinal que são muito mais eficientes se forem feitas por um bloco de hardware específico do que num processador embebido. Assim, é frequente os processadores embebidos utilizarem aceleradores de hardware reconfigurável, que podem ser programados para várias funções.

Este trabalho consiste na implementação de um compilador para o acelerador de hardware reconfigurável Versat, desenvolvido no INESC-ID, o qual no início deste trabalho era programável apenas na sua linguagem *assembly* própria.

O Versat é uma arquitectura reconfigurável de grão grosso (Coarse Grain Reconfigurable Array, ou CGRA) dotado de um controlador muito simples mas suficiente para implementar a auto-reconfiguração parcial e a transferência de dados de e para a memória central, que são características distintivas do Versat.

Ao programar o Versat em *assembly* o programador tem de projectar previamente as configurações do Versat (circuitos de hardware utilizando os recursos do Versat) e incluir instruções no código que escrevem essas configurações nos registos de configuração.

O compilador desenvolvido neste trabalho permite ao programador utilizar expressões de alto nível que são interpretadas como configurações do Versat e automaticamente traduzidas em instruções de escrita de configurações. A sintaxe da linguagem pode ser considerada um subconjunto da linguagem C++.

A grande dificuldade da implementação do compilador é a inexistência de uma abordagem padrão para a construção de compiladores para arquitecturas CGRA. É necessário ter em conta técnicas de compilação comuns mas também é preciso implementar técnicas de colocação de blocos de hardware e encaminhamento de sinais, bem como de gestão dinâmica dos recursos reconfiguráveis.

Palavras-chave: CGRA, compilador, processamento de sinal, C++

Abstract

There are sequences of mathematical operations in the signal processing area which are much more efficient if done by specific hardware than by an embedded processor. So, it is common to use reconfigurable hardware accelerators attached to embedded processors. Reconfigurable hardware accelerators can be programmed for many different functions.

This work consists in the implementation of a compiler for the Versat reconfigurable hardware accelerator developed in INESC-ID. Before the compiler implementation, this accelerator was only programable through assembly language.

Versat is a Coarse Grain Reconfigurable Array, or CGRA. Versat has a very simple controller to implement partial self-reconfiguration and transfer data to and from the central memory, which are distinctive features of Versat. An embedded processor that uses Versat is free of these tasks.

The Versat controller is a conventional machine that configures the reconfigurable part of Versat, writing words to programming registers. When programming Versat in assembly language, the programmer must first design the hardware circuits using the Versat resources, and then include in the code instructions that write those configurations in the configuration registers.

The compiler developed in this work allows the programmer to use high-level expressions that are interpreted as Versat configurations and are automatically translated into configuration write instructions. The language syntax can be considered a subset of the C++ language.

The difficulty of implementing the compiler is the inexistence of a standard approach to build compilers for CGRA architectures. It is necessary to take into account common compilation techniques but it is also necessary to implement hardware place and route techniques and dynamic management of reconfigurable resources.

Keywords: Versat, CGRA, compilador, processamento de sinal, C++

Conteúdo

Agradecimentos	iii
Resumo	v
Abstract	vii
Lista de Tabelas	xiii
Lista de Figuras	xv
Lista de Acrônimos	xvii
1 Introdução	1
1.1 Arquitecturas CGRA	1
1.2 Acoplamento ao sistema anfitrião	2
1.3 Reconfiguração e agendamento	3
1.4 Suporte ao paralelismo	4
1.5 Compiladores para CGRAs	4
1.6 Estrutura da tese	5
2 Arquitectura do Versat	7
2.1 Modelo de Topo	7
2.2 Controlador	8
2.3 Motor de Dados	10
2.4 Subsistema de configuração	14
2.5 Memória de instruções	14
2.6 Motor de transferências de dados	14
3 Compilador do Versat	17
3.1 Estrutura do compilador	17
3.1.1 Front end	17
3.1.2 Back end	21
3.2 Descrição da linguagem do Versat	25
3.2.1 Objectos e variáveis predefinidos	25
3.2.2 int main	25
3.2.3 Métodos para controlo do motor de dados	26

3.2.4	Métodos exclusivos das memórias	27
3.2.5	Métodos exclusivos das ALU e das ALULite	28
3.2.6	Métodos exclusivos do multiplicador	29
3.2.7	Métodos exclusivos do deslocador	29
3.2.8	Métodos de conexão	29
3.2.9	Expressões de registos	30
3.2.10	IF condicional	30
3.2.11	ELSE condicional	31
3.2.12	Ciclo while	31
3.2.13	Ciclo for	31
3.2.14	Ciclo do while	31
3.2.15	Salto incondicional	32
3.2.16	Expressões de memórias	32
3.2.17	Return	34
3.2.18	Asm	34
3.2.19	Comentários	35
3.2.20	DMA	35
3.2.21	Variáveis	36
4	Exemplos de código	39
4.1	Adição de vectores	39
4.2	Produto interno complexo	42
4.3	FFT	44
4.4	Filtro passa baixo de primeira ordem	47
4.5	Filtro passa baixo de segunda ordem	48
4.6	Raíz quadrada	50
5	Resultados	53
5.1	Tempo de compilação	53
5.2	Número de linhas do <i>assembly</i> gerado	54
5.3	Tempo de execução	55
6	Conclusão	57
6.1	Trabalho realizado	57
6.2	Trabalho Futuro	58
A	Código FFT apenas com configurações parciais	61
B	Código assembly do exemplo do ciclo for	65
C	Código em C++ da FFT usando a memória de configurações	69

Lista de Tabelas

2.1	Tabela das instruções assembly do Versat.	9
2.2	Parâmetros dos geradores de endereços.	10
2.3	Funções das ALUs.	11
2.4	Latências das unidades funcionais.	12
2.5	Registo de controlo do Motor de Dados.	13
2.6	DE status register.	13
3.1	Atributos do objecto comando.	21
3.2	Objectos e variáveis predefinidos.	26
3.3	Métodos para controlo do motor de dados	27
3.4	Métodos respectivos à configuração dos portos das memórias.	27
3.5	Método para configuração da ALU e ALULite.	28
3.6	Operações permitidas pela ALU.	28
3.7	Métodos do multiplicador.	29
3.8	Métodos exclusivos do deslocador.	29
3.9	Métodos de conexão.	30
3.10	Operações suportadas pelas expressões de registos.	30
3.11	Prefixos de etiquetas que o utilizador não pode usar.	32
3.12	Métodos do DMA Engine.	35
5.1	Tempo de compilação dos programas de teste.	54
5.2	Número de linhas de código dos programas de teste.	54
5.3	Tempos de execução dos programas de teste.	55
5.4	Tempos de execução dos programas no Motor de Dados e no controlador.	56

Lista de Figuras

2.1	Esquema de topo.	8
2.2	Esquema da unidade de controlo	9
2.3	Diagrama temporal da geração de endereços.	10
2.4	Esquema de ligações do Motor de Dados.	12
3.1	Árvore gerada do ciclo <i>while</i>	18
3.2	Árvore da instrução de configuração do gerador de endereços.	19
3.3	Árvore gerada de uma expressão de registos.	20
3.4	Árvore de representação de expressão.	23
3.5	Esquema de circuito de uma expressão de multiplicações com subtracção.	24
3.6	Esquema do <i>datapath</i> da expressão que exemplifica o uso de variáveis.	37
4.1	Esquema do <i>datapath</i> da adição de vectores.	40
4.2	Esquema do <i>datapath</i> do produto interno complexo.	42
4.3	FFT de um sinal de entrada com 8 pontos.	44
4.4	Borboleta da FFT	45
4.5	Esquema do <i>datapath</i> da soma de complexos.	46
4.6	Esquema do <i>datapath</i> do filtro passa baixo.	47
4.7	Esquema do Motor de Dados da equação às diferenças de segundo grau.	49

Lista de Acrônimos

CGRA Coarse Grain Reconfigurable Arrays

FPGA Field Programmable Gate Array

FU Functional Unit

MRRG Modulo Resource Routing Graphs

VLIW Very Long Instruction Word

Capítulo 1

Introdução

O objectivo deste trabalho é o desenvolvimento de um compilador para o Versat, um acelerador de *hardware* que usa uma arquitectura de computação reconfigurável do tipo Coarse Grain Reconfigurable Arrays (CGRA). Portanto, é de esperar que a implementação do compilador seja diferente de uma implementação clássica. A melhor forma de implementar compiladores para arquitecturas reconfiguráveis tem sido investigado intensamente nas últimas décadas.

Como dito anteriormente, o Versat é um acelerador de *hardware* que permite acelerar de uma forma bastante prática algoritmos que requerem bastante processamento da parte de processadores sequenciais. Devido ao tipo de arquitectura usado, é possível alterar o algoritmo usado mudando apenas o programa do Versat, o que flexibiliza bastante em relação ao uso de *hardware* dedicado.

Este capítulo introdutório está dividido em várias secções. Na secção 1.1 é feita uma introdução teórica sobre as arquitecturas CGRA, para se entender o tipo de arquitectura do Versat e como este tipo de arquitecturas se diferencia de outras arquitecturas. Na secção 1.2 explica-se como se utiliza externamente o Versat. Na secção 1.3 é debatido o tipo de reconfiguração e agendamento do Versat, comparando-o com outras arquitecturas. Esta componente é importante para a definição da forma de programar o sistema Versat. Na secção 1.4 fala-se no paralelismo existente em arquitecturas CGRA e como este pode ser útil na perspectiva da programação do sistema. Na secção 1.5 faz-se uma descrição de outros compiladores para arquitecturas CGRA e compara-se estes ao compilador do Versat. Na secção 1.6, é feita uma introdução do que será falado nos próximos capítulos.

1.1 Arquitecturas CGRA

As arquitecturas do tipo CGRA ganharam atenção nas duas últimas décadas [1, 2, 3, 4, 5] como forma de reduzir a complexidade, o tempo de configuração e o tempo de compilação em relação às arquitecturas do tipo Field Programmable Gate Array (FPGA). Isto é conseguido utilizando elementos de processamento de grão mais grosso, como por exemplo ALUs, multiplicadores, deslocadores, etc [6, 5], em menor número, necessitando consequentemente de menos ligações programáveis entre eles.

Nas CGRAs existem dois tipos de configuração: a configuração estática e a configuração dinâmica

(reconfiguração). Na configuração estática o sistema é configurado uma única vez para correr um programa completo [7], o que é menos flexível. Na configuração dinâmica é permitido reconfigurar o sistema múltiplas vezes durante um programa. No entanto, é necessário contabilizar o tempo de reconfiguração de modo a que o desempenho seja satisfatório.

Algumas CGRAs, como a arquitectura ADRES por exemplo [8], são totalmente dinâmicas, ou seja, a cada ciclo de execução é executada uma reconfiguração total. Podem ocorrer estagnamentos (*stalls*) e nesse caso não ocorre reconfiguração. Outras arquitecturas como a KressArray [9, 10, 11] são totalmente estáticas, o que significa que a CGRA é configurada antes de entrar num ciclo de programa e não existe nenhuma reconfiguração depois do ciclo arrancar.

Também existem arquitecturas híbridas como a arquitectura RaPiD[12, 13]. As arquitecturas híbridas suportam configurações parciais dinâmicas.

Existe uma desvantagem principal das arquitecturas CGRA. Estas apenas conseguem executar ciclos de programa, o que implica que é necessário conectar a CGRA a um processador principal (sistema anfitrião) onde o resto do programa é executado, havendo necessidade de os dois sistemas partilharem os dados, incorrendo-se numa latência suplementar [8].

O Versat possui um tipo de configuração dinâmica. Configura-se o Versat e depois de arrancar e reconfigura-se depois de acabar cada ciclo de programa. O Versat também suporta configurações parciais, tanto ao nível de *hardware* como de compilador. No limite, o programador pode configurar apenas um parâmetro de uma unidade funcional.

O Versat utiliza reconfiguração dinâmica de uma forma original. Baseia-se na observação de que os ciclos de programa não ocorrem isoladamente. A maior parte das vezes ocorrem vários ciclos seguidos, sendo que cada ciclo utiliza os dados provenientes do ciclo anterior. Assim, o Versat foi projectado para executar troços de programa que consistem em vários ciclos, auto reconfigurando-se para cada ciclo. Para isso possui um controlador de reconfigurações de baixo desempenho que corre um programa simples, gerando as configurações e controlando a execução do *hardware* reconfigurável. Programar o Versat resume-se a programar o controlador de reconfigurações, o que, antes deste trabalho, era feito em linguagem *assembly*. Com este trabalho passa a ser possível programar o Versat com uma linguagem de mais alto nível e com uma sintaxe próxima do C++.

1.2 Acoplamento ao sistema anfitrião

Para projectar um compilador de um sistema com uma CGRA é fundamental levar em linha de conta a forma como a CGRA está acoplada ao sistema anfitrião.

Alguns sistemas CGRA são acoplados de forma solta ao sistema anfitrião. Por exemplo, no caso do acelerador externo MorphoSys CGRA [14], a CGRA está acoplada a um processador TinyRISC [15] que é responsável por executar as instruções fora dos ciclos de programa, controlar o DMA (usado para transferências de dados) e controlar a CGRA.

Este tipo de acoplamento tem a vantagem de ser possível projectar o sistema CGRA independentemente do sistema mestre, o que permite paralelismo e eficiência. A grande desvantagem desta forma

de acoplamento é que é necessário transferir dados de e para a CGRA usando o DMA. Quando existem várias transferências de dados ocorre uma latência considerável.

Uma alternativa ao acoplamento solto é o acoplamento justo. No acoplamento justo existe um banco de registos partilhado entre o sistema anfitrião e o acelerador CGRA. Existem sistemas onde também se partilham memórias. Executando uma instrução para iniciar o sistema CGRA, o processador espera até que esta conclua a execução. A desvantagem do acoplamento justo é que devido à partilha de recursos, o processador e o CGRA não podem executar em paralelo[8]. Apesar desta desvantagem, o processador consegue trocar dados com a CGRA sem nenhuma latência.

O Versat utiliza um tipo de acoplamento solto. De forma geral, os sistemas CGRA são programados pelo próprio anfitrião, mas o Versat além do sistema CGRA, tem o controlador dedicado. Conseguindo executar cálculos simples entre ciclos de programa, o Versat consegue manter os dados nas suas memórias durante muito mais tempo que uma CGRA tradicional, mitigando assim as transferências de dados. O sistema anfitrião é completamente independente do Versat, cada um tem um compilador independente. O sistema anfitrião apenas tem de invocar o Versat, que por sua vez tem um programa próprio. O acoplamento do Versat também permite um nível de paralelismo eficiente, pois tanto o sistema anfitrião, como a componente CGRA e até mesmo o controlador dedicado do Versat e o DMA trabalham de forma paralela.

1.3 Reconfiguração e agendamento

Existem diversas formas de realizar a reconfiguração e o agendamento numa arquitectura CGRA. Podem ser realizadas estaticamente através de um compilador ou de forma dinâmica através de *hardware*, existindo também soluções híbridas.

Uma primeira classe de sistemas consiste em CGRAs com agendamento e reconfiguração dinâmicas como por exemplo, a arquitectura TRIPS [16]. Para esta classe de sistemas, o compilador determina em qual unidade funcional cada operação é executada e que conexões são necessárias, mas parte das reconfigurações necessárias e ordem das operações são determinadas no hardware durante a execução. Como são suportadas instruções de controlo de fluxo torna-se difícil determinar estaticamente agendamentos de alta qualidade, donde que se tenha optado por técnicas de agendamento dinâmico.

Uma segunda classe de arquitecturas com reconfiguração dinâmica suporta agendamento estático. As arquitecturas desta classe, como as CGRAs MorphoSys e Silicon Hive, realizam o agendamento estático através do compilador. Os algoritmos utilizados são baseadas em técnicas de software *pipelining* [17], bastante utilizadas em arquitecturas Very Long Instruction Word (VLIW).

Uma terceira classe de sistemas consiste em arquitecturas com reconfiguração estática e agendamento dinâmico [18]. O compilador executa a alocação de recursos e encaminhamento de sinais. Os dados circulam nestas arquitecturas acompanhados de sinais auxiliares que determinam dinamicamente os caminhos a seguir e as unidades funcionais a utilizar.

O Versat é uma CGRA que tem características próprias no que toca a reconfiguração e agenda-

mento. O seu tipo de reconfiguração dinâmica pode ser considerado quase-estático com agendamento estático. A originalidade do Versat é ser capaz de executar não apenas um ciclo de programa mas vários ciclos seguidos ou quase seguidos, autoreconfigurando-se para tal. No entanto, o controlo de fluxo no interior dos ciclos não é suportado devido a ter-se observado que este raramente ocorre em aplicações de processamento de sinal ou de vastas quantidades de dados (*big data*).

1.4 Suporte ao paralelismo

Um aspecto importante das arquitecturas CGRA é o suporte a diversas formas de paralelismo. Tal como visto na secção 1.2, com o acoplamento solto consegue-se paralelismo entre o processador principal e a CGRA. Este é um paralelismo ao nível de bloco de processamento. Mais interessante é o paralelismo que se pode conseguir dentro da própria CGRA.

Quando o agendamento é dinâmico e é implementado através de controlo distribuído baseado em eventos, implementar paralelismo de *threads* é relativamente barato e fácil [8]. Caso ocorram ciclos de programa independentes traduzidos em circuitos suficientemente pequenos na CGRA, vários ciclos podem ser processadas com recursos diferentes da CGRA.

Para arquitecturas com controladores, a única solução para correr *threads* em paralelo é acrescentar mais controladores, ou então o suporte de *threads* em cada controlador com grande perda de eficiência. Em geral, os controladores que suportam *threads* têm diferentes modos de funcionamento.

Também é possível suportar o paralelismo incorporando vários núcleos CGRA num *chip*, mas mais interessante que isso é o suporte do paralelismo internamente à CGRA. Existindo vários portos de memória com geradores de endereços independentes, tal como existem na arquitectura Versat, é possível realizar paralelismo ao nível de *thread*. O desenvolvimento de geradores de endereços capazes de suportar ciclos encadeados numa única configuração permitiu eliminar reconfigurações nestes ciclos [5]. A ideia foi inspirada no uso de contadores em cascata para o gerador de endereços[19].

Para explorar esta possibilidade ao nível de compilador é necessário distinguir entre a configuração da CGRA e a execução da mesma. Podem ser configurados vários circuitos independentes que depois se mandam executar em paralelo, habilitando os geradores de endereços respectivos.

1.5 Compiladores para CGRAs

Existem algoritmos específicos para compilar código para CGRAs. Estes algoritmos distinguem-se pelo tipo de agendamento e pelo tipo de linguagem suportada.

A linguagem suportada pode ser uma linguagem comum de programação, como o C++ ou o Java, mas também pode ser uma linguagem apenas descritiva da configuração do hardware, remetendo para o programador a responsabilidade de saber o que o hardware faz.

Em ambos os casos um compilador para este tipo de arquitecturas não pode usar apenas técnicas convencionais, precisando também de aplicar técnicas de colocação de componentes e encaminhamento de sinais (*Place and Route*), usadas também em FPGAs[20].

As técnicas mais usadas de agendamento usam Modulo Resource Routing Graphs (MRRG)s. As MRRGs são grafos de tempo/espaco, onde os recursos são representados por vértices e as ligações entre recursos são representadas por arcos. O agendamento, colocação de recursos e encaminhamento de sinais é representado por um grafo de dependências de dados (*Data Dependency Graph*). O problema resume-se então em mapear o grafo de dependências de dados na MRRG. O agendamento é usado para calcular o ciclo de relógio certo para realizar uma operação, a colocação de recursos é usada para encontrar as unidades funcionais certas e o encaminhamento de sinais é usado para determinar as ligações entre elas.

A granularidade dos vértices varia consoante o algoritmo do compilador. O compilador do sistema ADRES instancia cada Functional Unit (FU) como um vértice individual. Em norma, quanto mais complexo o vértice, menor tempo de compilação, mas também se reduz a flexibilidade da CGRA pois o número total de vértices será mais limitado.

O algoritmo do compilador DRESC [21, 22] usa a técnica de *simulated annealing* para explorar diferentes opções de colocação de recursos e encaminhamento de sinais. A função de custo usada é baseada no custo do encaminhamento de todos os sinais. Com esta técnica, um grande número de encaminhamentos possíveis é testado, o que torna o algoritmo lento.

Outros compiladores para arquitecturas CGRA [23, 24] usam técnicas de agendamento baseadas em listas[25]. Estas técnicas permitem a optimização de caminhos de dependências de dados, acabando por ser mais rápidas que o *simulated annealing*, apesar da menor qualidade da solução.

Devido à reconfiguração quase estática no Versat, tomou-se uma opção ao nível da arquitectura que simplificou drasticamente o compilador: existem ligações directas entre todas as unidades funcionais. Apesar desta opção custar alguma área de silício, o impacto na potência consumida é pouco relevante, pois devido à reconfiguração quase estática estas ligações comutam a uma frequência muito baixa, apenas quando se executa um novo ciclo de programa.

Assim, no compilador do Versat não são necessárias técnicas de colocação de recursos e encaminhamento de sinais. Usa-se apenas uma tabela de recursos em utilização, e as ligações são realizadas apenas seleccionando as entradas de cada unidade funcional em uso.

1.6 Estrutura da tese

Esta tese tem mais cinco capítulos. No capítulo 2 descreve-se sumariamente a arquitectura do Versat. No capítulo 3 faz-se a descrição completa do compilador desenvolvido. No capítulo 4 são descritos vários programas na linguagem desenvolvida para demonstrar o funcionamento do compilador. No capítulo 5 apresentam-se resultados experimentais que atestam a usabilidade e eficiência do compilador desenvolvido. Finalmente, o capítulo 6 conclui a tese.

Capítulo 2

Arquitectura do Versat

Neste capítulo descreve-se sumariamente a arquitectura CGRA do Versat de modo a melhor se entender as opções tomadas para o compilador desenvolvido neste trabalho. Nas secções que se seguem a arquitectura do Versat será descrita de acordo com a hierarquia do sistema.

2.1 Modelo de Topo

O diagrama de topo do Versat é dado na figura 2.1. O bloco *Controller* executa o programa contido no bloco *Program Memory*. Interage com um sistema anfitrião através de um banco de registos (*Control Register File*). Basicamente, o sistema anfitrião manda iniciar a execução de programas, passando os endereços dos dados, e verifica se a execução terminou. O motor de processamento de dados (*Data Engine*) é configurado pelo controlador, o qual escreve as configurações no bloco *Configuration Subsystem*. O motor de dados é carregado com dados da memória externa, recebe um sinal para iniciar o processamento, informa o controlador do término do processamento, e os dados produzidos são descarregados para a memória externa. Este processo normalmente repete-se várias vezes ao longo da execução de um programa, para várias configurações do motor de dados.

A troca de dados é realizada pela interface de dados (*Data IF*), utilizando o motor de transferências directas de/para a memória (*Direct Memory Access - DMA*). É também possível trocar dados e carregar programas pela interface de controlo, mas apenas quando a velocidade de transferência destes for pouco relevante - um exemplo disto é quando o Versat está em modo de depuração, sozinho num circuito integrado ou numa FPGA, e controlado externamente.

O controlador do Versat controla o barramento de leitura/escrita (*Read/Write Bus*), que é utilizado para efectuar leituras/escritas nos registos dos outros componentes ligados a este barramento. Desta forma, pode configurar, instruir e monitorizar os outros componentes.

A interface entre o sistema anfitrião e o banco de registos de controlo pode ser de dois tipos possíveis de seleccionar pelo utilizador: o barramento SPI e o barramento paralelo. O barramento SPI é usado quando se liga o Versat a um anfitrião externo, como por exemplo um PC, para realizar a depuração do hardware ou de programas. O escravo SPI é o Versat, enquanto o mestre SPI é o anfi-

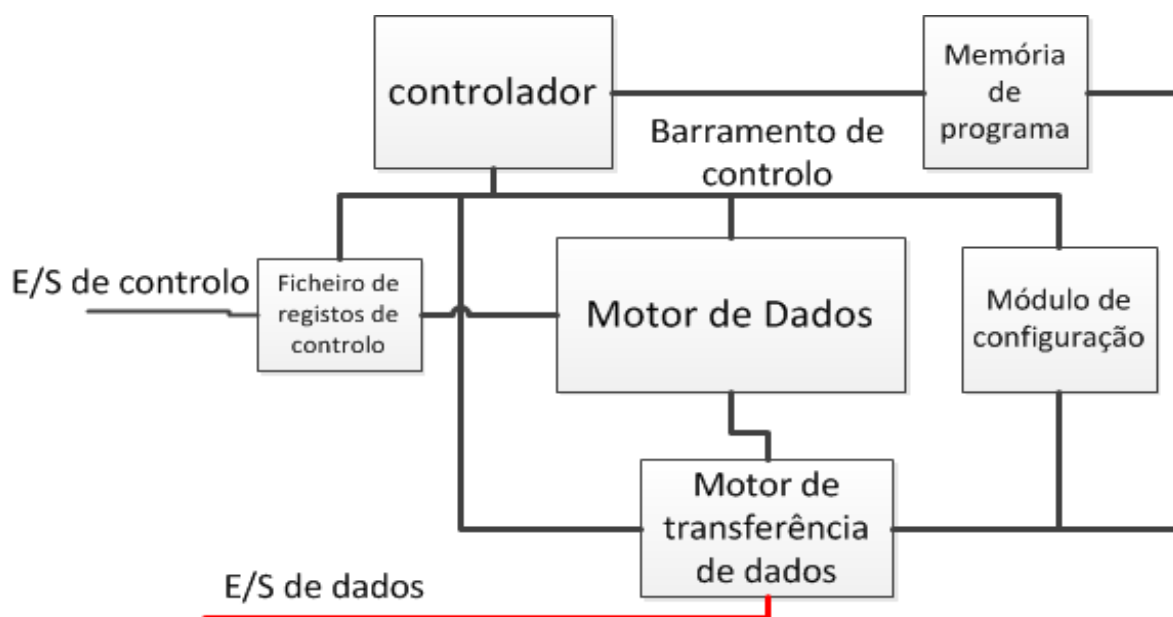


Figura 2.1: Esquema de topo.

trião externo. O barramento paralelo é utilizado quando se liga o Versat a um anfitrião embebido. Este é um barramento genérico mas pode ser necessário adaptar esta interface a um formato comercial, utilizando um barramento amba-AXI, por exemplo.

2.2 Controlador

Um esquema do controlador encontra-se representado na figura 2.2. O controlador realiza leituras/escritas nos seus periféricos, que são módulos do Versat. A arquitectura do controlador do Versat é constituída por três registos: o contador de programa, o acumulador (registo A) e o apontador (registo B).

O contador de programa é utilizado como endereço da memória de instruções. A saída da memória de instruções contém a instrução que o controlador necessita de ler.

O controlador do Versat usa uma arquitectura de acumulador. O registo A é o acumulador. O resultado de todas as operações realizadas pelo controlador são armazenadas no acumulador. É no bloco OP que se efectuem as várias operações possíveis com o controlador. O bloco OP recebe como entrada o próprio acumulador e um dado de entrada de 32 *bits*.

O registo B é utilizado para endereçamento indirecto, guardando o endereço da posição de memória a ler ou escrever numa instrução que utilize este tipo de endereçamento.

Em relação ao controlador, refere-se que as suas principais funções são a implementação do controlo dos programas do Versat, a reconfiguração e controlo do motor de dados e do DMA e a comunicação com o sistema anfitrião.

O controlador não pode ler nem escrever dados da memória de instruções. Este apenas consegue ler as próprias instruções com o objectivo de as executar, pois a memória de instruções contém apenas as instruções do programa.

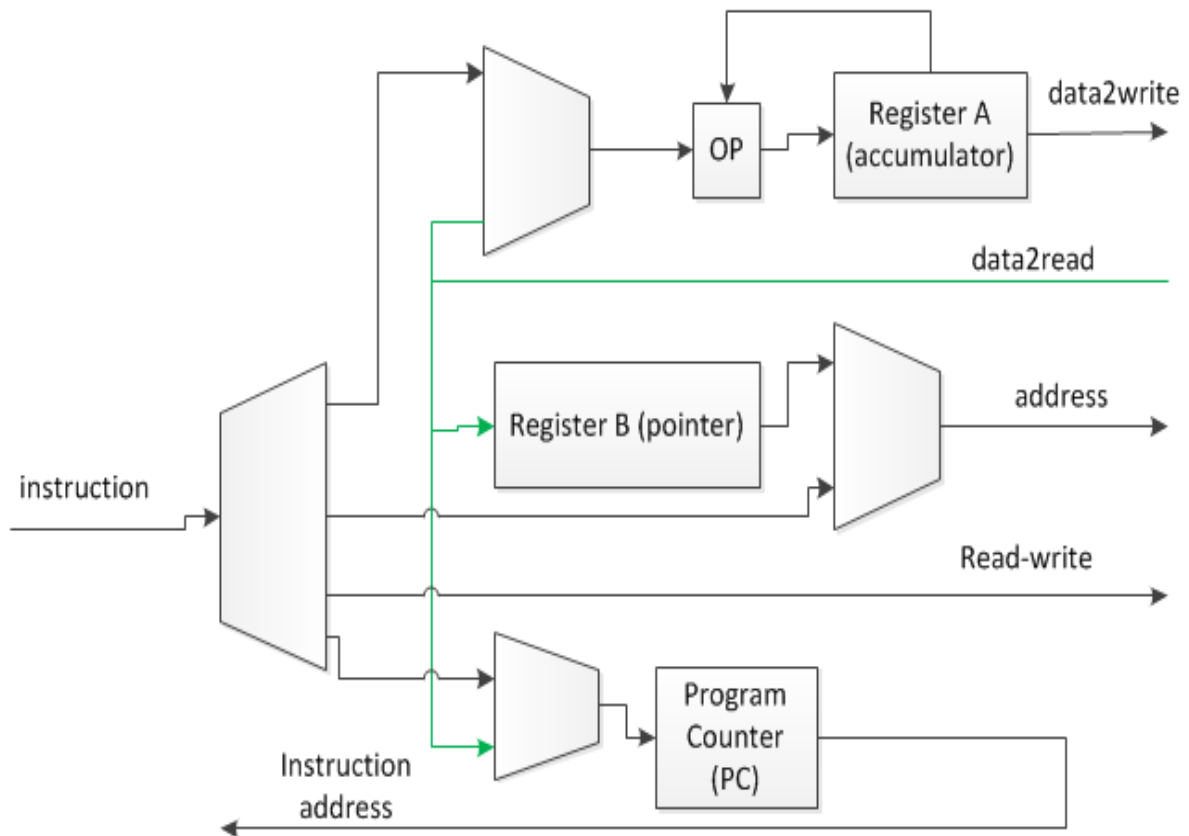


Figura 2.2: Esquema da unidade de controlo

O controlador pode escrever palavras de configuração no subsistema de configuração, de modo a reconfigurar parcialmente o motor de dados. Também pode realizar pequenos cálculos que sejam necessários, nomeadamente para produzir novas configurações.

O conjunto de instruções executadas pelo controlador encontra-se descrito na tabela 2.1. É importante referir que a sigla *imm* é o imediato e que este tem 16 bits de tamanho. **Revisão 3!!!!**

Tabela 2.1: Tabela das instruções assembly do Versat.

Instruções	Descrição
nop	No operation
rdw	$A \leftarrow (imm)$
wrw	$(imm) \leftarrow A$
wrc	$(imm1+imm2) \leftarrow A$
rdwb	$A \leftarrow (B)$
wrwb	$(B) \leftarrow A$
beqi	$PC \leftarrow imm$ if regA=0
beq	$PC \leftarrow (imm)$ if regA=0
bneqi	$PC \leftarrow imm$ if regA!=0
bneq	$PC \leftarrow (imm)$ if regA!=0
ldi	$A \leftarrow imm$
ldih	$A[31:16] \leftarrow imm$
add	$A \leftarrow A + (imm)$
addi	$A \leftarrow A + imm$
sub	$A \leftarrow A - (imm)$
and	$A \leftarrow A \& (imm)$

2.3 Motor de Dados

O motor de dados é o componente CGRA do sistema Versat. É constituído por 15 unidades funcionais, organizadas como indicado na figura 2.4. A arquitectura utilizada tem 32 bits.

As memórias embebidas (MEM) de porto duplo são memórias que têm um gerador de endereços por porto. É possível configurar cada um dos geradores de endereços de forma independente. Um gerador de endereços pode ser configurado com o número de iterações, período de cada iteração, incremento por ciclo, endereço de início e deslocamento entre períodos. Os portos das memórias podem ser configurados para leitura ou escrita. No caso de serem configurados para escrita, é necessário também configurar os portos de modo a seleccionarem os dados correctamente.

Na figura 2.3 está representado um diagrama temporal de um gerador de endereços. O significado dos parâmetros está explicitado na tabela 2.2.

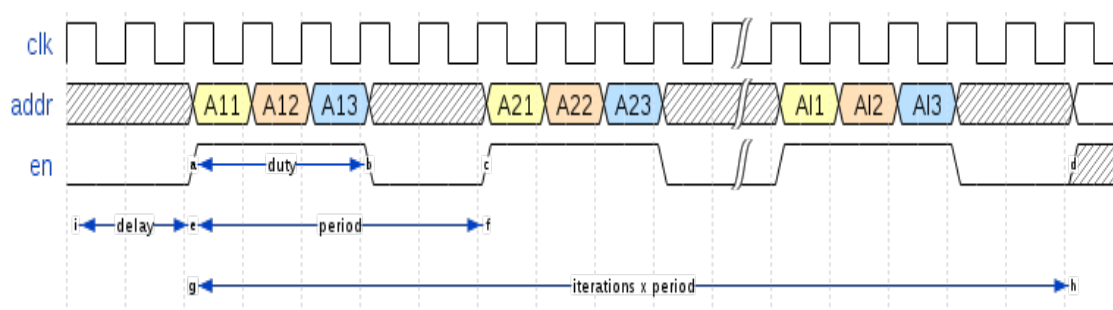


Figura 2.3: Diagrama temporal da geração de endereços.

Tabela 2.2: Parâmetros dos geradores de endereços.

Parâmetro	Descrição
Start (não ilustrado)	Indica a partir de que endereço de memória se começa a ler ou a escrever. Quando não se configura nada, o valor por omissão a nível de <i>hardware</i> é zero.
Duty	Representa o número de ciclos de relógio num período em que a memória está habilitada.
Incr (não ilustrado)	É o incremento de posições de memória que o gerador de endereços faz. Por exemplo, quando o incremento é 2, as posições de memória são lidas de dois em dois.
Iterations	É o número de iterações que o gerador de endereços faz até estar concluído. No registo de estado só se indica que um porto de memória está pronto depois das iterações acabarem.
Period	É o número de ciclos de relógio por iteração. O período é usado para fazer malhas aninhadas no programa.
Shift (não ilustrado)	É um incremento adicional do endereço no fim de cada período.
Delay	É o número de ciclos de relógio que os geradores de endereços esperam antes de começarem a trabalhar, relativamente ao primeiro gerador a arrancar.
Reverse (não ilustrado)	É o espelhamento dos endereços de memória. Em certas aplicações, como por exemplo a FFT, é necessário fazer espelhamento de endereços de memória.

As ALUs são unidades aritméticas lógicas que exercem diversas funções. As funções das ALUs

estão representadas na tabela 2.3. As ALULite executam apenas as primeiras 6 funções das ALUs.

Tabela 2.3: Funções das ALUs.

Operação	Descrição
OR	OR lógico.
AND	AND lógico.
NAND	NAND lógico.
XOR	XOR lógico.
soma	soma aritmética.
subtracção	subtracção aritmética.
EXT8	Extensão de sinal de 8 para 32 bits.
EXT16	Extensão de sinal de 16 para 32 bits.
SHR	<i>Shift right</i> aritmético.
SLR	<i>Shift right</i> lógico.
SMPS	Comparação com sinal.
SMPU	Comparação sem sinal.
CLZ	Contagem dos bits que estão a 0 à esquerda.
MAX	Máximo.
MIN	Mínimo.
ABS	Valor absoluto.

Os multiplicadores efectuam uma multiplicação de dois operandos de 32 bits, obtendo um resultado de 64 bits, dos quais apenas 32 bits são utilizados. É possível optar-se pela parte alta da multiplicação (bit 32 ao bit 63) ou pela parte baixa (bit 0 ao bit 31). Também é possível multiplicar (ou não) por dois o resultado final, o que é útil quando se trabalha em notações de vírgula fixa.

Os deslocadores são unidades especializadas em deslocamento de bits (*barrel shifters*). Um dos operandos é a palavra a deslocar e o outro operando é a magnitude do deslocamento. Os deslocadores são configurados com o sentido de deslocamento (esquerda ou direita) e o tipo de deslocamento (aritmético ou lógico).

Cada unidade de processamento contribui com 32 bits para o barramento de dados. Cada unidade de processamento é capaz de seleccionar qualquer secção de 32 bits existente no barramento de dados, o que permite que todas as unidades de processamento possam estar interligadas umas às outras. Isto facilita o trabalho do compilador, que assim não tem a necessidade de fazer *Place and Route*.

As unidades de processamento são configuradas com o modo de operação e com as respectivas entradas. Não existem configurações globais; apenas as unidades de processamento podem ser configuradas, dado que esta arquitectura permite a realização de qualquer circuito de dados (*datapath*) usando as unidades existentes. Cada *datapath* realiza uma tarefa específica.

Se num *datapath* estiverem várias memórias, estas necessitam de trabalhar em sincronismo, para tal é necessário configurá-las coerentemente. Se existirem recursos suficientes, vários *datapaths* podem ser executados em paralelo, permitindo realizar paralelismo ao nível de tarefa.

Cada unidade funcional tem uma latência. Desta forma, ao configurar um *datapath* no motor de dados, é necessário ter em conta esta latência. Para tal, os geradores de endereços possuem o parâmetro *delay* como já foi explicado. As latências de cada unidade funcional são dadas na tabela 2.4.

Revisão 2!!!! Para controlar o motor de dados é necessário escrever no seu registo de controlo,

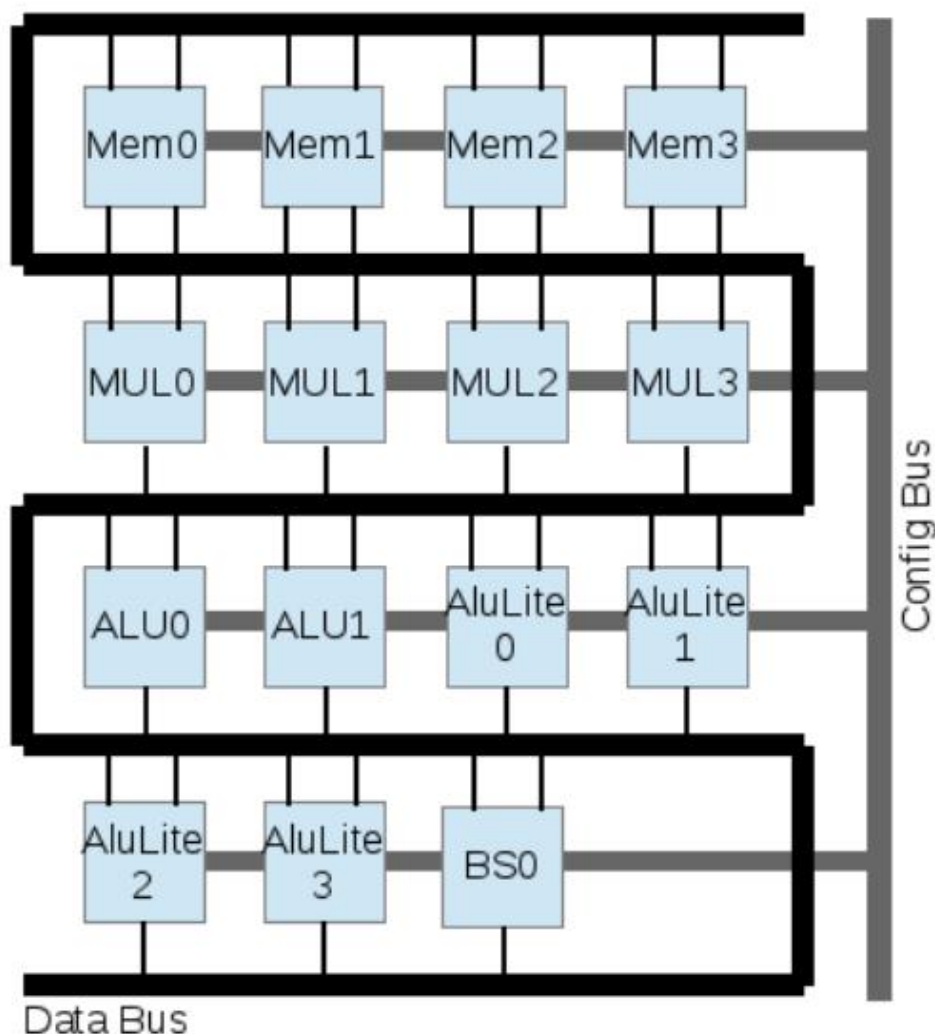


Figura 2.4: Esquema de ligações do Motor de Dados.

Tabela 2.4: Latências das unidades funcionais.

Unidade funcional	Latência
multiplicador	3
memória	1
ALU	2
ALULite	2
barrel shifter	1

descrito na tabela 2.5. Pode essencialmente inicializar-se as unidades de processamento (bit Init) ou mandar correr geradores de endereços (bit Run). Os restantes bits deste registo indicam quais as unidades de processamento a inicializar ou correr. A inicialização de memórias tem como resultado o carregamento das configurações dos registos de endereços de cada um dos portos (A ou B). A inicialização de outras unidades funcionais tem como resultado a inicialização do valor de saída a zero. Mandar correr um porto de memória faz iniciar a geração de endereços nesse porto. Mandar correr qualquer outra unidade de processamento tem apenas o efeito de anular a sua saída mesmo antes do arranque, pois entretanto podem ter-se propagado valores espúrios lidos das memórias, os quais

podem vir a ser acumulados no caso do circuito ter realimentação.

Tabela 2.5: Registo de controlo do Motor de Dados.

Bit	Descrição
0	Init
1	Run
2	BS0
3	Mult3
4	Mult2
5	Mult1
6	Mult0
7	ALULite3
8	ALULite2
9	ALULite1
10	ALULite0
11	ALU1
12	ALU0
13	MEM3B
14	MEM3A
15	MEM2B
16	MEM2A
17	MEM1B
18	MEM1A
19	MEM0B
20	MEM0A
21-31	Reserved

Depois de mandar executar o motor de dados é necessário saber quando a sua operação termina. Para tal, utiliza-se o registo de estado do Versat, descrito na tabela 2.6. Este registo indica quais as memórias que terminaram de gerar a sua sequência de endereços.

Tabela 2.6: DE status register.

Bit	Description
0	MEM0B done
1	MEM0A done
2	MEM1B done
3	MEM1A done
4	MEM2B done
5	MEM2A done
6	MEM3B done
7	MEM3A done
8-31	Reserved

2.4 Subsistema de configuração

O subsistema de configuração é um sistema de memória onde as configurações do Motor de Dados encontram-se armazenadas. A configuração principal está guardada num registo parcialmente endereçado. Existe também uma réplica do registo de configuração principal (registo sombra) que permite manter a configuração do Motor de Dados, enquanto o registo de configuração principal é modificado para conter a próxima configuração. Existe também uma memória de configurações, que permite armazenar configurações utilizadas frequentemente. Deste modo, após a escrita de uma configuração para o registo principal, e após a sua utilização no Motor de Dados, pode-se guardar o seu valor na memória de configurações para uma reutilização posterior.

2.5 Memória de instruções

A memória de instruções é constituída por uma memória RAM e uma memória ROM. A memória RAM tem 2048 posições enquanto a ROM tem 256 posições. No futuro poderá ser necessário aumentar o tamanho destas memórias. A memória RAM pode vir a transformar-se numa cache, para se suportar programas maiores.

O carregamento do programa realiza-se numa fase inicial, antes de se usar o Versat, sendo feito através da interface de controlo do Versat ou do DMA.

A ROM (também designada de boot ROM) contém um programa fixo para carregamento de programas do Versat ou dados, e para executar programas previamente carregados. O programa da boot ROM também é utilizado para descarregar os dados calculados pelo Versat. A RAM é usada após o carregamento do programa, para executá-lo.

2.6 Motor de transferências de dados

O motor de transferências de dados ou DMA permite ter acesso directo a uma memória externa, dispensando o sistema anfitrião de realizar estas transferências mais lentamente através da interface de controlo. A transferência de dados é feita em blocos de no máximo 256 palavras de 32 bits.

O DMA é operado pelo programa em execução no Versat que pode assim transferir dados da memória externa para as memórias do motor de dados ou vice-versa. O controlador do Versat tem, no seu mapa de memória, os registos que controlam o DMA. Estes são:

- Direcção da transacção, se a transferência é de dentro para fora ou vice-versa;
- Tamanho dos dados a transferir em palavras de 32 bit;
- Endereço de origem;
- Endereço de destino;
- Registo de estado.

O registo de estado é utilizado para indicar se o DMA está ocupado ou livre para iniciar uma nova transferência.

O DMA introduz um novo nível de paralelismo. É possível iniciar uma transferência no DMA enquanto o motor de dados está em funcionamento, com os devidos cuidados para não corromper os dados que o motor de dados está a utilizar. Ou pode aproveitar-se o tempo de transferência de dados para configurar o motor de dados para a próxima execução.

Também é possível carregar o subsistema de configurações através do DMA, fornecendo o endereço correcto.

Capítulo 3

Compilador do Versat

Neste capítulo é descrito o compilador para a arquitectura Versat desenvolvido no âmbito deste trabalho. A arquitectura do Versat é explicada no capítulo 2, sendo fundamental para compreender o compilador. Este capítulo está dividido em duas partes: a primeira explica a estrutura do compilador e a segunda a linguagem desenvolvida para o Versat, a qual foi inspirada na linguagem C++.

3.1 Estrutura do compilador

Um compilador converte um código numa linguagem definida (Java, C, C++, etc) para uma linguagem máquina. Como existem máquinas diferentes, tal resultaria num compilador diferente para cada linguagem e para cada máquina. Contudo, este inconveniente é minorado graças a uma representação intermédia interna ao compilador. O processo de compilação inclui os seguintes passos: o *front end*, optimização da representação intermédia, e o *back end*. No *front end* é realizada a conversão para a representação intermédia e no *back end* converte-se esta representação em código máquina. Assim, apenas o passo de *back end* é dependente da máquina.

O compilador do Versat tem apenas os passos de *front end* e *back end*. Na etapa de desenvolvimento em que incidiu este trabalho e devido à simplicidade dos programas contemplados, não foram implementados passos de optimização. A necessidade de passos de optimização surgirá naturalmente no futuro com o aumento da complexidade dos programas.

O compilador não produz o código máquina. Para a produção do código máquina, é necessário utilizar o *assembler* do Versat, o qual foi desenvolvido anteriormente.

3.1.1 Front end

O *front end* está dividido em duas fases: *scanning* (análise lexical) e *parsing* (análise sintáctica).

O *scanning* realiza a leitura de sequências de caracteres presentes na descrição do programa na linguagem do Versat, convertendo-os em *tokens*. Um *token* é uma sequência de símbolos detectada durante a análise lexical.

Depois da análise lexical, é feita a análise sintáctica. A análise sintáctica é a construção da árvore

de *parse*, em que se analisa uma sequência de *tokens* e detecta-se alguma regra gramatical compatível com a sequência recebida. Caso não seja detectada nenhuma regra gramatical, o compilador informa o utilizador que existe um erro.

Depois de concluída a análise sintáctica, é preenchida a estrutura de dados intermédia, com o objectivo de, através dela, produzir o *assembly*.

No compilador do Versat, tal como nos compiladores convencionais, a estrutura de dados intermédia é construída usando ferramentas que efectuem as análises lexical e sintáctica. As ferramentas utilizadas para a construção do *front end* são o Flex (analisador lexical) e o Bison (analisador sintáctico).

O Flex lê sequências de caracteres e converte-os em *tokens*, sendo responsável por reconhecer palavras-chave na linguagem, ignorar espaços em branco e comentários e reconhecer números em vários formatos.

Depois de detectados os *tokens*, é feita a análise sintáctica usando o programa Bison. O Bison é um gerador de *parsers*. Recebe *tokens* sequencialmente e procura uma regra gramatical entre as definidas pelo programador que seja compatível com a sequência que recebeu. Também avisa caso exista alguma ambiguidade gramatical. Caso não seja detectada nenhuma regra gramatical compatível com a sequência recebida, activa o estado *yyerror* e informa que houve um erro.

Considere-se o seguinte ciclo *while* em linguagem do Versat (para mais detalhes sobre o comando *while* ver a secção 3.2.12):

```
while (R1<10)
```

A árvore de *parse* gerada pelo Bison está representada na figura 3.1.

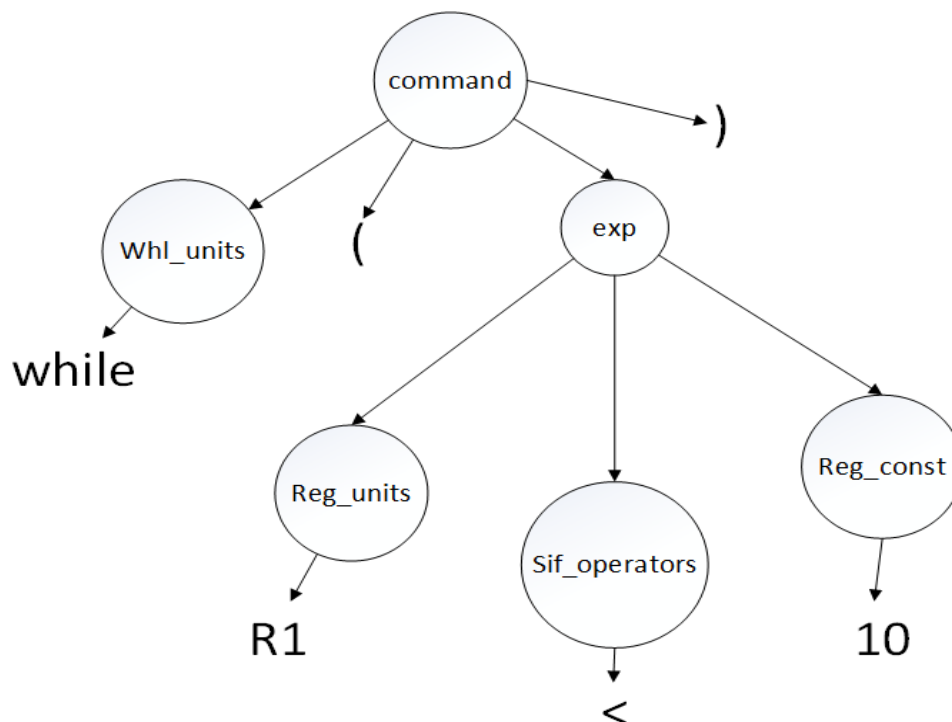


Figura 3.1: Árvore gerada do ciclo *while*.

A respeito da figura 3.1, refere-se que:

- `whl_units` é a folha que espera receber o *token* `while`;
- `reg_units` é a folha que recebe o registo usado no primeiro operando;
- `sif_operators` tem os operadores condicionais;
- `reg_const` representa as constantes;
- `exp` representa uma expressão;
- `command` representa um comando.

Ao detectar um comando este é guardado na estrutura de dados intermédia que não é mais do que uma lista de comandos. A cada comando está associada a chamada de funções responsáveis pelo preenchimento da estrutura.

Considere-se agora o seguinte comando que atribui o valor 2 ao parâmetro *start* do gerador de endereços do porto B da memória 0:

```
mem0B.setStart(2);
```

A árvore do *parse* gerada pelo Bison está representada na figura 3.2.

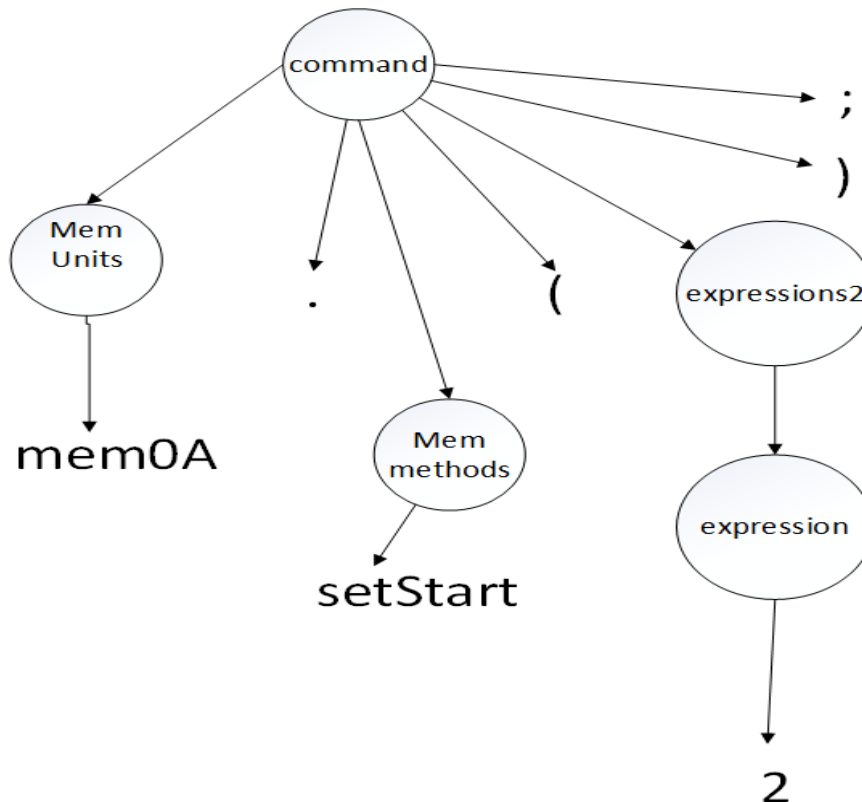


Figura 3.2: Árvore da instrução de configuração do gerador de endereços.

Acerca da figura 3.2, é importante referir que:

- Mem units é a memória em causa mais o respectivo porto;
- Mem_methods é o método em causa;
- Expressions2 e expression estão no caminho que gera a expressão que permite calcular o parâmetro.

Considere-se agora a seguinte expressão em linguagem do Versat:

$R5 = R4 + R3 + (2 - R4) ;$

Para processar expressões deste tipo é utilizada uma estrutura de dados em forma de árvore binária, que se associa ao objecto comando. A árvore de *parse* da expressão mostra-se na figura 3.3, onde Reg_target é o registo onde o resultado vai ser guardado e expressions são os nós responsáveis por reconhecer e guardar a árvore binária que representa a expressão.

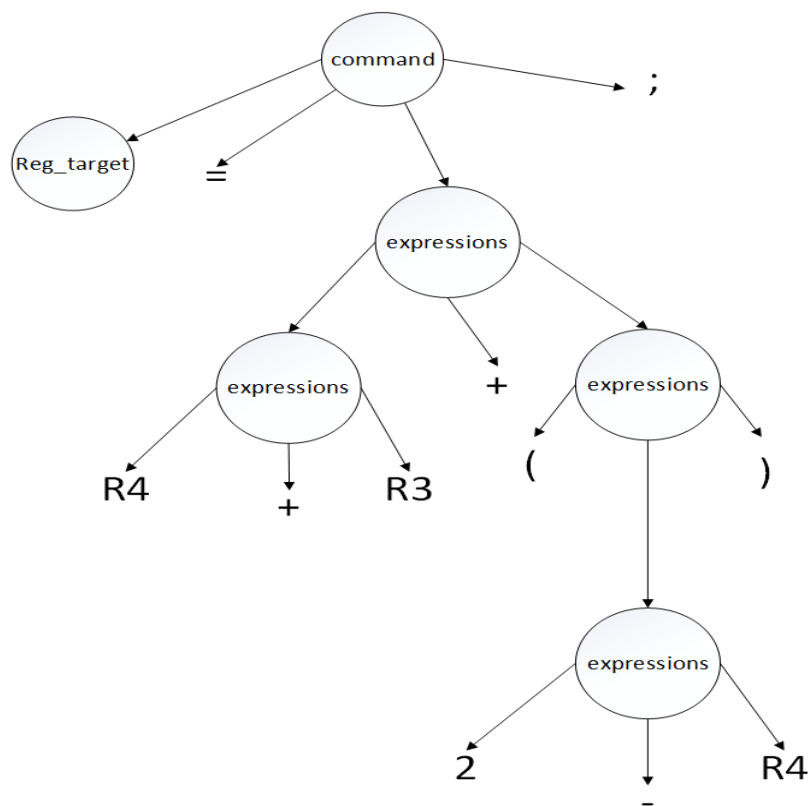


Figura 3.3: Árvore gerada de uma expressão de registos.

3.1.2 Back end

Depois de concluído o *parse*, a estrutura intermédia de dados está preenchida. A estrutura de dados é uma lista de elementos do tipo comando, que tem todas a informação necessária para gerar o *assembly*. A lista é percorrida do principio ao fim linearmente, apesar de cada comando poder ir buscar informação a comandos anteriores. Ao percorrer-se a lista, lê-se cada comando e depois gera-se o respectivo *assembly*.

Os atributos principais do objecto comando estão na tabela 3.1. Existem outros atributos auxiliares não dados na tabela.

Tabela 3.1: Atributos do objecto comando.

Atributo	Descrição
tipo de comando	Cada comando pode ser do tipo expressão, salto, ciclo, expressão, configuração, conexão, etc.
unidade	Unidade do motor de dados que está a ser usada.
porto	Porto da memória a ser usado.
método	Método em causa. Pode ser um método de configuração de ALUs, de memórias, de registo de controlo, de estado, etc.
árvore de registos	Apontador para a raiz da árvore de expressões de registos. A ordem das operações é representada por esta árvore.
operador	Operação a usar. Útil para a configuração de unidades funcionais.
operando1	Origem do primeiro operando da unidade funcional. Pode ser indicado por constante ou registo.
operando2	Origem do segundo operando da unidade funcional. Pode ser indicado por constante ou registo.
tipo do operando1	Indica se é constante ou registo.
tipo do operador2	Indica se é constante ou registo.
árvore de memórias	Apontador para a raiz da árvore de expressões de memória. A ordem das operações é representada por esta árvore.
etiqueta	Etiqueta usada para o salto incondicional.
genFU	Gerador do código <i>assembly</i> das instruções de configuração ou de conexão das unidades funcionais excepto as memórias.
genMem	Gerador do código <i>assembly</i> das instruções configuração ou de conexão das memórias.
ctrlReg	Gerador do código <i>assembly</i> do registo de controlo e de estado do motor de dados (ver secção 2.3).
controller	Gerador do código <i>assembly</i> relacionado com o controlador.
forCycle	Estrutura de dados do ciclo <i>for</i> . Apenas o ciclo <i>for</i> usa esta estrutura de dados, os outros ciclos usam os outros atributos já existentes.

Quando um comando está a ser processado, o primeiro atributo a ser lido é o tipo de comando. Consoante o tipo de comando serão processados os dados correspondentes. Por exemplo, caso o tipo de comando seja uma expressão de memórias ou de registos, será lida a respectiva árvore e gerado o respectivo *assembly*.

Para conseguir compilar código que use ciclos ou condições é necessário ir buscar informação a comandos anteriores. As condições e os ciclos utilizam etiquetas para identificar a instrução para onde se salta e geram-nas automaticamente. As etiquetas são numeradas internamente por um contador que conta o número de etiquetas em utilização concatenando-o com o identificador da etiqueta. Por exemplo, considere-se o seguinte código em linguagem do Versat:

```
1 int main() {
2     for (R1=0;R1<10;R1=R1+1) {
3         R5 = R5+R1; }
4
5 return 0; }
```

Este exemplo é um exemplo de um ciclo *for* em C++ do Versat. Os registos são usados como se fossem variáveis sem ser necessário declarar. O respectivo código *assembly* gerado é:

```
1     ldi 0
2     wrw R1
3 forDirectStatment0    rdw R1
4     addi -10
5     wrw RB
6     ldi 0
7     ldih 0x8000
8     and RB
9     beqi forStatment0
10    nop
11    nop
12    rdw R5
13    add R1
14    wrw R5
15    rdw R1
16    addi 1
17    wrw R1
18    ldi 0
19    beqi forDirectStatment0
20    nop
21 forStatment0 nop
22    ldi 0
23    beqi 0
24    nop
```


Na linha 3, a etiqueta usada tem como objectivo voltar ao início do ciclo. Na linha 19 é onde se dá o salto para a linha 3. Nas linhas 4 a 9 é feita a verificação da condição de continuar o ciclo. Caso a condição não seja cumprida, é feito o salto da linha 9 para a linha 21. As linhas 12 a 17 são a operação a realizar dentro do ciclo. As instruções das linhas 10 e 11 podem ser necessárias, pois o controlador do Versat executa sempre as duas instruções a seguir a seguir a um salto. As linhas 22 a 24 são o equivalente ao *return 0*.

Considere-se novamente a seguinte expressão em linguagem do Versat:

$$R5 = R4 + R3 + (2 - R6);$$

A respectiva árvore binária que representa a expressão é armazenada num único objecto do tipo comando e é mostrada na figura 3.4.

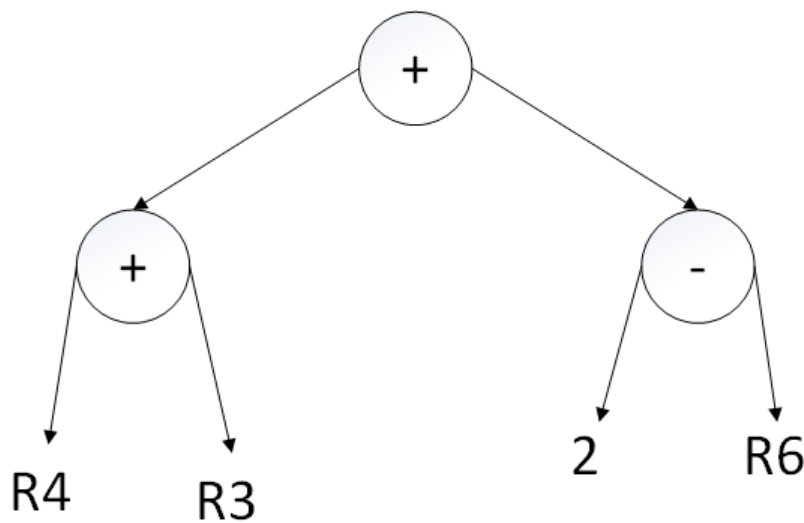


Figura 3.4: Árvore de representação de expressão.

Esta árvore dá origem à geração do seguinte código:

```

1      ldi 2
2      sub R6
3      wrw RB
4      rdw R4
5      add R3
6      add RB
7      wrw R5
8      ldi 1
9      wrw R6
10     ldi 0
11     beqi 0
12     nop
13     ldi 0
  
```

```

14      beqi 0
15      nop

```

As linhas 1 a 3 são o equivalente à expressão $(2 - R6)$. Como está entre parêntesis, é a primeira parte da expressão a ser calculada. O registo RB é usado para armazenar este resultado temporário. O resto da operação é realizada por ordem, pois não existem mais prioridades matemáticas a seguir.

A linguagem do Versat também suporta expressões envolvendo memórias do motor de dados. Considere-se o seguinte código em linguagem do Versat que ilustra um ciclo duplo com uma expressão de memórias no seu corpo:

```

1      for (j=0;j<R14;j++) {
2          for (i=0;i<R2;i++) {
3              mem0B[R1+j*R13+i] =
4                  (mem1A[R1+j*R13+i] * mem2B[1025+j*R2+R10*i]) -
5                  (mem0A[R1+j*R13+i] * mem2A[1024+j*R2+R10*i]);
6          } }

```

A árvore que representa esta expressão de memórias é mostrada na figura 3.5. Esta permite elaborar um mapa das conexões entre as várias unidades funcionais do motor de dados e configurar as unidades funcionais (consultar a secção 3.2.16). O código *assembly* gerado está no anexo B.

No ciclo *for* exterior é indicado o número de iterações a fazer. No ciclo *for* interior é indicado o período. Estes valores e outros extraídos dos índices dos portos de memória indicados na expressão permitem configurar os geradores de endereços. Repare-se que apenas 6 linhas de código geram 117 linhas de código *assembly*.

Graças às expressões de memórias, o programador não tem de calcular manualmente os parâmetros de configuração das unidades funcionais. No entanto, também é suportada a configuração manual das unidades funcionais para casos não abrangidos pelas expressões de memórias.

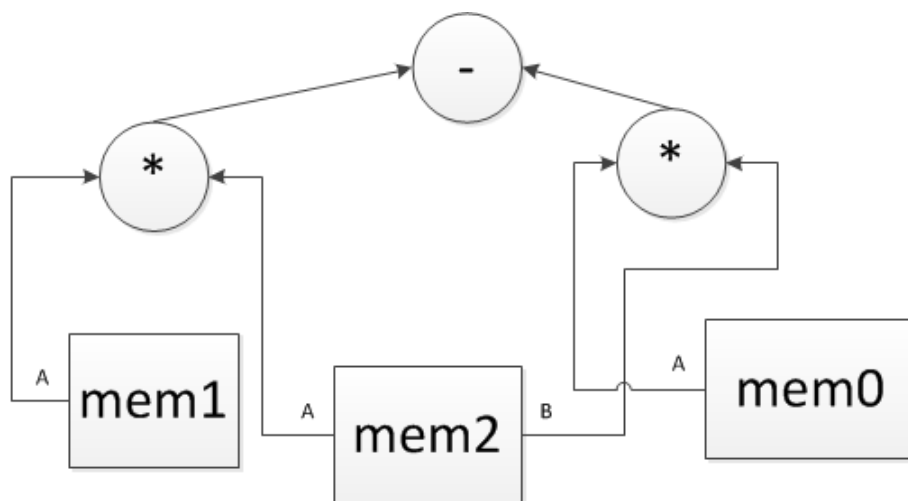


Figura 3.5: Esquema de circuito de uma expressão de multiplicações com subtracção.

3.2 Descrição da linguagem do Versat

Como já foi referido, optou-se por uma linguagem semelhante à linguagem C++ para a linguagem do compilador. Esta decisão advém do facto de esta linguagem ser conhecida, o que facilita o trabalho de um programador, melhorando o processo de aprendizagem. Entretanto, é conveniente referir que o Versat não tem actualmente memória disponível para uma *stack* própria, não sendo suportada a definição de funções pelo utilizador.

Como os recursos são limitados, na linguagem do Versat existem uma série de objectos e variáveis predefinidos. Os objectos e variáveis predefinidos representam características do hardware do Versat, que assim ficam expostas ao programador. Tal como na linguagem C, é possível trabalhar a um nível próximo da máquina, mas com uma sintaxe muito mais inteligível que a do *assembly*. Utilizando os objectos e variáveis predefinidos é possível configurar o motor de dados configurando individualmente cada unidade funcional. Chamamos a isto configuração *manual*.

Também é possível configurar o motor de dados de forma mais automática utilizando o que designamos por *expressões de memórias*. Estas expressões estão contidas no corpo de um ciclo *for* ou de dois ciclos *for* aninhados. Nestas expressões de memórias apenas as memórias são indicadas explicitamente, daí o seu nome. Os operadores, ALUs e multiplicadores são alocados automaticamente.

Dado um *datapath*, é necessário ter em conta que uma programação correcta em C++ do Versat consiste por fazer primeiro as configurações manuais necessárias, o que aloca os recursos necessários. Depois das configurações manuais, pode-se fazer as expressões de memórias. Esta ordem tem de ser cumprida para o compilador saber quais os recursos em uso nas configurações manuais, para não usar esses recursos nas expressões de memórias (ver secção 3.2.16).

Depois das expressões de memórias vêm eventuais conexões manuais que possam ser necessárias. A ideia do C++ do Versat é usar as expressões de memórias o máximo possível, embora existam situações em que não é possível a sua utilização.

3.2.1 Objectos e variáveis predefinidos

Os elementos principais da arquitectura do Versat estão expostos ao programador sob a forma de objectos e variáveis predefinidos, os quais estão descritos na tabela 3.2.

3.2.2 `int main`

Tal como em C/C++ regular, os programas na linguagem do Versat começam na função *main*. Não são suportadas as variáveis *argc* e *argv*. Directivas de pré-processador, tais como `#define` e `#include`, não são suportadas. Um exemplo de um pequeno programa na linguagem do Versat é dado a seguir:

```
1 int main() {  
2     R4=5;  
3     return 0; }
```

Tabela 3.2: Objectos e variáveis predefinidos.

Objecto/Variável	Tipo	Sufixo	Descrição
mem	objecto	[0-3]	Memórias existentes. Utilizado para facilitar a indicação de memórias a iniciar/arrancar (ver secção 3.2.3).
mem	objecto	[0-3]/[A-B]	Vector que permite aceder ao conteúdo da memória indicada pelo porto indicado. Usado nas expressões de memórias.
alu	objecto	[0-1]	ALUs existentes.
aluLite	objecto	[0-3]	ALULites existentes.
mult	objecto	[0-3]	Multiplicadores existentes.
bs	objecto	0	O deslocador (<i>barrel shifter</i>) existente.
de	objecto	-	Motor de dados
dma	objecto	-	Motor de DMA.
R	variável	[1-15]	Registos para uso como variáveis. Também existe o registo R0, mas está reservado para uso da boot ROM.
Ralu	variável	[0-1]	Registos de saída das unidades ALU. Podem ser usados como registos de propósito geral.
RaluLite	variável	[0-3]	Registos de saída das unidades ALULite. Podem ser usados como registos de propósito geral.
Rmult	variável	[0-3]	Registos de saída das unidades MULT. Podem ser usados como registos de propósito geral.
Rbs	variável	-	Registos de saída da unidade BS. Pode ser usado como registo de propósito geral.
i, j	variáveis	-	Iteradores de ciclos <i>for</i> .

Este programa é executado quando o sistema anfitrião dá uma instrução de execução à boot ROM do Versat, passando-lhe o endereço do programa na memória de instruções do Versat. Simplesmente atribui o valor 5 ao registo R4 e retorna à *boot* ROM para aguardar mais instruções do anfitrião.

Não é obrigatório colocar o comando de retorno, pois quando não existe comando de retorno o programa retorna à *boot* ROM quando chega ao final do programa.

3.2.3 Métodos para controlo do motor de dados

Os métodos para controlo do motor de dados são os métodos do objecto *de*, que se encontram descritos na tabela 3.3.

Estes métodos traduzem-se em instruções de escrita do registo de controlo do motor de dados ou de leitura do registo de estado do motor de dados. O método *run* escreve no registo de controlo, ao passo que o método *wait* lê do registo de estado. O compilador constrói uma máscara para escrever ou ler desses registos. Para atribuir cada bit da máscara usa-se a equação 3.1:

$$mask = mask + 2^{REG.SIZE-INDEX.FU} \quad (3.1)$$

onde *mask* é a máscara utilizada, REG.SIZE é o número de *bits* do registo e INDEX.FU é índice do bit

Tabela 3.3: Métodos para controlo do motor de dados

Método	Descrição
run(FU, FU, ...)	Inicializa e executa o motor de dados. Recebe como argumento os portos de memória a correr e as unidades funcionais a reinicializar. Adiciona automaticamente à lista de argumentos todas as unidades funcionais da última expressão de memórias.
wait(memPort, memPort, ...)	Espera que o motor de dados acabe de correr. Recebe como argumento os portos de memória que se pretende esperar que conclua o seu trabalho. Se for dado sem argumentos espera até que todos os portos de memória acabem o seu trabalho.
autoDuty(bit)	Indica se se quer que o parâmetro <i>duty</i> seja calculado automaticamente (bit=1) ou não (bit=0). Por defeito está habilitado.
clearConfig()	Reinicializa configurações e marca unidades funcionais como não estando em uso. Reinicializa e desaloca todas as unidades funcionais.
loadConfig(int)	Carrega uma configuração de um dado endereço da memória de configurações para o Motor de Dados. Realoca os recursos usados pela configuração. Recebe o endereço como argumento.
saveConfig(int)	Guarda a configuração actual do Motor de Dados na memória de configurações. Guarda também os recursos usados pela configuração. Recebe o endereço como argumento.

correspondente à unidade funcional no registo em causa.

As expressões de memória (ver secção 3.2.16) configuram o parâmetro *duty* automaticamente com um valor igual ao período *per*. A excepção ocorre quando o programador deseja construir circuitos com realimentação, algo que é difícil fazer apenas com expressões de memória. Nesses casos, o *duty* deve ser calculado manualmente e colocado através do método *setDuty*. O método existente está indicada na tabela 3.4.

3.2.4 Métodos exclusivos das memórias

Os métodos existentes para a configuração dos geradores de endereços dos portos das memórias encontram-se descritos na tabela 3.4. Ver a secção 2.3 onde estão explicados os vários parâmetros.

Tabela 3.4: Métodos respectivos à configuração dos portos das memórias.

Método	Descrição
setStart(expression)	Atribui o parâmetro <i>start</i> .
setDuty(expression)	Atribui o parâmetro <i>duty</i> .
setIncr(expression)	Atribui o parâmetro <i>incr</i> .
setIter(expression)	Atribui o parâmetro <i>iter</i> .
setPer(expression)	Atribui o parâmetro <i>per</i> .
setShift(expression)	Atribui o parâmetro <i>shift</i> .
setDelay(expression)	Atribui o parâmetro <i>delay</i> .
setReverse(expression)	Atribui o parâmetro <i>reverse</i> .

Com estes métodos o compilador mantém a possibilidade de programar manualmente os geradores de endereços. Alguns exemplos de utilização destes métodos são dados a seguir:

```
mem0A.setDuty(R2);
mem0A.setPer(R2);
```

```
mem0A.setIncr(1);
mem0B.setIter(R6-1);
```

Entretanto, oferece-se também a possibilidade do programador utilizar expressões de memórias. Estas configuram automaticamente todo o motor de dados, incluindo os geradores de endereços. Na secção 3.2.16 serão analisadas as expressões de memórias.

3.2.5 Métodos exclusivos das ALU e das ALULite

O único método existente para configuração das ALU e das ALULite pode ser visualizado na tabela 3.5.

Tabela 3.5: Método para configuração da ALU e ALULite.

Método	Descrição
setOper(oper)	Define uma operação para a ALU seleccionada.

As operações que são permitidas fazer nas ALU estão descritas na tabela 3.6.

Tabela 3.6: Operações permitidas pela ALU.

Operação	Descrição
+	Operação de soma.
-	Operação de subtracção.
&	And lógico.
	Or lógico.
~&	Nand lógico.
^	Xor lógico.
SEXT8	Extensão de sinal de 8 bits.
SEXT16	Extensão de sinal de 16 bits.
SRA	<i>Shift right</i> aritmético.
SRL	<i>Shift right</i> lógico.
SMPU	Comparação sem sinal. Coloca à saída o resultado da operação feita para a comparação.
SMPS	Comparação com sinal. Coloca à saída o resultado da operação feita para a comparação.
CLZ	Conta os bits que estão a 0 à esquerda.
MAX	Coloca à saída o máximo das duas entradas.
MIN	Coloca à saída o mínimo das duas entradas.
ABS	Coloca a saída o valor absoluto da entrada A.

As operações suportadas pela ALULite são as primeiras seis da tabela 3.6. Ao programa-las para uma tarefa que somente uma ALU pode realizar, o compilador dá um erro. Um exemplo de como indicar à alu0 para realizar a operação de soma é dado por:

```
alu0.setOper(' + ' );
```

Este método faz com que a ALU respectiva seja memorizada numa tabela de recursos usados, impedindo a sua utilização em expressões de memórias.

3.2.6 Métodos exclusivos do multiplicador

Os métodos existentes para configuração do multiplicador estão indicados na tabela 3.7.

Tabela 3.7: Métodos do multiplicador.

Método	Descrição
setLonhi(bit)	Indica ao multiplicador se é para colocar à saída a parte baixa (bit=1) ou a parte alta (bit=0, valor por defeito) do resultado de 64 bits.
setDiv2(bit)	Usado no formato de vírgula fixa Q1.31. O resultado aparece multiplicado por dois à saída (bit=0, valor por defeito), ou não (bit=1).

Exemplos do uso dos métodos são dados a seguir:

```
mult0 . setLonhi ( 1 );  
mult0 . setDiv2 ( 1 );
```

Tal como com a ALU e a ALULite, qualquer um destes métodos permite alocar o recurso na tabela de recursos usados.

3.2.7 Métodos exclusivos do deslocador

Os métodos exclusivos do deslocador (*barrel shifter*) são dados na tabela 3.8.

Tabela 3.8: Métodos exclusivos do deslocador.

Método	Descrição
setLNA(bit)	Escolhe se o deslocamento é aritmético (bit=1) ou lógico (bit=0, valor por defeito).
setLNR(bit)	Escolhe se o deslocamento é para a esquerda (bit=1) ou para a direita (bit=0, valor por defeito).

Exemplos do uso dos métodos são dados por:

```
bs0 . setLNA ( 1 );  
bs0 . setLNR ( 1 );
```

3.2.8 Métodos de conexão

Nesta secção podem ser encontrados os métodos de conexão que permitem ligar unidades funcionais umas às outras. Estes estão explicitados na tabela 3.9.

Exemplos do uso dos métodos de conexão são dados a seguir:

```
mem0A . connect ( mem2A );  
mem0B . connect ( alu0 );  
alu0 . connectPortA ( mult1 );
```

Tabela 3.9: Métodos de conexão.

Método	Descrição
connectPortA(FU)	Conecta o porto de entrada A de uma unidade funcional à saída da unidade funcional FU.
connectPortB(FU)	Conecta o porto de entrada B da unidade funcional à saída da unidade funcional FU.
connect(FU)	Conecta um porto de entrada de uma memória à saída da unidade funcional FU.

```
mult3 . connectPortB (mem2A);
```

3.2.9 Expressões de registos

A linguagem C++ do Versat suporta expressões com registos, cujo tamanho está limitado pelo facto de se usar apenas um registo para armazenar um resultado temporário. O recurso em causa é o registo RB que é utilizado como auxiliar no cálculo de expressões grandes; se for preciso armazenar o resultado de mais do que um nó da árvore que representa a expressão, então a expressão não é suportada. Isto acontece porque o Versat não possui uma pilha. As operações permitidas estão disponíveis na tabela 3.10.

Tabela 3.10: Operações suportadas pelas expressões de registos.

Método	Descrição
+	Soma.
-	Substracção.
&	And lógico.
>>	<i>Shift right.</i>
<<	<i>Shift left.</i>

Também é suportado o uso de parêntesis e é respeitada a ordem convencional das operações aritméticas em expressões: primeiro calcula-se o que está entre parêntesis, seguido dos deslocamentos e ANDs lógicos, finalizando com as operações de soma e substracção. Um exemplo do uso de expressões é dado por:

```
R1 = R2+R3+R4-R5&R6+R7-R8;
```

3.2.10 IF condicional

O compilador de C++ do Versat suporta expressões condicionais, assim como expressões condicionais encadeadas. A condição suporta apenas expressões com dois operandos. Exceptuando esta limitação, a condição é utilizada de forma análoga ao C++ regular. Um exemplo do uso da condição IF é:

```
if ( (R1+R3) != 0 ) {
    R5=2; }
```


3.2.11 ELSE condicional

O uso da condição ELSE é feito de forma idêntica ao C++ regular. Um exemplo de aplicação é:

```
if ( R5!=0 ) {  
    if (R6!=0)  R6=1;  
    else R6=2; }  
else {  
    R5=2; }
```

3.2.12 Ciclo while

Tal como nas condições, os ciclos são usados da mesma maneira que em C++ regular. No corpo do ciclo admitem-se expressões de registos, condicionais ou não. A limitação de suportar apenas expressões com dois operandos na condição de paragem mantém-se. Um exemplo de aplicação do ciclo *while* é:

```
1 int main() {  
2     R5=1000;  
3     while (R5<=40) {  
4         R5=6+R5;  
5         R7=R7+R4+2; }  
6     return 0; }
```

3.2.13 Ciclo for

Os ciclos *for* são iguais ao C++ regular mas no local do incremento, as expressões Rx++, Rx--, ++Rx e --Rx não são suportadas. Um exemplo de aplicação do ciclo *for* é:

```
1 int main() {  
2     for (R4=0;R4<20;R4=R4+1) {  
3         for (R5=0;R5<5;R5=R5+1) {  
4             R6=R7+R8+R6; }  
5         R7=R7+R4; }  
6     return 0; }
```

3.2.14 Ciclo do while

Tal como as outras expressões de ciclos, os ciclos *do while* são iguais ao C++ regular. Um exemplo de aplicação do ciclo *do while* é:

```
1 int main() {  
2     do {
```

```

3      do {
4          R1=R1+2+R5; }
5      while (R1!=20);
6      R5=R5+1; }
7  while( R5<50 );
8  return 0; }

```

3.2.15 Salto incondicional

O salto incondicional é implementado pelo comando *goto*, tal como em C++ regular. Este comando faz o programa saltar para uma posição identificada por uma etiqueta. Este comando em C++ regular raramente é usada, mas no C++ do Versat é útil na programação de pseudo-rotinas.

Existem etiquetas que não se podem utilizar na instrução *goto*, pois são utilizadas internamente pelo compilador para implementar as expressões de *if*, *while*, *for*, *do while* e o método *wait*. Estas etiquetas estão listadas na tabela 3.11. Um exemplo do uso da instrução *goto* é:

```

begin : R4=3;
goto begin ;

```

Tabela 3.11: Prefixos de etiquetas que o utilizador não pode usar.

Prefixos	Usado internamente com
ifStatment	if
whileStatment	while
whileDirectStatment	while
forDirectStatment	for
forStatment	for
elseStatment	else
waitres	wait

3.2.16 Expressões de memórias

Até aqui foram analisados elementos da linguagem do Versat que permitem configurar o motor de dados, configurando individualmente cada unidade funcional, o que é muito próximo do que se faz em linguagem *assembly*.

A característica mais importante do compilador desenvolvido é a capacidade de programar o motor de dados, incluindo várias unidades funcionais de uma só vez, a partir de um tipo de construção válida em C++ regular. Designou-se este tipo de construção por *expressões de memórias*, que são expressões que envolvem termos que são conteúdos das memórias do motor de dados, contidas dentro de ciclos *for*. Suportam-se expressões de memórias com um ciclo *for* ou no máximo dois ciclos *for* aninhados. Nas expressões de memórias apenas as memórias são invocadas explicitamente; os operadores implementados com ALUs e multiplicadores são alocados automaticamente. Por exemplo, uma expressão de memória com um ciclo *for* pode ser dada por:

```

for (j=0;j<iter;j++) {
    mem1A[ start+incr*j ] = mem0A[ start+incr*j ] + mem2B[ start+incr*j ];
}

```

em que se invocam explicitamente os portos de memória mem1A, mem0A e mem2B e o somador é alocado automaticamente. Nestas expressões podem constar também como termos os registos das unidades funcionais Ralu, Rmult e Rbs.

Note-se que os parâmetros *iter*, *start* e *incr* são constantes dadas pelo utilizador que são usadas na configuração dos geradores de endereço. Estes parâmetros só podem ser dados por constantes não podendo ser dados por expressões. Para expressões de memória com apenas um ciclo *for*, os parâmetros *per* e *shift* não aparecem nas expressões e são deixados com os seus valores por defeito.

Numa expressão de memória com dois ciclos *for* aninhados um porto de memória pode ser invocado da seguinte forma:

```

for (j=0;j<iter;j++) {
    for (i=0;i<per;i++)
        memXY[ start+j*perPLUSshift + incr*i ] = ...
}

```

O período *per* e o número de iterações *iter* são comuns a todos os geradores de endereços utilizados nas expressões. Os outros parâmetros variam consoante o porto de memória em causa. O parâmetro *perPLUSshift* é usado para extrair o parâmetro *shift* através da igualdade $shift = perPLUSshift - per$. Os recursos necessários para efectuar as operações entre memórias são indicados pelos operadores usados e alocados automaticamente.

Os multiplicadores usam os parâmetros *lonhi* e *div2* com o valor por defeito que é zero. É possível realizar multiplicações com *lonhi* = 1 e *div2* = 1 usando a notação '*' em vez de *.

Além das multiplicações, é permitido usar qualquer operação compatível com as ALULites (ver tabela 3.6).

O parâmetro *delay* é calculado internamente pelo compilador nas expressões de memórias, o que constitui uma vantagem de relevo, dado que o cálculo manual deste parâmetro torna a programação do Versat significativamente mais difícil. O algoritmo para o cálculo do *delay* utiliza a árvore extraída duma expressão de memórias. Uma vez que os portos de memória constituem as folhas da árvore, o *delay* de um determinado porto é dado pela equação 3.2.

$$delay = h - l \quad (3.2)$$

em que *h* é a altura da árvore (ie, o comprimento do caminho mais longo) e *l* é o comprimento do caminho que contém o porto. O cálculo do comprimento dos caminhos leva em conta as latência das unidades funcionais. O comprimento de um caminho é a soma das latências das unidades funcionais nesse caminho.

Alguns exemplos da aplicação das expressões de memórias são dados a seguir.

```

for (j=0;j<1023;j++) {
    for (i=0;i<10;i++) {
        mem0A[j*1+i*2]=mem2A[j*1+i*3] + mem2B[1024+j*1];
        mem0B[1024+j*1]=mem3B[1024+j*1+i*5];
        mem1A[j*1]=mem3A[j*1+i*2]; }    }

```

O compilador efectua a verificação dos parâmetros das memórias. Se num ciclo houver duas configurações diferentes para o mesmo porto de memória, o compilador acusa erro. Se num ciclo o mesmo porto aparecer repetido em duas árvores, o compilador verifica se o parâmetro *delay* tem o mesmo valor nas duas árvores, reportando um erro em caso negativo.

No processamento das expressões de memórias, o compilador mantém um registo dos recursos usados na tabela existente para esse fim. Uma expressão de memórias usa apenas os recursos não alocados, dando erro no caso de não existirem recursos suficientes. A libertação de recursos ocorre apenas quando se chama o comando `de.clearConfig()`.

Uma boa programação em C++ do Versat é fazer primeiro as configurações manuais, com o objectivo de alocar essas unidades funcionais e não as deixar ser usadas pelas expressões de memórias. Configurar manualmente um recurso depois de uma expressão de memória é perigoso, pois esse pode ter sido usado na expressão.

3.2.17 Return

O comando de *return* faz o programa retornar ao programa da Boot ROM. Na ausência deste comando o programa retorna à Boot ROM no fim da função `main`. Um exemplo da utilização do comando *return* é o seguinte:

```

1 int main() {
2
3 if (R3 == 0)
4     return 0;
5
6 else
7     R3 = 0;
8 }

```

3.2.18 Asm

O comando *asm* permite embeber código *assembly* no código C++ do Versat. O comando *asm* varia consoante o compilador. No presente caso, optou-se por uma utilização análoga a do compilador *g++*. Um exemplo do uso do comando *asm* é o seguinte:

```

1 int main() {
2 asm {

```

```

3      Idi ALU.ADD
4      wrc ALU0.CONFIG.ADDR,ALU_CONF_FNS_OFFSET
5      Idi smul1
6      wrc ALU0.CONFIG.ADDR,ALU_CONF_SELA_OFFSET
7      Idi salu0
8      wrc ALU0.CONFIG.ADDR,ALU_CONF_SELB_OFFSET
9      Idi 1      }
10     return 0;  }

```

3.2.19 Comentários

Os comentários em C++ do Versat são exactamente iguais ao C++ regular. Um exemplo da utilização de comentários em C++ do Versat é dado a seguir:

```

// R4=0;
/* if (R5==0) {
    R2=R3+R4;  }  */

```

3.2.20 DMA

Na presente secção descrevem-se os métodos relacionados com o uso do DMA (ver a secção 2.6), os quais são dados na tabela 3.12.

Tabela 3.12: Métodos do DMA Engine.

Método	Descrição
config(ExtAddr, IntAddr, Size, Direction)	Configura o DMA. ExtAddr e IntAddr são os endereços na memória externa e interna, respectivamente, que podem ser dados por expressões de registos. Size é o número de dados a transferir que é no máximo 256. Direction é a direcção da transferência: 1 se for do endereço externo para o endereço interno e 2 se for do endereço interno para o endereço externo.
run()	Executa o DMA.
wait()	Espera que o DMA acabe a execução. No caso de erro na transferência de dados, este método termina a execução do programa deixando o estado de erro no registo R1.
setExtAddr(expression)	Indica o endereço da memória externa.
setIntAddr(expression)	Indica o endereço da memória interna.
setSize(const)	Indica ao DMA o número de dados a enviar. O valor máximo é 256.
setDirection(const)	Indica a direcção dos dados. Se o argumento for 1, a direcção do envio dos dados será do endereço externo para o endereço interno. Se o argumento for 2, a direcção do envio dos dados será do endereço interno para o endereço externo.

No DMA, a memória externa é uma RAM exterior ao Versat, enquanto que a memória interna são as memórias do motor de dados do Versat. Alguns exemplos de aplicação dos métodos do DMA são dados a seguir:

```

dma.config(DMA_EXT_ADDR, DMA_INT_ADDR+100, 128, 1);
dma.run();
dma.wait();
dma.setExtAddr(R4);
dma.setIntAddr(MEM0);
dma.setSize(128);
dma.setDirection(1);

```

É possível usar as etiquetas MEM[0-3] nas expressões dos endereços das memórias internas como mostrado acima. Além destas etiquetas, ele também reconhece as etiquetas *DMA_EXT_ADDR*, que representa o endereço externo onde o DMA parou e *DMA_INT_ADDR* representa o endereço interno onde o DMA parou.

3.2.21 Variáveis

O compilador do Versat suporta a declaração de apenas um tipo de variáveis, o tipo *node*, que é usado para representar unidades funcionais do motor de dados. As variáveis deste tipo podem ser atribuídas numa expressão de memórias.

As variáveis são úteis para adicionar manualmente unidades funcionais a um *datapath* construído com expressões de memórias. Um exemplo de aplicação de variáveis é dado a seguir:

```

node X;
node Y;

alu0.setOper(' + ');
alu0.connectPortB(alu0);

alu1.setOper(' + ');
alu1.connectPortB(alu1);

for(j=0;j<255;j++) {
    X=(mem0B[1+j]*'mem1B[1+j])-(mem0A[j]*'mem1A[j]);
    Y=(mem0A[j]*'mem1B[1+j])+(mem0B[1+j]*'mem1A[j]);
}

alu0.connectPortA(X);
alu1.connectPortA(Y);

```

Neste exemplo, as variáveis X e Y são usadas para capturar a saída das duas expressões de memória dentro do ciclo *for* e ligá-las à entrada dos acumuladores formados pelas unidades ALU0 e ALU1. O circuito resultante pode ser visto na figura 3.6. A implementação de acumuladores é difícil de realizar de forma eficiente apenas com expressões de memórias, donde que a utilização de variáveis é bastante útil neste caso e noutros.

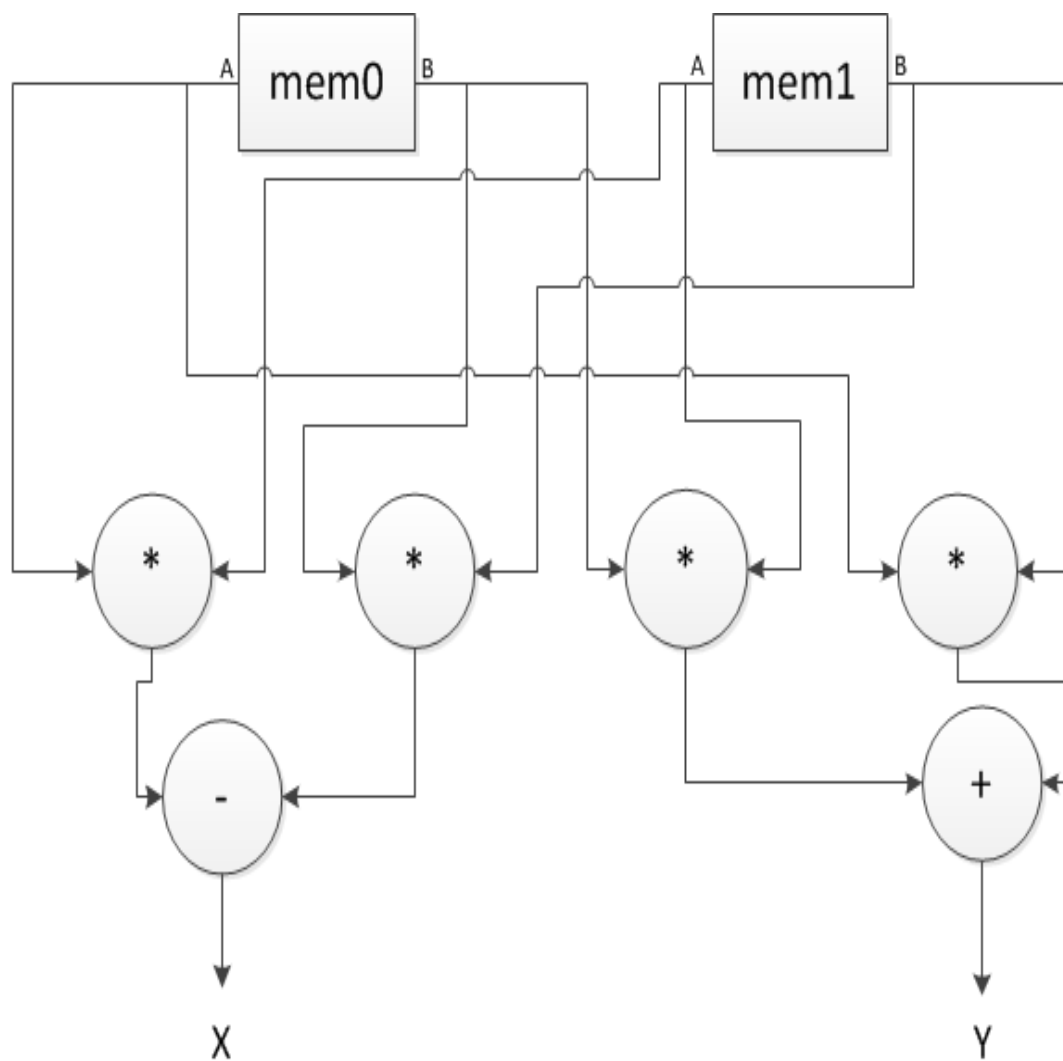


Figura 3.6: Esquema do *datapath* da expressão que exemplifica o uso de variáveis.

Capítulo 4

Exemplos de código

Neste capítulo são apresentados vários programas escritos na linguagem C++ do Versat, para exemplificar o uso do compilador. No programa da secção 4.1 usa-se apenas expressões de memórias para configurar o motor de dados. Nos programas das secções 4.2, 4.5 e 4.3 usa-se um misto de configurações manuais e expressões de memórias. Este misto é necessário para representar realimentações, pois não é possível representá-las apenas com expressões de memórias. No programa da secção 4.4 usa-se apenas configurações manuais. O programa descrito na secção 4.6 usa apenas o controlador, com o objectivo de exemplificar o uso de algumas operações do controlador.

4.1 Adição de vectores

Nesta secção considera-se uma soma de dois vectores. Usa-se os dois portos da memória 0 e guarda-se os resultados na memória 1. A ALU usada é escolhida internamente pelo compilador. O *datapath* do circuito está representado na figura 4.1.

O código respectivo em C++ do Versat é dado a seguir:

```
1 int main() {
2     de.clearConfig();
3     for(j=0;j<256;j++) {
4         mem1A[j*1]=mem0A[j*2]+mem0B[1+j*2]; }
5     de.run();
6     de.wait(mem1A);
7 return 0; }
```

Neste exemplo, são lidas as posições pares (porto A) e impares (porto B) da memória 0. O número de iterações a fazer é indicado no ciclo *for*.

É possível fazer o mesmo programa com instruções de configuração manual, mais perto do *assembly*. Contudo, o uso de configurações manuais é uma programação de mais baixo nível, sendo que o tamanho do programa em C++ seria também maior. Com expressões de memória, o compilador automatiza uma série de tarefas que melhoram muito a produtividade do programador. O código *assembly*

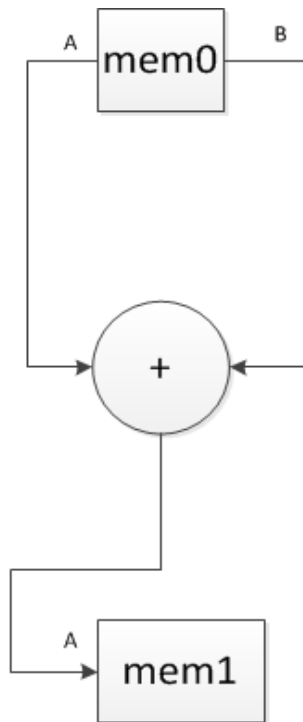


Figura 4.1: Esquema do *datapath* da adição de vectores.

gerado pelo compilador está listado a seguir:

```

1      wrw CLEAR_CONFIG_ADDR
2      ldi 255
3      wrw MEM0A_CONFIG_ADDR, MEM_CONF_ITER_OFFSET
4      ldi 2
5      wrw MEM0A_CONFIG_ADDR, MEM_CONF_INCR_OFFSET
6      ldi 0
7      wrw MEM0A_CONFIG_ADDR, MEM_CONF_DELAY_OFFSET
8      ldi 1
9      wrw MEM0B_CONFIG_ADDR, MEM_CONF_START_OFFSET
10     ldi 255
11     wrw MEM0B_CONFIG_ADDR, MEM_CONF_ITER_OFFSET
12     ldi 2
13     wrw MEM0B_CONFIG_ADDR, MEM_CONF_INCR_OFFSET
14     ldi 0
15     wrw MEM0B_CONFIG_ADDR, MEM_CONF_DELAY_OFFSET
16     ldi ALU_ADD
17     wrw ALU0_CONFIG_ADDR, ALU_CONF_FNS_OFFSET
18     ldi smem0A
19     wrw ALU0_CONFIG_ADDR, ALU_CONF_SELA_OFFSET
20     ldi smem0B
  
```

```

21      wrw ALU0_CONFIG_ADDR, ALU_CONF_SELB_OFFSET
22      ldi 255
23      wrw MEM1A_CONFIG_ADDR, MEM_CONF_ITER_OFFSET
24      ldi 1
25      wrw MEM1A_CONFIG_ADDR, MEM_CONF_INCR_OFFSET
26      ldi salu0
27      wrw MEM1A_CONFIG_ADDR, MEM_CONF_SELA_OFFSET
28      ldi 3
29      wrw MEM1A_CONFIG_ADDR, MEM_CONF_DELAY_OFFSET
30      ldi 4097
31      ldih 28
32      wrw ENG_CTRL_REG
33      wrw ENG_CTRL_REG
34      ldi 4098
35      ldih 28
36      wrw ENG_CTRL_REG
37 waitres0 ldi 8
38      and ENG_STATUS_REG
39      addi -8
40      bneqi waitres0
41      nop
42      ldi 0
43      beqi 0
44      nop
45      ldi 0
46      beqi 0
47      nop

```

As instruções de *assembly* entre as linhas 2 e 29 realizam a configuração do motor de dados para a expressão de memórias dada. Como todas as unidades funcionais estão livres e disponíveis para uso, o algoritmo de atribuição de unidades funcionais atribui logo a *alu0* para realizar a operação. Note-se que algumas configurações têm uma codificação diferente no *assembly/hardware* para melhor eficiência. Por exemplo, se o programador deseja colocar período igual a 2, este escreve 2 no programa em C++, mas é escrito 1 no *assembly*.

As linhas de código *assembly* entre a linha 30 e a 41 representam a inicialização e arranque do Motor de Dados. Entre a linha 42 e 44 está representado o comando *return0*.

Como existe apenas um ciclo *for*, os parâmetros *period*, o *duty* e o *shift* usam valores por omissão a nível do *hardware*, não sendo escritos para poupar tempo de configuração. Para evitar ficarem com valores de configurações anteriores (lixo), é importante usar o comando *de.clearConfig()* para reinicializar registo de configuração no início do programa.

4.2 Produto interno complexo

O produto interno complexo é dado pela equação 4.1,

$$v.u = \sum_{j=0}^N ((v_r[j] + v_i[j] * i)(u_r[j] + u_i[j] * i)) \quad (4.1)$$

onde, v_r e u_r são componentes reais dos vectores de entrada, v_i e u_i são componentes imaginárias dos vectores de entrada e N é o tamanho do vector.

O esquema das ligações do *datapath* do circuito está representado na figura 4.2. Notar os dois acumuladores antes da escrita do resultado para a memória 2, que realimentam as suas saídas para uma das suas entradas.

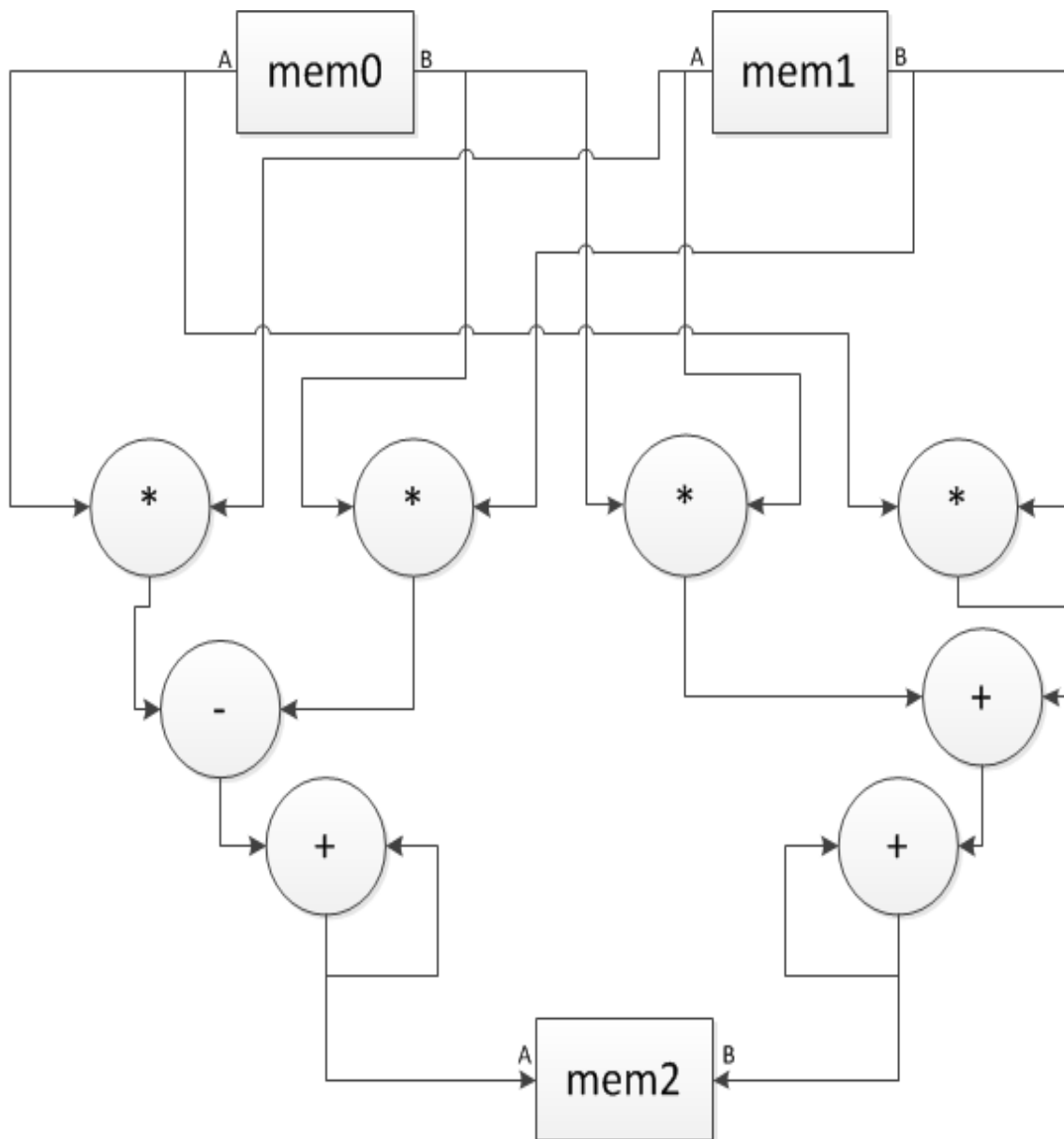


Figura 4.2: Esquema do *datapath* do produto interno complexo.

O código respectivo em C++ do Versat é:

```

1 int main() {
2     node X;
3     node Y;
4     de.clearConfig();
5     de.disableAutoDuty(1);
6     aluLite0.setOper('+');
7     aluLite1.setOper('+');
8     for(j=0;j<256;j++) {
9         for(i=0;i<2;i++) {
10             X=(mem0B[1+j*2+2*i]*'mem1B[1+j*2+2*i])
11             -(mem0A[j*2+2*i]*'mem1A[j*2+2*i]);
12             Y=(mem0A[j*2+2*i]*'mem1B[1+j*2+2*i])+
13             (mem0B[1+j*2+2*i]*'mem1A[j*2+2*i]); } }
14     aluLite0.connectPortA(X);
15     aluLite0.connectPortB(aluLite0);
16     aluLite1.connectPortA(Y);
17     aluLite1.connectPortB(aluLite1);
18     mem2A.connect(aluLite0);
19     mem2B.connect(aluLite1);
20     mem2A.setDelay(8);
21     mem2B.setDelay(8);
22     mem2A.setIter(256);
23     mem2B.setIter(256);
24     mem2B.setStart(1);
25     mem2A.setPer(2);
26     mem2B.setPer(2);
27     de.run(mem2, aluLite0, aluLite1);
28     de.wait(mem2A, mem2B);
29     return 0; }

```

Como é impossível fazer expressões com realimentação apenas com expressões de memórias, é necessário fazer um misto. As linhas 8 a 12 implementam um produto complexo com expressões de memórias.

Nas linhas 2 e 3, declaram-se 2 nós, X e Y, que servem para implementar as entradas dos dois acumuladores, limpando-se de seguida o registo de configuração na linha 4.

Os acumuladores, por terem realimentações, configuram-se nas linhas 6 e 7 (ALULite0 e ALULite1), o que impede o seu uso na expressões de memórias.

É necessário ter em conta que a latência das ALULites é 2 ciclos de relógio. Portanto, só aparecerão resultados novos nas saídas dos acumuladores de 2 em 2 ciclos. É necessário então configurar o período das expressões de memórias com o valor 2 e o *duty* com valor 1. Como pode defeito o *duty*

é configurado com o mesmo valor do período nas expressões de memórias é necessário desactivar o cálculo automático do *duty* na linha 5.

Nas expressões de memórias são capturados os valores do produto complexo nas variáveis X e Y. São usadas variáveis com o objectivo de não ser necessário ter conhecimento do algoritmo usado na atribuição de unidades funcionais. Quando se escreve o resultado numa variável, esta fica associada à unidade funcional na raiz da expressão, que pode assim ser ligada a outras.

Nas linhas 12 a 15 efectua-se as ligações dos acumuladores e nas linhas 16 a 24 faz-se a configuração manual da memória 2, dado que está fora das expressões de memória. Nas restantes linhas executa-se o motor de dados e espera-se pelo fim da execução.

4.3 FFT

A transformada rápida de Fourier (Fast Fourier Transform - FFT) é um algoritmo que calcula a transformada discreta de Fourier de um sinal digital. Também calcula a sua inversa.

O algoritmo usado para a FFT é o algoritmo de Cooley-Tukey. Considerando um exemplo de aplicação do algoritmo com apenas oito pontos, o algoritmo está esquematicamente representado na figura 4.3.

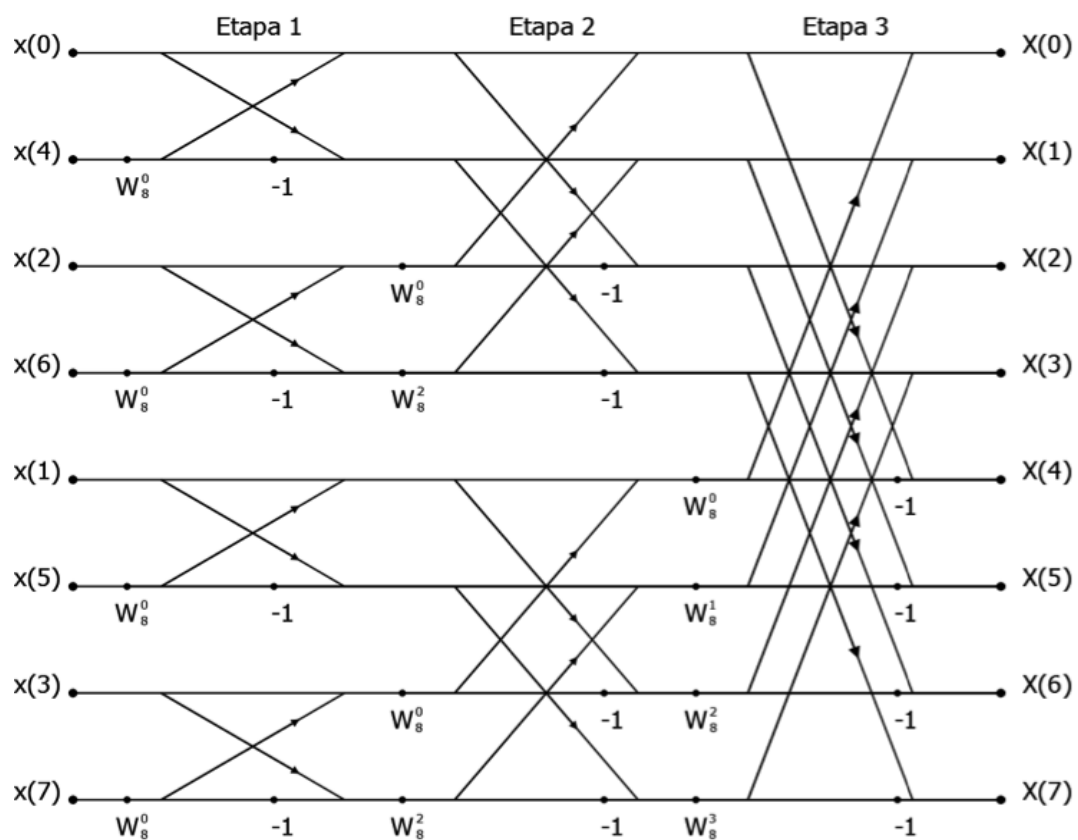


Figura 4.3: FFT de um sinal de entrada com 8 pontos.

O primeiro passo é realizar uma cópia de coeficientes e espelhamento dos endereços do sinal de entrada. A configuração que efectua esta operação é dada por:

```

for (j=0;j<1024;j++) {
    mem0B[1024+1*j]=mem2B[1024+1*j]; //copy coefficients
    mem0A[1*j]=mem2A[2*j]; //copy data while mirroring data addresses
    mem1A[1*j]=mem3A[2*j]; }
mem2A.setReverse(1); //instruction to mirror the address
mem3A.setReverse(1);
de.run();

```

O espelhamento é a troca de dados de uma dada posição de memória com a posição equivalente ao espelhamento. Por exemplo, olhando para a figura 4.3, o conteúdo da posição 1 (001) troca com o conteúdo da posição 4(100) e vice-versa.

Depois de feito o espelhamento do sinal de entrada, é possível começar a fazer as operações das *borboletas*. Cada borboleta realiza as operações definidas nas equações 4.2 e 4.3. A ilustração do cálculo está representada na figura 4.4.

$$A = a + W_N^r \times b \quad (4.2)$$

$$B = a - W_N^r \times b \quad (4.3)$$

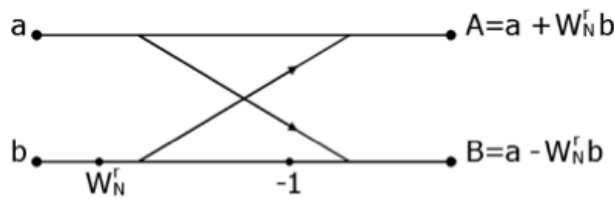


Figura 4.4: Borboleta da FFT

De seguida existem dois circuitos diferentes para fazer as borboletas: o circuito do produto complexo e o circuito da soma. O código do produto complexo é dado por:

```

for (j=0;j<R6;j++) {
    for (i=0;i<R14;i++) {
        mem0B[R1+j*R13+i] = (mem1A[R1+j*R13+i] *
mem2B[1025+j*R2+R10*i]) - (mem0A[R1+j*R13+i] * mem2A[1024+j*R2+R10*i]);
        mem1B[R1+j*R13+i] = (mem1A[R1+j*R13+i] *
mem2A[1024+j*R2+R10*i]) + (mem0A[R1+j*R13+i] * mem2B[1025+j*R2+R10*i]);} }

```

O código da soma de complexos é dado por:

```

for (i=0;i<R6;i++) {
    for (j=0;j<R14;j++) {
        mem2A[R7+j*R13+1*i] = mem0A[R7+j*R13+1*i] + mem0B[R1+j*R13+1*i];
        mem3A[R7+j*R13+1*i] = mem1A[R7+j*R13+1*i] + mem1B[R1+j*R13+1*i];

```

```

mem2B[R1+j*R13+1*i] = mem0B[R1+j*R13+1*i] - mem0A[R7+j*R13+1*i];
mem3B[R1+j*R13+1*i] = mem1B[R1+j*R13+1*i] - mem1A[R7+j*R13+1*i];
} }

```

O esquema equivalente deste *datapath* é dado na figura 4.5.

Existem duas variantes do código da FFT. Uma delas configura as expressões de memórias a cada nova iteração. Esta versão é apresentada no anexo A. Outra versão realiza as configurações necessárias todas antes do começo das iterações, guardando estas na memória de configurações. Dentro das iterações, as configurações que variam são feitas manualmente. Esta versão é apresentada no anexo C. Como se verá no capítulo dos resultados experimentais a versão com configurações parciais consegue reduzir significativamente o tempo de execução do programa, ilustrando uma característica inovadora determinante da arquitectura Versat.

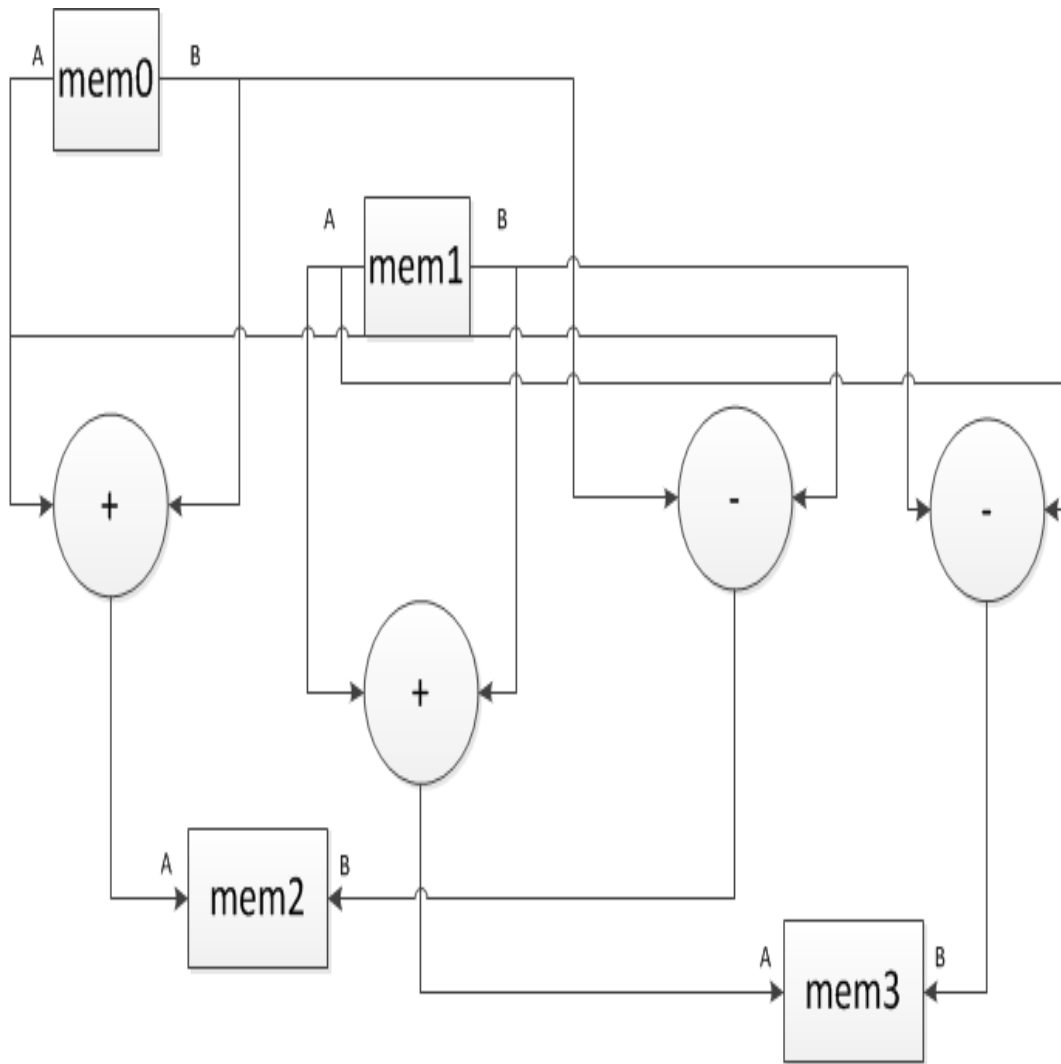


Figura 4.5: Esquema do *datapath* da soma de complexos.

4.4 Filtro passa baixo de primeira ordem

Considera-se um filtro passa baixo de primeira ordem. O filtro, construído através de C++ do Versat, é um filtro digital. O algoritmo usado para a construção do filtro é dado pela equação 4.4.

$$y(n) = 0.09x(n) + 0.91y(n - 1) \quad (4.4)$$

onde, x é o sinal de entrada e y é o sinal de saída. O circuito do Motor de Dados é dado na figura 4.6.

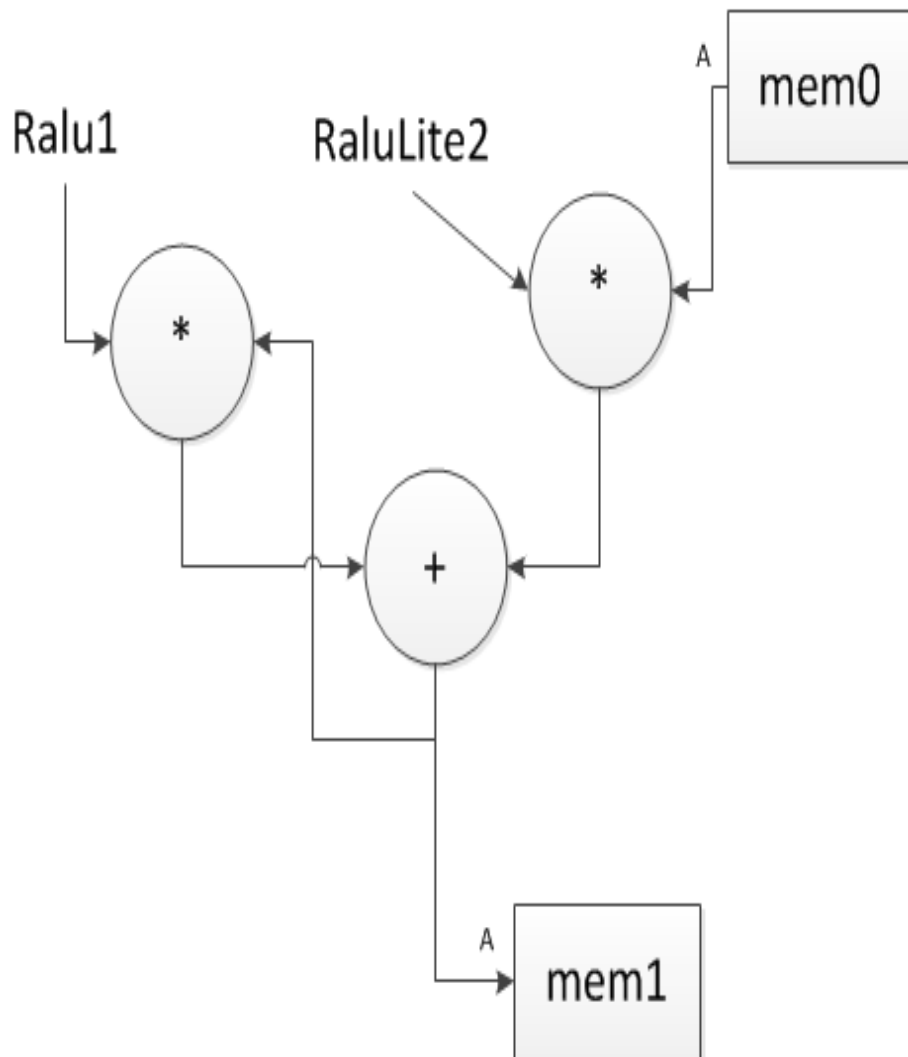


Figura 4.6: Esquema do *datapath* do filtro passa baixo.

O código em C++ do Versat para o filtro é dado por:

```
1 int main() {
2     de.clearConfig();
3     Ralu1 = 0x73333333;
4     RaluLite2 = 0x0CCCCCD;
5
6     mem0A.setIncr(1);
```

```

7      mem0A.setIter(512);
8      mem0A.setPer(5);
9      mem1A.connect(alu0);
10     mem1A.setDelay(6);
11     mem1A.setPer(5);
12     mem1A.setStart(1);
13     mem1A.setIncr(1);
14     mem1A.setIter(512);
15     mult0.connectPortA(alu1);
16     mult0.connectPortB(alu0);
17     mult1.connectPortA(aluLite2);
18     mult1.connectPortB(mem0A);
19     alu0.setOper(' + ');
20     alu0.connectPortA(mult0);
21     alu0.connectPortB(mult1);
22     de.run(mem0, mem1, mult0, mult1, alu0);
23     de.wait(mem1A);
24     return 0; }

```

Este caso podia também ser um misto de expressões de memória e configurações manuais, mas para se demonstrar como se implementa um programa apenas com configurações manuais optou-se por usar apenas configurações manuais.

As constantes do filtro foram armazenadas na saída das unidades funcionais ALU1 e ALULite2, ilustrando-se assim esta capacidade do Versat.

O número de iterações (linhas 7 e 14) representa o tamanho do vector de entrada. O período é 5 (linhas 8 e 11) por causa das latências do multiplicador e da alu0. É necessário esperar 5 ciclos de relógio até se preparar uma nova iteração. Como não se configurou nada no parâmetro *duty*, o valor por omissão é 1 (devido à limpeza de registos feita na linha 2). O *duty* omissivo para este caso indica que se lêem e escrevem novos dados de 5 em 5 ciclos.

4.5 Filtro passa baixo de segunda ordem

Considera-se um filtro passa baixo de segunda ordem dado pela equação às diferenças 4.5.

$$y(n) = x(n) + 1.8y(n-1) - 0.81y(n-2) \quad (4.5)$$

O circuito equivalente usado no Motor de Dados está ilustrado na figura 4.7.

As constantes usadas no cálculo estão guardadas na memória 2. Visto que cada memória tem apenas dois geradores de endereços, foi usada realimentação no circuito para representar $y(n-1)$. O sinal $y(n-1)$ está à saída do *barrel shifter*. O *barrel shifter* é usado para fazer cumprir a notação de

vírgula fixa desejada na equação às diferenças. O $y(n-2)$ vem do porto B da memória 1. As constantes 1.8 e -0.91 estão na memória 2.

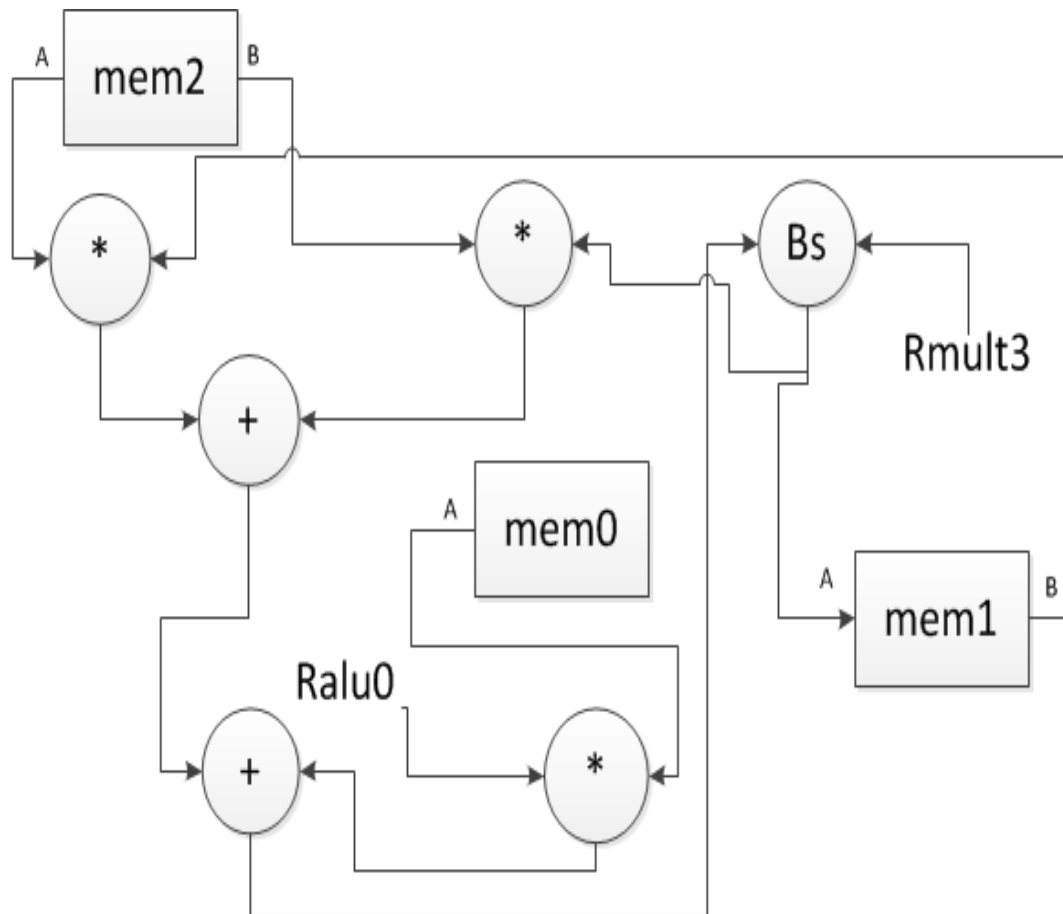


Figura 4.7: Esquema do Motor de Dados da equação às diferenças de segundo grau.

O código em C++ do Versat é dado por:

```

1 int main() {
2     de.clearConfig();
3     node Exp1;
4     node Exp2;
5     Ralu0 = 0x00a2339c;
6     Rmult3 = 1;
7     mem1A.connect(bs0);
8     mem1A.setDelay(9);
9     mem1A.setPer(8);
10    mem1A.setStart(2);
11    mem1A.setIncr(1);
12    mem1A.setIter(512);
13    mem2B.setStart(148);
14    mem2B.setPer(8);

```

```

15     mem2B.setIter(512);
16     aluLite3.setOper(' ');
17     alu1.setOper(' ');
18     de.disableAutoDuty(1);
19     mult0.connectPortA(mem2B);
20     mult0.connectPortB(bs0);
21     for(j=0;j<512;j++) {
22         for(i=0;i<8;i++) {
23             Exp1 = Ralu0*mem0A[j*8+i];
24             Exp2 = mem2A[147+j*8]*mem1B[j*8+i];        } }
25     aluLite3.connectPortA(Exp2);
26     aluLite3.connectPortB(mult0);
27     alu1.connectPortA(aluLite3);
28     alu1.connectPortB(Exp1);
29     bs0.connectPortA(alu1);
30     bs0.connectPortB(mult3);
31     bs0.setLNR(1);
32     de.run(mem1A, mem2B, aluLite3, alu1, mult0, bs0);
33     de.wait(mem2A); }

```

O MULT2 não precisa de ser configurado antes de se conectar (linhas 29 e 30), pois ele usa as configurações por defeito.

Todas as técnicas usadas neste código já foram explicadas nos exemplos anteriores mas reite-ramos aqui que: é necessário efectuar primeiro as configurações manuais e só depois é que confi-gura as expressões de memórias; o número de iterações é o tamanho do vector de entrada; devido à realimentação, o período é 8, isto é, as memórias enviam dados para processamento uma vez por cada 8 ciclos.

4.6 Raíz quadrada

Considera-se um programa de cálculo da raíz quadrada. Este programa não usa o Motor de Dados, foi feito apenas com o objectivo de testar o controlador. O método numérico usado é o método de Newton. O método de Newton é dado pela equação 4.6 e calcula aproximações cada vez mais precisas de uma raíz de uma função real.

$$x(n+1) = x(n) - \frac{f(x_n)}{f'(x_n)} \quad (4.6)$$

Para calcular a raíz quadrada de um dado valor é necessário usar a função dada na equação 4.7.

$$f(x) = x^2 - S \quad (4.7)$$

onde S é o valor do qual se pretende calcular a raíz.

O código em C++ do Versat para o cálculo da raíz quadrada é dado pelo seguinte código:

```
1 int main() {
2     R1 = 16; //S
3     R3=R1-1; //result
4     R5=0; //division S/xn
5     R8=0;
6     for(R2=0;R2<6;R2=R2+1) {
7         goto compDivision;
8 aaa:     R3 = (R3+R5)>>1;    }
9     R14 = R3;
10    return 0;
11
12 compDivision: R6=R3; //dummy
13    R5 = R1;
14    R8=0;
15    while(R5 >= R6) {
16        R5 = R5-R6;
17        R8=R8+1;
18    }
19    R5=R8;
20    goto aaa; }
```

Para calcular as divisões necessárias foi usado um algoritmo simples de divisão inteira, através de subtracções sucessivas (linhas 12 a 20). O número de iterações é fixo, pois não é usado nenhum algoritmo de cálculo do erro. No registo R1 é colocado o valor de entrada, enquanto que no registo R3 é colocado o resultado. Este algoritmo calcula apenas a parte inteira das raízes.

Capítulo 5

Resultados

Foram obtidos resultados experimentais utilizando os testes realizados para garantir o correcto funcionamento do compilador apresentados no capítulo 4. Para fácil referência foram adoptadas designações curtas dos vários exemplos:

`vec_add` (secção 4.1)

`complex_dot_product` (secção 4.2)

`fft` (secção 4.3, versão sem configurações parciais)

`fft_pr` (secção 4.3, versão com configurações parciais)

`lpf` (secção 4.4)

`lpf2` (secção 4.5)

Os resultados experimentais foram obtidos com o objectivo de estudar a eficiência do compilador. Existem dois factores que é necessário ter em conta para garantir a eficiência do compilador: o tempo de compilação e a eficiência do *assembly* gerado. Para analisar esta última compararam-se o número de linhas do *assembly* gerado com uma implementação manual dos mesmos programas em *assembly*, e mediram-se os tempos de execução das duas versões.

5.1 Tempo de compilação

O tempo de compilação para os diversos programas de teste é mostrado na tabela 5.1.

Os tempos de compilação são baixos, o que reflete a eficiência dos algoritmos usados. A parte mais complexa do compilador é a análise das expressões de memórias. Numa expressão de memória, a árvore que representa o circuito é percorrida na totalidade duas vezes: uma para gerar o circuito do motor de dados que a representa e outra para calcular o parâmetro *delay* de cada porto de memória. O tamanho máximo de uma árvore é 15 nós, o que equivale ao motor de dados completo.

Tabela 5.1: Tempo de compilação dos programas de teste.

Programa	Tempo de compilação [ms]
vec_add	3.458
complex_dot_product	5.202
fft	7.507
fft_pr	11.408
lpf	5.041
lpf2	5.346

5.2 Número de linhas do *assembly* gerado

Contabilizou-se o número de linhas de código *assembly* que foram geradas pelo compilador. Este número será comparado com a versão de cada programa escrita manualmente em *assembly*. Também se contabilizou o número de linhas escritas em C++ do Versat para cada programa. Os valores estão disponibilizados na tabela 5.2.

Tabela 5.2: Número de linhas de código dos programas de teste.

Programa	Linhas de código do programa em C++	Linhas de código do <i>assembly</i> gerado	Linhas de código do programa em <i>assembly</i>
vec_add	7	47	38
complex_dot_product	31	145	102
fft	98	959	830
fft_pr	258	1236	870
lpf	23	61	54
lpf2	33	111	85

Os programas lpf e lpf2 têm uma diferença entre o número de linhas de código C++ e o *assembly* gerado mais pequena em relação aos outros programas. Isto acontece devido ao facto de nesses dois programas se usar muitas configurações manuais (no lpf apenas se usa configurações manuais). As configurações manuais têm a vantagem de produzirem um *assembly* mais eficiente, mas têm a desvantagem de ser muito mais complicado para programar, pois o programador necessita de gerir directamente os recursos de *hardware* em comparação com as expressões de memórias. Apenas se aconselha o uso de configurações manuais quando certos circuitos com realimentações não são possíveis de construir apenas com expressões de memórias.

Nos restantes programas a relação entre o número de linhas do *assembly* gerado e o número de linhas de código em C++ é semelhante. Isto acontece devido ao uso de expressões de memórias. Em geral pode dizer-se que à medida que a complexidade do programa aumenta, torna-se mais vantajoso em termos de produtividade escrever em C++ do que em *assembly*.

A diferença entre o *assembly* gerado e o programa escrito manualmente em *assembly* deve-se essencialmente ao facto de o compilador não tentar memorizar em tempo de compilação o estado do acumulador e dos registos de configuração. O compilador faz as configurações serem sempre precedidas pelo carregamento do seu valor para o acumulador, independentemente se este já lá está ou não. Nas configurações manuais, muitas vezes o programador carrega um determinado valor para o

acumulador que depois é escrito em vários registos de configuração. Muitas vezes o compilador volta a configurar uma unidade funcional mesmo se a nova configuração é igual à configuração anterior, o que poucas vezes acontece na programação manual em *assembly*.

Isto sugere uma optimização para o futuro, que é a do compilador manter em memória o estado do acumulador e dos registos de configuração, de modo a evitar gerar instruções que os recarreguem com os mesmos valores que já tinham.

Mesmo assim, na maior parte dos programas o *assembly* gerado é parecido em tamanho com o *assembly* manual. Isto acontece porque os programas essencialmente configuram e mandam executar o motor de dados várias vezes. É para isso que basicamente serve o controlador do Versat. No caso do programa *fft_pr* a diferença de tamanho entre o *assembly* produzido e manual é maior, pois este exemplo utiliza muitas vezes configurações manuais em relação aos outros programas.

5.3 Tempo de execução

Contabilizou-se o tempo de execução de cada programa de teste, utilizando o simulador da arquitectura Versat já disponível. O tempo de execução é comparado com os tempos de execução da versão do programa de teste escrita manualmente em *assembly*. Os resultados estão disponibilizados na tabela 5.3.

Tabela 5.3: Tempos de execução dos programas de teste.

Programa	Ciclos de relógio do programa em C++	Ciclos de relógio do programa em <i>assembly</i>
<i>vec_add</i>	329	318
<i>complex_dot_product</i>	681	642
<i>fft</i>	14246	13038
<i>fft_pr</i>	12259	12112
<i>lpf</i>	2647	2645
<i>lpf2</i>	4227	4211

É possível concluir que a diferença entre o número de ciclos de relógio dos programas em C++ e dos programas em *assembly* é muito baixa. Isto demonstra que o compilador consegue produzir código *assembly* eficiente. A maior diferença ocorre no programa da FFT, devido ao uso mais recorrente do controlador e a ligeiras diferenças na forma como o algoritmo foi implementado. Outro factor determinante é o tamanho do programa da FFT em relação aos outros programas.

A diferença é praticamente nula nos programas *lpf* e *lpf2* provocada pelo uso de instruções de configuração manual, o que torna os programas compilados muito próximos dos escritos manualmente em *assembly*. Devido a darem origem a circuitos com realimentação nestes exemplos não foi possível usar exclusivamente expressões de memórias. Mesmo assim é muito mais cómodo realizar as configurações manuais usando os objectos predefinidos em C++ do que em *assembly*.

Os programas podem estar a usar o controlador, o Motor de Dados ou os dois ao mesmo tempo. Como o Motor de Dados é o componente mais importante de todo o sistema Versat, é importante que o

uso do controlador não condicione os programas de forma significativa. O número de ciclos de relógio em que os programas usam apenas o controlador e que usam o Motor de Dados (independentemente de se estar a usar o controlador em paralelo) estão na tabela 5.4.

Tabela 5.4: Tempos de execução dos programas no Motor de Dados e no controlador.

Programa	Ciclos de relógio de execução no Motor de Dados	Ciclos de relógio do uso exclusivo do controlador
vec_add	279	50
complex_dot_product	540	141
fft	11474	2812
fft_pr	11474	785
lpf	2586	61
lpf2	4124	103

Os valores da tabela 5.4 foram calculados através do ficheiro de teste, tal como o número de ciclos de relógio total. O ficheiro indica o número de ciclos de relógio total mais o número de ciclos exclusivos do controlador.

Na tabela 5.4 os parâmetros do número de uso exclusivo do controlador são associados ao programa em C++.

Com uma programação correcta, é possível efectuar configurações novas ao mesmo tempo que o Motor de Dados está a executar uma operação.

Capítulo 6

Conclusão

Os objectivos para esta tese foram cumpridos com sucesso. Uma primeira versão do compilador para a arquitectura Versat ficou completa.

6.1 Trabalho realizado

Neste trabalho, começou-se por fazer o estudo da arquitectura do Versat, com o objectivo de implementar o compilador.

O Versat tem uma arquitectura Coarse Grain Reconfigurable Arrays (CGRA), que tem as características inovadoras de conseguir gerar as suas configurações e de ser parcialmente reconfigurável. Se uma configuração difere pouco da configuração anterior, o Versat consegue escrever apenas as diferenças no registo de configuração. Mais ainda, o Versat consegue armazenar até 64 configurações completas numa memória especial para o efeito, para as aplicar mais tarde, fazendo ou não ajustes nos campos de configuração. Estas técnicas permitem esconder quase completamente o tempo de reconfiguração, que ocorre preferencialmente enquanto o motor de dados da CGRA está a trabalhar. Isto resulta num desempenho próximo do ideal.

Para implementar estas características, o Versat é dotado de um minúsculo processador de arquitectura convencional, suportando apenas 16 instruções. É uma máquina de acumulador com instruções de leitura e escrita, controlo de fluxo, lógica e aritmética simples. Todo o hardware do Versat está mapeado no espaço de endereçamento deste controlador e pode assim ser controlado e monitorizado.

O Versat é especializado em realizar operações em grandes quantidades de dados organizados de forma vectorial. Para aceder a estes dados, basta incrementar a posição da memória respectiva. Como qualquer CGRA o Versat possui um motor de dados constituído por unidades funcionais que podem ser interligadas e configuradas livremente, de acordo com o conteúdo do registo de configuração. Como nunca uma CGRA é grande o suficiente para nela se mapear qualquer computação, o Versat aposta na flexibilidade para decompor problemas grandes em subproblemas mais pequenos, reconfigurando-se as vezes que forem necessárias para os resolver.

O código é compilado para gerar um programa em assembly que é executado pelo controlador do

Versat depois de passar pelo assembler, o qual já existia no início deste trabalho. A sintaxe da linguagem do Versat é um subconjunto restrito da linguagem C++, o que permite a sua rápida aprendizagem. No compilador desenvolvido existem uma série de variáveis e objectos predefinidos que representam módulos e sinais do hardware, que fica assim exposto ao programador. Isto permite uma programação de baixo nível, próxima do assembly e da máquina, mas com a conveniência da inteligibilidade do C++. Este modo de programação permite a descrição estrutural de circuitos de dados (*datapaths*).

Para além deste modo de programação foi implementada uma programação de mais alto nível, utilizando o que se deu o nome de *expressões de memórias*. Sendo que as CGRAs são usadas essencialmente para executar ciclos de programa, neste modo de programação o compilador interpreta ciclos *for*, em cujo corpo estão expressões com vectores. Os vectores são as memórias internas do Versat, que são os únicos recursos que necessitam ser invocados explicitamente, donde a designação *expressões de memórias*. São permitidos até 2 ciclos *for* aninhados. As expressões no corpo do ciclo *for* podem envolver tantos operadores quantos existam disponíveis no Versat.

O compilador é responsável pela atribuição dos operadores a unidades funcionais e de as interligar. Visto que as unidades funcionais estão completamente interligadas, foi possível utilizar técnicas triviais de colocação de recursos e encaminhamento de sinais (*Place and Route*). A colocação é realizada atribuindo-se a primeira unidade funcional disponível do tipo desejado, dado que todas são equivalentes. O encaminhamento é feito seleccionando para cada entrada de cada unidade funcional em uso a saída de outra unidade funcional. A geração de configurações é implementada colocando as expressões de memórias ou descrições estruturais dentro de ciclos de programa do controlador, em que os parâmetros de configuração vão variando. A reconfiguração parcial faz-se alterando selectivamente os campos de configuração das unidades funcionais.

Fizeram-se diversos programas de teste para demonstrar as capacidades do compilador. Estudou-se a conveniência da linguagem comparando os programas de teste a implementações directas em assembly dos mesmos programas. Mediu-se o tempo de compilação e execução dos programas, que se encontra muito próximo das implementações em assembly. Mediu-se ainda o tempo de uso exclusivo do controlador, durante o controlo do fluxo de programa e reconfigurações, chegando-se à conclusão de que se trata de um tempo muito curto comparado ao tempo total de execução. Isto significa que a maior parte do tempo de execução do controlador decorre em paralelo com a operação do motor de dados, que é o elemento que está na realidade a produzir os resultados dos cálculos.

6.2 Trabalho Futuro

Como trabalho futuro é proposto o desacoplamento do compilador com a arquitectura Versat, com o objectivo de se obter um compilador mais independente da arquitectura. Para tal o primeiro passo é suportar o uso de ficheiros de configuração, que descrevem o *hardware* existente. Também se propõe a implementação de técnicas mais sofisticadas de *Place and Route*, com o objectivo de não ser obrigatório ter uma arquitectura completamente interligada.

Ao nível de melhoramentos mais modestos propõe-se a implementação de *Look-up Tables*, com o

objectivo de implementar funções de cálculo matemático, como por exemplo, o cálculo trigonométrico. Para tal é necessário acrescentar às memórias a possibilidade de usarem um dos portos como endereço. A ideia é carregar a memória com os valores da função em causa, e sempre que se necessita de efectuar uma operação, envia-se a posição de memória como valor de entrada, e à saída é dado o resultado da respectiva função.

Outro melhoramento interessante que pode ser implementado no curto prazo é o compilador manter a configuração actual do motor de dados em memória, poupando em instruções de configuração sempre que for necessário configurar um campo com um valor igual ao que já lá está. Na mesma ordem de ideias, o compilador podia manter em memória, se possível, o último valor presente no acumulador, evitando carregar o mesmo valor. Esta situação ocorre frequentemente quando é necessário configurar vários campos com o mesmo valor. O programador em assembly explora facilmente esta situação carregando o valor uma vez para o acumulador e escrevendo-o de seguida para os vários campos de configuração.

Anexo A

Código FFT apenas com configurações parciais

O código completo do programa fft é dado em:

```
1 int main() {
2   R3=0; //used to compute address increments
3   R4=0; //used to compute address increments
4   R5=0; //used to compute address increments
5   R6=512; //number of blocks
6   R8=1; //number of butterflies per block
7   R10=1024; //number of fft points
8   R11=1; //number of configs per stage
9   //mirror address bits and copy coefs
10  de.clearConfig();
11  mem2A.setReverse(1);
12  mem3A.setReverse(1);
13  for(j=0;j<1024;j++) {
14    //mirror address bits
15    mem0A[j]=mem2A[2*j];
16    mem1A[j]=mem3A[2*j];
17    //copy coefs
18    mem0B[1024+j]=mem2B[1024+j];  }
19  de.run();
20  //FFT main loop
21  for(R9=1;R9<11;R9++) { //iterate over all stages
22    //set registers used in partial reconfigurations
23    if(R9==7) {
24      R11=16;
```

```

25     R4=1;
26     R3=1024;
27     R6=R6<<1;    }
28   if (R9>6) {
29     R11=R11>>1;
30     R6=R6<<2;
31     R5=R8;        }
32   R14 = R8-R5+R4;
33   R13 = (R8-R5)<<1+R4; //address increment
34   R12=R11; //do-while iterator
35   R7=0; //start address offset
36   R2 = R3-1024+R14; //address increment
37   de.clearConfig();
38   R15=1&R9; //get stage parity
39   if (R15==1) { //stage is odd
40     //complex product b*w
41     do { //partial reconfiguration until all done
42       de.clearConfig();
43       R1=R8+R7;
44       for (j=0;j<R6;j++) {
45         for (i=0;i<R14;i++) {
46           mem0B[R1+j*R13+i] = (mem1A[R1+j*R13+i]*mem2B[1025+j*R2+R10*i])
- (mem0A[R1+j*R13+i]*mem2A[1024+j*R2+R10*i]);
47           mem1B[R1+j*R13+i] = (mem1A[R1+j*R13+i]*mem2A[1024+j*R2+R10*i])
+ (mem0A[R1+j*R13+i]*mem2B[1025+j*R2+R10*i]);    }      }
48     de.wait(mem0A);
49     de.run();
50     R7=R7+R6+R6;
51     R12=R12-1;
52   } while (R12!=0);
53   //complex sum a+b*w and a-b*w
54   R12=R11;
55   R7=0;
56   de.clearConfig();
57   do { //partial reconfiguration until all done
58     R1=R8+R7;
59     de.clearConfig();
60     for (j=0;j<R6;j++) {
61       for (i=0;i<R14;i++) {

```



```

62     mem2A[R7+j*R13+i] = mem0A[R7+j*R13+i]+mem0B[R1+j*R13+i];
63     mem3A[R7+j*R13+i] = mem1A[R7+j*R13+i]+mem1B[R1+j*R13+i];
64     mem2B[R1+j*R13+i] = mem0B[R1+j*R13+i]-mem0A[R7+j*R13+i];
65     mem3B[R1+j*R13+i] = mem1B[R1+j*R13+i]-mem1A[R7+j*R13+i];      }  }
66     de.wait(mem2A);
67     de.run();
68     R7=R7+R6+R6;
69     R12=R12-1;
70 } while(R12!=0);  }
71 else { // stage is even
72     //complex product b*w
73     do { //partial reconfiguration until all done
74         R1=R8+R7;
75         de.clearConfig();
76         for(j=0;j<R6;j++) {
77             for(i=0;i<R14;i++) {
78                 mem2B[R1+j*R13+i] = (mem3A[R1+j*R13+i]*mem0B[1025+j*R2+R10*i])
-(mem2A[R1+j*R13+i]*mem0A[1024+j*R2+R10*i]);
79                 mem3B[R1+j*R13+i] = (mem2A[R1+j*R13+i]*mem0B[1025+j*R2+R10*i])
+(mem3A[R1+j*R13+i]*mem0A[1024+j*R2+R10*i]);      }  }
80         de.wait(mem2A);
81         de.run();
82         R7=R7+R6+R6;
83         R12=R12-1;
84     } while(R12!=0);
85     //complex sum a+b*w and a-b*w
86     R12=R11;
87     R7=0;
88     do { //partial reconfiguration untill all done
89         de.clearConfig();
90         R1=R8+R7;
91         for(j=0;j<R6;j++) {
92             for(i=0;i<R14;i++) {
93                 mem0A[R7+j*R13+i] = mem2A[R7+j*R13+i]+mem2B[R1+j*R13+i];
94                 mem1A[R7+j*R13+i] = mem3A[R7+j*R13+i]+mem3B[R1+j*R13+i];
95                 mem0B[R1+j*R13+i] = mem2B[R1+j*R13+i]-mem2A[R7+j*R13+i];
96                 mem1B[R1+j*R13+i] = mem3B[R1+j*R13+i]-mem3A[R7+j*R13+i];      }  }
97         de.wait(mem0A);
98         de.run();

```

```

99      R7=R7+R6+R6;
100     R12=R12-1;
101     } while (R12!=0);      }
102     R8 = R8 << 1;
103     R10 = R10 >> 1;
104     R6=R6>>1;
105 } //end of main loop
106 de.wait(mem0, mem1, mem2, mem3); }

```

Anexo B

Código assembly do exemplo do ciclo for

```
1      rdw R1
2      wrd MEM1A_CONFIG_ADDR, MEM_CONF_START_OFFSET
3      rdw R14
4      addi -1
5      wrd MEM1A_CONFIG_ADDR, MEM_CONF_ITER_OFFSET
6      ldi 1
7      wrd MEM1A_CONFIG_ADDR, MEM_CONF_INCR_OFFSET
8      rdw R13
9      sub RB
10
11     wrd MEM1A_CONFIG_ADDR, MEM_CONF_SHIFT_OFFSET
12     rdw RB
13     addi -1
14     wrd MEM1A_CONFIG_ADDR, MEM_CONF_PER_OFFSET
15     rdw RB
16     addi -1
17     wrd MEM1A_CONFIG_ADDR, MEM_CONF_DUTY_OFFSET
18     ldi 0
19     wrd MEM1A_CONFIG_ADDR, MEM_CONF_DELAY_OFFSET
20     ldi 1025
21     wrd MEM2B_CONFIG_ADDR, MEM_CONF_START_OFFSET
22     rdw R14
23     addi -1
24     wrd MEM2B_CONFIG_ADDR, MEM_CONF_ITER_OFFSET
25     rdw R10
```

```

26      wrd MEM2B_CONFIG_ADDR, MEM_CONF_INCR_OFFSET
27      rdw R2
28      sub RB
29
30      wrd MEM2B_CONFIG_ADDR, MEM_CONF_SHIFT_OFFSET
31      rdw RB
32      addi -1
33      wrd MEM2B_CONFIG_ADDR, MEM_CONF_PER_OFFSET
34      rdw RB
35      addi -1
36      wrd MEM2B_CONFIG_ADDR, MEM_CONF_DUTY_OFFSET
37      ldi 0
38      wrd MEM2B_CONFIG_ADDR, MEM_CONF_DELAY_OFFSET
39      ldi smem1A
40      wrd MULT0_CONFIG_ADDR, MUL_CONF_SELA_OFFSET
41      ldi smem2B
42      wrd MULT0_CONFIG_ADDR, MUL_CONF_SELB_OFFSET
43      rdw R1
44      wrd MEM0A_CONFIG_ADDR, MEM_CONF_START_OFFSET
45      rdw R14
46      addi -1
47      wrd MEM0A_CONFIG_ADDR, MEM_CONF_ITER_OFFSET
48      ldi 1
49      wrd MEM0A_CONFIG_ADDR, MEM_CONF_INCR_OFFSET
50      rdw R13
51      sub RB
52
53      wrd MEM0A_CONFIG_ADDR, MEM_CONF_SHIFT_OFFSET
54      rdw RB
55      addi -1
56      wrd MEM0A_CONFIG_ADDR, MEM_CONF_PER_OFFSET
57      rdw RB
58      addi -1
59      wrd MEM0A_CONFIG_ADDR, MEM_CONF_DUTY_OFFSET
60      ldi 0
61      wrd MEM0A_CONFIG_ADDR, MEM_CONF_DELAY_OFFSET
62      ldi 1024
63      wrd MEM2A_CONFIG_ADDR, MEM_CONF_START_OFFSET
64      rdw R14

```

```

65      addi -1
66      wrd MEM2A_CONFIG_ADDR, MEM_CONF_ITER_OFFSET
67      rdw R10
68      wrd MEM2A_CONFIG_ADDR, MEM_CONF_INCR_OFFSET
69      rdw R2
70      sub RB
71
72      wrd MEM2A_CONFIG_ADDR, MEM_CONF_SHIFT_OFFSET
73      rdw RB
74      addi -1
75      wrd MEM2A_CONFIG_ADDR, MEM_CONF_PER_OFFSET
76      rdw RB
77      addi -1
78      wrd MEM2A_CONFIG_ADDR, MEM_CONF_DUTY_OFFSET
79      ldi 0
80      wrd MEM2A_CONFIG_ADDR, MEM_CONF_DELAY_OFFSET
81      ldi smem0A
82      wrd MULT1_CONFIG_ADDR, MUL_CONF_SELA_OFFSET
83      ldi smem2A
84      wrd MULT1_CONFIG_ADDR, MUL_CONF_SELB_OFFSET
85      ldi ALU_SUB
86      wrd ALU0_CONFIG_ADDR, ALU_CONF_FNS_OFFSET
87      ldi smul0
88      wrd ALU0_CONFIG_ADDR, ALU_CONF_SELA_OFFSET
89      ldi smul1
90      wrd ALU0_CONFIG_ADDR, ALU_CONF_SELB_OFFSET
91      rdw R1
92      wrd MEM0B_CONFIG_ADDR, MEM_CONF_START_OFFSET
93      rdw R14
94      addi -1
95      wrd MEM0B_CONFIG_ADDR, MEM_CONF_ITER_OFFSET
96      ldi 1
97      wrd MEM0B_CONFIG_ADDR, MEM_CONF_INCR_OFFSET
98      rdw R13
99      sub RB
100
101      wrd MEM0B_CONFIG_ADDR, MEM_CONF_SHIFT_OFFSET
102      rdw RB
103      addi -1

```

```
104    wrd MEM0B_CONFIG_ADDR, MEM_CONF_PER_OFFSET
105    rdw RB
106    addi -1
107    wrd MEM0B_CONFIG_ADDR, MEM_CONF_DUTY_OFFSET
108    ldi salu0
109    wrd MEM0B_CONFIG_ADDR, MEM_CONF_SELA_OFFSET
110    ldi 6
111    wrd MEM0B_CONFIG_ADDR, MEM_CONF_DELAY_OFFSET
112    ldi 0
113    beqi 0
114    nop
115    ldi 0
116    beqi 0
117    nop
```

Anexo C

Código em C++ da FFT usando a memória de configurações

```
1 int main() {
2   R3=0; //used to compute address increments
3   R4=0; //used to compute address increments
4   R5=0; //used to compute address increments
5   R6=512; //number of blocks
6   R8=1; //number of butterflies per block
7   R10=1024; //number of fft points
8   R11=1; //number of configs per stage
9   //mirror address bits and copy coefs
10  de.clearConfig();
11  mem2A.setReverse(1);
12  mem3A.setReverse(1);
13  for(j=0;j<1024;j++) {
14    //mirror address bits
15    mem0A[j]=mem2A[2*j];
16    mem1A[1*j]=mem3A[2*j];
17    //copy coefs
18    mem0B[1024+1*j]=mem2B[1024+1*j]; }
19  de.run();
20  //complex product b*w
21  de.clearConfig();
22  for(j=0;j<R6;j++) {
23    for(i=0;i<R14;i++) {
24      mem0B[i] = (mem1A[i]*mem2B[1025]) - (mem0A[i]*mem2A[1024]);
```

```

25     mem1B[i] = (mem1A[i]*mem2A[1024])+(mem0A[i]*mem2B[1025]);    } }
26 de.saveConfig(0);
27 //complex sum a+b*w and a-b*w
28 de.clearConfig();
29 for(j=0;j<R6;j++) {
30     for(i=0;i<R14;i++) {
31         mem2A[i] = mem0A[i]+mem0B[i];
32         mem3A[i] = mem1A[i]+mem1B[i];
33         mem2B[i] = mem0B[i]-mem0A[i];
34         mem3B[i] = mem1B[i]-mem1A[i];    } }
35 de.saveConfig(1);
36 //complex product b*w
37 de.clearConfig();
38 for(j=0;j<R6;j++) {
39     for(i=0;i<R14;i++) {
40         mem2B[i] = (mem3A[i]*mem0B[1025])-(mem2A[i]*mem0A[1024]);
41         mem3B[i] = (mem2A[i]*mem0B[1025])+(mem3A[i]*mem0A[1024]);    } }
42 de.saveConfig(2);
43 //complex sum a+b*w and a-b*w
44 de.clearConfig();
45 for(j=0;j<R6;j++) {
46     for(i=0;i<R14;i++) {
47         mem0A[i] = mem2A[i]+mem2B[i];
48         mem1A[i] = mem3A[i]+mem3B[i];
49         mem0B[i] = mem2B[i]-mem2A[i];
50         mem1B[i] = mem3B[i]-mem3A[i];    } }
51 de.saveConfig(3);
52 //FFT main loop
53 for(R9=1;R9<11;R9=R9+1) { //iterate over all stages
54     //set registers used in partial reconfigurations
55     if(R9==7) {
56         R11=16;
57         R4=1;
58         R3=1024;
59         R6=R6<<1;    }
60     if(R9>6) {
61         R11=R11>>1;
62         R6=R6<<2;
63         R5=R8;    }

```



```

64   R14 = R8-R5+R4;
65   R13 = R8-R5+R14; // address increment
66   R12=R11; //do-while iterator
67   R7=0; // start address offset
68   R2 = R3-1024+R14; // address increment
69   R15=1&R9; // get stage parity
70   if (R15==1) { // stage is odd
71       //complex product b*w
72       de.loadConfig(0);
73       //1 st expression
74       mem0B.setIter(R6);
75       mem0B.setPer(R14);
76       mem0B.setDuty(R14);
77       mem0B.setShift(R13-R14);
78       //2nd expression
79       mem1B.setIter(R6);
80       mem1B.setPer(R14);
81       mem1B.setDuty(R14);
82       mem1B.setShift(R13-R14);
83       //common
84       mem1A.setIter(R6);
85       mem1A.setPer(R14);
86       mem1A.setDuty(R14);
87       mem1A.setShift(R13-R14);
88       mem0A.setIter(R6);
89       mem0A.setPer(R14);
90       mem0A.setDuty(R14);
91       mem0A.setShift(R13-R14);
92       mem2B.setIter(R6);
93       mem2B.setPer(R14);
94       mem2B.setDuty(R14);
95       mem2B.setShift(R2-R14);
96       mem2B.setIncr(R10);
97       mem2A.setIter(R6);
98       mem2A.setPer(R14);
99       mem2A.setDuty(R14);
100      mem2A.setShift(R2-R14);
101      mem2A.setIncr(R10);
102      do { // partial reconfiguration until all done

```

```

103     mem0A.setStart(R8+R7);
104     mem0B.setStart(R8+R7);
105     mem1A.setStart(R8+R7);
106     mem1B.setStart(R8+R7);
107     de.wait(mem0A);
108     de.run();
109     R7=R7+R6+R6;
110     R12=R12-1;
111     } while(R12!=0);
112     //complex sum a+b*w and a-b*w
113     R12=R11;
114     R7=0;
115     de.loadConfig(1);
116     mem2A.setIter(R6);
117     mem2A.setPer(R14);
118     mem2A.setDuty(R14);
119     mem2A.setShift(R13-R14);
120     mem0A.setIter(R6);
121     mem0A.setPer(R14);
122     mem0A.setDuty(R14);
123     mem0A.setShift(R13-R14);
124     mem0B.setIter(R6);
125     mem0B.setPer(R14);
126     mem0B.setDuty(R14);
127     mem0B.setShift(R13-R14);
128     mem3A.setIter(R6);
129     mem3A.setPer(R14);
130     mem3A.setDuty(R14);
131     mem3A.setShift(R13-R14);
132     mem1A.setIter(R6);
133     mem1A.setPer(R14);
134     mem1A.setDuty(R14);
135     mem1A.setShift(R13-R14);
136     mem1B.setIter(R6);
137     mem1B.setPer(R14);
138     mem1B.setDuty(R14);
139     mem1B.setShift(R13-R14);
140     mem2B.setIter(R6);
141     mem2B.setPer(R14);

```

```

142     mem2B.setDuty(R14);
143     mem2B.setShift(R13–R14);
144     mem3B.setIter(R6);
145     mem3B.setPer(R14);
146     mem3B.setDuty(R14);
147     mem3B.setShift(R13–R14);
148     do { // partial reconfiguration until all done
149     mem0A.setStart(R7);
150     mem0B.setStart(R8+R7);
151     mem1A.setStart(R7);
152     mem1B.setStart(R8+R7);
153     mem2A.setStart(R7);
154     mem2B.setStart(R8+R7);
155     mem3A.setStart(R7);
156     mem3B.setStart(R8+R7);
157     de.wait(mem2A);
158     de.run();
159     R7=R7+R6+R6;
160     R12=R12–1;
161     } while (R12!=0);    }
162 else { // stage is even
163     //complex product
164     de.loadConfig(2);
165     //1 st expression
166     mem2B.setIter(R6);
167     mem2B.setPer(R14);
168     mem2B.setDuty(R14);
169     mem2B.setShift(R13–R14);
170     //2nd expression
171     mem3B.setIter(R6);
172     mem3B.setPer(R14);
173     mem3B.setDuty(R14);
174     mem3B.setShift(R13–R14);
175     //common
176     mem3A.setIter(R6);
177     mem3A.setPer(R14);
178     mem3A.setDuty(R14);
179     mem3A.setShift(R13–R14);
180     mem2A.setIter(R6);

```

```

181     mem2A.setPer(R14);
182     mem2A.setDuty(R14);
183     mem2A.setShift(R13-R14);
184     mem0B.setIter(R6);
185     mem0B.setPer(R14);
186     mem0B.setDuty(R14);
187     mem0B.setShift(R2-R14);
188     mem0B.setIncr(R10);
189     mem0A.setIter(R6);
190     mem0A.setPer(R14);
191     mem0A.setDuty(R14);
192     mem0A.setShift(R2-R14);
193     mem0A.setIncr(R10);
194     do { // partial reconfiguration until all done
195     mem3A.setStart(R8+R7);
196     mem3B.setStart(R8+R7);
197     mem2A.setStart(R8+R7);
198     mem2B.setStart(R8+R7);
199     de.wait(mem2A);
200     de.run();
201     R7=R7+R6+R6;
202     R12=R12-1;
203     } while(R12!=0);
204     //complex sum a+b*w and a-b*w
205     R12=R11;
206     R7=0;
207     de.loadConfig(3);
208     mem0A.setIter(R6);
209     mem0A.setPer(R14);
210     mem0A.setDuty(R14);
211     mem0A.setShift(R13-R14);
212     mem2A.setIter(R6);
213     mem2A.setPer(R14);
214     mem2A.setDuty(R14);
215     mem2A.setShift(R13-R14);
216     mem2B.setIter(R6);
217     mem2B.setPer(R14);
218     mem2B.setDuty(R14);
219     mem2B.setShift(R13-R14);

```

```

220     mem1A.setIter(R6);
221     mem1A.setPer(R14);
222     mem1A.setDuty(R14);
223     mem1A.setShift(R13–R14);
224     mem3A.setIter(R6);
225     mem3A.setPer(R14);
226     mem3A.setDuty(R14);
227     mem3A.setShift(R13–R14);
228     mem3B.setIter(R6);
229     mem3B.setPer(R14);
230     mem3B.setDuty(R14);
231     mem3B.setShift(R13–R14);
232     mem0B.setIter(R6);
233     mem0B.setPer(R14);
234     mem0B.setDuty(R14);
235     mem0B.setShift(R13–R14);
236     mem1B.setIter(R6);
237     mem1B.setPer(R14);
238     mem1B.setDuty(R14);
239     mem1B.setShift(R13–R14);
240     do { // partial reconfiguration untill all done
241     mem0A.setStart(R7);
242     mem0B.setStart(R8+R7);
243     mem1A.setStart(R7);
244     mem1B.setStart(R8+R7);
245     mem2A.setStart(R7);
246     mem2B.setStart(R8+R7);
247     mem3A.setStart(R7);
248     mem3B.setStart(R8+R7);
249     de.wait(mem0A);
250     de.run();
251     R7=R7+R6+R6;
252     R12=R12–1;
253     } while(R12!=0);    }
254     R8 = R8 << 1;
255     R10 = R10 >> 1;
256     R6=R6>>1;
257 } //end of main loop
258 de.wait(mem0, mem1, mem2, mem3); }

```


Bibliografia

- [1] Bingfeng Mei, A. Lambrechts, J.-Y. Mignolet, D. Verkest, and R. Lauwereins. Architecture exploration for a reconfigurable architecture template. *Design & Test of Computers, IEEE*, 22(2):90–101, March 2005.
- [2] Ming hau Lee, Hartej Singh, Guangming Lu, Nader Bagherzadeh, and Fadi J. Kurdahi. Design and implementation of the MorphoSys reconfigurable computing processor. In *Journal of VLSI and Signal Processing-Systems for Signal, Image and Video Technology*. Kluwer Academic Publishers, 2000.
- [3] V. Baumgarte, G. Ehlers, F. May, A. Nüchel, M. Vorbach, and M. Weinhardt. PACT XPP – a self-reconfigurable data processing architecture. *The Journal of Supercomputing*, 26(2):167–184, 2003.
- [4] M. Quax, J. Huiskens, and J. Van Meerbergen. A scalable implementation of a reconfigurable WCDMA RAKE receiver. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, 2004*, volume 3, pages 230–235 Vol.3, Feb 2004.
- [5] J.T. de Sousa, V.M.G. Martins, N.C.C. Lourenco, A.M.D. Santos, and N.G. do Rosario Ribeiro. Reconfigurable coprocessor architecture template for nested loops and programming tool, September 25 2012. US Patent 8,276,120.
- [6] Justin L Tripp, Jan Frigo, and Paul Graham. A survey of multi-core coarse-grained reconfigurable arrays for embedded applications. *Proc. of HPEC*, 2007.
- [7] Reiner Hartenstein. Coarse grain reconfigurable architecture (embedded tutorial). In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, ASP-DAC '01, pages 564–570, New York, NY, USA, 2001. ACM.
- [8] Bjorn De Sutter, Praveen Raghavan, and Andy Lambrechts. *Handbook of Signal Processing Systems*, chapter Coarse-Grained Reconfigurable Array Architectures, pages 553–592. Springer, 2 edition, 2013. ISBN: 978-1-4614-6858-5.
- [9] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger. Mapping applications onto reconfigurable kressarrays. In Patrick Lysaght, James Irvine, and Reiner Hartenstein, editors, *Field Programmable Logic and Applications: 9th International Workshop, FPL'99, Glasgow, UK, August 30 - September 1, 1999. Proceedings*, pages 385–390. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

- [10] Reiner Hartenstein. Coarse grain reconfigurable architecture (embedded tutorial). In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference, ASP-DAC '01*, pages 564–570, New York, NY, USA, 2001. ACM.
- [11] Bjorn Sutter, Praveen Raghavan, and Andy Lambrechts. Coarse-grained reconfigurable array architectures. In S. Shuvra Bhattacharyya, F. Ed Deprettere, Rainer Leupers, and Jarmo Takala, editors, *Handbook of Signal Processing Systems*, pages 449–484. Springer US, Boston, MA, 2010.
- [12] D.C. Cronquist, C. Fisher, M. Figueroa, P. Franklin, and C. Ebeling. Architecture design of reconfigurable pipelined datapaths. In *Advanced Research in VLSI, 1999. Proceedings. 20th Anniversary Conference on*, pages 23–40, Mar 1999.
- [13] Ebeling. The general rapid architecture description. *Tech. Rep. UW-CSE-02-06-02, University of Washington*, 2002.
- [14] Safaa J. Kasbah, Issam W. Damaj, and Ramzi A. Haraty. Multigrid solvers in reconfigurable hardware. *Journal of Computational and Applied Mathematics*, 213(1):79 – 94, 2008.
- [15] Arthur Abnous, Christopher Christensen, Jeffrey Gray, John Lenell, Andrew Naylor, and Nader Bagherzadeh. Design and implementation of the 'tiny risc' microprocessor. *Microprocessors and Microsystems - Embedded Hardware Design*, 16(4):187–193, 1992.
- [16] Mark Gebhart, Bertrand A. Maher, Katherine E. Coons, Jeff Diamond, Paul Gratz, Mario Marino, Nitya Ranganathan, Behnam Robatmili, Aaron Smith, James Burrill, Stephen W. Keckler, Doug Burger, and Kathryn S. McKinley. An evaluation of the trips computer system. *SIGPLAN Not.*, 44(3):1–12, March 2009.
- [17] Monica S Lam. Software pipelining: an effective scheduling technique for vliw machines. *ACM SIGPLAN Notices*, 39(4):244–256, 2004.
- [18] Henk Corporaal. *Microprocessor Architectures: From VLIW to Tta*. John Wiley & Sons, Inc., New York, NY, USA, 1997.
- [19] Salvatore M. Carta, Danilo Pani, and Luigi Raffo. Reconfigurable coprocessor for multimedia application domain. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, 44(1-2):135–152, 2006.
- [20] Vaughn Betz, Jonathan Rose, and Alexander Marquardt, editors. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.
- [21] Bjorn De Sutter, Paul Coene, Tom Vander Aa, and Bingfeng Mei. Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, LC-TES '08*, pages 151–160, New York, NY, USA, 2008. ACM.

- [22] Bingfeng Mei, Serge Vernalde, Diederik Verkest, and Rudy Lauwereins. Design methodology for a tightly coupled vliw/reconfigurable matrix architecture: A case study. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2, DATE '04*, pages 21224–, Washington, DC, USA, 2004. IEEE Computer Society.
- [23] Stephen Friedman, Allan Carroll, Brian Van Essen, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. Spr: an architecture-adaptive cgra mapping tool. In Paul Chow and Peter Y. K. Cheung, editors, *FPGA*, pages 191–200. ACM, 2009.
- [24] Taewook Oh, Bernhard Egger, Hyunchul Park, and Scott Mahlke. Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 2009 ACM SIG-PLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES '09*, pages 21–30, New York, NY, USA, 2009. ACM.
- [25] Joseph A. Fisher, Paolo Faraboschi, and Cliff Young. *Embedded computing - a VLIW approach to architecture, compilers, and tools*. Morgan Kaufmann, 2005.

