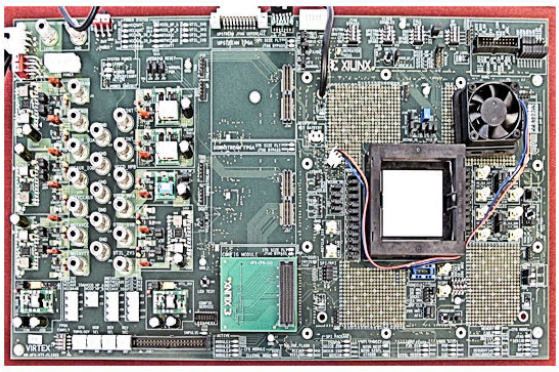
TÉCNICO LISBOA



Compiler for the VERSAT Reconfigurable Processor

Rui Manuel Alves Santiago

Relatório de Introdução à Investigação e Projecto de

Engenharia Electrotécnica e de Computadores

Júri

Orientador:	José Teixeira de Sousa
Vogal:	Paulo Flores

Maio de 2015

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

A computação reconfigurável (CR) tem tido um grande foco na investigação na última década. As máquinas reconfiguráveis mudam dinamicamente a sua arquitectura de acordo com as instruções executadas. Foi demonstrado que as CRs podem ter acelerações de grande magnitude em aplicações como processamento de sinal. Visto que o ramo das telecomunicações usa muitos sistemas de processamento de sinal, a importância dos CRs é enorme.

O Versat é um sistema que usa uma arquitectura reconfigurável. O objectivo principal do Versat é configurar e correr o Data Engine um certo número de vezes até ser produzido um resultado esperado. O Versat não está desenhado para um alto desempenho ao nível da execução de instruções. O controlador tem o mínimo de instruções possível e serve essencialmente para controlar a execução do Data Engine.

O Versat estará ligado a um sistema mestre, sendo o próprio um sistema escravo. A interface de controlo é usada para efectuar as leituras e escritas necessárias entre mestre e escravo.

O Data Engine é onde os cálculos de alto desempenho são executados. Os dados de entrada estão organizados em vectores colocados em memória RAM de porto duplo, sendo que cada um dos portos tem um gerador de endereços.

O problema deste tipo de sistemas é o desenvolvimento de um compilador adequado. Ainda não existe um compilador eficiente para este tipo de máquinas.

O objectivo deste trabalho é investigar o desenvolvimento de um compilador para o Versat. Deve ser levado em linha de conta que o Versat é um acelerador de *hardware* que usa uma arquitectura de computação reconfigurável. Portanto é de esperar que a implementação do compilador seja muito diferente de uma implementação clássica. Serão escritos vários programas na linguagem desenvolvida para demonstrar o funcionamento correcto do compilador. Será também feito um estudo da arquitectura do Versat, com o objectivo de entender melhor o *hardware* e perceber a forma como fazer o compilador o mais eficiente possível.

Capítulo 2

Trabalho anterior

As *Coarse Grain Reconfigurable Arrays* (CGRAs) ganharam atenção nas duas últimas décadas [1, 2, 3, 4, 5]. As CGRAs são estruturas de hardware programável que podem ser construídas com conjuntos de processadores RISC ou apenas com componentes simples como ALUs, multiplicadores, shifters[6, 5]. As arquitecturas CGRA permitem normalmente reduzir o consumo de energia.

O desenvolvimento de geradores de endereços capazes de suportar ciclos encadeados numa única configuração permitiu reduzir o tempo de reconfiguração das CGRAs[7]. A ideia foi inspirada no uso de contadores em cascata para o gerador de endereços[8].

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, o que é menos flexível[9]. Na configuração dinâmica pode-se reconfigurar o sistema multiplas vezes durante um programa. No entanto, é necessário contabilizar o tempo de reconfiguração de modo que o desempenho seja satisfatório.

Existem dois tipos de sistemas reconfiguráveis: homogéneos e heterogéneos. No sistema homogéneo, todas as unidades de processamento são idênticas[10], e permitem realizar um leque alargado de funções. No caso dos sistemas heterogéneos, cada unidade realiza uma tarefa específica[11]. Apesar das redes homogéneas potêncialmente serem mais eficientes, as redes heterogéneas compensam com um rácio de uso de silício melhor e com uma melhor utilização energética[12].

Um compilador para este tipo de arquitecturas não pode usar apenas técnicas convencionais, mas também precisa de usar técnicas de colocação de componentes e encaminhamento de sinais (*Place and Route*), usadas também em FPGAs[13].

Capítulo 3

Arquitectura do Versat

O Versat usa uma arquitectura *Coarse Grain Reconfigurable Arrays* (CGRAs). As arquitecturas CGRA surgiram com o objectivo de reduzir a complexidade, o tempo de configuração e o tempo de compilação em relação às arquitecturas FPGA.

Todas as CGRAs possuem elementos de processamento (ALUs, multiplicadores, etc), onde são realizadas as operações. Também possuem uma rede de interconexão, que é usada para unir os vários elementos de processamento e memórias que armazenam configurações da arquitectura. As CGRAs variam em relação ao tipo de unidades de processamento utlizadas e à respectiva rede de interconexão.

Dado que os sistemas homogéneos causam um desperdício de recursos, o Versat usa unidades de processamento heterogéneas. Para a realização da interconexão, as estruturas com nós todos conectados uns com os outros têm sido evitadas, visto que têm escalabilidade pobre em termos de área, um atraso entre os nós maior e um consumo maior. No entanto, no caso do Versat é usada esta topologia, pois é privilegiada a flexibilidade de configuração face à escalabilidade. Cada sistema Versat utiliza um número reduzido de nós, prevendo-se que a escalabilidade passe pela utilização de multiplos sistemas Versat em vez de um sistema Versat de grandes dimensões. O facto de ser utilizado um número reduzido de nós compensa o facto dos nós estarem todos interligados. Um número restrito de nós é usado também para evitar uma área muito grande. Note-se que a área varia quadraticamente com o número de ligações.

Numa tecnologia CMOS, a potência varia de acordo com a seguinte relação:

$$P \propto CV^2 f A \tag{3.1}$$

onde:

- P é a potência do circuito;
- C é a capacidade da porta;
- V é o VDD;
- f é a frequência de relógio;

- A é a área do circuito.

Assim pode pensar-se que a redução de potência pode conseguir-se por redução da frequência de trabalho e consequente redução da tensão de alimentação, compensada pelo elevado grau de paralelismo possível durante a execução de programas no Versat.

3.1 Modelo de Topo

O diagrama de topo do Versat é dado pela figura 3.1.

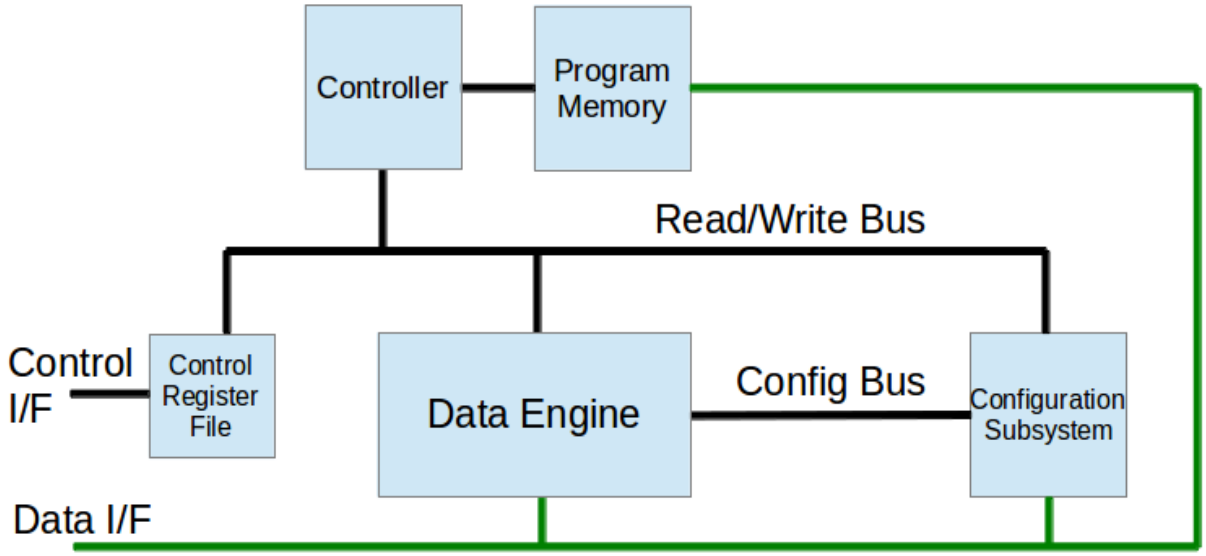


Figura 3.1: Esquema da unidade de topo.

No Versat existem vários sub-componentes:

- Data engine;
- Subsistema de configuração;
- Memória de instruções;
- Controlador;
- Ficheiro de registos de controlo;
- Descodificador de endereços;
- Sistema host-guest.

O Versat é controlado centralmente pelo seu controlador. O controlador controla o barramento de Leitura/Escrita, que é usado para efectuar leitura/escrita nos registos dos outros sub-componentes.

A interface de controlo é usada por um sistema *host*, que está a controlar o Versat. Os comandos são dados por um *host* e o Versat executa o que o *host* lhe ordena. Pela interface de controlo, são

trocados preferencialmente comandos e informações de estado para indicar, por exemplo, o fim de algum comando executado. A troca de dados é realizada pela interface de dados. É também possível trocar dados pela interface de controlo, mas apenas quando a velocidade de transferência dos dados for pouco relevante. Por exemplo, caso o Versat esteja em modo de depuração, é usada a interface de controlo.

Existem dois tipos de barramentos de controlo possíveis de seleccionar: 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 por exemplo, realizar a depuração de programas. O escravo SPI é o Versat, enquanto o mestre SPI é o anfitrião externo. O barramento paralelo é usado quando se liga o Versat a um anfitrião embebido. Este barramento está ligado ao ficheiro de registos de controlo e opera de uma forma simples e genérica. Pode ser necessário adaptar esta interface a um formato comercial, por exemplo, utilizando um barramento amba-AXI.

3.2 Controlador

O controlador é descrito pela figura 3.2.

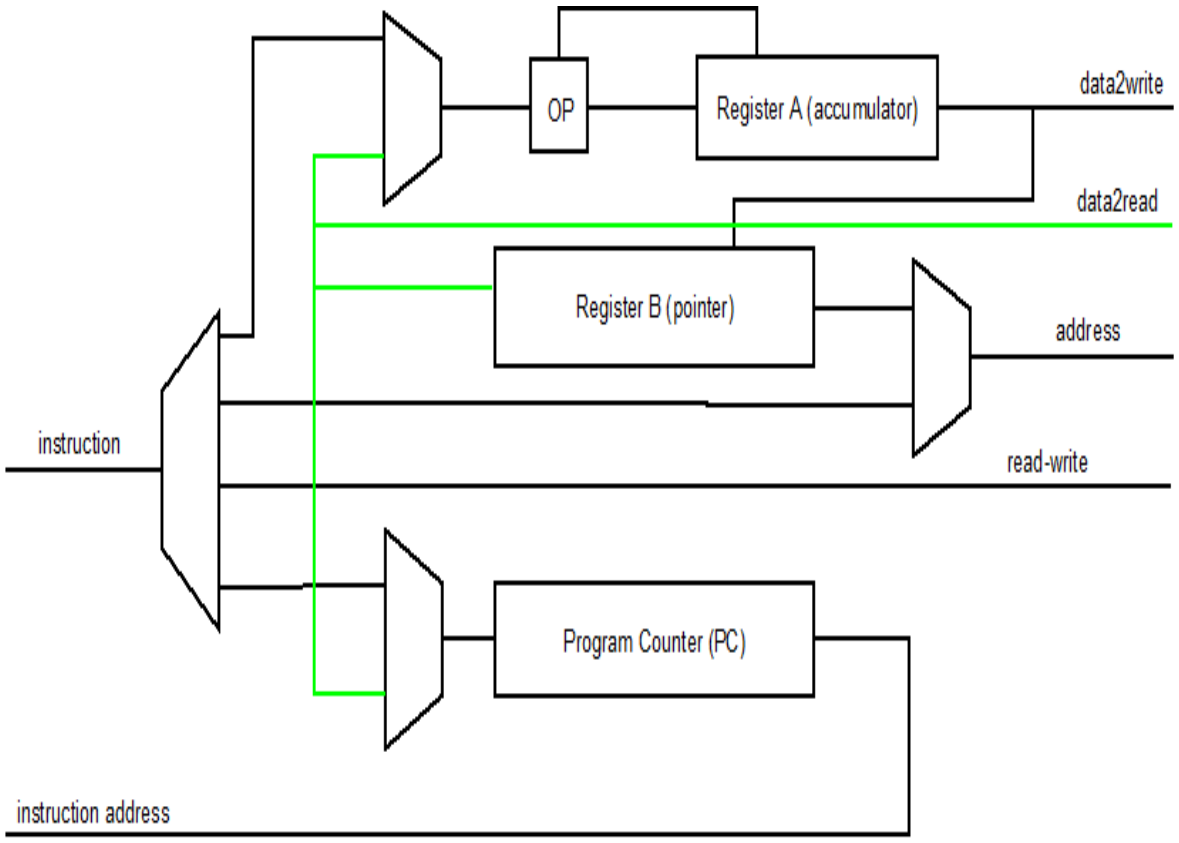


Figura 3.2: Esquema da unidade de controlo

O controlador usa o barramento R/W para realizar leituras/escritas nos seus periféricos, que são módulos do Versat. A arquitectura do controlador do Versat é constituída por 3 registos:

- Contador de programa;
- Acumulador (registo A);
- Apontador (registo B).

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

O Versat é uma arquitectura de acumulador. O registo A é o acumulador. O resultado de todas as operações realizadas pelo controlador vão parar ao acumulador, que armazena o valor. É no bloco OP que se efectuam 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 usado para endereçamento indirecto. O registo B guarda o endereço da próxima posição de memória a ler ou escrever numa instrução que utilize endereçamento indirecto.

As funções principais do controlador são:

- Comunicação com o sistema anfitrião;
- Implementação da estrutura de controlo dos procedimentos do Versat;
- Reconfiguração e controlo do Data Engine.

Caso o Versat esteja a trabalhar sozinho, em modo de depuração por exemplo, ele pode realizar o *upload*/*download* de dados e *upload* de procedimentos.

O controlador pode escrever na memória de instruções mas não pode ler da memória. Esta funcionalidade é usada para carregar procedimentos.

O controlador consegue ler e escrever no ficheiro de registos de controlo, que é partilhado com o sistema anfitrião. A função principal do ficheiro de registos de controlo é a de um canal de comunicação entre um anfitrião e o Versat. Os parâmetros necessários para correr os procedimentos no Versat são passados através deste canal.

O controlador pode escrever configurações parciais no subsistema de configuração de modo a reconfigurar o Data Engine. Também pode realizar pequenos cálculos que sejam necessários.

o conjunto de instruções executadas pelo controlador está descrito na tabela 3.1.

Tabela 3.1: Tabela das instruções assembly do Versat

Instruções	Pseudo-código
nop	Nop
rdw	A <= (imm)
wrw	(imm) <= A
wrc	(imm1+imm2) <= A
rdwb	A <= (B)
wrwb	(B) <= A
beqi	PC <= imm if regA=0
beq	PC <= (imm) if regA=0
bneqi	PC <= imm if regA!=0
bneq	PC <= (imm) if regA!=0
ldi	A <= imm
ldih	A[31:16] <= imm
add	A <= A + (imm)
addi	A <= A + imm
sub	A <= A - (imm)
and	A <= A & (imm)

3.3 Data Engine

O Data Engine é constituído por 15 unidades funcionais, organizadas como indicado na figura 3.3. A arquitectura usada tem 32 bits.

As memórias embebidas 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 um número de iterações, período de cada iteração, incremento por ciclo, endereço de inicio e deslocamento entre períodos. As memórias podem ser configuradas para leitura ou escrita. No caso de serem configuradas para escrita, é necessário também configurar as entradas das memórias de modo a seleccionarem os dados certos.

As ALUs são unidades aritméticas lógicas que exercem diversas funções. As ALU-Lite executam apenas as primeiras 6 funções das ALUs, que são:

- OR lógico;
- AND lógico;
- NAND lógico;
- XOR lógico;
- Soma;
- Subtacção;
- Extensão de sinal de 8 para 32 bits;

- Extensão de sinal de 16 para 32 bits;
- SHIFT RIGHT aritmético;
- SHIFT RIGHT lógico;
- Comparação com sinal;
- Comparação sem sinal;
- Contagem dos *bits* que estão a 0 à esquerda.
- Máximo;
- Mínimo;
- Valor absoluto.

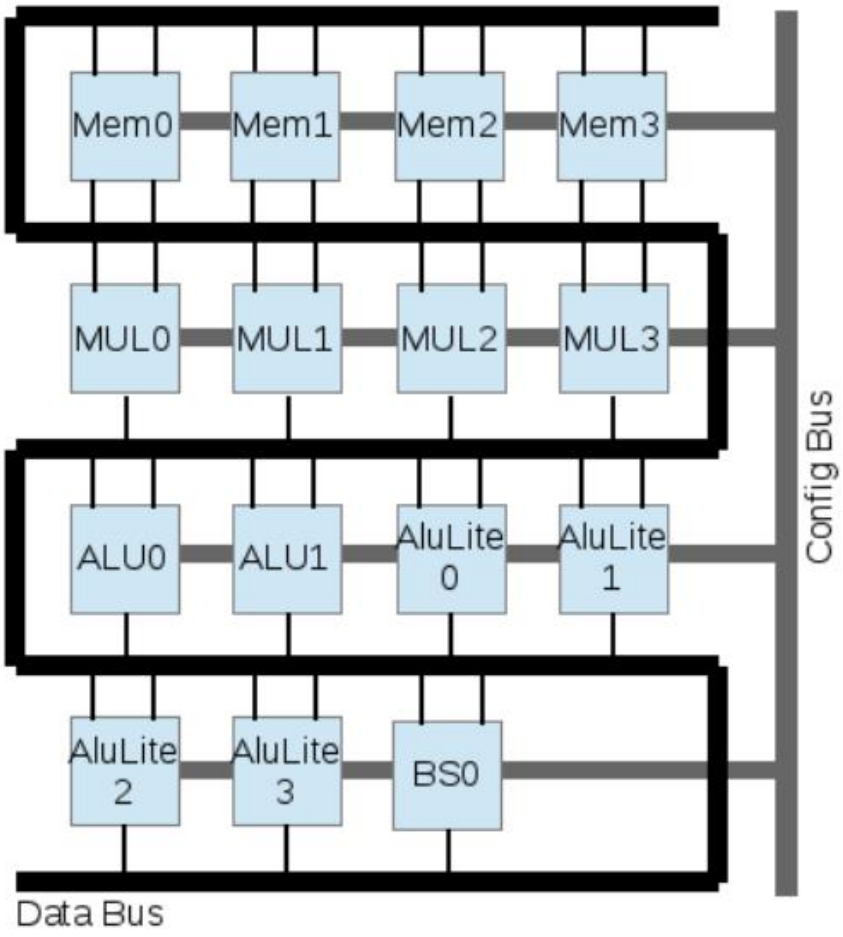


Figura 3.3: Esquema de ligações do Data Engine

Os multiplicadores efectuam uma multiplicação de dois operandos de 32 bits com um resultado de 64 bits, do qual só 32 bits são utilizados. É possível escolher se se quer a parte alta da multiplicação (bit 32 ao bit 63) ou a parte baixa (bit 0 ao bit 31). Também é possível escolher efectuar ou não a divisão por 2 do resultado final, o que ajuda quando se trabalha em notações de vírgula fixa.

Os *shifters* são unidades especializadas em deslocamento de bits. Um dos operandos é a palavra a deslocar e o outro operando é a magnitude do deslocamento. Os *shifters* 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 Data Bus. Cada unidade de processamento é capaz de seleccionar qualquer secção de 32 bits existente no Data Bus, o que permite que todas as unidades de processamento possam estar interligadas umas às outras. Isto facilita o trabalho do compilador, que não precisa 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, todas as configurações existentes são acerca de cada FU independente. Cada conjunto de unidades de processamento configuradas origina um datapath para uma tarefa específica.

Se num *datapath* estiverem várias memórias, as memórias vão trabalhar em sincronismo. Se existirem recursos suficientes, mais do que um *datapath* pode executar em paralelo, permitindo realizar paralelismo ao nível de tarefa.

Para controlar o Data Engine, é necessário atribuir valores no seu registo de controlo, tal como descrito na tabela 3.2. Pode essencialmente inicializar-se as unidades de processamento (bit 0, Init) ou manda-las correr (bit 1, Run). Os restantes bits deste registo (bit 2 a 20) 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. A inicialização de outras unidades funcionais têm como resultado o anulamento do valor de saída. Mandar correr as memórias faz iniciar a geração de endereços nas memórias indicadas. Mandar correr qualquer outra unidade de processamento tem apenas o efeito de anular as suas saídas inicialmente.

Quando se manda correr o Data Engine é necessário saber quando a sua operação termina. Para tal usa-se o registo de estado do Versat descrito na tabela 3.3. Este registo indica quais as memórias que terminaram a sua sequência de endereços.

3.4 Subsistema de configuração

O subsistema de configuração é um sistema de memória onde as configurações do Data Engine estão guardadas.

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 Data Engine enquanto que 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 Data Engine, pode-se guardar o seu valor na memória de configurações para reutilização mais tarde.

Tabela 3.2: Registo de controlo do Data Engine.

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

Tabela 3.3: 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

3.5 Memória de instruções

A memória de instruções é constituída por uma memória RAM e uma memória ROM. A memória RAM tem 2048 posições enquanto que a ROM tem 256 posições. No futuro poderá ser necessário aumentar o tamanho da memória, ou transforma-la numa cache, pois o uso de compiladores pode produzir código pouco optimizado.

O carregamento das instruções realiza-se numa fase de *setup*, antes de se usar o Versat. O carregamento das instruções é feita através da interface de dados entre o controlador do Versat e o exterior.

A ROM (também designada de boot ROM) contém um programa fixo para carregamento de programas do Versat, dados e para executar programas previamente carregados. A boot ROM também pode ser usada para descarregar dados calculados pelo Versat. A RAM é usada após o carregamento do programa, para executar esse mesmo programa.

Capítulo 4

Compilador básico

Existem duas abordagens possíveis para realizar o compilador: fazer uma compilação clássica, através da construção de árvores de sintaxe abstracta (*abstract syntax trees*), cada uma representando uma expressão, realizando a respectiva decomposição e gerando as várias instruções.

Outra abordagem que é analisada neste capítulo é fugir à implementação de um compilador clássico e utilizar uma linguagem de orientação por objectos para representar os componentes de *hardware*, onde cada classe é um componente de *hardware*. O programador utiliza essas classes com o objectivo de configurar o Data Engine.

Os métodos das classes permitem que o programador utilize as várias funções dos componentes de *hardware*. O programador utiliza as classes para descrever acções de controlo ou estruturas de *hardware* que permitam realizar a tarefa em questão.

O objectivo é construir uma linguagem de muito baixo nível para depois com o passar do tempo ser possível subir o nível de abstracção.

4.1 Abordagem da construção das árvores de sintaxe abstracta

Construindo árvores de sintaxe abstracta é possível descodificar as expressões em instruções e gerar o respectivo *assembly*. A árvore de sintaxe abstracta é usada para analisar gramaticalmente as expressões.

4.1.1 Soma simples

Considerando a soma de dois registos:

```
R5 = R1+R2;
```

A árvore de análise respectiva encontra-se na figura 4.1. No momento da construção da árvore, a árvore é percorrida da esquerda para a direita. A descodificação é feita pela seguinte ordem:

- Entrada em R1 e memorização da instrução RDW, que vai executar uma leitura;

- Entrada em +, memorizando a instrução ADD;
- Entrada em R2, que vai indicar qual o regsto que se vai somar ao conteúdo do registo A;
- Gravar o resultado em R5, memorizando a instrução WRW.

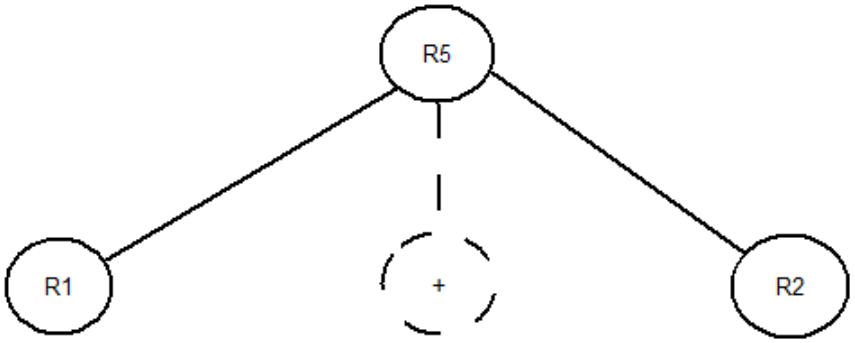


Figura 4.1: Árvore de sintaxe abstracta de uma soma.

O resultado é apenas gravado quando se termina a recursividade. O respectivo código *assembly* é:
RDW R1
ADD R2
WRW R5

Esta abordagem é boa para compilar expressões para o controlador controlador, pois o controlador funciona como uma máquina convencional. No entanto, é uma abordagem menos boa para configurar e controlar o Data Engine.

4.1.2 Ciclo *for*

Para a realização de um ciclo *for*, é necessário o uso do Data Engine. Considera-se o caso do ciclo *for* seguinte:

```
for (i=0;i<N;i++)      {
    MEM3[i] = MEM2[2i]+MEM2[2i+1];          }
```

Para o uso de ciclos *for*, não é necessário fazer a decomposição das expressões em instruções *assembly* usando compilação clássica, pois o Data Engine funciona de maneira diferente do controlador. No Data Engine usa-se instruções *assembly* apenas para sua configuração, enquanto que no controlador cada expressão é descodificada para um equivalente em *assembly* para a execução. Conclui-se assim que para o Data Engine as árvores de sintaxe abstractas não são muito úteis.

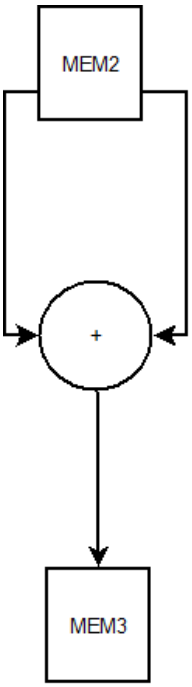


Figura 4.2: Árvore lexical de uma soma dentro de um ciclo for

Para o uso do Data Engine, é necessário usar um grafo em vez de uma árvore de sintaxe abstracta. Para um exemplo dado acima, construiu-se o grafo mostrado na figura 4.2. O compilador teria de identificar este grafo e depois controlar a execução do Data Engine.

O código *assembly* que deveria ser produzido é dado a seguir:

```
#clear conf_reg
        wrw CLEAR_CONFIG_ADDR
#configure mem2 for reading vectors x1 and x2
        ldi 0
        wrc MEM2A_CONFIG_ADDR, MEM_CONF_START_OFFSET
        ldi 1
        wrc MEM2B_CONFIG_ADDR, MEM_CONF_START_OFFSET
        ldi 1
        wrc MEM2A_CONFIG_ADDR, MEM_CONF_DUTY_OFFSET
        wrc MEM2B_CONFIG_ADDR, MEM_CONF_DUTY_OFFSET
        wrc MEM2A_CONFIG_ADDR, MEM_CONF_PER_OFFSET
        wrc MEM2B_CONFIG_ADDR, MEM_CONF_PER_OFFSET
        ldi 2
        wrc MEM2A_CONFIG_ADDR, MEM_CONF_INCR_OFFSET
        wrc MEM2B_CONFIG_ADDR, MEM_CONF_INCR_OFFSET
        ldi N
        wrc MEM2A_CONFIG_ADDR, MEM_CONF_ITER_OFFSET
        wrc MEM2B_CONFIG_ADDR, MEM_CONF_ITER_OFFSET
```

```
#configure mem3 for writing result
        ldi salu0
        wrc MEM3A_CONFIG_ADDR,MEM_CONF_SELA_OFFSET
        ldi 0
        wrc MEM3A_CONFIG_ADDR,MEM_CONF_START_OFFSET
        ldi 1
        wrc MEM3A_CONFIG_ADDR,MEM_CONF_DUTY_OFFSET
        wrc MEM3A_CONFIG_ADDR,MEM_CONF_PER_OFFSET
        wrc MEM3A_CONFIG_ADDR,MEM_CONF_INCR_OFFSET
        ldi 6
        wrc MEM3A_CONFIG_ADDR,MEM_CONF_DELAY_OFFSET
        ldi N
        wrc MEM3A_CONFIG_ADDR,MEM_CONF_ITER_OFFSET
#configure alu0 for adding x1 and x2
        ldi ALU_ADD
        wrc ALU0_CONFIG_ADDR,ALU_CONF_FNS_OFFSET
        ldi smul1
        wrc ALU0_CONFIG_ADDR,ALU_CONF_SELA_OFFSET
        ldi salu0
        wrc ALU0_CONFIG_ADDR,ALU_CONF_SELB_OFFSET
        ldi 1
#init engine
        ldi 0xfffd
        ldih 1
        wrw ENG_CTRL_REG
        wrw ENG_CTRL_REG
#run engine
        ldi 0xc002
        ldih 1
        wrw ENG_CTRL_REG
#wait for completion to display results
waitres ldi 256
        and ENG_STATUS_REG
        beqi waitres
        nop
        nop
#branch to boot ROM
        ldi 0
        beqi 0
```

nop

nop

O número de iterações é definido por N, que é necessário indicar nos parâmetros MEM_CONF_ITER_OFFSET de cada memória usada. Visto que o i inicial é zero por pré-definição no compilador desenvolvido, os parâmetros MEM_CONF_INCR_OFFSET e MEM_CONF_START_OFFSET são usados para calcular os endereços na memória X da seguinte forma:

MEMx[INCR_OFFSET*i+START_OFFSET]

O gerador de endereços usado entre os dois disponíveis em cada memória é indicado pelo programador. Numa primeira abordagem, o período de cada iteração será também definido pelo programador. No futuro poderão usar-se mecanismos para automatizar o cálculo do período.

Os valores usados para inicializar e correr o Data Engine serão calculados internamente pelo compilador, recebendo instruções mais simples do utilizador.

No ciclo *waitres* é usada uma máscara consoante a memória que se quer monitorizar. Quando uma memória acaba de ser lida ou escrita, é indicado no ENG_STATUS_REG que a memória terminou a sua execução.

4.2 Utilização de uma linguagem orientada a objectos

A outra abordagem é através do uso de várias classes que representem os componentes do *Data Engine*. Utilizando esta abordagem, o programador escreve um programa com as classes disponíveis, descrevendo várias configurações do Data Engine. Para as instruções do controlador, é usada uma linguagem interpretada. O UML das classes usadas no Data Engine é dado na figura 4.3.

A compilação do código será feita pelo script automaticamente. Cada classe construirá o seu respectivo *Assembly*.

A nível do *Data Engine*, as ligações serão feitas directamente nas classes, que usando métodos do *Data Engine*, geram o próprio *Assembly*.

Considerando o exemplo da soma vectorial da secção anterior, o respectivo pseudo-código na linguagem Versat é:

```
mem2A.config(0, 1, 2, N, 1, 0); // inicializar a memoria 2A
mem2B.config(1, 1, 2, N, 1, 0); // inicializar a memoria 2B
mem3A.config(0, 1, 6, N, 1, 0); // inicializar a memoria 3A

alu0.config("ADD"); // inicializar a alu

alu0.connectA(mem2A); // conectar a entrada A da alu a mem2A
alu0.connectB(mem2B); // conectar a entrada B da alu a mem2B
```

```
mem3A.connect(alu0); // conectar a memoria 3A a saida da ALU

mem2A.init(); // inicializar a memoria 2A
mem2B.init(); // inicializar a memoria 2B
mem3A.init(); // inicializar a memoria 3A
alu0.init(); // inicializar a alu0

mem3A.wait(); // marcar esta memoria para que se espere por ela


engine.init();
engine.run();
engine.wait();
```

Nos métodos de configuração das memórias, é passado o início, o período, o *duty*, o incremento, o *delay* e o número de iterações. De início decidiu-se passar também o *delay* mas poderão vir a ser experimentadas formas de o compilador calcular o *delay*. A ALU precisa apenas de saber qual a operação que vai realizar.

As conexões são feitas utilizando métodos do objecto destino. Existe um método para cada conexão, um para a entrada A e outro para a entrada B.

Como se pode ver, a descrição acima, usando um paradigma de programação por objectos, consegue realizar a mesma função que o código *assembly* dado para este exemplo. A diferença é que se consegue uma descrição muito mais compacta e fácil de ler quando se usa uma representação por objectos.

Poderia pensar-se em usar qualquer linguagem de programação orientada para objectos para realizar descrições de configurações do Data Engine e do seu controlo. Até se poderia usar uma linguagem de *scripting* como Python, a qual tem suporte para classes e métodos. Estes programas quando executados gerariam o código assembly ou o código máquina directamente. Esta abordagem evitaria de todo o desenvolvimento de um *parser* para o compilador do Versat.

Contudo, uma parte importante do problema é compilar o código que é corrido pelo controlador do Versat. Uma linguagem para esse fim seria próxima de uma linguagem de programação convencional e teria de incluir expressões condicionais (*if*) e ciclos de programa (*for, while, do while*, etc). Estas expressões são mais difíceis de traduzir apenas chamando métodos de classes, e mesmo que se arranjasse uma solução para tal, o código produzido não seria compacto e elegante.

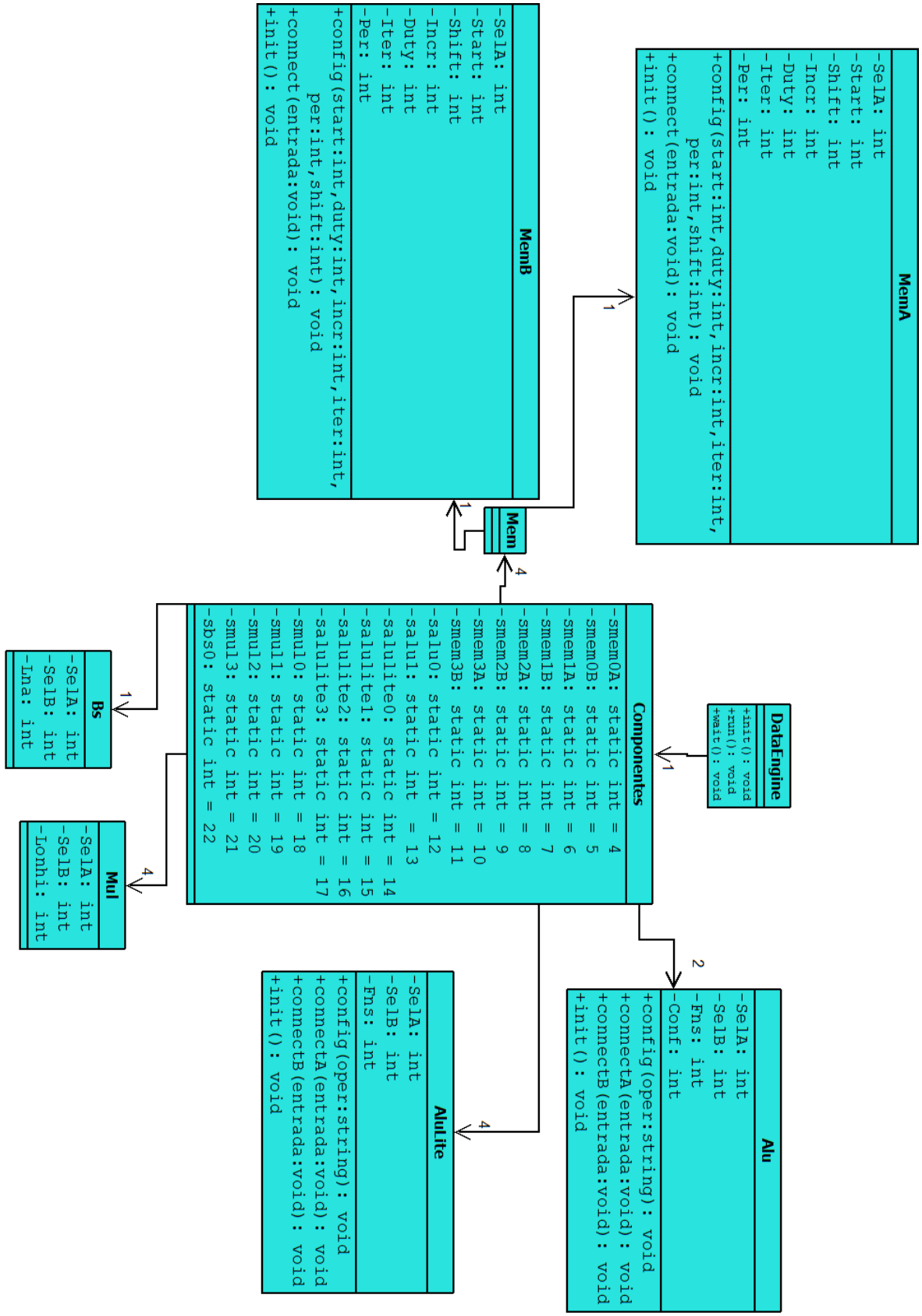


Figura 4.3: UML do Data Engine.

Capítulo 5

Estrutura do compilador

Um compilador converte um código numa linguagem definida (como Java, C, C++, etc) para uma linguagem máquina. Quando se constrói um compilador, é necessário construir a conversão da linguagem para um *assembly* compativel com a máquina onde se está a trabalhar, o que dava um compilador diferente para cada linguagem e para cada máquina. Tal não acontece graças a uma representação intermediária universal, que divide o processo de compilação entre dois passos: o *front end* e o *back end*. Tal como no GCC, o compilador do Versat está dividido em duas partes: o *front end* e o *back end*. Em muitos compiladores existe um passo intermédio entre a representação intermédia e o *back end* que é o passo de optimização. Devido à simplicidade dos programas em Versat, não foi implementado o passo de optimização.

No *front end* é realizada a conversão para a representação intermédia. O *front end* está dividido em 3 fases:

- *Scanning*;
- *Parsing*;
- Análise semantica. Tradução da árvore de sintaxe abstracta para a estrutura de dados intermédia.

O *scanning* lê os caracteres e converte-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*, analisando uma sequência de *tokens* e arranjar alguma regra gramatical compatível com a sequência recebida. Caso não seja detectada alguma regra gramatical existente, 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*. O *back end* é a parte responsável por ler a estrutura intermédia e gerar o respectivo *assembly*. O *back end* está dividido em 3 fases:

- Leitura da estrutura intermédia e geração do respectivo *assembly*;
- Optimização do *assembly*;

- Geração do código máquina através do *assembly*.

discutir com o professor sobre a junção do assembler com o compilador e melhoramento do assembler!!!!!!!!!! O compilador não produz o código máquina. Para a produção do código máquina, é necessário usar o *assembler* do Versat que está à parte do compilador. O compilador do Versat não faz optimizações de *assembly*.

5.1 Front end

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

O Flex é o responsável pela análise lexical do compilador. O Flex lê os caracteres e converte-os em *tokens*, também é responsável por:

- Reconhecer palavras chave na linguagem;
- Ignorar espaços em branco e comentários;
- Reconhecer números e *bits*.

Depois de detectados os *tokens*, é feita a análise sintáctica. A análise sintáctica é realizada pelo programa Bison. O Bison é um gerador de *parse*. Recebe uma série de *tokens* e procura uma regra gramatical que seja compativel com a sequência que recebeu. Também avisa caso exista alguma ambiguidade gramatical e caso não seja detectada nenhuma regra gramatical compativel com a sequência recebida, activa a directoria yyerror e informa que houve um erro.

Considera-se um ciclo while em C++ do Versat.

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

O bison recebe os vários *tokens* e procura uma regra gramatical existente.

A árvore do *parse* gerada pelo bison está representada na figura 5.1.

Onde:

- whl_units é o ramo que espera receber o *token* while;
- reg_units é o ramo que recebe o registo usado no primeiro operando;
- sif_operators tem as condições sentenciais;
- reg_const representa as constantes.

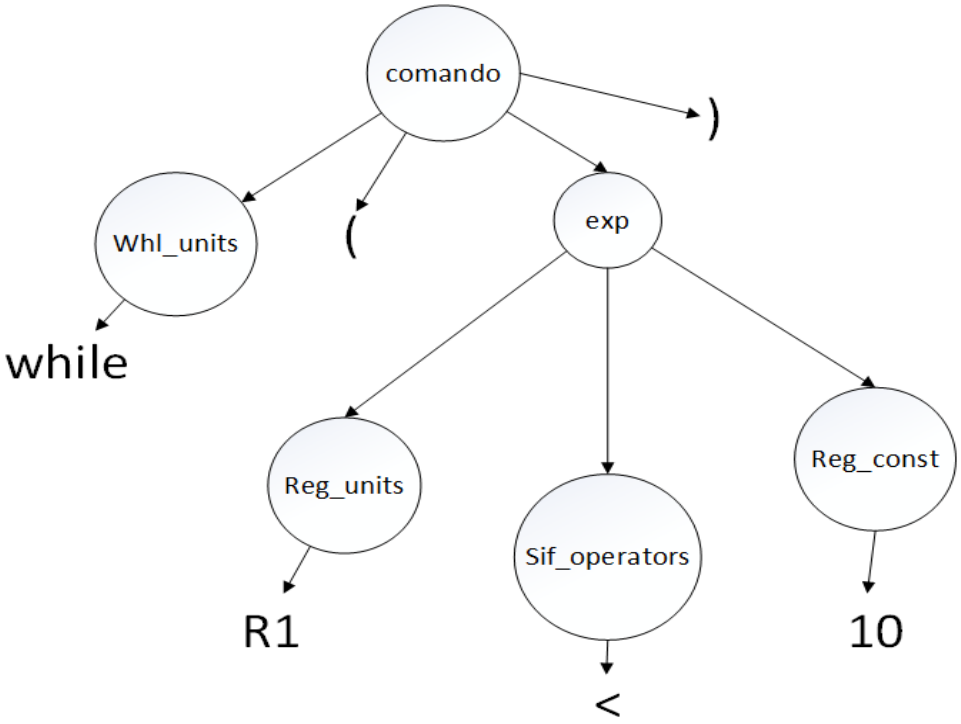


Figura 5.1: Árvore gerada do ciclo *while*.

É necessário existir ramos com o objectivo de guardar os dados recebidos na estrutura de dados intermédia.

Considera-se uma instrução do Data Engine.

```
mem0.portB.setStart(2);
```

A árvore do *parse* gerada pelo bison está representada na figura 5.2.

Onde:

- Mem units é a memória em causa;
- Ports é o porto de memória em causa;
- Mem_methods é o método em causa;
- Mem units é a memória em causa;
- Expressions2 e expression são os ramos que geram a expressão da árvore.

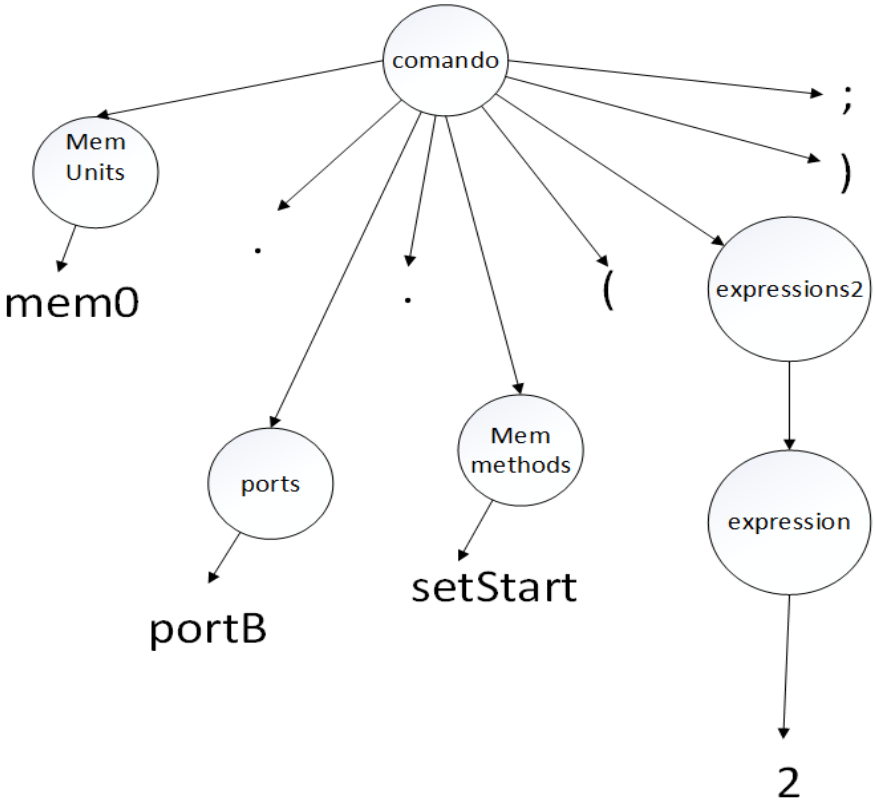


Figura 5.2: Árvore da instrução de configuração do gerador de endereços.

O estado de erro não é representado nos esquemas das figuras 5.3, 5.2 e 5.1, mas quando não se encontra nenhuma sequência compativel com a sequência que se recebeu vai para o estado de erro.

(perguntar isto ao professor)

Considera-se uma expressão em C++ do Versat.

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

A árvore do *parse* da expressão está representada na figura 5.3.

Onde:

- Reg_target é o registo onde o resultado vai ser guardado;

- Expressions é o ramo responsável por guardar os dados da expressão na estrutura de dados intermédia.

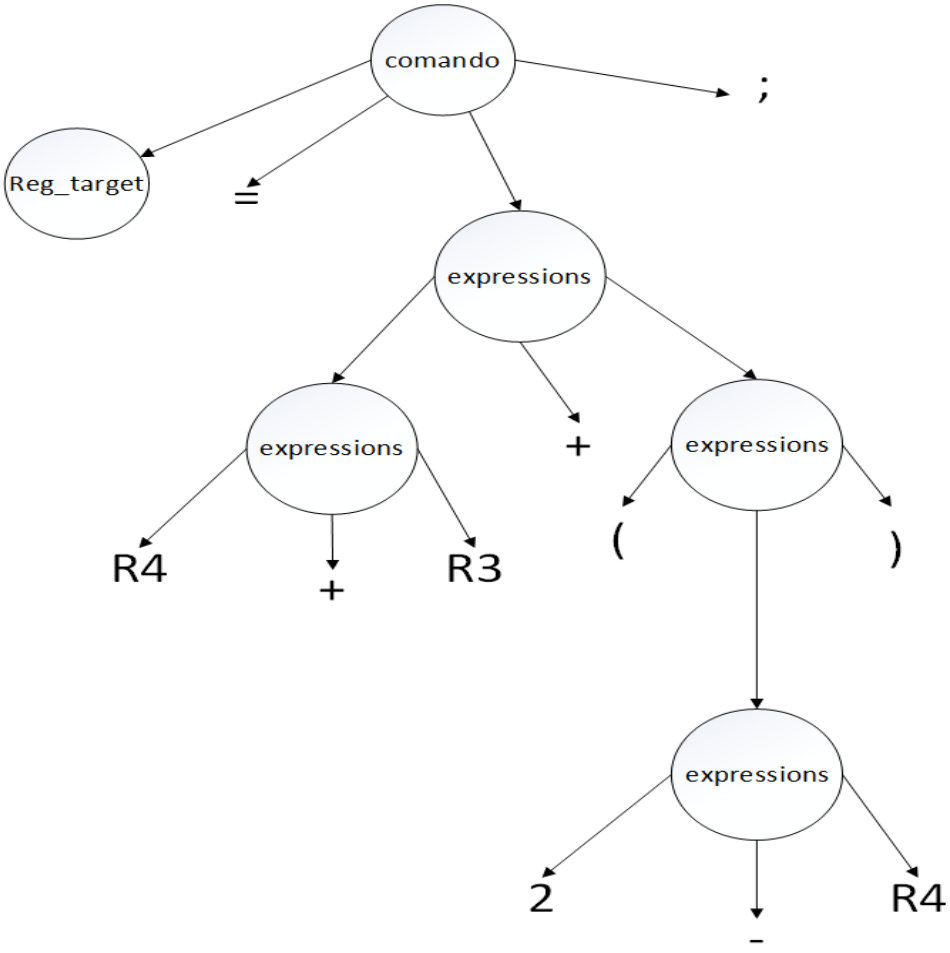


Figura 5.3: Árvore gerada do ciclo *while*.

Para processar as expressões, é usada uma estrutura de dados em forma de árvore binária, associada ao comando.

5.2 Back end

Depois de concluido o *parse*, a estrutura de dados é preenchida. A estrutura de dados é uma lista de tipo comando, que tem todas as estruturas necessárias para gerar o *assembly*. Esta estrutura de dados é constituída pelo seginte pseudo-código:

Tirar isto e meter pseudo-código!!!!!!!!!!!!!!!!!!!!!!!

```
tipo de comando;
unidade;
porto;
m todo;
NODE * rvore de registos;
NODE2 * rvore de express es;
```

```
operador;
operando1;
operando2;
tipo do operador1;
tipo do operador2;

etiqueta;

GenFU _genFU;
GenMem _genMem;
ctrlReg _ctrlReg;
controller _controller;
```

Existem outros atributos auxiliares usados no *back end*, porém não estão representados no pseudo-código.

O apontador de tipo NODE é a base da árvore que gera as expressões. É nesta estrutura que se guarda as expressões de registos e se gera o *assembly* pela ordem correcta. O tipo NODE2 é responsável pela geração das expressões de memória.

As classes _genFU, _genMem, _ctrlReg e _controller são responsáveis pela geração de *assembly* das unidades funcionais, pelas instruções de inicialização, arranque, espera e pelas instruções ligadas ao controlador como por exemplo, as expressões.

Fazer um diagrama UML da estrutura de dados

Considera-se um ciclo for em C++ do Versat.

```
int main() {

        for(R1=0;R1<10;R1=R1+1) {
                R5 = R5+R1; }

return 0;       }
```

O respectivo código *assembly* gerado é:

```
        ldi 0
        wrw R1
forDirectStatment0      rdw R1
        addi -10
        wrw RB
        ldi 0
        ldih 0x8000
        and RB
```

```
        beqi forStatment0
        nop
        nop
        rdw R5
        add R1
        wrw R5
        rdw R1
        addi 1
        wrw R1
        ldi 0
        beqi forDirectStatment0
        nop
forStatment0 nop
        ldi 0
        beqi 0
        nop
```

Estão representadas algumas das etiquetas que o utilizador não pode usar no programa, por estarem relacionadas com a geração dos ciclos (consultar tabela 6.13).

Considera-se uma expressão em Versat.

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

A respectiva árvore de sintaxe abstracta é dada na figura 5.4.

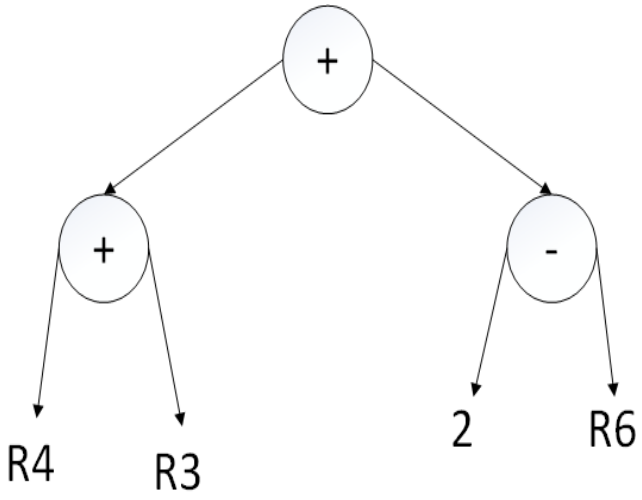


Figura 5.4: Árvore de sintaxe abstracta.

A árvore é contruída no *front end*, enquanto no *back end* é realizada a leitura da estrutura de dados com o objectivo de gerar o respectivo *assembly*.

Capítulo 6

Variáveis e objectos pré-definidos

É conveniente referir que optou-se por uma abordagem baseada em C++ para a linguagem do compilador. Esta decisão está directamente atribuída ao facto de esta linguagem ser conhecida, o que facilita o trabalho de um programador, não necessitando de um processo de aprendizagem. Entretanto, é convenente referir que o Versat não tem uma *stack* própria, sendo impossível criar funções em C++. Assim, neste processo de programação não estão envolvidas funções.

No Versat trabalha-se com objectos e variáveis. Como os recursos são limitados, tanto as variáveis como os objectos já estão pré-definidos.

6.1 Objectos e variáveis existentes

Os objectos e variáveis existentes no Versat encontram-se na tabela 6.1.

Visto que cada memória tem dois portos com configurações independentes, os portos das memórias são tratados como objectos. É possível usar o registo RB como variável, mas este é usado internamente pelo compilador para cálculos auxiliares. O objecto de_ctrl_reg representa os registos de controlo e de estado, embora estes apresentem-se em separado. Esta abordagem tem o intuito de facilitar o trabalho do programador.

É preciso proibir o uso do RB!!!!!!!!!!!!!!!!

Tabela 6.1: Objectos e variáveis existentes.

Objecto/Variável	Número/letra da unidade funcional	Descrição
mem	0-3	Memórias existentes no C++ do Versat.
alu	0-1	Alus existentes no C++ do Versat.
aluLite	0-3	AluLites existentes no C++ do Versat.
mult	0-3	Multiplicadores existentes no C++ do Versat.
de_ctrl_reg	-	Representante dos registos de controlo e de estado.
port	A-B	Portos de memória existentes no Versat. Cada memória tem dois portos com configurações independentes.
Bs	0	*Barrel shifters* existentes no C++ do Versat.
R	1-14	Registos permitidos para uso como variáveis no C++ do Versat. Também existem os registos R0 e R15, mas estes são usados internamente pelo *hardware* do Versat.
RB	-	Registo B. Apesar deste registo ser utilizado como apontador, o uso dele como variável é permitido.
mem	0-3/A-B	Variáveis das expressões de memória. Nas expressões de memória, as memórias são tratadas como variáveis.

6.2 int main

Tal como em C++ regular, os programas em C++ do Versat começam na função *main*. Não são suportadas as variáveis de argc e argv.

Um exemplo do inicio de uma função em C++ do Versat é dado da seguinte forma:

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

Refere-se que as directivas de pré-processador, tais como #define e #include, não são suportadas.

Tirar o suporte do bison ao argc e ao argv

6.3 Métodos dos registos de controlo e de estado

Os métodos existentes no objecto de_ctrl_reg encontram-se na tabela 6.2.

Tabela 6.2: Métodos respectivos ao inicio/arranque do Data Engine.

Método	**Descrição**
init(FU)	Inicializa o Data Engine. Recebe como argumento todas as unidades funcionais que são para inicializar.
run(FU)	Arranca o Data Engine. Recebe como argumento todas as unidades funcionais que são para arrancar.
wait(FU)	Espera que o Data Engine acabe de correr. Recebe como argumento todas as unidades funcionais que se pretende esperar que concluam o seu trabalho. Recebe apenas memórias.

Depois de configurar as unidades funcionais do Data Engine desejadas, é necessário inicializar e correr. Não é obrigatório inicializar e correr todas as unidades programadas anteriormente. Como argumento são enviadas todas as unidades funcionais que se pretende inicializar e correr. Pode-se enviar, como argumento, as unidades funcionais que se desejar.

Para se verificar que as memórias, das quais se está à espera, acabaram o seu trabalho, é necessário construir uma máscara. O objectivo desta é conseguir-se comparar com o registo de estado. Entretanto, se o utilizador indicar as memórias de que está à espera, o compilador constrói a máscara automaticamente. O algoritmo utilizado pelo compilador na sua construção é dado pela equação **X**,

$$mask = mask + 2^{CTRL\_BIT\_SIZE - FU} \tag{6.1}$$

ver como se mete referencias nas equacoes!!!!!!!!!!!!!!!!

onde mask é a máscara utlizada para inicializar/correr o Data Egine, CTRL_BIT_SIZE é o número de *bits* que o registo de configuração tem e FU é o número da unidade funcionar a inicializar/correr.

Quando se deseja inicializar os dois portos de uma memória, escreve-se apenas esta, sem indicar qualquer porto. Assim, o algoritmo inicializa simultâneamente ambos os portos. Um exemplo de como se utiliza as instruções de inicialização, arranque e espera é dado por:

```
de_ctrl_reg.wait(mem0A);
de_ctrl_reg.init(mem0, mem1, mem2B);
de_ctrl_reg.run(mem0, mem1B, mem2A, alu0, mult0);
```

Os argumentos enviados são tratados como variáveis. No método *init*, além das memórias também é possível a inicialização das unidades funcionais, embora não exista utilidade prática em fazê-lo.

6.4 Limpeza dos registos do Data Engine

Tendo em conta a inexistência de um objecto associado à limpeza dos registos do Data Engine, recorre-se à utilização de funções. As funções existentes para a limpeza dos registos podem ser encontradas na tabela 6.3.

Tabela 6.3: Funções respectivas à limpeza dos registos.

Função	Descrição
clearConfigReg()	Faz a limpeza dos registos de configuração do Data Engine.

Quando é necessária a eliminação de uma configuração antiga, utiliza-se esta função, não sendo necessário passar argumentos. A utilização da função de limpeza de registos dá-se através do seguinte comando:

```
clearConfigReg();
```

6.5 Métodos exclusivos das memórias

Os métodos existentes para a configuração dos geradores de endereços dos portos das memórias encontram-se na tabela 6.4.

Tabela 6.4: Métodos respectivos à configuração dos portos das memórias.

Método	Descrição
setStart(expression)	Coloca o valor da posição de memória de onde começa a iteração.
setDuty(expression)	Coloca o valor do *duty* na memória.
setIncr(expression)	Coloca o valor do incremento na memória.
setIter(expression)	Indica à memória o número de iterações que esta precisa de fazer.
setPer(expression)	Coloca o valor do período na memória.
setShift(expression)	Coloca o valor do *shift* na memória.
setDelay(expression)	Coloca o valor do *delay* na memória.
setReverse(expression)	Coloca o valor do *reverse* na memória.

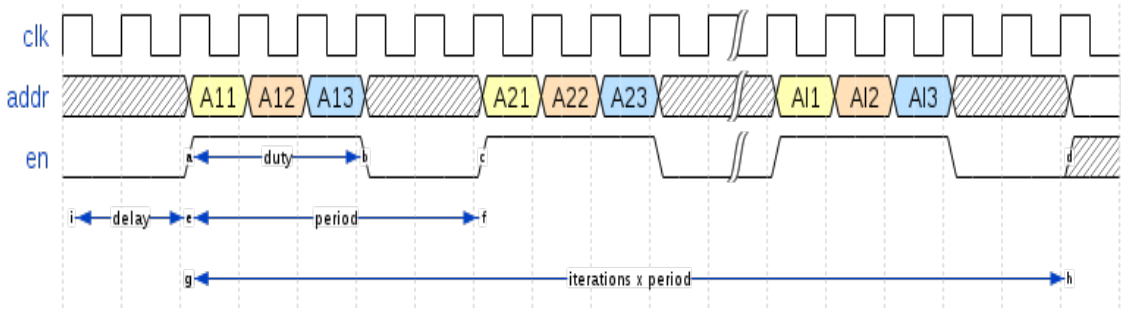


Figura 6.1: Representação dos parâmetros de configuração dos geradores de endereços.

Na figura 6.1 estão representados alguns dos parâmetros dos geradores de endereços, assim como a relação que estes têm entre si. O entendimento correcto destes parâmetros é importante, na medida que possibilita a automatização de algumas operações complexas, exigindo um maior trabalho por parte do programador. Refere-se que no compilador mantém-se a possibilidade de programar manualmente os geradores de endereços. Entretanto, oferece-se também a possibilidade do programador utilizar expressões de memória. Estas traduzidas para as respectivas configurações e, posteriormente, para *assembly*.

A utilização de instruções de automatização implica uma perda de eficiência a nível do *assembly*. Todavia, havendo dispobilidade do programador perder alguns ciclos de relógio, que facilitem-lhe a tarefa de programar, é indicado fazê-lo. Nesta secção fala-se apenas das instruções de configuração manuais, devendo-se referir na secção 6.18 serão analisadas as instruções automáticas. O significado dos parâmetros está explicitado na tabela 6.5.

As instruções de configuração dos geradores de endereços são dadas por:

```
mem0.portA.setDuty(RB);
mem0.portA.setPer(RB);
mem0.portA.setIncr(1);
```

Tabela 6.5: Métodos respectivos à configuração dos portos das memórias.

Parâmetro	Descrição
Start(expression)	Indica a partir de que endereço de memória se começa a ler. Quando não se configura nada, o valor por omissão a nível de *hardware* é zero.
Duty(expression)	Representa o número de sinais de relógio que o *enable* está a '1'. Quando o *enable* está a '1' os endereços estão a ser lidos e estão a aparecer à saída.
Incr(expression)	É o salto 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.
Iter(expression)	São o número de iterações que o gerador de endereços faz até estar concluído. No registo de estado só indica que um porto de memória está pronto depois das iterações acabarem.
Per(expression)	É a diferença de sinais de relógio entre cada *enable* do gerador de endereços. O período é usado para fazer *loops* interiores ao equivalente de um ciclo *for* encadeado em C.
Shift(expression)	Coloca o valor do *shift* na memória.
Delay(expression)	É o número de ciclos de relógio que os geradores de endereços esperam antes de começarem a trabalhar.
Reverse(expression)	É o espelhamento dos endereços de memória. Em certas aplicações como por exemplo, a FFT, é necessário fazer espelhamento das posições de memória para aplicar o algoritmo.

```
mem0.portB.setIter(R6-1);
```

Como os portos são tratados como objectos, é necessário indicar a unidade funcional. No caso de ser uma memória, também é necessária a indicação do porto a ela associado. Esta situação acontece apenas nas memórias, uma vez que os portos não têm apenas a utilidade de fazer ligações (ver secção 6.5.

6.6 Métodos exclusivos das Alu e das AluLite

Os métodos existentes para a configuração das Alu e das AluLite podem ser visualizados na tabela 6.6.

Tabela 6.6: Métodos referentes à Alu e à AluLite.

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

Apesar do método ser o mesmo, existem determinadas operações que a AluLite não consegue realizar, mas que a Alu realiza. As operações que são permitidas fazer no Versat estão representadas na tabela 6.7.

Como as AluLites são mais limitadas, ao programa-las para uma tarefa que somente uma Alu pode realizar, surge um impedimento por parte do compilador. As operações suportadas pela AluLite são as

Tabela 6.7: Operações permitidas pela Alu e pela AluLite.

Método	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	*Shift right* aritmético.
SRL	*Shift right* 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 sinal de entrada mais alto.
MIN	Coloca à saída o sinal de entrada mais baixo.
ABS	Define uma operação para a Alu seleccionada.

primeiras seis da tabela 6.7.

A instrução

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

é um exemplo de como indicar à alu0 para realizar a operação de soma. O objecto é a alu0, o método é o setOper e a operação é a soma.

6.7 Métodos exclusivos do multiplicador

Os métodos existentes para configuração do multiplicador estão indicados na tabela 6.8.

Tabela 6.8: Métodos do multiplicador.

Método	Descrição
setLonhi(bit)	Indica ao multiplicador se é para colocar à saída a parte baixa ou a parte alta do resultado. Recebe como argumento um bit.
setDiv2(bit)	Indica ao multiplicador se o resultado final é para dividir por 2. Recebe como argumento um bit.

No método setLonhi, quando se envia o bit a '1', o multiplicador coloca à saída a parte mais baixa do resultado. A parte mais alta do resultado é colocada à saída quando o bit enviado é '0'. Este parâmetro é necessário, uma vez que os multiplicadores produzem resultados de 64 *bits*, mas apenas 32 *bits* podem ser postos à saída. Quando nada é indicado, o valor é '0'.

Este processo ocorre de forma análoga no método setDiv2. Assim, quando se envia '1', o resultado é dividido por 2 e, quando se envia '0', nada acontece. Tal como no método setLonhi, o valor por omissão é '0'.

Um exemplo do uso das instruções é:

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

6.8 Métodos exclusivos do Barrel Shifter

Os métodos exclusivos do *barrel shifter* estão na tabela 6.9.

Tabela 6.9: Métodos exclusivos do *barrel shifter*.

Método	Descrição
setLNA(oper)	Meter aqui a descrição
setLNR(oper)	Meter aqui a descrição

6.9 Métodos de conexão

Nesta secção podem ser encontrados os métodos de conexão entre as unidades funcionais. Estes estão explicitados na tabela 6.10.

Tabela 6.10: Métodos respectivos à memória.

Método	Descrição
connectPortA(FU)	Conecta o porto A de uma unidade funcional à saída da unidade funcional passada como argumento.
connectPortA(mem, port)	Conecta o porto A da unidade funcional à saída de uma memória passada como argumento. Quando se conecta a uma memória é encessário indicar o porto.
connectPortB(FU)	Conecta o porto B da unidade funcional à saída da unidade funcional passada como argumento.
connectPortB(mem, port)	Conecta o porto B da unidade funcional à saída da unidade funcional passada como argumento. Quando se conecta a uma memória é encessário indicar o porto.
connect(FU)	Conecta um porto de memória a uma unidade funcional diferente de outra memória. Recebe como argumento a respectiva unidade funcional.
connect(mem, port)	Conecta um porto de memória a uma memória. Recebe como argumento a memória e o porto ao qual se quer conectar.

Os métodos connect são aplicados apenas em memórias, enquanto os demais são utilizados para as restantes unidades funcionais. Os métodos de conexão das memórias são diferentes dos métodos das restantes unidades funcionais, uma vez que os portos das memórias são tratados como objectos, como já referido anteriormente.

Um exemplo do uso das instruções é:

```
mem0.portA.connect(mem2, 'a');
mem0.portB.connect(alu0);
alu0.connectPortA(mult1);
mult3.connectPortB(mem2, 'a');
```

6.10 Métodos para desactivar unidades funcionais

Os métodos para desactivar unidades funcionais encontram-se na tabela 6.11.

Tabela 6.11: Métodos para desactivar unidades funcionais.

Método	Descrição
disableFU()	Desactiva a unidade funcional à qual o objecto está associado.

Esta instrução é utilizada quando se quer trabalhar a unidade funcional como um registo. É impossível desactivar memórias. Um exemplo do uso é:

```
mult0.disableFU();
```

6.11 Expressões de registos

A linguagem C++ do Versat suporta expressões com muitos registos, que podem ter uma dimensão muito grande, desde que existam recursos. Como não existe uma pilha de memória, é impossível declarar variáveis. O registo RB é utilizado como auxiliar no cálculo de expressões grandes, sendo necessário ter cuidado com o seu uso para guardar dados, uma vez que podem ser apagados em certa altura. Não é necessário declarar os registos.

Devido ao *assembly* limitado do controlador, nem todas as operações são permitidas. As operações permitidas estão disponíveis na tabela 6.12.

Tabela 6.12: Operações suportadas pelas expressões de registos.

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

Também é suportado o uso de parêntesis e são respeitadas as ordens aritméticas - estas consistem primeiro no cálculo das operações matemáticas entre parêntesis, seguido dos deslocamentos e ands lógicos, finalizando com as operações de soma e subtracção).

Tirar do flex o suporte para a divisão.

Um exemplo do uso de expressões é dado por:

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

6.12 If condicional

O compilador de C++ do Versat suporta instruções condicionais, assim como condicionais encadeadas. Um exemplo do uso da condição if é:

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

A condição suporta apenas expressões com dois operandos. Exceptuando esta limitação, a condição é utilizada de forma análoga a C++ regular.

6.13 Else condicional

O uso da condição *else* é feito de forma idêntica a C++ regular. Um exemplo da aplicação da condição *else* é:

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

else {
    R5=2; }
```

6.14 Ciclo while

Tal como nas condições, os ciclos são usados da mesma maneira que em C++ regular. A limitação de suportar apenas expressões com dois operandos mantém-se.

Um exemplo de aplicação do ciclo *while* é:

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

6.15 Ciclo for

Os ciclos *for* são iguais ao C++ regular. No local do incremento, as instruções Rx++, Rx- -, ++Rx e - -Rx não são suportadas.

Um exemplo de aplicação do ciclo *for* é:

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

                R7=R7+R4;  }

return 0; }
```

6.16 Ciclo do while

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

```
int main() {
      do {
           do  {
                R1=R1+2+R5; }
           while(R1!=20);
           R5=R5+1;  }
      while(R5<50);

return 0; }
```

6.17 Salto goto

Existem etiquetas que não se podem utilizar na instrução *goto*. Assim, o compilador aplica etiquetas internas para implementar as instruções de *if*, *while*, *for*, *do while* e *wait*. Estas etiquetas podem ser encontradas na tabela 6.13.

ainda não está a dar erro quando as etiquetas não são permitidas

Um exemplo do uso da instrução *goto* é:

```
begin  R4=3;
```

```
goto(begin);
```

Esta instrução em C++ regular raramente é usada, mas no C++ do Versat é extremamente útil na programação de pseudo-rotinas.

Tabela 6.13: Etiquetas que o utilizador não pode usar.

Etiqueta	Instrução relacionada
ifStatment	if
whileStatment	while
whileDirectStatment	while
forDirectStatment	for
forStatment	for
elseStatment	else
waitres	wait

6.18 Expressões de memória

O compilador do Versat também permite expressões de memória, sendo o seu algoritmo dado por:

```
for(i=0;i<iter;i++) {
  for(j=0;j<per;j++)
  memxX[start+j(per+shift)+incr*i] = memxX[start+j(per+shift)+incr*i];  }
```

O período e o número de iterações é comum a todas as memórias utilizadas nas expressões. Os outros parâmetros variam consoante a memória em causa. Os multiplicadores e alus necessários para efectuar as operações entre memórias são definidas automaticamente. Assim, não é preciso indicar as alus e os multiplicadores a aplica, uma vez que isto é feito internamente.

Fazer a libertação automática do hardware!!!!!!!!

As variáveis i e j são fixas, não sendo necessária uma declaração prévia.

Um exemplo da aplicação das expressões de memória é:

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

Quando o *for* interior não existe, o valor do período por omissão é '1'.

6.19 Return

A instrução de *return* é o final da função *main* em C++ do Versat. Após *return* é possível escrever código, porém este é acessível apenas pela instrução *goto* (explicada na secção 6.17). Um exemplo da

utilização do *return* é:

Meter um goto boot ROM no código para quando estes problemas acontecerem o Versat não continuar a correr. Meter um exemplo do return com codigo por baixo deste!!!!!!!

```
int main() {

    return 0;  }
```

6.20 Asm

A instrução asm é aplicada para permitir a escrita de *assembly* em código C++ do Versat. O asm varia consoante o compilador. No presente caso, optou-se por uma utilização análoga a do g++. Um exemplo do uso da instrução asm é:

```
int main() {

asm {
      ldi ALU_ADD
        wrc ALU0_CONFIG_ADDR,ALU_CONF_FNS_OFFSET
        ldi smul1
        wrc ALU0_CONFIG_ADDR,ALU_CONF_SELA_OFFSET
        ldi salu0
        wrc ALU0_CONFIG_ADDR,ALU_CONF_SELB_OFFSET
        ldi 1     }

        return 0;

}
```

6.21 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 em:

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

Capítulo 7

Exemplos de código

7.1 Adição de vectores

7.2 Produto interno complexo

7.3 FFT

meter o codigo fft_expressions.cpp

Capítulo 8

Resultados

Capítulo 9

Conclusão

Os objectivos para esta introdução à tese foram cumpridos com sucesso. Houve dificuldade tanto a achar a solução a usar para fazer o compilador como a entender certos pormenores e limitações do *hardware*. Nesta fase não foi feito nenhum troço do compilador.

9.1 Trabalho feito

Nesta fase conseguiu-se fazer o estudo da arquitectura do Versat, com o objectivo de fazer o compilador e saber as limitações do *hardware*. Conseguiu-se também achar maneiras diferentes para fazer o compilador. Fez-se exemplos diferentes de código para se perceber as vantagens e desvantagens de cada abordagem de compilação.

Devido à presença do controlador do Versat, que é uma máquina de acumulador convencional, não se consegue excluir os aspectos comuns das linguagens de programação tais como as expressões condicionais e os ciclos. Por outro lado, observou-se que o Data Engine consegue ser modelado por classes e a sua programação feita apenas chamando métodos dessas classes.

9.2 Trabalho Futuro

Na próxima fase será estuda a forma de unificar a programação do controlador do Versat com a programação do Data Engine. Será feita a construção e respectivo teste do compilador. Serão feitos exemplos de código na linguagem construída com o objectivo de testar o funcionamento e a eficiência do compilador.

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. Nückel, 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. Huisken, 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] Bjorn De Sutter, Praveen Raghavan, and Andy Lambrechts. Coarse-grained reconfigurable array architectures. In Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, and Jarmo Takala, editors, *Handbook of Signal Processing Systems*, pages 449–484. Springer US, 2010.

[8] 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.

[9] 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.

[10] Carl Ebeling, Darren C. Cronquist, and Paul Franklin. Rapid - reconfigurable pipelined datapath. In *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, FPL '96, pages 126–135, London, UK, 1996. Springer-Verlag.

[11] P.M. Heysters and G.J.M. Smit. Mapping of DSP algorithms on the MONTIUM architecture. In *Proceedings of the International Parallel and Distributed Processing Symposium, 2003*, pages 6–, April 2003.

[12] Yongjun Park, J.J.K. Park, and S. Mahlke. Efficient performance scaling of future CGRAs for mobile applications. In *International Conference on Field-Programmable Technology (FPT), 2012*, pages 335–342, Dec 2012.

[13] Vaughn Betz, Jonathan Rose, and Alexander Marquardt, editors. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.