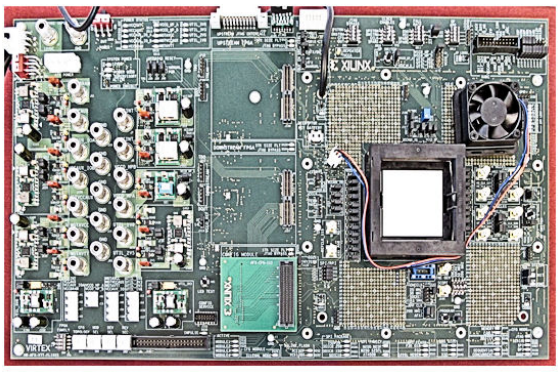
TÉCNICO LISBOA



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.

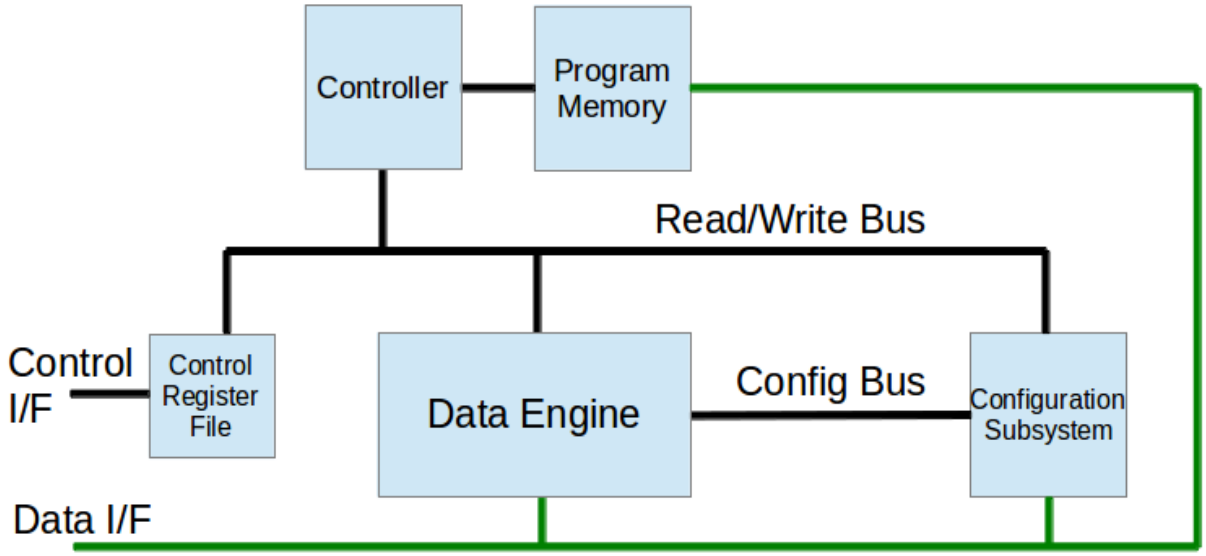


Figura 3.1: Esquema da unidade de topo.

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.

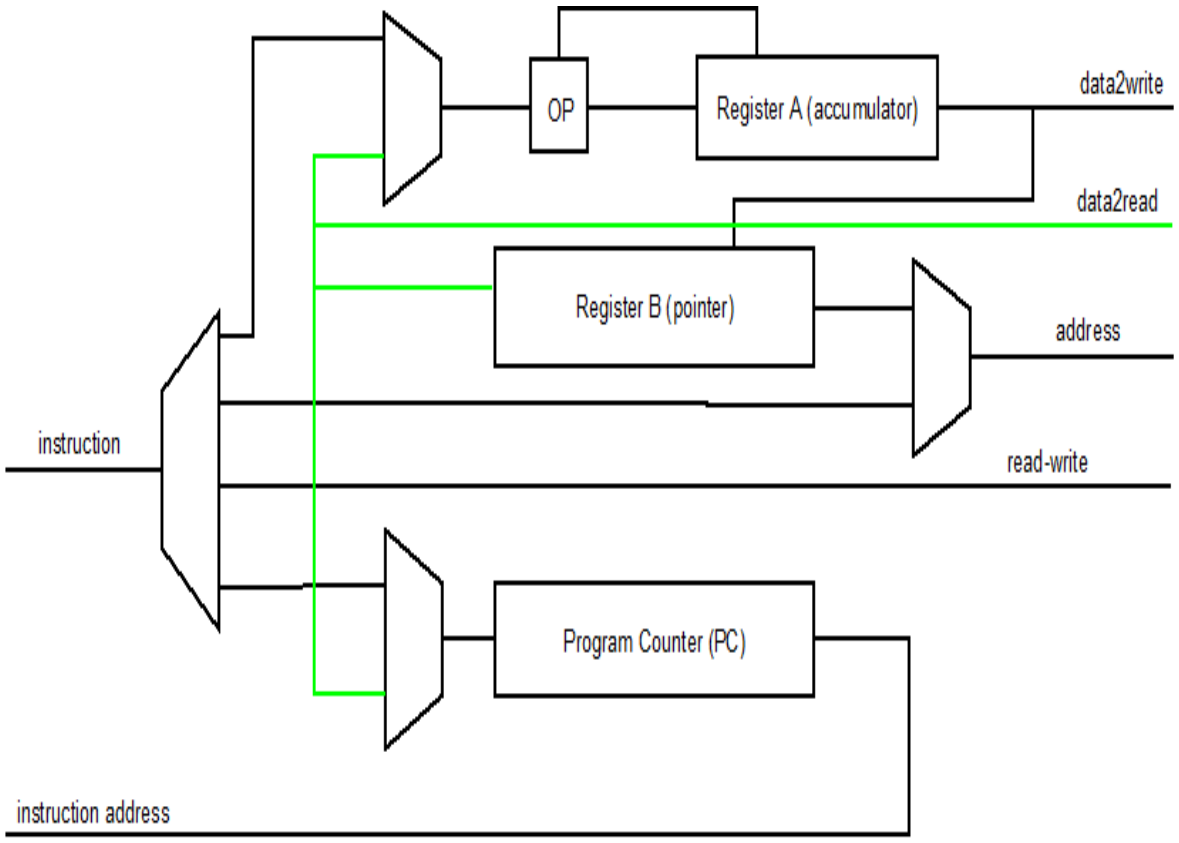


Figura 3.2: Esquema da unidade de controlo

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;
- Máximo;
- Mínimo;
- Valor absoluto.

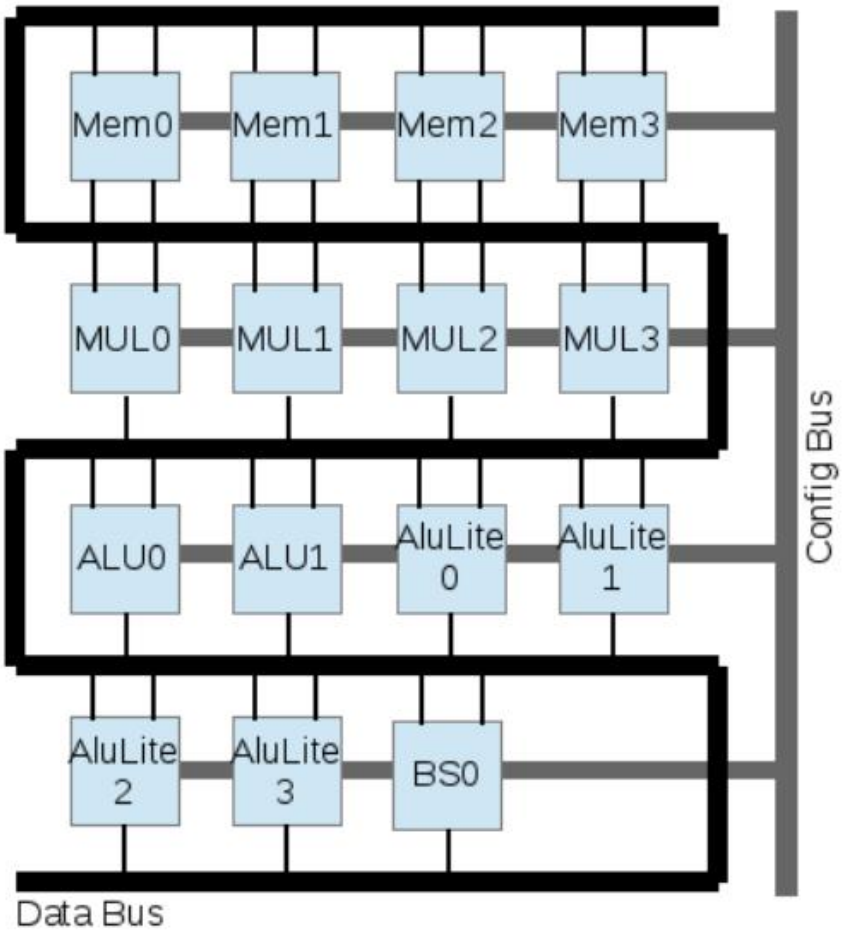


Figura 3.3: Esquema de ligações do Data Engine

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 5.1. 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.

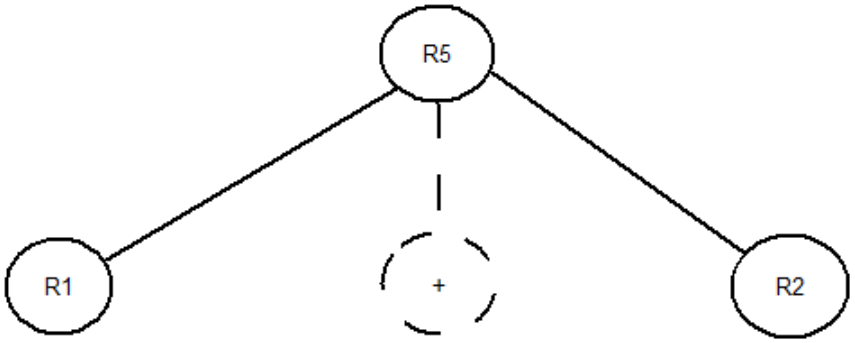


Figura 4.1: Árvore de sintaxe abstracta de uma soma.

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.

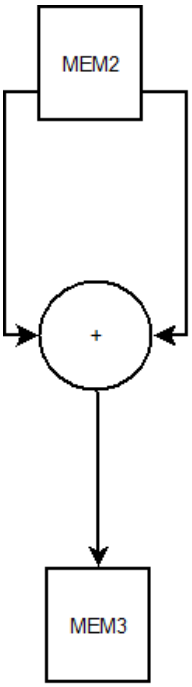


Figura 4.2: Árvore lexical de uma soma dentro de um ciclo for

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.

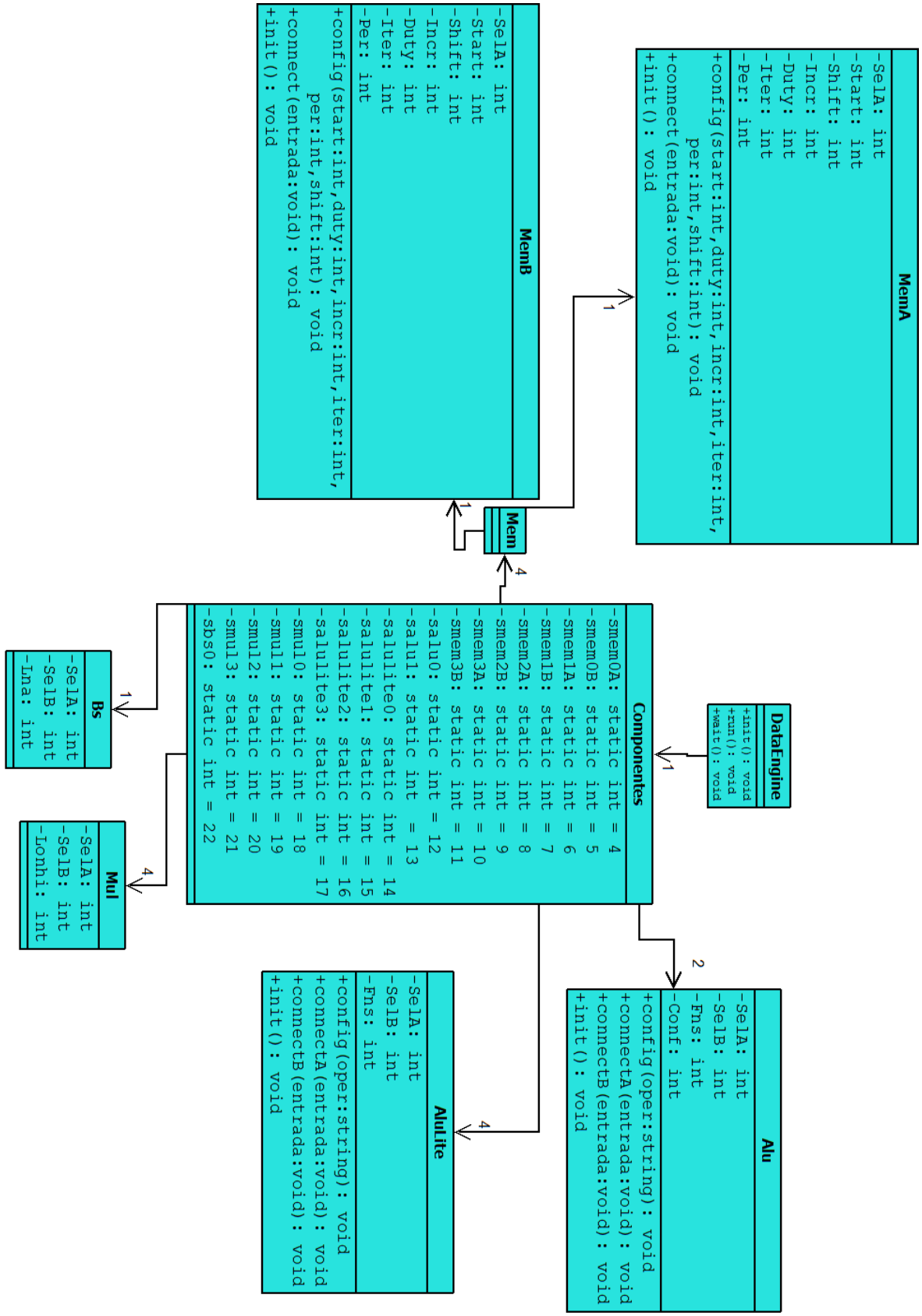


Figura 4.3: UML do Data Engine.

Capítulo 5

Resultados

falar com o professor para dividir os resultados em vários capítulos

5.1 Flex/bison

Tirado da wikipedia

O Flex é um gerador lexical. Ele normalmente é usado em conjunto com o gerador de *parse* Bison. Ao contrário do Bison, o Flex não faz parte do projecto GNU. O analisador lexical Flex tem complexidade temporal O(n) em relação ao tamanho da entrada. Isso significa que que faz um número constante de operações por cada simbolo de entrada.

O Bison é um gerador de *parse* que faz parte do projecto GNU. O Bison lê a especificação de uma linguagem criada pelo utilizador, verifica a existência de alguma ambiguidade e gera o *parse* em C, que lê sequências de *tokens* e decide se a sequência obedece à gramática definida.

5.1.1 Linguagem definida

Definiu-se a linguagem do compilador. Decidiu-se que o melhor era construir uma linguagem baseada em C++. Visto que o Versat não tem uma *stack* própria, é impossível criar funções em C++ do Versat. Para organizar o código em pseudo-rotinas é necessário usar a instrução *goto*. As instruções relativas às Alu e às AluLite estão definidas na tabela 5.2.

Tabela 5.1: Instruções em C do Versat para a Alu e a AluLite.

Unidade	Nº da unidade	Método	Descrição

Unidade	Nº da unidade	Método	Descrição
alu	0-1	setOper(oper)	Define um operador para a Alu seleccionada.
aluLite	0-3	setOper(oper)	Define um operador para a AluLite seleccionada.
alu	0-1	connectPortA(FU)	Connecta o porto A da alu à saída da unidade funcional passada como argumento.
aluLite	0-3	connectPortA(FU)	Connecta o porto A da aluLite à saída da unidade funcional passada como argumento.
alu	0-1	connectPortB(FU)	Connecta o porto B da alu à saída da unidade funcional passada como argumento.
aluLite	0-3	connectPortB(FU)	Connecta o porto B da aluLite à saída da unidade funcional passada como argumento.
alu	0-1	connectPortA(mem,port)	Connecta o porto A da alu à saída da memória passada como argumento. Quando se faz conecções a memórias, é necessário indicar o porto ao qual se quer conectar.
aluLite	0-3	connectPortA(mem,port)	Connecta o porto A da aluLite à saída da memória passada como argumento. Quando se faz conecções a memórias, é necessário indicar o porto ao qual se quer conectar.
alu	0-1	connectPortB(mem,port)	Connecta o porto B da alu à saída da memória passada como argumento. Quando se faz conecções a memórias, é necessário indicar o porto ao qual se quer conectar.
aluLite	0-3	connectPortB(mem,port)	Connecta o porto B da aluLite à saída da memória passada como argumento. Quando se faz conecções a memórias, é necessário indicar o porto ao qual se quer conectar.

Tabela 5.2: Instruções respectivas à Alu e à AluLite.

5.1.2 Instruções do Data Engine

5.1.3 Instruções do controlador

5.2 Geração de assembly

5.3 Teste e eficiência do compilador

Capítulo 6

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.

6.1 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.

6.2 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.