UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

ALEXIS ANTON LAZZAROTTO

Implementação em hardware de um Módulo de Tradução Binária para uma Arquitetura Reconfigurável

Trabalho de Graduação.

Prof. Dr. Luigi Carro
Orientador

Porto Alegre, julho de 2011.

AGRADECIMENTOS

Em primeiro lugar, um Muito Obrigado aos meus pais, que considero pais maravilhosos, a quem devo grande parte de minhas conquistas e cujo apoio tem importância inestimável para mim.

Agradeço aos meus irmãos e aos meus amigos, cuja companhia e apoio são para mim um privilégio.

Ao Prof. Luigi, pelo voto de confiança.

Ao Mateus, pela acolhida ao LSE e pelo acompanhamento ao longo de meu trabalho no grupo de pesquisa.

A todos os outros colegas do LSE, pela ajuda e companhia.

A todos que de alguma forma me apoiaram ou contribuíram para a realização deste trabalho.

Por fim, muito obrigado à Lúcia, pelo apoio, carinho e compreensão.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

CAD	*Computer-Aided Design*
DIM	*Dynamic Instruction Merging*
DSP	*Digital Signal Processing*
FPGA	*Field-Programmable Gate Array*
ILP	*Instruction-Level Parallelism*
IPC	*instructions per cycle*
ISA	*Instruction Set Architecture*
LSE	Laboratório de Sistemas Embarcados
LUT	*Look-Up Table*
PC	*program counter*
RAW	*Read After Write*
RBT	*Read Bitmap Table*
RISC	*Reduced Instruction Set Computer*
RTL	*Register Transfer Level*
TB	Tradutor Binário
TLP	*Thread-Level Parallelism*
WAR	*Write After Read*
WAW	*Write After Write*
WBT	*Write Bitmap Table*
ULA	Unidade Aritmética e Lógica
UFR	Unidade Funcional Reconfigurável
UFRGS	Universidade Federal do Rio Grande do Sul
VHDL	*VHSIC Hardware Description Language*
VLIW	*Very Long Instruction Word*

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A complexidade dos sistemas embarcados está crescendo devido à agregação de funcionalidades em um único dispositivo eletrônico. Além disso, a aceleração da execução dos processadores superescalares está estagnada, e a extração de paralelismo no modelo von Neumann está chegando ao limite teórico. Arquiteturas Reconfiguráveis aparecem como uma solução viável para estes problemas, sendo factível a implementação deste tipo de arquitetura nas atuais tecnologias CMOS. O presente trabalho consiste na implementação em hardware de um módulo de Tradução Binária para a arquitetura reconfigurável DIM, desenvolvida no Laboratório de Sistemas Embarcados do Instituto de Informática da UFRGS.

Palavras-Chave: Arquiteturas reconfiguráveis, tradutor binário, VHDL, implementação, módulo de hardware.

Hardware Implementation of a Binary Translation Module for a Reconfigurable Architecture

ABSTRACT

The complexity of embedded systems is growing due to the aggregation of multiple functionalities into a single electronic device. Moreover, acceleration of the execution of superscalar processors is stagnated, and the extraction of parallelism in the von Neumann model is reaching its theoretical limit. Reconfigurable architectures show up as a viable solution to these problems, and their implementation is feasible in current CMOS technology. This work consists of the implementation of a Binary Translation hardware module for the DIM reconfigurable architecture, developed by the Embedded Systems Research Group at the Federal University of Rio Grande do Sul (UFRGS).

Keywords: Reconfigurable architectures, binary translation, VHDL, implementation, hardware module.

1 INTRODUÇÃO

A continuada redução das dimensões dos transistores devido a processos de fabricação cada vez mais sofisticados tem permitido um aumento exponencial, nas últimas décadas, do número de dispositivos numa única pastilha de silício (chip), fato previsto pela conhecida Lei de Moore. No entanto, tal aumento na densidade de componentes não implica um aumento proporcional no desempenho dos processadores. O desempenho está atrelado, entre outros fatores, ao projeto de micro-arquitetura do sistema.

Em se tratando de uma arquitetura no domínio de Sistemas Embarcados, entretanto, alto desempenho não é a única característica desejável. Em muitos casos, estes sistemas são alimentados por baterias, limitadas tanto no tempo de vida útil quanto em sua capacidade instantânea de fornecer corrente e tensão. Por isso, arquiteturas para sistemas embarcados devem ser projetadas levando em conta a operação dentro de limites de potência bem definidos, e de forma a consumirem a menor energia possível. As arquiteturas atuais de propósito geral que apresentam o maior desempenho, as chamadas arquiteturas superescalares, são complexas e altamente dispendiosas em termos energéticos (AUSTIN et al., 2004), características que as tornam não-atrativas para uso nos sistemas embarcados futuros.

Existem alternativas às arquiteturas superescalares. Arquiteturas VLIW são um exemplo. Estas relegam à palavra de instrução (e, por extensão, ao compilador) o controle de suas várias unidades funcionais. Por isso, apresentam um custo de projeto reduzido e mostram-se eficientes do ponto de vista energético (HENNESSY; PATTERSON, 2003). No entanto, como dependem da existência de um compilador específico para a alocação eficiente de instruções, exigem que as aplicações já existentes sejam recompiladas para que possam ser executadas neste tipo de arquitetura. Em outras palavras, as arquiteturas VLIW não provêem compatibilidade binária com aplicações existentes. Isso dificulta enormemente a adoção massiva das mesmas no mercado de sistemas embarcados, onde o *time-to-market* é fator crucial para o sucesso (ou insucesso) dos produtos.

Paralelamente à questão energética, o desempenho é outra preocupação no domínio embarcado. Com a recente convergência de várias funções a um único dispositivo (como *smartphones* e *tablets*), é cada vez maior a necessidade, por parte do hardware, de suportar (com bom desempenho) a execução de aplicações com perfis diversos, como, por exemplo, aplicações que envolvem processamento digital de sinais (DSP) ou processamento gráfico. A chave para esta capacidade de adaptação a cenários distintos é a capacidade de explorar os diferentes níveis de paralelismo inerentes às aplicações, de modo que os recursos de hardware sejam aproveitados da melhor forma possível. Uma

arquitetura que é capaz de identificar e aproveitar grande parte do paralelismo de diferentes tipos de aplicações terá um bom desempenho numa ampla gama de cenários.

Arquiteturas superescalares têm demonstrado limitações nesse sentido. Estas são uma implementação sofisticada do modelo de computação de von Neumann, o qual impõe um limite à exploração de paralelismo em nível de instruções (ILP) das aplicações. De fato, como exposto em (SIMA, 2004), os processadores superescalares da Intel têm apresentado uma estagnação na capacidade de extração de ILP das aplicações (Figura 1.1). Assim, faz-se necessária a investigação de outros modelos de computação que permitam obter alto desempenho numa ampla gama de cenários de aplicação.

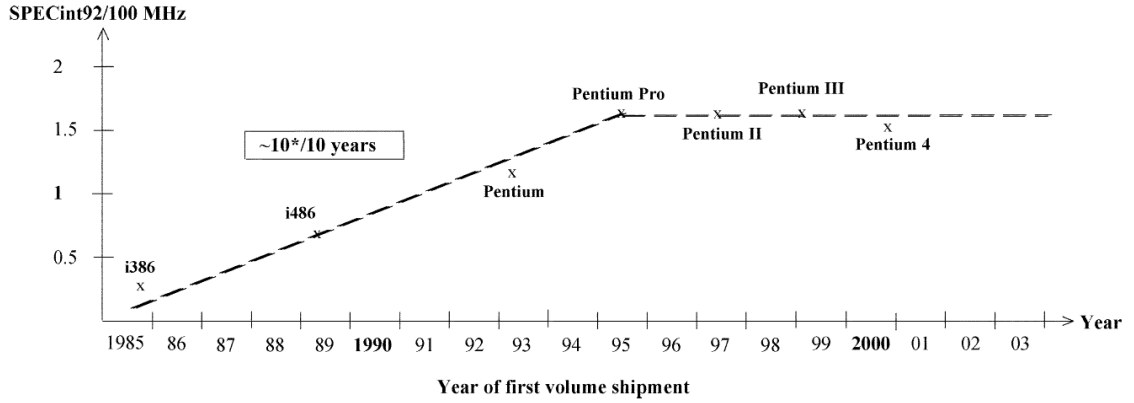

Figura 1.1: Estagnação da exploração de ILP nos processadores Pentium (SIMA, 2004)

Nesse contexto, arquiteturas reconfiguráveis têm se mostrado uma alternativa promissora. A capacidade de adaptação a aplicações de diferentes domínios é uma característica comum das mesmas. Algumas das arquiteturas se propõem a estender o modelo de von Neumann; outras se baseiam em modelos de computação diferentes, sendo que com freqüência o modelo *dataflow* é utilizado. Outras arquiteturas, ainda, combinam ambos os modelos.

Em geral, arquiteturas reconfiguráveis têm demonstrado grande potencial, tanto para o aumento de desempenho quanto para redução do consumo energético. Elas comumente apresentam uma malha de blocos funcionais reconfiguráveis, isto é, blocos cuja operação e interconexão são *configuradas* (definidas) por bits de controle. É justamente essa reconfigurabilidade que dá a essas arquiteturas a capacidade de adaptação a diferentes aplicações. Há duas estratégias distintas para se realizar o processo de configuração. Uma delas é definir os bits de controle estaticamente, ou seja, em tempo de compilação. Muitas arquiteturas reconfiguráveis adotam essa estratégia. Nesses casos, é comum que a arquitetura possua um conjunto de instruções próprio (ISA, *instruction set architecture*). A desvantagem dessa abordagem é a quebra da compatibilidade binária com aplicações existentes, pois estas devem ser recompiladas para que possam ser executadas na malha reconfigurável, o que implica a existência de um *toolchain* especial para o desenvolvimento de software nessas arquiteturas. Isso impede sua adoção massiva, da mesma forma como acontece com arquiteturas VLIW.

A segunda estratégia é realizar a reconfiguração dinamicamente, em tempo de execução. Nesse caso, ao mesmo tempo em que a aplicação executa num processador convencional, o código binário da mesma é analisado e os bits de configuração para a malha reconfigurável são determinados. Assim, na próxima vez em que o programa chegar ao mesmo bloco de código, a malha reconfigurável será redefinida pelos bits da

configuração correspondente e o trecho de código será executado de forma otimizada nessa malha. Assim, não há quebra de compatibilidade binária, visto que a aplicação foi desenvolvida e compilada para o processador convencional. O sistema DIM (BECK, 2008), arquitetura na qual este trabalho se baseia, desenvolvido no Laboratório de Sistemas Embarcados do Instituto de Informática da UFRGS e descrito em detalhes no Capítulo 3, é um dos sistemas reconfiguráveis que justamente mantêm a compatibilidade de software com arquiteturas existentes.

Um aspecto crucial para a reconfigurabilidade das arquiteturas dinâmicas acima mencionadas é a tradução das seqüências de instruções do código binário da aplicação para bits de configuração correspondentes, de modo que a semântica da aplicação seja mantida. No sistema DIM, essa tradução é realizada pelo módulo denominado Tradutor Binário (TB), que é justamente o objeto do presente trabalho. A implementação "em hardware" do módulo consiste na descrição do mesmo em VHDL.

A motivação para a implementação em VHDL é a possibilidade de extrair informações precisas de potência, energia, desempenho e área, pois o compromisso entre esses parâmetros é a questão essencial na exploração do espaço de projeto no domínio embarcado, como já mencionado. Essas informações, por sua vez, permitem verificar a complexidade real do hardware do TB, comparando-a com a complexidade de módulos similares de detecção de paralelismo.

No Capítulo 2, será apresentado um panorama do contexto de arquiteturas reconfiguráveis. Serão abordados trabalhos já propostos na área e será descrita uma breve classificação de tais arquiteturas.

O sistema DIM, no qual este trabalho se baseia, é descrito em detalhes no Capítulo 3. O entendimento do sistema é necessário para a compreensão do trabalho desenvolvido. São discutidos seu modelo de execução, a arquitetura e o módulo de Tradução Binária.

O Capítulo 4 detalha a implementação do Tradutor Binário, enquanto que o Capítulo 5 apresenta os resultados do trabalho. A conclusão e sugestões de trabalhos futuros estão no Capítulo 6.

2 CONTEXTUALIZAÇÃO

Neste capítulo é apresentada uma visão geral sobre o estado-da-arte da pesquisa em arquiteturas reconfiguráveis. Primeiramente será apresentada uma possível classificação dos sistemas existentes. Após, serão descritas brevemente algumas das arquiteturas reconfiguráveis já propostas.

2.1 Classificação

Não existe um consenso na comunidade acadêmica em relação a uma classificação das arquiteturas reconfiguráveis. Assim, será considerada a classificação apresentada em (COMPTON, 2002), em que as arquiteturas são distinguidas com base em três aspectos: acoplamento, granularidade e mecanismo de reconfiguração.

2.1.1 Acoplamento

O termo "acoplamento" refere-se à proximidade entre o processador de propósito geral do sistema e a unidade funcional reconfigurável (UFR). No projeto de uma arquitetura reconfigurável, a escolha do tipo de acoplamento reflete diretamente na eficiência do sistema. Uma UFR que é implementada como uma unidade funcional dentro do processador é chamada de *fortemente acoplada*. A comunicação entre o processador e a UFR ocorre somente dentro do núcleo, possibilitando um alto desempenho.

Outra forma de modelar a UFR é como um co-processador do processador principal. Normalmente a escolha deste tipo de acoplamento está ligada a área disponível de silício dentro do núcleo. Quando não existe área suficiente para armazenar a UFR dentro do núcleo, este tipo de acoplamento é utilizado. Nesta abordagem, a UFR é acoplada ao processador por um barramento externo ao núcleo. Assim, o custo de comunicação entre o processador e a UFR é mais elevado do que na abordagem anterior.

Existem outras técnicas de acoplamento, chamadas de *fracamente acopladas*, que apresentam alto custo de comunicação. A UFR anexada é um exemplo deste tipo de abordagem: ela fica localizada no barramento que conecta a memória cache e a interface de entrada/saída ao processador. O custo de comunicação é alto; no entanto, é menor do que na abordagem de acoplamento independente. Nesta última, o processador se comunica com a UFR através do barramento de entrada/saída. Dentre todas as abordagens, esta é a mais fracamente acoplada.

2.1.2 Granularidade

Cada UFR pode ser implementada através de diferentes tipos e tamanhos de blocos funcionais. Por exemplo, pode-se formar uma UFR somente com somadores independentes de um bit, ou utilizar um somador de 32 bits como unidade básica. Este tipo de escolha de projeto é denominado granularidade da UFR.

A granularidade escolhida para os blocos funcionais tem influência no número de bits necessários para reconfiguração. Utilizando o exemplo anterior, uma UFR formada por somadores de um bit como unidade básica implicam um elevado número de bits para realizar uma soma de 32 bits. Entretanto, se for realizado o encapsulamento de 32 somadores de um bit em uma caixa preta, abstrai-se a complexidade da configuração dos 32 somadores na reconfiguração da UFR, tornando o controle mais simples para a execução da mesma tarefa. Nesse caso, denomina-se a primeira abordagem uma arquitetura de *grão fino* e a segunda, uma arquitetura de *grão grosso*.

2.1.3 Mecanismo de reconfiguração

Várias propostas já foram apresentadas em relação ao mecanismo de reconfiguração da UFR. Em (HUTCHINGS, 1997; HUTCHINGS, 1999) foram demonstrados mecanismos que, em tempo de compilação, extraem partes do programa que podem ser executadas de forma mais eficiente na UFR. Entretanto, estas técnicas são dependentes de algum tipo de ferramenta para realizar esta tarefa, modificando o código compilado original. A conseqüência deste tipo de abordagem é o aumento do tempo do projeto com a adição de uma nova etapa no fluxo, além de não prover compatibilidade de software, já que existe a necessidade de recompilação do código. Em geral, essas são consideradas técnicas de reconfiguração *estática*.

Stitt (LYSECKY, 2006) foi um dos pioneiros a utilizar técnicas que, em tempo de execução, detectam partes do código da aplicação que podem ser executadas de forma eficiente na unidade reconfigurável. A vantagem provida por esta abordagem é a não-modificação do código compilado, provendo compatibilidade de software e a conseqüente diminuição do *time-to-market* do dispositivo. Essas técnicas são denominadas *dinâmicas*.

2.2 Trabalhos relacionados

Nesta seção são apresentados alguns trabalhos que já se preocuparam com a questão da performance em sistemas computacionais futuros e propuseram arquiteturas alternativas para uma melhor extração do paralelismo de aplicações e maior eficiência energética.

2.2.1 Chimaera

Chimaera (HAUCK, 1997) é uma arquitetura reconfigurável fortemente acoplada. Consiste num processador de propósito geral com uma unidade funcional reconfigurável (UFR) integrada, projetada como um FPGA simplificado otimizado para a aceleração de pequenas funções. O bloco lógico da UFR pode ser configurado como uma 4-LUT, como duas 3-LUTs ou como uma 3-LUT com computação de *carry*.

Existem instruções específicas que fazem o processador executar uma função presente na unidade reconfigurável. São usadas técnicas de reconfiguração parcial em tempo de execução para gerenciar a lógica reconfigurável. O mapeamento de operações

para a UFR está contido no segmento de código do binário da aplicação. Se a operação requisitada pela instrução não estiver presente na UFR, o processador é parado até que esta busque a configuração da memória e se reconfigure apropriadamente.

A UFR tem acesso direto de leitura a um subconjunto dos registradores do processador – são as configurações que determinam quais são eles. Uma única instrução da UFR pode usar até nove registradores. O acesso a estes pode ser implementado como uma cópia do banco de registradores do processador hospedeiro ou como conexões de leitura ao mesmo.

Para se fazer uso efetivo da unidade reconfigurável, fazem-se necessárias ferramentas auxiliares específicas para a tecnologia (compilador e ferramentas de CAD – mapeamento, posicionamento e roteamento). Estas ferramentas produzem um grafo de blocos lógicos que são posicionados em LUTs na UFR e roteados apropriadamente.

A arquitetura Chimaera, em suma, torna a UFR uma unidade funcional do processador. Diminui, com isso, o tempo de comunicação e de reconfiguração. No entanto, não apresenta compatibilidade binária com arquiteturas existentes, por exigir ferramentas especiais para permitir a execução de aplicações.

2.2.2 TRIPS

A arquitetura TRIPS (SANKARALINGAM et al., 2003) permite que um único conjunto de elementos de processamento e de armazenamento possa ser configurado de modo a apresentar alto desempenho em vários domínios de aplicação. Para isso, utiliza núcleos grandes e de grão grosso, de modo que uma aplicação de uma única thread com alto ILP possa ser executada com alto desempenho. No entanto, o sistema pode ser logicamente subdividido para suportar igualmente bem aplicações com TLP e DLP, o que é feito por meio de mecanismos como gerenciamento de estações de reserva e de blocos de memória, mecanismos considerados “polimórficos” pelos autores.

Um *TRIPS chip* (ou seja, um processador com a arquitetura TRIPS) é composto por bancos de memória e *TRIPS cores*. Cada *core* (núcleo) contém caches de dados e de instruções, bancos de registradores e uma matriz homogênea de nós processantes, cada um contendo uma ULA, uma unidade de ponto flutuante, um conjunto de estações de reserva e conexões de roteamento na entrada e na saída.

Os autores apresentam três (dentre várias) configurações possíveis para um processador TRIPS, uma para cada tipo de paralelismo, explicando suas diferenças no gerenciamento dos recursos polimórficos dos núcleos. O objetivo do sistema é alcançar o desempenho de sistemas de propósito específico, como, por exemplo, processadores gráficos e DSPs, evitando, assim, a necessidade de sistemas heterogêneos. Os resultados mostram que a arquitetura é capaz de atingir tal objetivo. No entanto, ela tem uma ISA própria e, portanto, torna necessária a existência de um *toolchain* específico para o sistema.

2.2.3 WaveScalar

WaveScalar (SWANSON et al., 2003) consiste numa ISA e num modelo de execução dataflow que preservam a semântica tradicional (estilo von Neumann) de memória, característica que nenhuma outra arquitetura dataflow anterior possui. As instruções WaveScalar são armazenadas e executadas numa cache de instruções distribuída chamada *WaveCache*. Esta consiste numa grade de aproximadamente 2K elementos processantes agrupados em clusters de 16, cada cluster cercado por módulos

de cache e por buffers de armazenamento (para comunicação entre clusters). Dentro de um cluster, a comunicação se dá por barramentos compartilhados. A WaveCache lê as instruções da memória e as atribui aos elementos processantes para execução. Cada elemento processante contém lógica para controlar o posicionamento e execução de instruções, filas de entrada e saída para os operandos, lógica de comunicação e uma unidade funcional.

Um executável WaveScalar contém uma codificação do grafo do fluxo de dados (*dataflow graph*) do programa. Além de instruções típicas de arquiteturas RISC, a ISA apresenta instruções especiais para gerenciar o fluxo de controle. Assim, o grafo do fluxo de dados torna-se também o grafo do fluxo de controle (*control flow graph*). O grafo pode ser visto como composto por múltiplas *waves*, porções acíclicas, dirigidas, conexas e de uma única entrada do mesmo. O compilador particiona o grafo da aplicação em *waves* maximais e adiciona a elas instruções de gerenciamento de *waves*, como por exemplo INDIRECT-SEND, que permite a chamada de subprogramas e o retorno/passagem de valores entre *waves*.

A manutenção da semântica von Neumann de memória é feita por meio da anotação das instruções de leitura/escrita da memória. O compilador atribui estaticamente um número de seqüência único (dentro da *wave*) a cada operação de memória, percorrendo o grafo em largura. Além disso, cada instrução de memória é marcada também com os números de seqüência das operações predecessora e sucessora, se estes puderem ser determinados. Com estes números, o sistema de memória pode executar as operações na ordem correta, forçando a ordenação load-store que o programa espera.

A WaveCache supera uma arquitetura superescalar agressiva (em IPC) por um fator de 3,1 em média, usando benchmarks SPEC e MediaBench. Também supera o sistema TRIPS em várias aplicações. Naturalmente, entretanto, como possui ISA própria, exige um *toolchain* próprio e impede a compatibilidade binária com aplicações existentes.

2.2.4 Warp processor

O processador Warp, descrito em (LYSECKY, 2006), é um sistema que combina um microprocessador, um módulo FPGA, uma unidade de perfilamento (*profiling*) de software e um módulo de CAD. Sua operação básica consiste em identificar (via *profiling*) as regiões críticas de um código binário e traduzi-las dinamicamente para circuitos mapeáveis no FPGA. Essa tradução é feita pelo módulo de CAD. Após, o binário é modificado para que nas próximas vezes em que a região crítica for executada, a execução se dê no FPGA em vez de no processador.

O módulo de CAD realiza as tarefas de decompilação, particionamento, síntese comportamental e RTL, síntese lógica, mapeamento para a tecnologia (no caso, blocos lógicos configuráveis e LUTs), posicionamento e roteamento. Essas tarefas são implementadas em software e são executadas num processador dedicado, embora seja possível sua execução no próprio processador principal, competindo com a aplicação por recursos de memória e de computação. Além disso, os respectivos algoritmos são simplificados, pois devem poder ser executados em tempo mínimo e com pouco uso de memória.

O módulo FPGA foi desenvolvido especialmente para otimizar o funcionamento sistema. Isso é alcançado por meio de matrizes de chaveamento limitadas, que permitem

um processo de posicionamento e roteamento mais simples. O módulo de CAD, portanto, leva em conta essa estrutura específica para conseguir realizar o roteamento eficientemente.

A arquitetura Warp dispensa o uso de compiladores especiais, visto que o código binário é analisado em tempo de execução. Essa característica difere da de vários sistemas comerciais que também realizam particionamento hardware/software, os quais utilizam um compilador próprio para realizar tal particionamento. Assim, nesses sistemas, não há compatibilidade binária com softwares existentes, o que dificulta sua adoção massiva.

Como desvantagem, pode-se mencionar que não são todas as aplicações que podem ser aceleradas nesta arquitetura. Esta é apropriada apenas para aplicações passíveis de aceleração usando FPGAs, cujas regiões críticas não contêm aritmética de ponto flutuante, alocação dinâmica de memória, recursão ou ponteiros (a não ser para acesso a arrays).

2.2.5 DIM: Dynamic Instruction Merger

O sistema DIM (BECK, 2008), arquitetura na qual este trabalho se baseia, descrito em detalhes no Capítulo 3, é um dos sistemas reconfiguráveis considerados dinâmicos, que justamente mantêm a compatibilidade de software com arquiteturas existentes. Ele analisa o código binário da aplicação paralelamente à execução no processador principal e constrói configurações para a UFR, armazenando-as numa cache de configurações. Posteriormente, a configuração é recuperada da cache e usada para reconfigurar a UFR, que passa a executar o bloco de código correspondente de forma combinacional. O sistema pode ser considerado de grão grosso, visto que é composto por ULAs, unidades de load/store e multiplicadores, por exemplo.

3 O SISTEMA RECONFIGURÁVEL DIM

O presente trabalho se baseia no sistema DIM (*Dynamic Instruction Merger*), descrito em (BECK, 2008) e desenvolvido no Laboratório de Sistemas Embarcados do Instituto de Informática da UFRGS. A ideia básica do sistema é acelerar a execução de aplicações de diversos domínios, ao mesmo tempo provendo compatibilidade binária com as aplicações existentes de uma dada ISA.

O sistema é composto por um processador de propósito geral, uma unidade funcional reconfigurável (UFR), um tradutor binário (TB) e uma cache de configurações. Esses componentes podem ser vistos na Figura 3.1. A UFR é uma matriz de unidades funcionais interconectada por um conjunto de multiplexadores que recebem dados de um contexto de entrada e escrevem os resultados do processamento em um contexto de saída. Tanto a UFR quanto o TB serão descritos em mais detalhes nas Seções 3.2 e 3.3. O modelo de execução será descrito a seguir.

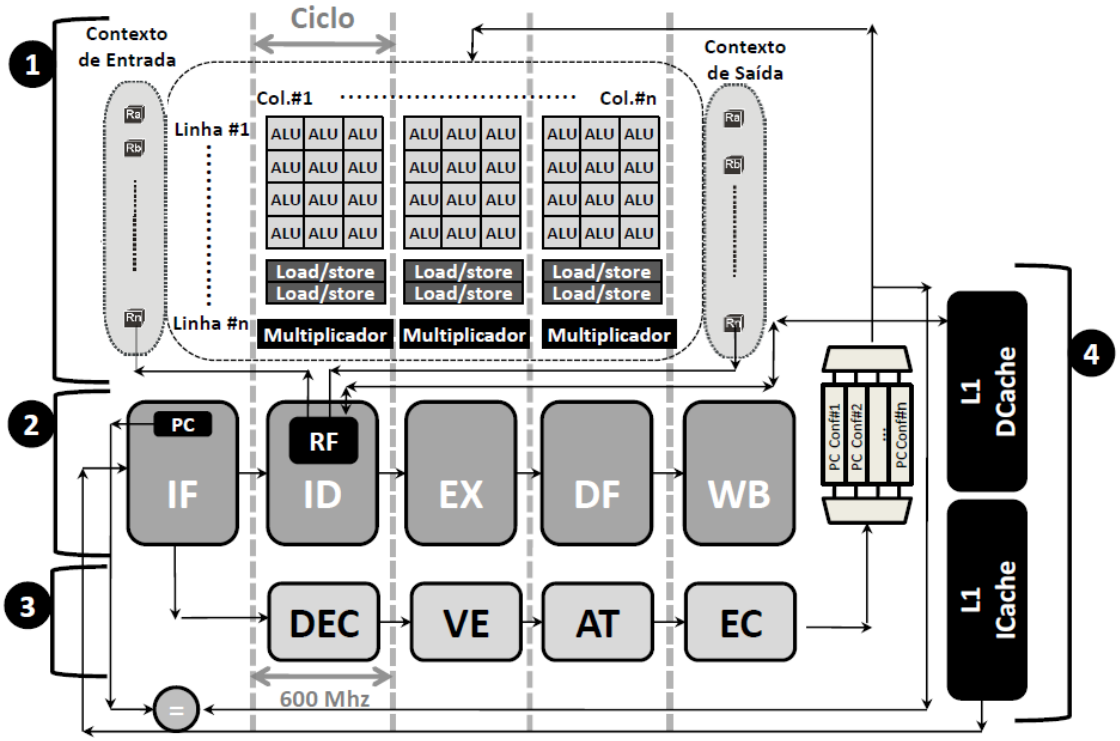

Figura 3.1: Organização do sistema DIM. (1) UFR; (2) processador; (3) TB (4) cache de configurações, juntamente com as caches de Instrução e de Dados do processador – adaptada de (RUTZIG, 2011)

3.1 Modelo de execução

O sistema reconfigurável DIM opera em três modos distintos, aqui chamados de "fases": a Fase de Detecção, a Fase de Reconfiguração e a Fase de Execução.

O sistema inicia na Fase de Detecção, em que o processador (bloco 2 na Figura 3.1) executa normalmente as instruções que busca da cache de instruções, enquanto que a UFR (bloco 1 na Figura 3.1) fica temporariamente inativa. A cache de configurações está inicialmente vazia. Em paralelo ao processador, o Tradutor Binário (bloco 3) recebe as instruções e monta progressivamente, com base nas dependências de dados destas, uma *configuração* para a UFR.

Uma configuração é um conjunto de informações (bits) que determinam com exatidão o caminho de dados da UFR, de tal modo que esta execute a seqüência de instruções correspondente. Além disso, uma configuração também contém os valores imediatos necessários e os bits que identificam a operação necessária em cada unidade funcional. Cada configuração é univocamente identificada pelo valor do contador de programa (PC, do inglês *program counter*) quando da chegada da primeira instrução.

Assim que uma configuração é encerrada no tradutor binário, o mesmo adiciona os *bits de configuração* correspondentes (ou a palavra de configuração) à cache de configurações. O *tag* na cache é justamente o identificador da configuração – o PC da primeira instrução (mais informações sobre a cache na Seção 3.4).

Na próxima vez que o processador chegar neste mesmo valor do PC, ocorrerá um *cache hit*, pois existe uma configuração na cache para o código que está por ser executado. O processador é, então, congelado para que a execução se dê na UFR. Inicia-se, assim, a Fase de Reconfiguração:

- O caminho de dados entre as unidades funcionais da UFR é modificado de acordo com os bits fornecidos pela cache;
- As unidades funcionais (ULAs, unidades de load/store, etc.) são configuradas para executarem as operações apropriadas;
- O contexto de entrada da configuração (valores de registradores) é lido do processador e carregado na UFR. Os valores imediatos, que também fazem parte do contexto, estão contidos na palavra de configuração.

Em síntese: nesta fase, a UFR é preparada para executar as operações descritas na configuração. Isso é necessário somente se a configuração em questão ainda não tiver sido "carregada" na UFR; nas vezes subseqüentes em que ocorrer *cache hit* com o mesmo valor do PC, a configuração já estará carregada, e não será necessário efetuar novamente a reconfiguração.

A UFR, então, passa a executar as instruções de forma combinacional, ou seja, de forma otimizada. Esta é a Fase de Execução. Depois da execução, os valores dos registradores do contexto são escritos no processador, o controle retorna para o mesmo e o processamento segue.

A quantidade de instruções que se pode incluir numa configuração da UFR depende, em qualquer momento, de vários fatores: do tamanho da UFR (quantidade de unidades funcionais em paralelo e em seqüência na matriz), da dependência de dados entre as instruções e, por fim, do caráter das mesmas – não podem ser instruções de

desvio, pois uma configuração corresponde no máximo a um bloco básico do programa. Isso será exposto com mais clareza na seção 3.3.

Na Figura 3.1, os conectores entre os blocos indicam o caminho de dados permitido pelo modelo de execução descrito.

3.2 A Unidade Funcional Reconfigurável

Um exemplo da estrutura da UFR é mostrado na Figura 3.2. A UFR é basicamente uma matriz de unidades funcionais. Na figura, as unidades de cada linha horizontal operam em paralelo. No sentido vertical, a computação ocorre seqüencialmente ao longo do tempo, iniciando pela linha de baixo. Tanto o número de linhas quanto o número de colunas da matriz são parâmetros que podem ser alterados na síntese do circuito, de modo que existe espaço para experimentação com dimensões diferentes do sistema.

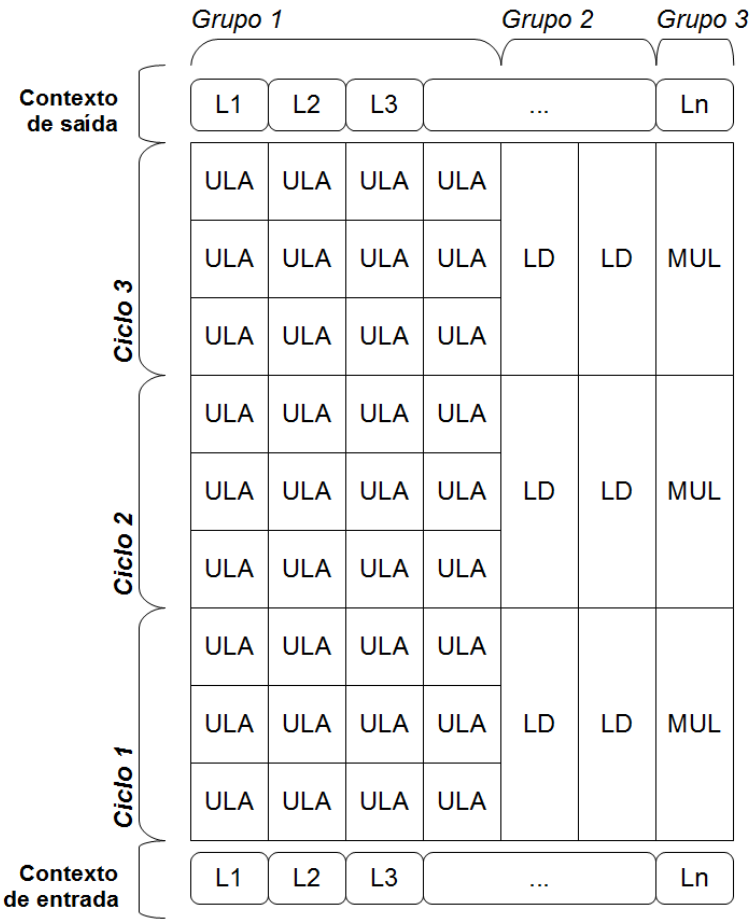

Figura 3.2: Estrutura da UFR

Essa matriz é permeada por *linhas de contexto*, que consiste numa lógica de roteamento de valores de registradores e valores imediatos através das linhas da matriz. Inicialmente, os valores dos registradores são lidos do banco de registradores do processador, enquanto que os valores imediatos estão contidos nos bits de configuração da cache. Ambos fazem parte do *contexto de entrada* da UFR. Cada linha de contexto corresponde a um registrador fixo (na configuração) ou a um valor imediato. Sua estrutura é mostrada na Figura 3.3. Na figura, é mostrado o caminho de dados a partir

do contexto de entrada de uma linha da UFR até o contexto de saída da mesma, passando por uma ULA. Os multiplexadores que selecionam a entrada de cada ULA são chamados *multiplexadores de entrada*; o multiplexador que seleciona o valor de cada linha de contexto é chamado *multiplexador de saída*.

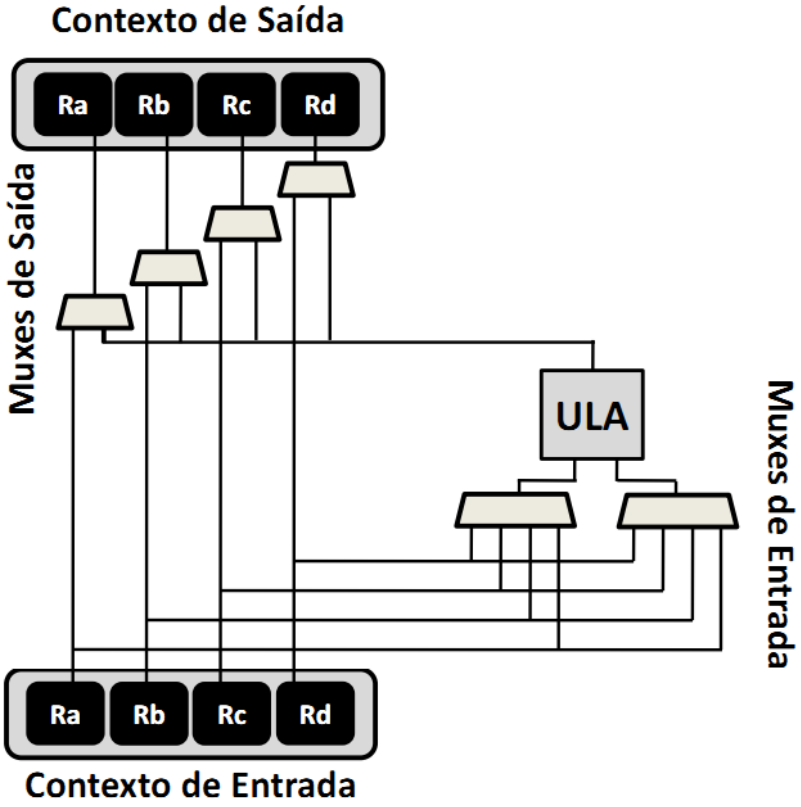

Figura 3.3: Estrutura das linhas de contexto da UFR (RUTZIG, 2011)

Cada linha de unidades funcionais é intercalada por tal lógica de roteamento de modo que cada unidade funcional possa receber a saída de qualquer unidade da linha anterior da matriz ou ainda o valor anterior, inalterado, de qualquer linha do contexto. Em termos mais precisos: cada unidade funcional da linha *n* da UFR pode receber qualquer saída ou qualquer entrada da linha *n* – 1 da UFR. Dessa forma, os valores contidos no contexto podem permanecer inalterados de uma linha da UFR para a outra.

Como se pode perceber claramente na Figura 3.2, há diferentes grupos de unidades funcionais. Os grupos 1, 2 e 3 correspondem, respectivamente, aos grupos ULA, LD (unidades de load/store) e MUL (multiplicadores). Pela figura, fica claro que diferentes grupos podem ter diferentes atrasos. No caso, o atraso de três ULAs é equivalente ao de uma unidade de load/store. Como o acesso à memória é síncrono, o atraso da unidade de load/store é exatamente um período de clock. Assim, cada conjunto de unidades com atraso igual a um período de clock é denominado ciclo (ou nível) da UFR.

3.3 O Tradutor Binário

O Tradutor Binário é o módulo responsável por construir configurações para a unidade reconfigurável e armazená-las na cache de configurações. O TB recebe como entrada as mesmas instruções que o processador principal lê da memória, analisa suas dependências com instruções já presentes na configuração atual, atualiza o contexto, dispõe as informações da configuração num formato apropriado e liga o sinal de escrita

na cache de configurações quando a configuração atual é encerrada. Por causa disso, o TB do sistema DIM não é um tradutor binário convencional, pois não traduz um binário de uma ISA para alguma outra. Ele traduz um binário da ISA do processador principal para palavras de configuração da UFR.

O TB tem a forma de um pipeline: é composto por vários estágios em série, como mostrado na Figura 3.1 (mais informações sobre a organização do TB na seção 4.1). Como já mencionado, ele opera em paralelo ao processador principal; portanto, não pode ter um caminho crítico maior que o do processador, caso em que o TB degradaria o desempenho do sistema.

O TB encerra uma configuração nos seguintes casos:

- A instrução é inválida, o que pode significar que:
  - a instrução atual não é suportada pela UFR, ou
  - o bloco básico encerra, isto é, a instrução atual é um desvio (*jump* ou *branch*, por exemplo)
- Ocorre estouro de recursos, nos casos em que
  - não há mais recursos (unidades funcionais) livres na configuração, ou
  - não há mais linhas, na configuração, em que não ocorre violação de dependência de dados
- Ocorre estouro no contexto da configuração: alcança-se o limite do número de registradores ou valores imediatos na tabela de contexto da configuração.

A operação do TB é baseada em várias tabelas. Isso ocorre por dois motivos: seu estado interno é apropriadamente mantido por meio de tabelas, visto que é necessária uma representação das células da UFR e das linhas de contexto já ocupadas na configuração sendo construída; e, em segundo lugar, a própria saída do TB (a palavra de configuração) consiste em tabelas, justamente para controlar a configuração da UFR. A seguir, são descritas as tabelas utilizadas.

3.3.1 Descrição das tabelas

Abaixo são descritas todas as tabelas nas quais o Tradutor Binário se baseia para construir uma configuração para a UFR. Os nomes das tabelas seguem a nomenclatura da descrição original do sistema em (BECK, 2008).

- *Write Bitmap Table* (WBT): um bit para cada registrador do contexto; uma linha para cada linha de unidades funcionais da UFR. Cada bit indica se o registrador correspondente é escrito em alguma operação realizada na respectiva linha de unidades funcionais. A tabela é necessária para a verificação de dependências RAW e WAW.
- *Resource Table* (tabela de recursos): um bit para cada "recurso" (unidade funcional – ULA, unidade de load/store ou multiplicador). Há uma tabela para cada grupo de unidades funcionais. Cada bit indica se há alguma instrução alocada no recurso correspondente na configuração atual.
- *Context Table* (tabela de contexto): consiste nos registradores do contexto, além dos operandos imediatos presentes nas instruções executadas pela UFR. Subdivide-se em *Current Table*, *Start Table* e *Immediate Table*.

  - *Current Table* (tabela “corrente”): contém os registradores utilizados na configuração atual. Atenta-se para o fato de que a tabela contém o identificador, ou o código, de cada registrador, e não seu conteúdo. O estado final desta tabela – o estado da tabela quando a configuração é encerrada – representa os registradores que serão escritos de volta (*written back*) ao processador quando a execução da UFR terminar. Esta tabela é utilizada na Fase de Detecção e na de Reconfiguração. (Obs.: como os operandos imediatos não se alteram durante a execução, não faz sentido os mesmos estarem presentes na *Current Table*.)
  - *Start Table* (tabela “inicial”): é um bitmap que indica quais registradores da *Current Table* devem ser lidos no início da execução da UFR, ou seja, informa o contexto de entrada. Esta tabela é utilizada na busca dos operandos de entrada na Fase de Reconfiguração.
  - *Immediate Table* (tabela de imediatos): contém os valores imediatos utilizados na configuração. Estes são inseridos, pelo TB, na palavra de configuração.

- *Read Table*: para cada unidade funcional da UFR, indica quais registradores (dentre os presentes na *Context Table*) serão lidos pela unidade. Há uma tabela para cada grupo. Esta tabela controla os multiplexadores de entrada das unidades funcionais da UFR (Figura 3.3).
- *Write Table*: Para cada linha de contexto, a tabela informa de onde a linha receberá o valor: ou a tabela contém a coluna de origem do valor, ou contém zero, caso em que o valor anterior é mantido. Esta tabela controla os multiplexadores de saída da UFR (Figura 3.3).
- *Function Table*: Informa a função realizada pela unidade funcional. Há uma tabela para cada grupo de unidades funcionais. Essa tabela é essencial para ULAs e unidades de load/store, visto que cada uma delas é capaz de realizar mais de uma operação.

Na Figura 3.4 é mostrado um exemplo de uma possível *Write Bitmap Table* (WBT). A tabela da figura corresponde a uma UFR com 6 linhas de unidades funcionais e a um processador com 16 registradores. Na primeira linha, os registradores R1 e R4 são escritos. Na terceira linha, são escritos R1, R3 e R5, e assim por diante. Uma possível *Resource Table* para este caso é mostrada na Figura 3.5, onde se supõe que a UFR tenha 4 unidades funcionais em cada linha. A figura mostra que há 7 instruções alocadas na configuração.

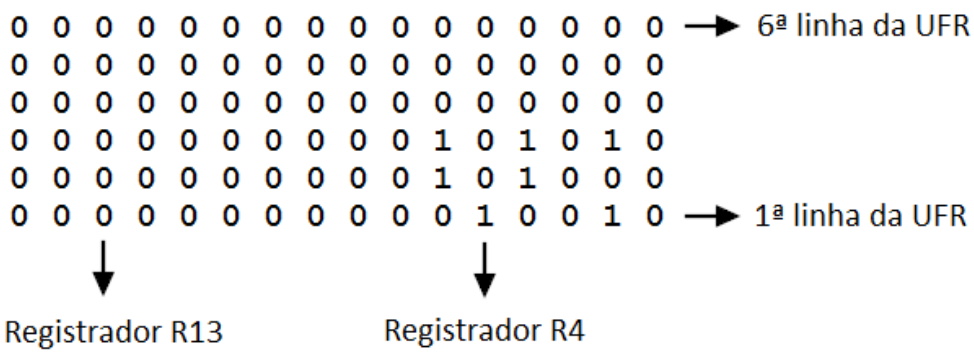

Figura 3.4: Exemplo da tabela *Write Bitmap Table* (WBT)

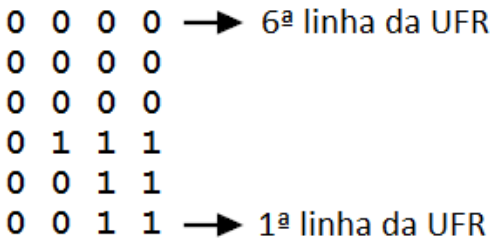

Figura 3.5: Exemplo da tabela *Resource Table*

Um exemplo da tabela *Read Table*, para o mesmo caso apresentado nas Figuras 3.4 e 3.5, é mostrado na Figura 3.6. Pode-se perceber que sua estrutura é semelhante à da *Resource Table*, apenas contendo dois inteiros em vez de um bit em cada posição. Conforme mostrado na tabela, as duas instruções alocadas na primeira linha da UFR lêem os registradores R1, R2 e R3. A *Function Table* também tem a mesma estrutura, com alguns bits em cada posição para controlar a operação realizada na unidade funcional correspondente.

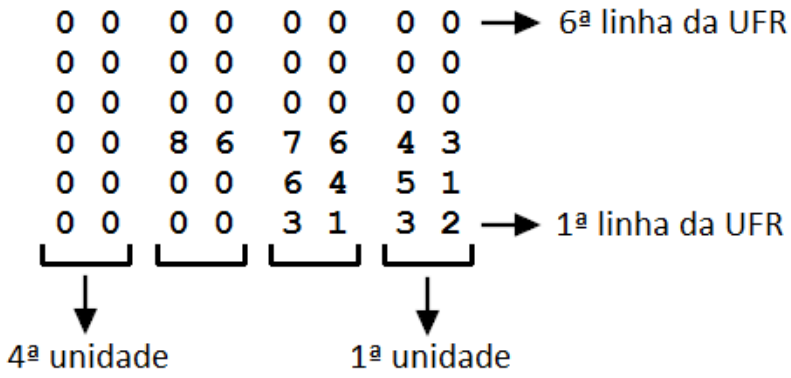

Figura 3.6: Exemplo da tabela *Read Table*

De posse da *Read Table* e da WBT, pode-se perceber por que as instruções foram alocadas conforme mostrado na *Resource Table* da Figura 3.5. Pela WBT, sabe-se que as instruções da 1ª linha escrevem em R1 e R4. Pela *Read Table*, vê-se que as instruções na 2ª linha lêem esses mesmos registradores, o que configura uma dependência de dados do tipo RAW (*Read After Write*). É por isso que essas instruções foram alocadas na segunda linha da UFR: elas dependem do resultado da linha anterior.

Além disso, na segunda linha, os registradores R3 e R5 são escritos, segundo a WBT. Isso também ocorre na terceira linha, o que consiste numa dependência WAW (*Write After Write*). Como duas instruções não podem escrever ao mesmo tempo num mesmo registrador, as últimas instruções que escrevem em R3 e R5 foram alocadas na terceira linha.

A Figura 3.7 ilustra a *Context Table*, ou seja, as tabelas *Current Table*, *Start Table* e *Immediate Table*. Neste exemplo, a tabela de contexto tem espaço para 16 registradores e 8 valores imediatos. O registrador R4 não tem seu bit ativo na *Start Table* porque seu valor é gerado pela unidade funcional da UFR, não sendo necessária sua leitura do banco de registradores do processador.

A tabela que completa o conjunto de tabelas é a *Write Table*, mostrada na Figura 3.8. Cada coluna corresponde a um *slot* (posição) da tabela *Current Table* (Figura 3.7). A *Write Table* está de acordo com a Figura 3.5, que mostra que as duas primeiras linhas

tem apenas duas instruções, enquanto que a terceira linha da UFR tem 3 instruções alocadas. Esta tabela também está de acordo com a WBT da Figura 3.4: na primeira linha, são escritos R1 e R4 e, na terceira linha, são escritos R1, R3 e R5, por exemplo.

Current Table

R2	R3	R1	R5	R4	R6	R7	R8	--	--	--	--	--	--	--	--

Start Table

1	1	1	1	0	1	1	1	0	0	0	0	0	0	0	0

Immediate Table

42	-1	9	7	--	--	--	--

Figura 3.7: Exemplo de *Context Table*

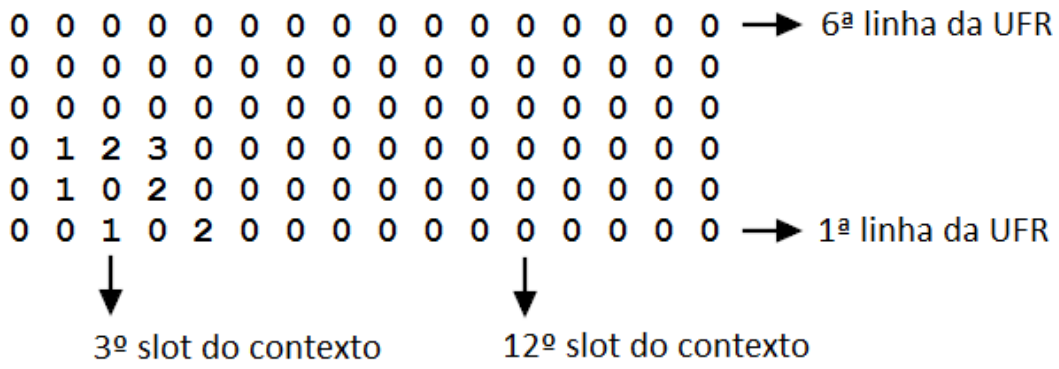

Figura 3.8: Exemplo de *Write Table*

Com os exemplos apresentados, fica claro que a tabela WBT tem importância central na determinação de dependências de dados dos tipos RAW e WAW entre instruções.

A descrição do sistema em (BECK, 2008) sugere, embora não descreva em detalhes, um buffer circular para implementar o suporte à verificação de dependências falsas (WAR e WAW). No presente trabalho, as dependências WAW são verificadas com a própria WBT, conforme descrição e exemplos acima. A dependência WAR é verificada com o uso de uma tabela adicional, descrita a seguir:

- *Read Bitmap Table* (RBT): estrutura semelhante à WBT. Cada bit indica se o registrador correspondente é lido em alguma operação realizada na respectiva linha da UFR. É necessária para a verificação de dependências WAR.

As tabelas que são armazenadas na palavra de configuração são a *Context Table*, a *Read Table*, a *Write Table* e a *Function Table*, pois controlam diretamente a UFR. As demais tabelas são usadas para manter o estado do TB, ou seja, são utilizadas apenas na Fase de Detecção.

3.3.2 Algoritmo de tradução

O algoritmo de tradução apresentado a seguir é menos detalhado que o algoritmo em (BECK, 2008), pois tem por finalidade apenas a enumeração das "tarefas" que devem ser realizadas pelo TB com cada instrução fornecida na entrada. Além disso, diferentemente do algoritmo no trabalho mencionado, este algoritmo não pressupõe qualquer organização específica do módulo de tradução e, justamente por isso, não restringe a implementação do módulo. Este algoritmo serve como base para o entendimento da operação do tradutor.

Observe-se o fato de que a seqüência dos passos no algoritmo a seguir não é obrigatória. Por exemplo, o passo 5, *Atualização da Context Table*, poderia tranquilamente estar imediatamente após a *Decodificação* (passo 1).

Algoritmo de Tradução Binária

Entrada: instrução
Saída: bits de configuração

1. **Decodificação**

   Decodificar a instrução, separando seus campos. Determinar o grupo da instrução. Indicar instruções especiais (lui, store). Indicar se há uso de operando imediato.

   Se a instrução for inválida (desvio ou instrução não-suportada pela UFR), a configuração deve ser fechada. Ver passo 7.

2. **Análise de dependências de dados** (escolha da linha)

   Escolher uma linha da UFR na qual não haja violação de dependências de dados (RAW, WAW ou WAR). Utilizar, para isso, WBT e RBT.

   Se não houver linha livre de violação de dependências, a configuração deve ser fechada. Ver passo 7.

3. **Determinação de recurso livre** (escolha da coluna)

   Na linha escolhida no passo 2, escolher a primeira coluna livre da UFR, se houver. Se não houver, escolher a primeira coluna livre da linha seguinte, se houver. E assim por diante. Utilizar, para isso, Resource Table.

   Se não houver recurso livre, a configuração deve ser fechada. Ver passo 7.

4. **Atualização das tabelas de bitmap**

   Atualizar WBT, RBT e Resource Table na posição determinada nos passos 2 e 3.

5. **Atualização da Context Table**

   Se o registrador ainda não estiver na tabela, adicioná-lo à Current Table. Se o registrador for de leitura e ainda não estiver na Current Table, adicioná-lo à Start Table. Se o operando imediato ainda não estiver na Context Table, adicioná-lo.

   Se houver estouro em qualquer uma das tabelas, a configuração deve ser fechada. Ver passo 7.

6. **Atualização das tabelas da Fase de Reconfiguração**

   Atualizar Function Table, Read Table e Write Table na posição determinada nos passos 2 e 3.

7. **Fechamento da configuração**

   A configuração atual deve ser encerrada.

   Se um número mínimo de instruções foi alcançado na configuração fechada, esta deve ser escrita na cache de configurações. Caso contrário, ela é descartada.

3.4 Cache de configurações

A cache de configurações, na implementação atual do sistema, é uma cache completamente associativa, com uma política FIFO de substituição. No entanto, o simulador do sistema DIM desenvolvido em nosso laboratório permite considerar tipos e tamanhos diferentes da memória.

4 A IMPLEMENTAÇÃO

A implementação em VHDL do módulo de Tradução Binária do sistema DIM foi motivada pela possibilidade de extrair, a partir da descrição nessa linguagem, informações precisas de desempenho, área, potência e energia do circuito. A investigação de tais parâmetros é essencial no projeto de sistemas embarcados, pois esses correspondem exatamente às restrições de operação impostas neste tipo de sistema.

Além disso, a descrição em VHDL permite a síntese do circuito em FPGA, o que, por sua vez, permite a prototipação do sistema DIM para executar aplicações reais. As outras partes do sistema, como o processador e a UFR, já estavam disponíveis em VHDL quando do início do presente trabalho.

O processador utilizado foi o miniMIPS (MINIMIPS, 2008), uma implementação aberta em VHDL da arquitetura do processador MIPS R3000, cuja ISA é a MIPS I. No início da implementação do TB, este processador já havia sido modificado para ser acoplado ao resto do sistema (UFR + TB + cache de configurações).

Uma questão fundamental na implementação do tradutor é de que forma transformar as tarefas do algoritmo (seção 3.3.2) num módulo de hardware. A seção a seguir descreve em detalhes como se chegou à organização do módulo.

4.1 Organização do Tradutor Binário

Considerando-se que a operação de tradução binária em questão pode ser modelada em passos discretos, conforme seção 3.3.2, e que ela envolve múltiplas tabelas, é natural pensar numa organização em blocos, na qual blocos distintos são responsáveis por tarefas distintas e possivelmente contêm tabelas distintas.

Observe-se, no algoritmo da seção 3.3.2, que existem dependências entre os passos do algoritmo. A atualização da *Write Table*, por exemplo, depende da escolha de uma posição aonde alocar a instrução. Isso sugere a existência de certa seqüencialidade no algoritmo. Levando-se em conta o fato de que o Tradutor Binário opera em paralelo ao processador principal, tem-se que o TB dispõe de certo número de ciclos para completar seu trabalho, número esse igual ao número de estágios do pipeline do processador. Assim, a organização na forma de um pipeline parece ser natural para o módulo.

Essa é, de fato, a organização proposta na descrição original do TB, em (BECK, 2008). Neste trabalho, a divisão de tarefas entre os estágios é como segue:

- 1º estágio: decodificação da instrução

- 2º estágio: determinação da linha da UFR em que a instrução pode ser alocada sem que haja violação de dependências de dados. Isso é feito com base na *Write Table*.
- 3º estágio: escolha de um recurso (coluna) livre. Isso é feito com base na *Resource Table*.
- 4º estágio: atualização da *Write Bitmap Table*, *Resource Table* e *Function Table*, com base na posição escolhida nos estágios anteriores. Verificação de estouro na *Context Table*.
- 5º estágio: Atualização da *Context Table*. Atualização da *Write Table* e da *Read Table*.

No entanto, esta atribuição de responsabilidades (divisão de tarefas) entre os estágios apresenta uma peculiaridade: não existe restrição quanto à decisão de fechamento de uma configuração. Na divisão acima, o fechamento pode ser determinado no 2º estágio (por esgotamento de linhas da UFR), no 3º (por esgotamento de recursos disponíveis) ou no 4º estágio (por estouro na *Context Table*).

Isso não seria problema se cada estágio fosse puramente combinacional, não armazenando qualquer estado (não contendo registradores internos). Porém, esse não é o caso: os estágios 2, 3, 4 e 5, na proposta original do TB, contêm estado interno (tabelas). A conseqüência disso é a perda de instruções pelo TB em certas situações, o que será demonstrado a seguir.

Um exemplo é mostrado na Figura 4.1. Suponhamos uma organização do TB em pipeline com 4 estágios. Suponhamos que o 1º estágio seja puramente combinacional e que os restantes contenham estado. Na figura, a cada ciclo (t = 1 até t = 7) uma instrução nova chega à entrada do TB (instruções 1 a 7). O estado de cada um dos três últimos estágios consiste nas instruções que estarão na configuração atual (nas tabelas internas, por exemplo) no fim do ciclo. Em t = 6, ocorre algum estouro no 3º estágio, de modo que a configuração {4, 3, 2, 1} tenha de ser encerrada. Ocorre que o estado do 2º estágio já contém as instruções 1 a 4 juntamente com a instrução 5. Isso causa dois problemas. Um deles é que o 3º estágio, em t = 7, pode não ser capaz de separar a instrução 5 das outras instruções nas informações recebidas do 2º estágio para que uma nova configuração seja iniciada. O outro problema é que o 2º estágio não tem como saber que a configuração foi encerrada. Uma alternativa seria o 3º estágio informar o 2º do fechamento da configuração, em t = 7; no entanto, o mesmo problema já mencionado recai sobre o 2º estágio: não é possível separar as instruções 1 a 4 das instruções 5 e 6. Assim, o 2º e o 3º estágios devem descartar seu estado inconsistente em t = 7, fazendo com que as instruções 5 e 6 sejam perdidas pelo TB. A configuração seguinte tem a instrução 7 como instrução inicial.

Esta perda de instruções foi considerada um problema a ser eliminado do tradutor binário. A solução encontrada para este problema foi atribuir a responsabilidade de decidir sobre o fechamento de uma configuração inteiramente ao primeiro estágio que contenha estado (no caso do exemplo apresentado, o 2º). Em outras palavras, toda e qualquer causa de fechamento de uma configuração deve ser conhecida no máximo até o 2º estágio. Assim, não ocorre perda de instruções, como mostrado na Figura 4.2. O fechamento da configuração {4, 3, 2, 1} é determinado no 2º estágio em t = 5. No ciclo seguinte (t = 6), este simplesmente descarta seu estado e adiciona a instrução 5 numa configuração nova. Em t = 7, o 3º estágio não terá problemas em fazer o mesmo, pois as

informações que recebe do 2º estágio já consideram apenas a instrução da nova configuração. O mesmo ocorre com o 4º estágio, que, por fim, escreve a configuração {4, 3, 2, 1} na cache de configurações e adiciona a instrução 5 numa nova configuração. Como se pode ver, nenhuma instrução foi perdida no processo.

		estágios			
		1º	**2º**	**3º**	**4º**
tempo	t = 1	1			
	t = 2	2	1		
	t = 3	3	2, 1	1	
	t = 4	4	3, 2, 1	2, 1	1
	t = 5	5	4, 3, 2, 1	3, 2, 1	2, 1
	t = 6	6	5, 4, 3, 2, 1	**4, 3, 2, 1**	3, 2, 1
	t = 7	7	**???**	**???**	**4, 3, 2, 1**

Figura 4.1: Exemplo de perda de instrução no TB

		estágios			
		1º	**2º**	**3º**	**4º**
tempo	t = 1	1			
	t = 2	2	1		
	t = 3	3	2, 1	1	
	t = 4	4	3, 2, 1	2, 1	1
	t = 5	5	**4, 3, 2, 1**	3, 2, 1	2, 1
	t = 6	6	5	**4, 3, 2, 1**	3, 2, 1
	t = 7	7	6, 5	5	**4, 3, 2, 1**
	t = 8	8	7, 6, 5	6, 5	5
	t = 9	9	8, 7, 6, 5	7, 6, 5	6, 5

Figura 4.2: Solução da perda de instruções

Foi essa a abordagem utilizada na implementação do módulo. O Tradutor Binário é composto por quatro estágios em série (Figura 4.3), formando um pipeline em paralelo ao pipeline do processador. A atribuição de responsabilidades aos estágios, enfim, foi a seguinte:

- 1º estágio (DEC): decodificação das instruções;
    - Passo 1 do algoritmo (seção 3.3.2)
    - Não possui estado (tabelas)

- 2º estágio (VE): verificação de estouro na configuração;
    - Passos 2, 3, 4 e 5 do algoritmo (e parcialmente passo 7)
    - Contém e atualiza as tabelas WBT, RBT, *Resource Table*, *Context Table* e *Function Table*
- 3º estágio (AT): atualização de tabelas da fase de reconfiguração;
    - Passo 6 do algoritmo
    - Contém e atualiza *Write Table* e *Read Table*
- 4º estágio (EC): escrita na cache de configurações.
    - Passo 7 do algoritmo (parcialmente)
    - Contém a palavra de configuração a ser escrita na cache

Essa organização, como já exposto, tem como principal motivação evitar a perda de instruções pelo TB. Além disso, tem-se como objetivo a manutenção do caminho crítico original do processador MIPS R3000, de modo que o TB não interfira na freqüência de operação do sistema.

Nas seções a seguir, cada estágio será descrito com mais detalhes.

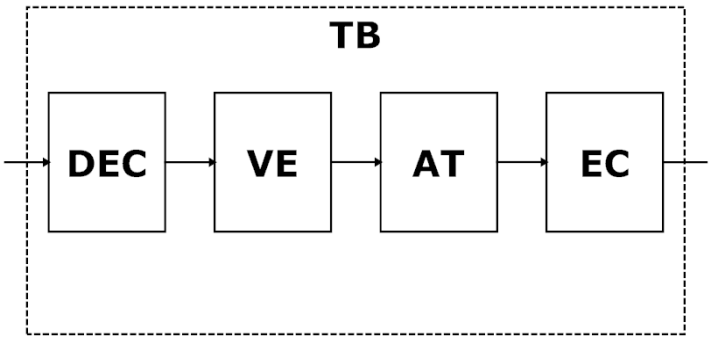

Figura 4.3: Organização do TB. DEC: decodificador, VE: verificação de estouro, AT: atualização de tabelas, EC: escrita na cache

4.2 Primeiro estágio: Decodificação de instruções

O código do decodificador é bastante simples, pois é puramente combinacional, não possuindo estado. Os formatos de instrução utilizados pelo decodificador, a saber, tipos R e I, conforme a especificação da arquitetura MIPS (MIPS, 2001), são mostrados nas Figuras 4.4 e 4.5. A explicação de cada campo consta na Tabela 4.1.

As instruções no formato J, não mostrado aqui, causam a ativação do flag de instrução inválida. Além dessas, todas as instruções de desvio e aquelas não suportadas pela UFR também são consideradas instruções inválidas.

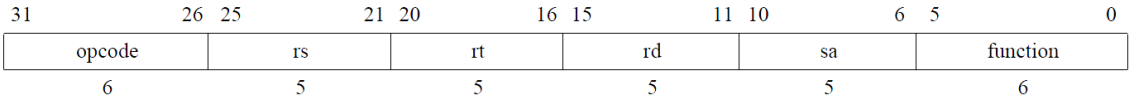

Figura 4.4: Tipo R de instruções (MIPS, 2001)

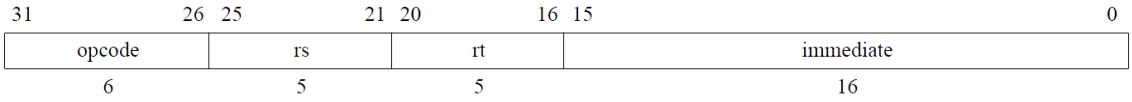

Figura 4.5: Tipo I de instruções (MIPS, 2001)

Além de identificar o formato da instrução e separar seus campos para o estágio seguinte, o decodificador também liga alguns bits de flag indicativos de certas condições especiais:

- **`use_immed`**: indica se a instrução contém um operando imediato, ou seja, se é do Tipo I (Figura 4.5).
- **`instr_lui`**: indica se a instrução é lui. A instrução lui não lê quaisquer registradores; lê apenas o operando imediato e o escreve no registrador destino.
- **`instr_store`**: indica se a instrução é store. Esta instrução não tem registrador destino, apenas registradores de leitura.

Tabela 4.1: Descrição dos campos dos formatos R e I – adaptado de (MIPS, 2001)

Campo	Descrição
opcode	código primário da operação (6 bits)
rd	identificador do registrador de destino (5 bits)
rs	identificador do registrador de origem (*source register*) (5 bits)
rt	identificador do registrador alvo (*target register*) (origem/destino) (5 bits)
immediate	imediato de 16 bits com sinal usado como operando lógico, operando aritmético com sinal, *byte offset* de endereços de load/store e deslocamento relativo ao PC para *branches*
sa	*shift amount* (5 bits)
function	código de 6 bits usado para especificar funções numa instrução com *opcode* primário 000000

4.3 Segundo estágio: Verificação de estouro na configuração

O segundo estágio é o que determina se a configuração atual deve ser fechada ou pode continuar recebendo mais instruções. As condições que determinam o fechamento de uma configuração são as seguintes:

(a) A instrução é "inválida": é um desvio ou não é suportada;

(b) A instrução causaria estouro na Tabela de Contexto, isto é, não há espaço suficiente na configuração para os registradores ou operando imediato da instrução;

(c) Não há como alocar a instrução na configuração atual sem violar dependências de dados;

(d) Não há recursos (unidades funcionais) livres para a instrução na configuração atual.

Assim, este estágio verifica se há espaço para a instrução atual na tabela de contexto e analisa a dependência de dados entre a instrução atual e as instruções prévias na configuração.

O diagrama de estados que descreve a operação do 2º estágio é apresentado na Figura 4.6. Na figura, a entrada *INVÁLIDA*, que determina a transição entre os estados, é fornecida pelo decodificador (conforme a Seção 4.2) e corresponde à condição (a) acima. O estado inicial é ZERA_CFG, no qual as tabelas do 2º estágio (e outros elementos de estado) são zeradas.

Quando da chegada da primeira instrução válida, passa-se para o estado INICIALIZA_CFG, em que as tabelas são atualizadas com a instrução atual e os bits não-modificados das mesmas são zerados. Nas instruções subseqüentes, o estado é VERIFICA_E_ATUALIZA, no qual todas as condições listadas acima são verificadas. Se nenhuma delas for verdadeira, as tabelas internas e as respectivas saídas são simplesmente atualizadas. Se pelo menos uma das condições (b), (c) ou (d) for verdadeira, o que significa que a instrução causaria estouro em alguma tabela, a configuração atual é fechada e a instrução é adicionada numa nova configuração, da mesma forma que ocorre no estado INICIALIZA_CFG: os bits não-atualizados das tabelas são zerados. Dessa forma, se a instrução que causa o fechamento da configuração for uma instrução válida, ela não é perdida, pois logo é adicionada a uma nova configuração.

Em qualquer momento em que uma instrução inválida chegar [condição (a)], o estado volta a ser ZERA_CFG, pois a configuração atual é fechada e não é possível abrir uma configuração nova.

Em qualquer caso, se pelo menos uma das condições de fechamento for verdadeira, um sinal de fechamento é ligado na saída do estágio para indicar ao próximo estágio que a configuração deve ser fechada.

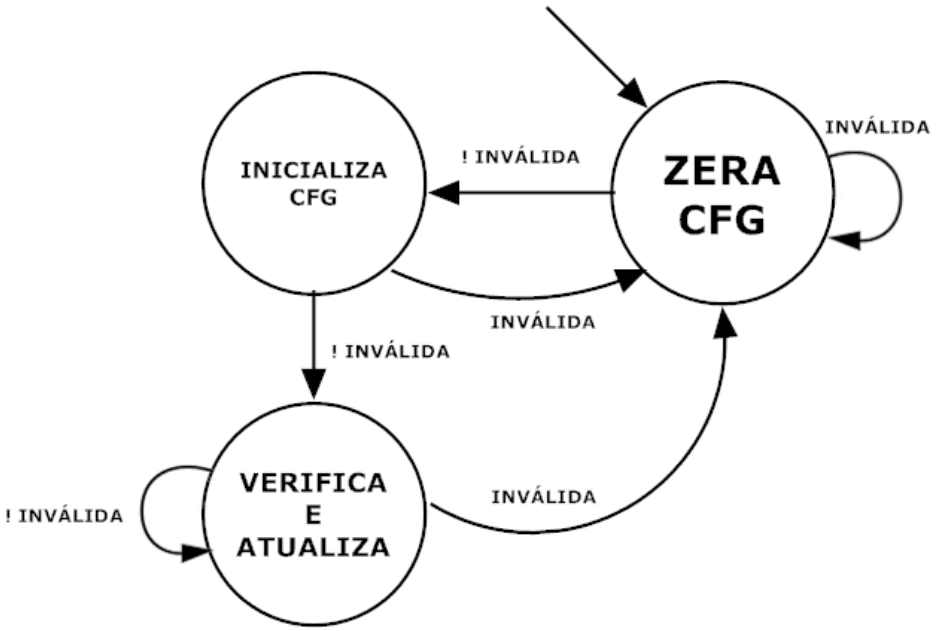

Figura 4.6: Diagrama de estados do 2º estágio do TB

Independentemente do sinal de fechamento, há um sinal na saída que indica que há uma configuração aberta. Ele é ligado nos casos (b), (c) e (d) e é desligado no caso (a). Ou seja, é a negação da entrada *INVÁLIDA* recebida do 1º estágio.

Um aspecto bastante delicado do 2º estágio é a verificação de dependências de dados entre instruções. A questão é delicada por causa dos vários grupos de unidades

funcionais da UFR, cada um com atrasos possivelmente diferentes. Por exemplo: verificou-se que um ciclo de clock do processador principal é equivalente ao atraso de três ULAs em série na UFR. Como o acesso à memória é síncrono, uma instrução de leitura ou escrita na memória (load/store) deve ocorrer entre os limites de um ciclo de clock do processador. Portanto, uma instrução desse tipo pode ser alocada em paralelo a três instruções aritméticas em série na UFR. Isso complica a análise de dependências entre as instruções. Um detalhamento dessa análise é feito a seguir.

4.3.1 Análise de dependências de dados

O objetivo da análise de dependências, no tradutor binário do sistema DIM, é determinar a linha da UFR na qual a instrução atual pode ser alocada de modo que ela não viole qualquer dependência de dados com instruções previamente alocadas na configuração atual. Devem ser considerados os três tipos de dependência de dados (RAW, WAR e WAW) entre instruções. A análise para cada tipo é feita separadamente e, após, é tomado o valor máximo dos números de linha resultantes na análise de cada tipo.

Embora as dependências WAR e WAW possam ser resolvidas com técnicas de renomeação de registradores, o que é feito em processadores superescalares, esta implementação do TB limita-se a detectar as dependências e evitar violações das mesmas por meio da alocação das instruções nas linhas apropriadas. Para uma alocação correta e eficiente, visto que é este o objetivo do TB, tomou-se bastante cuidado na análise de instruções pertencentes a grupos de unidades funcionais que apresentam atrasos diferentes. Na implementação realizada, foram considerados os grupos LD e ULA. A seguir, será descrito em detalhes o processo da análise.

A partir da instrução sendo analisada, são construídos dois bitmaps: um deles corresponde aos registradores que ela lê e o outro, ao registrador em que ela escreve. Por exemplo, num sistema com um processador com 32 registradores, se uma instrução lê os registradores R3 e R8, o bitmap de leitura correspondente será 00000000000000000000000100001000.

Esses bitmaps, juntamente com as tabelas de bitmap WBT e RBT (descritas na subseção 3.3.1), permitem determinar a existência de qualquer tipo de dependência entre a instrução sendo analisada e todas as instruções previamente alocadas na configuração atual. Isso é feito com um and (“e” lógico) entre os bitmaps e cada linha das tabelas. Seja B_R o bitmap de leitura e B_W o bitmap de escrita. A tabela e o bitmap usados dependem do tipo de dependência que está sendo analisado. Essa relação é mostrada na Tabela 4.2.

Tabela 4.2: Relação entre tipos de dependência e bitmaps/tabelas de bitmap utilizados

Tipo de dependência	Bitmap e tabela usados
RAW (Read After Write)	B_R , WBT
WAW (Write After Write)	B_W , WBT
WAR (Write After Read)	B_W , RBT

A linha das tabelas de bitmap que é determinante na análise de dependências é sempre a última linha em que o and é não-nulo, pois é esta linha que define as regiões da UFR em que a instrução viola ou não viola alguma dependência, ou seja, as regiões

em que a instrução pode ou não ser alocada. Nas definições matemáticas a seguir, esta última linha será chamada de *u*.

Considerando apenas a dependência entre instruções do mesmo grupo, a análise seria definida como segue. Seja *tipo_dep* o tipo de dependência sendo analisado, seja *T* a tabela de bitmap sendo avaliada, seja *T*[*i*] a *i*-ésima linha da tabela, e seja *B* o bitmap de leitura/escrita da instrução. Seja u_{tipo_dep} o máximo *i* tal que *T*[*i*] and *B* seja não-nulo, para todo $i \in [0, NUM\_LINHAS - 1]$. A análise é feita conforme mostrado na Tabela 4.3.

Adicionalmente, se *T*[*i*] and *B* é nulo para toda linha *i*, então não há dependência de dados; a instrução pode ser alocada na primeira linha da UFR.

Tabela 4.3: Análise de dependências entre instruções de um mesmo grupo

Tipo de dependência	**Se houver (*T*[*i*] and *B*) não-nulo, então a instrução pode ser alocada a partir da linha...**
RAW (Read After Write)	$u_{raw} + 1$
WAW (Write After Write)	$u_{waw} + 1$
WAR (Write After Read)	u_{war}

A Tabela 4.3 baseia-se no seguinte fato: as dependências RAW e WAW não permitem alocar a instrução em paralelo à instrução causadora da dependência, como pode ser visto na Figura 4.7. Se a instrução ADD lê o registrador R5 e este já está sendo escrito por outra instrução alocada previamente na UFR, a instrução ADD deve ser alocada na linha subseqüente, pois ela depende do resultado da instrução anterior. O caso da dependência WAW é mostrado na Figura 4.8, e é análogo: a instrução atual escreve em R2 e uma instrução já alocada também escreve em R2. Como as duas instruções não podem escrever ao mesmo tempo num mesmo registrador, a instrução ADD deve ser alocada na linha seguinte à instrução causadora da dependência.

A dependência WAR (Figura 4.9), por outro lado, permite a alocação em paralelo, pois o contexto na saída de uma linha da UFR não interfere no contexto de entrada da mesma linha (conforme Figura 3.3). Em outras palavras, uma escrita num registrador pode ser efetuada na saída sem interferir na entrada correspondente ao mesmo registrador. No exemplo da Figura 4.9, o registrador R2 pode ser escrito na mesma linha em que é lido por uma instrução previamente alocada.

Considerando que há grupos de unidades funcionais com atrasos distintos, a situação se complica. As unidades do grupo LD, pelo fato de realizarem operações de memória, devem ser "alinhadas" a ciclos de clock do processador. Portanto, seu atraso é o de um ciclo. Segundo experimentos realizados em nosso laboratório, o atraso de uma ULA é pouco menos de um terço de um ciclo de clock do processador. Dito de outra forma, o atraso de uma única operação de load/store é equivalente ao atraso de várias operações aritméticas (no caso, três) em série.

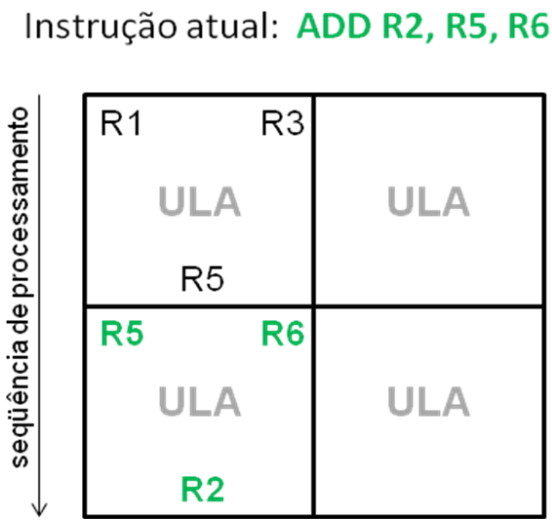

Figura 4.7: Dependência RAW (*Read After Write*)

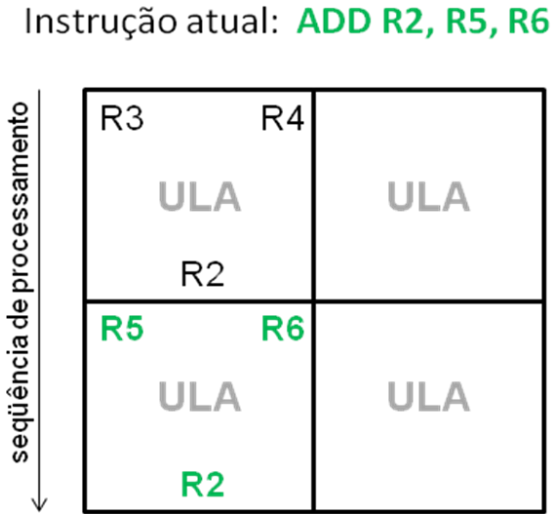

Figura 4.8: Dependência WAW (*Write After Write*)

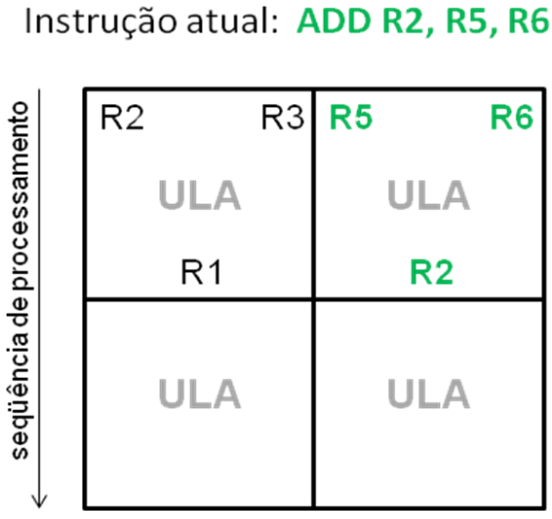

Figura 4.9: Dependência WAR (*Write After Read*)

Tendo isso em mente, para se fazer a análise de dependências, deve-se fazer uma correspondência entre as linhas dos grupos LD e ULA, pois os dois grupos têm um número diferente de linhas na matriz de unidades. Seja *ULAS_POR_LD* o número de ULAs que, em série, têm um atraso equivalente ao de uma unidade do grupo LD. Assim, considerando a numeração das linhas partindo de zero, a relação entre uma linha L_{ULA} (do grupo ULA) e uma linha L_{LD} (do grupo LD) é

$$L_{\text{ULA}} = L_{\text{LD}} \cdot ULAS\_POR\_LD + offset\_ciclo,$$

onde *offset_ciclo* é o deslocamento em relação ao início do ciclo, indo de zero a *ULAS_POR_LD – 1*. A relação inversa, L_{LD} em função de L_{ULA}, pode ser expressa como

$$L_{\text{LD}} = L_{\text{ULA}} \text{ div } ULAS\_POR\_LD,$$

onde div é a divisão inteira. Tendo tais relações, a análise pode ser feita utilizando-se a numeração de linhas mais fina (a de ULAs, no caso). Caso a instrução sendo analisada seja do grupo LD, o número da linha determinado pela análise é, então, convertido apropriadamente para um número de linha do grupo LD. Isso simplifica o processo e evita que se usem tabelas de bitmap duplicadas para cada grupo. Assim, as tabelas (WBT e RBT) simplesmente têm o mesmo número de linhas do grupo de unidades com o menor atraso.

A análise de dependências entre instruções de grupos distintos é descrita informalmente na Tabela 4.4. A descrição mais formal encontra-se na Tabela 4.5. O significado de u_{tipo_dep} mantém-se o mesmo. Nesta tabela, nos casos RAW e WAW para o grupo LD, o efeito da soma com *ULAS_POR_LD* é simplesmente incrementar (em 1) a linha do grupo LD – como já dito, a análise é feita primeiramente com as linhas do grupo com menor atraso (grupo ULA).

O raciocínio por trás da análise de dependências entre grupos com atrasos distintos (Tabelas 4.4 e 4.5) é análogo ao da análise entre instruções de mesmo tipo. Porém, existe uma peculiaridade no caso de dependências WAW, a qual é mostrada na Figura 4.10. Se a instrução atual (LW, na figura) tem um atraso maior do que a instrução causadora da dependência WAW, ela *pode* ser alocada em paralelo a esta, *desde que* esta não esteja alinhada com a saída da instrução atual. Na figura, como a instrução previamente alocada não está no fim do ciclo, a instrução LW escreverá em R2 naturalmente após a instrução anterior, respeitando, assim, a ordem "*write after write*" do programa. Isso não aconteceria se a instrução causadora da dependência estivesse no fim do ciclo, caso em que a instrução LW teria de ser alocada na linha seguinte de unidades de *load*/*store*.

Se não houver linhas livres de violação de dependências, a configuração atual é fechada e a instrução é adicionada a uma nova configuração.

Na Figura 4.11 é apresentado um exemplo de alocação de instruções pelo TB. A parte esquerda da figura consiste numa sequência de instruções recebida pelo TB. À direita, estão representadas as primeiras linhas da UFR. A instrução 2 apresenta dependência verdadeira (RAW) com a instrução 1, por causa da leitura de R1. Por isso, ela é alocada na segunda linha de ULAs. A instrução 3 apresenta dependência falsa (WAR) com a instrução 1, por causa de R3. No entanto, conforme a Tabela 4.4, não há problema em alocá-la em paralelo à instrução 1. Assim, ela é alocada na mesma linha. A instrução 4 não apresenta qualquer dependência de dados com instruções anteriores. A instrução 5, por sua vez, apresenta dependência WAW com a instrução 2 por causa da escrita em R4. Entretanto, conforme a Tabela 4.4, também não há problema em alocar a

instrução de load no primeiro ciclo, visto que a instrução 2 escreve em R4 antes do fim do ciclo.

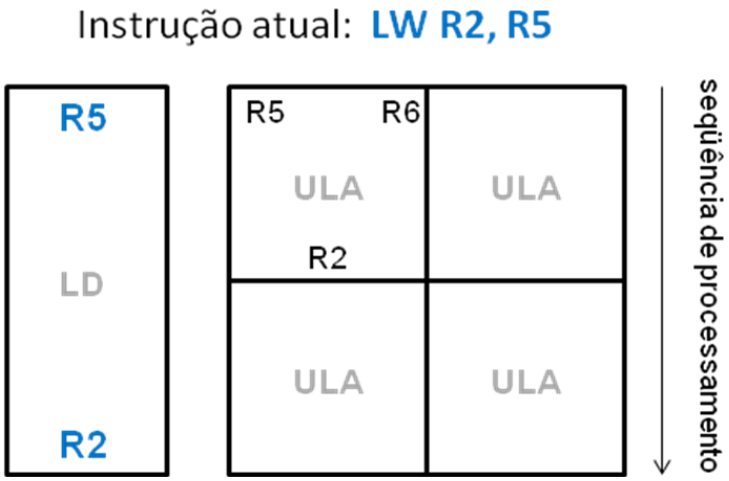

Figura 4.10: WAW entre grupos com atrasos distintos

Tabela 4.4: Análise de dependências entre instruções de grupos distintos (descrição informal)

	Grupo ALU **Exemplo: ADD W, R1, R2**	**Grupo LD** **Exemplo: LD W, R**
RAW	Se um dos registradores sendo lidos R1 ou R2 já estiver sendo escrito por uma instrução load, não pode haver qualquer sobreposição no tempo entre a instrução ADD e a instrução load em que R1 ou R2 estiverem sendo lidos. O ADD deve ser alocado *depois* da instrução de load.	Se o registrador sendo lido R já estiver sendo escrito por outra instrução, não pode haver qualquer sobreposição no tempo entre a instrução load e as instruções em que R estiver sendo escrito – isto é, a instrução de load deve ser alocada completamente após as outras instruções.
WAW	Se o registrador a ser escrito W já estiver sendo escrito por uma instrução de load, não pode haver qualquer sobreposição no tempo entre a instrução ADD e a instrução load em que W estiver sendo escrito. O ADD deve ser alocado *depois* da instrução de load.	Se o registrador a ser escrito W já estiver sendo escrito por outra instrução, não há problema em alocar a instrução load paralelamente à última instrução que escreve em W, *desde que esta não o faça no final de um ciclo*. Em outras palavras, se *ULAS_POR_LD* for *k*, então só será possível alocar o load paralelamente se a última instrução que escreve em W estiver entre as primeiras *k*–1 ULAs do ciclo.
WAR	Se o registrador a ser escrito W já estiver sendo lido por uma instrução de load, não há problema em alocar a instrução ADD em paralelo à instrução de load: o load lê no início do ciclo, sendo que a escrita é feita posteriormente, mesmo se for feita na primeira unidade ALU do ciclo. Assim, a ordem "write after read" do programa será respeitada.	Se o registrador a ser escrito W já estiver sendo lido por outra instrução, não há problema em alocar a instrução load paralelamente à última instrução que lê W, porque a unidade de load só alterará efetivamente o registrador W na saída do ciclo. Assim, a ordem "write after read" do programa será respeitada.

Tabela 4.5: Análise de dependências entre instruções de grupos distintos

	Grupo ALU	**Grupo LD**	
RAW	u_{raw} + 1	u_{raw} + *ULAS_POR_LD*	
WAW	u_{waw} + 1	**se u_{waw} está no fim de um ciclo**	**se u_{waw} não está no fim de um ciclo**
		u_{waw} + *ULAS_POR_LD*	u_{waw}
WAR	u_{war}	u_{war}	

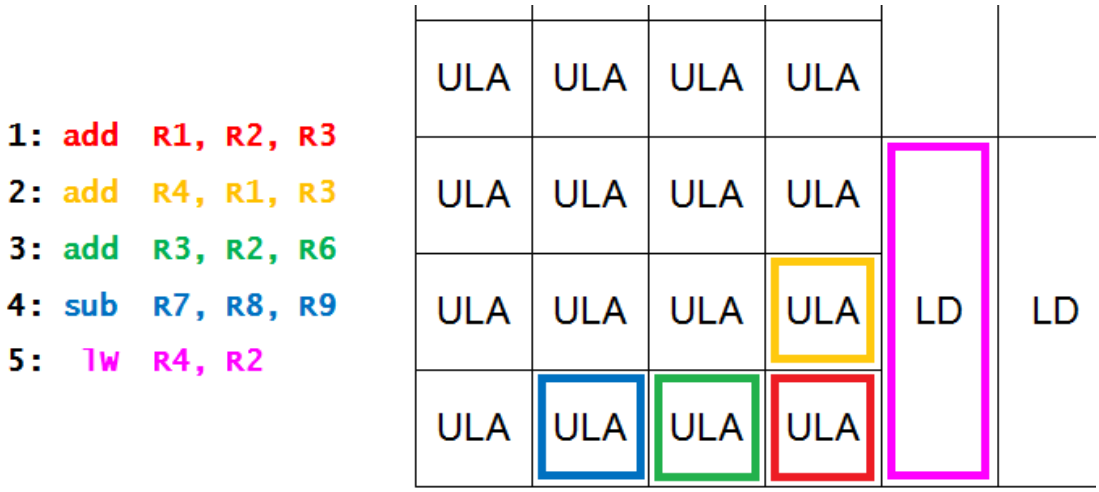

Figura 4.11: Exemplo de alocação de instruções pelo TB

4.4 Terceiro estágio: Atualização de tabelas

O terceiro estágio atualiza a Write Table, a Read Table e a Function Table. Com essas tabelas e com a tabela de contexto vinda do segundo estágio (e mais algumas informações), é possível montar completamente uma palavra de configuração para a UFR.

É cabível, neste momento, uma observação sobre o nome do estágio. O estágio anterior (2º) também realiza atualização de tabelas. No entanto, aquelas servem aos propósitos do próprio TB, ou seja, contêm o estado interno do TB referente à análise das instruções (correspondente à Fase de Detecção). As tabelas do terceiro estágio são as tabelas que de fato compõem uma configuração, na saída do TB, e que são usadas na Fase de Reconfiguração. Daí a importância das tabelas do 3º estágio e a justificativa para o nome dado.

A partir dos sinais de saída do estágio anterior, descritos na seção 4.3, é possível definir a operação do 3º estágio em três casos:

- sinal de fechamento está ativo e não há configuração aberta: instrução inválida. As tabelas devem ser zeradas.
- sinal de fechamento está inativo e há uma configuração aberta: operação normal. As tabelas são atualizadas com a nova instrução.

- sinal de fechamento está ativo e há uma configuração aberta: a configuração anterior foi fechada e uma nova foi aberta. As tabelas internas devem ser zeradas e a nova instrução é adicionada.

O diagrama de estados do 3º estágio, na Figura 4.8, contém exatamente esses três casos (estados). Como essas entradas são repassadas ao 4º estágio, o diagrama de estados deste é o mesmo.

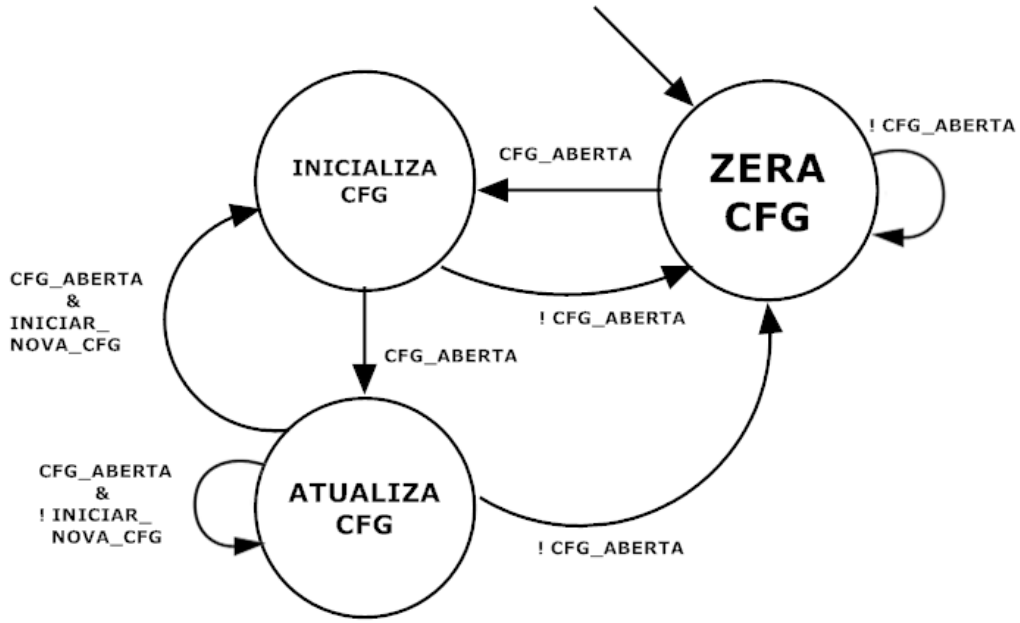

Figura 4.12: Diagrama de estados do 3º e 4º estágios

4.5 Quarto estágio: Escrita na cache de configurações

Este estágio recebe do anterior todas as informações necessárias numa configuração e dispõe-nas no formato apropriado, ou seja, no formato esperado pela UFR. Além disso, este estágio ativa o sinal de escrita na cache de configurações quando uma configuração é fechada. Somente são escritas na cache aquelas configurações que contém certo número mínimo de instruções pré-determinado.

A implementação supõe que a escrita na cache de configurações possa ser feita em apenas um ciclo.

A operação deste estágio também recai nos três casos apresentados na descrição do 3º estágio (Figura 4.8).

5 RESULTADOS

Neste capítulo serão apresentados os resultados da implementação do Tradutor Binário. Inicialmente será descrita a metodologia de obtenção dos resultados. Após, estes serão apresentados na forma de informações de desempenho, potência, energia e área.

A implementação do Tradutor Binário consiste em 3110 linhas de código VHDL. O código encontra-se no Anexo A.

5.1 Metodologia

Os resultados de desempenho foram obtidos através do acoplamento do simulador de precisão de instrução Simics (MAGNUSSON, 2002) e do simulador de precisão de ciclo que emula o sistema DIM. Foram utilizados benchmarks largamente utilizados em sistemas embarcados que estão disponíveis na suite MiBench (GUTHAUS, 2002).

As informações de área e caminho crítico foram obtidas com o software Synopsys Design Compiler. A potência e energia foram determinadas com as ferramentas Cadence NC-Verilog (CADENCE, 2011) e Synopsys Power Compiler (SYNOPSYS, 2011). Em todos os casos, usou-se a biblioteca de células CORE9GPLL, em tecnologia de 90 nm.

Como o TB contém tabelas e estas estão relacionadas à estrutura da UFR (número de linhas, número de unidades por linha, etc.), a área do TB é diretamente ligada ao número de unidades funcionais contidas na UFR. Assim, em alguns experimentos, foram consideradas duas versões do TB, correspondendo a dois tamanhos diferentes da UFR (Tabela 5.1). A versão maior (Versão 2) foi escolhida por representar o tamanho máximo com que se obteve um caminho crítico de até 1,66 ns, que é o caminho crítico do processador MIPS utilizado. Os resultados de potência e energia foram obtidos apenas para esta versão. A versão menor foi escolhida para que seja possível uma avaliação superficial da variação de características do TB em função das variáveis consideradas na Tabela 5.1.

Tabela 5.1: Dimensões do sistema DIM consideradas

	Versão 1	**Versão 2**
Linhas ULA	9	15
Colunas ULA	3	4
Colunas LD	2	2
ULAs por LD (atraso)	3	3

5.2 Resultados

5.2.1 Desempenho

Na Tabela 5.2 são mostrados os resultados de caminho crítico de cada estágio, para as duas versões do TB apresentadas na Tabela 5.1. Percebe-se que o 2º estágio é o estágio com o maior atraso, e conseqüentemente o que limita a freqüência máxima de operação do circuito. Isso se deve à decisão de projeto descrita na seção 4.1, que consiste em não permitir que instruções sejam perdidas pelo TB e, para tanto, atribuir vários passos do algoritmo do TB ao 2º estágio. As freqüências máximas de operação das Versões 1 e 2 são, respectivamente, 892 MHz e 735 MHz. Desta forma, verifica-se que é possível a implementação do módulo de forma que não penalize a freqüência do sistema original, que é de 600 MHz.

Tabela 5.2: Caminhos críticos por estágio

	Versão 1	**Versão 2**
1º estágio (Dec)	0.16 ns	0.16 ns
2º estágio (VE)	1.12 ns	1.36 ns
3º estágio (AT)	0.17 ns	0.19 ns
4º estágio (EC)	0.12 ns	0.13 ns

Os resultados apresentados na Tabela 5.2 foram resultados obtidos sintetizando cada estágio do TB separadamente. Entretanto, sintetizando todos os estágios do TB, obtiveram-se caminhos críticos de 1,32 e 1,49 para as Versões 1 e 2, respectivamente, correspondendo às freqüências máximas 758 MHz e 671 MHz, freqüências que também não afetam o caminho crítico do sistema original. A diferença deve-se à capacidade de otimização da ferramenta utilizada para síntese.

Os dados de desempenho obtidos por meio de simulação são mostrados na Figura 5.1. São mostrados, para cada aplicação, o speedup (aceleração) da execução da aplicação em si (*user*) e do código do sistema operacional (*kernel*). O gráfico mostra que o sistema DIM de fato possibilita a aceleração de aplicações, permitindo um *speedup* de quase 300% para o tamanho da UFR avaliado, no caso do aplicativo *susan_smoothing*.

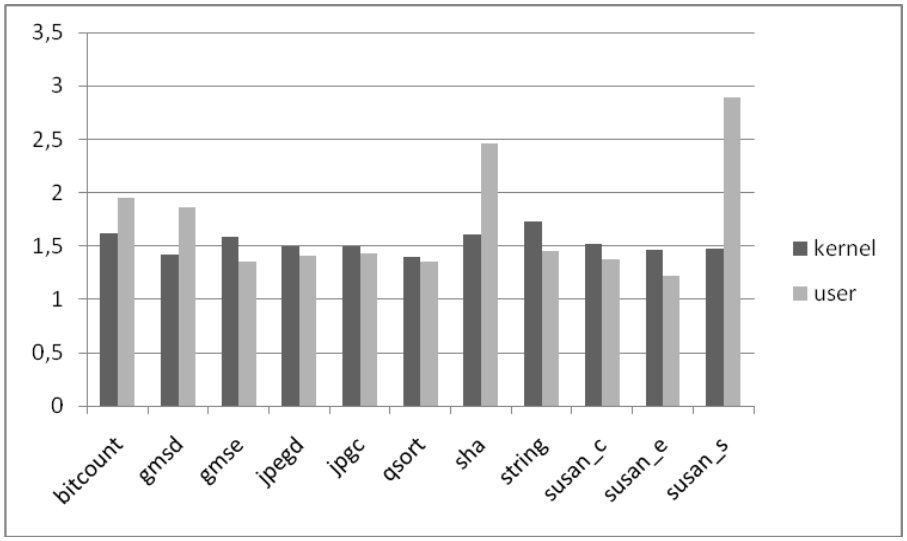

Figura 5.1: Speedup de kernel e de usuário para a Versão 2

5.2.2 Potência

A Figura 5.2 ilustra o consumo de potência de cada componente do sistema reconfigurável considerando a Versão 2 do Tradutor Binário. Assim, pode-se concluir que o TB apresenta um consumo de potência considerável: em torno de 25% do total consumido pelo sistema reconfigurável. Entretanto, como o mesmo somente está ativo, em média, 30% do tempo total de execução das aplicações, pode-se notar que este consumo de potência não implicará proporcionalmente no aumento de energia do sistema.

Este resultado permite afirmar que, em comparação a arquiteturas superescalares, o TB revela-se um mecanismo barato de execução fora de ordem (*out-of-order execution*).

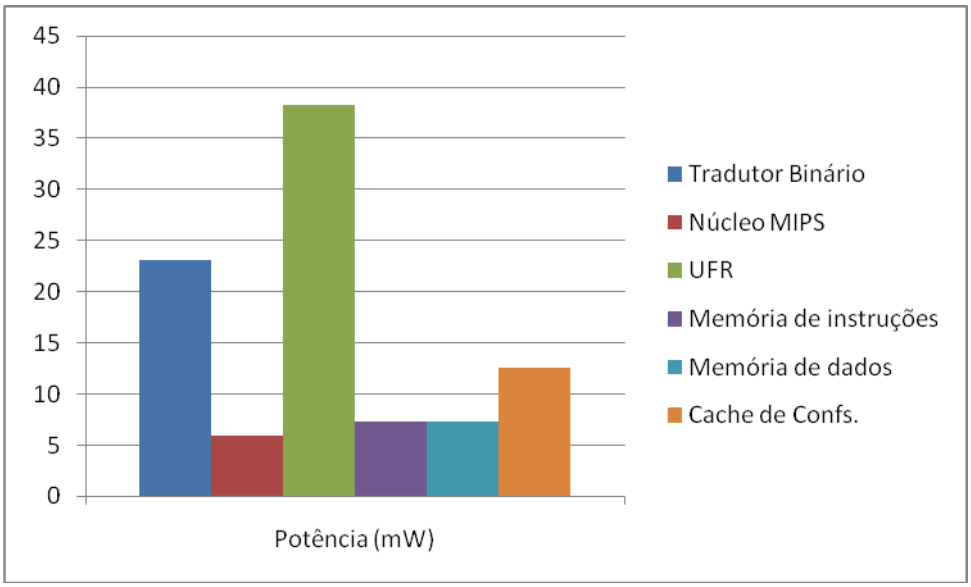

Figura 5.2: Resultados de potência (versão 2)

5.2.3 Energia

Nas Figuras 5.3 e 5.4 são apresentados os gráficos do consumo de energia nos benchmarks considerados. Como explicitado anteriormente, embora o TB tenha uma potência considerável, o mesmo não acarreta um consumo energético marcante, visto que o TB está ativo somente enquanto que o processador principal está executando. Nas fases de Reconfiguração e de Execução (Seção 3.1), o mesmo não realiza nenhum trabalho; portanto, não dissipa potência dinâmica.

Tabela 5.3: Dados de energia (versão 1) (em mJ)

	bitcount	gmsd	gmse	jpegd	jpgc	qsort	sha	string	susan_c	susan_e	susan_s
Tradutor Binário	1.25	1.66	1.44	0.14	0.15	8.48	0.26	0.10	0.70	2.12	1.57
Núcleo MIPS	0.68	0.90	0.78	0.08	0.08	4.60	0.14	0.05	0.38	1.15	0.85
UFR	5.68	9.66	2.26	0.38	0.41	28.99	1.95	0.45	1.39	2.99	5.01
Cache de instruções	0.94	1.68	1.07	0.11	0.12	6.56	0.26	0.09	0.53	1.54	1.15
Cache de dados	0.63	1.63	0.56	0.08	0.08	6.03	0.27	0.08	0.36	0.87	0.83
Cache de Confs.	1.05	1.39	1.20	0.12	0.13	7.12	0.22	0.08	0.59	1.78	1.32
Energia Total	10.23	16.92	7.31	0.91	0.97	61.78	3.1	0.85	3.95	10.45	10.73

Tabela 5.4: Dados de energia (versão 2) (em mJ)

	bitcount	gmsd	gmse	jpegd	jpgc	qsort	sha	string	susan_c	susan_e	susan_s
Tradutor Binário	2.65	3.52	3.05	0.29	0.32	18.03	0.56	0.21	1.49	4.49	3.34
Núcleo MIPS	0.68	0.90	0.78	0.08	0.08	4.60	0.14	0.05	0.38	1.15	0.85
UFR	12.37	21.04	4.92	0.82	0.90	63.16	4.25	0.98	3.04	6.52	10.91
Cache de instruções	0.88	1.17	1.02	0.10	0.11	6.00	0.19	0.07	0.50	1.50	1.11
Cache de dados	0.63	1.17	0.56	0.08	0.08	6.03	0.27	0.08	0.36	0.87	0.83
Cache de Confs.	1.45	1.93	1.67	0.16	0.18	9.86	0.30	0.12	0.81	2.46	1.83
Energia Total	18.66	29.73	12	1.53	1.67	107.68	5.71	1.51	6.58	16.99	18.87

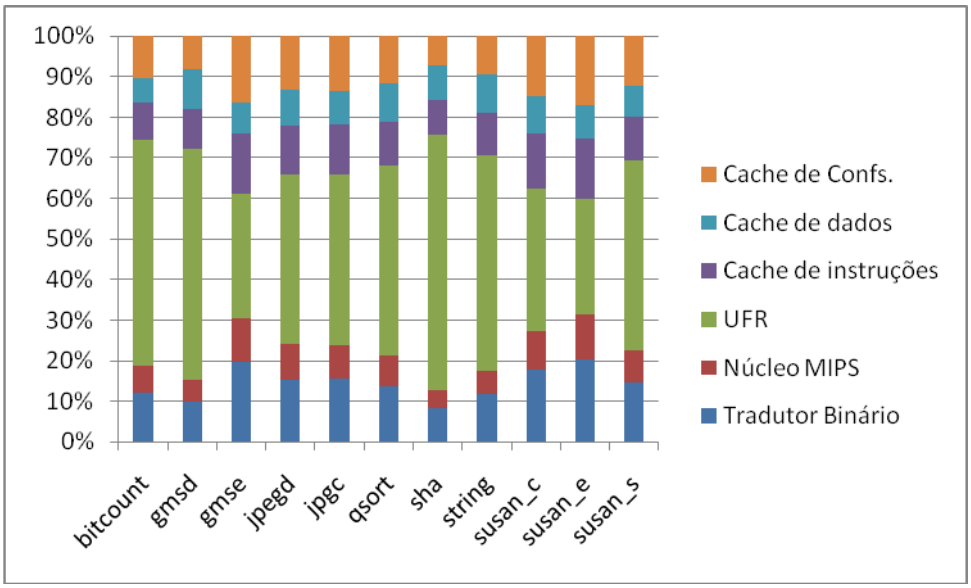

Figura 5.5: Resultados de energia (versão 1)

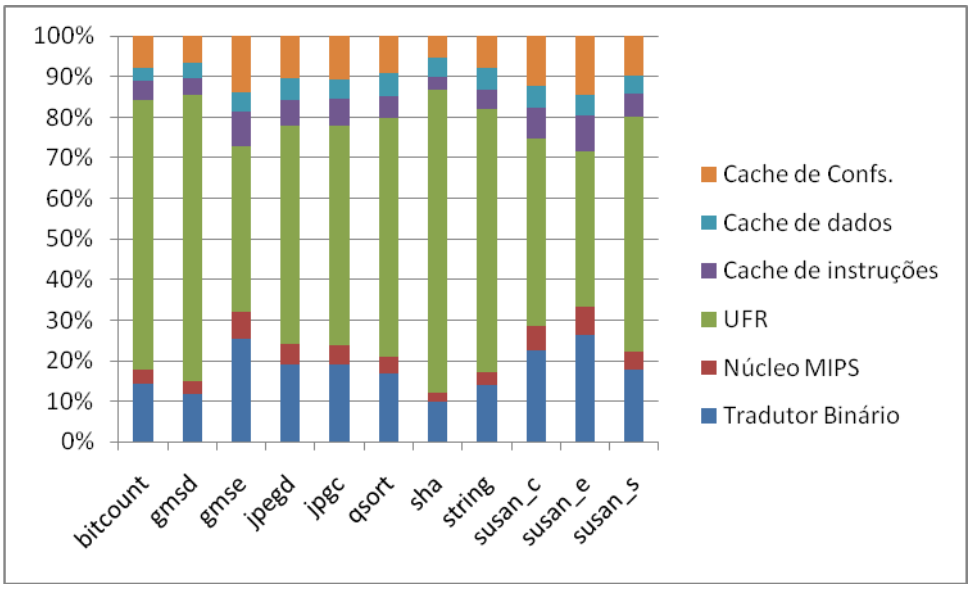

Figura 5.6: Resultados de energia (versão 2)

Na Tabela 5.5 e na Figura 5.5 são mostrados os dados de energia apenas do processador MIPS, sem a UFR, o TB e a cache de configurações acoplados. Percebe-se que o sistema acoplado à Versão 2 da UFR apresenta consumo de energia

aproximadamente equivalente ao do MIPS sozinho, ao mesmo tempo em que apresenta um melhor tempo de execução, conforme Figura 5.1.

Tabela 5.5: Dados de energia – processador MIPS (em mJ)

	bitcount	gmsd	gmse	jpegd	jpgc	qsort	sha	string	susan_c	susan_e	susan_s
Núcleo MIPS	2.75	4.74	1.66	0.22	0.24	15.46	0.90	0.23	0.92	2.30	2.68
Cache de instrução	3.41	5.89	2.06	0.27	0.30	19.20	1.11	0.28	1.13	2.85	3.33
Cache de dados	0.63	1.63	0.56	0.08	0.08	6.03	0.27	0.08	0.36	0.87	0.83
Energia Total	6.79	12.26	4.28	0.57	0.62	40.69	2.28	0.59	2.41	6.02	6.84

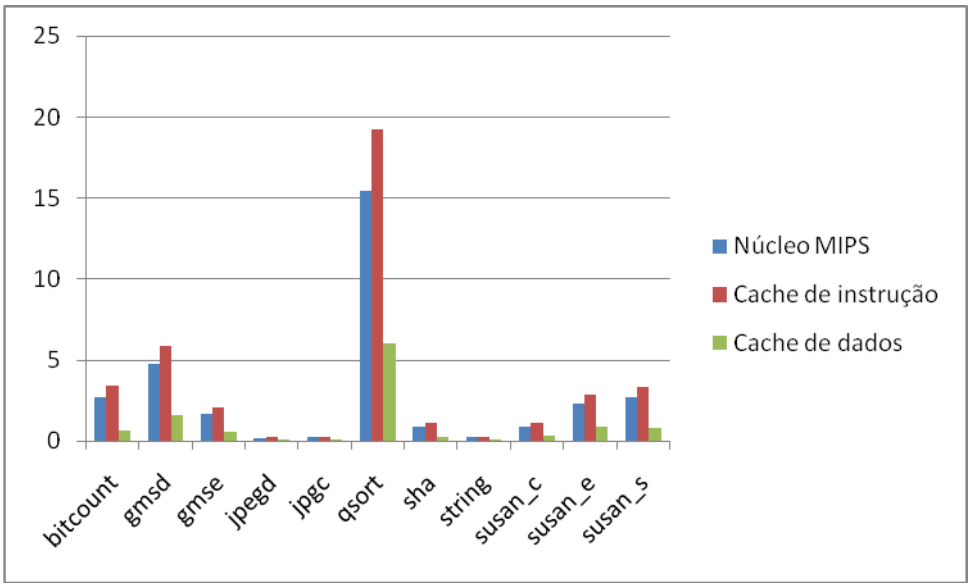

Figura 5.7: Resultados de energia – processador MIPS sozinho

5.2.4 Área

Os resultados de área são apresentados na Tabela 5.6, novamente para os dois tamanhos do sistema e para cada estágio. Esses dados foram obtidos sintetizando-se cada estágio separadamente. Sintetizando o TB inteiro, obtiveram-se as áreas 97.118 e 201.691 μm^2 para as versões 1 e 2, respectivamente, que são áreas menores do que as áreas totais indicadas na tabela. Essa diferença provavelmente se deve à capacidade da ferramenta Design Compiler de realizar otimizações mais pontuais no desempenho dos módulos individuais, aumentando, assim, sua área.

Tabela 5.6: Área do TB por estágio (em μm^2)

	Versão 1	**Versão 2**
1º estágio (Dec)	4.128	4.128
2º estágio (VE)	58.557	94.308
3º estágio (AT)	43.994	94.098
4º estágio (EC)	22.029	50.318
Total	128.708	242.852

Na Tabela 5.7 encontram-se as áreas dos demais blocos do sistema, para fins de comparação. As Figuras 5.6 e 5.7 são os respectivos gráficos, um para cada tamanho da UFR considerado. Nos gráficos, a área da UFR não foi mostrada, pois é muito maior que a dos outros componentes. Pode-se observar que a área do TB é comparável, nos

dois casos considerados, à área do núcleo do processador MIPS. Percebe-se também que o crescimento da área do TB acompanha, de modo geral, o crescimento da área da UFR e da cache de configurações, visto que o mesmo lida com tabelas de tamanho proporcionalao tamanho do sistema.

Tabela 5.7: Área dos componentes do sistema DIM (em μm^2)

	Versão 1	**Versão 2**
Tradutor Binário	97.118	201.692
Núcleo MIPS	137.563	137.563
UFR	5.322.969	10.467.765
Cache de instruções	130.980	130.980
Cache de dados	130.980	130.980
Cache de Configurações	743.988	1.305.455
Área Total (um2)	6.563.598	12.374.435

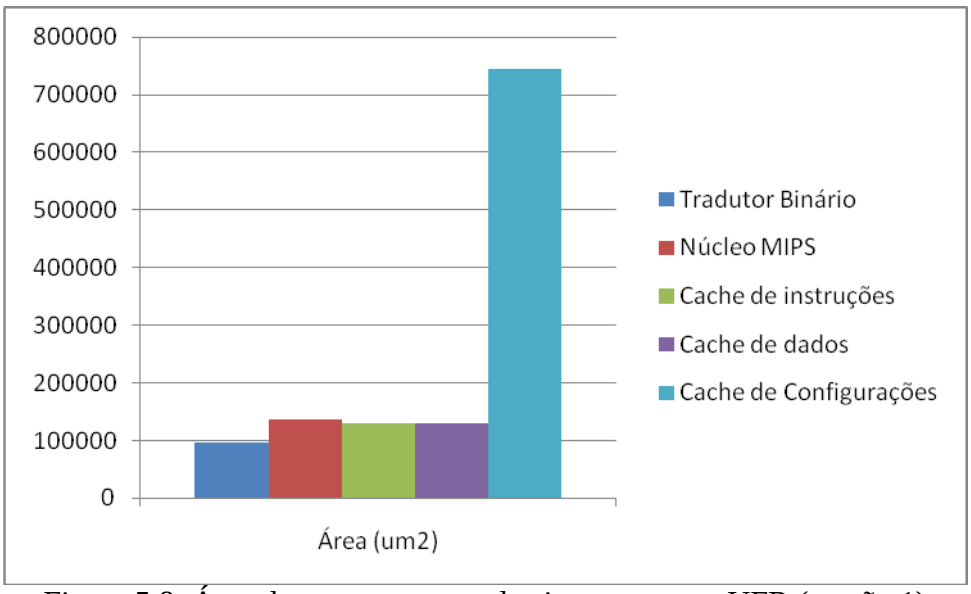

Figura 5.8: Área dos componentes do sistema exceto UFR (versão 1)

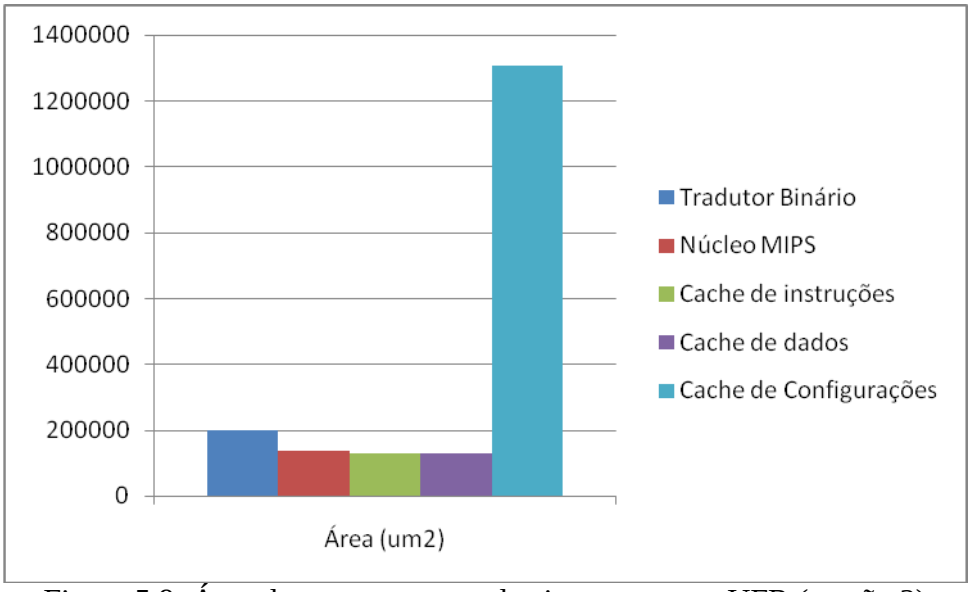

Figura 5.9: Área dos componentes do sistema exceto UFR (versão 2)

6 CONCLUSÃO E TRABALHOS FUTUROS

Os resultados apresentados no Capítulo 5 permitem afirmar que é possível a implementação do módulo de Tradução Binária do sistema DIM com desempenho satisfatório para a síntese do sistema com o processador miniMIPS. No entanto, existe espaço para melhorias no desempenho, conforme evidenciado pelos caminhos críticos individuais dos estágios.

A seção 6.1 aborda a questão do caminho crítico. A seção 6.2 apresenta a problemática do suporte à multiplicação no TB e uma proposta de solução.

6.1 Caminho crítico

Uma das decisões que guiaram a implementação do TB foi evitar a perda de instruções pelo módulo, o que era visto como algo que devia ser evitado. Com isso, o 2º estágio ficou sobrecarregado, ficando relegada somente a ele a responsabilidade de decidir sobre o fechamento de uma configuração. Conseqüência disso é o fato de que o 2º estágio apresenta um atraso muito maior que os demais estágios do TB. Assim, conclui-se, ao final deste trabalho, que a eventual perda de algumas instruções pode não representar um problema quando confrontada com uma possibilidade de ganho significativo em freqüência máxima de operação. Tal possibilidade existe, caso o 2º estágio seja simplificado.

Uma medida possível que tem como base a organização atual é passar a atualização da tabela de contexto (Context Table) para o 1º estágio. No entanto, isso exige um mecanismo de *rollback* no 2º estágio, para o caso de este determinar o fechamento da configuração. Nesse caso, a tabela de contexto recebida do 1º estágio já conteria os registradores/imediatos da instrução causadora do estouro, e esta deveria ser retirada da tabela. Isso, por sua vez, pode causar um overhead, no 2º estágio, que aumente novamente seu caminho crítico.

Outra alteração paliativa na organização atual seria a transferência da atualização da Function Table do 2º para o 3º estágio. No entanto, somente isso provavelmente não causaria mudança significativa no caminho crítico do módulo. O que realmente pode reduzir o caminho crítico é redistribuir as responsabilidades dos estágios, de modo que não haja restrição quanto a qual deles pode fechar uma configuração. Isso implica o redesenho das máquinas de estados de cada estágio, visto que o fechamento de uma configuração torna-se mais complexo, conforme discutido na seção 4.1.

6.2 Suporte à multiplicação no TB

Além da diminuição do caminho crítico, um dos trabalhos futuros em relação ao módulo do TB é a implementação do suporte à multiplicação. A operação de multiplicação, na arquitetura MIPS, é composta não apenas por uma instrução, mas por duas ou três: mult, mfhi (*move from HI*) e mflo (*move from LO*). A instrução mult realiza de fato a multiplicação, armazenando o resultado nos registradores especiais HI e LO. Para recuperar o resultado, é necessário haver pelo menos uma das duas outras instruções após a instrução mult, as quais movem o conteúdo desses registradores para algum outro registrador especificado.

A implementação do suporte à multiplicação no TB não é trivial. Em primeiro lugar, a instrução mult não tem, em sua codificação, registrador destino. Nesse caso, como fazer a análise de dependências? A análise de dependência RAW pode ser feita de imediato, mas a análise de dependências WAR e WAW, não. A instrução, claro, tem dois registradores destino implícitos, que chegarão com as instruções mfhi e mflo, mas estes não são conhecidos de imediato.

Além disso, não há garantias sobre a adjacência das instruções mult e mfhi ou mflo no código do programa, ou seja, pode haver instruções entre mult e mfhi/lo, ou até mesmo entre mfhi e mflo. Ambas as instruções não são necessárias: em muitos casos, apenas uma delas é suficiente.

Uma possível solução para o suporte à multiplicação é como segue. Uma instrução mult causaria, no TB, a adição dos registradores LO e HI na tabela de contexto da configuração. Quando alguma das instruções mfhi/lo chegasse, a instrução seria transformada num add do registrador HI/LO (já presente no contexto da configuração atual) com o valor zero. Esta soma seria escrita no registrador de destino especificado pelo mfhi/lo. Dessa forma, uma ULA seria usada para redirecionar o resultado da multiplicação aos registradores apropriados.

Caso uma configuração fosse encerrada com uma multiplicação pendente (ou seja, antes de um mfhi/lo chegar), não haveria problema. Na configuração seguinte, quando alguma de tais instruções chegasse, os registradores HI/LO também seriam adicionados ao contexto da configuração, incluindo, nesse caso, a adição à Start Table, para que esses registradores sejam lidos do processador no início da execução na UFR. Desse modo, a instrução mult e as instruções de recuperação do resultado não precisariam estar numa mesma configuração, o que simplifica muito o controle de fechamento de configurações por parte do TB.

Em qualquer caso, a análise de dependências ocorre normalmente, visto que os registradores HI e LO são tratados como registradores comuns.

Observe-se que a solução aqui descrita implica que a UFR tenha acesso aos registradores HI/LO, pois tanto no início quanto ao final da execução de uma configuração com multiplicação na UFR, os registradores HI/LO devem poder ser lidos do e escritos no processador. Portanto, não basta modificar o TB; é necessário modificar também a interface UFR/processador.

REFERÊNCIAS

Monografia no todo

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture: A Quantitative Approach**. 3rd ed. Amsterdam: Morgan Kaufmann, 2003.

Dissertações, teses, trabalhos individuais, etc.

BECK FILHO, A.C.S. **Transparent Reconfigurable Architecture for Heterogeneous Applications**. 2008. 188 f. Tese (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

RUTZIG, M. B. **A Transparent and Energy Aware Reconfigurable Multiprocessor Platform for Efficient ILP and TLP Exploitation**. 2011. 79 f. Proposta de Tese (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

Artigo de periódico

AUSTIN, T. et al. Mobile Supercomputers. **Computer**, Los Alamitos, v.37, n.5, p. 81-83, maio 2004.

COMPTON, K.; HAUCK, S. Reconfigurable Computing: A Survey of Systems and Software. **ACM Computing Surveys**, New York, v.34, n.2, p. 171-210, junho 2002.

GUTHAUS, M. R. et al. MiBench: A free, commercially representative embedded benchmark suite. **IEEE International Workshop on Workload Characterization**. [S.l.]: [s.n.]. 2002.

HAUCK, S. et al. The Chimaera reconfigurable functional unit. In: FPGA-BASED CUSTOM COMPUTING MACHINES, 1997, Napa Valley. **Proceedings…** Washington: IEEE Computer Society, 1997. p. 87 – 96.

HUTCHINGS, B. L. Exploiting reconfigurability through domain-specific systems. In: INTERNATIONAL WORKSHOP ON FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS, FPL, 7., 1997, London. **Proceedings…** Berlin: Springer, 1997. p.193 – 202. (Lecture Notes in Computer Science, v.1304).

HUTCHINGS, B. L. et al. A CAD suite for high-performance FPGA design. In: SYMPOSIUM ON FIELD-PROGRAMMABLE CUSTOM COMPUTING MACHINES, 1999, Napa Valley. **Proceedings…** Washington: IEEE Computer Society, 1999. p. 12–24.

LYSECKY, R.; STITT, G.; VAHID, F. Warp Processors. **ACM Transactions on Design Automation of Electronic Systems**, New York, v.11, n. 3, p. 659 – 681, julho de 2006.

MAGNUSSON, P. S. et al. Simics: A Full System Simulation Platform. **Computer**, Los Alamitos, v.35, p. 50-58, ACM, 2002.

RUTZIG, M. B.; BECK FILHO, A. C. S., CARRO, L. CReAMS: An Embedded Multiprocessor Platform. **ARC 2011**, p. 118-124.

SANKARALINGAM, K. et al. Trips: A polymorphous architecture for exploiting ilp, tlp, and dlp. **ACM Transactions on Architecture and Code Optimization**, New York, v.1, n. 1, p. 62 – 93, maio de 2004.

SWANSON, S. et al, The WaveScalar Architecture. **ACM Transactions on Computer Systems**, New York, v.25, n. 2, maio de 2007.

Em meio eletrônico

CADENCE Design Systems Homepage. **NC-Verilog Datasheet**. Disponível em: <http://w2.cadence.com/datasheets/IncisiveVerilog_ds.pdf>. Acessado em maio de 2011.

MINIMIPS VHDL. Disponível em: <http://www.opencores.org>. Acessado em julho de 2008.

MIPS Technologies, Inc. **MIPS32 Architecture for Programmers Volume 1: Introduction to the MIPS32 Architecture**. Mountain View, 2001, p. 37. Disponível em <http://www.cs.cornell.edu/courses/cs3410/2008fa/mips_vol1.pdf>. Acessado em maio de 2011.

SYNOPSYS Power Compiler Homepage. Disponível em: <http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Pages/PowerCompiler.aspx>. Acessado em maio de 2011.

ANEXO A – CÓDIGO VHDL DO TRADUTOR BINÁRIO

Em mídia digital.

ANEXO B – TRABALHO DE GRADUAÇÃO I

Implementação de um módulo de Tradução Binária para uma Arquitetura Reconfigurável

Alexis Anton Lazzarotto

Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brasil

aalazzarotto@inf.ufrgs.br

Abstract. *The complexity of embedded systems is growing due to the aggregation of multiple functionalities into a single electronic device. Moreover, acceleration of the execution of superscalar processors is stagnated, and the extraction of parallelism in the Von Neumann model is reaching its theoretical limit. Reconfigurable architectures show up as a viable solution to these problems, and their implementation is feasible in current CMOS technology. The hereby proposed work consists of the implementation of a Binary Translation hardware module for a reconfigurable architecture developed by the Embedded Systems Research Group at the Federal University of Rio Grande do Sul (UFRGS).*

Resumo. *A complexidade dos sistemas embarcados está crescendo devido à agregação de funcionalidades em um único dispositivo eletrônico. Além disso, a aceleração da execução dos processadores superescalares está estagnada, e a extração de paralelismo no modelo Von Neumann está chegando ao limite teórico. Arquiteturas Reconfiguráveis aparecem como uma solução viável para estes problemas, sendo factível a implementação deste tipo de arquitetura nas atuais tecnologias CMOS. O trabalho aqui proposto consiste na implementação em hardware de um módulo de Tradução Binária para a arquitetura reconfigurável desenvolvida no Laboratório de Sistemas Embarcados do Instituto de Informática da UFRGS.*

1. Contextualização

Embora o número de sistemas embarcados esteja aumentando, observa-se uma nova tendência: dispositivos multifuncionais, que executam um amplo espectro de aplicações com comportamentos diversos – por exemplo, telefones portáteis modernos ou PDAs. Conseqüentemente, simples processadores não são mais suficientes para lidar com os requisitos computacionais dos novos sistemas, forçando projetistas a criar novas soluções para aumentar seu desempenho, mantendo a dissipação de energia tão baixa quanto possível.

Soluções como processadores superescalares estão disponíveis no mercado. A exploração do paralelismo em tempo de execução, em nível de instruções (ILP – *instruction level parallelism*), realizada por esta abordagem fornece um alto índice de aceleração na execução de aplicações. No entanto, a complexidade e o custo de área e potência do hardware de exploração de ILP destes processadores, juntamente com a diversidade do grau de paralelismo das aplicações [Xu 1999; Wall 1991], formam um cenário não adequado para a utilização dos mesmos nos futuros dispositivos embarcados.

Os sistemas embarcados têm demandado um poder de processamento cada vez maior. Como já mencionado, embora o desempenho de processadores superescalares seja alto, eles não são adequados para o uso em sistemas embarcados. No entanto, além dessa limitação, a aceleração da execução dos processadores superescalares está estagnada, e a extração de paralelismo no modelo Von Neumann está chegando ao limite teórico, como se pode ver na Figura 1. Atualmente, os projetistas de processadores estão buscando outro paradigma de computação para ser empregado neste tipo de dispositivo.

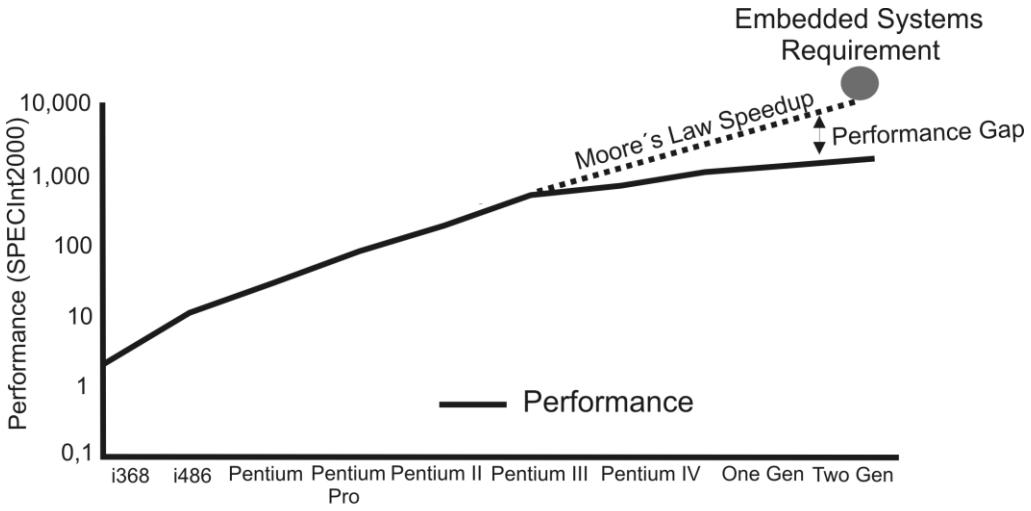

Figura 1. Requisito de processamento para os futuros sistemas embarcados [Austin 2004]

Arquiteturas reconfiguráveis têm se mostrado uma solução viável para esses problemas. Vários estudos já foram feitos explorando arquiteturas reconfiguráveis com vista à aceleração de execução de aplicações, e seus resultados são promissores. Entretanto, o uso desse tipo de arquitetura, em geral, exige compiladores e ferramentas especiais, quebrando a compatibilidade de software com arquiteturas distintas. Além disso, tais arquiteturas apresentam um aumento de desempenho apenas em blocos de código orientados a fluxo de dados (modelo *dataflow*). Portanto, mesmo o uso de arquiteturas reconfiguráveis apresenta desafios para os problemas de compatibilidade e de desempenho dos sistemas embarcados atuais.

Várias pesquisas já foram feitas nas áreas de arquiteturas reconfiguráveis e otimização dinâmica de execução com o objetivo de solucionar os problemas discutidos. Alguns destes trabalhos serão apresentados na seção 2. O trabalho aqui proposto se

baseia em dois trabalhos desenvolvidos no Laboratório de Sistemas Embarcados do Instituto de Informática da UFRGS, os quais serão descritos na seção 3.

2. Estado da arte

Vários trabalhos já foram propostos explorando arquiteturas reconfiguráveis com vista à aceleração de execução de aplicações. Esses trabalhos demonstraram ganhos significativos nesse sentido [Gupta 1993; Gajski 1998; Henkel 1997; Venkataramani 2001]. Além disso, pela redução do tempo de execução das aplicações, estes trabalhos mostram-se eficientes do ponto de vista energético em relação ao sistema original [Henkel 1999; Wan 1998; Stitt 2002].

Na literatura, ainda não existe um consenso na classificação de arquiteturas reconfiguráveis. Aqui será apresentada uma classificação baseada em [Compton 2002]. Basicamente, neste trabalho as arquiteturas reconfiguráveis são distinguidas por três aspectos: acoplamento, granularidade e mecanismo de reconfiguração.

2.1. Acoplamento

Em um projeto de uma arquitetura reconfigurável, a escolha do tipo de acoplamento entre a unidade funcional reconfigurável (UFR) e o processador de propósito geral (PPG) se reflete diretamente na eficiência do sistema. Uma UFR que é implementada como uma unidade funcional dentro do processador é chamada de *fortemente acoplada*. A comunicação entre o PPG e a UFR ocorre somente dentro do núcleo, resultando em um alto desempenho. Outra forma de modelar a UFR é como um co-processador do PPG. Normalmente a escolha deste tipo de acoplamento esta ligada à área disponível de silício dentro do núcleo. Quando não existe área suficiente para armazenar a UFR dentro do núcleo, este tipo de acoplamento é utilizado. Nesta última abordagem a UFR é acoplada ao PPG por um barramento externo ao núcleo. Assim, o custo de comunicação entre o processador e a UFR é mais elevado do que na abordagem anterior.

Existem outras técnicas de acoplamento, chamadas de *fracamente acopladas*, fornecendo um alto custo de comunicação. A *UFR anexada* é um exemplo deste tipo de abordagem. A UFR fica localizada no barramento que conecta a memória cache e a interface de entrada/saída. O custo de comunicação é alto, porém menor do que a abordagem de *acoplamento independente.* Nesta última, o processador se comunica com a UFR através do barramento de entrada/saída. Dentre todas as abordagens, esta é a mais fracamente acoplada.

2.2. Granularidade

Cada UFR pode ser implementada através de diferentes tipos e tamanhos de blocos funcionais. Por exemplo, pode-se formar uma UFR somente com somadores independentes de um bit ou, em vez disso, utilizar um somador de 32 bits como unidade básica. Este tipo de escolha de projeto é denominado granularidade da UFR.

A granularidade escolhida para os blocos funcionais tem influência no número de bits necessários para reconfiguração. Utilizando o exemplo anterior, uma UFR formada por somadores de um bit como unidade básica implicam um elevado número de bits para realizar uma soma de 32 bits. Esse número elevado é devido aos bits de controle da configuração do caminho de dados pelos vários somadores. Entretanto, se

for realizado o encapsulamento de 32 somadores de um bit em uma caixa preta, abstrai-se a complexidade da configuração dos 32 somadores de um bit na reconfiguração da UFR, tornando o controle mais simples para a execução da mesma tarefa. Neste caso, denomina-se *grão fino* para a primeira abordagem e *grão grosso* para a segunda.

2.3. Mecanismo de reconfiguração

Várias propostas já foram apresentadas em relação ao mecanismo de reconfiguração da UFR. Em [Hutchings 1997; Hutchings 1999] foram demonstrados mecanismos que, em tempo de compilação, identificam partes do programa que podem ser executadas de forma mais eficiente na UFR. Entretanto, estas técnicas são dependentes de algum tipo de ferramenta para realizar esta tarefa, modificando o código compilado original. A conseqüência deste tipo de abordagem é o aumento do tempo do projeto com a adição de uma nova etapa no fluxo, além de não prover compatibilidade de software, já que existe a necessidade de recompilação do código.

Stitt [Lysecky 2006] foi um dos pioneiros a utilizar técnicas que, em tempo de execução, detectam partes do código da aplicação que podem ser executadas de forma eficiente na unidade reconfigurável. A vantagem provida por esta abordagem é a não modificação do código compilado, provendo compatibilidade de software e a conseqüente diminuição do *time-to-market* do dispositivo.

2.4. Exemplos de abordagens propostas

Existem várias abordagens propostas no meio científico e comercial que utilizam um meio reconfigurável para prover aceleração e flexibilidade na execução de aplicações. Nesta subseção serão demonstrados os trabalhos mais relevantes que contribuíram na propagação do uso de arquiteturas reconfiguráveis.

O principal componente do sistema proposto em [Hauck 1997] é uma arquitetura reconfigurável que consiste em um FPGA projetado para suportar computações intensivas. Este sistema, chamado Chimaera, é acoplado fortemente a um processador de propósito geral que compartilha recursos, como banco de registradores, com a arquitetura reconfigurável. O acoplamento utilizado por esta abordagem torna a UFR uma unidade funcional do processador, diminuindo o tempo de comunicação e de reconfiguração do FPGA.

O processador GARP [Hauser 1997] estende o conjunto de instruções MIPS-II para executá-las em uma arquitetura reconfigurável fracamente acoplada a um processador de propósito geral. A comunicação entre o PPG e a UFR é realizada através de instruções dedicadas. Um ambiente completo de processamento foi projetado para esta arquitetura, incluindo bibliotecas e memória virtual.

Em [Vassiliadis 2001] foi proposta a arquitetura reconfigurável Molen, baseada em FPGA, sendo foco de otimização desta abordagem as partes críticas da aplicação, ou seja, laços e sub-rotinas. A reconfiguração do FPGA é realizada em um grão fino. Além da arquitetura, Molen propõe um novo paradigma de programação que permite que o código de processadores de propósito geral e descrições de hardware sejam agregados em um mesmo código de programa. Ainda, neste trabalho, é proposto um novo conjunto de instruções e uma micro-arquitetura baseada em micro-código.

PipeRench [Goldstein 2000] propõe uma computação reconfigurável que utiliza a técnica de pipeline para diminuir a latência de reconfiguração e execução em um FPGA. O esclarecimento desta técnica, chamada virtualização, está ilustrado na Figura 2. Nesta figura é demonstrada uma aplicação que foi dividida em 5 estágios de pipeline (Figura 2-a), levando 7 ciclos para ser configurada e executada. Entretanto, a Figura 2-b demonstra a aplicação da técnica de virtualização nesta aplicação, onde somente 3 estágios foram necessários para executá-la. O estágio 1 da Figura 2-b foi utilizado, em diferentes períodos de tempo, para executar os estágios 1 e 4 da Figura 2-a. PipeRench propõe um acoplamento da UFR anexado ao processador, além de utilizar um compilador para realizar otimizações no código, focando o favorecimento da técnica de virtualização.

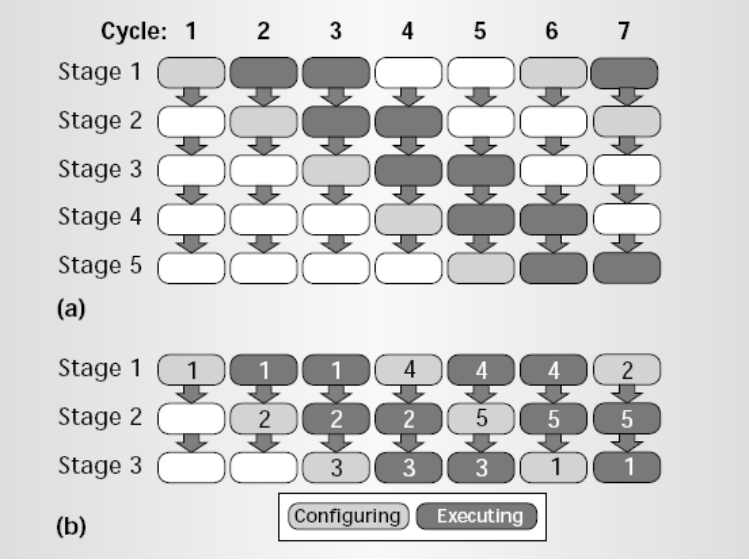

Figura 2. Exemplo de uma execução na arquitetura PipeRench [Goldstein 2000]

Os primeiros estudos sobre os benefícios de realizar dinamicamente o particionamento do software para ser executados em um hardware reconfigurável foram reportados em [Lysecky 2006]. O *warp processing* é composto por dois microprocessadores, um sendo responsável por executar a aplicação, e outro gerenciando um algoritmo de particionamento. Além destes processadores, uma memória local e um FPGA compõem a arquitetura. Nesta abordagem existe um hardware que detecta as regiões criticas da aplicação, repassando estas para o algoritmo de particionamento realizar a compilação para um grafo de controle. Este mesmo algoritmo sintetiza e mapeia na estrutura do FPGA o grafo previamente construído.

TRIPS [Sankaralingam 2004] é um exemplo de uma arquitetura híbrida Von-Neumann/*Dataflow*. Para melhor explorar o paralelismo da aplicação, esta arquitetura disponibiliza um grande número de recursos, além de utilizar três diferentes modos de execução: D-Morph, que explora o paralelismo em nível de instrução; T-Morph, que trabalha no nível de threads; e S-Morph, que é apropriada para aplicações do tipo *streaming* e que apresentam um alto nível de paralelismo de dados. Uma abordagem totalmente *Dataflow* é apresentada em [Swanson 2003]: Wavescalar, que abandona totalmente o conceito de contador de programa e a forma linear de execução de Von-Neumann, que, como explicitado anteriormente, limita a quantidade de paralelismo a ser

explorado na aplicação. A principal característica desta arquitetura é que ela possui centenas de centros de processamento distribuídos em vez de uma única unidade central.

Tanto TRIPS quanto Wavescalar mostraram excelentes resultados em relação a aumento de desempenho. Entretanto, a complexidade de implementação de um compilador, causada pela complexa alocação das operações nas unidades de processamento, e a indisponibilidade atual de área de silício para implementá-la, torna esta uma abordagem inadequada para a tecnologia atual.

Em [Beck 2005] foi proposto um algoritmo de tradução binária que dinamicamente realiza a tradução de uma seqüência de instruções em um mapeamento da unidade reconfigurável. Neste caso, a arquitetura alvo desta abordagem é um processador que executa nativamente Java. A estrutura reconfigurável é caracterizada como grão grosso por conter replicações de unidades funcionais equivalentes às unidades do processador. Além disto, a comunicação entre o PPG e UFR é desempenhada por um barramento dedicado dentro do núcleo, o que a classifica como uma UFR fortemente acoplada. Entretanto, em [Beck 2006] foi feita uma modificação do algoritmo de tradução binária, sendo uma arquitetura RISC o alvo de implementação deste estudo.

Este último estudo mostra-se parecido, no que diz respeito ao acoplamento, à arquitetura Chimaera. Entretanto, em Chimaera e Garp a execução na plataforma reconfigurável necessita de uma ferramenta que, em tempo de compilação, modifica o código e adiciona instruções especiais. Este passo é evitado em [Beck 2006] com a implementação do algoritmo dinâmico de tradução binária. Em [Lysecky 2006] a detecção de partes da aplicação também é realizada dinamicamente. Porém, o algoritmo de particionamento requer uma grande quantidade de memória (aproximadamente 8 MB) para sua execução. Ainda, o FPGA que compõe a arquitetura provê uma longa latência de configuração, além de consumir uma quantidade considerável de área e potência estática. A latência de reconfiguração também é um problema destacado em Molen, devido a sua granularidade fina de reconfiguração.

3. Motivação

Em [Beck 2006], uma unidade funcional reconfigurável acoplada a um processador RISC é combinada com as técnicas de Tradução Binária (*Binary Translation*) e de reuso de seqüências de instruções (*Trace Reuse*). Assim, esta arquitetura permitiu explorar o paralelismo em nível de instruções no processador RISC e, ao mesmo tempo, revelou-se uma atraente solução para o problema da compatibilidade de software. Em [Beck 2008] demonstra-se que, com tal arquitetura, é possível aumentar o desempenho de sistemas com aplicações heterogêneas – que é o caso de sistemas embarcados – e ao mesmo tempo reduzir o consumo energético.

O trabalho proposto neste artigo tem como base os dois trabalhos acima mencionados, os quais foram desenvolvidos no grupo de Sistemas Embarcados da UFRGS e serão descritos a seguir.

O sistema reconfigurável proposto em [Beck 2006] divide-se em duas partes: a Unidade Funcional Reconfigurável (UFR) e o módulo de Tradução Binária (TB). A plataforma alvo da implementação do sistema é o processador MIPS R3000 [Beck

2008], amplamente utilizado no domínio de sistemas embarcados. A UFR já foi implementada em hardware e está funcional. O TB ainda não foi implementado.

3.1. A Unidade Funcional Reconfigurável (UFR)

Uma visão geral da organização da UFR pode ser vista na Figura 3. A unidade é bidimensional, e cada instrução é alocada numa posição especificada pelas respectivas linha e coluna da matriz. Cada coluna é homogênea, contendo apenas um tipo de unidade funcional ordinária (exemplos: unidade lógica e aritmética (ULA), *shifter*, multiplicador, unidade de *load*/*store*).

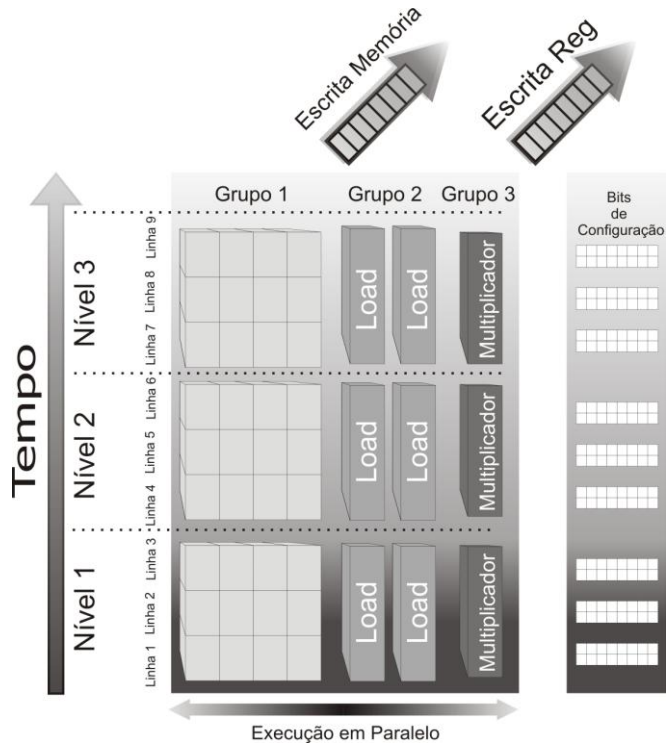

Figura 3. Estrutura da Arquitetura Reconfigurável [Beck 2006]

Para simplificar a alocação das instruções na estrutura reconfigurável, as unidades funcionais foram classificadas em grupos. No exemplo ilustrado na Figura 3, no Grupo 1 existem quatro ULAs dispostas horizontalmente. Assim, é possível executar até quatro operações deste tipo em paralelo. Após o cálculo do caminho crítico do processador MIPS R3000, foi verificado que, em um ciclo de relógio do processador, é possível realizar três operações lógicas e aritméticas. Conseqüentemente, em um nível (respectivo a um ciclo de relógio do processador) da arquitetura reconfigurável foram encadeadas três unidades funcionais deste grupo.

De forma análoga às unidades funcionais do Grupo 1, a Figura 3 ilustra um alinhamento horizontal de duas unidades de *load*/*store*, representando o Grupo 2. Neste exemplo, a arquitetura é capaz de executar até dois acessos à memória em um mesmo ciclo de relógio. O número de unidades dispostas horizontalmente deste grupo é dependente da disponibilidade de portas existente no modelo de memória utilizado. Finalmente, para o Grupo 3 no exemplo utilizado, somente uma multiplicação pode ser executada por nível da arquitetura.

Basicamente, as unidades funcionais que compõem a estrutura reconfigurável são interligadas através de multiplexadores e demultiplexadores. A Figura 4 ilustra o esquema de interconexão da arquitetura. À esquerda da Figura 4 existe o barramento que conecta o contexto de entrada ao contexto de saída, abrangendo todos os níveis da estrutura reconfigurável. O objetivo do barramento é fornecer os operandos, advindos do contexto de entrada, para as unidades funcionais e receber os resultados das operações para posterior gravação destas no contexto de saída.

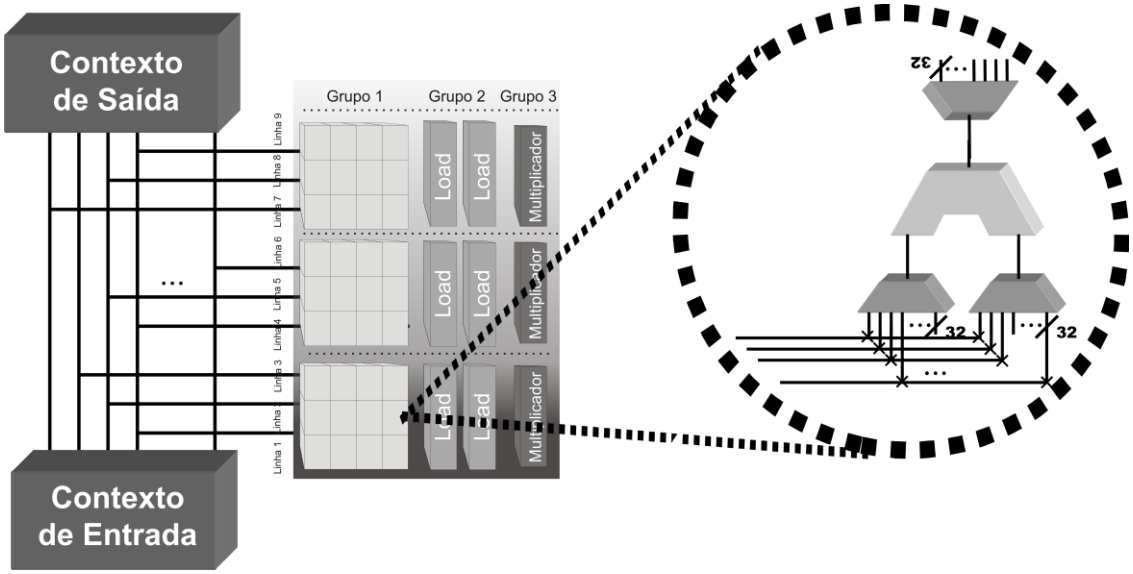

Figura 4. Esquemático do modelo de interconexão da estrutura reconfigurável [Rutzig 2008]

Ao iniciar a execução na arquitetura reconfigurável, todos os operandos estão armazenados no contexto de entrada. Para realizar o roteamento destes até as entradas das unidades funcionais, são necessários dois multiplexadores para cada uma destas unidades, como ilustrado à direita da Figura 4. Estes selecionam os registradores corretos a serem carregados do barramento e repassam seus valores à unidade funcional. O número de entradas de cada um destes multiplexadores é igual ao número de registradores que compõem o contexto de entrada.

Após o término da execução da instrução na unidade funcional, deve-se realizar a escrita do resultado no registrador destino. Assim, um demultiplexador é inserido na saída de cada unidade funcional. Seu papel é selecionar a linha do barramento onde o resultado deve ser propagado, para finalmente este ser escrito no registrador destino no contexto de saída. Análogo ao multiplexador de entrada, o número de saídas dos demultiplexadores é igual ao número de registradores que compõem o contexto de saída.

3.2. O Tradutor Binário (BT)

A idéia básica do mecanismo é prover compatibilidade de software a partir da tradução binária de seqüências de instruções, em tempo de execução, para que as mesmas possam ser futuramente executadas num mecanismo mais eficiente – neste caso, em uma unidade funcional reconfigurável.

O tradutor binário trabalha em paralelo ao processador. Basicamente, o mecanismo avalia cada instrução executada pelo processador, agrupando-as em blocos chamados de *configuração* da UFR. A cada instrução executada pelo processador, é verificada a possibilidade da execução da mesma na unidade reconfigurável juntamente com as outras instruções já alocadas na configuração. Caso positivo, a mesma é alocada na configuração corrente da UFR. Ao final da construção de uma configuração da UFR, esta é armazenada em uma cache de configurações, indexada pelo valor do contador de programa da primeira instrução do bloco. Quando este valor novamente for alcançado pelo contador de programa, o mecanismo reconfigurável é ativado seguindo os seguintes passos:

- Efetua-se a busca da configuração, da seqüência de instruções em questão, na cache de configurações;
- Configura-se a UFR com os bits fornecidos pela cache;
- Os valores dos registradores necessários para executar a configuração são carregados no contexto de entrada da UFR;
- A execução é realizada na UFR.

Após o término da execução na UFR, o valor do contador de programa é atualizado, assim como os valores dos registradores modificados no banco de registradores. Do mesmo modo, as escritas na memória de dados são realizadas. É importante ressaltar que enquanto o mecanismo reconfigurável está ativo, o processador não realiza qualquer operação.

Diferentemente dos processadores superescalares, esta abordagem não realiza repetidas vezes, para a mesma seqüência de instruções, a verificação das dependências entre as instruções. A utilização da técnica de reuso de rastros (do inglês *trace reuse*) evita a repetição desta tarefa. Portanto, se não houver falha na cache de configurações, o mecanismo de TB somente será aplicado uma única vez em cada seqüência de instruções.

3.3. Objetivos do trabalho

O trabalho proposto consiste na implementação, em hardware, do módulo de Tradução Binária já apresentado e a integração do mesmo com a Unidade Funcional Reconfigurável e com o processador MIPS R3000. O objetivo final do grupo de pesquisa é a prototipação em FPGA de um *player* de música em formato MP3, utilizando o processador MIPS acoplado ao sistema reconfigurável.

Para a implementação, será utilizada a linguagem VHDL, uma linguagem de descrição de hardware. Tanto a UFR quanto o processador MIPS R3000 estão descritos nesta linguagem. A vantagem da utilização desta é a possibilidade de extração de informações precisas de área, potência e freqüência máxima de operação numa dada tecnologia de fabricação. Além disso, a modelagem em VHDL torna possível a prototipação do sistema em FPGA.

4. Considerações finais

Este artigo apresentou, superficialmente, o contexto atual da pesquisa em sistemas embarcados e arquiteturas reconfiguráveis. Foi apresentado o trabalho produzido no

grupo de Sistemas Embarcados da UFRGS nesta área, com base no qual foi feita uma proposta de implementação. Esta, quando concluída, servirá como base para a prototipação do sistema reconfigurável do grupo e permitirá a extração de informações precisas de área e potência do sistema em tecnologias de fabricação atuais.

Referências

BECK FILHO, A. C. S.; CARRO, L. Dynamic reconfiguration with binary translation: breaking the ILP barrier with software compatibility. In: DESIGN AUTOMATION CONFERENCE, DAC, 42., 2005, Anaheim. **Proceedings…** New York: ACM Press, 2005. p. 732 – 737.

BECK FILHO, A.C.S.; CARRO, L. Automatic Dataflow Execution with Reconfiguration and Dynamic Instruction Merging. In: VERY LARGE SCALE INTEGRATION, VLSI-SOC, 2006, Perth. **Proceedings…** New York: IEEE Computer Society, 2007. p. 30–35.

BECK FILHO, A.C.S.; RUTZIG, M.B.; CARRO, L. Transparent Reconfigurable Acceleration for Heterogeneous Embedded Applications. Aceito para DATE, 2008, Munich.

COMPTON, K.; HAUCK, S. Reconfigurable computing: A survey of systems and software. **ACM Computing Surveys**, New York, v. 34, n. 2, p. 171- 210, junho de 2002.

GAJSKI, D. et al. SpecSyn: An Environment Supporting the Specify-Explore-Refine Paradigm for Hardware/Software System Design. **IEEE Transactions on VLSI Systems**, Princeton, v. 6, n. 1, p. 84-100, março de 1998.

GOLDSTEIN, S. C.; SCHMIT, H.; BUDIU, M.; CADAMBI, S.; MOE, M.; TAYLOR, R. R. PipeRench: A Reconfigurable Architecture and Compiler. **Computer**, vol. 33, n. 4, p. 70-77, abril de 2000.

GUPTA, R. K.; MICHELI, G. D. Hardware-software co-synthesis for digital systems. **IEEE Design and Test of Computers**, Santa Barbara, v. 10, n. 3, p. 29 – 41, setembro de 1993.

HAUCK, S. et al. The Chimaera reconfigurable functional unit. In: FPGA-BASED CUSTOM COMPUTING MACHINES, 1997, Napa Valley. **Proceedings**… Washington: IEEE Computer Society, 1997. p. 87 – 96.

HAUSER, J. R.; WAWRZYNEK J. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In: FPGA-BASED CUSTOM COMPUTING MACHINES, 1997, Napa Valley. **Proceedings**… Washington: IEEE Computer Society, 1997. 12-21.

HENKEL, J.; ERNST, R. A Hardware/Software Partitioner using a Dynamically Determined Granularity. In: DESIGN AUTOMATION CONFERENCE, DAC, 42., 2005, Anaheim. **Proceedings…** New York: ACM Press, 2005. p. 732 – 737.

HENKEL, J. A low power hardware/software partitioning approach for core-based embedded systems. In: DESIGN AUTOMATION CONFERENCE, DAC, 36., 1999, Anaheim. **Proceedings…** New York: ACM Press, 2005. p. 122 – 127.

HUTCHINGS, B. L. Exploiting reconfigurability through domain-specific systems. In: INTERNATIONAL WORKSHOP ON FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS, FPL, 7., 1997, London. **Proceedings…** Berlin: Springer, 1997. p.193 – 202. (Lecture Notes in Computer Science, v.1304).

HUTCHINGS, B. L. et al. A CAD suite for high-performance FPGA design. In: SYMPOSIUM ON FIELD-PROGRAMMABLE CUSTOM COMPUTING MACHINES, 1999, Napa Valley. **Proceedings…** Washington: IEEE Computer Society, 1999. p. 12–24.

LYSECKY, R.; STITT, G.; VAHID, F. Warp Processors. **ACM Transactions on Design Automation of Electronic Systems**, New York, v.11, n. 3, p. 659 – 681, julho de 2006.

RUTZIG, M. B. **Gerenciamento Automático de Recursos Reconfiguráveis Visando a Redução de Área e do Consumo de Potência em Dispositivos Embarcados**, Porto Alegre, março de 2008.

SANKARALINGAM, K. et al. Trips: A polymorphous architecture for exploiting ilp, tlp, and dlp. **ACM Transactions on Architecture and Code Optimization**, New York, v.1, n. 1, p. 62 – 93, maio de 2004.

STITT, G.; VAHID, F. The Energy Advantages of Microprocessor Platforms with On-Chip Configurable Logic, **IEEE Design and Test of Computers**, Los Alamitos, v. 19, n. 6, p. 36 – 43, novembro de 2002.

SWANSON, S. et al, The WaveScalar Architecture. ACM Transactions on Computer Systems, New York, v.25, n. 2, maio de 2007.

VASSILIADIS, S.; WONG, S.; COTOFANA, S. The MOLEN ρμ-coded processor. In: INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS, FPL, 11., 2001, Belfast. **Proceedings…** Berlin: Springer, 2001. p. 275 – 285. (Lecture Notes in Computer Science, v. 2147)

VENKATARAMANI, G. et al. A Compiler Framework for Mapping Applications to a Coarse-grained Reconfigurable Computer Architecture. In: INTERNATIONAL CONFERENCE ON COMPILERS, ARCHITECTURE AND SYNTHESIS FOR EMBEDDED SYSTEMS, 2001, Atlanta. **Proceedings…** New York: ACM Press, 2001. p. 116 – 125.

XU, B.; ALBONESI, D.H. Methodology for the analysis of dynamic application parallelism and its application to reconfigurable computing. In: INTERNATIONAL SYMPOSIUM ON VOICE, VIDEO AND DATA COMMUNICATIONS, 1999, Bellingham. **Proceedings…** Bellingham: SPIE, 1999. p. 78-86.

WALL, D. W. Limits of instruction-level parallelism. In: INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, 1991, Santa Clara. **Proceedings...** New York: ACM Press, 1991. p.176 – 188.

WAN, M. et al. An Energy Conscious Methodology for Early Design Space Exploration of Heterogeneous DSPs. In: CUSTOM INTEGRATED CIRCUITS CONFERENCE,

1998, Santa Clara. **Proceedings...** Washington: IEEE Computer Society, 1998. p. 111 – 117.