UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

FERNANDA MATHIAS CAPELLA

Arquitetura Reconfigurável Multi-ISA

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Luigi Carro
Co-orientador: Prof. Dr. Antonio Carlos S. Beck Filho

Porto Alegre
2014

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Capella, Fernanda Mathias

Arquitetura Reconfigurável Multi-ISA [manuscrito] / Fernanda Mathias Capella. – 2014.

86 f.:il.

Orientador: Luigi Carro; Co-orientador: Antonio Carlos S. Beck Filho.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2014.

1.Tradução Binária. 2.Arquiteturas Reconfiguráveis. 3.Execução Transparente. I. Carro, Luigi. II. Beck Filho, Antonio C. S.. III. Arquitetura Reconfigurável Multi-ISA.

AGRADECIMENTOS

Agradeço primeiramente ao meu orientador Luigi Carro pela paciência e pela confiança depositada em mim. Ao meu co-orientador Caco, agradeço imensamente pela paciência, pelas revisões e sobretudo pelo apoio e pelas palavras de incentivo.

Agradeço ao Marcelo Brandalero por suas contribuições, fundamentais para a finalização desse trabalho. A todos os colegas que estiveram comigo ao longo desses anos, partilhando seu conhecimento e amizade, fica também meu muito obrigado.

Finalmente, agradeço à minha família e amigos pela torcida e compreensão nos momentos de ausência.

RESUMO

O mercado de sistemas embarcados tem demandado uma variada gama de aplicações, aplicações estas cada vez mais complexas. Para atender tal demanda, visto o declínio da lei de Moore e os processadores chegando ao seu limite de dissipação térmica, os projetistas são pressionados a desenvolverem novas organizações computacionais. Para manter a compatibilidade binária, de forma que a grande quantidade de aplicativos e ferramentas já desenvolvidas possa ser reutilizada, as empresas desenvolvem seus produtos focando em melhorias de um dado processador que irá executar a mesma ISA *(Instruction Set Architecture)*. Essa necessidade de compatibilidade de código impõe muitas restrições à equipe de projeto, haja vista as limitações impostas pela ISA legada. A Tradução Binária (TB) abre novas possibilidades aos projetistas, visto que permite a execução de códigos previamente compilados para uma determinada arquitetura em outra arquitetura. No entanto, a TB acrescenta mais uma camada entre o código e sua execução, trazendo perdas de desempenho. Este trabalho explora um novo mecanismo de tradução binária dinâmico de dois níveis que, ao trocar o primeiro nível, pode executar ISAs diferentes de forma totalmente transparente e ainda amortiza os custos de tradução. Da mesma forma ao trocar o segundo nível de tradução binária pode-se trocar a arquitetura alvo. Com base nesse tradutor de dois níveis, é apresentado como estudo de caso um sistema computacional composto por uma arquitetura reconfigurável capaz de executar códigos x86, ARM, PowerPC e MIPS de forma transparente, com compatibilidade binária e com ganhos de desempenho.

Palavras-chave: Tradução Binária, Arquiteturas Reconfiguráveis, Execução Transparente.

Multiple-ISA Reconfigurable Architecture

ABSTRACT

The embedded systems market is demanding a wide range of applications, and these applications are increasing in complexity. In order to meet this demand, since the decline of Moore's law and processors reaching their thermal dissipation limits, designers are pushed to develop new computer organizations. In order to support binary compatibility, so that the large quantity of applications and tools already deployed can be reused, companies develop their products focusing on improvement of a given processor that will execute the same ISA (Instruction Set Architecture) as before. This need for code compatibility impose a lot of restrictions to the design team, considering the limitations imposed by the legacy ISA. Binary Translation (BT) open new possibilities for designers, since it allows the execution of a code previously compiled to a specific architecture in another architecture. However, BT adds another layer between code and actual execution, therefore bringing performance penalties. This work explores a dynamic two-level binary translation system that, by changing the first BT level, allows the execution of different ISAs in a transparent fashion and still amortizes translation costs. In the same way, it is possible to switch to another target architecture by only changing the second BT level. Based on this two-level translator this work presents, as a case study, a computational architecture comprising of an dynamic reconfigurable array that can execute x86, ARM, PowerPC and MIPS binary codes in a transparent way, maintaining binary compatibility with performance gains.

Keywords: Binary Translation. Reconfigurable Architecture. Transparent Execution.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

BPU	Branch Processing Unit
CMS	Code Morphing Software
CPSR	Current Program Status Register
DAISY	Dynamically Architected Instruction Set from Yorktown
DLL	Dynamic-Link Library
FPU	Floating-point Processing Unit
FXU	Fixed-point Processing Unit
ILP	Instruction Level Parallelism
ISA	Instruction Set Architecture
NOC	Network On-Chip
SPSR	Saved Program Status Register
TB	Tradução Binária
VLIW	Very Large Instruction Word
ULA	Unidade Lógica Aritmética

SUMÁRIO

1 INTRODUÇÃO

O crescimento e a popularização do mercado de sistemas embarcados, alavancados em grande parte pelo uso de smartphones, têm exigido dos fabricantes o desenvolvimento de novas arquiteturas, com capacidade de processar aplicações com características distintas (KUMAR, 2010), em um mesmo dispositivo, e que atendam às restrições de consumo de energia e potência (BUTTAZZO, 2006). Buscando atender essa demanda, os projetistas tem buscado o desenvolvimento de novas organizações computacionais, haja vista o declínio da lei de Moore e os processadores chegando ao seu limite de dissipação térmica (FLYNN e HUNG, 2005) (SIMA, 2004).

No entanto, a entrada dessas novas organizações de processadores no mercado fica bastante restrita se não for mantida a compatibilidade binária com as ISAs (Instruction Set Architecture) existentes, visto que os fabricantes relutam em desenvolver arquiteturas diferentes pelo risco de perder as vantagens comerciais de sua base de software disponível (ALTMAN, KAELI e SHEFFER, 2000). Grande parte do valor que o usuário dá a um novo dispositivo se dá à quantidade de software disponível para ele, vide, por exemplo, o sucesso e crescimento de dispositivos Android. Por outro lado, manter a compatibilidade binária com uma determinada ISA pode limitar o desenvolvimento de uma nova arquitetura, impondo uma série de restrições ao projeto. Restrições essas inerentes à ISA legada.

Nesse cenário, são necessárias novas alternativas para minimizar esse problema. Pela perspectiva de desenvolvimento de software, o trabalho de passar o código fonte pelo processo de compilação, que envolve adaptação de código, depuração e teste, deve ser minimizado, de forma que o *time to market* seja o menor possível. Desenvolvedores de hardware, por outro lado, devem ter a liberdade de implementar quaisquer arquiteturas e organizações que considerem a melhor alternativa, levando em conta o mercado e os requisitos não funcionais, tais como desempenho e consumo de potência.

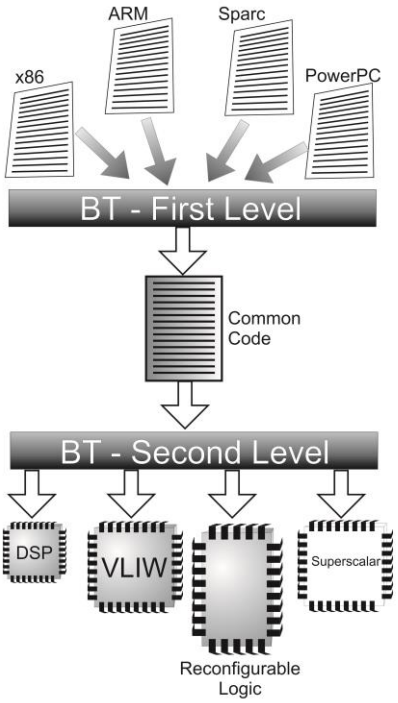
Uma solução que vem ao encontro desse impasse é a tradução binária (SITES, 1993). A Tradução Binária (TB) permite que códigos previamente compilados para uma arquitetura sejam executados em outra arquitetura. Dessa forma, a TB pode devolver aos projetistas de hardware a liberdade até então perdida, visto que não precisam ficar amarrados a uma ISA específica e às suas limitações inerentes. Da mesma forma, nem engenheiros de software, nem usuários precisam sofrer com problemas de portabilidade de código.

Toda tradução binária tem um custo, seja ele de hardware adicional e/ou tempo de processamento. Para que a uma solução de tradução binária seja competitiva ela deve mitigar

os custos de tradução, seja por otimização de software ou por aceleração de hardware, tornando sua execução competitiva com a execução na arquitetura nativa.

O trabalho aqui apresentado investiga um sistema de tradução binária dinâmico de dois níveis que além de manter a compatibilidade binária amortiza seus custos de tradução. Uma visão geral do sistema de tradução pode ser visto na Figura 1.1. O primeiro nível de tradução binária é responsável por traduzir o código fonte original para um código intermediário (comum). O segundo nível de tradução binária é responsável por transformar o código intermediário para o código da arquitetura alvo e otimizá-lo.

Figura 1.1 – Visão geral do sistema proposto



Fonte: Fajardo (2013 p. 2)

A vantagem do tradutor proposto é que, tendo uma interface bem definida entre os dois níveis de tradução, e apenas mudando o primeiro nível de tradução, é possível migrar para ISAs diferentes de forma transparente para o segundo nível. Assim, pretende-se facilitar o porte de ISAs radicalmente diferentes sem a necessidade de modificações na arquitetura alvo, contando com diferentes blocos de tradução no primeiro nível. Da mesma forma, é possível trocar para outra arquitetura alvo, de acordo com as necessidades da aplicação ou com a arquitetura disponível no momento. Nenhum dos trabalhos correlatos, discutidos na próxima seção, pode facilmente chavear entre arquitetura nativa/alvo (quando implementada

em hardware), ou utiliza uma arquitetura especial (como um array reconfigurável), para acelerar a execução do código após a tradução.

No trabalho desenvolvido por (Fajardo, 2011) foi proposto um sistema binário de primeiro nível responsável por traduzir códigos x86 para MIPS (código intermediário). No segundo nível foi utilizado um array reconfigurável dinâmico, previamente proposto por (Beck et al. 2008). No presente trabalho foi implementado o TB de primeiro nível para outras duas arquiteturas ARM e PowerPC e mantido o mesmo TB de segundo nível. A escolha de processadores ARM como extensão desse sistema foi feita por sua grande penetração no mercado de embarcados. Enquanto os processadores x86 tem grande relevância no mercado de processadores de propósito geral, os processadores ARM tem dominado o mercado de embarcados. A terceira arquitetura escolhida, PowerPC tem abrangência em diversos mercados, desde sistemas embarcados até servidores.

A principal contribuição desse trabalho foi um tradutor binário de primeiro nível para o sistema de TB de dois níveis, capaz de traduzir das arquiteturas ARM e PowerPC para a arquitetura MIPS. O desenvolvimento desse tradutor inclui:

- Desenvolvimento de um simulador em linguagem C (cerca de 5mil linhas de código). O simulador, a partir do trace de execução de um programa na arquitetura nativa (ARM ou PowerPC), traduz as instruções gerando o *assembly* da execução na arquitetura comum (MIPS). O simulador permite ainda extrair dados sobre a execução dos programas, como instruções mais executadas, uso de registradores, entre outras informações. Com o uso desse simulador foi possível extrair os dados de desempenho do tradutor para execução dos benchmarks;
- Desenvolvimento de um protótipo em hardware em uma FPGA Xilinx Virtex5 do TB de primeiro nível que inclui as unidades de decodificação das duas arquiteturas nativas (ARM e PowerPC) e codificação para a arquitetura comum (MIPS).

1.1 Organização

Este trabalho segue organizado da seguinte maneira. No Capítulo 2 são abordados os principais conceitos de tradução binária, procurando familiarizar o leitor com os termos, e definições utilizados em sistemas de tradução binária. Neste mesmo capitulo, é ainda feita

uma revisão bibliográfica das principais arquiteturas de tradução binária com uma breve explanação sobre seus objetivos e funcionamento.

No Capitulo 3 são apresentadas as duas arquiteturas nativas e a arquitetura alvo utilizadas nesse trabalho, com objetivo de facilitar ao leitor o entendimento das diferenças arquiteturais que impactarão no processo de tradução.

No Capitulo 4 é apresentado o projeto fruto desse trabalho, detalhando o sistema proposto, a organização dos dois níveis de tradução binária e exemplificando o processo de tradução.

No Capitulo 5 são mostrados os resultados obtidos em termos de área e desempenho, bem como a metodologia empregada.

Finalmente, no Capítulo 6 discutimos os resultados obtidos, as melhorias que podem ser feitas e como o sistema pode ser estendido para outras arquiteturas.

2 TRADUÇÃO BINÁRIA

2.1 Conceitos de Tradução Binária

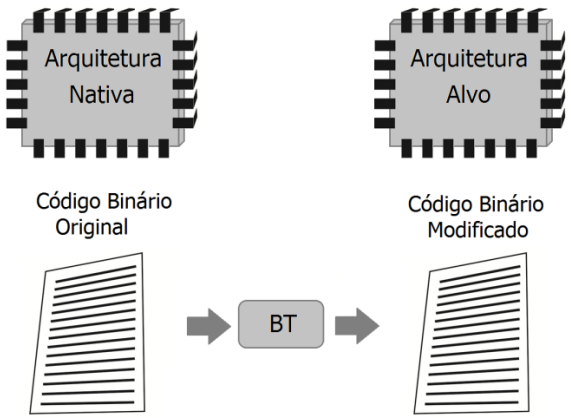
Atualmente a maioria dos microprocessadores comerciais se mantém presos firmemente a uma ISA legada. Apesar das deficiências conhecidas dessas ISAs, os fabricantes são relutantes em abandoná-las devido ao risco de perda das vantagens comerciais de sua base de software existente.

Por outro lado, para os desenvolvedores de software, portar código para uma nova arquitetura é dispendioso, e se essa nova arquitetura não ganhar mercado, estarão perdendo um investimento significativo em tempo de desenvolvimento.

Ambos os fatores conspiram contra inovações no desenvolvimento de processadores, limitando inclusive o surgimento de novos competidores.

A tradução binária surgiu pela necessidade das empresas em reduzir seu *time-to-market* e manter a compatibilidade com sua base de software existente. Ela permite a execução de aplicações, ou até sistemas operacionais completos, previamente compilados para uma determinada arquitetura, em arquiteturas diferentes. A Figura 2.1 ilustra o processo de tradução de um código binário gerado para uma determinada ISA para outra ISA, permitindo assim a sua execução no processador alvo.

Figura 2.1 – Processo de Tradução Binária (TB)



Os conceitos de tradução binária (ALTMAN, KAELI e SHEFFER, 2000) são amplos e aplicados a diversos níveis em uma arquitetura computacional. Basicamente, um Tradutor Binário (TB) é um sistema que pode ser implementado em hardware ou software, responsável por analisar um código binário de um programa previamente compilado para uma

determinada arquitetura, e então transformá-lo em código de outra arquitetura. Ou seja, a tradução binária permite a execução de código binário legado em outra arquitetura sem necessidade de recompilação.

Os principais conceitos de tradução binária, e que serão usados ao longo desse trabalho, são os seguintes:

- Arquitetura nativa: a arquitetura original (legada), para a qual o código foi originalmente compilado. O código nativo serve de entrada para o tradutor binário.
- Arquitetura alvo: é a arquitetura do processador no qual queremos executar o código. O código na arquitetura alvo é o produto do tradutor binário.
- Cache de tradução (Transalation Cache – TCache): memória utilizada para armazenar o código traduzido para futura reutilização.
- Virtual Machine Monitor (VMM): parte do tradutor binário responsável pelo controle do processo de tradução.

Segundo (ALTMAN, KAELI e SHEFFER, 2000), os sistemas de tradução binária podem ser classificados em três tipos, ainda que soluções híbridas sejam usadas buscando unir as melhores características de cada um:

- Emuladores: interpretam as instruções em tempo de execução, mas não salvam o código traduzido para uso futuro. Como nenhum código traduzido é salvo, não oferece otimização de execução. Em geral, esse tipo de tradutor não requer intervenção do usuário;
- Dinâmicos: traduz o código de uma ISA nativa para uma ISA alvo, salvando partes do código para uso futuro. Os trechos de código salvos podem ser usados para melhorar o desempenho do sistema. Por exemplo, códigos frequentemente executados podem ser mantidos em uma TCache. Em geral, esse tipo de tradutor não requer intervenção do usuário;
- Estáticos: traduz o código off-line, antes de sua execução, podendo assim aplicar mais otimizações do que um tradutor dinâmico. Para realizar tais otimizações pode analisar todo o programa antes da execução, bem como utilizar profiles de uma execução prévia. Ao contrário dos emuladores e tradutores dinâmicos, geralmente requerem alguma intervenção do usuário e, portanto, a tradução pode não ser transparente.

A tradução binária pode ser aplicada em diversos níveis em uma arquitetura computacional:

- Aplicação: nesse caso o tradutor binário comporta-se como um aplicativo, utilizando bibliotecas do sistema operacional. Somente é utilizado quando um aplicativo da arquitetura nativa precisa ser executado. O sistema operacional, nesse caso, é compilado para a arquitetura alvo;
- Interface: o tradutor binário situa-se numa camada abaixo do sistema operacional e, portanto, pode executar tanto o sistema operacional como aplicativos compilados para a arquitetura nativa. Do ponto de vista do sistema operacional, o TB funciona de maneira similar a uma máquina virtual, pois ele faz a interface com o hardware da arquitetura alvo;
- Hardware: o tradutor binário é implementado como um componente de hardware adicional e com controle próprio independente em relação à arquitetura alvo, permitindo que o processo de tradução seja paralelo ao de execução do processador alvo.

De acordo com (ALTMAN, KAELI e SHEFFER, 2000), como todas as máquinas são baseadas do Modelo de Turing, qualquer computação feita em uma máquina pode ser emulada em outra. No entanto, alguns desafios, discutidos brevemente nos próximos parágrafos, existem para que um tradutor binário execute o código nativo em uma arquitetura alvo com a mesma eficiência que a execução nativa.

Uma necessidade básica de um tradutor binário é realizar o mapeamento dos registradores da arquitetura nativa para a arquitetura alvo. No entanto, em alguns casos a arquitetura alvo possui menos registradores que a arquitetura nativa e alguns registradores devem ser mantidos em memória. A escolha do conjunto de registradores a serem mantidos em memória influenciará no desempenho da execução, visto que os acessos à memória têm mais custo. Além disso, alguns registradores de uso específico da arquitetura nativa poderão não estar disponíveis na arquitetura alvo (por exemplo, registradores que mantêm *flags* da ULA).

O mapeamento em memória de dispositivos de entrada e saída também pode ser diferente entre arquiteturas nativa e alvo e representa um desafio no processo de tradução.

Outro cuidado necessário durante o processo de tradução é o mapeamento de instruções atômicas, e de instruções que na arquitetura nativa são completadas em um ciclo e na arquitetura alvo em mais de um ciclo. Temos que ter cuidado para que não ocorram

interrupções até que todo o comportamento da instrução nativa seja executado, bem como cuidado para não ocorrer reordenamento dessas instruções no processador alvo.

Em alguns casos, o tradutor binário também precisa lidar da com porções de código automodificável, por exemplo – neste caso, teria de invalidar entrada antiga na TCache correspondente ao código que fora traduzido e depois modificado. Códigos que se auto referenciam também podem causar problema: o cálculo de um *checksum* da nova versão será diferente, por exemplo. Nesse caso, uma solução seria manter uma cópia da versão original do programa para realizar o cálculo.

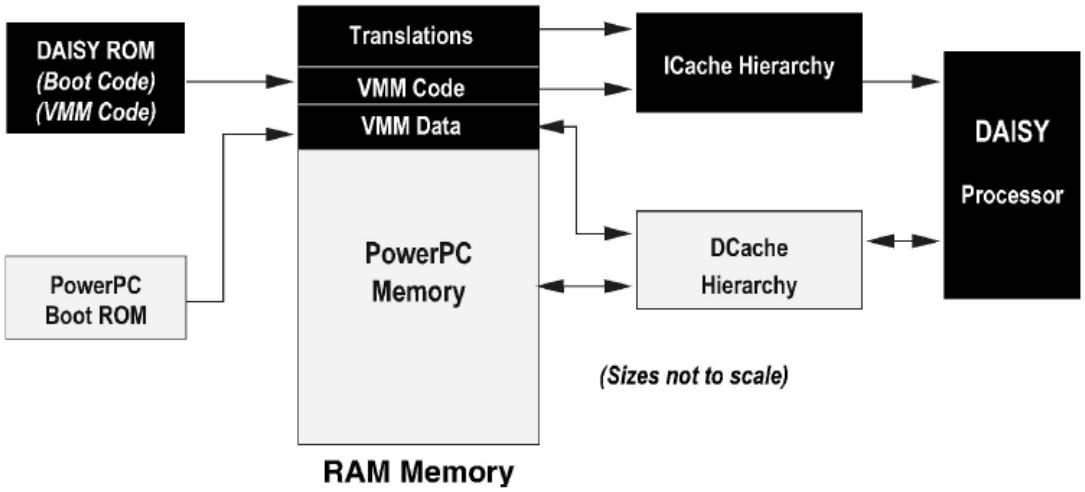
2.2 Trabalhos Correlatos

2.2.1 DAISY

O sistema DAISY (*Dynamically Architected Instruction Set from Yorktown*) visa principalmente a execução de código nativo PowerPC em um processador VLIW, embora alguns trabalhos mostrem Daisy utilizando S/390 e x86 como arquitetura nativa (EBCIOGLU; GSCHWIND, 2001).

Os componentes do DAISY podem ser vistos na Figura 2.2. O processo de tradução é totalmente transparente para a aplicação PowerPC. No momento em que o sistema inicializa, o controle é transferido para o software DAISY, referido como VMM (*Virtual Machine Monitor*). A ideia básica do VMM é implementar uma máquina virtual PowerPC, que é invisível ao software que está sendo executado. O VMM é parte do firmware, atuando em todo conjunto de instruções, que inclui qualquer tipo de operações de sistema. Sendo assim, diferente de alguns sistemas de TB que serão discutidos a seguir, o DAISY não é parte ou um serviço do sistema operacional que executa na arquitetura alvo, e pode inclusive realizar o boot de um sistema operacional compilado para PowerPC.

Figura 2.2 - Componentes do sistema Daisy



Fonte: Ebcioglu et. al (2001, p. 532).

O código VMM DAISY é armazenado na memória flash DAISY ROM. Quando o sistema inicializa, o código VMM é copiado para a memória RAM na parte alocada para o DAISY, e o processador VLIW começa sua execução. Concluída a inicialização do VMM em si e do sistema, o VMM começa a traduzir o código PowerPC da memória flash ROM para execução no processador VLIW. Então, esse código traduzido carrega o sistema operacional (no caso, Unix AIX), o qual o DAISY também traduz e executa. Depois disso, qualquer aplicação que é executada sobre o AIX se beneficia do mecanismo de tradução e da execução no processador VLIW.

A primeira vez que o VMM encontra um fragmento de instruções PowerPC, estas são interpretadas. Durante a interpretação, também é realizada a análise do código (*profiling*): os dados gerados serão usados posteriormente para geração de código. Considerando que o propósito do sistema é alcançar o desempenho máximo, a decisão sobre se o fragmento vale a pena ou não ser traduzido recai sobre quantas vezes ele já foi interpretado, seu nível de ILP *(Instruction Level Paralelism)* e número de operações realizadas. A vantagem de usar um limiar antes da tradução de fato, é que códigos raramente executados (tal como processo de inicialização) não serão traduzidos, visto que toda tradução tem um custo associado (e.g.: memória de tradução).

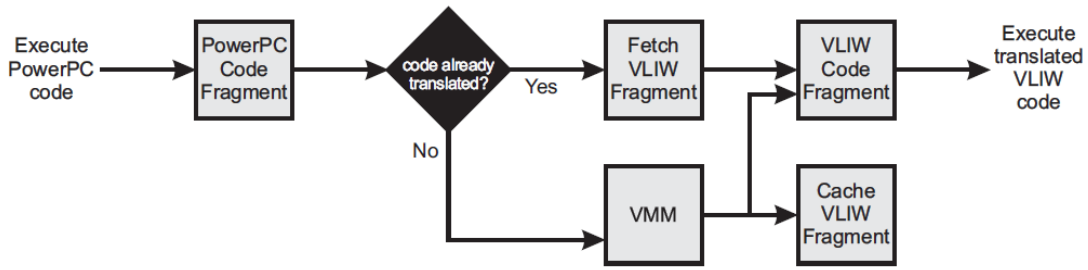
As unidades básicas de tradução são chamadas de árvores, que correspondem a sequências de códigos. Elas têm somente um ponto de entrada, e múltiplos pontos de saída. O fim da geração de uma árvore ocorre quando um salto para trás é encontrado (tipicamente são laços), ou quando limites de sub-rotinas são encontrados. As regiões de árvores são guardadas

na cache de tradução, uma área de memória acessível somente ao VMM, e não visível ao sistema sobre ele.

A próxima vez que um fragmento (uma árvore) de código é encontrado (o código já foi traduzido e está na cache de tradução), ele é diretamente executado, e nenhuma tradução é necessária (há um caso de exceção: quando uma parte do código já traduzido foi descartada da memória por restrições de espaço). Para facilitar esse processo, o processador DAISY tem uma instrução especial chamada LVIA (*Load VLIW Instruction Address*). Esta instrução é responsável por carregar instruções VLIW que correspondem a uma porção de códigos PowerPC que são previamente traduzidos. Esse processo é ilustrado na Figura 2.3.

Além do agendamento de instruções (*scheduling*), o sistema de TB DAISY também realiza uma série de otimizações, tais como agendamento ILP com especulação de dados e controle, desenrolamento de laços, análise de *alias*, eliminação de código morto, *load-store telescoping* entre outras (EBCIOGLU, 1997).

Figura 2.3 – Processo de tradução DAISY



Fonte: Beck; Lang; Carro (2012, p. 174).

2.2.2 Dynamo

O objetivo principal do Dynamo é a otimização: ele não traduz código de uma arquitetura nativa para uma arquitetura alvo, mas sim otimiza um sequência de instruções nativas. O sistema é implementado completamente em software. O Dynamo é completamente transparente, visto que não depende de qualquer ajuda do programador, como anotação ou instrumentação de código. Além disso, nenhum suporte do compilador, SO ou hardware é necessário. Dessa forma, binários nativos legados ou estaticamente otimizados podem ser acelerados pelo Dynamo. Um protótipo foi apresentado por (BALA et. al, 2000), rodando em um processador HP PA-8000 com sistema operacional HPUX. Como o sistema Dynamo foca

em otimizações em tempo de execução, ele tem a possibilidade de aplicar certas técnicas de otimização que seriam difíceis para um compilador estático explorar.

O Dynamo trabalha observando o comportamento da aplicação durante execução, e inicia a tradução somente quando um *hot spot* é encontrado. Pontos de entrada para um *hot spot* candidato são saltos tomados para trás, visto que a sequência dessa instrução provavelmente pertence a um laço. Os pontos de saída são saltos. Cada vez que o mesmo candidato é encontrado novamente, um contador é incrementado. Quando um limite é excedido, a geração de código inicia. Então o Dynamo gera uma versão otimizada dessa sequência de instruções e a salva numa cache de tradução (chamada de *fragment cache*). O Dynamo somente atua nesse momento, visto que interpretar código que não será otimizado iria reduzir a performance (o Dynamo por si só é um programa sendo executado no processador). Quando a mesma sequência é encontrada novamente, o código otimizado será buscado da *fragment cache* para ser executado diretamente no processador (o mecanismo de tradução não atua nesse momento). Quando a execução termina, o Dynamo reinicia todo processo novamente. Dessa forma, a *fragment cache* vai sendo pouco a pouco preenchida com código da aplicação otimizado, durante a execução da aplicação. O Dynamo se baseia na ideia de que pequenas porções de código são responsáveis pela maior parte de tempo de execução de uma aplicação (BALA et. al, 2000), e assim é possível se beneficiar do uso repetido de sequencias otimizadas encontradas na *fragment cache*.

O protótipo mostrou que, utilizando Dynamo, os ganhos médios de performance utilizando o conjunto de testes SpecInt95 compilado com –O2 é comparável com o mesmo benchmark compilado utilizando –O4 (que inclui otimizações específicas ao processador) sem o uso do Dynamo. As otimizações alcançadas pelo sistema são baseadas em redução de redundância, tais quais eliminações de saltos e atribuições e remoções de *load*. Outras otimizações tais quais propagação de cópias e constantes, *strength reduction*, *loop invariant code motion*, e desenrolamento de laços também são feitos (BALA et. al, 2000).

O Dynamo também procura manter a ocupação da memória de tradução tão baixa quanto possível, pois há a preocupação em manter os fragmentos na cache e no TLB (*Translation Lookaside Buffer*). Dessa forma, existe um mecanismo de limpeza que atua quando o espaço em memória se torna grande o suficiente a ponto de apresentar uma queda potencial de performance.

O Dynamo suporta manipulação de sinais. Sinais assíncronos (tais como uma interrupção de teclado) são tratados de forma distinta dos síncronos (e.g.: *segmentation faults*). Quando um tipo assíncrono chega, é colocado em uma fila aguardando até que o

fragmento que está sendo executado termine. No caso de um sinal síncrono, no entanto, o tratamento é mais complexo, pois o sinal não pode ser postergado. Dessa fora, foi implementado dois tipos de otimização de código: conservativa e agressiva. A conservativa é usada quando o tratamento de sinais síncronos é necessário. Embora ela não apresente tantos ganhos em desempenho quanto à agressiva, a abordagem conservativa permite que contextos precisos sejam construídos, de forma que sinais síncronos sejam corretamente manipulados.

2.2.3 Transmeta Crusoe

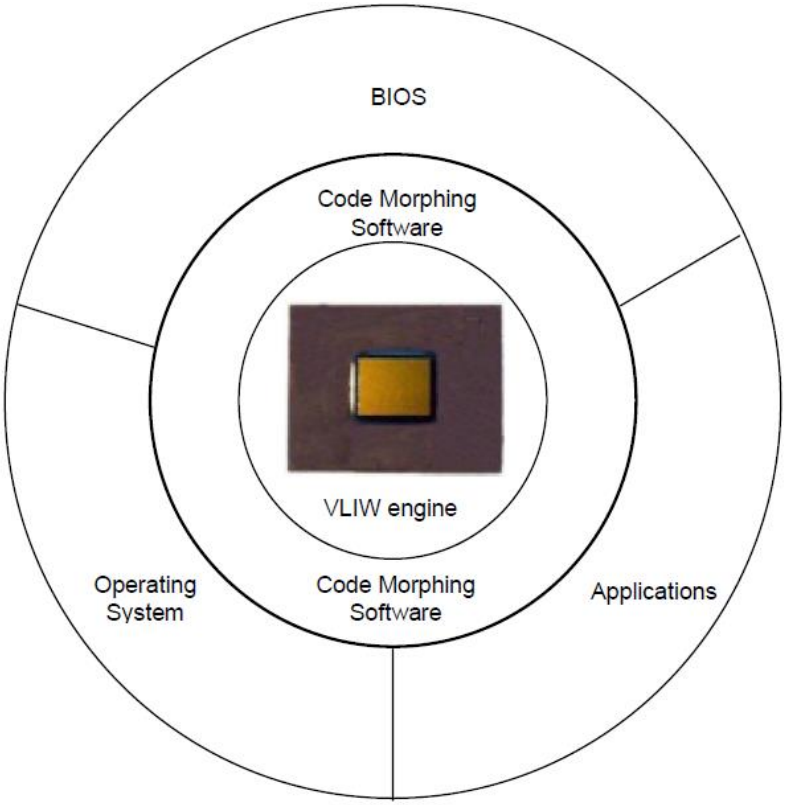
Desenvolvido pela empresa Transmeta, o Crusoe é um processador VLIW (*Very Large Instruction Word*) que executa a arquitetura x86 em nível de sistema e com baixo consumo de energia, através de uma camada de software chamada CMS (*Code Morphing Software*). A camada de software CMS pode ser vista na Figura 2.4.

Segundo (DEHNERT, et al., 2003), os desafios que o CMS visa superar frente a outros sistemas de tradução binária são:

- implementar fielmente a arquitetura x86: todo conjunto de instruções (inclusive mapeamento de I/O em memória), registradores da arquitetura, e o comportamento completo de exceções;
- não realizar suposições sobre o sistema operacional a ser utilizado e não depender de qualquer informação ou ajuda do sistema. É uma implementação em nível de sistema, não de aplicação, podendo assim até executar código da BIOS. Toda tradução é realizada *on-line*, conforme a arquitetura VLIW executa;
- fornecer desempenho robusto para uma ampla gama de sistemas e aplicações, desde jogos e processamento de mídias até programas para escritório ou servidores e lidar com dificuldades como por exemplo códigos auto modificáveis.

O CMS realiza a interpretação, tradução binária dinâmica, otimização e execução do sistema. Cada instrução no processador VLIW (chamada de molécula) pode prover entre duas a quatro operações do tipo RISC (chamadas de átomos) para as unidades funcionais (FUs). São cinco unidades funcionais: duas ULAs, uma unidade de ponto flutuante, uma unidade de memória e uma unidade para tratamento de saltos. Além disso, o processador tem um conjunto de 64 registradores de propósito geral e 32 registradores de ponto flutuante.

Figura 2.4 – A camada de software CMS entre o software x86 e o processador Crusoe



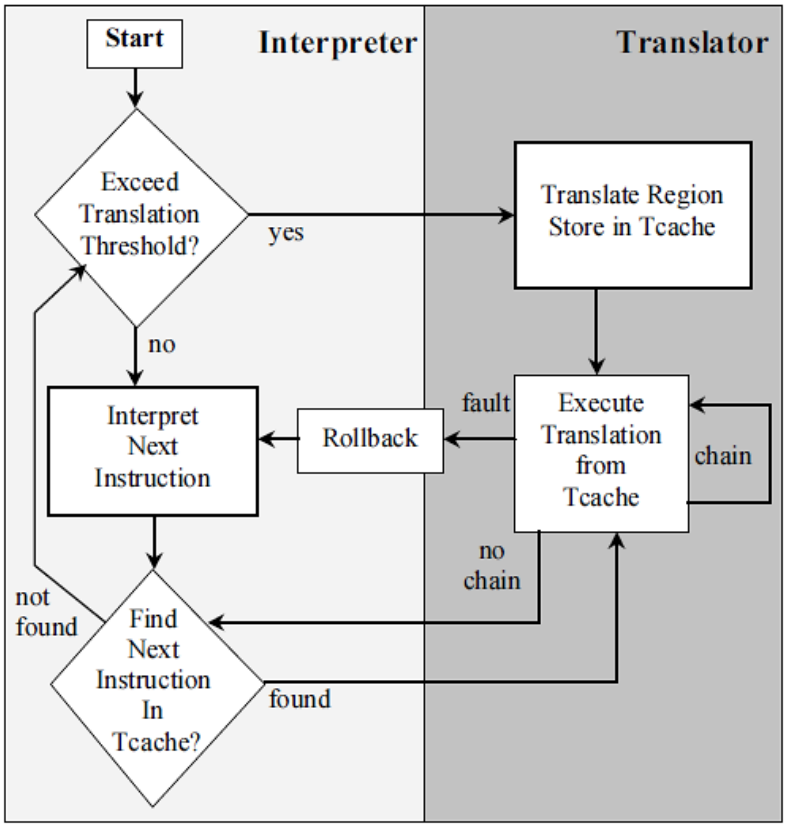
Fonte: Klaiber (2000 p. 7)

O papel do tradutor é decodificar instruções x86, selecionar regiões para tradução, analisar o código e gerar o código nativo VLIW, agendar e otimizar essas instruções. Todas essas tarefas fazem do tradutor o componente mais complexo do CMS. O otimizador, além de realizar otimizações específicas da arquitetura, realiza o agendamento do código VLIW em um *trace* com um único ponto de entrada e múltiplas saídas (para não limitar a blocos básicos). O sistema de execução é responsável pelo tratamento de dispositivos, interrupções, exceções, gerenciamento de potência e *garbage collector* da cache de tradução.

A Figura 2.5 mostra o processo de tradução. Enquanto as instruções x86 são interpretadas, são colhidos dados sobre a frequência de execução, direção de saltos e mapeamento em memória de I/O. Quando um limiar é atingido, o tradutor entra em ação, produzindo código VLIW nativo para aquela sequência x86. A sequência traduzida é guardada em uma cache de tradução e será reutilizada da próxima vez que a mesma sequência é encontrada, a não ser que essa entrada na cache de tradução seja invalidada por alguma razão. A saída do salto ao final de cada bloco traduzido chama uma rotina de procura (saída representada pela seta "*no chain*" no fluxograma), que pode então transferir o controle para um novo bloco traduzido ou então para o interpretador, que então poderá continuar a tradução

do código x86. Se o controle é transferido para outra porção de código já traduzido, a operação de salto é modificada para ir diretamente para lá, num processo chamado de *chaining*. Dessa forma, regiões executadas frequentemente serão executadas diretamente no processador VLIW, evitando os atrasos causados pelas instruções de salto.

Figura 2.5 – Controle de fluxo típico do CMS



Fonte: Dehnert (2003, p. 2).

O CMS também suporta especulação no sentido de fazer especulações durante a tradução, tais como memory disambiguation. No entanto, se tais especulações se provam falsas durante a execução, estas precisam ser tratadas corretamente, senão seriam gerados resultados incorretos. Para tal, o conjunto de registradores é duplicado, de forma que as instruções VLIW (átomos) atuam somente na cópia de conjunto de registradores atuais. O *commit* somente acontecerá quando a execução dessa porção de código traduzido chegar ao fim. Se alguma condição de exceção ocorrer como, por exemplo, alguma falha de especulação durante tradução, o sistema de execução é responsável por reverter (*rollback*) o contexto para o último ponto realizado um *commit* (as operações de *commit* e *rollback* também são aplicadas a operações de memória). Então, iniciando novamente desse ponto, o CMS inicia a

interpretação de instruções x86 correspondentes àquela exceção, executando o código original em ordem e tratando exceções quando necessário. Quando exceções ocorrem repetidamente para uma porção de código traduzido, o CMS gera uma tradução mais conservativa, tentado assim diminuir o número de exceções.

2.2.4 Vest

O sistema VEST visa traduzir binários em VAX e Ultrix MIPS para execução em computadores Alpha AXP. Além de garantir reutilização binária, o mecanismo também mostra ganhos de desempenho sobre a execução nativa. Os autores (SITES et. al, 2009) trabalharam na hipótese de usar um interpretador, mas desistiram quando perceberam que o desempenho seria muito pobre. Também vislumbraram que uma implementação utilizando microcódigo seria inconsistente com o modelo Alpha RISC.

O processo de transformação é estático e completamente automático. Pode reproduzir o comportamento de instruções complexas, atomicidade (para execução de aplicações *multithread*), e também exceções aritméticas e tratamento de erros. Se por alguma razão a tradução não for possível, uma mensagem explicita de erro é gerada, com detalhes sobre o que aconteceu.

A tradução estática de um binário VAX envolve duas fases: análise do código VAX e a tradução em si. A imagem traduzida é executada com assistência de um ambiente especial. O processador Alpha tem mais registradores que o VAX, portanto mapeamento de registradores não é considerado um problema. O Alpha tem registradores separados para operações de inteiros e de ponto flutuante, enquanto no VAX não há tal distinção. Dessa forma, o mapeamento de registradores depende do tipo de operação. Por exemplo, o R1 de uma instrução VAX pode ser mapeado para um registrador inteiro ou de ponto flutuante, dependendo de qual operação essa instrução realiza. Os bits de condição do VAX também são mapeados em registradores Alpha. Além disso, dependendo da operação, há um compromisso entre desempenho e precisão. Exemplificando, é possível emular por software um número de mantissa de 56 bits declarado pelo VAX em um Alpha, ou utilizar uma mantissa de 53 bits que é suportado nativamente em hardware pelo Alpha. No último caso, a execução será mais rápida, mas irá perder precisão. Contudo, há imagens que não podem ser traduzidas, tais quais as que apresentam casos específicos de tratamento de exceções, uso de serviços de sistema não documentados e software que dependa de exato gerenciamento de memória para executar de forma adequada.

Como já mencionado, o código fonte pode também ser de um processador Ultrix MIPS. Esse processo é mais simples que uma tradução de VAX, visto que ambas arquiteturas nativa e alvo são máquinas RISC. Dessa forma, para muitas instruções a tradução é direta, ou seja, uma instrução Ultrix MIPS gera uma instrução VAX (*one-to-one*). O processo de tradução segue dois passos básicos: o programa é primeiramente analisado, um grafo é gerado e então, o gerador de código é chamado. O gerador de código é também responsável pelo mapeamento de registradores e processamento de blocos básicos. Como não há registradores suficientes no Alpha para mapeamento direto, os registradores Ultrix MIPS mais usados são mapeados diretamente para registradores Alpha, enquanto os demais registradores devem ser salvos da memória para o banco de registradores e vice-versa. Algumas classes de códigos não podem ser traduzidas, tais quais aquelas aplicações que usam opcodes privilegiados ou chamadas de sistema.

Algumas otimizações também são feitas. Por exemplo, a cada chamada de subrotina, a ferramenta utiliza um algoritmo de reconhecimento de padrões de forma a descobrir se uma subrotina corresponde a uma localizada em uma biblioteca (por exemplo, strcpy). Se for encontrada, a chamada é substituída por uma rotina otimizada para ser executada no Alpha.

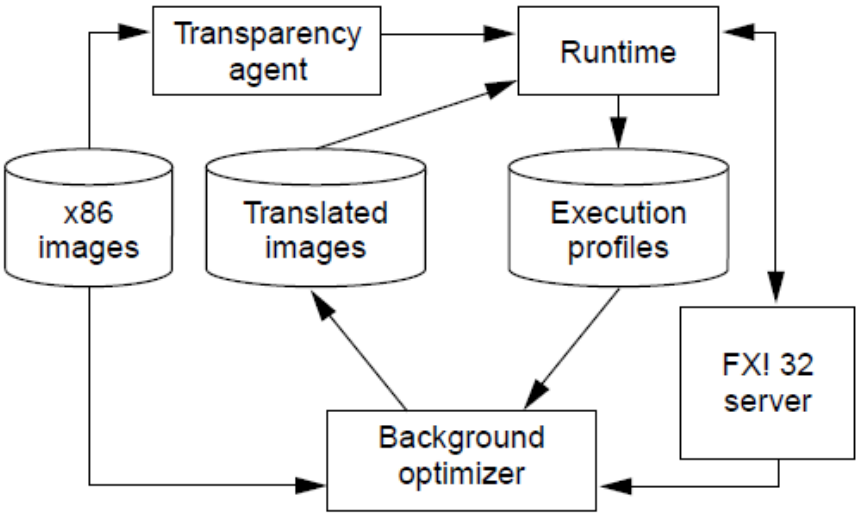
2.2.5 FX!32

O principal objetivo do FX!32 é permitir a execução transparente de aplicações nativas x86 Win32 em sistemas Alpha. Embora seja transparente para o usuário, o sistema oferece uma interface gráfica de forma que o usuário possa monitorar e gerenciar recursos. Por exemplo, a interface informa quais partes são mais executadas, e quais não são importantes. O sistema é mostrado na Figura 2.6.

A primeira vez que a aplicação x86 é executada, ela é somente interpretada: o FX!32 não tem qualquer conhecimento da aplicação. Contudo, junto com essa interpretação é gerado um perfil de execução. Então, através de um otimizador em segundo plano, é feita a tradução do código para instruções Alpha, utilizando o perfil de execução. O tradutor atua em unidades maiores que blocos básicos. De acordo com (HOOKWAY, 1997), a granularidade da tradução se aproxima do tamanho e estrutura de uma rotina. Dessa forma, na próxima vez que a mesma aplicação é executada, é utilizado o código Alpha ao invés do x86. O processo de geração de código Alpha é repetido várias vezes até que haja uma estabilização no perfil, e assim ganhos suficientes de desempenho sejam alcançados. De acordo com os autores, isso ocorre após duas ou três interações, indicando que quase todas rotinas foram traduzidas.

Então, o perfil dessa imagem é retirado da lista do otimizador, visto que não é mais necessário realizar qualquer otimização. Como a otimização é feita após várias interações, a primeira execução de uma aplicação será mais lenta. Nas próximas, a maior parte do código frequentemente executado já está traduzido para instruções Alpha, e a velocidade de execução será maior. Os autores afirmam que, após a tradução do código, são alcançados ganhos de 10 vezes em desempenho comparados com a simples interpretação. Essas partes de código traduzidas permanecem em um banco de dados do sistema FX!32, como DLLs (*Dynamic-Link Library*) Windows, e carregadas pelo sistema operacional nativo Alpha Windows na próxima vez que a aplicação x86 é executada. Como a execução de códigos nativo e não traduzidos coexistem, uma série de transformações deve ser feita para gerenciar as chamadas entre rotinas com código Alpha nativo e uma parte de código traduzido ou emulado. Por exemplo, a forma como uma rotina é chamada é diferente comparando sistemas x86 e Alpha (o primeiro usa a pilha para passagem de parâmetros, enquanto o Alpha utiliza registradores).

Figura 2.6 – Sistema FX!32



Fonte: Chernoff (1998, p. 4).

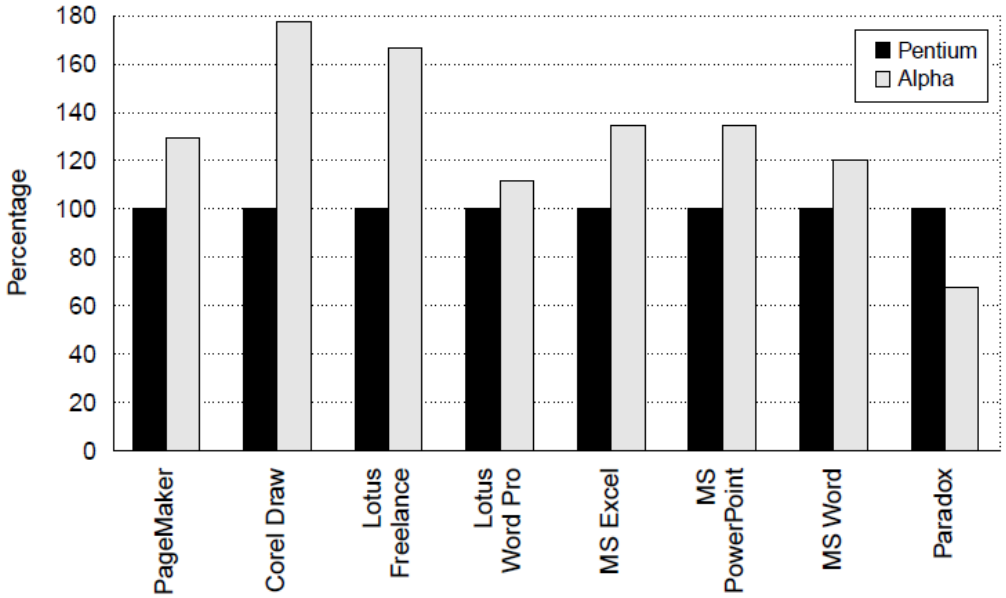
O FX!32 Server é responsável por coordenar a interface e ações de ambos interpretador e otimizador. Certos parâmetros do servidor podem ser configurados pelo usuário. Após um programa x86 terminar sua execução e ser descarregado, o servidor processa seu perfil de execução, juntamente com cada perfil anterior, podendo então invocar o tradutor. Em certos casos, partes do código que não foram analisados antes precisarão de otimizações adicionais.

A inicialização de um aplicação x86 é responsabilidade do *Transparency Agent*. A inicialização de uma aplicação Windows NT sempre resulta em uma chamada à função *CreateProcess*. O *Tranparency Agent* intercepta todas chamadas dessa função, examinado cada imagem que está para ser executada e quando verifica que uma imagem x86, ele invoca o FX!32 *Runtime* para executar esta imagem.

O FX!32 *Runtime* é responsável pela execução transparente, contendo um emulador que implementa o conjunto completo de instruções x86 em modo de usuário.

A Figura 2.7 mostra que o FX!32 em um processador Alpha rodando a 500 MHz mostrou desempenho superior em média sobre um processador Pentium rodando a 200MHz.

Figura 2.7 – Execução Pentium versus Alpha FX!32



Fonte: Chernoff (1998, p. 7)

2.2.6 Godson

O processador Godson3 (HU et. Al, 2009) tem o mesmo objetivo que o Transmeta Crusoe se considerarmos a arquitetura nativa: através de uma camada de software implementada através do emulador baseado no QEMU (BELLARD, 2005), ele converte instruções x86 para MIPS. No entanto ele usa uma estratégia diferente para otimizar o programa em execução. Godson3 é uma arquitetura multicore escalável, que utiliza um misto entre um NOC (*Network On-Chip*) e um sistema *crossbar* para sua infraestrutura de comunicação. Dessa forma, até 64 cores são suportados. Em muitos casos, traduzir uma

instrução x86 requer dezenas de instruções MIPS por causa das diferenças entre as ISAs x86 e MIPS. O tradutor binário do Godson-3 facilita a tradução do binário x86 para binário MIPS com mínimo suporte de hardware. Para conseguir isso, o Godson-3 define novas instruções e ambientes de execução através da UDI (*User-Defined Interface*) MIPS para preencher a lacuna entre as ISAs x86 e MIPS. São definidas e implementadas novas instruções no formato MIPS para funções que estão na ISA x86, mas não na ISA MIPS64.

O tradutor binário do Godson-3 é construído no topo do sistema operacional Linux. O sistema operacional Linux baseado na ISA MIPS do Godson foi melhorado para proporcionar chamadas de sistema compatível com Linux do x86, e assim melhorar a eficiência do tradutor binário. O tradutor binário foi implementado em nível de processos para executar aplicações x86. Os autores sugerem que ele também poderia ser implementado em nível de sistema para conseguir compatibilidade binária em nível de ISA. Além de implementar um novo interpretador e tradutor sob o framework do QEMU, sua representação intermediária foi modificada para permitir otimizações. Como outros sistemas de tradução binária tradicionais, a VMM (Virtual Machne Monitor) do Godson-3 inicialmente interpreta instruções x86 no processador MIPS e monitora o comportamento enquanto realiza a interpretação. Quando a VMM encontra *hot spots*, traduz o código x86 para código MIPS para execução e otimiza eles em diferentes níveis de sistema conforme a sua frequência de execução. O processador multicore Godson-3 também pode realizar otimizações paralelas para as partes mais frequentemente executadas (*very hot spots*). Experimentos realizados mostraram que o suporte em hardware do Gordon-3 acelerou em 5x a tradução via QEMU não modificado, e na média a execução de um código nativo x86 no QEMU alcançou 70% a performance de um código nativo MIPS.

2.2.7 Apple Rosetta

O Rosetta (APPLE, 2006) foi usado em sistemas Apple para manter compatibilidade entre o PowerPC (que era utilizado anteriormente em computadores Apple) e x86 empregados nas gerações futuras. Ele atua no nível de aplicações com o objetivo único de manter compatibilidade binária. O Rosetta pode traduzir código nativo das ISAs G3, G4 e AltiVec do PowerPC para x86, ainda que com alguns problemas de compatibilidade: algumas aplicações não podem ser traduzidas. O processo de tradução insere um *overhead* significativo, o que é percebido principalmente quando executadas aplicações que consomem muita CPU. O sistema operacional é responsável por chamar a VMM quando detecta que uma aplicação não

é compatível com a ISA x86. Dessa forma, o processo de tradução é transparente para o usuário.

3 ARQUITETURAS UTILIZADAS

A seguir é feito um breve resumo sobre as arquiteturas utilizadas nesse trabalho, salientando as principais caraterísticas que impactam no processo de tradução entre as diversas ISAs, como registradores, formato das instruções, modos de endereçamento e instruções específicas da arquitetura.

3.1 ARM

O primeiro processador ARM foi desenvolvido em 1985 pela Acorn Computers Limited em Cambridge, Inglaterra. Nesse época ARM derivava de Acorn RISC Machine, até a formação da Advanced RISC Machines Limited em 1990.

A arquitetura ARM é classificada, em sua essência, como uma arquitetura RISC, pois incorpora como características:

- extenso número de registradores uniformes;
- arquitetura load/store, na qual os acessos à memória são feitos exclusivamente através de operações sobre registradores;
- modos de endereçamento simples, com endereços de load/store determinados somente pelo conteúdo de registradores e campos de instrução.

Algumas melhorias foram acrescentadas à arquitetura ARM em relação a uma máquina RISC convencional, visando alcançar melhor relação entre desempenho, tamanho de programa, consumo de energia e tamanho de silício do processador (ARM, DDI0406C):

- instruções que combinam deslocamentos (*shift*) com operações lógicas ou aritméticas;
- modos de endereçamento com auto-incremento e auto-decremento para otimizar laços de programas;
- instruções de load e store múltiplos para maximizar *throughput* de dados;
- execução condicional de instruções para maximizar *throughput* de execução.

A arquitetura ARM é composta basicamente por dois conjuntos de instruções: ARM e Thumb. O conjunto de instruções ARM é composto por instruções de 32 bits enquanto o conjunto de instruções Thumb é composto por instruções de 16 bits e que apresentam um subconjunto das funcionalidades do conjunto de instruções ARM. As instruções Thumb aumentam significativamente a densidade de código ao custo de redução de desempenho. Os

processadores ARM podem trocar o modo durante a execução do programa entre ARM ou Thumb através da instrução BX. Dessa forma, é possível que o programador escolha a forma de execução de determinados trechos de código, privilegiando área ou desempenho.

A evolução da arquitetura nos últimos 24 anos introduziu uma série de características e extensões que não se encaixam necessariamente ao conceito de uma arquitetura RISC ideal (ARM DAI0235C, 2010). A partir da revisão ARMv6T2, foi introduzida a tecnologia Thumb-2, que acrescenta ao conjunto de instruções Thumb uma série de instruções de 32 bits. Essas novas instruções permitem que os códigos em Thumb-2 alcancem desempenho similar ao dos códigos em ARM, com maior densidade que ao antecessor Thumb [ARM, DDI0406C]. A partir da revisão ARMv5TEJ, a arquitetura prevê implementação da extensão Jazelle. A extensão Jazelle provê suporte na arquitetura para aceleração em hardware da execução de bytecodes por uma Máquina Virtual Java (JVM) específica. Na implementação mais simples, chamada de implementação trivial da extensão Jazelle, o processador não provê hardware para aceleração da execução, mas sim utiliza rotinas de software para executar todos bytecodes. Todas as implementações que proveem aceleração em hardware são chamadas de implementações não-triviais da extensão Jazelle. As extensões Advanced SIMD e VFP são duas versões opcionais da versão ARMv7. A extensão Advanced SIMD promove operações do tipo Single Instruction Multiple Data (SIMD), tanto com inteiros ou ponto flutuante de precisão simples. A extensão VFP (Virtual Floating Point), promove operações de ponto flutuante de precisão simples e de dupla precisão. A Tabela 3.1 mostra a evolução da arquitetura ARM.

Tabela 3.1 – Revisões da arquitetura ARM

Versão	*Características*
ARMv4	Inclui somente o conjunto de instruções ARM.
ARMv4T	Adiciona o conjunto de instruções Thumb.
ARMv5T	Melhora a co-execução de instruções Thumb e ARM. Adiciona instruções CLZ e BKPT
ARMv5TE	Melhora o suporte aritmético para algoritmos de processamento digital de sinais (DSP). Adiciona as instruções: *Adds Preload Data* (PLD), *Load Register Dual* (LDRD), *Store Register Dual* (STRD), e tranferência de 64-bit entre registrador e coprocessador (MCRR, MRRC).
ARMv6	Adiciona várias instruções ao conjunto ARM.
ARMv6K	Adiciona instruções para suporte a multiprocessamento.

ARMv6T2	Introduz a tecnologia Thumb-2
ARMv7-A	Perfil de aplicação: - implementa a tradicional arquitetura ARM com múltiplos modos. - suporte a *Virtual Memory System Architecture* (VMSA) baseado uma MMU - suporte a instruções ARM e Thumb
ARMv7-R	Perfil de tempo real: - implementa a tradicional arquitetura ARM com múltiplos modos. - suporta uma arquitetura com memória protegida baseada em MPU (pode ser chamado de ARMv7 PMSA) - suporte a instruções ARM e Thumb
ARMv7-M	Perfil de microcontrolador: - implementa modelo de programação destinado ao tratamento de interrupções com baixa latência, com empilhamento de registradores por hardware e suporte para escrita de funções de tratamento de interrupções (*interrrupt handlers*) em linguagens de alto nível. - implementa uma variação do ARMv7 PMSA - suporta uma variante do conjunto de instruções Thumb
ARMv8	Introduz um novo conjunto de instruções de 64 bits

3.1.1 Banco de Registradores

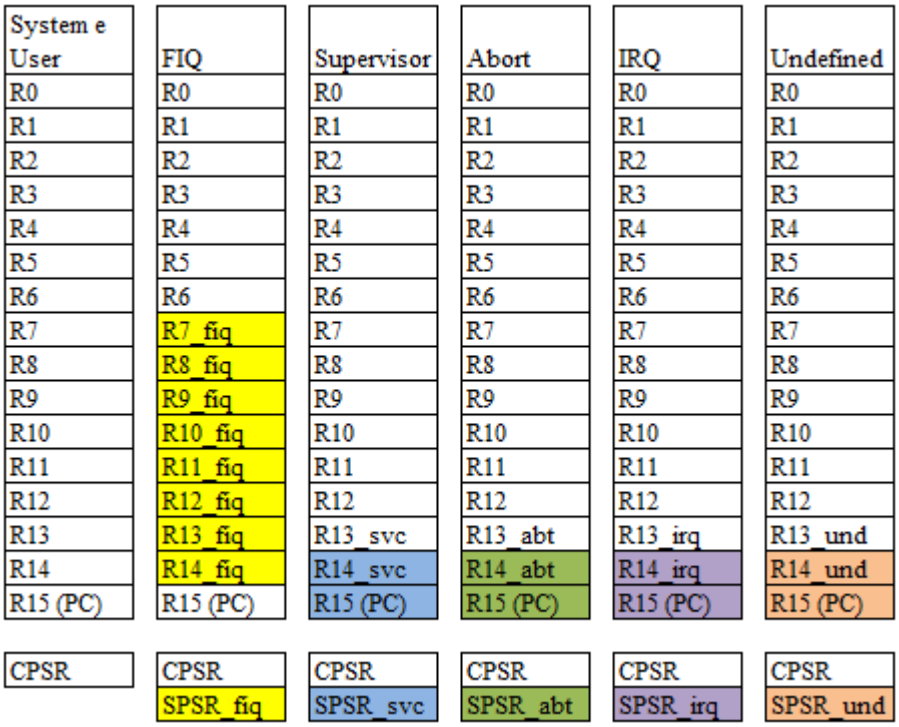
Por se tratar de uma arquitetura *load/store*, para realizar qualquer instrução de processamento de dados os dados devem ser primeiramente movidos da memória para registradores, as operações realizadas sobre registradores e então os dados movidos novamente para memória.

A arquitetura ARM possui seis modos operacionais básicos. O código das aplicações normalmente irá executar no modo de usuário (*User Mode*) com acesso aos registradores R0 a R15 e ao CPSR. No entanto, na ocorrência de uma interrupção ou chamada de sistema, o processador vai mudar de modo. Quando isto ocorre, os registradores R0 a R12 e R15 (PC) permanecem os mesmos, mas os registradores R13 (SP) e R14 (LR) são substituídos por outros registradores próprios do novo modo. O *System Mode* é um modo privilegiado do *User Mode* para o sistema operacional, compartilhando os mesmos registradores desse modo.

O modo *Suspervisor Mode* é um modo protegido para o sistema operacional. O *Abort Mode* é utilizado para sinalizar ao sistema operacional que um valor associado a um acesso em memória é invalido. O *Undefined Mode* entra quando o processador não encontra uma instrução desconhecida para ele e para qualquer coprocessador do sistema. Os modos FIQ e

IRQ são utilizados para o tratamento de interrupções. Interrupções do tipo FIQ têm maior prioridade e menor latência de atendimento que interrupções do tipo IRQ. Além disso, o modo FIQ, além dos registradores R13 e R14 próprios, possui os registradores R7 a R12 exclusivos. Com um maior número de registradores, o modo FIQ permite uma troca mais rápida de contexto, pois menos registradores precisam ser salvos na pilha. A Figura 3.1 mostra os diversos registradores associados aos modos de operação.

Figura 3.1 – Banco de registradores associados aos modos de operação



O banco central de registradores é composto por 16 registradores (R0 a R15), todos com largura de 32 bits. Os registradores R0 a R12 são registradores de uso geral, e não possuem função específica. O registrador R13 é o *stack pointer*. O registrador R14 é o *link register*, usado para armazenar o endereço (PC) de retorno das funções. R15 é o *program counter* (PC).

Além dos 16 registradores, existe um registrador especial chamado de CPSR (*Current Program Status Register*), ilustrado na Figura 3.2.

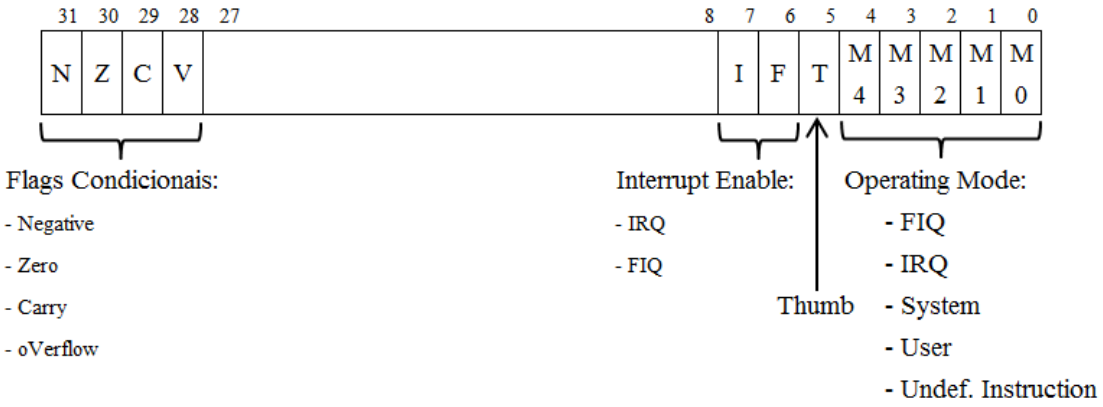
Os quatro bits mais significativos do registrador CPSR contêm flags que são escritos pela CPU, e que são cópias dos *flags* de status da ULA quando a instrução executada possui o bit “S” (status bit) em “1”. Os *flags* são: N (resultado negativo), Z (resultado zero), C (*carry out*) e V (*overflow*).

Os bits I e F do registrador CPSR permitem habilitar e desabilitar as interrupções do tipo IRQ e FIQ respectivamente.

O bit T do registrador CPSR indica qual conjunto de instruções está sendo executado: ARM ou Thumb. Já os cinco últimos bits menos significativos indicam em qual modo que o processador está operando.

O registrador SPSR é usado para cópia do registrador CPSR. Assim, na troca entre modos, o conteúdo do CPSR é restaurado a partir do SPSR.

Figura 3.2 – Registrador CPSR



3.1.2 Modos de Endereçamento

Nas instruções de *load/store*, o endereço de memória é formado adicionando-se dois componentes assim como outros processadores:

Endereço efetivo = conteúdo de endereço base + offset.

O endereço base pode ser qualquer registrador de propósito geral, incluindo o PC. Quando o PC é usado, temos endereçamento relativo ao PC, o que facilita a codificação de código independente de posição.

Embora o modo de endereçamento possa parecer igual a outros processadores, ele oferece grande flexibilidade pela forma que o offset pode ser especificado, e pelo modo que o registrador base e *offset* são computados para formar o endereço efetivo.

A especificação do *offset* pode ser feita de três formas:

- Valor imediato: o *offset* pode ser uma constante sem sinal de 8 ou 12 bits. Em outras arquiteturas, esse valor pode ser com sinal. No entanto, no ARM esse valor pode ser adicionado ou subtraído do conteúdo do registrador base para

formar o endereço efetivo (um bit na instrução define se o *offset* deve ser somado ou subtraído). Para instruções de transferência de byte sem sinal e *word* (32 bits), esse valor pode ser de 12 bits. Para instruções de byte com sinal ou *halfword* (16 bits), esse valor é de 8 bits.

- Valor em registrador: nesse modo o *offset* é o conteúdo de um registrador de propósito geral. Ou seja, endereço é dado pela soma de dois registradores.
- Valor escalonado em registrador: esse modo é similar ao anterior, exceto que o conteúdo do registrador é deslocado para esquerda ou direita antes de ser somado ao conteúdo do registrador base para formar o endereço.

O modo de valor escalonado em registrador é útil para acessar arrays. Por exemplo, se tivermos um *array* de *doubles* e o índice do *array* estiver no registrador de *offset*, basta deslocarmos em três posições o registrador de *offset* (e assim, multiplicando por oito) para acessarmos a nova posição do *array*.

Existem três modos que o registrador base e *offset* podem ser computados para formar o endereço efetivo

- Modo *offset*: esse é o modo básico de computação do endereço, onde o endereço é computado somando o valor do registrador base e o *offset*.
- Modo pré-indexado: nesse modo o endereço de memória é computado como no modo *offset*. No entanto, o endereço efetivo também é carregado no registrador base.
- Modo pós-indexado: o endereço efetivo é calculado da mesma forma que os anteriores. No entanto o endereço usado na transferência é o conteúdo do endereço base. O endereço efetivo calculado é então carregado no registrador base.

3.1.3 Formato de Instruções

O formato das instruções ARM não é tão simples quanto ao da arquitetura MIPS, pois possui uma grande variedade de formatos como pode ser visto na Figura 3.3. A maioria das instruções ARM pode opcionalmente atualizar os flags de status (N, Z, C e V). O bit S indica se o *flag* deve ser atualizado (S = 1) ou não (S = 0) pela instrução.

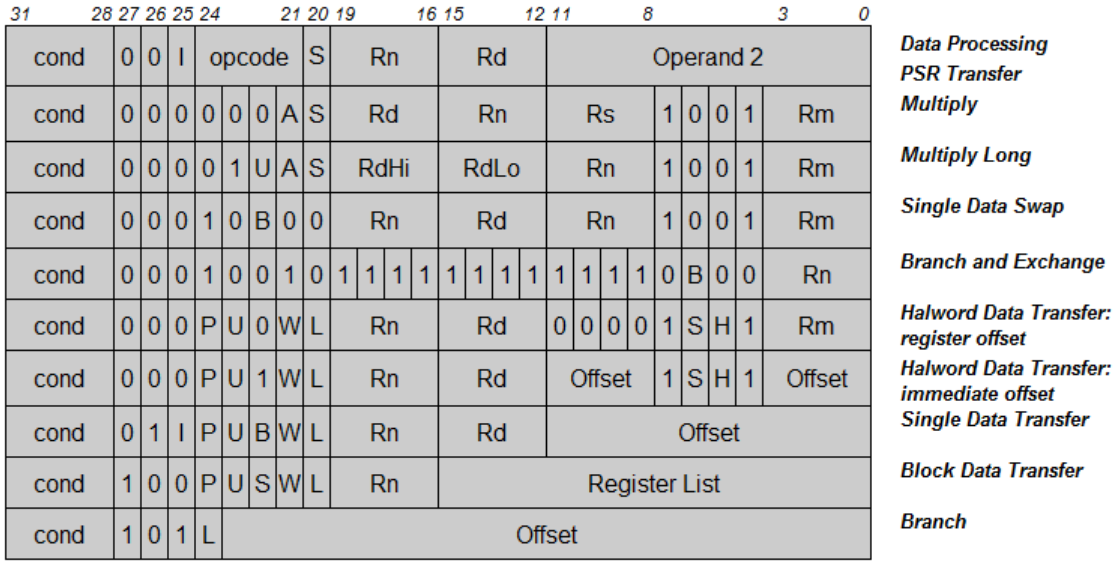
As instruções de *load/store* usam no mínimo dois formatos. O destino para *load* (origem para *store*) é dado pelo campo Rd. O registrador Rn e os 12 bits menos significativos

definem o modo de endereçamento. O bit L indica se a instrução é *load* (L=1) ou *store* (L=0). O bit B especifica se a operação de *load* é um byte ou palavra sem sinal. O bit P indica se o modo de operação é pré (P=1) ou pós (P=0) indexado, ou seja, se o *offset* é adicionado antes ou após a transferência. O bit W (*write-back*) indica se o endereço efetivo deve ser escrito (W=1) no registrador base. O bit U indica se o *offset* deve ser somado (U=1) ou subtraído (U=0). O bit I indica se o *offset* é um valor imediato (I=1) ou está contido em um registrador (nesse caso o campo *offset* define um registrador e seu escalonamento).

O ARM possui instruções para transferência de um bloco de registradores para memória (stm) e da memória para registradores (ldm). A Figura 3.3, mostra a codificação dessas instruções (*block data transfer*). Um campo de 16 bits define uma máscara que contém a lista de registradores que devem ser usados na transferência. Sendo assim pode-se, com apenas uma instrução, transferir de 1 até o total dos 16 registradores de propósito geral de/para a memória.

As instruções de *branch* podem utilizar um imediato com sinal de 24 bits como endereço destino. O bit L é utilizado para indicar se o endereço de retorno deve ser preservado no *link register*.

Figura 3.3 – Formato das instruções ARM



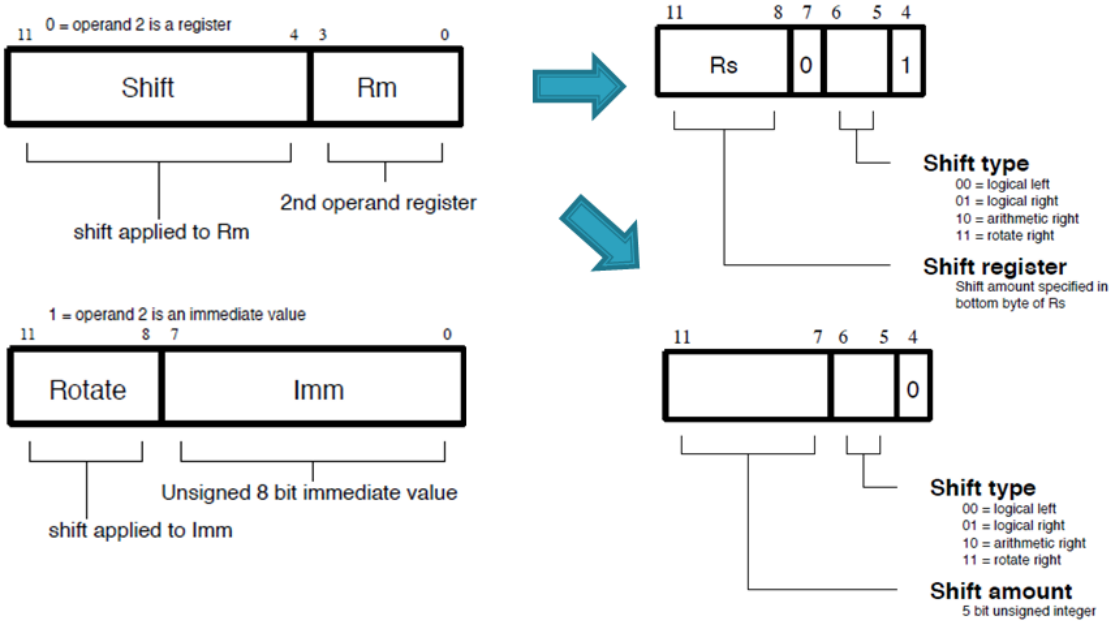
Nas instruções lógicas e aritméticas (*Data Processing*) o opcode especifica o tipo de operação como, por exemplo, add ou cmp. O campo Rd especifica o registrador destino, que recebe o resultado. Um dos operandos é o conteúdo do registrador Rn, o segundo operando é um operando complexo, que pode ser definido de várias maneiras. O bit I define se o segundo

operando é um valor imediato (I=1) ou um registrador (I=0). A Figura 3.4 mostra a complexidade desse segundo operando.

Se o segundo operando for um registrador (I=0), os bits 0 a 3 definem qual é esse registrador. O valor desse registrador pode, ainda, sofrer uma rotação ou deslocamento antes de ser usado como segundo operando. Se o bit 4 da instrução for 1, o número de bits a serem rotacionados ou deslocados é definido pelo registrador Rs. Mas se o bit 3 da instrução for 0, o valor do deslocamento ou rotação é um valor de 5 bits. Dois bits do campo de segundo operando definem o tipo de deslocamento ou rotação (deslocamento lógico à esquerda, deslocamento lógico à direita, deslocamento aritmético à direita ou rotação à direita).

Se o segundo operando for um imediato (I=1), o valor imediato de 8 bits pode ser rotacionado para direita por valores pares, até 32. O valor da rotação é definido pelos bits 8 a 11, multiplicado por dois. Dessa forma, muitas constantes comuns podem ser geradas, como por exemplo, potências de dois.

Figura 3.4 – Segundo operando ARM



Toda instrução ARM possui um campo de quatro bits chamado "*Condition Code*", que indica em que condição a instrução deve ser executada ou não baseada nos valores dos flags condicionais do registrador CPSR. Dessa forma, cada instrução ARM pode ser executada por uma das 16 condições diferentes, conforme pode ser visto na Tabela 3.2. A condição AL permite que a instrução seja executada incondicionalmente (sempre).

Tabela 3.2 – Códigos condicionais das instruções ARM

Código Condicional	*Sufixo Assembler*	*Flag*	*Significado*
0000	EQ	Z = 1	Igual
0001	NE	Z = 0	Diferente
0010	CS	C = 1	*unsigned* maior ou igual
0011	CC	C = 0	*unsigned* menor
0100	MI	N = 1	negativo
0101	PL	N = 0	positive ou zero
0110	VS	V = 1	overflow
0111	VC	V = 0	sem overflow
1000	HI	C = 1 e Z = 0	*unsigned* higher
1001	LS	C = 0 ou Z = 1	*unsigned* menor ou igual
1010	GE	N = V	maior ou igual
1011	LT	N != V	menor que
1100	GT	Z = 0 e N = V	maior que
1101	LE	Z = 1 or N != V	menor que ou igual
1110	AL		sempre
1111	reservado		

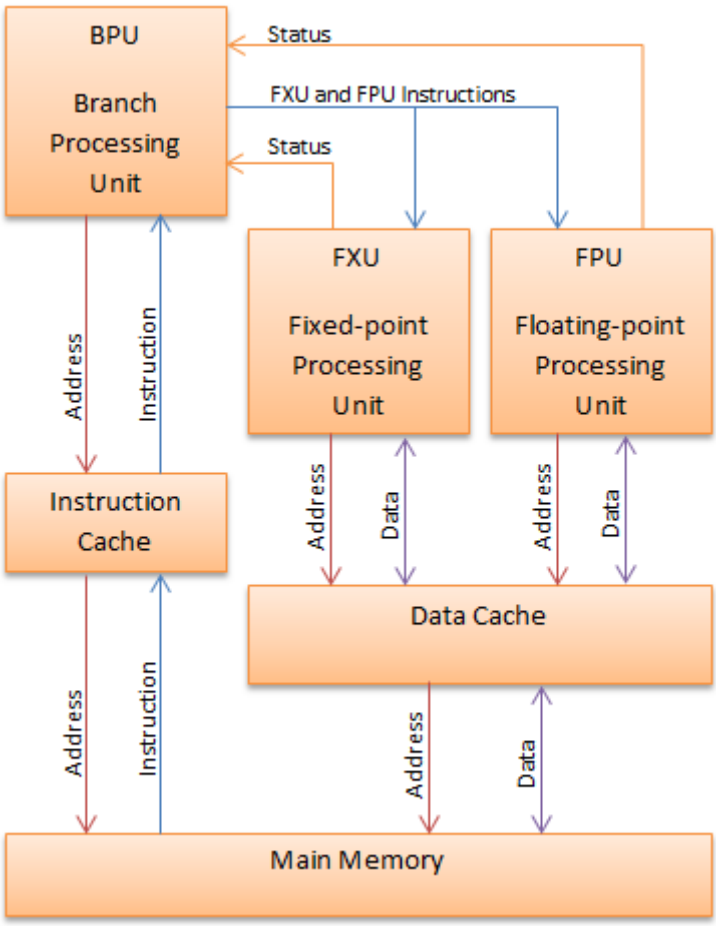
3.2 PowerPC

Muitos dos conceitos dessa arquitetura nasceram em 1975, em um protótipo de processador desenvolvido pela IBM. Uma versão comercial desse protótipo foi introduzida em 1986 e, quatro anos mais tarde, a IBM introduziu a família RS/6000 de processadores baseados em sua arquitetura POWER. Já em 1991, um grupo formado pela Motorola, IBM e Apple começou a desenvolver a arquitetura PowerPC utilizando a arquitetura IBM POWER como base. A arquitetura PowerPC tem muitas das instruções POWER, sendo que a primeira versão da arquitetura PowerPC, a PowerPC 601, apenas não implementava duas instruções da POWER.

O PowerPC define uma arquitetura de 64 bits que pode operar em dois modos: 64 bits e 32 bits. Uma visão geral de um processador PowerPC pode ser vista na Figura 3.5. Ele é composto por três unidades funcionais para facilitar implementações superescalares, são elas:

- *Branch Processing Unit* (BPU): é responsável por buscar as instruções e pela execução de *branches* e instruções relacionadas a *branches*. Além disso, a BPU é responsável pelo envio de instruções para as unidades de ponto fixo e ponto flutuante.
- *Fixed-point Processing Unit* (FXU): a FXU executa as funções de ponto fixo, e também calcula os endereços das instruções de *load* e *store* para a unidade de ponto flutuante.
- *Floating-point Processing Unit* (FPU): a FPU executa as instruções de ponto flutuante e também pelas instruções *load/store* sobre pontos flutuantes.

Figura 3.5 – Visão geral de um processador PowerPC



Fonte: Dandamudi (2005, p. 80).

As instruções do PowerPC podem ser divididas em três classes, dependendo de qual dessas unidades as executa: instruções de salto, instruções de ponto fixo, e instruções de ponto-flutuante. As instruções de ponto fixo operam sobre bytes (8 bits), *halfword* (16bits),

word (32 bits), e *doubleword* (64 bits). As instruções de ponto flutuante operam sobre números de ponto flutuante simples ou de dupla precisão.

3.2.1 Banco de Registradores

O PowerPC possui 32 registradores de propósito geral para números inteiros (GPR0 a GPR31) e 32 registradores de ponto-flutuante (FPR0 a FPR31), como mostrado na Figura 3.6. Além desses, existem cinco registradores especiais: CR, LR, CTR. FPSCR e XER. Os registradores de propósito geral, LR, XER, e *Count Register* possuem 32 ou 64 bits de largura, dependendo da implementação da arquitetura. Os demais registradores possuem tamanho fixo independente da implementação. O conjunto de registradores pode ser particionado em três grupos, cada qual associado a uma unidade de processamento.

A BPU (*Branch Processing Unit*) usa os seguintes registradores: *Conditional Register* (CR), *Link Register* (LR), e *Count Register* (CTR). Os dois últimos registradores são de 64 bits, enquanto o CR possui 32 bits. A FXU (*Fixed-point Unit*) utiliza os 32 registradores de propósito geral GPR0 a GPR31 e o *Fixed-point Exception Register* (XER). A FPU (*Fixed-point Unit*) utiliza os 32 registradores de ponto flutuante FPR0 a FPR31, o *Float-point Status Register* (FPSCR).

O registrador FPSCR possui dois conjuntos de bits: os bits 0 a 23 são usados para gravar o status das operações de ponto flutuante. Os demais bits são bits de controle que guardam qualquer exceção gerada por operações de ponto flutuante. Por exemplo, o bit 23 grava a exceção de operação inválida que ocorre quando um operando é inválido para determinada operação de ponto flutuante. A exceção de ponto flutuante de divisão por zero é capturada no bit 5.

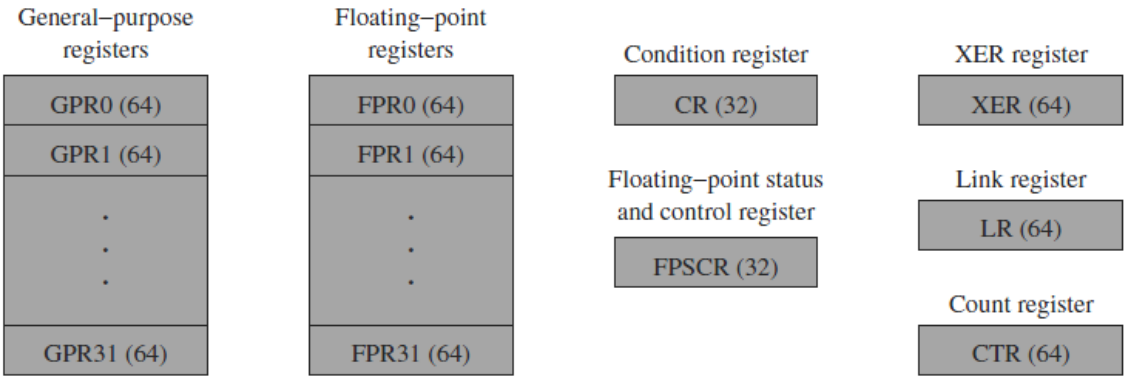
O registrador CR (*Conditional Register*) de 32 bits é dividido em oito campos de quatro bits (CR0 a CR7). O CR0 é usado para capturar o resultado de instruções de ponto fixo. Cada bit do CR0 determina um tipo de resultado: bit 0 indica que o resultado é negativo (*less than*, LT), bit 1 indica que o resultado é positivo (*greater than*, GT), bit 2 indica que o resultado é zero (*equal*, EQ) e o bit 3 indica *overflow* (*Sumary Overflow*, SO). Este último é uma cópia do XER, discutida a seguir. O campo CR1 é usado para capturar status de exceções de ponto flutuante, ele é uma cópia dos quatro bits menos significativos do registrador FPSCR. Os demais campos CR podem ser usados tanto por instruções inteiras ou de ponto flutuante para capturar condições LT, GT, EQ e SO. Existem instruções que realizam

operações lógicas em bits individuais do CR e instruções de salto que testam bits específicos do CR. Essas instruções podem ainda especificar qual dos oito campos utilizar.

O registrador LR (Link Register) é utilizado para armazenar o endereço de retorno de uma rotina. Chamadas de rotinas são implementadas por instruções de salto ou de salto condicional. Para essas instruções, o LR recebe o endereço absoluto da instrução imediatamente após o saldo.

O registrador CTR (*Count Register*) é utilizado para manter o contador de um laço. Instruções de salto podem especificar uma série de condições sob as quais o salto deve ocorrer. Por exemplo, um salto condicional pode decrementar o CTR e saltar somente se CTR = 0 ou CTR ≠ 0.

Figura 3.6 – Registradores do PowerPC



Fonte: Dandamudi (2005, p. 82).

O registrador XER (*Fixed-point Exception Register*) serve para dois objetivos. Os bits 0, 1 e 2 são usados para guardar *Summary Overflow* (SO), *Overflow* (OV) e *Carry* (CA). O bit OV salva o evento de overflow durante a execução de uma instrução. O bit SO guarda o mesmo evento do OV, mas ele mantém seu estado até que uma instrução especial é executada para limpá-lo. O bit CA é acertado por operações de soma e subtração, bem como instrução de deslocamento à direita.

Os bits 57 a 63 do registrador XER são usados como um contador de bytes de 7 bits, utilizados para definir o número de bytes a serem transferidos entre memória e registradores. Esse campo é utilizado pelas instruções lswx (*Load String Word Indexed*) e stswx (*Store String Word Indexed*). Utilizando somente uma instrução lswx pode-se carregar 128 bytes contíguos da memória nos 32 registradores de propósito geral. De forma similar, a transferência de registradores para memória pode ser feita através da instrução stswx.

3.2.2 Modos de Endereçamento

O PowerPC suporta dois modos básicos de endereçamento com variações. É possível especificar três registradores de propósito geral: RA, RB, e RD/RS em instruções load/store. Os registradores RA e RB são usados para calcular o endereço efetivo. O terceiro registrador é tratado tanto como o registrador destino "rd" em instruções de *load*, ou registrador origem RS em instruções de *store*. Estes modos são listados abaixo:

- Registrador indireto com imediato: nesse modo de endereçamento, as instruções contém um valor de imediato com sinal. O endereço efetivo é calculado adicionando esse valor ao conteúdo do registrador de propósito geral RA definido pela instrução. É possível especificar um "0" no lugar de RA. Nesse caso, o endereço efetivo é o valor imediato definido na instrução. Assim, é simples converter endereçamento indireto para endereçamento direto ao definir o campo de RA como "0". O valor constante do imediato pode ser tanto 14 bits ou 16 bits, dependendo do formato de instrução utilizado. É interessante notar que esse é o único modo de endereçamento suportado pela arquitetura MIPS.

  Endereço efetivo = conteúdo de RA (ou 0) + imediato

- Registrador indireto com imediato e atualização: é uma variação do modo anterior, onde, após seu cálculo, o endereço efetivo é carregado no registrador RA.

  Endereço efetivo = conteúdo de RA (ou 0) + imediato

  RA = Endereço efetivo

- Registrador indireto com indexação: as instruções que utilizam esse modo de endereçamento definem dois registradores de propósito geral, RA e RB. O endereço efetivo é calculado como sendo a soma desses dois registradores. Como no modo anterior, podemos definir "0" no campo de RA.

  Endereço efetivo = conteúdo de RA (ou 0) + conteúdo de RB

- Registrador indireto com indexação e atualização: é uma variação do modo anterior, onde, após seu cálculo, o endereço efetivo é carregado no registrador RA.

  Endereço efetivo = conteúdo de RA (ou 0) + conteúdo de RB

RA = Endereço efetivo

3.2.3 Formato das Instruções

Todas as instruções PowerPC são codificadas utilizando quatro bytes, com alinhamento de palavra (32 bits). Dessa forma, o processador pode ignorar os dois bits menos significativos do endereço de todas as instruções.

Uma amostra dos formatos de instruções utilizadas pelo PowerPC pode ser vista na Figura 3.7. Os bits 0 a 5 especificam o opcode primário, sendo que várias instruções também possuem um opcode estendido. Os demais bits são utilizados por diversos campos, dependendo do tipo de instrução. São apresentados apenas alguns formatos, visto que diferentemente do MIPS, o PowerPC possui inúmeros formatos de instruções.

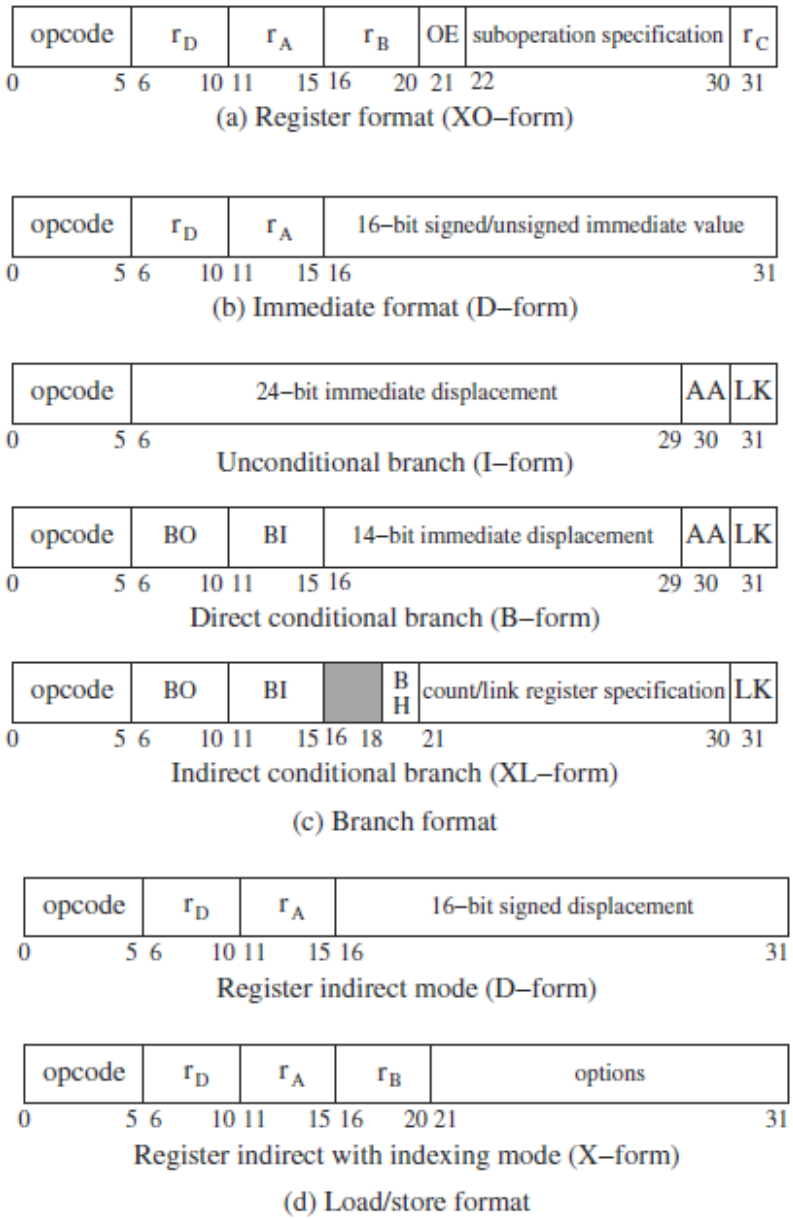
A maioria das instruções utiliza o formato XO, mostrado na Figura 3.7.a. Em instruções aritméticas, lógicas e outros tipos de instruções similares, os registradores RA e RB definem os operandos origem e RD define o registrador destino. O bit OE indica se os bits SO e OV do registrador XER são alterados (OE = 1) ou não (OE = 0). O bit "rc" especifica se os bits LT, GT e EQ, do campo CR0, devem ser alterados. A alteração ou não do bit CA do registrador XER é definida por instruções com opcodes distintos: por exemplo, existe a instrução addc que altera o flag CA e a instrução *add* que não altera esse flag.

O formato imediato mostrado na Figura 3.7.b é utilizado por instruções que utilizam um imediato como operando, como por exemplo a instrução addi.

O formato mostrado na Figura 3.7.c é utilizado por instruções de salto. O formato de salto incondicional (formato I) permite a especificação de um endereço de 24 bits. Essa mesma figura mostra os formatos utilizados para salto direto condicional (formato B) e salto indireto condicional (formato XL). O bit AA indica se o endereço é um endereço absoluto ou um endereço relativo ao PC. O bit LK é usado para guardar o endereço de retorno, de forma que a instrução de salto possa ser usada para chamas de rotinas. O operando BO (*branch option*) especifica em que condição o salto é executado, sendo que o operando BI (*branch input*) define qual bit do campo CR deve ser usada para o teste da condição. Exemplo: decrementa CTR e salta se CTR $\neq$ 0 e a condição é falsa. O campo de dois bits BH (*branch hint*) dá uma predição do salto: a instrução é um retorno de uma rotina, a instrução não é um retorno de rotina (o endereço alvo é provavelmente o mesmo endereço usado da última vez que o saldo foi executado) ou o endereço de salto não é previsível.

O formato para instruções de *load/store* é mostrado na Figura 3.7.d. O primeiro formato (formato D) é utilizado por instruções de *load/store* que utilizam endereçamento por registrador indireto com imediato. Essa instrução utiliza o registrador RA e um valor imediato de 16 bits para computar o endereço efetivo. O segundo formato (formato X) é utilizado por instruções de *load/store* que utilizam endereçamento por registrador indireto com indexação.

Figura 3.7 – Formato das instruções PowerPC



Fonte: Dandamudi (2005, p. 85).

3.3 MIPS

A arquitetura MIPS evoluiu durante o tempo, desde a ISA MIPS I até a ISA MIPS V. No final dos anos 1990, as arquiteturas MIPS foram divididas em duas arquiteturas básicas: MIPS32 para arquiteturas de 32 bits e MIPS64 para arquiteturas de 64 bits. A arquitetura MIPS32 é baseada na ISA MIPS II com adição de algumas instruções das ISAs MIPS III, IV e V. A arquitetura MIPS64 é baseada na arquitetura MIPS V.

O primeiro processador MIPS, o R2000, foi lançado em meados dos anos 1980. Uma versão melhorada, o R3000 tornou-se disponível três anos depois. Ambos os processadores eram implementações de 32 bits. O primeiro MIPS de 64 bits, o R4000, foi lançado no início dos anos 1990.

Como outros processadores RISC, o MIPS é baseado numa arquitetura *load/store*, na qual a maioria das instruções tem seus operandos em registradores.

3.3.1 Banco de Registradores

A arquitetura MIPS32 possui 32 registradores de propósito geral, um *Program Counter* (PC) e dois registradores de uso especial. Todos registradores possuem 32 bits de largura. Em *assembly*, esses registradores são identificados como $0, $1 até $31. Dois desses registradores de propósito geral, o primeiro e o último, são reservados para funções especiais:

- o registrador $0 é fixado em ‘0’ em hardware. Esse registrador é usado algumas vezes como operando quando um valor zero é necessário. Se esse registrador é usado como destino em alguma instrução, o resultado é descartado;
- o último registrador, $31, é usado como *link register* pela instrução jal (*jump and link*), que é usada para chamada de funções.

Os dois registradores de uso especial, chamados HI e LO, são usados para guardar o resultado de instruções de multiplicação de divisão de inteiros:

- nas operações de multiplicação de inteiros, os registradores HI e LO guardam o resultado de 64 bits, com a parte mais significativa em HI e a parte menos significativa em LO;
- nas operações de divisão de inteiros, o quociente de 32 bits é guardado no registrador LO, enquanto o resto é guardado no registrador HI.

Embora não seja um requisito do ponto de vista do hardware do processador, existe uma convenção sobre o uso dos 32 registradores, conforme mostrada na Tabela 3.3.

Tabela 3.3 – Convenção de uso dos registradores de propósito geral MIPS

Nome do Registrador	Número	Uso
Zero	0	Constante Zero
$at	1	Reservado para assembler
$v0, $v1	2, 3	Resultado de uma função
$a0, $a1, $a2, $a3	4 – 7	São usados para passagem de argumentos (não preservado após chamadas de subrotinas)
$t0 - $t7	8 – 15	Temporários (valores não são preservado entre chamadas de subrotinas)
$s0 - $s7	16 – 23	Temporários salvos
$t8, $t9	24, 25	Temporários não salvos
$k0, $k1	26, 27	Reservados para o kernel do sistema operacional
$gp	28	Global Pointer.
$sp	29	*Stack Pointer*
$fp ou $s8	30	*Frame Pointer*. Aponta para o endereço da pilha que o *stack pointer* apontava no início de uma chamada de subrotina. Se não utilizado como *Frame Pointer,* é utilizado como registrador $s8.
$ra	31	Endereço de retorno.

Os registradores $v0 e $v1 são usados como retornar resultados de sub-rotinas. Registradores $a0 a $a3 são usados para passar os quatro primeiros argumentos para funções. Os demais argumentos são passados através da pilha. Esses registradores não são preservados entre chamadas de funções (isto é, a função chamada pode modificar livremente o conteúdo desses registradores).

Os registradores $t0 a $t9 são registradores que não precisam ser preservados entre chamadas de funções. Assume-se que esses registradores são salvos pela função que chama. Por outro lado, os registradores $s0 a $s7 devem ser preservados entre chamadas.

O registrador $sp é o *stack pointer,* e aponta para o topo da pilha. O registrador $ra é usado para guardar o endereço de retorno de uma chamada de função.

O registrador $gp aponta para a área de memória que guarda constantes e variáveis globais. O registrador $at é utilizado pelo *assembler*. O *assembler* às vezes utiliza esse registrador para traduzir pseudo-instruções. Pseudo-instruções não são instruções do

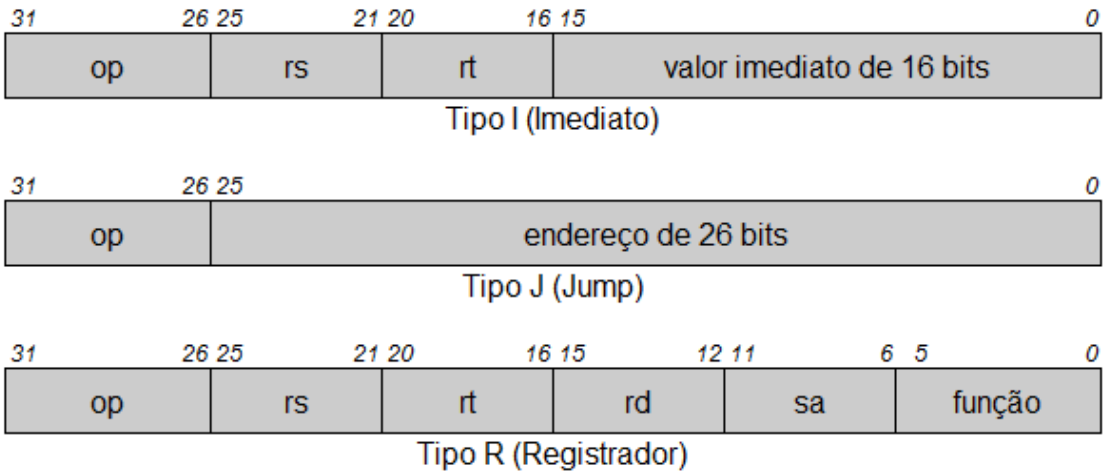
processador; são instruções suportadas pelo *assembler*, sendo que cada pseudo-instrução é traduzida pelo *assembler* em uma ou mais instruções do processador.

3.3.2 Formato das Instruções

O MIPS, sendo uma arquitetura RISC, utiliza formatos de instruções de tamanho fixo. Cada instrução possui 32 bits e apresenta um dos três formatos mostrados na Figura 3.8:

- Tipo I (Imediato): todas as instruções *load* e *store* usam esse formato de instrução. O valor imediato é um inteiro de 16 bits com sinal. Instruções lógicas e aritméticas que utilizam um valor imediato fazem uso desse formato. Instruções de *branch* usam um *offset* de 16 bits com sinal relativo ao PC e são codificadas com esse formato.
- Tipo J (*Jump*): instruções de *jump* que especificam um endereço de 26 bits utilizam esse formato. Esses 26 bits são deslocados dois bits para e esquerda combinados com os bits mais significativos do PC para obter o endereço absoluto.
- Tipo R (Registrador): instruções lógicas e aritméticas utilizam esse formato. Além dessas, instruções de *jump* nas quais o endereço destino é especificado indiretamente via um registrador, utilizam esse formato.

Figura 3.8 – Formatos de Instruções MIPS



O uso de um número limitado de formatos de instruções simplifica a decodificação das instruções. No entanto, três formatos de instruções e um modo simples de endereçamento

significa que operações e modos de endereçamento complexos precisam ser sintetizados pelo compilador.

3.3.3 Modos de Endereçamento

Como o MIPS é uma arquitetura *load/store*, somente as instruções de *load* e *store* acessam a memória. Todas as outras instruções utilizam seus operandos em registradores, ou seja, modo de endereçamento em registrador.

O MIPS utiliza a instrução do tipo I para codificar instruções do tipo *load* e *store*. O endereço é definido pelo conteúdo de um registrador base mais um valor imediato com sinal de 16 bits.

Apesar de haver apenas um modo de endereçamento, podemos fazer o registrador base ou o valor imediato zero e obter variações desse modo de endereçamento básico. Por exemplo, se especificarmos $0 como registrador base, o endereço é o próprio valor imediato. E se, especificarmos zero como valor imediato, o endereço é o conteúdo do registrador base.

4 SISTEMA PROPOSTO

A Figura 4.1 apresenta uma visão geral da arquitetura proposta. O tradutor de primeiro nível se comunica com a memória e o resto do sistema, que é composto do mecanismo de tradução de primeiro nível, de tradução de segundo nível, a cache de tradução (TCache), o processador MIPS para executar o código comum quando necessário e um *array* dinamicamente reconfigurável. O tradutor binário de primeiro nível traduz código de três diferentes ISAs, a saber: x86, ARM e PowerPC.

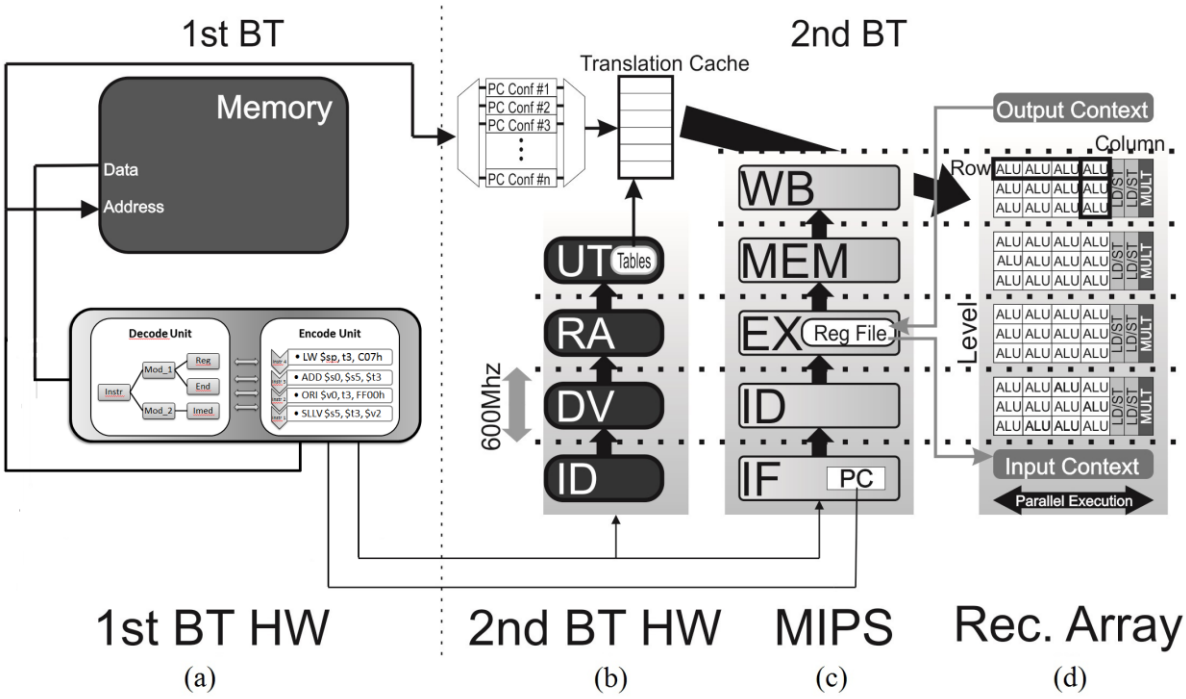
A utilização de um *array* reconfigurável para o processo de otimização foi escolhido ao invés de um processador Superscalar ou VLIW pois, já foi provado que ele acelera a execução de software e reduz o consumo de energia, mostrando ganho sobre ambos sistemas (BECK; CARRO, 2005) (LYSECHY et al., 2006) (BECK; CARRO, 2010). Aliás, é senso comum que quanto mais a tecnologia evolui (evolução do *die*), mais importância ganha uma característica dos sistemas embarcados: a regularidade. Além de serem mais previsíveis, circuitos regulares têm também menor custo pois, quanto mais customizável um circuito é, mais caro ele se torna. Dessa forma, circuitos regulares podem resolver o custo de máscara e outras tantas questões tais como manufaturabilidade (*printability*), integridade de potência e outros aspectos das futuras tecnologias. Essa arquitetura reconfigurável, utilizada no segundo nível de tradução, foi estendida do trabalho de (BECK et al., 2008), sendo responsável pela otimização do código comum (MIPS) após a tradução de primeiro nível.

Como um exemplo de operação, consideremos uma aplicação compilada para a ISA ARM, e que será executada pela primeira vez. Considerando que o sistema proposto visa execução de diversas ISAs, a arquitetura nativa a ser utilizada deve ser escolhida a priori, no momento de *boot* do sistema. O TB de primeiro nível então realiza a busca das instruções e traduz em instruções MIPS. Nesse momento não é realizado salvamento de instruções para futuro reuso: todas as instruções são processadas em tempo de execução, evitando custo de memória. Então, enquanto o processador MIPS executa o código MIPS, o TB de segundo nível traduz esse código em uma configuração que será executada no *array*, e salva essa configuração na Cache de Tradução (TCache) para uso futuro. O segundo nível de tradução somente guarda sequências otimizadas de código quando identificado que são *hot spots* (partes de código frequentemente executadas).

Cada entrada na TCache é indexada pelo *Program Counter* (PC) da arquitetura nativa. Assim, da próxima vez que uma porção de código ARM que já foi traduzida para MIPS e otimizada para o array é encontrada, sua forma otimizada (uma configuração) é buscada

diretamente da TCache. Quando isso acontece, o primeiro nível de tradução, o processador MIPS e o segundo nível de tradução ficam “desativados” e o array reconfigurável inicia sua reconfiguração e execução. À medida que mais e mais sequências de instruções são executadas e traduzidas, e a TCache vai sendo preenchida, o impacto dos dois níveis de tradução vai sendo amortizado e os ganhos gerados pelo array começam a aparecer.

Figura 4.1 – (a) TB de primeiro nível. (b) pipeline de 4 estágios do TB de segundo nível. (c) pipeline de 5 estágios do processador MIPS. (d) *array* reconfigurável.



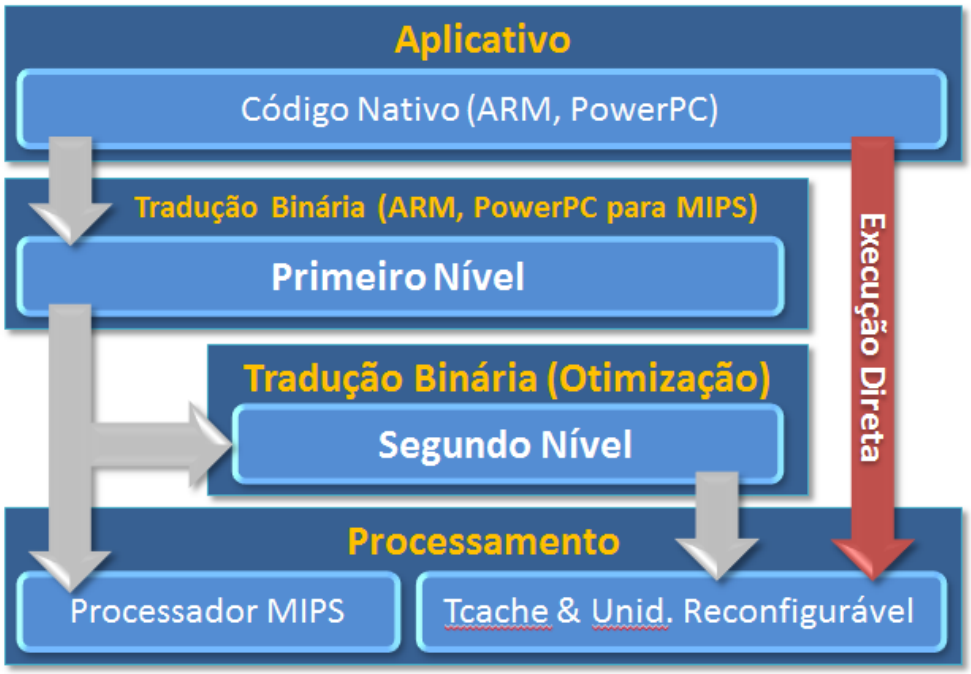
É considerado que, em média, 80% do tempo de computação de um programa é empregado na execução de laços (KLAIBER, 2000). Senso assim, a solução proposta apresenta a seguinte vantagem: uma vez que determinado trecho de código passa através dos dois níveis de tradução binária, da próxima vez que ele é executado, o código traduzido pelo primeiro nível e otimizado pelo segundo nível de tradução já estará presente na TCache, e a execução será direta. Dessa forma não teremos os custos de tradução durante essa execução, amortizando os custos totais, conforme ilustrado pela Figura 4.2.

Resumindo, os ganhos de desempenho são decorrentes de:

- Termos os dois níveis de tradução implementados em hardware, sendo o controle das unidades são realizados em paralelo. Dessa forma, a tradução binária tem baixo impacto no desempenho;

- Uma vez que o código é traduzido e otimizado, ele será executado diretamente na arquitetura reconfigurável;
- A arquitetura reconfigurável é utilizada para promover paralelismo em nível de instrução na execução do código, acelerando a execução em relação a um processador MIPS padrão.

Figura 4.2 – Execução do tradutor proposto



No trabalho apresentado por (FAJARDO, 2011), no primeiro nível de tradução, foi implementada a tradução de 50 instruções de um total de cerca de 190 instruções disponibilizadas pela ISA x86. Todos os modos de endereçamento foram suportados. O subconjunto de instruções foi suficiente para compilar e executar todos benchmarks testados.

Para a ISA ARM o *User Mode* com o conjunto de instruções ARMv4 foi suportado, com exceção das instruções de coprocessador, todas outras instruções foram suportadas, incluindo todos modos de endereçamento. O conjunto de instruções implementado foi suficiente para executar todos os benchmarks testados. Thumb e outras extensões multimídia da ISA ARM não foram suportadas.

Para a ISA PowerPC, foram implementadas 78 das 131 instruções da categoria *basic* do conjunto de instruções Power ver. 2.06b. Esse conjunto de instruções foi suficiente para executar todos benchmarks testados. O custo de implementação do conjunto completo da categoria *basic* foi extrapolado para medição de área que será mostrada a seguir.

O tradutor x86 foi implementado como uma unidade separada dos tradutores ARM/PowerPC, visto que as diferenças entre arquiteturas (exemplo: CISC versus RISC) levam a escolhas distintas de implementação. A seguir apresentamos a implementação do tradutor de primeiro nível da arquitetura ARM/PowerPC, que é o escopo desse trabalho.

4.1 Organização do Primeiro Nível de Tradução – ARM e PowerPC

O primeiro nível de tradução, para as arquiteturas ARM e PowerPC, é composto por duas unidades distintas de hardware, a saber: Unidade de Decodificação e Unidade de Codificação. A Figura 4.3 mostra o diagrama em blocos deste primeiro nível de tradução. Apresentaremos a seguir o papel de cada módulo no processo de tradução e suas implementações.

4.1.1 Unidade de Decodificação

Essa unidade é responsável por analisar uma instrução buscada da memória e extrair sua semântica. A semântica da instrução refere-se à operação a ser realizada e a localização de seus operandos (registrador ou endereço de memória). Cada informação extraída é encaminhada para próxima unidade.

Essa unidade é composta apenas de lógica combinacional, que separa os campos de cada instrução baseada em seu formato. Esse processo é feito de forma paralela: a instrução é usada como entrada para múltiplos blocos, cada um dos quais separa os campos assumindo um formato particular. Alguns campos podem ser fundidos, pois diferentes operações podem especificar a mesma informação no mesmo campo (por exemplo, registrador de destino).

Vamos considerar, por exemplo, a arquitetura ARM, e as estruturas necessárias para decodificar instruções de processamento de dados. Essa classe de instruções representa uma das codificações possíveis e cobre operações aritméticas e lógicas básicas sobre registradores. Todas as instruções ARM são de tamanho fixo e, portanto, é sabido quais campos analisar para determinar o tipo de instrução. Para decodificar uma instrução tal como “ADD r2, r1, r0”, os seguintes bits devem ser verificados:

- Bits 27 = ‘0’ e 28 = ‘0’. Se essa condição for verdadeira então é uma instrução de processamento de dados;

- Bit 25 = ‘0’. Se essa condição for verdadeira, então o segundo operando é um registrador que precisa ser deslocado por um valor imediato ou por um valor especificado em outro registrador;
- Bit 4 = ‘0’. Se essa condição for verdadeira, então o modo de endereçamento para a operação é um registrador deslocado por um valor imediato.

As condições acima são realizadas pelo bloco de hardware que identifica o formato das instruções. Ao mesmo tempo, o seguinte conteúdo da instrução é extraído:

- A operação (ADD) é especificada pelos bits 24 a 21, o registrador destino (r2) pelos bits 19 a 16, e o primeiro operando pelos bits 15 a 12.
- O registrador do segundo operando (r0) é especificado nos bits 3 a 0, e o valor do deslocamento é especificado nos bits 11 a 7.

O procedimento acima exposto cobre todas as instruções de processamento de dados que usam esse modo de endereçamento na arquitetura ARM.

4.1.2 Unidade de Codificação

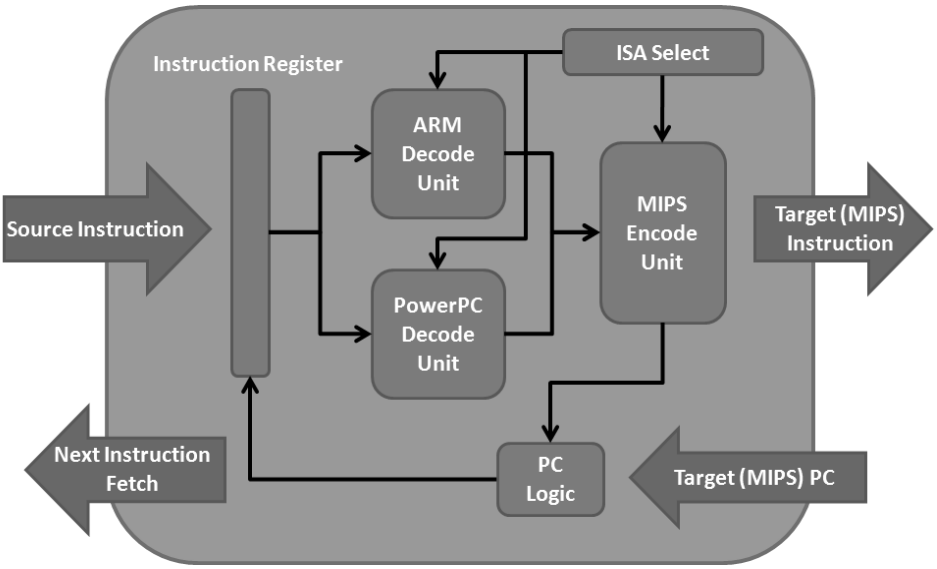
Essa unidade recebe as informações extraídas da unidade de decodificação e as utiliza para gerar uma sequência de instruções MIPS equivalente. Como a tradução é feita de um para vários (ou seja, uma instrução da arquitetura nativa pode gerar mais que uma instrução MIPS), o *Program Count* (PC) do processador MIPS não pode ser usado para endereçar a memória de instruções. Essa unidade ainda possui um mecanismo interno que controla outro PC, o qual endereça a memória de instruções (PC_{BT}). Portanto, uma distinção entre esses dois PCs deve ser feita: temos o PC_{BT} que endereça a memória de instruções e o PC_{MIPS} interno ao processador MIPS.

Essa unidade é uma Máquina de Estados Finitos, cujas entradas são todas saídas da Unidade de Decodificação, e que gera uma instrução MIPS cada vez que o processador MIPS tenta buscar uma nova instrução (alterando o valor do PC_{MIPS}). Quando a última instrução MIPS está prestes a ser gerada, um sinal é enviado para incrementar o PC_{BT} a fim de buscar a próxima instrução.

Como o circuito de tradução possui um caminho crítico menor que o do processador MIPS, a frequência de relógio não fica comprometida, e o desempenho é limitado pelo número de instruções geradas. No entanto, o número de ciclos necessários para buscar uma nova instrução e iniciar a execução das instruções traduzidas no processador MIPS devem ser

levados em conta, visto que poderiam levar a grandes *stalls* (isto é, o processador da arquitetura alvo teria de parar para esperar por instruções). Outra questão que foi considerada foi o tratamento das instruções de salto (*branches*). O esquema de tradução para esse tipo de instrução está fortemente atrelado à arquitetura nativa considerada, visto que algumas arquiteturas podem permitir operações diretamente sobre o registrador PC, enquanto outras permitem saltos indiretos via registrador. No entanto, o PC que precisa ser atualizado não é o interno ao processador MIPS (PC_{MIPS}), mas sim o PC_{BT}. Portanto, foram inseridas instruções MIPS para transferir o valor do PC_{BT} para o processador MIPS, operar sobre ele se necessário, e capturar de volta para o PC_{BT} as alterações feitas sobre o valor do PC_{MIPS}. O PC_{BT} é carregado em um registrador MIPS através de duas instruções: a instrução LUI é inserida para carregar a parte mais significativa do PC_{BT} e a seguir a instrução ORI carrega a parte menos significativa do PC_{BT} nesse mesmo registrador. São inseridas então instruções MIPS de acordo com o tipo de instrução de salto da arquitetura nativa e finalmente uma instrução JR (*Jump Register*), após a qual o PC_{MIPS} será atualizado com o endereço da próxima instrução nativa a ser traduzida. O PC_{BT} é então atualizado com o valor do PC_{MIPS} ($PC_{BT} = PC_{MIPS}$).

Figura 4.3 – Diagrama em blocos do primeiro nível do TB para arquiteturas ARM e PowerPC



4.2 Tradução ARM – MIPS

4.2.1 Mapeamento de Registradores

O MIPS possui um número superior de registradores de propósito geral do que o ARM: são 32 registradores no MIPS versus 16 registradores no ARM (considerando o User Mode somente). Com exceção dos registradores $0 e $31 os demais registradores MIPS não possuem uso específico, apenas certas convenções para uso desses registradores, como visto na seção 3.3.1. Dessa forma, foi possível o mapeamento direto dos 16 registradores ARM em registradores MIPS, conforme a Tabela 4.1.

Além dos 16 registradores utilizados para mapeamento de registradores ARM, foram utilizados outros seis registradores MIPS para auxiliar no processo de tradução:

- $0: esse registrador MIPS possui conteúdo fixo com valor zero. Foi utilizado durante o processo de tradução quando necessário carregar um valor imediato em um registrador. Exemplo: addi $r8, $r0, 3;
- $1, $4: utilizados como registradores de rascunho. Algumas instruções ARM geram mais do que uma instruções MIPS, e podem ser necessários cálculos intermediários. Para não interferir nos registradores mapeados, utilizamos esses registradores para operar valores intermediários;
- $2, $3: utilizado para manter os *flags* da ULA e a cópia de bits da ULA respectivamente, de forma a auxiliar a execução condicional de instruções, como será explicado a seguir;
- $5: utilizado para manter o valor do *carry*. Algumas instruções ARM realizam operações aritméticas com o bit de *carry* da ULA. Para que possamos traduzir essas instruções em instruções MIPS, a solução foi disponibilizar o *carry* em um registrador MIPS.

Tabela 4.1 – Mapeamento dos Registradores ARM no MIPS

Registrador ARM	Função ARM	Registrador MIPS	Função MIPS
R0 – R7	propósito geral	$8 - $ 15 (t0 – t7)	propósito geral
R8 – R9	propósito geral	$24 - $25 (t8 – t9)	propósito geral
R10 – R12	propósito geral	$16 - $18 (s0 – s7)	propósito geral
R13 (sp)	*Stack Pointer*	$29 (sp)	*Stack Pointer*

R14 (lr)	*Link Register*	$31 (ra)	*Return Address*
R15 (pc)	*Program Counter*	$28 (gp)	*Global Pointer*

4.2.2 Execução Condicional de Instruções

Como visto na seção 3.1, toda instrução ARM pode ser condicionalmente executada de acordo com seu campo *Condition Field* e do estado dos *flags* condicionais do registrador CPSR. Se o estado dos *flags* C, N, Z e V atendem às condições da instrução, ela é executada, do contrário não. Os *flags* do registrador CPSR, por sua vez, podem ser preservados ou atualizados como resultado de determinadas operações lógicas e aritméticas, de acordo com o bit S dessas instruções (S=1, atualiza flags; S=0, preserva flags).

Para reproduzirmos esse comportamento no MIPS, precisamos:

- salvar os *flags* condicionais se a instrução assim determinar;
- implementar a execução condicional de instruções de acordo com esses flags.

Para salvar os *flags* foram utilizados dois registradores MIPS: $2 e $3. O primeiro ($2) guarda os *flags* diretamente da ULA, para tal uma pequena modificação no processador MIPS foi necessária: o registrador $2 foi acoplado à ULA. O segundo ($3) é utilizado para atualizar ou manter a cópia atual do registrador ($2) conforme o bit S da instrução ARM assim determinar. A Tabela 4.2 mostra dois exemplos de tradução de uma instrução aritmética, a primeira com atualização de *flags* e a segunda não.

Tabela 4.2 – Tradução de instrução aritmética com e sem atualização de *flags*

Instrução ARM	Operação	Tradução MIPS
ADDS r2, r3, r4	r2 := r3+r4 e atualiza flags	ADDU $10, $11, $12 ADDU $3, $0, $2
ADD r2, r3, r4	r2 := r3+r4 e preserva flags	ADDU $10, $11, $12

Para execução condicional das instruções, instruções extras de teste e salto foram introduzidas sempre que uma instrução ARM possui *condition field* diferente de 1110 (*always*). A Tabela 4.3 mostra o exemplo de tradução de uma instrução com execução condicional. Nesse exemplo, é possível observar o *overhead* que a execução condicional de instruções insere no processo de tradução. As primeiras sete instruções são utilizadas apenas

para testar os *flags* de condição, enquanto a última instrução realiza a operação de fato da instrução ARM.

Tabela 4.3 – Exemplo de tradução de uma instrução com execução condicional

Instrução ARM	Tradução MIPS		Ação
MOVGT r0, #3 (move imediato para registrador R0, se flag Z = 0 & flag N = V)	ANDI $1, $3, 4	*(1)*	Máscara do bit Z (0x4) é armazenada no registrador temporário R1
	BGTZ1, $1, 28	*(2)*	Pula todas instruções se flag Z = 0
	ANDI $1, $3, 8	*(3)*	Máscara do bit N (0x8) é armazenada no registrador temporário R1
	ANDI $4, $3, 1	*(4)*	Máscara do bit V (0x1) é armazenada no registrador temporário R4
	BGTZ, $1, 8	*(5)*	Se N != 0, pula para instrução (7)
	BEQ $4, $0, 8	*(6)*	N = 0, agora testa se V = 0, se for, pula para instrução (8)
	BEQ $4, $0, 8	*(7)*	N = 1, agora testa se V = 0, se for a pula instrução seguinte.
	ADDI $8, $0, 3	*(8)*	Move imediato para registrador $8

4.2.3 Tradução do Segundo Operando

Como visto na seção 3.1.3, as instruções lógicas e aritméticas ARM possuem um segundo operando flexível. Esse operando pode ser o conteúdo de um registrador ou imediato e seu valor ainda pode ser deslocado ou rotacionado antes de ser operado pela instrução. A tradução deste segundo operando é um dos fatores que onera a tradução ARM para MIPS, gerando diversas instruções MIPS para uma única instrução ARM, visto que no MIPS não temos esse tipo de operando.

Por exemplo, a instrução da Tabela 4.4 desloca o conteúdo do registrador r7 logicamente para a direita em número de bits definido pelo byte menos significativo de r2, subtrai esse valor do conteúdo de r5 e armazena no registrador r4.

Outro exemplo, a instrução da Tabela 4.5 subtrai de r5 o valor 1020 e armazena em r4. A valor imediato 1020 é definido na instrução ARM pelo segundo operando como um imediato de 8 bits 0xFF rotacionado para a direita 30 vezes. Nesse caso o código MIPS equivalente é composto por três instruções, pois precisamos de uma instrução para carregar o imediato em um registrador, outra para rotacionar esse valor, para finalmente na terceira instrução realizar a subtração.

Tabela 4.4 – Tradução de instrução com segundo operando registrador

Instrução ARM	Tradução MIPS
SUB r4, r5, r7, LSR r2	SRLV $1, $15, $10 SUBU $12, $13, $1

Tabela 4.5 – Tradução de instrução com segundo operador imediato

Instrução ARM	Tradução MIPS

SUB r4, r5, #1020	ADDI $1, $0, 0xff
	ROTR $1, $1, 0x1e
	SUBU $12, $13, $1

4.2.4 Execução de Branches

Como visto na seção 4.1.2, o *Program Counter* (PC) do processador MIPS não pode ser usado para endereçar a memória de instruções e sendo assim é necessário um mecanismo especial para a tradução de *branches*. Os *branches* no ARM podem ser diretos ou indiretos.

Os *branches* diretos são executados pelas instruções B (*Branch*) e BL (*Branch with Link*), e realizam um salto para frente ou para trás, ao somar ao PC um valor imediato de 24 bits em complemento de 2, deslocado 2 bits para esquerda. Dessa forma é possível saltar até +/-32Mbytes em relação ao PC.

Como instruções de *branch* indireto, temos a instrução BX, na qual o endereço do saldo é definido em um registrador. Ainda, na arquitetura ARM, instruções lógicas e aritméticas podem ter como operando o registrador R15 (PC) e, sendo assim, quando o registrador R15 é usado como operando destino nesses tipos de instruções, teremos um caso de *branch* indireto.

Posto isso, vejamos como fica o processo de tradução de *branches* para cada caso:

- *Loads* ou operações lógicas ou operações aritméticas com R15 (PC) como operando destino: nesses casos, após a tradução da operação em si, é acrescida a instrução JR $28. O tradutor irá atualizar o PC_{BT} com o valor do PC_{MIPS} sempre que a instrução JR for executada.
- Instruções que utilizem como operando o registrador R15, devem inserir duas instruções antes da tradução da instrução em si para que seja carregado no PC_{MIPS} o valor do PC_{BT}. Por exemplo, a instrução MOV r14, r15 (ou MOV LR, PC) é traduzida conforme a Tabela 4.6.

Tabela 4.6 – Tradução da instrução MOV r14, r15 , PC (0x00012d3c) como operando

LUI $28, #0x1	Carrega os dezesseis bits mais significativos do PC_{BT} no $28
ORI $28, $28, #2d3c	Realiza "or" entre a parte menos e mais significativa do PC_{BT}. Agora $28 contém o $PC_{BT.}$
ADDIU $31, $28, #0	Carrega o PC mapeado no MIPS, para o LR mapeado no MIPS.

- B (*Branch*): nessa instrução o endereço do *branch* é definido por um valor imediato que é somado ao PC. Para realizar essa tradução é preciso primeiramente carregar o PC_{BT} no PC_{MIPS} para que o MIPS possa realizar a soma do imediato ao PC. Abaixo, na Tabela 4.7, podemos ver a tradução completa da instrução "B #16", supondo que o PC seja 0x12d3c. Note que o PC 0x12d3c é o mesmo PC_{BT}, ou seja o PC que endereça o código ARM e portanto esse valor é conhecido pelo tradutor de primeiro nível. A tradução dessa instrução gera um overhead de tradução de 6x1 instruções MIPS/ARM.

Tabela 4.7 – Tradução da instrução B (*Branch*)

Instrução MIPS	Resultado
LUI $28, #0x1	Carrega os dezesseis bits mais significativos do PC_{BT} no $28
ORI $28, $28, #2d3c	Realiza "or" entre a parte menos e mais significativa do PC_{BT}. Agora $28 contém o $PC_{BT.}$
LUI $1, #0	Carrega os dezesseis bits mais significativos do imediato (#16) no registrador de rascunho $1.
ORI $1, #16	Realiza "or" entre a parte menos e mais significativa do imediato. Agora $1 contém o imediato 16.
ADDU $28, $28, r1	Soma o PC_{BT} ao imediato 16 e armazena em $28
JR $28	Salta para $28.

- BL (*Branch with Link*): a instrução ARM “BL” é muito similar à instrução “B” exceto pelo fato que a instrução BL salva o valor atual no PC no *Link Register*. Dessa forma a tradução dessa instrução fica muita parecida com a “B”, com a adição de mais uma instrução MIPS. Abaixo, na Tabela 4.8, podemos ver a tradução completa da instrução “BL #16”, supondo que o PC seja 0x12d3c. A tradução dessa instrução gera um overhead de tradução de 7x1 instruções MIPS/ARM.

Tabela 4.8 – Tradução da instrução BL

Instrução MIPS	Resultado
LUI $28, #0x1	Carrega os dezesseis bits mais significativos do PC_{BT} no $28
ORI $28, $28, #2d3c	Realiza “or” entre a parte menos e mais significativa do PC_{BT}. Agora $28 contém o $PC_{BT.}$
ADD $31, $0, $28	O registrador $31 (registrador que mapeia o LR no mips) recebe o $PC_{BT.}$
LUI $, #0	Carrega os dezesseis bits mais significativos do imediato (#16) no registrador de rascunho $1.
ORI $1, #16	Realiza “or” entre a parte menos e mais significativa do imediato. Agora $28 contém o imediato 16.
ADDU $28, $28, $1	Soma o PC_{BT} ao imediato 16 e armazena em $28
JR $28	Salta para $28.

- BX (*Branch and Exchange*): nessa instrução o endereço do branch é o conteúdo de um registrador na instrução. Essa instrução gera apenas uma instrução MIPS. Por exemplo, a instrução “BX R14” (ARM) é traduzida para a instrução “JR $28” (MIPS). O PC_{BT} é atualizado para PC_{MIPS} após a execução dessa instrução, assim o PC_{BT} endereça a próxima instrução ARM a ser traduzida.

4.2.5 Outros Exemplos de Tradução

As instruções LDM e STM permitem a transferência entre um bloco de registradores e a memória, ou seja, *loads* e *stores* múltiplos. A instrução define um registrador para o endereço base e um campo de 16 bits define através de um mapa de bits quais registradores serão usados na transferência. No exemplo da Tabela 4.9, a instrução LDMIA transfere para quatro

registradores 16 bytes do *stack*, as letras “IA” definem que o endereçamento é com pós incremento e o símbolo “!” define que o registrador base deve ser atualizado (*write-back*).

Tabela 4.9 – Tradução da instrução ARM “LDMIA R13!, {r4-r6, r14}”

LW $12, 0 ($29)	Carrega no registrador $12 (registrador ARM R4) o conteúdo da memória apontada pelo registrador $29 (registrador ARM R13/SP).
ADDIU $29, $29, #4	Incrementa $29 em quatro bytes, apontando para próxima posição de memória.
LW $13, 0 ($29)	Carrega no registrador $13 (registrador ARM R5) o conteúdo da memória apontada pelo registrador $29 (registrador ARM R13/SP).
ADDIU $29, $29, #4	Incrementa $29 em quatro bytes, apontando para próxima posição de memória.
LW $14, 0 ($29)	Carrega no registrador $14 (registrador ARM R6) o conteúdo da memória apontada pelo registrador $29 (registrador ARM R13/SP).
ADDIU $29, $29, #4	Incrementa $29 em quatro bytes, apontando para próxima posição de memória.
LW $31, 0 ($29)	Carrega no registrador $31 (registrador ARM R14/LR) o conteúdo da memória apontada pelo registrador $29 (registrador ARM R13/SP).
ADDIU $29, $29, #4	Incrementa $29 em quatro bytes, apontando para próxima posição de memória.

Nas instruções LD (*load*) e ST (*store*), o *offset* pode ser pré ou pós indexado. O registrador base pode ou não ser atualizado com seu valor somado ou subtraído o offset. Por exemplo, na instrução da Tabela 4.10, após carregar o conteúdo da memória no registrador, o offset é somado ao endereço base.

Tabela 4.10 – Tradução da instrução ARM “LDR r14, [r13], #4”

LW $31, 0($29)	Carrega no registrador $31 (ARM r14) o conteúdo de memória apontado por $29.
ADDIU $29, $0,#4	Soma ao registrador base o offset.

4.3 Tradução PowerPC – MIPS

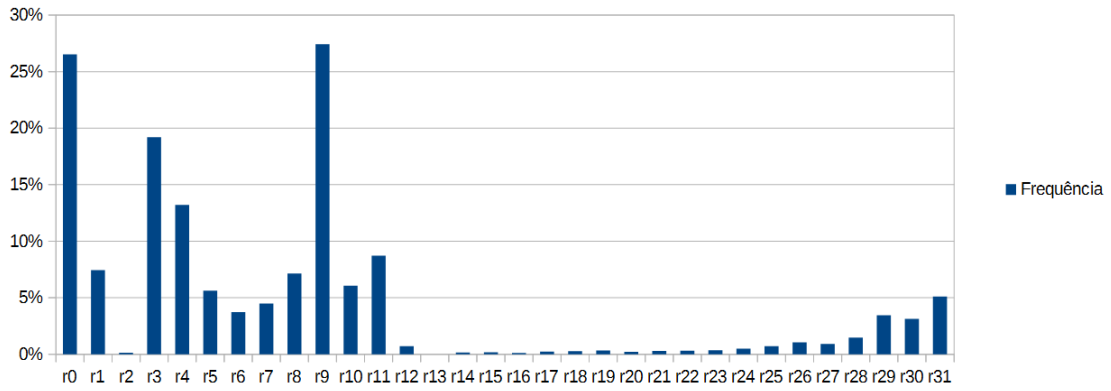
4.3.1 Mapeamento de Registradores

O MIPS possui o mesmo número de registradores de propósito geral que o PowePC: são 32 registradores. No entanto, mapear somente os registradores de propósito geral não é suficiente, é preciso mapear os registradores de uso especifico do PowerPC (CTR, CR, LR e XER) e outros tantos para auxiliar no processo de tradução.

Para viabilizar o mapeamento de todos os registradores PowerPC, treze de seus registradores ficaram mapeados em memória. Foram utilizados três registradores temporários MIPS para carregar até três desses treze registradores da memória a cada instrução PowerPC traduzida. Assim, toda vez que um registrador mapeado em memória é utilizado em determinada instrução PowerPC, o tradutor primeiramente carrega ele da memória para um registrador temporário através de uma instrução de *load*, traduz a instrução PowerPC para uma ou mais instruções MIPS e então insere uma instrução de *store*, salvando esse registrador em memória.

Para minimizar o overhead de instruções com o mapeamento de registradores em memória, os treze registradores PowerPC que apresentaram uso menos frequente foram mapeados em memória. A frequência de uso dos registradores PowerPC apresentou distribuição similar em todos benchmarks. A Figura 4.4 mostra a frequência de uso dos registradores nas instruções dos benchmarks testados. Os treze registradores menos usados tem um frequência de 2.55% no total de instruções, o que presenta um overhead de 5.10% no total de instruções ARM/PowerPC.

Figura 4.4 – Frequência de uso dos registradores PowerPC



A Tabela 4.11 mostra o mapeamento dos registradores PowerPC em registradores MIPS. Além dos 16 registradores utilizados para mapeamento de registradores ARM, foram utilizados outros seis registradores MIPS para auxiliar no processo de tradução:

- $0: esse registrador MIPS possui conteúdo fixo com valor zero. Foi utilizado durante o processo de tradução. Algumas instruções Power utilizam possuem o valor 0 quando o registrador for definido como R0;
- $23, $24, $25: utilizados para carregar os registradores PowerPC mapeados em memória;
- $26 a $28: utilizados como registradores de rascunho. Algumas instruções PowerPC geram mais do que uma instruções MIPS, e podem ser necessários cálculos intermediários. Para não sobrescrever em registradores mapeados, utilizamos esses registradores para operar valores intermediários;
- $29, $30: utilizado para manter o *carry* da ULA e uma cópia deste, o *carry_r*. Certas instruções PowerPC fazem uso do *carry*, e certas instruções afetam o *carry*. Foi utilizado um mecanismo similar à execução condicional do ARM. O registrador *carry* é atualizado diretamente pela ULA. Quando uma instrução que afeta o *carry* é traduzida, uma instrução para copiar *carry* para *carry_r* é executada. As instruções que utilizam *carry* são traduzidas utilizando uma instrução adicional para operar o registrador *carry_r*.

Tabela 4.11 – Mapeamento dos Registradores PowerPC no MIPS

Registrador PowerPC	Função PowerPC	Registrador MIPS	Função MIPS
R0, R1, R3-R12, R25-R31	propósito geral	$1 - $ 19	propósito geral
CR/XER	*Condition Register*	$20, $21	propósito geral
CTR	*Count Register*	$22	propósito geral
LR	*Link Register*	$31	*Return Address*
R2, R13-R24	propósito geral	Mapeados em memória	

4.3.2 Execução de Branches

O PowerPC possui instruções de *branch* diretos e indiretos. As instruções de *branch* calculam o endereço efetivo (EA) do salto em uma das quatro formas abaixo:

- Adicionando um *offset* ao endereço da instrução de *branch* (instruções *Branch* e *Branch Conditional* com campo AA=0);
- Especificando um endereço absoluto (instruções *Branch* e *Branch Conditional* com campo AA=1);
- Utilizando o endereço contido no *Link Register* (instrução *Branch Conditional to Link Register*);
- Utilizando o endereço contido no registrador *Count Register* (*Branch Conditional to Count Register*).

As instruções de *branch* podem ser condicionais e incondicionais, e o endereço de retorno pode ser opcionalmente fornecido. Se o endereço de retorno for fornecido (campo LK=1), o endereço imediatamente seguinte ao endereço da instrução de *branch* é salvo no registrador LR, independente se o *branch* foi tomado ou não.

O mecanismo de tradução de *branches* é o mesmo adotado para o ARM: quando o *branch* for relativo ao PC, o PC do tradutor (PCBT) deve ser carregado para então ser operado. Na execução da instrução JR o TB captura o novo valor do PC. A execução condicional de *branches* também utiliza o mesmo mecanismo da execução condicional do ARM.

A Tabela 4.12 mostra a tradução da instrução "bc 0xC, 0, #0x200". Esta instrução indica que o *branch* é tomado se a condição é verdadeira (0xC) se o bit 0 do registrador CR é '1' (0). O endereço do *branch* é o endereço da instrução + 0x200. No exemplo, foi considerado PCBT = 0x12d3c.

Tabela 4.12 – Exemplo de tradução da instrução *bc 0xC, 0, #0x200*

ANDI r27, r21, #1	Mascara o bit 0 do registrador mapeado CR (r21) e armazena no registrador temporário r27.
BEQZ r27, #16	Salta para instrução depois do JR se r27 = 0 (condição é falsa), *branch* não é tomado.
LUI r28, #0x1	Carrega os dezesseis bits mais significativos do PC_{BT} no $28
ORI r28, r28, #2d3c	Realiza "or" entre a parte menos e mais significativa do PC_{BT}. Agora $28 contém o $PC_{BT.}$

ADDIU r28, r28, #0x200	Soma o offset ao $PC_{BT.}$
JR r28	Realiza o *branch*

4.3.3 Outros Exemplos de Tradução

O PowerPC possui diversas funções de load/store, podendo transferir bytes, *words* ou *halfwords*. O endereço de memória pode ser dado por um endereço base definido em um registrador, somado um offset de 16 bits, ou ainda através da soma de dois registradores.

Na Tabela 4.13 é apresentado um exemplo de tradução de uma instrução de *load*. Esta instrução carrega um byte da memória para um registrador e atualiza o registrador base com o endereço da posição de memória (registrador + offset imediato).

Tabela 4.13 – Tradução instrução PowerPC “LBZU r0, #4(r5)”

LBU $1, #4 ($6)	Carrega no byte menos significativo do registrador $1 (registrador PowerPC R0) o byte da memória apontada pelo registrador $6 + 4 (registrador PowerPC R6). Demais bytes de $1 são preenchidos com zero *(zero extended)*.
ADDIU $6, $6, #4	Incrementa registrador com o offset. $6 contém o endereço do byte que foi carregado.

A Tabela 4.14 mostra um exemplo de tradução de uma instrução de multiplicação. Nessa instrução é feita a multiplicação de dois inteiros sem sinal e os 32 bits mais significativos do resultado são armazenados em um registrador.

Tabela 4.14 – Tradução da instrução PowerPC “MULHWU r3, r4, r5”

MULTU $5, $6	Multiplica dois números de 32 bits como inteiros sem sinal e armazena nos registradores MIPS Hi e Lo
MFHI $4	Transfere a parte mais significativa da multiplicação para o registrador $4 (registrador PowerPC $3 mapeado).

4.4 MIPS Estendido

A grande vantagem do uso da ISA MIPS como código comum é a regularidade de código com comportamento bem conhecido, facilitando a tradução de outras ISAs para esta.

No entanto, a tradução de um conjunto de instruções complexas como o x86 é ineficiente, pois, em sua maioria, uma única instrução x86 é convertida para várias instruções MIPS. Por exemplo, o suporte ao registrador de flags na ISA x86, que pode ser usado para saltos condicionais, inexiste na arquitetura MIPS. Nesse caso, mais de 20 instruções MIPS deveriam ser geradas por instrução x86 para emular corretamente esse suporte a *flags* no processador MIPS. Outras caraterísticas, como segmentação, também sofrem essa falta de suporte.

Quanto à ISA ARM, embora classificada como uma arquitetura RISC tal qual a ISA MIPS, possui o suporte a execução condicional a todas as instruções ARM. O suporte a execução condicional de instruções pode também sobrecarregar o processo de tradução, visto que instruções extras de teste e salto devem ser inseridas juntamente com uma instrução MIPS padrão de forma a implementar corretamente a execução condicional. Foi observada, nos benchmarks testados, uma taxa de 22% de instruções condicionais na média, o que significa que uma melhora significativa de desempenho poderia ser obtida se algum tipo de suporte para essas instruções fosse implementada no processador MIPS. Uma outra diferença entre ARM e MIPS é o uso de um segundo operando complexo, que pode ser deslocado (*shifted*) de várias formas antes de ser operado. Aproximadamente 39% de todas as instruções ARM traduzidas nos *benchmaks* testados fazem uso desse tipo de operando, o que ocasiona um enorme custo no processo de tradução.

Algumas instruções da ISA PowerPC fazem uso do flag de carry, e algumas outras devem atualizar certos campos do registrador XER (*Fixed-Point Exception Register*) ou do Conditional Register. Na média, 10% de instruções desses tipos foram encontradas nos benchmarks testados, o que também causaria um custo substancial em número de instruções traduzidas.

A fim de reduzir esse custo (*overhead*) causado por essas deficiências da ISA MIPS, consideramos que algumas modificações poderiam ser feitas no processador MIPS, estendendo o suporte em hardware para esses problemas:

- Suporte a execução condicional de instruções: o que implicaria mudanças de ISA para codificar os *flags* condicionais;
- Uso de flags de estado: o que implicaria em modificações na ULA para exposição de *flags*;
- Segundo operando flexível: implicaria na inclusão de um *barrel-shifter* na entrada da ULA.

As modificações supracitadas são uma proposta, não tendo sido implementadas em hardware, apenas simuladas. O simulador do TB de primeiro nível foi utilizado para extrair os

resultados de execução com e sem suporte em hardware. No modo com suporte em hardware, foram eliminadas da tradução as instruções que foram inseridas com intuito de prover compatibilidade binária para as incompatibilidades de ISAs apresentadas.

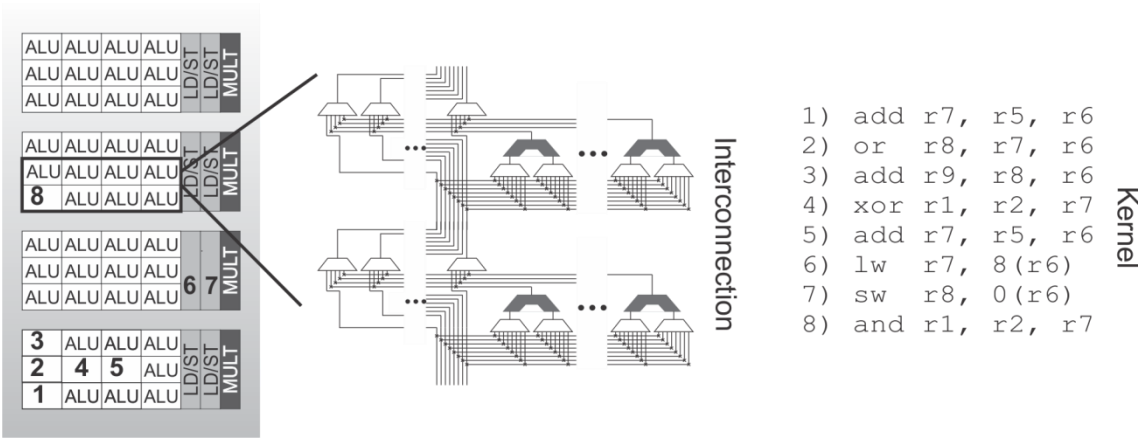
4.5 Array Reconfigurável

A unidade reconfigurável utilizada, baseada na apresentada por (BECK et al., 2008), é uma matriz dinâmica de grão grosso, ou seja, sua reconfiguração ocorre em unidades funcionais completas, fortemente acoplada ao processador (COMPTON; HAUCK, 2002). Ela funciona como uma unidade funcional a mais no estágio de execução do pipeline, utilizando um mecanismo similar ao Chimaera (HAUCK, 1997). Dessa forma, nenhum acesso externo ao *array* (com respeito ao processador) é necessário. O *array* reconfigurável já provou, em trabalhos anteriores, ser capaz de acelerar diversos tipos de aplicações, mesmo aquelas com baixo nível de ILP (*Instruction Level Parallelism*). Uma análise em alto nível das aplicações (discutindo seu grau de paralelismo e o quão *data/control flow* elas são) e seu potencial de otimização foi feita em (BECK et al., 2008), que também aponta as vantagens de usar *arrays* reconfiguráveis de grão grosso. Em (BECK et al., 2008) foi provado que o *array* aqui usado é capaz de acelerar aplicações com alta taxa de instruções de controle (pequenos blocos básicos que limitam a quantidade de ILP disponível para exploração) e baixo ILP pois tira vantagem do agrupamento de instruções. Além disso, (BECK; CARRO, 2007) demonstrou que esse sistema reconfigurável apresenta um ILP maior que um processador superescalar 4-*issue*.

Uma visão geral de sua organização é mostrada na Figura 4.1(d). O *array* é bidimensional, e cada instrução é alocada na interseção de uma linha e coluna. Se duas instruções não tiverem dependência de dados, elas podem ser executadas em paralelo, na mesma linha. Cada coluna é homogênea, contendo um determinado número de unidades funcionais de mesmo tipo (ALU, multiplicadores, etc.). Dependendo do atraso de cada unidade funcional, mais que uma operação pode ser executada no ciclo equivalente ao processador. É o caso de instruções aritméticas simples. Por outro lado, instruções complexas, tais como multiplicações, podem levar mais tempo. O atraso é dependente da tecnologia e da forma que a unidade funcional é implementada. Unidades de *Load/Store* permanecem em um grupo diferente do *array*. O número de unidades em paralelo depende da quantidade de portas disponíveis na memória. A versão utilizada do *array* reconfigurável não suporta operações de ponto flutuante.

O mecanismo de interconexão do *array* pode ser visto em detalhes na Figura 4.5. Para os operandos de entrada, há um barramento de operandos que recebe os valores vindos dos registradores. Cada linha deste barramento é conectada a todas as unidades funcionais por meio de multiplexadores, chamados de multiplexadores de entrada. Na sequência das unidades funcionais temos os multiplexadores de saída, um para cada linha, que selecionam para qual linha do barramento irá o resultado da operação. Como alguns dos valores do contexto de entrada ou valores antigos gerados por operações anteriores podem ser reutilizados por outras unidades funcionais, a primeira entrada de cada multiplexador de saída sempre guarda o resultado anterior da mesma linha de barramento.

Figura 4.5 – Mecanismo de interconexão do array, e exemplo de execução de uma sequência de instruções



4.6 Segundo Nível de Tradução Binária

O hardware do segundo nível de tradução binária foi estendido de (BECK, 2008). Ele começa a trabalhar na primeira instrução encontrada após a execução de um *branch*, e para a tradução assim que encontra uma instrução não suportada ou outro *branch*. Se mais do que três instruções são encontradas, uma nova entrada na cache (baseada em FIFO) é criada e os dados de um buffer especial, utilizado para manter a tradução temporária, são salvos na Cache de Tradução (TCache). O mecanismo proposto baseia-se em um conjunto de tabelas, utilizadas para guardar a informação na sequência das instruções processadas, ou seja, configuração para as unidades funcionais e o roteamento dos operandos entre elas. Outras tabelas temporárias são também necessárias, no entanto, por elas serem usadas somente na

fase de detecção (não durante a reconfiguração), sua informação não é salva na cache de contexto.

O algoritmo de tradução binária tira vantagem da estrutura hierárquica do *array*: para cada instrução que chega, a primeira tarefa é a verificação de dependências verdadeiras. Os operandos de origem são comparados com uma lista de registradores de destino em cada linha. A tabela de dependências é composta de pequenos bitmaps (um por linha). Cada bitmap mantém informação sobre todos registradores destino das instruções alocadas na linha correspondente, assim não há necessidade de armazenar a informação individualmente para cada instrução. Se a linha atual e todas linhas acima não tem aquele registrador destino igual a um dos operandos origem da instrução atual, a instrução pode ser alocada naquela linha, na primeira coluna disponível mais à esquerda, de acordo com o grupo de instruções à qual pertence. Em seguida, os operandos origem/destino de/para o barramento de operandos são configurados através de multiplexadores de entrada/saída. A tabela de entrada informa que operandos do contexto de entrada deve ser alocado para a unidade funcional responsável pela execução da instrução. Cada entrada nessa tabela guarda dois valores, visto que há dois operandos origem para cada unidade funcional. O contexto de entrada é uma tabela indireta, assim a primeira linha do barramento de contexto não irá necessariamente transmitir o valor para o registrador R1. A tabela de saída informa qual valor cada linha do barramento de contexto irá receber. Essa tabela é diferente da tabela de entrada: na tabela de entrada, os multiplexadores são responsáveis por escolher quais valores das linhas do barramento de contexto serão alocados para cada unidade funcional. A tabela de saída, por outro lado, informa quais valores do conjunto total de unidades funcionais que compõem cada linha serão enviados para as linhas do barramento de contexto.

Finalmente, a tabela de contexto é responsável por manter a informação de quais valores do barramento de contexto serão escritos de volta. Essa tabela possui somente duas linhas (inicial e corrente). A primeira representa o contexto de entrada, e será usada na fase de reconfiguração para a busca de operandos. A segunda é usada durante a fase de detecção. Seu estado final representa quais valores e quando eles estarão prontos para serem escritos de volta enquanto o *array* está executando uma dada configuração.

5 RESULTADOS

5.1 Ambiente de Simulação

Para a realização dos testes, foi utilizado o processador MIPS R3000 com uma memória cache de 32 Kbytes unificada para dados/instruções. O *array* reconfigurável tem 48 linhas e 16 colunas. Cada coluna tem 8 ULAs, 6 unidades Load/Store e 2 multiplicadores. A Cache de Tradução pode guardar 512 configurações. Em trabalhos anteriores (BECK et al., 2008), essa configuração mostrou ser o ponto ótimo entre custo de área e aumento de performance.

O conjunto de *benchmaks* Mibench foi executado em um sistema operacional Linux, através do simulador GEM5. Todos benchmarks foram compilados utilizando GCC com otimização –O3 e linkados estaticamente. No simulador GEM5 foram gerados traces de execução para cada programa. Os arquivos trace gerados foram utilizados como entrada para o simulador BT de primeiro nível. O simulador BT de primeiro nível então foi responsável pela tradução de cada programa, fornecendo os dados de desempenho (número de instruções nativa / instruções MIPS). Após isso, simuladores com precisão de ciclo (descritos em SystemC em nível RTL) foram usados para o segundo nível BT, arquitetura reconfigurável e o processador MIPS.

No simulador BT de primeiro nível foi implementado todo conjunto de instruções ARMv4, com exceção das instruções de coprocessador. Já para a arquitetura PowerPC, foram cobertas 78 das 131 instruções da categoria *basic* do conjunto de instruções ver. 2.06b. Para ambas arquiteturas, o conjunto foi suficiente para execução de todos *benchmarks*.

Um protótipo em hardware do simulador foi implementado em uma FPGA Xilinx Virtex5 e usado para extrair dados de área. Os resultados foram então convertidos em número de portas lógicas, baseados na biblioteca TSMC 90nm, permitindo assim a comparação com os resultados extraídos de (FAJARDO et al., 2011). Nesse protótipo, foram implementadas todas instruções lógicas, aritméticas, load e store da arquitetura ARM (total de 80% das instruções executadas), sendo os resultados extrapolados para todo o conjunto. Para a arquitetura PowerPC os dados de área extrapolados a partir da implementação ARM.

5.2 Impacto das Extensões no MIPS

Como explicado anteriormente, o processador MIPS foi modificado para dar suporte adicional ao sistema de tradução binária. Para título de comparação, é mostrado o impacto das extensões MIPS para execução x86 do trabalho desenvolvido por (FAJARDO, 2011). A

Figura 5.1 mostra o número médio de instruções MIPS geradas de uma instrução x86 quando não há suporte para a tradução (hardware original); quando há suporte somente para a computação dos EFlags (bits que guardam resultado de operações e estados do processador); e quando outras modificações são também incluídas (ISA estendida). O número de instruções MIPS geradas por instrução x86 chega a dezenas sem as otimizações, o que impacta numa perda de desempenho do código executado MIPS. Nesse gráfico é possível ver a perda de desempenho que o segundo nível terá que superar.

Considerando a ISA ARM, com exceção das instruções condicionais e poucas outras (por exemplo, aquelas que usam modo de endereçamento com pré ou pós incremento), na maioria das instruções são geradas duas instruções MIPS para cada instrução ARM. A Figura 5.2 demonstra o impacto da implementação de suporte em hardware para tradução MIPS, considerando: tradução para o MIPS nativo (hardware original), tradução ARM para MIPS com suporte a execução condicional; tradução ARM com suporte a execução condicional e suporte ao segundo operando complexo. Podemos notar uma redução de 28% do total de instruções traduzidas quando a execução condicional é implementada em MIPS e uma redução adicional de 16% com o suporte ao segundo operando flexível. Na média, 44% menos instruções foram geradas com essas simples modificações de hardware.

Figura 5.1 – O impacto do uso de suporte em hardware para a tradução x86/MIPS

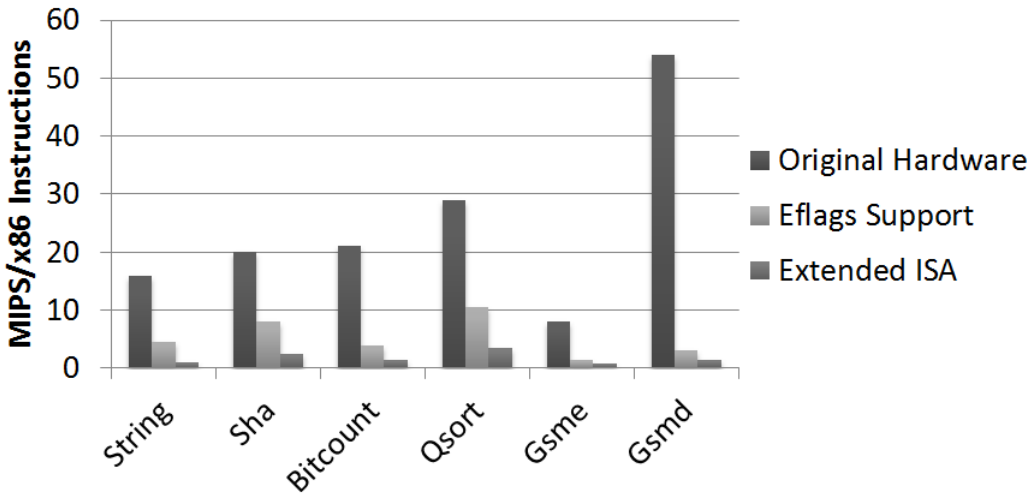
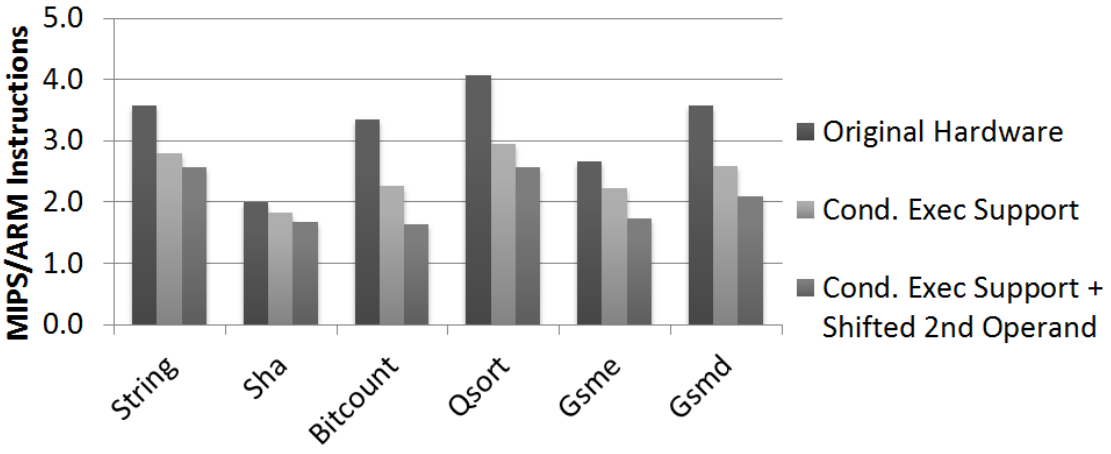
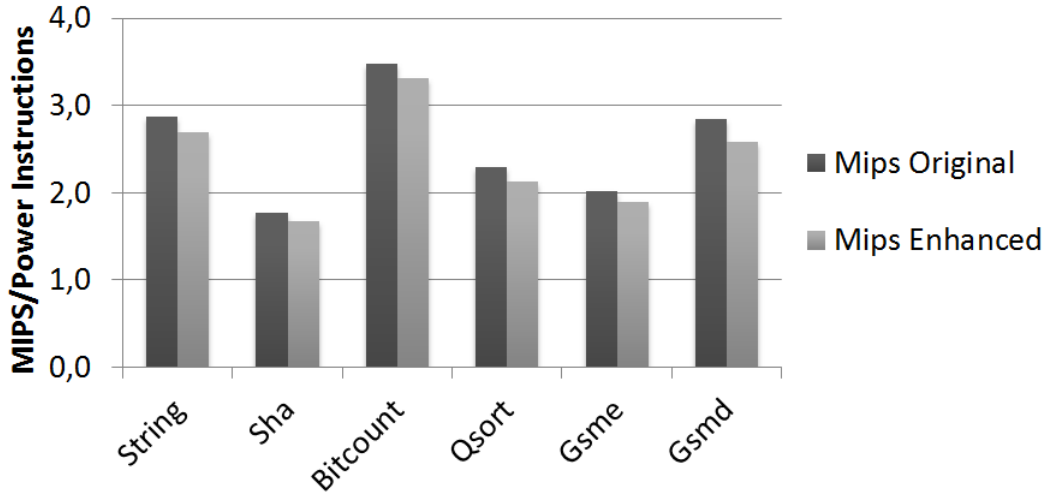


Figura 5.2 – O impacto do uso de suporte em hardware para tradução ARM/MIPS



Considerando a ISA PowerPC, a Figura 5.3 mostra o impacto das modificações no MIPS para suportar a tradução binária. Como já mencionado, as modificações foram feitas para suportar instruções que fizeram uso do *Conditional Register* e *Fixed-point Exception Register*, aproximadamente 10% do total de instruções. Na média, notamos um total de 8% de redução no número de instruções geradas com essas modificações.

Figura 5.3 – O impacto do uso de suporte em hardware para tradução PowerPC/MIPS



5.3 Desempenho

Considerando que a adição do primeiro nível de tradução ao sistema não afeta o caminho crítico do processador MIPS, as métricas de desempenho são baseadas em quantas instruções MIPS são geradas por instruções fonte. A Tabela 5.1, Tabela 5.2 e Tabela 5.3

mostram os dados de desempenho de tradução para as arquiteturas x86, ARM e PowerPC respectivamente, sendo que a Figura 5.4, Figura 5.5 e Figura 5.6, representam graficamente o comparativo dos mesmos dados de desempenho. Ambos, tabelas e gráficos, representam os seguintes cenários:

- Native: representa execução do código nativo MIPS diretamente no processador MIPS.
- Native + RA: execução de código nativo MIPS com aceleração da arquitetura reconfigurável. Nesse caso, o primeiro nível é ignorado: somente o segundo nível de tradução mais a Arquitetura Reconfigurável (RA) são utilizados;
- x86 + First Level BT / ARM + First Level BT / PowerPC + First Level BT: execução do código x86, ARM ou PowerPC sendo traduzido para MIPS mas não otimizado pelo sistema reconfigurável;
- x86 + Two-Level BT / ARM + Two-Level BT / PowerPC + Two-Level BT: execução do código x86, ARM ou PowerPC no sistema proposto, considerando os dois níveis de tradução.

O número de ciclos tomados para executar o código nativo no processador MIPS foi normalizado para 100% (Nativa). A execução Nativa + RA apresenta ganhos de desempenho de mais de duas vezes na média. Por exemplo, o algoritmo SHA apresenta um ganho de mais de 3.43 vezes, *Bitcount* tem ganhos de 2.42 vezes, enquanto o GSM *Encoder* apesenta *speedup* de 1.53 vezes, que é o pior caso no conjunto de benchmarks testados.

Agora, na Tabela 5.1 e Figura 5.4, é considerado o código x86 sendo traduzido para MIPS mas não otimizado pelo sistema reconfigurável (x86 + First Level BT). Como era esperado, há perda de desempenho devido ao mecanismo de tradução. Por exemplo, no *GSM Decoder*, uma desaceleração de mais de 2 vezes é notada quando comparamos com a execução nativa do mesmo algoritmo em código MIPS. No entanto, a execução do código x86 no sistema proposto (x86 + Two-level BT) é mais rápida quando comparada com a execução nativa do mesmo algoritmo diretamente no processador MIPS padrão, portanto amortizando o custo original a tradução binária. A aceleração sobre a execução MIPS padrão variou entre 1,11 e 1.96 vezes. Na média, os ganhos de desempenho ficaram em 45%.

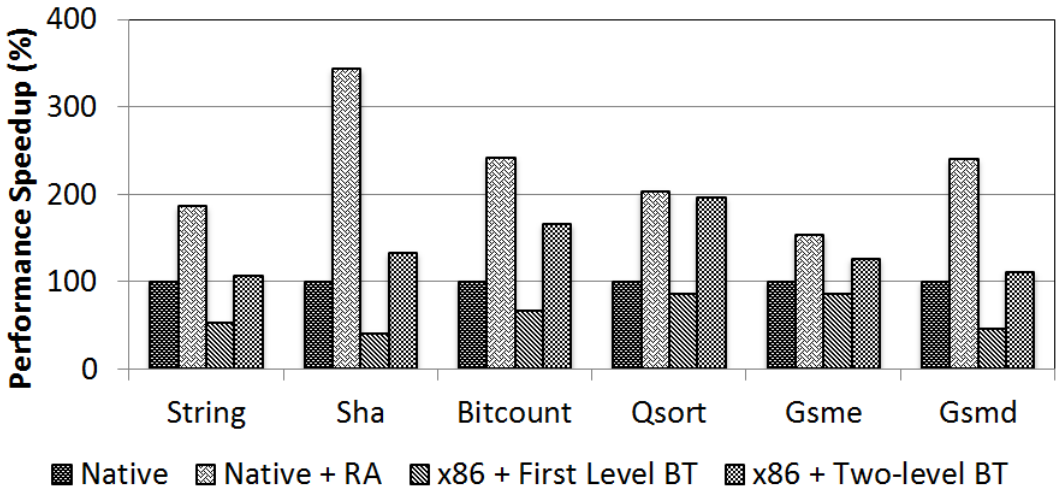
É importante notar que os níveis de otimização do array reconfigurável com o código nativo (código fonte diretamente compilado para MIPS) e com o código pós traduzido de x86 para MIPS são muito similares. Isso pode ser observado ao comparar a primeira e segunda barras de cada algoritmo com a terceira e quarta, na Figura 5.4, ou primeira e segunda coluna

da Tabela 5.1. Isso significa que o *array* é capaz de otimizar códigos independentemente se foi gerado por um compilador "poderoso" (nesse primeiro caso, o GCC), ou por um simples (o que seria equivalente a geração do primeiro nível do tradutor binário).

Tabela 5.1 – Desempenho x86

Algoritmo	Native (%)	Native + RA (%)	x86 + First Level BT (%)	x86 + Two-level BT (%)
String Search	100	186	53	106
Sha	100	343	40	133
Bitcount	100	242	66	166
Qsort	100	203	86	196
Gsme	100	153	86	126
Gsmd	100	240	46	111

Figura 5.4 – Desempenho x86



De mesma maneira, a Tabela 5.2 e Figura 5.5 demonstram o desempenho para a tradução da arquitetura ARM. O desempenho sobre o a execução MIPS padrão varia de 0.73 a 2.05. Na média, os ganhos de desempenho são de 18%. Analisando os algoritmos com pior desempenho (*String*) e melhor desempenho (*Sha*), é evidente que o desempenho está inversamente proporcional à incidência de instruções de *branches*.

Na Tabela 5.3 e Figura 5.6 podemos notar o desempenho do processador PowerPC sobre MIPS, que varia de 0.69 a 2.06, com ganhos médios de 3%. Comparando os algoritmos com pior (*Bitcount*) e melhor desempenho *(Sha)* é possível verificar que o primeiro é composto em sua maioria de instruções simples, que geram uma ou duas instruções MIPS,

enquanto o segundo utiliza instruções mais complexas, que em sua maioria, geram 6 instruções MIPS.

Tabela 5.2 – Desempenho ARM

Algoritmo	Native (%)	Native + RA (%)	ARM + First Level BT (%)	ARM + Two-level BT (%)
String Search	100	186	39	73
Sha	100	343	60	205
Bitcount	100	242	62	149
Qsort	100	203	39	79
Gsme	100	153	58	88
Gsmd	100	240	48	115

Figura 5.5 – Desempenho ARM

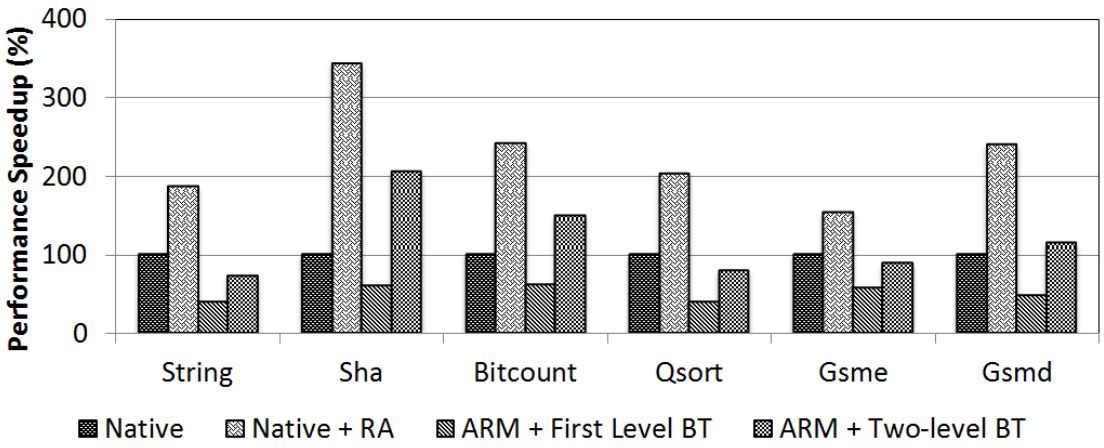
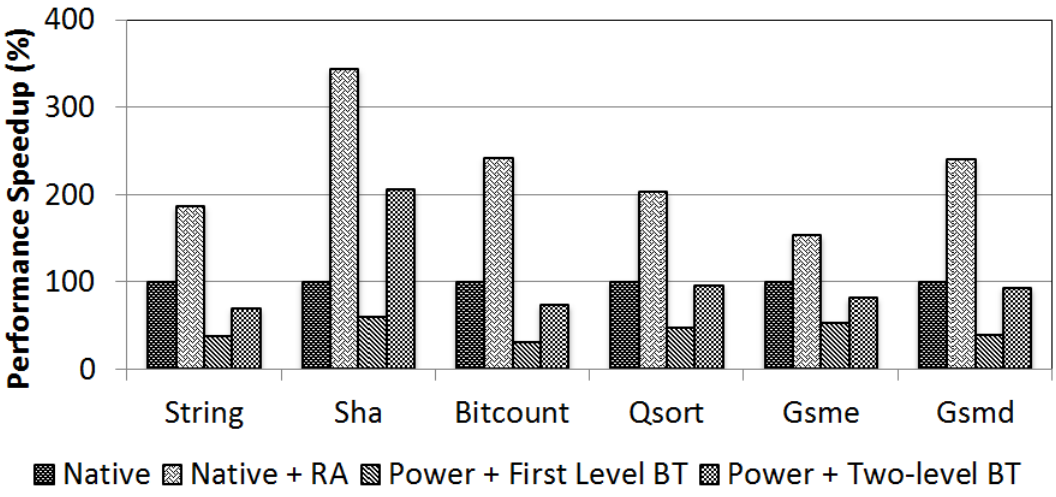


Tabela 5.3 – Desempenho PowerPC

Algoritmo	Native (%)	Native + RA (%)	PowerPC First Level BT (%)	PowerPC + Two-level BT (%)
String Search	100	186	37	69
Sha	100	343	60	206
Bitcount	100	242	30	73
Qsort	100	203	47	95
Gsme	100	153	53	81
Gsmd	100	240	39	93

Figura 5.6 – Desempenho PowerPC

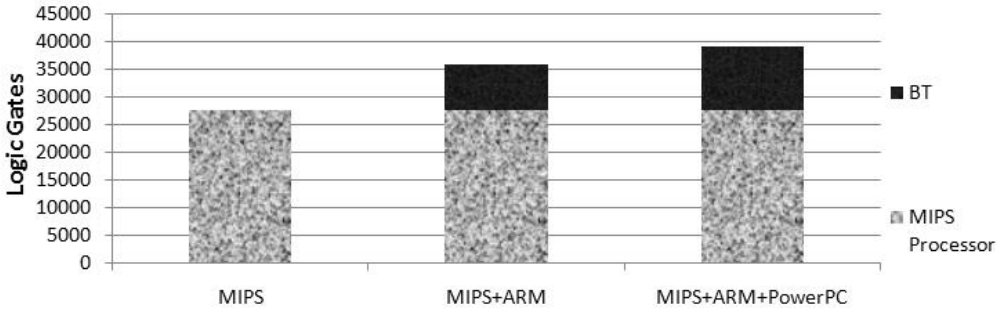


5.4 Área

Primeiramente apresentamos os resultados do primeiro nível do tradutor binário comparando sua área com a área do processador MIPS padrão. Também apresentamos o sistema inteiro, com o segundo nível de tradução e o array reconfigurável.

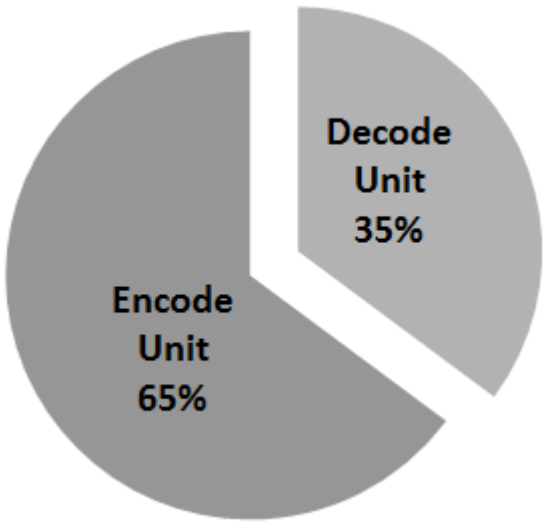
Consideremos os três cenários, mostrados na Figura 5.7, primeiro a área do processador MIPS sozinho, então adicionando o primeiro nível de tradução binária para ARM e finalmente com a adição do tradutor para arquitetura PowerPC. Os resultados mostram um acréscimo de 29% quando adicionado a primeira arquitetura (ARM), e um adicional de área de 12% ao adicionar suporte para a segunda arquitetura (PowerPC). Portanto, um aumento de 41% em área para o suporte das duas arquiteturas. O tradutor x86, devido à complexidade inerente de sua arquitetura, foi implementado como um módulo a parte, e representa uma adição de 83% em área sobre o processador MIPS.

Figura 5.7 – Custo de área do sistema proposto



Para a análise de distribuição de área do primeiro nível de tradução, mostrada na Figura 5.8, consideramos somente as ISAs ARM e PowerPC implementadas juntas. Nesse cenário, do total de 11510 portas lógicas necessárias para a implementação, 65% dessa área é utilizada para a unidade de codificação e 35% para a unidade de decodificação. A unidade de decodificação em si compreende dois módulos distintos, um para decodificação ARM e outro para decodificação PowerPC. Ambas arquiteturas utilizam um número similar de portas lógicas. Como será discutido a seguir, o módulo que cresce mais, conforme novas arquiteturas são adicionadas ao sistema, é módulo de decodificação.

Figura 5.8 – Distribuição de área do primeiro nível de tradução



Quando considerado o sistema completo, temos a distribuição de área da Tabela 5.4. A área do tradutor representa menos que 6% do sistema completo.

Tabela 5.4 – Distribuição de área do sistema completo

Unit	Área (gates)	%
First-Level BT (x86)	22406	2.05 %
First-Level BT (ARM & PowerPC)	11510	1.05%
MIPS	26866	2.46%
Second-Level BT	15264	1.40%
Recovery Array	1017620	93.05%

5.5 Escalabilidade

Façamos agora uma análise da escalabilidade do sistema considerando múltiplas arquiteturas. O objetivo seria encontrar uma representação intermediária para todos conjuntos de instruções, que é encontrada após a decodificação de uma instrução, e seu posterior mapeamento para instruções MIPS pela unidade de codificação. Dessa forma, a unidade de decodificação deve ser reprojetada para cada nova ISA adicionada, enquanto a unidade de codificação é reusada.

Em nossa implementação, o processo de decodificação é realizado tomando-se instruções de tamanho fixo (32 bits de largura), extraindo sua operação e analisando seus campos. Para ambas implementações ARM e PowerPC, esse módulo consumiu entre 200 e 250 LUTs. O fator que mais influencia na área dessa unidade é o número possível de codificações e de operações suportadas pela arquitetura nativa. Dois fatores que aumentam a complexidade da unidade de decodificação são: instruções de largura variada na arquitetura origem (ISA x86) e a necessidade de mapear um número maior de registradores da arquitetura nativa para a arquitetura alvo.

A unidade de codificação foi projetada como uma máquina de estados finitos, na qual cada estado corresponde a uma instrução MIPS e as transições permitem que diferentes sequências de instruções sejam geradas. Uma vez que todos estados foram implementados, o número de registradores utilizados nesse módulo não cresce com a adição de uma nova arquitetura. Cada nova ISA adicionada ao sistema irá somente alterar a transição entre os estados, pouco influenciando o número de portas lógicas necessárias para implementar a transição para o próximo estado lógico.

6 CONCLUSÃO E TRABALHOS FUTUROS

A tradução binária tem surgido como uma solução para o impasse entre o uso de novas arquiteturas e a manutenção da base de software legado de ISAs tradicionais. Neste trabalho foi explorado um sistema de tradução binária totalmente flexível, completamente implementado em hardware, no qual ambas as arquiteturas, nativa e alvo, podem ser facilmente substituídas. Foi provado ser possível a execução de aplicações ARM, PowerPC ou x86 em uma arquitetura não nativa de forma totalmente transparente, na qual nenhuma intervenção é necessária, e com ganhos médios de desempenho em todas as arquiteturas.

Os dados de desempenho foram adquiridos utilizando o simulador desenvolvido para o primeiro nível de tradução juntamente com os dados do simulador existente do segundo nível. Para as duas arquiteturas, considerando a média de todos benchmarks testados, o sistema apresentou ganhos de desempenho. No entanto, alguns algoritmos apresentaram perda de desempenho, perdas essas inseridas exclusivamente pelo primeiro nível de tradução. Todavia, o objetivo principal foi alcançado: compatibilidade binária com execução transparente.

Considerando a arquitetura PowerPC, o sistema apresentou ganhos médios de 3% de desempenho. No algoritmo *Sha*, que apresentou o melhor desempenho no primeiro nível (60% da execução nativa), mais de 50% das instruções PowerPC são compostas por instruções dos tipos: add, xor, lwz e or, que geram uma ou duas instruções MIPS. Por outro lado, no algoritmo *Bitcount*, que apresentou o pior desempenho (30% da execução nativa), a maioria das instruções PowerPC são: add, rlwinm, addi e bc. A instrução rlwinm *(rotate left word immediate then and with mask*) é uma instrução complexa, que gera 6 instruções MIPS. Já a instrução bc *(branch conditional*) pode gerar até 7 instruções MIPS.

O desempenho para execução de código ARM foi melhor, 18% superior sobre a execução nativa. O algoritmo com pior desempenho no tradutor de primeiro nível foi o *String Search*, com 39% do desempenho sobre a execução nativa, e cuja taxa de instruções de *branch* chegou a 24%. O algoritmo com melhor desempenho, o Sha foi o que apresentou menor incidência de instruções de *branch* e melhor desempenho, 60% do desempenho da execução nativa.

Notam-se dois fatores que distinguem as duas arquiteturas, PowerPC e ARM, no processo de tradução. A primeira é a quantidade de registradores. Enquanto no ARM é possível realizar o mapeamento completo de registradores no MIPS, no PowerPC é preciso salvar parte dos registradores em memória, adicionando instruções de *load/store* no processo de tradução. O mapeamento de registradores PowerPC em memória onerou em torno de 5% o

desempenho do tradutor de primeiro nível. O segundo fator, é a quantidade de operações aritméticas complexas que o PowerPC tem superior ao ARM. O ARM não possui operações aritméticas complexas, apenas o segundo operando flexível, e que vale para várias instruções e pode, dessa forma, ser implementado com baixo custo na arquitetura intermediária. Já o PowerPC possui várias instruções aritméticas mais complexas que o MIPS, e não há como adicionar um único hardware que atenda a todas essas instruções.

Em relação à ocupação de área, tivemos um adicional de área de 41% sobre o hardware do MIPS para suporte das duas arquiteturas. A adição de hardware não afetou o caminho crítico do processador MIPS R3000, podendo ser mantida a mesma frequência de operação de 600MHz.

6.1 Trabalhos Futuros

Melhorias e outras linhas de pesquisa se abrem a partir desse trabalho. Como melhorias podemos citar:

- Adicionar suporte a instruções de ponto flutuante;
- Adicionar suporte a interrupções;
- Implementar extensões da arquitetura ARM (SIMD, Thumb);
- Melhorar o mapeamento de registradores, para que não seja necessário mapear registradores em memória;
- Suporte a instruções atômicas;
- Suporte a código auto modificável.

A utilização de uma arquitetura intermediária alternativa à arquitetura MIPS poderia ser investigada, pois apesar de ser uma arquitetura regular que facilita a montagem de instruções, possui certas deficiências que mostraram onerar o processo de tradução. Não utiliza, por exemplo, *flags* de estados da ULA na execução das instruções.

Um estudo sobre otimizações para o primeiro nível de tradução poderia ser feito, ainda que o objetivo principal seja compatibilidade binária. Na solução atual, a tradução do primeiro nível é dinâmica, instrução por instrução, onerando o desempenho desse nível de tradução.

Ainda, alterações no primeiro nível de tradução poderiam ser feitas para execução de diversas ISAs em um único processador. A implementação atual prevê a execução ARM/MIPS, PowerPC/MIPS ou x86/MIPS.

Pode-se pensar ainda no uso de multiprocessadores heterogêneos, onde uma única ISA é executada em diversos processadores de arquiteturas distintas. Essa arquitetura poderia utilizar o tradutor atual acoplado a um sistema de despacho que encaminharia as instruções para cada processador conforme o tipo de instruções ou do comportamento do código.

REFERÊNCIAS

ALTMAN, E. R.; KAELI, D.; SHEFFER, Y. Welcome to the opportunities of binary translation", IEEE **Computer**, v. 33, n. 3, p. 40-15, mar. 2000.

APPLE INC. Rosetta, Disponível em: < www.apple.com/asia/rosetta/>. Acesso em: 30 mai. 2014.

ARM, ARM7TDMI Technical Reference Manual, 2000, Disponível em: <http://infocenter.arm.com/help/topic/com.arm.doc.ddi0210c/DDI0210B.pdf>. Acesso em: 03 mai. 2014.

BALA, V.; DUESTERWALD, E.; BANERJIA, S. Dynamo: A Transparent Dynamic Optimization System. In: PLDI, 2000, Vancouver, CAN. **Proceedings…** New York, NY: ACM Press, 2000, p. 1-12.

BECK, A. C.; RUTZGI M. B.; GAYDADJIEV, G.; CARRO, L. Transparent reconfigurable acceleration for heterogeneous embedded applications. In: DESIGN, AUTOMATION AND TEST IN EUROPE (DATE), 2008, Munique, ALE**. Procedings…** Washinton, DC: IEEE, 2008. p. 1208-1213.

BECK, A. C.; CARRO, L. Dynamic Reconfiguration with Binary Translation: Breaking the ILP barrier with Software Compatibility. In: DESIGN AUTOMATION CONFERENCE (DAC), 2005, Anaheim. EUA. **Proceedings…** New York, NY: ACM Press, 2005. p. 732-737

BECK, A. C.; CARRO, L. **Dynamic Reconfigurable Architectures and Transparent Optimization Techniques**: Automatic Acceleration of Software Execution. 1st ed. Berlin/Heidelberg: Springer Publishing Company, Incorporated, 2010.

BECK, A.C.S. et al. Run-time adaptable architectures for heterogeneous behavior embedded systems. In: WORKSHOP ON RECONFIGURABLE COMPUTING, 2008, Londres, UK. **Proceedings…** Berlin/Heidelberg: Springer, 2008. p. 111-124

BECK, A.C.S.; CARRO, L. Transparent acceleration of data dependent instructions for general purpose processors. In: IFIP WG 10.5 VLSI-SOC, 2007, Atlanta, USA. **Proceedings…** New York: IEEE, 2007. p. 15-17

BECK, A.C.S.; LANG, L.L.; CARRO, L. **Adaptable Embedded Systems**. New York, NY: Springer Publishing Company, Incorporated, 2012

BUTTAZZO, G. Research trends in real-time computing for embedded systems. **ACM SIGBED Review**, New York, NY, v. 3 n. 3, p. 1-10, jul. 2006.

CHERNOFF, A. et al. FX!32: A Profile-Directed Binary Translator. **IEEE Micro**, v. 18, n. 2, p. 56-64, mar./abr. 1998.

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

DANDAMUDI, S. P. **Guide to RISC Proceseers** for Programmers and Engineers. 2005 ed. New York, NY: Springer Publishing Company, Incorporated, 2005

DEHNERT, J. C. et al. The Transmeta Code Morphing™ Software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In: INTERNATIONAL SYMPOSIUM ON CODE GENERATION AND OPTIMIZATION: FEEDBACK-DIRECTED AND RUNTIME OPTIMIZATION, San Francisco, CA, USA, 2003. **Proceedings…** Washington, DC: IEEE, 2003, p. 15-24

EBCIOGLU, K.; ALTMAN E. R. DAISY: Dynamic compilation for 100% architectural compatibility. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE (ISCA), Denver, CO, USA, 1997. **Proceedings…** New Youk, NY: ACM, 1997. p. 26-37.

EBCIOGLU, K. et al. Dynamic Binary Translation and Optimization. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 50, n. 6, p. 529-548, June, 2001.

FAJARDO JR, J. **Sistema de Tradução Binária de Dois Níveis para Execução Multi-ISA.** 2011. 76 f. Dissertação (Mestrado em Microeletrônica) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2011.

FAJARDO JR, J. et al. Towards a Multiple-ISA Embedded System. **Journal of Systems Architecture**, New York, NY, v. 59, n. 2, p. 103-109,feb. 2013.

FAJARDO JR, J. et al. Towards an Adaptable Multiple-ISA Reconfigurable Processor In: INTERNATIONAL WORKSHOP ON APPLIED RECONFIGURABLE COMPUTING, 2011, Belfast, IRL. Lecture Notes in Computer Science: Reconfigurable Computing: Architectures, Tools and Applications, 2011, v.6578. p.157-168.

FLYNN, M. J.; HUNG, P. Microprocessador design issues: thoughts on the road ahead. **IEEE MICRO**, v. 25, n. 3, p. 16-31, 2005

GEM5. The gem5 Simulator System. Disponível em: < http://www.gem5.org>. Acesso em: 20 mai. 2012.

GUTHAUS, M. R. et al. MiBench: A free, commercially representative embedded benchmark suite. In: IEEE INTERNATIONAL WORKSHOP ON THE WORKLOAD CHARACTERIZATION (WWC-4), 2001, Austin, TX. **Proceedings…** Washinton, DC: IEEE, 2001. p. 3-14.

HAUCK, S. et al. The Chimaera reconfigurable functional unit. In: SYMPOSIUM FPGAS FOR CUSTOM COMPUTING MACHINES, 1997, Napa Valley, CA. **Proceedings…** Washinton, DC: IEEE, 1997. p. 87–96.

HOOKWAY, R. J.; HERDEG, M. A. DIGITAL FX!32: combining emulation and binary translation. **Digital Technical Journal**, Acton, MA, USA, v. 9, n. 1, p. 3-12, jan. 1997.

HU, W. et al. Godson-3: A Scalable Multicore RISC Processor with x86 Emulation. **IEEE Micro**, v.29 n.2, p. 17-29, mar./abr. 2009.

KIM, N. S. et al. Leakage current: Moore's law meets static power. IEEE **Computer**, v. 36, n. 12, p. 68-75, dez. 2003.

KLAIBER, A. The Technology Behind Crusoe Processors. **Transmeta Corporation**, January 2000. Disponível em: <http://web.archive.org> Acesso em: 30 mai. 2014.

KUMAR, A. et al. Iterative Probabilistic Performance Prediction for Multi-Application Multiprocessor Systems. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 29, n. 4, p. 538-551, April, 2010.

LYSECKY, R.; STITT, G.; VAHID, F. Warp Processors. **ACM Transactions on Design Automation of Electronic Systems (TODAES)**, v. 11, n. 3, p. 659-681, July 2006.

MAK, J.; MYCROFT, A. Limits of parallelism using dynamic data dependence graphs. In: WODA, 2009, Chicago, Illinois, USA. **Proceedings…** New York, NY: IEEE, 2009. p. 42-28

MIPS TECHNOLOGIES. MIPS Architecture For Programmers Volume II-A: The MIPS32 Instruction Set, 2013. Disponível em: < http://www.imgtec.com/mips/architectures/mips32.asp>. Acesso em: 30 mai. 2014.

POWER. Power ISA™ Version 2.06 Revision B, 2010. Disponível em: < https://www.power.org/documentation/power-isa-version-2-06-revision-b/>. Acesso em: 30 mai. 2014.

RUTZIG, M. **Gerenciamento Automático de Recursos Reconfiguráveis Visando a Redução de Área e do Consumo de Potência em Dispositivos Embarcados**. 2012. 112 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2011.

SIMA, D. Decisive aspects in the evolution of micoprocessors. **Proceedings of the IEEE**, v. 92, n. 12, p. 1896-1926, 2004.

SITES, R. L. et al. Binary Translation, **Communications of the ACM** , New York, v. 36, n. 2, p. 69-81, Feb. 1993.

XILINX, ISE DESIGN SUITE(2012). Disponível em <http://www.xilix.com>. Acesso em: 05 out. 2012.

ANEXO A PUBLICAÇÕES

CAPELLA, F. M. BRANDALERO, M.; FAJARDO, J. ; BECK, A.C.S. ; CARRO, L.. A Multiple-ISA Reconfigurable Architecture. Design Automation for Embedded Systems Journal, Springer. (SUBMETIDO)

SARTOR, A. ; CAPELLA, F. M. ; BRANDALERO, M. ; CARRO, Luigi ; BECK, A. C. S. . A Transparent Multiple-ISA MPSoC Architecture. In: Workshop on SoCs, Heterogeneous Architectures and Workloads (SHAW/HPCA), 2014, Orlando. Proceedings of the Workshop on SoCs, Heterogeneous Architectures and Workloads, 2014.

CAPELLA, F. M. ; BRANDALERO, M. ; FAJARDO JUNIOR, J. ; BECK, A. C. S. ; CARRO, Luigi . A Multiple-ISA Reconfigurable Architecture. In: 2013 III Brazilian Symposium on Computing Systems Engineering (SBESC), 2013, Niteroi. 2013 III Brazilian Symposium on Computing Systems Engineering. p. 71.