PBIW: Um Esquema de Codificação Baseado em Padrões de Instrução

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Rafael Fernandes Batistella e aprovada pela Banca Examinadora.

Campinas, 28 de Fevereiro de 2008.

Rodolfo Jardim de Azevedo

Rodolfo Jardim de Azevedo (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Júlia Milani Rodrigues – CRB8a / 2116

Batistella, Rafael Fernandes

B32p PBIW: um esquema de codificação baseado em padrões de instrução / Rafael Fernandes Batistella -- Campinas, [S.P. :s.n.], 2008.

Orientador : Rodolfo Jardim de Azevedo

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Memória cache. 2. Compressão de dados (Computação). 3. Arquitetura de computador. 4. Compiladores (Computadores). I. Azevedo, Rodolfo Jardim de. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: PBIW : an encoding technique based on instruction patterns.

Palavras-chave em inglês (Keywords): 1. Cache memory. 2. Data compression (Computer science). 3. Computer architecture. 4. Compiling (Electronic computers)

Área de concentração: Arquitetura de Computadores

Titulação: Mestre em Ciência da Computação

Banca examinadora: Prof. Dr. Rodolfo Jardim de Azevedo (IC-UNICAMP)
Prof. Dr. Paulo César Centoducatte (IC-UNICAMP)
Profa. Dra. Edna Natividade da Silva Barros (UFPE)
Prof. Dr. Guido Costa de Araújo (IC-UNICAMP)

Data da defesa: 28-02-2008

Programa de Pós-Graduação: Mestrado em Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 28 de fevereiro de 2008, pela Banca examinadora composta pelos Professores Doutores:

Profª. Drª. Edna Natividade da Silva Barros
CIN - UFPE

Prof. Dr. Paulo Cesar Centoducatte
IC – UNICAMP

Prof. Dr. Rodolfo Jardim de Azevedo
IC – UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

PBIW: Um Esquema de Codificação Baseado em Padrões de Instrução

Rafael Fernandes Batistella

Fevereiro de 2008

Banca Examinadora:

- Rodolfo Jardim de Azevedo (Orientador)
- Edna Natividade da Silva Barros
  Centro de Informática - UFPE
- Paulo Cesar Centoducatte
  Instituto de Computação - UNICAMP
- Guido Costa Souza de Araújo
  Instituto de Computação - UNICAMP (Suplente)

Resumo

Trabalhos não muito recentes já mostravam que o aumento de velocidade nas memórias DRAM não acompanha o aumento de velocidade dos processadores. Mesmo assim, pesquisadores na área de arquitetura de computadores continuam buscando novas abordagens para aumentar o desempenho dos processadores. Dentro do objetivo de minimizar essa diferença de velocidade entre memória e processador, este trabalho apresenta um novo esquema de codificação baseado em instruções codificadas e padrões de instruções - PBIW (Pattern Based Instruction Word). Uma instrução codificada não contém redundância de dados e é armazenada em uma I-*cache*. Os padrões de instrução, de forma diferente, são armazenados em uma nova *cache*, chamada *Pattern cache* (P-*cache*) e são utilizados pelo circuito decodificador na preparação da instrução que será repassada aos estágios de execução. Esta técnica se mostrou uma boa alternativa para estilos arquiteturais conhecidos como arquiteturas VLIW e EPIC. Foi realizado um estudo de caso da técnica PBIW sobre uma arquitetura de alto desempenho chamada de 2D-VLIW. O desempenho da técnica de codificação foi avaliado através de experimentos com programas dos *benchmarks* MediaBench, SPECint e SPECfp. Os experimentos estáticos avaliaram a eficiência da codificação PBIW no aspecto de redução de código. Nestes experimentos foram alcançadas reduções no tamanho dos programas de até 81% sobre programas codificados com a estratégia de codificação 2D-VLIW e reduções de até 46% quando comparados à programas utilizando o modelo de codificação EPIC. Experimentos dinâmicos mostraram que a codificação PBIW também é capaz que gerar ganhos com relação ao tempo de execução dos programas. Quando comparada à codificação 2D-VLIW, o *speedup* alcançado foi de até 96% e quando comparada à EPIC, foi de até 69%.

Abstract

Past works has shown that the increase of DRAM memory speed is not the same of processor speed. Even though, computer architecture researchers keep searching for new approaches to enhance the processor performance. In order to minimize this difference between the processor and memory speed, this work presents a new encoding technique based on encoded instructions and instruction patterns - PBIW (Pattern Based Instruction Word). An encoded instruction contains no redundancy of data and it is stored into an I-*cache*. The instruction patterns, on the other hand, are stored into a new *cache*, named Pattern *cache* (P-*cache*) and are used by the decoder circuit to build the instruction to be executed in the execution stages. This technique has shown a suitable alternative to well-known architectural styles such as VLIW and EPIC architectures. A case study of this technique was carried out in a high performance architecture called 2D-VLIW. The performance of the encoding technique has been evaluated through trace-driven experiments with MediaBench, SPECint and SPECfp programs. The static experiments have evaluated the PBIW code reduction efficiency. In these experiments, PBIW encoding has achieved up to 81% code reduction over 2D-VLIW encoded programs and up to 46% code reduction over EPIC encoded programs. Dynamic experiments have shown that PBIW encoding can also improve the processor performance. When compared to 2D-VLIW encoding, the speedup was up to 96% while compared to EPIC, the speedup was up to 69%.

Agradecimentos

Primeiramente gostaria de agradecer a DEUS que, com certeza, me acompanhou durante todo o desenvolvimento deste mestrado e também durante toda a minha vida. Sem ELE não seria possível concluir este grande desafio.

Devo toda minha gratidão à minha namorada Amanda que, desde as primeiras disciplinas que cursei, tem sido paciente com minha "ausência presencial" pois, mesmo estando juntos, boa parte do tempo minha atenção estava voltada ao mestrado. Nos momentos difíceis a Amanda teve um papel decisivo para me ajudar a buscar forças e chegar ao dia de hoje para concluir este mestrado.

Meus pais Antônio e Maria e, meus irmãos Rafaelle e Diego, sempre tiveram, durante toda a minha vida, um papel muito importante nas minhas conquistas. Os conselhos de meus pais sobre meu futuro profissional tiveram grande influência para eu decidir iniciar este mestrado. Durante todo o mestrado eles também me apoiaram bastante e souberam me ajudar quando eu mais precisava. Minha irmã sempre foi um exemplo de perseverança e sucesso para mim. Além disso, mesmo estando longe, ela sempre me apoiou durante todo o mestrado. Meu irmão mais novo sempre foi um motivo de alegria para mim. Em todas as vezes que eu fui visitá-lo em Franca, desde pequeno, ele soube respeitar os momentos em que eu precisava estudar e ajudou a me distrair e pensar em outras coisas nos momentos difíceis.

Gostaria de agradecer ao professor Rodolfo pela sua ajuda e paciência desde à nossa primeira conversa em 2002, onde ele me explicou suas áreas de pesquisa e trabalhos em andamento, até hoje, nas últimas revisões desta dissertação. Também foi ele o responsável por despertar em mim um grande interesse pela área. Agradeço pelas dicas e por cada reunião semanal nas quais pude aprender muito.

No ambiente da UNICAMP eu gostaria muito de agradecer ao meu amigo Ricardo, ex-aluno de doutorado do IC. Desde o primeiro trabalho que fizemos juntos em uma disciplina sobre VHDL ele me ajudou bastante. Ao iniciar os trabalhos de mestrado, comecei estudando parte do trabalho dele. Escrevemos artigos juntos e durante todo meu mestrado, até a escrita da dissertação, o Ricardo colaborou muito comigo e teve bastante paciência ao esclarecer minhas dúvidas.

Gostaria de agradecer também ao Dr. Cincinato, que foi meu terapeuta e me ajudou muito durante todo o período deste mestrado. Infelizmente ele partiu desta vida de forma inesperada sem poder participar comigo deste momento tão importante.

Também agradeço muito ao CPqD, onde trabalho desde 2001, principalmente por me liberar para assistir as aulas das disciplinas do mestrado e à UNICAMP pela infraestrutura, pelos docentes e pelos funcionários que ajudaram a tornar possível a realização deste mestrado.

Lista de Acrônimos

2D-VLIW: Arquitetura de alto desempenho utilizada como referência durante o desenvolvimento deste trabalho de mestrado.

ARC: Argonaut RISC Core. Processador RISC configurável e de propósito geral, desenvolvido pela empresa ARC International.

ARM: Acorn RISC Machine. Processador RISC de 32 bits, desenvolvido pela empresa ARM Limited.

AVR: Processador RISC de 8 bits, desenvolvido pela empresa Atmel.

CISC: Complex Instruction Set Computer. Um modelo de arquitetura de processadores baseado em um conjunto de instruções complexas, capazes de executar várias tarefas em uma única instrução.

DRAM: Dynamic Read Access Memory. Memória de acesso dinâmico.

DSP: Digital Signal Processor. Processador especializado, desenvolvido especificamente para processamento de sinais digitais, geralmente em computação de tempo real.

EPIC: Explicitly Parallel Instruction Computing. Um modelo de arquitetura que explora paralelismo em nível de instrução através de diversas otimizações de compilação. Neste modelo as instruções são compactas e compostas por operações dependentes e independentes.

GRF: Global Register File. Banco de registradores globais.

HMDES: Hardware Machine Description language. Linguagem para descrição de arquiteturas de processadores acoplada ao compilador Trimaran.

HPL-PD: HP Labs PlayDoh. Processador parametrizável que implementa o modelo arquitetural EPIC para execução de operações de um programa.

IA64: Intel Architecture. Arquitetura Intel de 64 bits, sendo a primeira implementação

EPIC disponível comercialmente.

IBM: International Business Machines. Empresa multinacional, com origem nos Estados Unidos, voltada para a área de informática.

ILR: Instruction Level Reuse. Técnica de Reuso de Instruções.

IM: Instruction Memoization. Técnica de Reuso de Instruções.

IPC: Instruções Por Ciclo. Unidade de medida utilizada para avaliar o desempenho dos processadores.

IR: Instruction Reuse. Técnica que reutiliza resultados de instruções previamente calculados, ao invés de executar novamente as instruções.

LSC: Laboratório de Sistemas de Computação, localizado no Instituto de Computação da UNICAMP.

MIPS: Microprocessor without Interlocked Piped Stages. Designa uma família de microprocessadores que implementam o padrão RISC.

NOP: No Operation. Tipo de instrução que não realiza nenhuma operação, utilizada quando é necessário aguardar alguns ciclos antes da próxima instrução.

PA-RISC: Precision Architecture - Reduced Instruction Set Computing. Processador RISC desenvolvido pela HP.

PBIW: Pattern Based Instruction Word. Esquema de codificação baseado no reuso de padrões de instrução, proposto nesta Dissertação.

PDP-11: Série de miniprocessadores CISC comercializados pela Digital Equipment Corporation.

RB: Reuse Buffer. Buffer de reuso, utilizado para armazenar resultados que serão reutilizados na técnica de Reuso de Instruções.

RISC: Reduced Instruction Set Computer. Um modelo de arquitetura de processadores baseado em um conjunto de instruções reduzido.

RST: Register Source Table. Estrutura utilizada na técnica de Reuso de Instruções para fazer o mapeamento entre uma instrução consumidora e a última instrução produtora de valores de um determinado registrador.

RT: Registrador Temporário. É o registrador que armazena resultados temporários das computações na arquitetura 2D-VLIW. Existem vários bancos de RTs no interior da ma-

triz de UFs 2D-VLIW.

RUF: Registrador da Unidade Funcional . Registrador presente em cada unidade funcional da arquitetura 2D-VLIW.

SPARC: Scalable Processor ARChitecture. Arquitetura de processador desenvolvida pela Sun em 1987 baseada na arquitetura RISC.

SPEC: Standard Performance Evaluation Corporation. Uma empresa que comercializa uma família de programas benchmarks muito usada na avaliação do desempenho de processadores. Além disso, SPEC também define um conjunto de métricas que podem ser usadas na avaliação de desempenho de arquiteturas de processadores.

SPECfp: Conjunto de programas SPEC com valores de ponto flutuante.

SPECint: Conjunto de programas SPEC com valores inteiros.

TMS: Família de processadores DSP desenvolvidos pela Texas Instruments.

TP: Tabela de Predicados. Estrutura utilizada para mapeamento entre registradores globais e registradores de predicado, na arquitetura 2D-VLIW.

UF: Unidade Funcional. É a unidade que executa o processamento das operações em várias arquiteturas como, por exemplo, 2D-VLIW.

ULA: Unidade de Lógica e Aritmética. Elemento de hardware responsável pela execução de operações aritméticas e lógicas.

VAX: Processador CISC desenvolvido pela Digital Equipment Corporation (DEC) em meados dos anos 70.

VLIW: Very Long Instruction Word. Um estilo de arquitetura de processadores que buscam instruções longas na memória e executam as operações que compõem essas instruções de maneira paralela.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Trabalhos não muito recentes já mostravam que a taxa de melhora na velocidade dos microprocessadores supera a taxa de melhora na velocidade das memórias DRAM. A velocidade dos processadores aumentava à aproximadamente 80% ao ano, enquanto que a velocidade das DRAMs aumentava apenas 7% ao ano. Entretanto, a comunidade de arquitetura de computadores ainda está amplamente focada em aumentar o desempenho dos processadores [23]. Como resultado, a diferença entre a velocidade do processador e da memória tem crescido exponencialmente levando a um fenômeno conhecido como *Memory Wall* [23, 43, 47].

Recentemente, várias empresas anunciaram que a freqüência do processador não irá mais aumentar como vinha ocorrendo. Da mesma forma, a velocidade dos processadores também já não está mais aumentando como antes. Mesmo assim, o problema de *Memory Wall* será ainda uma preocupação para a comunidade de arquitetura de computadores, uma vez que a quantidade de processadores utilizados em um sistema e a necessidade de leitura da memória têm aumentado.

Um grande número de técnicas foram desenvolvidas na tentativa de minimizar o problema de *Memory Wall*. Muitas delas focaram em novos esquemas de codificação enquanto outras utilizavam técnicas para redução do tamanho dos programas, como as técnicas de compressão. Todas possuem um único objetivo: reduzir a quantidade de dados armazenados na memória principal. Especificamente, estas técnicas têm sido aplicadas em arquiteturas que buscam grandes instruções da memória e em arquiteturas voltadas para

domínios específicos de aplicações, como os sistemas embarcados [24, 26, 31, 34].

1.1 Proposta e Contribuições da Dissertação

Devido ao grande problema imposto pela diferença expressiva de velocidade entre processadores e memórias, esta dissertação de mestrado propõe uma nova abordagem para tentar minimizar o efeito da latência na busca de instruções e seu impacto no desempenho dos programas, utilizando uma estratégia diferente das técnicas de compressão de código e codificação já apresentadas.

Especificamente, este trabalho apresenta a técnica de codificação PBIW (*Pattern Based Instruction Word*), voltada para arquiteturas que buscam instruções grandes da memória como processadores EPIC, arquiteturas reconfiguráveis com várias unidades funcionais e arquiteturas de alto desempenho baseada em matriz de unidades funcionais. A técnica é composta por um algoritmo baseado em fatoração de operandos [2, 9, 11] e por uma memória *cache* chamada de *Pattern Cache* (P-*cache*).

O novo modelo de codificação proposto nesta dissertação é baseado em técnicas para redução de código, esquemas de codificação e reuso de instruções. Parte solução da proposta consiste em utilizar uma *cache* de padrões cuja estrutura foi definida a partir de estudos sobre desempenho em *caches*. Para validar o novo esquema de codificação, foi realizado um estudo de caso através de sua implementação na arquitetura de alto desempenho 2D-VLIW.

Depois das atividades de escalonamento das instruções e alocação dos registradores realizadas pelo *back-end* do compilador, o algoritmo de codificação PBIW extrai operandos redundantes entre as operações, criando uma instrução codificada sem redundância de operandos. Esta instrução é armazenada em uma I-*cache*. O algoritmo mantém o registro da posição original dos operandos criando um padrão de instrução que é armazenado na P-*cache*. Este padrão é uma estrutura de dados que contém informações necessárias para compor a instrução utilizada nos estágios de execução.

Uma instrução PBIW é composta por um valor utilizado na busca do respectivo padrão na P-*cache*, um conjunto de bits de anulação que indicam quais UFs da matriz de execução

devem ser executadas ou anuladas e um conjunto de campos de dados que armazenam números dos registradores e valores imediatos.

O padrão armazena os *opcodes* das operações e um conjunto de valores que indicam posições dos campos de dados da instrução PBIW. Ou seja, o padrão aponta para os campos de dados da instrução PBIW.

Ao adotar a técnica PBIW, a arquitetura passa a buscar instruções pequenas na I-*cache* e, no estágio de decodificação, busca um padrão na P-*cache*. É importante observar que as latências de busca da I-*cache* e da P-*cache* não são as mesmas em todas as buscas. *Misses* na I-*cache* não implicam em *misses* na P-*cache* uma vez que os padrões são reutilizados por instruções diferentes, pois, existe uma sobrejeção entre o conjunto de instruções codificadas e o conjunto de padrões. Depois do estágio de decodificação, uma instrução pode executar suas operações nos elementos de processamento (ou unidades funcionais) do processador.

Os impactos desta nova técnica de codificação foram avaliados através de simulações da I-*cache* e da P-*cache* com programas dos *benchmarks* MediaBench, SPECint e SPECfp. Os resultados mostram que, adotando esta estratégia de codificação, o desempenho (tempo de execução) é até 96% melhor do que a estratégia de codificação da arquitetura 2D-VLIW e até 69% melhor do que a codificação EPIC.

Outros experimentos realizados também mostram que, utilizando a técnica PBIW, o tamanho total do programa codificado, incluindo os dados da P-*cache* é até 81% menor do que o tamanho do programa codificado na estratégia 2D-VLIW e até 46% menor do que o código gerado através do esquema de codificação EPIC.

1.2 Organização da Dissertação

O texto desta dissertação está organizado em cinco capítulos que fornecem toda a base conceitual e empírica para o entendimento completo da técnica PBIW e comprovação de seus resultados. Os parágrafos a seguir descrevem o conteúdo de cada um destes capítulos.

O Capítulo 2 apresenta os trabalhos que formaram o alicerce teórico para o desenvolvimento do novo modelo de codificação PBIW. Neste capítulo são apresentados esquemas

de codificação, técnicas de redução de código, reuso de instruções, desempenho em *caches* e uma breve descrição da arquitetura 2D-VLIW.

O esquema de codificação PBIW é descrito no Capítulo 3. Este capítulo inclui a estrutura da palavra PBIW assim como a estrutura de um padrão. Também são apresentados neste capítulo o algoritmo de codificação e o circuito decodificador utilizado nesta técnica.

No Capítulo 4 são apresentados experimentos envolvendo o esquema de codificação PBIW. Esse capítulo detalha a infraestrutura utilizada, os experimentos realizados e, principalmente, justifica e discute os resultados obtidos para os experimentos estáticos e dinâmicos.

O Capítulo 5 fecha a dissertação apresentando as conclusões sobre o que foi realizado durante o desenvolvimento deste trabalho e relata as contribuições que foram alcançadas. Ao final, sugere-se a extensão desta proposta através de vários projetos que podem ser realizados como trabalhos futuros.

Capítulo 2

Trabalhos Relacionados

Algumas das alternativas para minimizar o gargalo existente na comunicação entre memória e processador estão diretamente associadas às técnicas utilizadas para reduzir o tamanho dos programas. Diminuindo a quantidade e/ou o tamanho das instruções, conseqüentemente o número de buscas e a largura de banda da memória serão reduzidos. Além disso, com instruções menores e em menor quantidade, o processador poderá atingir um desempenho melhor.

Na Seção 2.1 são apresentados os principais e mais utilizados Esquemas de Codificação. Na Seção 2.2 são apresentadas três abordagens distintas para reduzir o tamanho dos programas: Codificação com Instruções Pequenas, Compressão de Código e Agrupamento de Instruções. Dentro de cada uma destas abordagens são explicadas diferentes técnicas e aplicações. Os trabalhos apresentados Seção 2.3 mostram como diminuir o tempo de execução através do Reuso de Instruções. Na Seção 2.4 são apresentadas algumas técnicas para atingir alto desempenho em *caches* e a arquitetura 2D-VLIW é apresentada na Seção 2.5.

2.1 Esquemas de Codificação

O objetivo de representar os programas com instruções cada vez menores e em menor quantidade pode ser vislumbrado durante a definição do esquema de codificação de uma nova arquitetura. O conjunto de instruções determina o nível de complexidade e o custo do hardware a ser implementado.

O desafio encontrado pelos projetistas ao definir um esquema de codificação é desenvolver instruções pequenas o suficiente para simplificar o estágio de execução, grandes o suficiente para minimizar a quantidade de instruções buscadas na memória e simples o suficiente para não impactar muito o estágio de decodificação. Nesta seção são apresentados algumas das estratégias de codificação de instruções mais conhecidas e utilizadas.

O esquema de codificação **CISC** (*Complex Instruction Set Computer*) [1] teve sua primeira aplicação na arquitetura System/360, apresentada pela IBM em 7 de Abril de 1964 como resultado de pesquisas iniciadas nos anos 50. Este computador foi o primeiro a utilizar micro-programação e, só mais tarde, ele seria conhecido como uma arquitetura CISC. O sucesso do System/360 resultou no domínio das arquiteturas CISC sobre o desenvolvimento de computadores e microprocessadores por duas décadas.

Em um processador CISC, cada instrução pode executar várias operações de baixo nível como leitura da memória, operação aritmética e armazenamento em memória, todas em uma única instrução. A vantagem da arquitetura CISC é ter muitas das operações armazenadas no próprio processador, o que facilita o trabalho dos programadores de linguagem de máquina e gera código executável menor.

Do ponto de vista de desempenho, a codificação CISC têm algumas desvantagens em relação ao RISC, entre elas a impossibilidade de se alterar alguma instrução composta para melhorar o desempenho. Um outro fator é que as arquiteturas CISC possuem instruções mais compactas já por construção, de modo que a compressão de código nestas arquiteturas não produz resultados tão expressivos quanto em arquiteturas RISC [18]. As arquiteturas System/360, VAX, PDP-11, família Motorola 68000 e Intel x86 são exemplos de utilização do esquema de codificação CISC.

O primeiro computador com instruções **RISC** (*Reduced Instruction Set Computer*) [28, 29] surgiu em 1979, como resultado de uma pesquisa iniciada em Outubro de 1975 pelo Centro de Pesquisa Watson da IBM. Esta pesquisa teve como motivação o fato de que em meados dos anos 70, muitas ferramentas de medição de desempenho demonstraram que a execução de boa parte dos programas em sistemas CISC eram dominadas por um conjunto pequeno de instruções simples e as instruções complexas eram pouco utilizadas.

Este modelo de codificação é uma filosofia que recomenda um conjunto de instruções reduzidas em termos de complexidade e de tamanho com o objetivo de permitir uma implementação mais simples, maior paralelismo no nível de instrução e compiladores mais eficientes. As características da arquitetura RISC como desempenho, facilidade de decodificação das instruções, simplicidade do código e melhor capacidade de geração de código tornam esta codificação mais adequada ao uso em sistemas dedicados.

Por possuírem um pequeno conjunto de instruções de tamanho fixo e, com isso, uma grande quantidade de bits não utilizados, os programas gerados para esta arquitetura são extremamente redundantes. Atualmente, as famílias mais comuns de processadores RISC incluem DEC Alpha, ARC, ARM, AVR, MIPS, PA-RISC, Power Architecture (incluindo PowerPC), SPARC, HP Precision, PMC-Sierra's 64-bit e ARCtangent-A4.

O conceito de arquiteturas **VLIW** (*Very Long Instruction Word*) [10, 20] foi apresentado por Josh Fisher e seu grupo de pesquisa da Universidade Yale, no inicio de 1980. A pesquisa de Fisher era em torno de um compilador que pudesse gerar micro-código horizontal a partir de programas escritos em uma determinada linguagem de programação. Os primeiros computadores VLIW reais foram lançados no começo dos anos 80 pela Multiflow Computer, Inc.

Neste esquema de codificação, as instruções são compostas por um número fixo N de operações independentes (paralelas). O total de bits utilizado por uma instrução VLIW é $N\times$ bits por operação. Quando existem menos do que N operações para compor uma instrução, são inseridas operações do tipo NOP (*No OPeration*) para completar a instrução. O código VLIW fornece um plano de execução (criado estaticamente em tempo de compilação) explícito ao processador. Este plano é montado utilizando conhecimento total do processador. A Figura 2.1 mostra um esquema hipotético de uma instrução VLIW composta por cinco operações.

O código gerado indica quando cada operação deve ser executada, quais unidades funcionais serão utilizadas e quais registradores serão acessados. O compilador comunica o plano de execução através de um conjunto de instruções que representam o paralelismo explicitamente ao hardware. Este plano permite utilizar um hardware relativamente simples

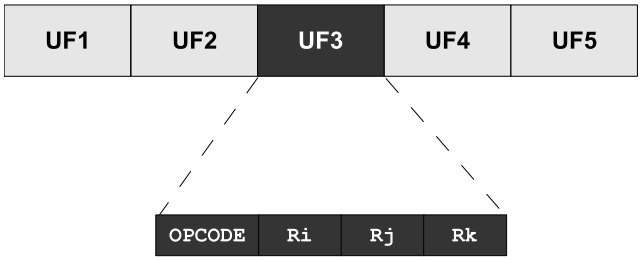


Figura 2.1: Instrução VLIW composta por 5 operações

que pode alcançar altos níveis de paralelismo em nível de instrução. Apenas duas empresas produziram comercialmente máquinas puramente VLIW: a Multiflow Computer, Inc. e a Cydrome, Inc. Pelo fato de não atingirem sucesso de mercado, há pouquíssimos dados sobre a eficiência real dessas máquinas, que ficaram restritas a poucas unidades entregues.

As instruções **EPIC** (*Explicitly Parallel Instruction Computing*) [44, 45] surgiram nos anos 90 com o objetivo de aumentar a habilidade dos microprocessadores em executar instruções em paralelo, utilizando o compilador, ao invés de hardware complexo, para identificar as oportunidades de execução paralela. Este esquema de codificação pode ser considerado como uma evolução do modelo VLIW.

Assim como as instruções VLIW, uma instrução EPIC é composta por um número fixo N de operações. Neste caso as operações podem ser dependentes ou independentes. O total de bits utilizado por uma instrução EPIC é ($N\times$ bits por operação) + bits que indicam dependência. A principal modificação em relação ao modelo VLIW foi criar um conjunto de bits (*Template Bits*) para indicar possíveis dependências entre as operações que compõe a instrução. A Figura 2.2 mostra um esquema hipotético de uma instrução EPIC composta por cinco operações.

O esquema de codificação **TMS** é uma outra abordagem que também utiliza instruções compostas por várias operações. Esta codificação é utilizada pela arquitetura TMS320C6x [14] que é uma nova geração do TMS32010 (o primeiro DSP da família TMS320), apresentado pela Texas Instruments em 1982. Atualmente, a família TMS320 consiste em várias gerações: C1x, C2x, C2xx, C5x, e C54x que são DSPs para operações

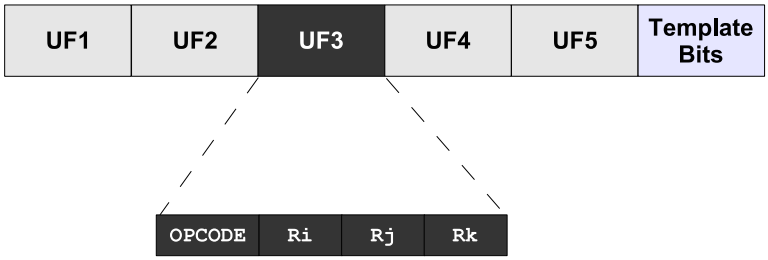


Figura 2.2: Instrução EPIC composta por 5 operações

de inteiros; C3x, C4x e C6x que são DSPs de ponto flutuante, e os multiprocessadores DSP C8x.

As instruções TMS são compostas por oito operações sendo que o grupo de execução de uma instrução é especificado por um bit denominado p-bit de cada operação. As instruções possuem 256 bits (8 palavras $\times$ 32 bits). Os p-bits controlam a execução paralela das operações e são lidos da esquerda para a direita (menor para maior endereço). Se o p-bit da instrução i for 0, então a instrução $i + 1$ será executada um ciclo após a instrução i.

Todas as operações que executam em paralelo constituem um pacote de execução. Um pacote de execução pode conter até 8 operações. Cada operação em um pacote de execução deve utilizar uma unidade funcional diferente. A Figura 2.3 mostra o formato básico de uma instrução TMS320C6x. Um pacote de execução não pode cruzar a fronteira das 8 operações. Por isso, o último p-bit em uma instrução sempre possui o valor 0 e, cada nova instrução buscada na memória, inicia um novo pacote de execução.

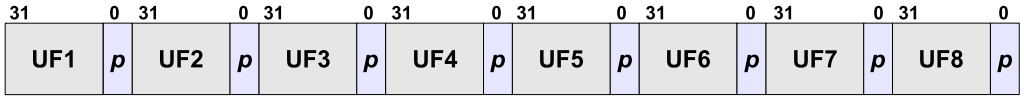


Figura 2.3: Formato básico de uma instrução TMS

Existem 3 tipos de padrões de p-bits para as instruções buscadas da memória. Estes três padrões resultam na seguinte seqüência de execução para as oito instruções: totalmente serial (todos os p-bits setados para 0), totalmente paralelo (todos os p-bits setados

para 1) ou parcialmente serial (alguns dos p-bits setados para 0 e outros para 1). Caso ocorra um *branch* para o meio de uma instrução, todas as operações anteriores à operação alvo do *branch* são ignoradas.

O modelo de codificação **IA64** [7] foi a primeira implementação EPIC disponível comercialmente. Trata-se de uma implementação da Intel que segue a filosofia EPIC, mas com algumas diferenças no que diz respeito ao modelo mais teórico EPIC, descrito anteriormente nesta seção.

As instruções IA64 possuem praticamente o mesmo algoritmo de escalonamento das instruções EPIC mas com apenas 120 bits que armazenam 3 operações (cada operação com 40 bits). Além destas 3 operações, a instrução possui um conjunto de 8 bits para evitar que operações dependentes sejam executadas ao mesmo tempo. Então, uma instrução IA64 possui 128 (120 + 8) bits. O compilador examina as dependências de todo o código e empacota as instruções em pacotes de 128 bits que podem ser executados em paralelo com segurança. A Figura 2.4 mostra o formato de uma instrução IA-64.

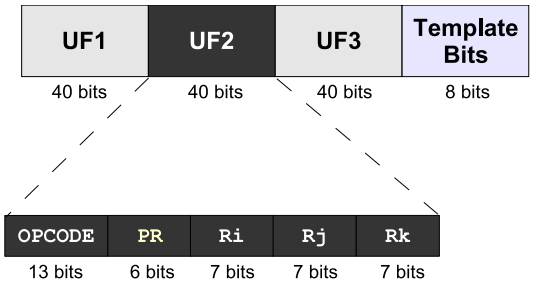


Figura 2.4: Formato de uma instrução IA64

Cada pacote de 128 bits possui 3 instruções e um conjunto de bits que especifica à CPU qual instrução depende da outra. Como estes bits especificam a dependência entre grupos de instruções, as unidades funcionais saberão como elas poderão processar estas instruções. Cada uma das 3 operações que compõe a instrução IA64 possui 40 bits divididos em 3 operandos de 7 bits, um registrador de predicado de 6 bits e *opcode* de 13 bits. As unidades funcionais assumem que o compilador tenha construído código seguro para ser executado em paralelo e que ele tenha utilizado os *Template Bits* (Figura 2.4)

para alertar a CPU de todas as dependências. Como a CPU não precisa examinar ou otimizar o código, tudo que é preciso fazer é colocar estes pacotes o quanto antes nas unidades funcionais.

Os esquemas de codificação RISC e CISC são os mais antigos e formam a base para todos os outros modelos utilizados atualmente. VLIW e EPIC são nomes mais conceituais do que práticos, são filosofias de codificação, modelos de execução de arquiteturas e podem ser considerados duas formas diferentes de se empacotar um conjunto de instruções RISC ou CISC. As codificações TMS320C6x e IA64 são exemplos de implementações EPIC criadas para fins diferentes e por empresas distintas. Não se pode afirmar diretamente qual técnica é a melhor. Durante o desenvolvimento de uma nova arquitetura deve-se avaliar as restrições do projeto como, por exemplo, quantidade máxima de memória disponível, consumo de energia desejado, desempenho mínimo exigido e área máxima do circuito para que se possa escolher o esquema de codificação mais adequado.

Mesmo com a melhor codificação para a arquitetura ainda pode ser possível melhorar o desempenho do processador através da utilização de alguma técnica complementar de redução de código. Na próxima seção são apresentadas algumas abordagens que podem ser utilizadas com o objetivo de reduzir o tamanho do código e conseqüentemente diminuir a quantidade de acessos à memória.

2.2 Técnicas para Redução de Código

O tamanho do código executável de um programa pode determinar a viabilidade de sua implementação em sistemas embarcados e também o desempenho em qualquer outro tipo de sistema. Quanto menor o código, menor o efeito do gargalo existente na comunicação entre memória e processador, especialmente em arquiteturas como VLIW e EPIC (Seção 2.1), que buscam instruções longas em memória. Mesmo utilizando codificações eficientes, estas podem demandar muitos acessos à memória prejudicando diretamente o desempenho do programa executado.

Nesta seção são apresentadas três abordagens diferentes que compartilham o mesmo objetivo: reduzir o tamanho do código de um programa. A primeira delas adota a uti-

lização de instruções pequenas, representando cada instrução com menos bits e, depois, durante a decodificação realiza o mapeamento destas instruções para as instruções maiores que serão interpretadas pelo processador. A segunda utiliza técnicas de compressão, que comprimem o código em memória e depois reconstroem o código original no interior do processador. A última técnica apresentada mostra como agrupar instruções. Nesta técnica as instruções dependentes são empacotadas em macro-instruções que são executadas por ULAs adaptadas.

2.2.1 Instruções Pequenas

Uma das primeiras abordagens utilizadas para diminuir o tamanho do código de um programa foi substituir o uso de instruções convencionais de 32 bits por instruções menores (16 bits). Esta idéia foi adotada no desenvolvimento de processadores como Thumb [22] e MIPS16 [17], definidos como subconjuntos das arquiteturas ARM [21] e MIPS respectivamente. As composições destes subconjuntos foram determinadas após análise de um grande conjunto de aplicações, onde foram escolhidas as instruções utilizadas com grande freqüência, que não utilizam 32 bits ou que são importantes para o compilador durante a geração do código intermediário.

O conjunto de instruções **Thumb** foi projetado a partir do conjunto padrão de instruções ARM de 32 bits que foi recodificado em instruções de 16 bits. Isto traz uma alta densidade de código pois as instruções Thumb possuem metade dos bits das instruções ARM. Antes da execução, estas instruções de 16 bits são mapeadas para instruções de 32 bits, pertencentes ao conjunto de instruções ARM. Desta forma, um programa Thumb roda normalmente em um processador ARM convencional.

Em um programa escrito para Thumb apenas os *branches* podem ser condicionais e muitos dos *opcodes* não possuem acesso a todos os registradores da CPU. O primeiro processador com um decodificador de instruções Thumb foi o ARM7TDMI. Este processador é capaz de executar tanto instruções de 16 bits quanto instruções de 32 bits e a distinção é feita através de um bit de estado do processador que pode ser alterado com a instrução `BX`. Depois deste processador, todos os ARM9 e famílias de processadores mais novas, incluindo o XScale, utilizaram um decodificador de instruções Thumb. O fator

de redução[1] dos códigos convertidos para Thumb pode chegar a até 30%, ou seja, um programa Thumb pode ser até 30% menor do que o mesmo programa ARM convencional. A Figura 2.5 mostra uma visão geral dos *pipelines* ARM, Thumb e Thumb-2.

Thumb-2 [30] é um superconjunto do conjunto de instruções Thumb. O conjunto de instruções Thumb-2 é composto por todas as instruções Thumb de 16 bits além de um grande conjunto de instruções de 32 bits para conseguir quase a funcionalidade total do conjunto original de instruções ARM. Isto significa que instruções de 16 e 32 bits podem ser utilizadas em conjunto permitindo ao compilador selecionar o conjunto ótimo de instruções. O resultado é que o código Thumb-2 pode conseguir o alto desempenho do código ARM e ao mesmo tempo o benefício da densidade do código Thumb. Os resultados apresentados em [35] mostram que um programa Thumb-2 pode ser até 25% mais rápido do que Thumb e 26% menor do que ARM.

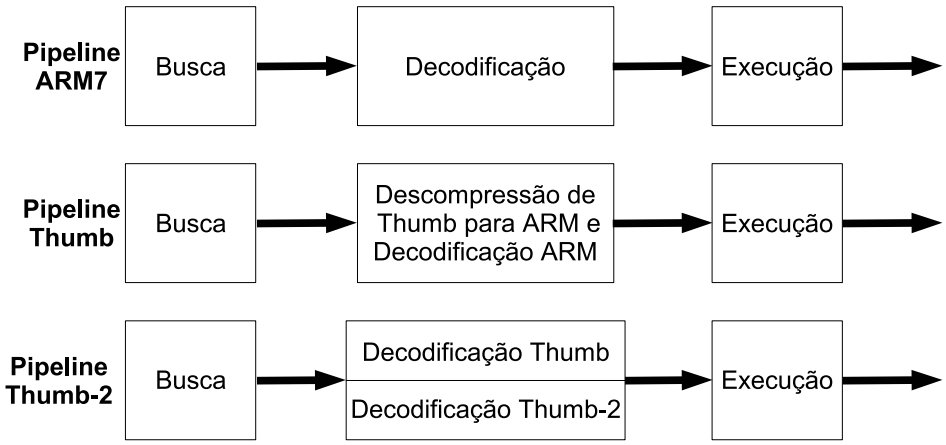


Figura 2.5: *Pipelines* dos processadores ARM, Thumb e Thumb-2

O processador **MIPS16** foi apresentado como uma solução para os problemas de densidade de código e largura de banda do MIPS RISC. Ele é classificado como uma extensão da arquitetura MIPS. O suporte para MIPS16 não é obrigatório para todas as implementações posteriores de MIPS, mas é o mecanismo padrão para compressão de

[1]Fator de Redução é a medida (%) que indica quanto foi a redução do tamanho de código de um programa quando codificado em outro esquema de codificação.

código dentre todos os fornecedores de CPUs MIPS RISC.

Como o nome sugere, MIPS16 é uma codificação de instrução de 16 bits. Ele foi projetado para ser 100% compatível com a arquitetura e programação dos outros processadores MIPS de 32 bits (MIPS-I/II) e de 64 bits (MIPS-III). De forma equivalente ao processador Thumb, MIPS16 possui a instrução `JALX` para trocar o conjunto de instruções utilizado (16 ou 32 bits). O processador MIPS16 oferece um fator de redução de até 40% sobre código MIPS convencional. A Figura 2.6 mostra uma visão geral do MIPS16.

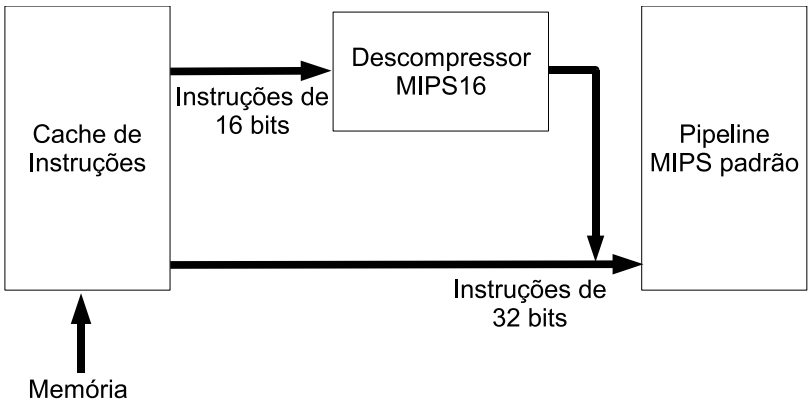


Figura 2.6: Visão geral do processador MIPS16

Reduzir o número de bits para representar uma instrução realmente pode trazer grandes benefícios relacionados ao tamanho final de um programa, porém o custo desta abordagem é a grande limitação de recursos de programação como menor número de registradores representáveis, imediatos com menos bits e, conseqüentemente, mais instruções para escrever o mesmo programa. Esta última conseqüência pode afetar diretamente o desempenho do processador pois o número de instruções que serão buscadas na memória, decodificadas e executadas será bem maior.

Para evitar a limitação de recursos utilizados pelo programador, pode-se utilizar uma outra abordagem de redução de código como, por exemplo, uma técnica de compressão. Neste caso, as instruções em memória são representadas em um formato comprimido, idealmente criado pelo compilador e, durante o estágio de decodificação, a instrução volta

ao seu formato original para que possa ser executada. A próxima seção descreve algumas das diversas técnicas de compressão de código conhecidas na literatura.

2.2.2 Compressão de Código

A compressão de código é uma técnica diferente do que foi utilizado em MIPS16 e Thumb pois os algoritmos de compressão vão além de realizar um simples mapeamento entre instruções menores e maiores. Após o estágio de busca, o mecanismo descompressor reconstrói cada uma das instruções comprimidas para que possam ser utilizadas no estágio de execução. Além disso, ao utilizar uma técnica de compressão não há necessidade de um conjunto de instruções especiais. Para o programador, isto é totalmente transparente. Diante disso, vários trabalhos com foco em redução do tamanho de instruções utilizaram conceitos e técnicas da área de compressão de código, sendo uma parte destes estudos voltados para as arquiteturas VLIW [27, 31, 34, 48, 49, 50], que é o foco deste trabalho de mestrado.

O grande desafio de uma técnica de compressão é ter um mecanismo de compressão eficiente o bastante para reduzir ao máximo o tamanho do código em memória e ao mesmo tempo, um esquema de descompressão muito simples a ponto de influenciar o mínimo possível na área, no consumo de energia e no tempo de execução do processador. Entretanto, algumas das propostas tentam melhorar a taxa de compressão do programa mas degradam o desempenho do processador ao realizar a descompressão [49]. Dentre as diversas técnicas de compressão de código, boa parte utiliza uma estrutura de armazenamento conhecida como dicionário. Este hardware mantém informações que são utilizadas durante o processo de descompressão.

A **Compressão por Instrução** [26] é a técnica mais simples das que utilizam dicionário. Nesta abordagem, cada vez que se encontra uma instrução diferente, é criada uma entrada em um dicionário. Esta entrada no dicionário (índice) representa a instrução no código comprimido. O método de descompressão também é o mais simples, conforme pode ser visto na Figura 2.7. Basta utilizar o índice armazenado na instrução comprimida e buscar a instrução correspondente no dicionário. Experimentos realizados [34] mostram um fator de redução de até $18,5\%$ para esta técnica.

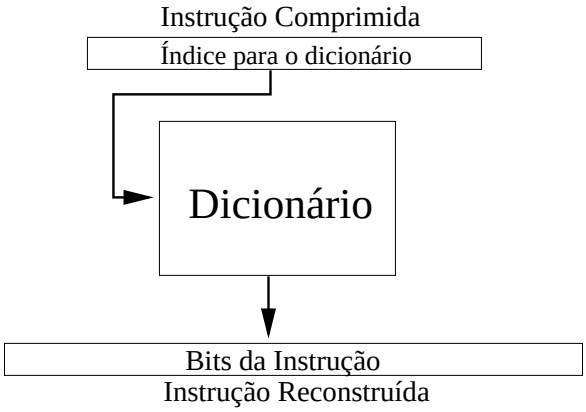


Figura 2.7: Visão geral da Descompressão por Instrução

A **Compressão por Fatoração de Instrução** [34] é semelhante ao método anterior mas, neste caso, existem dois dicionários: um para os *opcodes* e outro para os operandos. Nesta técnica uma instrução é dividida em dois campos: campo de bits de instrução, onde estão *opcode*, seqüências de escape, bits condicionais, bits específicos de hardware, etc, e campo de operandos, conforme pode ser visto na Figura 2.8. Como este método não é adequado para arquiteturas com instruções de formato variável, foi criada uma extensão dele através da utilização de mais um conjunto de bits para identificar a classe a qual pertence uma determinada instrução. Assim, a instrução comprimida possui quatro campos: índice para o dicionário de instruções, índice para o dicionário de operandos, bits de seleção de classe e outros bits, estes últimos podem ser bits especiais que aparecem em alguns tipos de instrução para indicar, por exemplo, que a instrução será executada em paralelo. Experimentos realizados mostram que esta técnica atinge um fator de redução de até $31,7\%$.

Na **Compressão por Fatoração de Operandos** [2] a estratégia é separar as árvores de expressões do programa em seqüências de *opcodes* e operandos (registradores e imediatos). Nos experimentos apresentados em [34], esta técnica é aplicada de forma um pouco diferente: as instruções são agrupadas em grupos de oito e os seus bits correspondentes são utilizados para formar uma entrada no dicionário. Estes experimentos mostram um fator de redução de até $15,3\%$ para esta técnica.

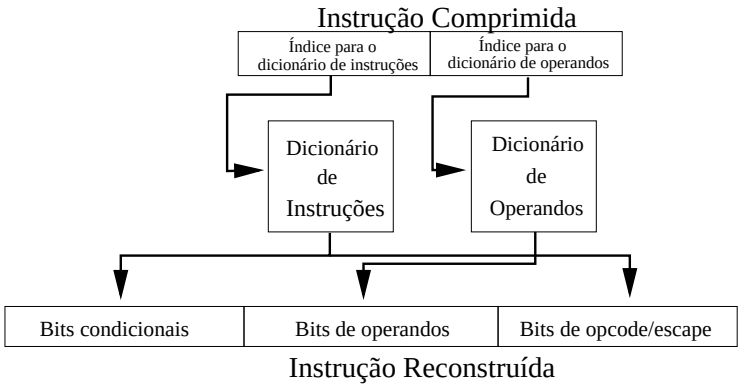


Figura 2.8: Visão geral da Descompressão por Fatoração de Instrução

A **Compressão por Isomorfismo de Instrução** [27] utiliza instruções isomorfas que são instruções com o mesmo *opcode* mas com pequenas diferenças no conjunto de operandos ou o mesmo conjunto de operandos com diferentes *opcodes*. Sua abordagem consiste em selecionar as instruções mais freqüentes e separar seus operandos e opcodes em dois dicionários. As instruções isomorfas entre si apontam para o mesmo índice nos dicionários. Neste trabalho os autores conseguiram um fator de redução de até 37%.

O trabalho apresentado em [3] mostra que é possível obter, ao mesmo tempo, ganho de desempenho, redução de código e redução no consumo de energia através de uma compressão de código eficiente e um descompressor bastante simples. Este método de compressão é baseado em dicionários e, para definir as instruções que serão comprimidas, propõe uma mistura de duas abordagens: avaliação estática (instruções que mais aparecem no código) e avaliação dinâmica (instruções mais executadas). Neste esquema de compressão o percentual de instruções selecionadas estaticamente ou dinamicamente é parametrizado através de um fator f. Isto permite avaliar qual a melhor proporção de cada abordagem. Se $f = 30\%$ significa que pelo menos 30% das instruções foram selecionadas pelo critério dinâmico. Os resultados mostraram que quando $f = 70\%$ consegue-se obter o melhor desempenho, redução de código e redução de energia. O descompressor utilizado é do tipo PDC (*Processor-Decompressor-Cache*), o qual posiciona o descompressor entre a *cache* e o processador. A técnica foi validada através de *benchmarks* executados em

uma implementação real do circuito descompressor sobre um processador Leon (SPARC V8). O fator de redução variou de 12% a 28%, o ganho de desempenho foi de até 45% e a redução no consumo de energia atingiu 35%.

As técnicas de compressão são uma alternativa interessante quando o desafio é minimizar a diferença de velocidade entre processador e memória pois muitas delas oferecem fatores de redução de código bastante significativos, permitindo reduzir drasticamente o número de acessos à memória. Porém, de uma forma geral, ao utilizar técnicas de compressão ocorre um aumento natural do tempo de compilação (compressão das instruções) proporcional à complexidade do algoritmo de compressão. Além disso, por mais simples que seja o circuito descompressor, ele influencia a área ocupada pelo processador e, dependendo da eficiência da técnica, pode aumentar o consumo de energia e até o tempo de ciclo do processador.

Outra linha de pesquisa que visa melhorar o desempenho dos processadores e pode ser utilizada como alternativa à compressão de código é o agrupamento de instruções. Nesta técnica as instruções dependentes são agrupadas de forma a constituir uma única macro-instrução que será executada dentro de uma ULA adaptada. Esta abordagem será explicada na próxima seção.

2.2.3 Agrupamento de Instruções

Com uma estratégia bem diferente das técnicas de compressão de código mas com o objetivo comum de melhorar o desempenho dos processadores, a técnica de Agrupamento de Instruções [15, 40, 41, 42] tem como foco principal reduzir o número de instruções a serem executadas. Nesta técnica são analisadas as cadeias de dependência das instruções e cada grupo de instruções dependentes é transformado em uma única instrução.

A proposta apresentada em [40] visa detectar cadeias de instruções conectadas por operandos temporários (`R1` na Figura 2.9), ou seja, operandos que fazem parte de operações, mas que são descartados pois o resultado final será armazenado em um único consumidor (`R9` na Figura 2.9). Estas instruções são agrupadas em uma única instrução e as instruções originais são removidas da linha de execução. Para isso é necessário um circuito simples e

também uma *cache*, ambos fora do caminho crítico do *pipeline*. Neste trabalho, define-se como *strand* um conjunto de instruções ligadas por operandos temporários (Figura 2.9). Cada instrução composta pode ser formada por no máximo três instruções.

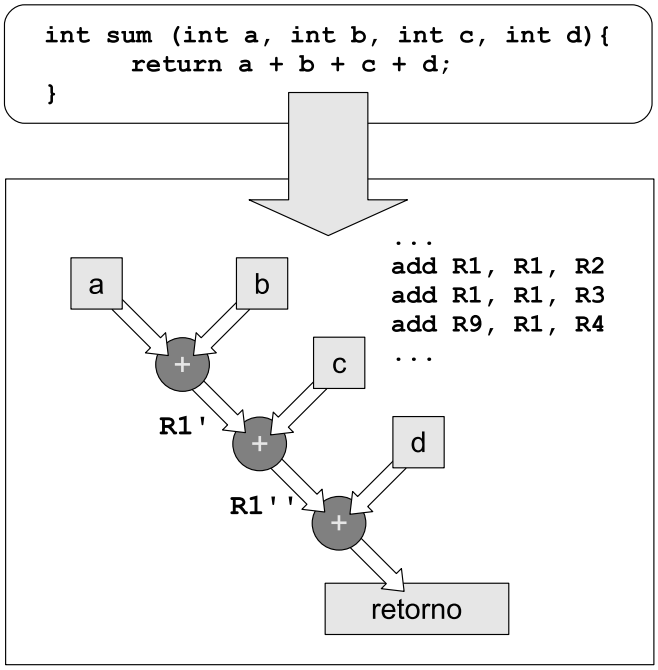


Figura 2.9: *Strand* composto por três instruções

Um dado interessante é que os operandos temporários têm vida curta, ou seja, em média, existem menos de quatro instruções separando o produtor do consumidor. As ULAs utilizadas nesta técnica possuem a capacidade de alimentar suas entradas com o último resultado processado. Segundo [40], as ULAs convencionais que gastam um ciclo, gastam na verdade meio ciclo para realizar a operação e meio ciclo para repassar o resultado. Como nesta técnica os resultados produzidos são reutilizados pela própria ULA, é possível realizar duas operações por ciclo. Assim, a latência total para se obter o resultado final de um *strand* é de $((0{,}5 \times H) + 0{,}5)$, onde H é a quantidade de instruções agrupadas no *strand*. Ao final do processamento de todo o *strand* gasta-se meio ciclo para repassar o valor. Os resultados mostraram que mais de 25% das instruções que são processadas em ULAs convencionais podem ser agrupadas. Em média, houve até 20% de

melhora no número de instruções executadas por ciclo (IPC).

Outro trabalho [41] nesta mesma linha aplicou a técnica de forma híbrida (estático + dinâmico), ou seja, a detecção de *strands* foi realizada em tempo de compilação (estaticamente) e a otimização dos mesmos em tempo de execução (dinamicamente). A principal motivação para esta proposta foi o fato de que para fazer todo o processo dinamicamente é necessário um hardware muito complexo e que consome energia suficiente para tornar sua aplicação impraticável em sistemas embutidos. Os experimentos realizados neste trabalho mostram que é possível reduzir o consumo de energia dos circuitos em até 24%, dos barramentos em até 20% e do banco de registradores em até 14%. Além disso, houve um aumento na capacidade efetiva dos recursos de *pipeline* em quase um terço e, em média, houve um aumento 15% no IPC.

A técnica que propõe o agrupamento de instruções dependentes realmente é bastante interessante no sentido de diminuir os acessos ao banco de registradores e a quantidade de instruções percorrendo o *datapath*. Isso pode levar a um ganho de desempenho e economia de energia. Porém, como descrito em [41], quando esta técnica é aplicada apenas em hardware, os circuitos se tornam muito complexos e o consumo de energia é relativamente alto. Além disso, é necessário pelo menos mais um estágio no *pipeline* para detecção de cadeias dependentes e também uma nova *cache* para armazenar os *strands*. Outra característica que pode ser alterada é o tempo de ciclo do processador.

A proposta da técnica de agrupamento de instruções é permitir que duas instruções possam ser executadas na mesma ULA em um único ciclo mas, em contrapartida, o tamanho do ciclo pode acabar sendo maior devido a troca de instruções RISC por instruções mais complexas (*strands*).

A forma como foi empregada a técnica em [41], a torna bem mais viável para qualquer tipo de processador, visto que além de reduzir o consumo de energia, produz um aumento significativo no IPC. Um ponto que não fica muito claro neste trabalho é com relação ao tamanho final do programa estático após o agrupamento das instruções dependentes. Ao agrupar instruções é necessário armazenar a identificação do início e o tamanho dos *strands*. Sendo assim, pode-se entender que esta representação, apesar de diminuir o

número de instruções, pode gerar um programa maior do que o original em memória, o que pode exigir uma largura de banda maior.

2.2.4 Discussão sobre Redução de Código

Nas seções anteriores foram apresentadas três abordagens bem diferentes com o objetivo comum de reduzir o tamanho do código de um programa. Enquanto que em alguns trabalhos a estratégia é utilizar um conjunto especial de instruções menores, outros mostram que é possível comprimir as instruções sem precisar definir um novo conjunto de instruções. Em outra abordagem, os autores mostram que as instruções podem ser agrupadas através da análise de suas dependências.

As técnicas de compressão de código foram as primeiras a surgir, dentre as três abordagens. O número de trabalhos encontrados na literatura é bem maior para as técnicas de compressão e os estudos mais recentes estão na área de agrupamento de instruções.

As três abordagens possuem vantagens e portanto, sem uma análise muito aprofundada é praticamente impossível afirmar qual é a melhor. Nos três casos o circuito necessário para a implementação da técnica gera *overhead* significativo no processamento mas em compensação, o desempenho alcançado é bem interessante. Dentre todas as abordagens apresentadas, talvez a mais promissora seja o agrupamento de instruções que utiliza a detecção de *strands* estaticamente conforme apresentado por [41].

Outra abordagem que pode ser utilizada para melhorar o desempenho do processador é evitar a execução de instruções desnecessárias cujo resultados já foram calculados em algum ponto do programa. Na próxima seção são apresentadas algumas técnicas de reuso que mostram resultados interessantes quando se consegue aproveitar valores previamente calculados ao invés de executar novamente um determinado conjunto de instruções.

2.3 Reuso de Instruções

Estudos mostram que boa parte dos resultados das instruções de um programa já foram computados previamente no mesmo programa [25, 33, 46]. As técnicas utilizadas para tirar proveito desta característica intrínseca aos programas têm como principal objetivo

evitar que a mesma instrução seja executada novamente sem que haja de fato necessidade. Ao evitar estas re-execuções é possível obter um ganho de desempenho considerável. Nesta seção são apresentadas técnicas que mostram como reutilizar instruções.

Na técnica chamada de *Instruction Reuse* (IR) [46], os resultados das instruções são armazenados para que possam ser reutilizados pelas mesmas instruções em outros pontos do programa. A estrutura de armazenamento dos resultados é chamada de *Reuse Buffer* (RB). Existem três esquemas para reutilizar instruções. Independente de qual for utilizado, deve-se atentar para três fatores: quais as informações armazenadas no RB, a forma como é realizado o teste para descobrir se será possível reutilizar a instrução e atualização/invalidação da informação no RB.

O primeiro esquema, chamado de S_v, é o mais simples e sua entrada no RB é representada por: valores dos operandos, resultado da operação, endereço calculado (para operações de leitura ou escrita na memória), *flag* que indica a validade do endereço de memória calculado, além de uma *tag* que armazena parcialmente o PC (*Program Counter*) e funciona como um índice para localizar a instrução (Figura 2.10(a)). Ao decodificar uma instrução, é verificada se existe uma entrada no RB correspondente e, caso exista, o resultado da operação é reutilizado. No segundo esquema, denominado S_n, a única diferença é que são armazenados os nomes dos registradores, ao invés dos valores (Figura 2.10(b)). Sendo assim, qualquer escrita em um registrador faz com que todas as linhas no *RB* que estejam utilizando este registrador sejam invalidadas. O campo *Resultado Válido* indica se o valor ainda pode ser reutilizado.

O terceiro esquema, conhecido por S_{n+d}, tenta minimizar as linhas que são invalidadas pelo esquema S_n, aumentado assim o número de instruções reutilizadas. O princípio utilizado é de que, se uma instrução pode ser reutilizada, então suas instruções dependentes também podem. Para implementar esta idéia é necessário a utilização de mais uma tabela (além do RB), a *Register Source Table* (RST) cujo papel é fazer uma ligação entre uma instrução consumidora e a última instrução produtora de valores de um determinado registrador. A entrada do RB para esta estratégia é bem parecida com a entrada do RB na estratégia S_n, porém, além dos nomes dos registradores, também são armazenados os índices (referentes ao RB) das instruções que geram valores para o registrador

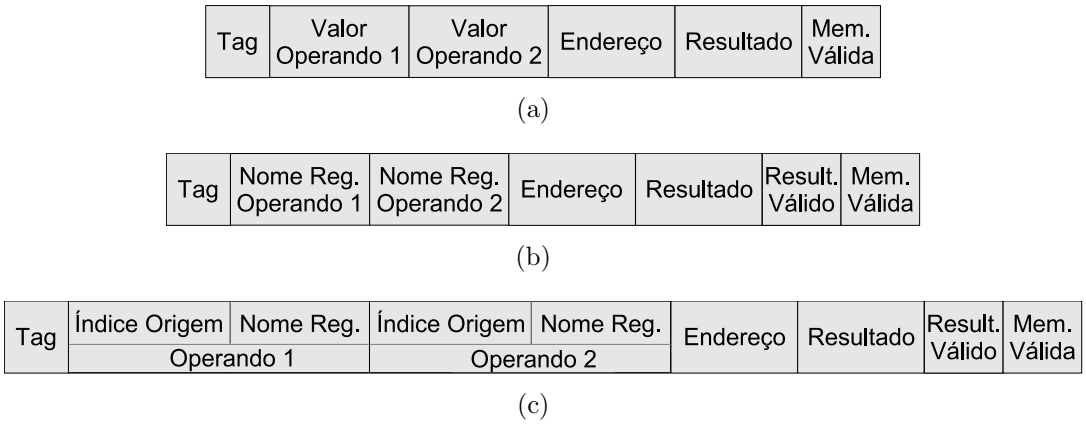


Figura 2.10: Entradas no RB para os esquemas (a) S_v, (b), S_n e (c) S_{n+d}

(Figura 2.10(c)). Uma instrução que possua dependência pode ser reutilizada se as instruções das quais a primeira depende forem as últimas produtoras para seus operandos (esta verificação é realizada através da RST).

Os experimentos realizados com programas $SPEC'92$ e $SPEC'95$ mostraram que utilizando o esquema S_v é possível conseguir um reuso de até 76%, enquanto que utilizando o S_n obtém-se reuso de até 25% e no S_{n+d} de até 59%. Este resultado pode ser explicado devido à quantidade de invalidações ocorridas nos esquemas S_n e S_{n+d}. As linhas do RB são raramente invalidadas quando se utiliza o esquema S_v, pouco invalidadas no esquema S_{n+d} mas são invalidadas com grande freqüência no esquema S_n pois qualquer escrita em um registrador invalida todas as linhas que referenciam este registrador no RB. Conseqüentemente, o *speedup* foi maior no esquema S_v, chegando a até 43%, enquanto que no esquema S_n foi de no máximo 17% e no esquema S_{n+d} foi de até 30%

Além da técnica *Instruction Reuse*, existem outras técnicas de reuso como *Instruction Memoization* (IM) [33] e *Instruction Level Reuse* (ILR) [25]. Em IM a procura por resultados já computados é realizada apenas na fase de execução e além disso, somente alguns tipos de instrução fazem parte do esquema de reuso, aquelas instruções que normalmente precisariam de mais de um ciclo para executar. No caso de IR, a procura por resultados ocorre no estágio de despacho ($issue$) (um estágio anterior à execução) e todas as ins-

truções são consideradas no esquema de reuso. A técnica ILR apresenta como principal vantagem sobre as outras duas o fato de considerar instruções que não são idênticas mas que geram os mesmos resultados, ou seja, instruções com o mesmo *opcode* que produzem o mesmo resultado podem ser reutilizadas. Desta forma, o reuso acaba sendo bem maior do que nas técnicas anteriores.

Um trabalho mais recente [6] propõe um novo estudo e revalidação destas três técnicas de reuso de instruções no contexto dos atuais microprocessadores para comprovar se realmente vale mais a pena recuperar o resultado computado por uma instrução em uma tabela ou simplesmente re-executar a instrução. Ao reproduzir os experimentos realizados pelos autores das três técnicas os resultados mostraram que no quesito *reuso* a técnica IM levou uma pequena vantagem na maior parte dos programas mas o *speedup* foi melhor quando se utilizou a técnica ILR.

Conforme pode ser visto em [6], o custo de se procurar por um resultado já computado pode ser maior do que simplesmente deixar que a instrução seja re-executada. É importante observar que boa parte das buscas nos *buffers* de reuso serão apenas um *overhead* para a fase de execução caso a instrução não esteja lá. De qualquer forma, é possível constatar com os trabalhos apresentados que existe um percentual considerável de instruções que se repetem durante a execução dos programas e que esta é mais uma característica que pode ser explorada para conseguir um melhor desempenho para os processadores.

Ao utilizar técnicas de reuso para melhorar o IPC do processador deve-se ter em mente que a latência da memória ainda pode continuar sendo o principal gargalo e, então, o desempenho alcançado pelo processador pode não ser tão evidente ao avaliar o sistema como um todo. Para minimizar esta diferença de velocidade entre memória e processador é importante melhorar o tempo de resposta da memória *cache*. Existem vários estudos realizados neste sentido e algumas estratégias como estruturar os dados dentro da *cache*, definir a organização da *cache* e até mesmo organizar ou reordenar o próprio programa para minimizar a quantidades de *misses*. Na próxima seção serão abordadas algumas técnicas que podem ser utilizadas para alcançar um bom desempenho em *caches*.

2.4 Desempenho em Caches

Enquanto que as técnicas de reuso de instruções possuem foco total em reduzir o número de instruções executadas, os trabalhos relacionados a *caches* se preocupam em conseguir responder às requisições do processador no menor tempo possível. Melhorando apenas o desempenho do processador, o tempo real de execução dos programas pode não ter ganhos significativos se o gargalo continuar sendo a comunicação entre processador e memória.

Muitos estudos já foram realizados com foco em minimizar o tempo da comunicação entre processador e memória. Mais especificamente, entre o processador e a memória *cache*. Segundo [16], a taxa de *miss* de uma *cache* normalmente pode ser reduzida aumentando o tamanho da *cache*, do bloco e/ou a associatividade. Porém, o tempo de acesso dos *hits* pode ser prejudicado se aumentar muito o tamanho da *cache* ou a associatividade. Além disso, o aumento do bloco freqüentemente reduz a taxa de *miss* mas aumenta o custo do *miss*. Para [4], quando a taxa de *miss* está acima de 30%, o desempenho é considerado baixo. Similarmente, a menos que o custo do *miss* seja alto (centenas de ciclos), uma taxa de *miss* menor que $0,3\%$ terá pouco impacto na execução do programa.

A técnica apresentada em [16] visa melhorar o desempenho de *caches* (minimizar a taxa de *miss* e o custo do *miss*) através de otimizações em tempo de compilação. A estratégia é excluir da *cache* os itens com baixa freqüência de uso. O esquema proposto pode ser utilizado em *caches* de instrução ou de dados. A essência do esquema é permitir apenas que as partes do código que são altamente utilizadas passem pelas *caches* e que as partes raramente utilizadas entrem direto nos *buffers* de instrução ou registradores do processador.

Para *caches* de instrução, a análise é realizada nos programas com o objetivo de identificar os blocos básicos mais utilizados. De acordo com a freqüência de utilização dos blocos básicos, eles são classificados em *High Usage* (HU), *Medium Usage* (MU) ou *Low Usage* (LU). A mesma análise e classificação é realizada para *caches* de dados, porém, ao invés de analisar blocos básicos, são analisadas as estruturas de dados utilizadas pelo programa.

O estudo mostra que, na prática, é difícil encontrar demarcações claras entre HU, MU e LU. Os limites não precisam ser fixos, podem variar de acordo com o tamanho do programa e da *cache*. Se o conjunto de código/dado for muito pequeno comparado ao tamanho da *cache*, o programa inteiro pode ser marcado como HU. Se tiver apenas um nível de *cache*, só são necessárias as categorias HU e LU. Baseado na análise do compilador, as instruções HU são marcadas para utilizarem a *cache* L1, as instruções MU a *cache* L2 e as LU, a *cache* L3. Os experimentos realizados mostraram que para *cache* de dados, houve um *speedup* de 12% a 41% em *caches* de 4kb a 64kb. Para *cache* de instrução, *speedup* de 0% a 12% em *caches* de 1kb a 4kb.

Diferente da proposta anterior, o trabalho apresentado em [4] visa melhorar o desempenho de *caches* através da reorganização do código dos programas. O *layout* dos programas pode ser alterados em três granularidades: módulo, procedimento e blocos básicos. Enquanto que o rearranjo de módulos e procedimentos pode ser feito de forma arbitrária, para rearranjar os blocos básicos é necessário adicionar instruções de salto incondicional para preservar o fluxo original. Isto faz com que o código fique maior. Por isso, este último tipo de arranjo foi desconsiderado nos experimentos.

A expectativa dos autores com relação ao rearranjo dos programas é de que a variação de *miss* aumente quando ocorrer rearranjos em granularidades mais finas, devido à localidade intra-módulo e intra-procedimento. Depois de referenciar um procedimento, espera-se que outro procedimento do mesmo módulo seja referenciado logo em seguida. *Caches* com mapeamento direto devem mostrar maior variação de *miss*, sendo que esta variação vai diminuindo com o aumento da associatividade. Além disso, espera-se que, diminuindo o nível de otimização do compilador, diminua a variação de *miss*. Após a realização de vários experimentos com variação de *layout*, configuração de *cache* e otimização em tempo de compilação, foram apresentadas algumas conclusões bastante interessantes:

- Não existe variação típica das taxas de *miss* para os programas: a variação vai desde zero até uma grande variação.
- A variação de *miss* é pequena em *caches* de tamanho pequeno (alta taxa de *miss*): ao aumentar o tamanho das *caches*, a variação também aumenta ou permanece

baixa. Os dois tamanhos de linha experimentados (16 e 64 bytes) tiveram quase que as mesmas variações.

- *Caches* de mapeamento direto apresentam muito mais variações de *miss* do que as *caches* 4-associativas que não demonstram variação.

- Diminuir a otimização normalmente diminui a variação: foram vistos dois exemplos e um contra exemplo. Alterando o conjunto de entrada não muda a variação, com apenas uma exceção.

- Alterar o tamanho da *cache* não influencia na classificação dos *layouts*: os ruins continuam ruins, mas não existem *layouts* bons consistentes.

- O tamanho da linha não tem efeito sobre os *layouts*; com apenas uma exceção.

- Alterar o nível de otimização afeta quais *layouts* são ruins, ou seja, alguns se tornam bons.

- Alterar o conjunto de entrada normalmente não altera quem são os *layouts* ruins.

- Para média de taxa de *miss* entre $0,3\%$ e $7,0\%$ as taxas de *miss* variam de 0,6m a 1,8m, onde m é a média das taxas de *miss*. Esta variação pode afetar significativamente o desempenho do sistema.

O estudo realizado em [32] enfatiza que normalmente os trabalhos relacionados a desempenho de *cache* têm como principal foco as taxas de *miss* e freqüência de tráfego. Porém, estas métricas de desempenho permanecem inalteradas se o custo de *miss* for de 2 ou de 20 ciclos, e elas não refletem precisamente o desempenho do processador. A idéia passada pelo autor é que, na prática, os projetistas de computador lidam com desempenho, custo, área e consumo de energia, e não com a utilização de barramentos e taxa de *miss*. Sendo assim, as várias decisões de organização (tamanho do conjunto, número de conjuntos, tamanho do bloco, etc.) afetam diretamente ou indiretamente o tempo de ciclo da *cache*.

Caches maiores necessitam de RAMs maiores. Memórias maiores e mais largas são geralmente mais lentas e mais caras. A associatividade dos conjuntos requer um multiplexador adicional no *datapath*, além de afetar a complexidade e o controle. O dilema entre tamanho de *cache* e tempo de ciclo é geralmente colocado em termos de maximizar o tamanho da *cache* sem aumentar o tempo de ciclo da CPU. Até certo tamanho, as *caches* grandes são boas mas chega em um ponto que as melhores são as pequenas.

Foram realizados experimentos variando o tempo de ciclo entre $20ns$ e $80ns$, e variando o tamanho da *cache* entre 2KB e 8MB. Com o aumento de tamanho da *cache*, o peso do componente memória no total de ciclos diminui. O tempo total de execução é o produto do tempo de ciclo pelo total de ciclos. Com *caches* pequenas, o aumento no tamanho das *caches* tem um efeito maior do que mudanças no tempo de ciclo, enquanto que em *caches* maiores, o inverso é verdade. Isto indica fortemente que existe uma região mediana onde ambos se encontram balanceados apropriadamente. Menos de $10ns$ de melhora no tempo de ciclo melhora o desempenho geral, porém, mais de $10ns$ de piora no tempo de ciclo, coloca a máquina em uma curva de desempenho pobre. As conclusões obtidas com os experimentos foram:

- Existe uma tendência em utilizar *caches* entre 32KB e 128KB pois dentro desta região o ganho de desempenho torna-se cada vez mais achatado.

- A motivação por se utilizar associatividade em *cache* primeiramente é porque existe uma redução na taxa de *miss* sobre uma *cache* de mesmo tamanho com mapeamento direto. Porém, aumentar o número de bits que devem ser lidos simultaneamente pode afetar dramaticamente a área e o custo da *cache*.

- Existe um aumento no tempo de acesso sobre uma *cache* com mapeamento direto pois o maior impacto temporal da associatividade é a necessidade de completar o acesso à *tag* e a comparação antes de enviar o dado para a CPU. Nos experimentos, a mudança de mapeamento direto para 2-associativa diminuiu em 20% a taxa de *miss* em *caches* maiores que 256KB. Melhoras menores podem ser vistas com associatividades maiores que 2.

- Deve-se tentar primeiro o tamanho da *cache* e, só depois, principalmente em *caches* integradas, é que se deve aumentar a associatividade. Além disso, reduzindo o tamanho ou associatividade para diminuir o tempo de ciclo pode melhorar o desempenho como um todo.

- A escolha de um tamanho de bloco afeta o custo de referência da memória através da taxa de *miss* e do custo do *miss*. Aumentando o tamanho do bloco diminui a taxa de *miss* devido à localização espacial das referências. O tamanho de bloco que otimiza o desempenho do sistema é significantemente menor do que o que minimiza a taxa de *miss*.

- Os melhores tamanhos de bloco são de 4 e 8 palavras.

- A existência de um segundo nível de *cache* modifica o dilema entre tamanho e velocidade do primeiro nível de *cache*, reduzindo o custo dos *misses* do primeiro nível, fazendo que *caches* pequenas e rápidas tornem-se uma alternativa viável.

As técnicas para melhorar o desempenho das *caches* seguem várias linhas diferentes como, separar as instruções por freqüência de uso entre os níveis de *cache*, reorganizar as instruções do programa ou então avaliar parâmetros diferentes de quantidade de *misses* antes de se definir associatividade, quantidade de blocos ou outros parâmetros que possam influenciar diretamente o desempenho da *cache*. Estas três diferentes visões podem ser consideradas ao se definir uma hierarquia de memória. São abordagens complementares que juntas podem garantir um desempenho satisfatório para a arquitetura.

Na próxima seção será apresentada a arquitetura que foi utilizada como referência para validar a nova técnica de codificação descrita nesta dissertação. Trata-se de uma arquitetura de alto desempenho, capaz de executar várias operações por ciclo e minimizar o número de acessos ao banco de registradores global.

2.5 Arquitetura 2D-VLIW

Os trabalhos apresentados nas seções anteriores deste capítulo formam a base teórica para o desenvolvimento do novo esquema de codificação descrito nesta dissertação. PBIW é

uma técnica de codificação assim como VLIW ou EPIC, utiliza uma *cache* de padrões similar a um dicionário das técnicas de compressão de código, está baseada na existência de reuso de padrões, de forma análoga às técnicas de reuso de instruções e utiliza estratégias para melhorar o desempenho da *cache* de padrões, de acordo com algumas estratégias de desempenho de *cache*. Porém, para avaliar a eficiência da codificação é necessário implementá-la em alguma arquitetura e comparar os resultados com outras codificações já conhecidas. A arquitetura escolhida para esta validação foi desenvolvida durante um trabalho de doutorado no LSC/IC-Unicamp pelo estudante Ricardo Ribeiro dos Santos. Neste sentido, o objetivo desta seção é apresentar apenas uma visão geral desta arquitetura. Maiores detalhes podem ser vistos em [36].

A arquitetura 2D-VLIW [36, 37, 38, 39] é uma arquitetura de alto desempenho que utiliza uma codificação VLIW-Like composta por operações RISC e possui os estágios de: Busca, Decodificação e N (número de linhas da matriz de execução) estágios de Execução.

Esta arquitetura recupera grandes instruções da memória, onde o número de operações dentro de cada instrução é equivalente ao número de unidades funcionais (UFs) da arquitetura. As operações que compõe a instrução 2D-VLIW passam por uma matriz de unidades funcionais e podem ler e escrever valores no banco de registradores globais (GRF) e em registradores temporários (RT) que estão espalhados pela matriz de execução. A Figura 2.11 apresenta uma visão geral da arquitetura 2D-VLIW. Nesta figura é possível observar o *datapath* utilizado por uma instrução e a organização das UFs através de uma matriz N linhas $\times$ M colunas, onde, neste caso, $N = M = 4$. A hierarquia de memória e os registradores temporários não foram exibidos.

2.5.1 Instruções 2D-VLIW

As instruções 2D-VLIW são formadas por operações simples (`add, sub, ld, st`, entre outras) além da operação `phi` que pode ser utilizada para a leitura dos registradores de predicado. Todas estas operações possuem 32 bits e são executadas no interior da matriz de unidades funcionais. O número de operações de uma instrução é equivalente à quantidade de unidades funcionais na matriz. Assim, para a matriz 4×4 da Figura 2.11, uma instrução 2D-VLIW possui 16 operações. Caso não seja possível encontrar 16 operações

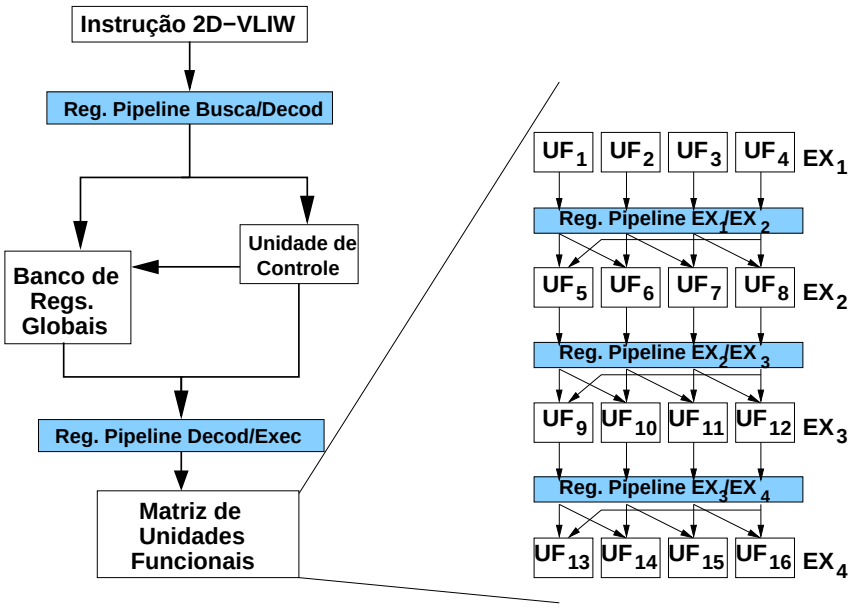


Figura 2.11: *Datapath* da arquitetura 2D-VLIW

no programa para preencher toda a matriz, operações especiais do tipo NOP (operações nulas que não alteram os estados dos registradores) são inseridas.

O tamanho das instruções 2D-VLIW é fixo e é calculado como $N \times M \times 32$, ou seja, 512 bits considerando a arquitetura da Figura 2.11. As instruções são compostas por operações dependentes e independentes, sendo que o compilador é responsável por resolver estas dependências. Cada instrução poderia ler até 32 registradores globais (2 registradores de leitura por operação $\times$ 16), entretanto, a estrutura arquitetural 2D-VLIW permite apenas a leitura de $(2 \times M)$ registradores globais por instrução.

2.5.2 Modelo de Execução

A cada ciclo de relógio, uma instrução 2D-VLIW é buscada na memória e colocada no registrador de *pipeline* Busca/Decod. No estágio de decodificação, a instrução é decodificada e os operandos que usam registradores globais são repassados ao banco de registradores globais. Depois disso, a instrução está pronta para ser repassada à matriz de unidades funcionais. Considerando a arquitetura da Figura 2.11, a cada ciclo de execução,

EX_1, EX_2, EX_3, EX_4, quatro operações de uma instrução são repassadas para as UFs. A instrução é levada de um estágio EX_i para um estágio EX_j, $j = i + 1, 1 \leq i < 4$, através dos registradores de *pipeline*.

Deve ser destacado que esse modelo permite que uma operação atribuída à uma UF no estágio EX_j use os resultados das operações que executaram no estágio EX_i. Obviamente, essa característica vale apenas para os casos onde o tempo de processamento da operação no estágio EX_i é igual ou menor que o tempo que a instrução leva para ser enviada do estágio EX_i para EX_j. Além disso, o sistema de conexão entre as UFs possibilita que uma UF do estágio EX_i forneça operandos para duas outras UFs no estágio EX_j. Isso é feito através dos registradores temporários que estão distribuídos ao longo da matriz de UFs e são os responsáveis principais pela comunicação direta entre UFs.

2.5.3 Estruturas de Armazenamento

Existem três classes de registradores dentro da arquitetura para dar suporte ao modelo de execução 2D-VLIW: os registradores globais, os registradores temporários e os registradores de predicado. A arquitetura prevê o uso de 32 registradores globais organizados em um banco de registradores, 2 registradores temporários por UF (32 registradores temporários considerando a Figura 2.11) e, por fim, 32 registradores de predicado. Para uso exclusivo de cada UF ainda existem os chamados registradores UF (RUF), os quais sempre armazenam o resultado da última operação executada por sua respectiva UF. Este registrador pode ser utilizado para leitura apenas pela UF a qual ele pertence.

Os registradores de predicado são mapeados para registradores globais (GRF) ou para os registradores temporários (RT) através da Tabela de Predicados (TP). Se o desvio for executado, o registrador de predicado recebe o valor '1' senão, recebe o valor '0'. Todo registrador que é destino de uma operação possui um registrador de predicado associado. No entanto, um registrador de predicado pode estar associado a diversos registradores de destino de diferentes operações.

A leitura dos registradores de predicado ocorre através de unidades funcionais específicas chamadas de unidades de *ϕ-function*. Estas unidades possuem dois registradores

(podem ser temporários ou do GRF) de entrada e sua semântica indica que a entrada selecionada é o primeiro registrador de entrada se o registrador de predicado (associado aos dois registradores de entrada) for '1', senão a entrada selecionada é o segundo registrador de entrada. A escrita nos registradores de predicado ocorre através de unidades de IF-CONV (*if-conversion*). Essas unidades têm como entrada um registrador de predicado e escrevem '0' ou '1' no registrador dependendo do resultado da comparação da instrução de desvio condicional.

2.5.4 Considerações sobre a Arquitetura 2D-VLIW

Nesta seção foram apresentadas as principais características da arquitetura 2D-VLIW, utilizada para validar o tema principal desta dissertação de mestrado. Esta arquitetura é extremamente flexível quanto ao número total de UFs e à quantidade linhas e colunas da matriz de execução. A implementação utilizada neste trabalho foi de 4×4 UFs, conforme apresentada na Figura 2.11.

2.6 Considerações Finais

Neste capítulo foram apresentadas várias estratégias para reduzir o gargalo da memória dentro de uma arquitetura. Primeiramente, foram apresentados alguns dos mais conhecidos modelos de codificação, desde os mais simples como RISC e CISC, até implementações mais recentes como o IA64 e o TMS320C6x, passando também pelos clássicos VLIW e EPIC.

O modelo RISC surgiu depois de observar-se que, na verdade, as instruções complexas quase nunca eram utilizadas nos programas CISC. VLIW era um modelo adequado para o aproveitamento de paralelismo em nível de instrução, porém, não fez muito sucesso. EPIC surgiu a partir de modificações no modelo VLIW, sendo a principal delas o acréscimo de bits que indicam se as instruções podem ou não ser executadas em paralelo e, finalmente, IA64 e TMS320C6x são modelos EPIC comerciais com alguns detalhes específicos.

Com exceção de RISC, que é ótima para o estágio de execução mas gera muita redundância em memória, todas as outras codificações são instruções longas que exigem

uma maior largura de banda no estágio de busca e um hardware um pouco mais complexo (do que RISC) durante a execução (exceção ao VLIW que não precisa de hardware complexo mas que também gera muitas instruções esparsas em memória, comprometendo o desempenho pela quantidade de instruções executadas).

Dentre as técnicas de redução de código a primeira e mais simples é a utilização de instruções pequenas, ilustrada nas arquiteturas MIPS16 e Thumb. A segunda técnica apresentada foi a compressão de código onde boa parte dos trabalhos mostra como reduzir o tamanho do código de um programa em memória utilizando uma estrutura parecida com uma *cache* de dados, chamada dicionário. Na última abordagem apresentada, a estratégia é agrupar instruções dependentes e assim reduzir ao máximo o número de instruções que serão executadas. Nesta abordagem as ULAs são capazes de utilizar com entrada o resultado computado no ciclo anterior.

Todas estas técnicas possuem em comum o custo de um hardware que pode influenciar na área do processador, no consumo de energia e mesmo no desempenho já que ele é utilizado para reconstruir as instruções (instruções pequenas e compressão) ou para detectar dependência de instruções (agrupamento).

Sobre as técnicas de reuso pode-se perceber que muitas instruções são re-executadas durante um programa e que é possível evitar isso armazenando os resultados computados pelas instruções em uma tabela. Antes de executar a instrução, caso seja identificado o valor já computado, este valor é passado diretamente para o último estágio do *pipeline* evitando pelo menos a fase de execução. A partir deste conceito de que existem instruções sendo re-executadas pode ser possível gerar alguma representação comum para todas estas instruções redundantes e tentar diminuir ao máximo a quantidade de informações que são armazenadas em memória.

Na seção sobre desempenho em *cache*, foram apresentadas três abordagens bem diferentes. Em uma delas é realizada a classificação dos dados por freqüência de utilização, mantendo os mais utilizados no nível mais alto da *cache* (L1), os utilizados de forma média, na *cache* L2 e os pouco utilizados na *cache* L3 ou mesmo fora da hierarquia de *caches*. No segundo trabalho, a técnica é tentar reorganizar os programas (módulos, procedimentos,

blocos básicos) de forma a conseguir melhores desempenhos e no último trabalho apresentado, foram realizados vários experimentos com o objetivo de definir alguns bons valores para tamanho de bloco, associatividade, etc.

No final do capítulo, a seção sobre a arquitetura 2D-VLIW apresentou uma visão geral e algumas das principais características como o formato das instruções, modelo de execução e estruturas de armazenamento. A característica mais interessante desta arquitetura é a capacidade de trafegar os dados dentro da matriz de execução sem a necessidade de ficar acessando o banco de registradores. Isso é feito através dos registradores temporários. Além disso, a existência dos RUFs (registrador de uso exclusivo de uma UF) permite que os dados computados na última operação sejam reaproveitados. Estes dois mecanismos possuem uma certa semelhança com a técnica de agrupamento de instruções no que diz respeito aos objetivos vislumbrados por ambos. Nos dois casos a principal estratégia utilizada é a execução de instruções dependentes sem a necessidade de acessar muitas vezes o banco de registradores aproveitando dados computados no ciclo anterior como entrada para a instrução corrente.

Diante disto, a lacuna a ser preenchida para tentar minimizar a diferença entre memória e processador consiste em desenvolver um novo mecanismo capaz de atingir altos níveis de redução de código como as melhores técnicas de redução, que seja simples o suficiente para não onerar o tempo de decodificação e execução do programa. Neste sentido, será apresentado no próximo capítulo o novo esquema de codificação desenvolvido durante este trabalho de mestrado, cuja proposta é representar os programas em memória com um tamanho bastante reduzido e, através de um hardware de decodificação simples, construir a instrução que será utilizada nos ciclos de execução do processador.

Capítulo 3

Esquema de Codificação PBIW

A arquitetura 2D-VLIW apresentada no capítulo anterior (Seção 2.5) possui uma codificação VLIW, porém permite escalonamento de forma que operações dependentes possam ser agrupadas na mesma instrução devido ao modelo de execução em matriz e a comunicação entre as UFs. Esta característica permite que um programa 2D-VLIW possa ser codificado com menos instruções do que um programa VLIW convencional, conforme dados dos experimentos apresentados no próximo capítulo.

Também apresentada no capítulo anterior (Seção 2.1) a codificação EPIC é simplesmente uma derivação do modelo VLIW tradicional, adicionado de um conjunto de bits que são utilizados durante a fase de execução para indicar dependências entre as operações. Portanto, da mesma forma que EPIC pode ser derivado de VLIW, também pode ser derivado de 2D-VLIW. A Figura 3.1 apresenta uma codificação EPIC derivada de 2D-VLIW, considerando uma matriz de execução composta por 4×4 UFs. Esta seria a forma mais direta de criar uma codificação EPIC derivada de 2D-VLIW. Cada 4 bits dos 64 bits de template são utilizados para indicar o número da UF na qual uma determinada operação será executada.

Se a posição (UF na matriz de execução) $\mathrm{Pos}(Op_i)$ de uma determinada operação Op_i for menor ou igual a posição $\mathrm{Pos}(Op_{i-1})$ da operação imediatamente anterior Op_{i-1} na instrução significa que durante a execução ocorrerão X *stalls* entre estas duas operações, onde X pode ser calculado de acordo com a Equação 3.1.

$$\textbf{Stalls} = (\text{Total de UFs - Pos}(Op_{i-1})) + (\text{Pos}(Op_i) - 1) \tag{3.1}$$

Considerando a Figura 3.1, caso a OP_3 esteja atribuída à UF_6 e a OP_4 esteja atribuída à UF_2, então o número de *stalls* entre estas duas operações é (16 - 6) + (2 - 1) = 11 (considerando a UF_1 como primeira UF da matriz e UF_{16} a última).

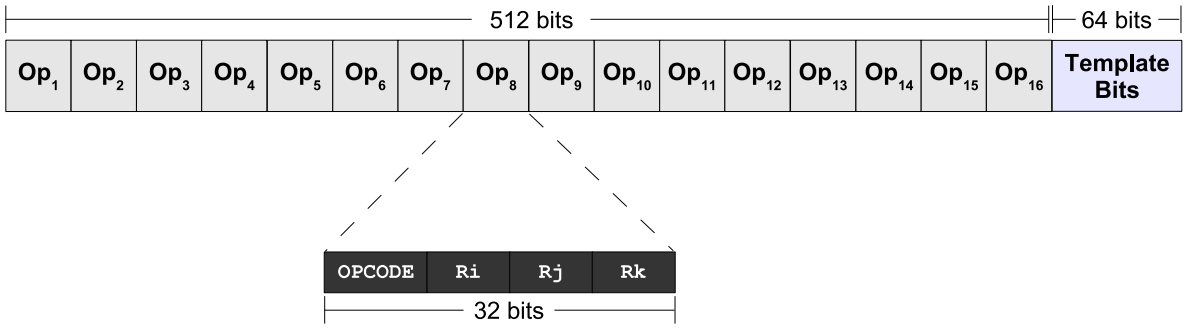


Figura 3.1: Instrução EPIC derivada de 2D-VLIW

Com 64 bits de template é trivial representar uma instrução 2D-VLIW como uma instrução EPIC. Este modelo, embora simples, já permite um grande ganho e, modelos melhores, diminuem a quantidade de bits extras. O número de bits extras pode variar entre 1 e 64. Alternativas com menos bits são apresentadas no próximo capítulo juntamente com resultados de experimentos utilizando estas codificações.

Este capítulo descreve detalhadamente todos os aspectos relacionados ao esquema de codificação PBIW. O texto aborda a estrutura da instrução, a organização da *cache* de padrões, o algoritmo de codificação e o mecanismo de decodificação. Ao final é apresentado um estudo de caso da codificação PBIW sobre a arquitetura 2D-VLIW.

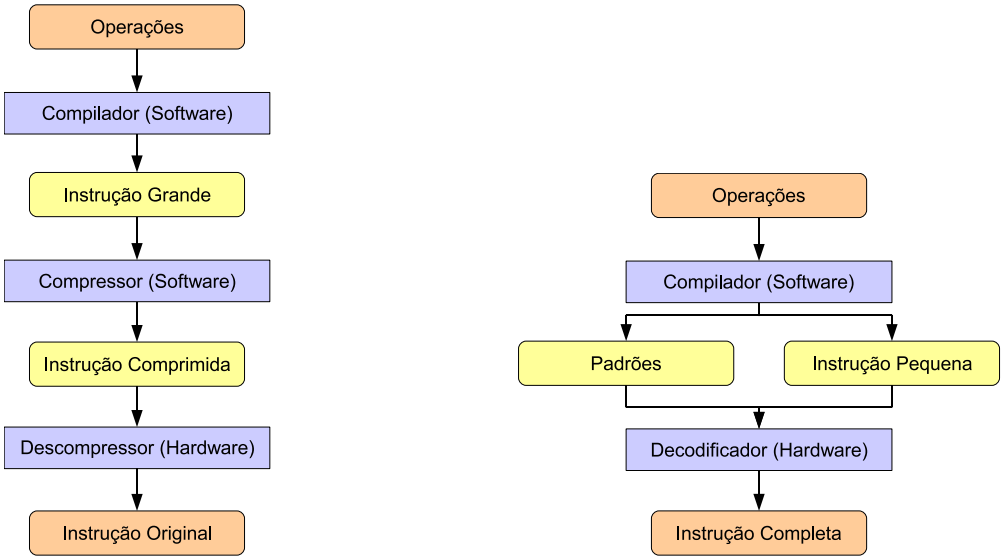
3.1 Visão Geral

De forma semelhante à codificação EPIC, outra alternativa que pode ser derivada da codificação 2D-VLIW é o esquema de codificação PBIW[1] (*Pattern Based Instruction Word*) que tem como principal objetivo representar instruções de forma bastante compacta em memória sem a necessidade de hardware complexo para se compor a instrução executável no interior do processador. Diferentemente do dicionário utilizado em algumas técnicas de compressão, na estratégia de codificação PBIW utiliza-se uma *cache* simples que armazena parte dos dados que compõe a instrução. Além disso, durante a busca de dados nesta *cache*, no estágio de decodificação da instrução, é possível realizar outras atividades como a leitura dos registradores de entrada.

Nas técnicas convencionais de compressão de código, uma instrução grande é comprimida e passa a ser representada em memória por uma instrução pequena. Antes do estágio de execução a instrução pequena volta a ser exatamente a mesma instrução grande inicial (Figura 3.2(a)). Na codificação PBIW a instrução inicial já é pequena e complementa dados armazenados em uma *cache* de padrões para constituir a instrução completa que é executada pelo processador (Figura 3.2(b)).

Assim como nas técnicas tradicionais de compressão de código baseadas em fatoração de operandos [2, 9, 11], o algoritmo utilizado na estratégia de codificação PBIW percorre as operações do programa extraindo padrões a partir de operandos redundantes. Essa estratégia gera uma considerável redução no tamanho da instrução uma vez que dados redundantes não estarão presentes em seu conteúdo. O fundamento dessa técnica é a sobrejeção intrínseca entre instruções e padrões. Pesquisas anteriores [6, 46] com foco na busca de padrões de instruções, descobriram que a função que mapeia o conjunto de instruções de um programa CI para o conjunto de padrões dessas instruções CP obedece ao comportamento de uma função sobrejetora. Em outras palavras, dado um conjunto de instruções codificadas $CI = \bigcup_{j=1}^{n} I_j$, um conjunto de padrões $CP = \bigcup_{i=1}^{m} P_i$, existe um mapeamento f tal que

[1] Considera-se como codificação por ser mais amplo mas pode ser considerado como esquema de compressão também.



(a) Compressão de Código (b) Codificação PBIW

Figura 3.2: Fluxo convencional de compressão (a) e Fluxo da codificação PBIW (b)

$$P_l = f(I_i) = f(I_j) = f(I_k), \quad \text{onde} \tag{3.2}$$
$$\{I_i,\ I_j,\ I_k\} \subset CI,\ i \neq j \neq k,$$
$$P_l \in CP,\ m < n$$

A função f na Equação 3.2 é sobrejetora pois mapeia instruções diferentes sobre um mesmo padrão P_l. Ao aplicar restrições arquiteturais durante a codificação de uma instrução PBIW, o número de instruções codificadas poderia se tornar maior do que o número de instruções que seriam criadas por outra técnica de codificação. Desta forma, a existência da sobrejeção entre o conjunto de instruções e o conjunto de padrões é uma condição necessária para que essa estratégia possa reduzir a quantidade de memória utilizada por um programa. Os resultados apresentados no próximo capítulo comprovam a existência da sobrejeção e, mais importante, mostram a redução no tamanho do programa codificado com instruções PBIW.

Esta é a maior motivação e o aspecto chave para a fatoração de padrões a partir das

instruções. Os padrões fatorados podem ser reutilizados por diferentes instruções. Um grande reuso de padrões torna possível reduzir o tamanho do código de um programa, pois a instrução codificada é menor, algumas vezes muito menor, do que seria uma instrução não codificada e a quantidade de padrões é menor do que o número de instruções codificadas.

Define-se como operação uma estrutura formada por um *opcode*, um operando de destino (registrador) e um conjunto de operandos de origem (registradores ou imediatos). Já uma instrução, é definida como um conjunto de operações. Uma instrução PBIW representa de forma compacta um conjunto de operações, sendo que os *opcodes* são armazenados no padrão e os operandos (registradores e imediatos) na instrução. Porém, um operando é armazenado uma única vez na instrução ou seja, não existem dados redundantes.

As instruções PBIW são armazenadas em uma I-*cache* como qualquer instrução de um programa. Um padrão (ou padrão de instrução) é uma estrutura de dados que contém os *opcodes* das operações e ponteiros para as posições da instrução codificada. Em outras palavras, os padrões armazenam valores que indicam um dos campos de dados da palavra codificada. Estes valores (ponteiros) são utilizados como mapeamento durante a construção da instrução completa dentro do *datapath*. Um padrão é armazenado em uma *cache* especial chamada P-*cache* (*Pattern Cache*), que é mostrada na Figura 3.22 da Seção 3.5.

Pode-se entender a instrução PBIW como uma instrução CISC composta por operações RISC (armazenadas no padrão). O mesmo padrão pode ser reutilizado por instruções CISC diferentes. O padrão também pode ser visto como um micro-código necessário para executar a instrução CISC. Este micro-código obtém seus parâmetros a partir dos campos da instrução CISC.

A próxima seção apresenta a estrutura da instrução PBIW com detalhes da disposição, tamanho e utilização de cada um de seus campos, além de descrever o relacionamento entre um padrão e suas respectivas instruções.

3.2 Estrutura da Instrução

Uma instrução PBIW é composta por quatro conjuntos de informações: número dos registradores cujos valores devem ser lidos, número dos registradores de destino (escrita) ou valores imediatos, índice para a tabela de padrões em memória e bits que indicam quais UFs da matriz de execução devem executar ou anular suas operações (a anulação de linhas/colunas é detalhada na Seção 3.4).

Em cada instrução, existem R campos para representar os registradores de leitura, onde R é igual ao número de portas de leitura do banco de registradores. Cada um destes campos é formado por $\log_2 Regs$ bits, onde $Regs$ é o número total de registradores da arquitetura. O índice para a tabela de padrões é composto por $\log_2 Pad$ bits, onde Pad é o número máximo de padrões suportados pela arquitetura. Os bits de anulação são formados por $(L + C)$ bits, onde L é o número de linhas e C o número de colunas da matriz de execução.

Cada campo que representa um registrador de escrita ou um valor imediato possui $\log_2 Regs$ bits, onde $Regs$ é o número total de registradores da arquitetura. A quantidade destes campos depende dos bits que sobram na instrução após definir todos os outros campos anteriores. É importante ressaltar que caso não seja possível armazenar os registradores de escrita ou imediatos nos seus respectivos campos, os campos reservados para leitura podem ser utilizados caso ainda exista espaço, mas a recíproca não é válida. Ou seja, se um registrador de leitura for armazenado em um campo destinado a registradores de escrita ou imediatos, a leitura deste registrador não ocorrerá.

Considerando uma arquitetura com 32 registradores ($Regs = 32$), 8 portas de leitura ($R = 8$), suporte para 64 padrões de instrução ($Pad = 64$) e matriz de execução com 4 linhas ($L = 4$) e 4 colunas ($C = 4$), neste caso são necessários 8 campos para representar registradores de leitura (cada campo com 5 bits, total de 40 bits), 6 bits para o índice da tabela de padrões, 4 bits para anulação de linhas e 4 bits para anulação de colunas. Supondo uma instrução de 64 bits, restam 10 bits que são utilizados como 2 campos de 5 bits para representar registradores de escrita e imediatos (Figura 3.3).

Um padrão é composto por $L \times C$ operações, ou seja, o mesmo número de UFs da

Figura 3.3: Exemplo de uma instrução PBIW com 64 bits

matriz de execução. Cada operação é formada por um *opcode* e um número fixo de operandos. Cada operando possui $\lceil \log_2 X \rceil$ bits, onde X é a quantidade total de registradores representados na instrução codificada (considerando os registradores de leitura, escrita e imediatos). O número de operandos deve ser suficiente para representar qualquer operação. Por exemplo, uma operação como (`addi r1, r2, 15`) precisa de 3 operandos caso os valores imediatos sejam representados por apenas um dos campos da instrução codificada (5 bits). Se forem necessários 2 campos da instrução para representar um imediato (10 bits), então, o número de operandos no padrão deve ser 4 (um para `r1`, um para `r2` e dois para o imediato 15).

Considerando a instrução da Figura 3.3, o seu respectivo padrão possui 16 operações. Cada operação possui 5 operandos (considerando imediatos de 15 bits, que utilizam três campos da instrução codificada), sendo que cada operando deve ter $\lceil \log_2 10 \rceil = 4$ bits. Supondo que existam 256 operações diferentes na arquitetura (8 bits para representar os *opcodes*), então, cada operação possui $(8 + 4 + 4 + 4 + 4 + 4)$ 28 bits. Assim, cada padrão utiliza (28×16) 448 bits, conforme Figura 3.4. Os campos foram numerados de 1 a 16 na figura apenas para facilitar a compreensão. A numeração correta seria de 0 a 15.

Com exceção dos *opcodes*, todos os outros campos (operandos) do padrão são ponteiros para campos da instrução codificada (Figura 3.5), ou seja, eles armazenam o índice referente ao respectivo campo da instrução. Supondo que um dos operandos de uma operação seja o registrador `r1` e que este esteja armazenado na posição "2" da instrução codificada, então, o valor armazenado pelo operando do padrão é o número "2". Em outras palavras, na instrução decodificada, este operando do padrão será substituído pelo valor que estiver armazenado na posição "2" da instrução codificada. Na Figura 3.5, por

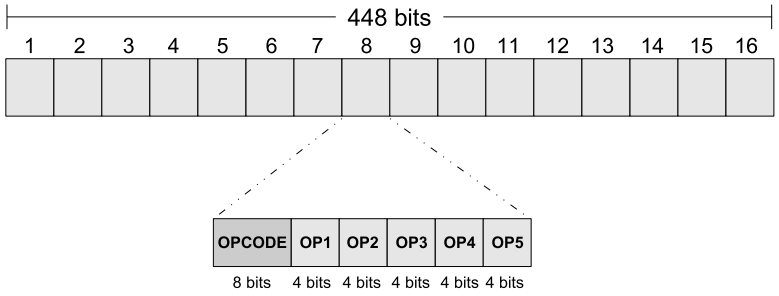


Figura 3.4: Exemplo de uma padrão com 448 bits

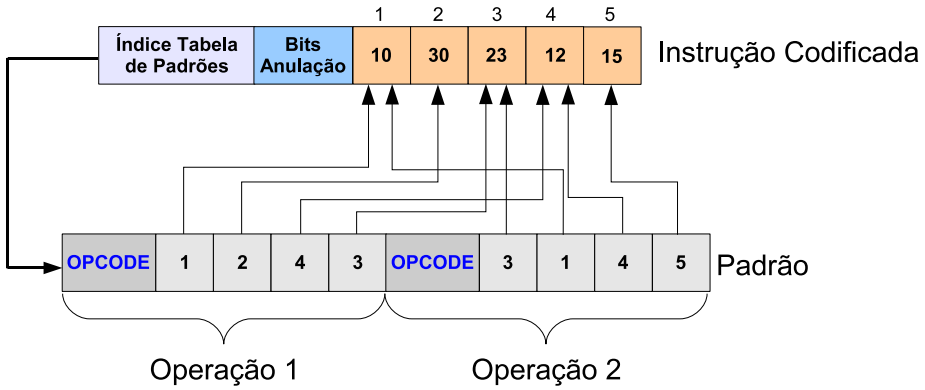


Figura 3.5: Ilustração dos ponteiros utilizando uma instrução codificada e um padrão hipotéticos

exemplo, o primeiro operando da "Operação 1" receberá o valor "10", já o último operando da "Operação 2" receberá o valor 15 quando for decodificado. Durante a execução de uma operação, a UF consegue identificar se o valor "10" refere-se a um imediato ou ao registrador `r10`, através da interpretação do *opcode* da operação.

Supondo que o padrão da Figura 3.4 seja o respectivo padrão da instrução na Figura 3.3, neste caso o total de bits necessários para representar uma instrução PBIW completa é (64 + 448) 512 bits. Porém o tamanho total do programa não deve ser calculado como o número de instruções × 512 bits, pois existe reuso de padrões (Figura 3.6), ou seja, várias instruções podem utilizar o mesmo padrão.

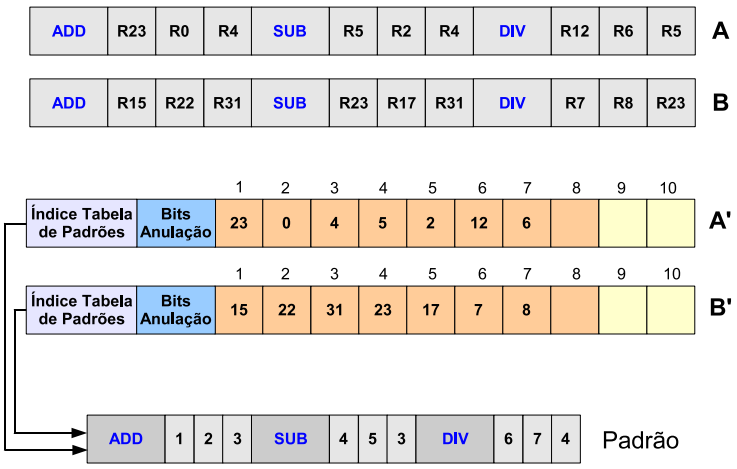


Figura 3.6: Instruções compartilhando o mesmo padrão

O tamanho do programa deve ser calculado como: número de instruções × 64 bits + número padrões × 448 bits. A Figura 3.6 mostra um único padrão utilizado por duas instruções diferentes. As instruções A e B são simplesmente agrupamentos de três operações gerando uma instrução VLIW hipotética. As instruções A' e B' são as respectivas instruções codificadas no modelo PBIW. Apesar de simplificado na figura, se for considerado um padrão de 448 bits, caso não houvesse reuso, cada instrução PBIW teria seu próprio padrão e o total de bits seria 2 × (64 + 448) bits, ou seja, 1024 bits. Porém, como as duas instruções utilizam o mesmo padrão, o tamanho total é (2 × 64) + 448, ou seja, 576 bits, gerando um ganho de 448 bits.

Na próxima seção são apresentadas as características da *cache* de padrões (P-*cache*), estrutura fundamental dentro do mecanismo de codificação PBIW. Ao final do texto é descrito o algoritmo para alocação de linhas na P-*cache* com o objetivo de obter uma *cache* de padrões com alto desempenho.

3.3 Pattern Cache

A *cache* de padrões (*Pattern Cache*) é uma *cache* simples que armazena os padrões detectados durante a codificação das instruções PBIW. As características como associatividade,

número de palavras por bloco, tamanho e política de substituição devem ser definidas de forma a atingir o melhor desempenho possível para o conjunto de aplicações alvo. Em outras palavras, ao implementar o esquema de codificação PBIW, deve-se avaliar a melhor estrutura da P-*cache* utilizando *benchmarks* de programas com características similares aos que serão executados na arquitetura.

A instrução utilizada durante os estágios de execução é formada parcialmente pelos dados da palavra em memória e parcialmente pelos dados dos padrões armazenados na P-*cache*. Durante o estágio de decodificação, os dados da P-*cache* são complementados com os dados da instrução codificada em memória para compor a instrução utilizada no estágio de execução, conforme é apresentado na Figura 3.21 da Seção 3.5.

Devido à variação da freqüência de utilização dos dados da P-*cache* durante a execução do programa, é importante manter os dados mais freqüentes o máximo possível na *cache*. Ou seja, deve-se evitar que os padrões mais utilizados compartilhem linhas com outros padrões, principalmente com padrões muito utilizados (freqüência de utilização alta).

Por se tratar de uma *cache* de dados, onde não há necessidade de manter as informações na mesma seqüência em que são utilizadas durante a execução do programa, os dados da P-*cache* podem ser organizados de forma a apresentar maiores ganhos de desempenho. Para isso, deve-se tentar atribuir endereços exclusivos aos padrões mais freqüentes e endereços compartilhados com o maior número de padrões, para os menos freqüentes. A freqüência de utilização de um padrão pode ser obtida através da análise de *trace* de execução do programa.

Quando há necessidade de compartilhar uma linha da *cache*, o ideal é que não exista intersecção de ocorrência entre os padrões que compartilham a linha. Ou seja, se todas as ocorrências de um determinado padrão A aparecerem somente antes (ou depois) de todas as ocorrências de um padrão B, então estes dois padrões não possuem intersecção de ocorrência. Isto significa que mesmo compartilhando a linha na *cache*, um padrão fará com que o outro saia da *cache* para ocupar o seu lugar uma única vez, ou seja, ocorrerão apenas *misses* compulsórios para estes padrões.

O problema de tentar maximizar o uso das linhas de uma *cache* e ao mesmo tempo

minimizar o número de linhas necessárias tentando manter apenas padrões sem intersecção de ocorrência em uma mesma linha é $\mathcal{NP}$-completo e pode ser modelado como um problema de coloração de grafos da seguinte forma: Seja I o conjunto de padrões que ocorrem pelo menos uma vez em um programa. Cada padrão é representado por um nó de um grafo $G(V, A)$ e dois padrões (nós) P_i e P_j são ligados por uma aresta se possuírem intersecção de ocorrência. Desta forma, o objetivo é encontrar o menor número de cores necessário para colorir todo o grafo. Na Figura 3.7 os padrões referentes às instruções A e C não possuem intersecção de ocorrência enquanto que existe intersecção entre os padrões referentes à A e B, B e C, C e D e também entre B e D.

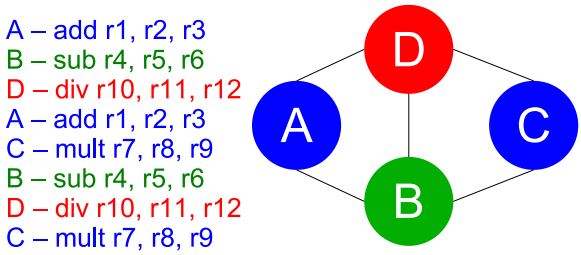


Figura 3.7: Intersecção de ocorrência

O Algoritmo 1 é uma proposta para tentar compartilhar o máximo possível cada uma das linhas da P-*cache* e conseqüentemente minimizar o número de linhas necessárias. É importante ressaltar que este algoritmo não é ótimo, é apenas uma estratégia para melhorar o desempenho da P-*cache*.

Para cada padrão P do conjunto CP (Linha 1) deve-se procurar uma linha L na P-*cache* onde os padrões que já estão associados à esta linha não possuam intersecção de ocorrência com relação ao padrão P (Linha 2). Ao encontrar uma linha L nestas condições, o padrão P passa a fazer parte desta linha (Linha 3). Caso não seja possível encontrar uma linha para o padrão P sem intersecção de ocorrências, deve procurar por uma linha onde esta intersecção seja a menor possível (Linha 5). Para isso devem ser analisados dois fatores: a quantidade de padrões que já ocupam a linha e a freqüência com que estes padrões são utilizados. Por exemplo, uma linha com apenas dois padrões já associados pode ser pior que uma linha onde já existam 10 padrões se, na primeira, a

Algoritmo 1 Algoritmo de alocação de linhas na P-*cache*.

ENTRADA: Conjunto de padrões CP, Conjunto de linhas CL da P-*cache*
SAÍDA: Elementos de CP mapeados em CL
Seja $p(L)$ a função que retorna o conjunto de padrões armazenados na linha L
Seja $O(P)$ o conjunto de ocorrências do padrão P durante a execução do programa
AlocaLinha(CONJUNTO_PADRÕES: CP**, CONJUNTO_LINHAS:** CL**)**
1. **Para** $P \in CP'$
2. **Se** $\exists\ L \in CL \mid O(P) \cap O(p(L)) = \emptyset$
3. $p(L) = p(L) \cup P$
4. **Senão**
5. $p(L) = p(L) \cup P$, para L, P onde $(O(P) \cap O(p(L))$ é mínima
6. **Fim Se**
7. **Fim Para**

freqüência de utilização de cada padrão for bem maior que a freqüência dos padrões da segunda linha. Isto está diretamente associado ao número de vezes em que um padrão fará com que o outro saia da P-*cache*, ou seja, o número de *misses* que será gerado.

É importante ressaltar que o Algoritmo 1 é apenas uma proposta das várias possíveis para tentar melhorar a disposição dos dados na P-*cache* com o objetivo de reduzir o número de *misses*. Algumas simplificações como simplesmente alocar qualquer linha para um padrão caso não encontre linha sem intersecção de ocorrências podem ser adotadas caso seja observado em experimentos que a diferença em número de *misses* é pequena.

Outra abordagem mais simples (menor custo de processamento) e que pode dar ganhos significativos é simplesmente atribuir os padrões às linhas em ordem decrescente de freqüência de utilização. Isto se justifica pelo fato de evitar que os padrões mais freqüentes compartilhem a mesma linha. Por exemplo, se o padrão P_i ocorre N vezes, P_j ocorre N - 1 vezes e P_k ocorre N - 200 vezes deve ser melhor deixar P_i compartilhar a linha L com P_k pois provavelmente P_k removerá P_i da linha L menos vezes do que P_j removeria.

Qualquer estruturação da P-*cache* baseada em *trace* de execução pode não ser tão útil se o código for executado apenas uma vez, porém é possível fazer recodificação dinâmica ou rodar o programa com um conjunto de dados pequeno primeiro para pegar essa estatística e depois recompilá-lo.

Na próxima seção é apresentado todo o processo de codificação de uma instrução

PBIW. O texto descreve os algoritmos de codificação e de junção de padrões, além de demonstrar passo a passo um exemplo de codificação de um conjunto de operações.

3.4 Codificação da Palavra

A codificação da palavra PBIW ocorre durante a fase final de compilação do programa. O compilador percorre todas as operações e, baseado no conjunto de operandos utilizados, cria a instrução codificada e o respectivo padrão. O algoritmo considera que todas as dependências ou foram resolvidas por software ou serão por hardware. Estas dependências não são verificadas pelo algoritmo pois elas estão diretamente associadas à arquitetura e por isso devem ser resolvidas estaticamente (pelo compilador) ou dinamicamente (pelo hardware). A Figura 3.8 ilustra o fluxo de execução da técnica para obter uma instrução codificada e o padrão.

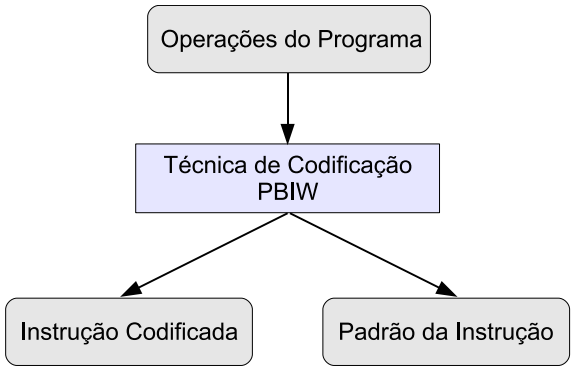


Figura 3.8: Fluxo de execução da técnica de codificação

O algoritmo de codificação analisa todas as operações na ordem gerada pelo escalonador. Para cada conjunto de N operações (N = número de UFs da arquitetura), a primeira tarefa do algoritmo é criar uma instrução codificada vazia I e um padrão vazio P. Considere um conjunto de operações S_j onde, para cada operação op_k de S_j , $1 <= k <= N$ o algoritmo armazena o código da operação op_k em um campo de P_j e verifica se os operandos de op_k estão na instrução codificada Ij . Em caso positivo, o algoritmo adiciona um ponteiro para esses operandos em P_j. Considerando que operandos dos registradores glo-

bais são lidos no estágio de decodificação (em paralelo com a decodificação da instrução), esses operandos são armazenados nas posições iniciais de I_j.

A quantidade de posições reservadas para os operandos de registradores globais depende do número de portas de leitura disponível no banco de registradores global (conforme apresentado na Seção 3.2). As operações que não puderem fazer parte de I_j (por restrições arquiteturais) devem compor uma nova instrução I_{j+1}. Ao terminar a análise sobre todas as operações de S_j, obtém-se então uma nova instrução codificada I_j e um padrão P_j. No entanto, antes de considerar efetivamente a existência desse padrão P_j, realiza-se uma busca no conjunto de padrões CP já criado com o intuito de averiguar se $\exists\ P_i \in CP \mid P_i = P_j$. Em caso positivo, a instrução I_j deve ser atualizada com a informação do endereço do padrão P_i enquanto que o padrão P_j será descartado. Em caso negativo, P_j é adicionado ao conjunto CP e tem seu endereço inserido em I_j.

O Algoritmo 2 apresenta todos os passos do processo de codificação. A entrada para o algoritmo é o conjunto de operações (CO), obtido após as fases de escalonamento e alocação de registradores. Dois conjuntos estão disponíveis na saída: o conjunto de instruções codificadas (CI) e o conjunto de padrões (CP).

Para cada grupo O de N operações (Linha 1), o algoritmo cria uma instrução vazia (Linha 2) e um padrão vazio (Linha 3). Para cada operação *op* pertencente ao grupo O (Linha 4), é inserido o respectivo *opcode* no padrão (Linha 5). Para cada operando *opnd* de uma operação *op* (Linha 6), é verificado se o operando já existe na instrução codificada I (Linha 7). Para os operandos que são lidos do banco de registradores, é verificado se o operando já existe em alguma das posições reservadas para este tipo de operando. Caso não exista nestas posições mas exista nas posições reservadas para imediatos e operandos de escrita, ocorre uma troca de posição (se ainda houver espaço). Ou seja, o operando é colocado em alguma das posições destinadas à leitura e um espaço é liberado dentre os operandos de escrita/imediatos. No caso de operandos de escrita/imediatos, é verificado se já existe em qualquer uma das posições de I.

Caso o operando não exista em I é verificado se ainda há espaço livre na instrução codificada (Linha 8). Para operandos que deverão ser lidos do banco de registradores glo-

Algoritmo 2 Algoritmo de codificação

ENTRADA: Conjunto de operações CO.
SAÍDA: Conjunto de instruções codificadas CI e padrões CP.
Codifica(CONJUNTO_OPERACOES: CO**)**
Seja O um grupo de N (total de UFs da arquitetura) operações escalonadas.

```
1. Para O ∈ CO
2.    cria novo I;
3.    cria novo P;
4.    Para op ∈ O
5.       P.add(op.opcode);
6.       Para opnd ∈ op
7.          Se op.opnd ∉ I
8.             Se espaco_livre(I) < 1
9.                Se P ∉ CP
10.                  CP = CP ∪ P;
11.               Fim Se
12.               I.add(índice de P em CP);
13.               CI = CI ∪ I;
14.               cria novo I;
15.               cria novo P;
16.            Fim Se
17.            I.add(op.opnd);
18.         Fim Se
19.         P.add(índice de op.opnd em I);
20.      Fim Para
21.   Fim Para
22.   Se P ∉ CP
23.      CP = CP ∪ P;
24.   Fim Se
25.   I.add(índice de P em CP);
26.   CI = CI ∪ I;
27. Fim Para
```

bal, esta verificação é realizada considerando apenas as posições destinadas para este tipo de operando. Para os operandos de escrita/imediatos basta ter um espaço em qualquer posição da instrução I.

Caso não exista mais espaço na palavra, é verificado se o padrão P já existe no conjunto de padrões CP (Linha 9). Se o padrão ainda não existir, então ele é adicionado ao conjunto de padrões (Linha 10). Além disso, ainda no caso em que não existe mais espaço livre na instrução I, o índice que aponta para a tabela de padrões é atualizado na instrução com a posição do padrão P na tabela de padrões em memória (Linha 12). A instrução I passa a fazer parte do conjunto de instruções CI (Linha 13) e são criados uma nova instrução I e um novo padrão P (Linhas 14 e 15).

Se existir espaço na instrução codificada I (ou se for criada uma nova instrução), o operando *op.opnd* é adicionado à instrução (Linha 17). Logo após, o índice da posição do operando na instrução é adicionado no padrão (Linha 19). No final do algoritmo, o último padrão criado é adicionado ao conjunto de padrões (Linha 23) caso não exista (Linha 22). O índice da instrução I que aponta para o padrão na tabela de padrões é atualizado (Linha 25) e, finalmente, a instrução codificada I é adicionada ao conjunto de instruções CI (Linha 26).

Como o número de operações de uma instrução é constante, a complexidade do algoritmo é limitada pelo tamanho do conjunto O e pela complexidade da consulta ao conjunto de padrões. Em uma situação de pior caso $|CP| = |O|$ e, portanto, $\mathcal{O}(|O|)$.

As próximas figuras descrevem todos os passos para construir uma instrução codificada e seu padrão utilizando o a técnica PBIW sobre o fragmento de código apresentado na Figura 3.9. Este fragmento de código é um exemplo com apenas quatro operações, de 32 bits cada, totalizando 128 bits. Em cada etapa, o algoritmo atualiza a instrução codificada e seu respectivo padrão. As setas indicam os operandos utilizados por cada operação inserida no padrão. Neste exemplo, a instrução codificada possui 10 campos sendo que os campos 1 até 5 são reservados para leitura dos operandos do banco de registradores globais e os campos de 6 a 10 são utilizados para registradores de escrita e imediatos. Além disso, é considerado que um imediato pode ser representado com apenas

5 bits, ou seja, um campo na palavra é suficiente para representar um imediato neste exemplo.

```
add r1, r2, r3
addu r4, r2, r6
addi r7, r6, 9
subu r9, r10, r6
```

Figura 3.9: Fragmento de código

A codificação é iniciada pela operação `add r1, r2, r3` (Figura 3.10). O *opcode* `add` é inserido no padrão. Depois disso, o primeiro registrador `(r1)`, que é um registrador de escrita (destino), é armazenado no campo "6" da instrução codificada. Um ponteiro para o campo "6" é armazenado em uma linha da *cache* de padrões. A codificação dos dois registradores de leitura (`r2` e `r3`) segue o mesmo procedimento, mas utiliza os campos reservados para leitura dos registradores globais.

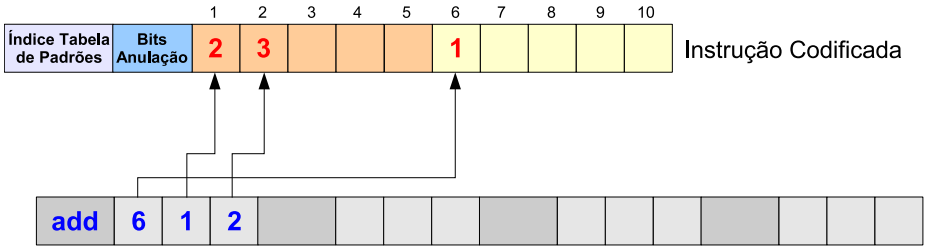


Figura 3.10: Codificação da operação `add r1, r2, r3`

Assim como na operação anterior, o primeiro passo para codificar `addu r4, r2, r6` (Figura 3.11) é inserir o seu respectivo *opcode* no padrão, depois armazenar o valor do registrador de escrita na instrução codificada (e seu ponteiro no padrão) e finalmente os registradores de leitura (e seus ponteiros no padrão). Como esta operação também utiliza o registrador `r2`, o campo "1" da instrução codificada será reutilizado, ou seja, neste ponto existem dois ponteiros no padrão para o campo "1" da instrução.

A codificação da terceira operação `addi r7, r6, 9` (Figura 3.12) segue os mesmos

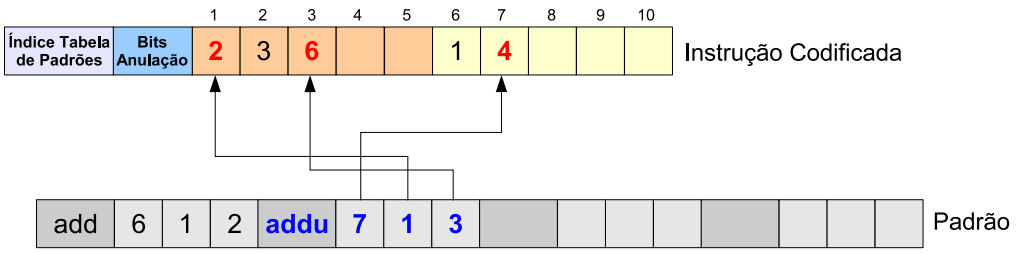


Figura 3.11: Codificação da operação `addu r4, r2, r6`

passos das operações anteriores e neste caso o campo "3" da instrução codificada será reutilizado pois o valor "6" já havia sido utilizado na codificação da operação anterior.

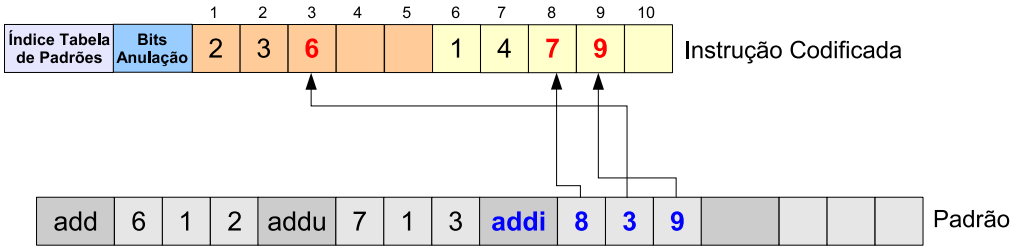


Figura 3.12: Codificação da operação `addi r7, r6, 9`

Na última operação (`subu r9, r10, r6`) ocorre a reutilização dos campos "3" e "9". É importante observar que ao armazenar o imediato "9" na instrução, este campo pode ser utilizado também pelo registrador `r9` uma vez que durante a execução, a UF saberá exatamente, através do *opcode*, se o valor trata-se de um imediato ou de um registrador.

No último passo da codificação (Figura 3.14) tem-se uma instrução codificada e um padrão. A instrução codificada é armazenada na *cache* de instruções enquanto que o padrão será armazenado na *cache* de padrões. Neste passo a informação sobre o endereço do padrão é adicionada à instrução codificada.

Após encerrar o algoritmo de codificação, existe um conjunto de instruções codificadas e um conjunto de padrões, sendo que muitos destes são compartilhados por várias instruções. Porém, alguns destes padrões podem ser esparsos, possuindo muitos espaços

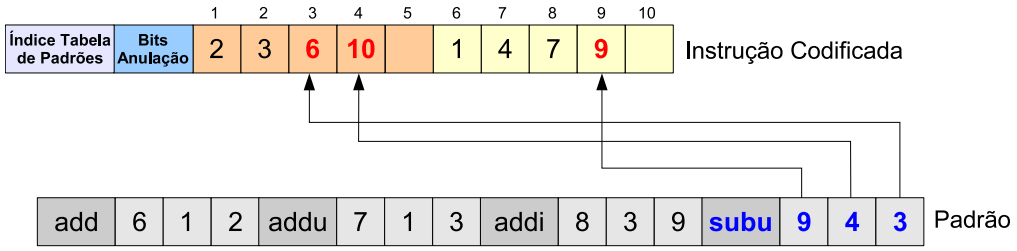


Figura 3.13: Codificação da operação `subu r9, r10, r6`

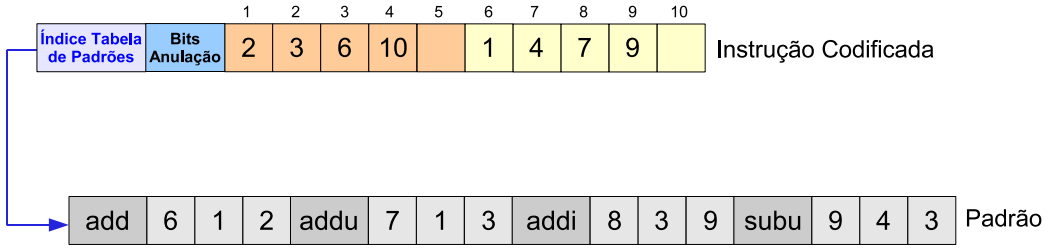


Figura 3.14: Atualização do endereço do padrão na instrução codificada

vazios, dependendo das restrições arquiteturais aplicadas durante o escalonamento das operações. Com o objetivo de obter um conjunto de padrões menor e mais denso e, conseqüentemente, reduzir o tamanho do programa, utiliza-se uma técnica para unir padrões.

A junção de padrões pode ser considerada uma otimização que une dois padrões quando a soma das operações de ambos é menor ou igual a N (número de UFs da arquitetura). Após a junção de dois padrões, as instruções que apontavam para os padrões antigos devem apontar para o novo padrão originado a partir da junção. Mais importante, cada instrução deve anular operações no padrão que não serão utilizadas durante a execução. Este otimização não impacta do desempenho da técnica de codificação pois é baseada na comparação de cada elemento do conjunto CP com o restante dos seus elementos, levando a uma complexidade de $\mathcal{O}(|CP|^2)$.

A Figura 3.15 ilustra um sinal de controle para anulação da execução de uma UF. Nesta figura é considerado um total de 16 UFs (4×4), ou seja, um bit para cada linha e um bit para cada coluna que indica se a linha/coluna deve ser executada ou não. A

seguinte lógica pode ser utilizada para saber se uma UF deve executar a operação: se o bit referente à linha E o bit referente à coluna estiverem com o valor "1", então a operação deve ser executada. Caso contrário ela deve ser anulada.

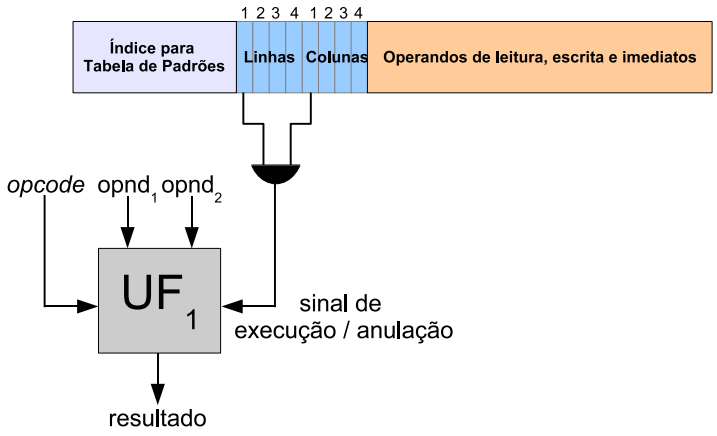


Figura 3.15: Ilustração de um esquema de anulação

O algoritmo de junção de padrões percorre o conjunto de padrões CP e para cada padrão P_i verifica se este padrão pode ser unido ao padrão P_j, onde $j = i+1, ..., |CP|$, de acordo com a utilização atual das linhas ou colunas da matriz de execução. Para que dois padrões possam ser unidos não pode haver intersecção de linhas ou colunas entre suas respectivas matrizes de UFs, ou seja, se o padrão P_i utiliza as colunas "1" e "3" então o padrão P_j pode utilizar no máximo as colunas "2" e "4" para que os dois possam ser unidos. De forma análoga, se o padrão P_i utiliza as linhas "1" e "2" então o padrão P_j pode utilizar no máximo as linhas "3" e "4" para que os dois possam ser unidos.

Considerando uma matriz de UFs 4 × 4 (Figura 3.16), a Figura 3.17 ilustra dois padrões que podem ser unidos (a) e dois padrões que não podem ser unidos (b) considerando a utilização de colunas da matriz de UFs. Na Figura 3.17(b) é possível observar que a operação `nor`, `andi` e `addi` do segundo padrão ocupam colunas já utilizadas pelo primeiro padrão.

A junção por linhas tem o comportamento análogo à junção por colunas. A única diferença é o critério utilizado (coluna ou linha) para verificar intersecção entre a utilização

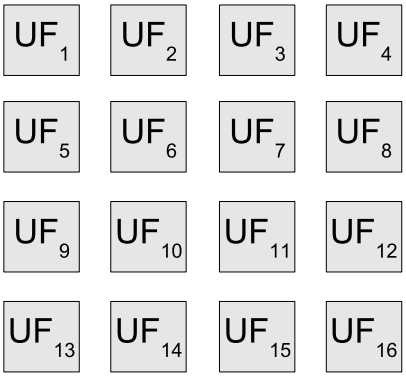


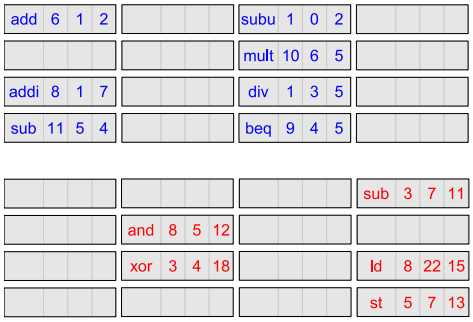
Figura 3.16: Disposição de UFs em uma matriz 4×4

das matrizes de UFs dos padrões candidatos à junção.

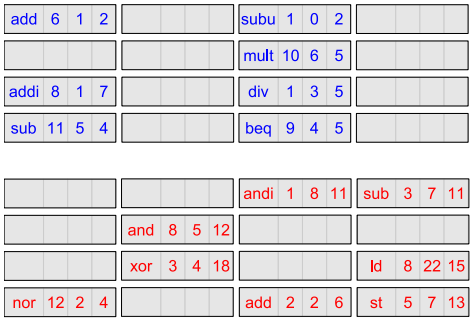
Estes dois critérios (utilização de linhas e colunas) podem ser utilizados em conjunto desde que o segundo método seja aplicado somente após a finalização do primeiro, ou seja, se iniciar com junção por linhas, deve-se unir todos os padrões possíveis e somente depois tentar unir os novos padrões considerando a utilização de colunas.

Antes de iniciar um segundo método de junção deve-se atualizar o conjunto de padrões, ou seja, dois padrões A e B unidos por colunas criam um novo padrão C que poderá ser unido por linhas a um padrão D mas, ao iniciar a junção por linhas, os padrões A e B já não devem mais fazer parte do conjunto de padrões. Após unir dois padrões por colunas, a utilização das linhas pode ser diferente (dos padrões originais) no novo padrão. A Figura 3.18 ilustra a junção de dois padrões por colunas e mostra a utilização de linhas antes e depois desta junção.

Ao unir dois padrões A e B um novo padrão C é criado e os antigos padrões são descartados. Neste momento é preciso alterar os índices de todas as instruções codificadas que apontavam para A ou B. Todas devem apontar para o novo padrão C. Outra informação que precisa ser preenchida nas instruções é o conjunto de bits de anulação. Considerando que o valor "1" indica execução e o valor "0" indica anulação, então, para que a $UF_{i,j}, i = 1, ..., M$ e $j = 1, ..., N$ (M é o número de linhas da matriz de UFs e N o número de colunas) execute a operação, ambos os bits referentes a i e j devem ter o



(a) Padrões que podem ser unidos



(b) Padrões que não podem ser unidos

Figura 3.17: Padrões que podem ser unidos (a) e padrões que não podem ser unidos (b) considerando utilização de colunas

valor "1". A Figura 3.19 ilustra o valor dos bits de anulação após junção ocorrida na Figura 3.18.

O Algoritmo 3 apresenta todos os passos do processo de junção de padrões independente da forma como é feita a verificação de intersecção (linhas ou colunas). A entrada para o algoritmo é o conjunto de padrões (CP) e o conjunto de instruções codificadas (CI), obtidos após a execução completa do Algoritmo 2. Dois conjuntos estão disponíveis na saída: o conjunto de instruções codificadas (CI'), com o mesmo número de elementos, mas com os bits de anulação preenchidos e o conjunto de padrões (CP'), com menos elementos que o conjunto disponível na entrada.

Para cada padrão P (Linha 1), é realizada uma comparação entre ele e cada um dos outros padrões (Linha 2) verificando se existe intersecção na utilização das colunas (linhas) entre os dois (Linha 3). Caso não exista intersecção, os padrões são unidos (Linha 4). O algoritmo busca todas as instruções que apontavam para os padrões antigos (Linhas

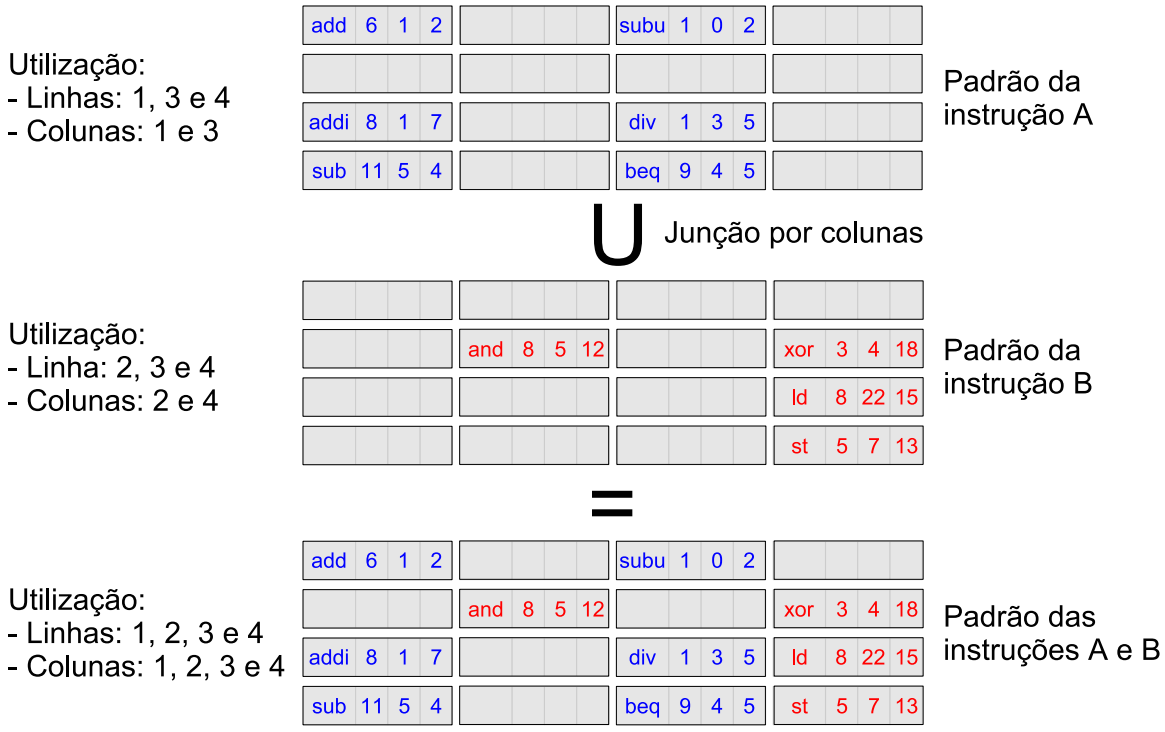


Figura 3.18: Junção por colunas e a representação da utilização das linhas

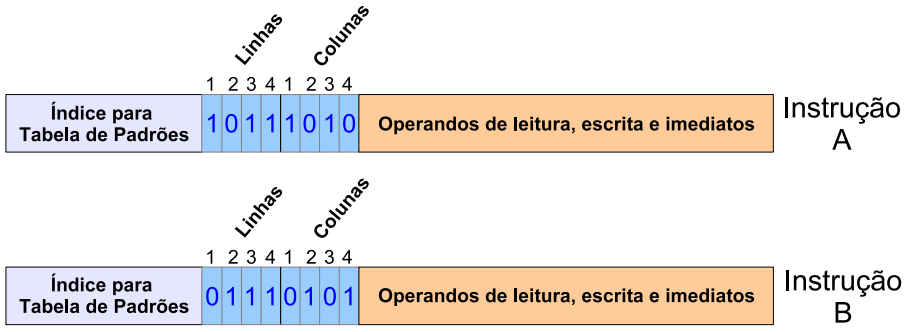


Figura 3.19: Bits de anulação decorrentes da junção ocorrida na Figura 3.18

Algoritmo 3 Junção de Padrões

ENTRADA: Conjunto de padrões CP e instruções codificadas CI.
SAÍDA: Conjunto de padrões CP' e instruções codificadas CI'
Junção(CONJUNTO_PADRÕES: CP**, CONJUNTO_INSTRUÇÕES:** CI**)**

```
1. Para P_i ∈ CP
2.   Para P_j ∈ CP
3.     Se P_i ∩ P_j = ∅
4.       P_j = P_i ∪ P_j;
5.       Para I ∈ CI
6.         Se I.padrao = P_i.indice
7.           I.padrao = P_j.indice
8.           insere_bits_anulacao(I)
9.         Fim Se
10.        Se I.padrao = P_j.indice
11.          insere_bits_anulacao(I)
12.        Fim Se
13.      Fim Para
14.      CP = CP-P_i;
15.    Fim Se
16.  Fim Para
```

5 e 6), atualiza o índice destas instruções para apontar para o novo padrão (Linha 7) e preenche os bits de anulação da instrução (Linhas 8 e 11). Finalmente, após trocar todas as referências ao padrão antigo, este último é eliminado do conjunto de padrões CP.

O algoritmo apresentado é exatamente igual quando aplicado à junção por linhas ou por colunas. A única diferença é a implementação da Linha 3. Quando a junção é por colunas, deve-se verificar se existe alguma coluna em comum (que será executada) nos dois padrões candidatos à junção. No caso de junção por linhas, deve-se verificar se existe alguma linha comum aos dois padrões candidatos. A Figura 3.20 mostra a junção de dois padrões considerando a utilização das colunas e depois, o padrão resultante unindo a outro padrão, considerando a utilização de linhas.

Na próxima seção é apresentado o mecanismo de decodificação de uma instrução PBIW assim como o respectivo circuito decodificador.

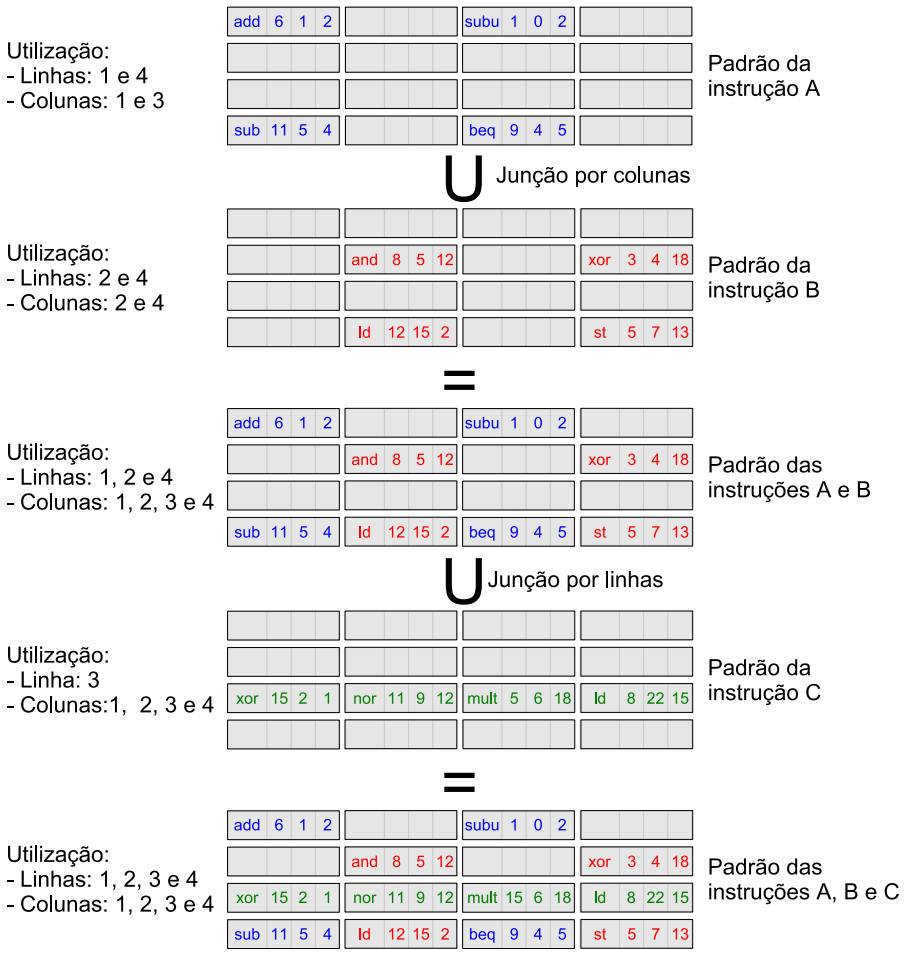


Figura 3.20: Junção por colunas seguida de junção por linhas

3.5 Decodificação da Palavra

Com a introdução das instruções codificadas e seus respectivos padrões, o estágio de decodificação passa a ter uma atividade adicional. Antes de apresentar essa nova atividade, deve-se ter em mente que as instruções codificadas são armazenadas na memória e na *cache* de instruções da mesma forma como instruções tradicionais. Os padrões de instruções são também armazenados na memória, em uma tabela de padrões. A diferença é que os padrões utilizam uma *cache* específica denominada *cache* de padrões (P-*cache*), mas que utiliza os mesmos mecanismos das *caches* comuns com relação a busca de instruções.

No estágio de busca, uma instrução codificada é buscada na *cache* de instruções. No estágio de decodificação, enquanto os registradores globais são lidos, busca-se um padrão de acordo com o campo de endereço presente na instrução codificada. Este campo indica a posição do padrão dentro de uma tabela de padrões armazenada na memória. Caso o padrão não se encontre na P-*cache*, ele é buscado na memória, assim como é feito nas *caches* tradicionais.

As instruções de desvio e instruções alvos de desvio são buscadas normalmente pois o compilador resolve estes saltos entre as instruções. O processo de codificação PBIW ocorre antes da linkedição do programa. As instruções de desvio apontam para *labels* que são calculados automaticamente já levando em conta o algoritmo de codificação das instruções. Esta abordagem não é possível em técnicas de compressão de código pois a criação das instruções compactas ocorre após a linkedição do programa, quando os *labels* já foram calculados.

A P-*cache* é um elemento no *datapath* do processador, ou seja, a busca de padrões na P-*cache* não cria um novo estágio no processo de execução de uma instrução pois é realizada simultaneamente com a leitura dos registradores. De posse do padrão e da instrução codificada, a decodificação da instrução é realizada e, a partir de então, tem-se uma instrução completa cujas operações serão executadas na matriz de UFs. A Figura 3.21 ilustra o processo de decodificação de uma instrução PBIW.

Depois que o padrão é recuperado da *cache* de padrões, o hardware de decodificação utiliza os ponteiros que estão armazenados dentro do padrão, juntamente com a instrução codificada buscada na memória (ou da I-*cache*) para construir a instrução completa que será utilizada durante os estágios de execução. Durante a decodificação, a instrução codificada funciona como um dicionário de operandos para os ponteiros do padrão. Enquanto ocorre a decodificação também ocorre a leitura dos registradores globais. A Figura 3.22 mostra uma visão geral do circuito utilizado no estágio de decodificação. Nesta figura é utilizado um padrão com apenas duas operações por simplicidade.

Após a busca da instrução codificada, já no estágio de decodificação, os números dos registradores que devem ser lidos são enviados às portas de leitura do banco de

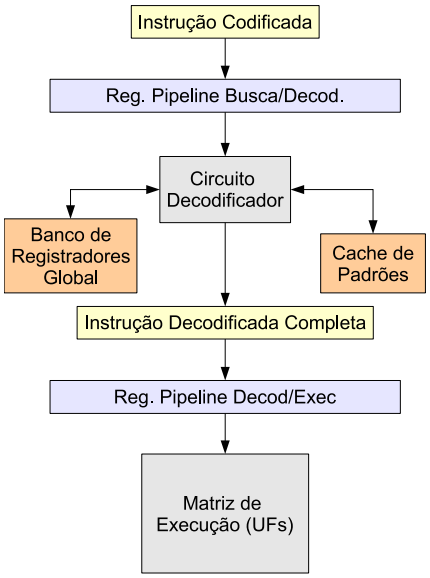


Figura 3.21: Fluxo de decodificação de uma instrução PBIW

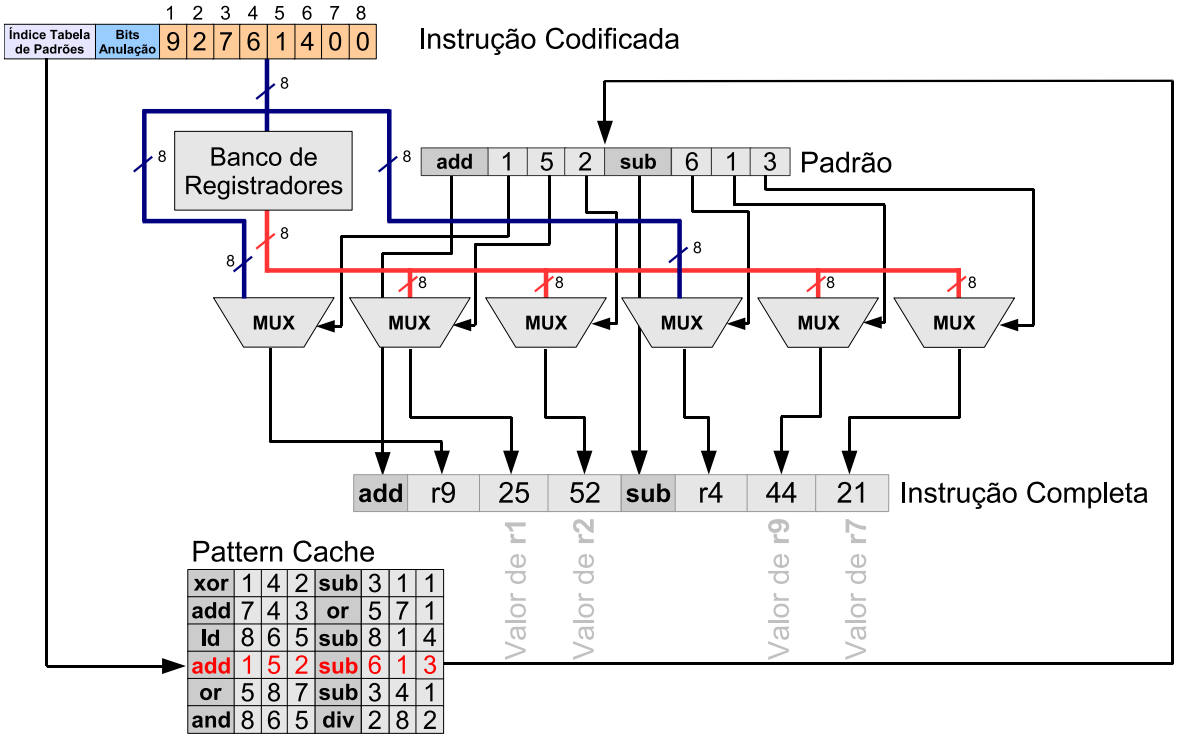


Figura 3.22: Visão geral do circuito de decodificação PBIW

registradores. Durante a leitura dos registradores, o padrão é recuperado da P-*cache* utilizando o índice armazenado na instrução.

Após a leitura dos registradores, os valores são disponibilizados na entrada dos multiplexadores referentes aos operandos de origem, e os números dos registradores/imediatos que não são lidos do banco de registradores são disponibilizados na entrada dos multiplexadores referentes aos operandos de destino. A escolha entre os valores disponibilizados nos multiplexadores é realizada utilizando o ponteiro do respectivo campo no padrão, em outras palavras, o valor armazenado no padrão é a chave de escolha do multiplexador. O número de multiplexadores necessário é exatamente igual ao número de operandos presentes em um padrão.

Na próxima seção é apresentado um estudo de caso utilizando a arquitetura 2D-VLIW com instruções codificadas segundo a estratégia de codificação PBIW.

3.6 Estudo de Caso: Arquitetura 2D-VLIW

A arquitetura 2D-VLIW, descrita no capítulo anterior, foi escolhida para a validação do esquema de codificação PBIW. O principal motivo para esta escolha foi a facilidade na manipulação e utilização da arquitetura uma vez que ela era alvo de um trabalho de doutorado no IC.

Considerando uma instância da arquitetura 2D-VLIW com 4×4 UFs, uma instrução 2D-VLIW possui 512 (16×32) bits. A primeira proposta avaliada para a instrução PBIW nesta arquitetura possui 128 bits (Figura 3.23). Esta instrução possui oito campos de cinco bits para os registradores de leitura (40 bits), 13 campos de cinco bits para registradores de escrita e imediatos (65 bits), 8 bits para anulação e 15 bits para índice da tabela de padrões em memória.

Um padrão para uma instrução PBIW de 128 bits possui 528 bits divididos em 16 campos de 33 bits. Cada um é dividido em um campo de 8 bits para *opcode* (o conjunto de operações 2D-VLIW, derivado do Trimaran via HPL-PD, exige esta quantidade de bits) e 5 campos de 5 bits para os operandos: três campos representam os operandos de escrita (1 campo) e de leitura (2 campos) e dois campos são utilizados para representar os

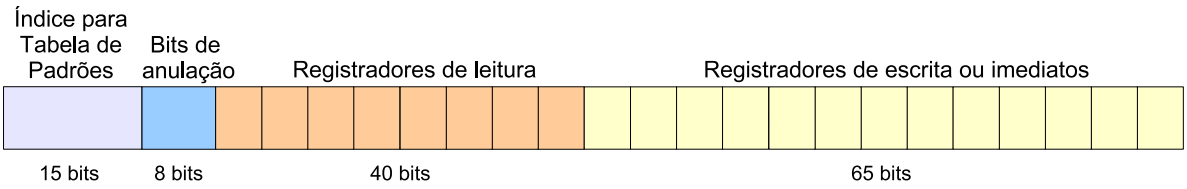


Figura 3.23: Instrução PBIW com 128 bits definida para a arquitetura 2D-VLIW

imediatos (um imediato 2D-VLIW utiliza 15 bits, então estes dois campos são utilizados juntos com mais um dos campos de operandos). Cada campo de operando aponta para um dos 21 campos existentes na instrução codificada (Figura 3.24).

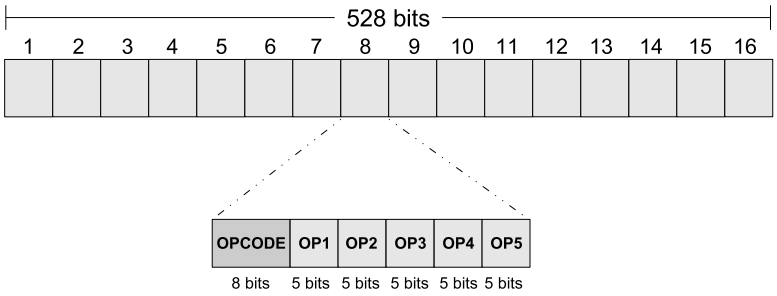


Figura 3.24: Padrão de uma instrução PBIW de 128 bits definido para a arquitetura 2D-VLIW

O total de bits da instrução completa (128 + 528 = 656) é maior que uma instrução 2D-VLIW (512 bits), porém, com a estratégia de codificação, um padrão de 528 bits é utilizado por várias instruções de 128 bits, ou seja, a taxa de redução do programa codificado em relação ao programa não codificado depende exclusivamente da capacidade de reuso de padrões no programa.

Após a realização dos primeiros experimentos para a palavra de 128 bits constatou-se que o maior número de padrões era aproximadamente 6.000, conforme Tabela 3.1. Desta forma, 13 bits seriam suficientes para referenciar todos os padrões na tabela de padrões em memória, para os programas avaliados.

Todos os experimentos com PBIW partiram de um conjunto de operações HPL-PD

Programa	Instruções	Padrões
168wupwise	7.901	1.072
175vpr	40.632	4.952
179art	9.320	1.235
181mcf	5.904	998
183equake	8.651	1.305
197parser	41.948	5.394
255vortex	80.120	5.909
256bzip2	17.712	2.151
300twolf	66.348	5.603
epic	3.075	675
g721decode	1.638	396
g721encode	1.650	397
gsmdecode	9.912	925
gsmencode	13.023	1.319
pegwit	13.649	1.515
Máximo	80.120	5.909

Tabela 3.1: Quantidade de instruções codificadas (128 bits) e padrões por programa

previamente escalonadas segundo algoritmo e restrições de uma arquitetura 2D-VLIW considerando uma matriz de execução de 4×4 UFs. O gráfico apresentado na Figura 3.25 mostra qual o percentual de instruções 2D-VLIW seria possível representar como PBIW sem que houvesse necessidade de quebrar uma instrução 2D-VLIW em mais de uma instrução PBIW.

A partir de 5 campos de dados é possível representar em média 65% das instruções 2D-VLIW como instruções PBIW. Com 8 campos, o que equivale a uma instrução PBIW de 64 bits, pode-se representar 88% das instruções 2D-VLIW e com 21 campos, que seria uma instrução PBIW de 128 bits, em média, poderiam ser representadas todas as instruções 2D-VLIW sem necessidade de qualquer quebra.

Desta forma optou-se por realizar experimentos com ambas as possibilidades, PBIW 64 bits e 128 bits para avaliar que vantagens teria uma codificação 8 vezes menor (em relação a 2D-VLIW), que precisaria quebrar em duas ou mais instruções, aproximadamente 12% das instruções equivalentes em 2D-VLIW e também uma codificação 4 vezes menor mas

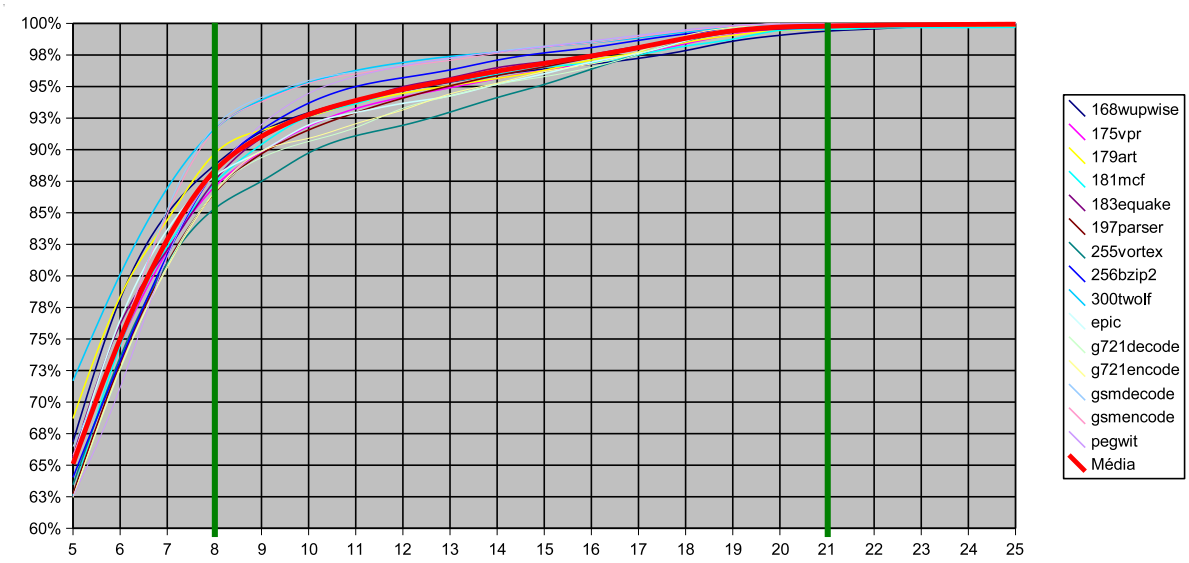


Figura 3.25: Utilização dos campos da palavra PBIW de 128 bits por programa codificado

que praticamente não precisaria de nenhuma quebra, comparando com 2D-VLIW.

Dos 64 bits disponíveis para codificar a instrução, 13 já estavam comprometidos para o índice da tabela de padrões (no próximo capítulo é mostrado que o número de padrões para instruções de 64 bits é menor do que para instruções de 128 bits) e 40 estavam reservados para os 8 registradores de leitura. Então sobravam (64 - 53) 11 bits. Destes 11 bits, 8 seriam utilizados para anulação (linhas e colunas) e portanto restavam apenas três bits, que não seriam suficientes nem para representar um outro campo na palavra (que precisa de 5 bits). Desta forma, estes três bits foram colocados no campo que representa o índice que aponta para a tabela de padrões permitindo codificar programas maiores (mais linhas na P-*cache*) do que os utilizados nos experimentos. A Tabela 3.2 ilustra as principais diferenças entre a codificação PBIW de 64 bits e a codificação PBIW de 128 bits. As colunas P_{64} e P_{128} apresentam, respectivamente, o tamanho em bits de cada um dos parâmetros para as codificações de 64 e 128 bits.

A opção de 64 bits para a instrução PBIW definida possui um decremento da ordem de 8× sobre o tamanho de uma instrução 2D-VLIW (512 bits). Neste caso, a instrução PBIW é dividida em 8 campos de 5 bits que podem armazenar registradores de leitura,

Parâmetro	P_{64}	P_{128}
Total da palavra	64	128
Índice para Tabela de Padrões	16	15
Bits de anulação	8	8
Registradores de leitura	40	40
Registradores de escrita	0	65
Cada campo para registrador	5	5
Total da P-*cache*	368	528
Opcode na P-*cache*	8	8
Operando na P-*cache*	3	5

Tabela 3.2: Comparação entre PBIW de 64 e 128 bits. Valores expressos em bits

escrita ou imediatos, 1 campo de 16 bits para o endereço do padrão (índice para a tabela de padrões), 4 bits para anulação de linha e 4 bits para anulação de colunas (Figura 3.26).

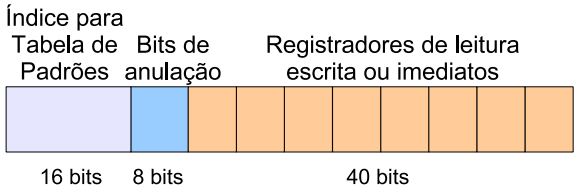


Figura 3.26: Instrução PBIW com 64 bits definida para a arquitetura 2D-VLIW

Um padrão para uma instrução PBIW de 64 bits possui 368 bits divididos em 16 campos de 23 bits. Cada um é dividido em um campo de 8 bits para *opcode* e 5 campos de 3 bits para os operandos: três campos representam os operandos de escrita (1 campo) e de leitura (2 campos) e dois campos são utilizados para representar os imediatos (um imediato 2D-VLIW utiliza 15 bits, então estes dois campos são utilizados juntos com mais um dos campos de operandos). Cada campo de operando aponta para um dos 8 campos existentes na instrução codificada (Figura 3.27).

O total de bits da instrução completa (64 + 368 = 432) é menor do que uma instrução 2D-VLIW (512 bits), e, de forma equivalente à instrução de 128 bits, com a estratégia de codificação, muitas instruções de 64 bits utilizam o mesmo padrão de 368 bits. O próximo

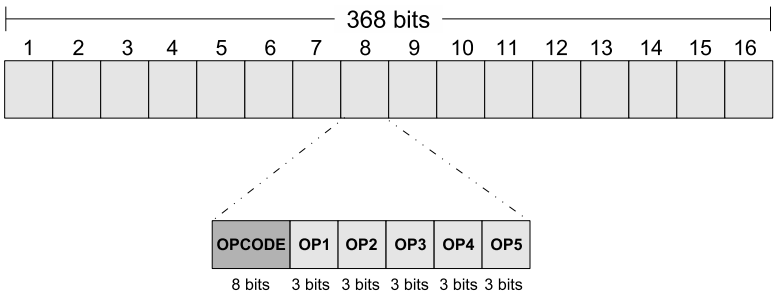


Figura 3.27: Padrão de uma instrução PBIW de 64 bits definido para a arquitetura 2D-VLIW

capítulo comprova esta característica de reuso através dos experimentos e resultados apresentados.

Para as duas opções de codificação (128 ou 64 bits), o campo de endereço do padrão indica a posição de um padrão na tabela de padrões em memória. Os oito bits para anulação de linhas e coluna são responsáveis por indicar quando uma coluna e/ou uma linha da matriz de UFs não pode executar as operações devido à junção de padrões. Cada bit de anulação de coluna é responsável por anular uma das colunas da matriz, ou seja, indica que as operações pertencentes à uma determinada coluna da matriz não devem ser executadas. Os bits que indicam anulação de linhas são análogos.

Na próxima seção são apresentadas as considerações finais relativas ao esquema de codificação PBIW descrito nas seções anteriores.

3.7 Considerações Finais

Neste capítulo foram apresentadas todas as características do esquema de codificação PBIW assim como as justificativas para cada uma das decisões tomadas com relação à estrutura da instrução.

Pode-se notar que esta codificação é bastante flexível e adaptável para qualquer arquitetura que trabalhe originalmente com instruções grandes. O esquema de junção de padrões é apenas um complemento para melhorar a densidade e diminuir o número de

padrões e desta forma a sua implementação não é obrigatória.

O algoritmo de codificação é simples e, portanto, não gera muito impacto no tempo total de compilação e o mecanismo de decodificação permite que a leitura dos registradores seja realizada em paralelo à composição da instrução que será executada. Isto minimiza o *overhead* de montagem da instrução.

No próximo capítulo são apresentados os experimentos e os resultados obtidos ao comparar a estratégia de codificação PBIW com outros modelos como a codificação 2D-VLIW e a estratégia de codificação EPIC. Também serão apresentadas comparações entre os formatos de 64 bits e de 128 bits para a palavra PBIW.

Capítulo 4

Experimentos e Resultados

O esquema de codificação PBIW apresentado no capítulo anterior propõe a redução no tamanho do programa em memória utilizando um algoritmo de codificação, um circuito decodificador simples e uma nova *cache* (P-*cache*) para armazenar os padrões que são utilizados na composição da instrução durante o estágio de decodificação.

Após a decodificação da instrução PBIW, em alguns casos, o número de bits total utilizado pela instrução e seu respectivo padrão pode ser maior do que o número de bits utilizado por uma instrução VLIW (ou EPIC) equivalente, porém, o diferencial desta técnica de codificação está no fato de que a instrução é bastante pequena e o reuso dos padrões é grande o suficiente para reduzir o tamanho dos programas e, conseqüentemente, aumentar o desempenho do processador.

Este capítulo cobre os experimentos realizados para validar empiricamente as considerações que compõem esta dissertação e que foram discutidas nos capítulos anteriores. Ao longo deste capítulo, procura-se apresentar e justificar os experimentos escolhidos assim como a infraestrutura utilizada na realização desses experimentos. Por não possuir ainda uma implementação real em hardware, os experimentos são baseados na simulação da execução e da decodificação utilizando a codificação PBIW para determinar a eficiência deste método.

4.1 Experimentos e Medidas de Desempenho

Os experimentos apresentados neste capítulo foram realizados através da geração real de código PBIW para a arquitetura 2D-VLIW (descrita na Seção 2.5 do Capítulo 2). O resultado (instruções codificadas) foi comparado ao código gerado de acordo com os esquemas de codificação 2D-VLIW e EPIC derivado de 2D-VLIW (ilustrado na Figura 3.1 do Capítulo 3).

Em todos os experimentos, procurou-se utilizar o mesmo conjunto de programas retirados de três *benchmarks* diferentes, com o intuito de facilitar análises comparativas e, ao mesmo tempo, utilizar um conjunto de programas que seja representativo sobre a metodologia avaliada. Estes programas são provenientes dos pacotes de *benchmarks* SPEC2000 [13] (SPECint - programas que lidam com números inteiros e SPECfp - programas que lidam com números de ponto flutuante) e MEDIABench [19] que foram escolhidos por serem amplamente utilizados em vários tipos de avaliação de desempenho e também pela variedade das aplicações que representam. Os programas escolhidos e utilizados nos experimentos discutidos neste capítulo são apresentados na Tabela 4.1.

Todos os programas (em todas os esquemas de codificação) foram compilados com o compilador Trimaran [5] (versão 3.7) e as opções de formação de hiperblocos e desenrolamento de laços (em até 32 vezes) habilitadas. Foram realizados dois tipos de experimentos: estático e dinâmico. Os experimentos estáticos indicam o número de instruções obtidas em cada técnica de codificação, o número de padrões (apenas para PBIW), taxa de reuso e fator de redução. Os experimentos dinâmicos comparam o tempo de execução do programa, considerando o desempenho da I-*cache*, para cada técnica de codificação, contra o desempenho da I-*cache* + P-*cache* utilizadas no esquema de codificação PBIW. Em ambos os experimentos (estático e dinâmico), o modelo de execução 2D-VLIW foi especificado através da linguagem de descrição de arquiteturas denominada HMDES [12].

Os experimentos apresentados na Seção 4.3 procuram determinar como a estratégia de codificação PBIW afeta o tamanho do código dos programas quando comparada a outras estratégias de codificação. Nesse caso, duas medidas merecem destaque: Reuso e Fator de Redução. A primeira medida indica a quantidade média de instruções codificadas

Programa	**Benchmark**	**Descrição**
168wupwise	SPECfp	Cálculos de cromodinâmica quântica
175vpr	SPECint	Análise e síntese de circuitos digitais
179art	SPECfp	Redes neurais e reconhecimento de imagens
181mcf	SPECint	Otimização de veículos em transporte público
183equake	SPECfp	Simulação da propagação de ondas sísmicas
197parser	SPECint	Parser de sentenças em inglês
255vortex	SPECint	Banco de dados orientado à objetos
256bzip2	SPECint	Compressão e descompressão de arquivos
300twolf	SPECint	Posicionamento e conexão de transistores
epic	MEDIABench	Compressão de imagens por meio de *wavelets*
g721decode	MEDIABench	Decodificação de sinais de voz
g721encode	MEDIABench	Codificação de sinais de voz
gsmdecode	MEDIABench	Decodificação de sinais de voz
gsmencode	MEDIABench	Codificação de sinais de voz
pegwit	MEDIABench	Criptografia e autenticação

Tabela 4.1: Programas utilizados nos experimentos

utilizando um mesmo padrão, ou seja, é a medida que verifica se há sobrejeção entre o conjunto de instruções e o conjunto de padrões. A segunda medida informa de quanto foi a redução no tamanho de código de um programa a partir do uso da técnica de codificação. É importante observar que, quanto maior o reuso e o fator de redução, mais eficiente é a técnica de codificação.

Os experimentos apresentados na Seção 4.4 procuram determinar como a estratégia de codificação PBIW afeta o desempenho dos programas quando comparada a outras estratégias de codificação. Nesse caso, duas medidas merecem destaque: *Overhead* dos *Misses* e Tempo de Execução. A primeira medida indica o número de ciclos gastos (além dos gastos com execução) com *misses*, sendo que na codificação PBIW são somados os *misses* da I-*cache* + os da P-*cache* ao comparar com outras codificações. A segunda medida informa o número estimado de ciclos que o programa levaria para executar, considerando cada uma das técnicas de codificação.

Na próxima seção é apresentada toda a infraestrutura utilizada nos experimentos estático e dinâmico, cujos resultados mostram a eficiência da codificação PBIW.

4.2 Infraestrutura para Execução dos Experimentos

Os experimentos estáticos são baseados na análise de informações que podem ser obtidas durante a compilação do programa. Desta forma o conjunto de ferramentas para a análise estática utiliza código intermediário escalonado e com registradores previamente alocados. No caso da avaliação dinâmica, a principal fonte de informação são os *traces* gerados durante a execução do programa.

Como não há implementação em hardware, toda a infraestrutura para execução de experimentos é composta por ferramentas de software. Além do Trimaran e do Dinero [8], que são ferramentas bastante conhecidas na área de arquitetura de computadores, foram desenvolvidas algumas outras ferramentas. A Figura 4.1 mostra o fluxo completo das ferramentas utilizadas nos experimentos e seus respectivos arquivos de entrada e saída. As próximas seções descrevem o funcionamento de cada uma destas ferramentas.

4.2.1 Trimaran

A ferramenta Trimaran foi utilizada para gerar código intermediário a partir de código fonte C e também para obter o *trace* de execução dos programas através de simulação. Conforme pode ser visto na Figura 4.2, o Trimaran é composto pelos seguintes elementos:

- Impact: responsável pela compilação do programa de C para uma linguagem intermediária com otimizações independentes de máquina;
- Elcor: responsável pelas otimizações dependentes de máquina do compilador e onde as fases de escalonamento e alocação de registradores são realizadas;
- Descrição HMDES: responsável por descrever o processador que deverá ser considerado pelas otimizações dependentes de máquina na etapa de simulação;
- Simulador: responsável por realizar a simulação do código a partir de uma descrição de máquina e do código com as otimizações dependentes da máquina descrita. A fase de simulação permite obter um *trace* das operações do programa que foram executadas e estatísticas gerais sobre a execução.

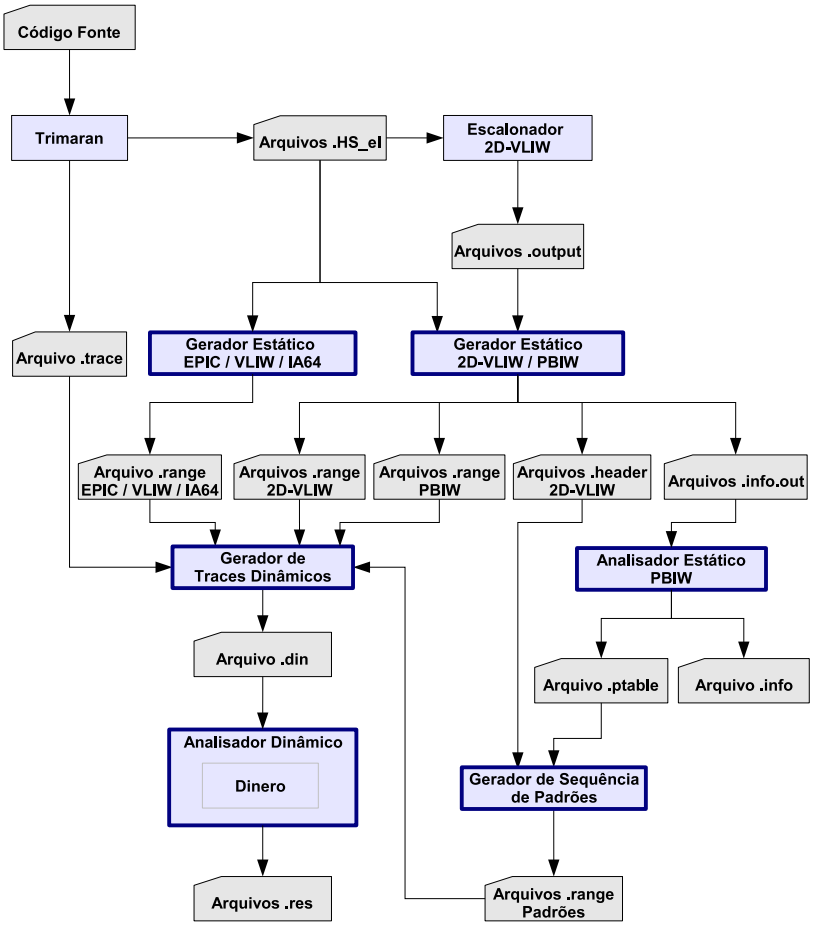


Figura 4.1: Fluxo das ferramentas utilizadas nos experimentos

A entrada para a ferramenta Trimaran é o código fonte de um programa e a saída é composta por dois tipos de arquivos: *.HS_el* e *.trace*. A Figura 4.2 ilustra os componentes da ferramenta Trimaran e os respectivos arquivos de entrada e saída utilizados nos experimentos.

Os arquivos **.HS_el** armazenam código intermediário de todos os procedimentos de um programa. Neste arquivo (Figura 4.3) as operações são representadas através de *opcodes*, operandos, além de informações de *scheduling time* que são utilizadas para determinar as dependências entre as operações. As operações estão organizadas de acordo com a seguinte hierarquia: procedimento → bloco básicos → operações.

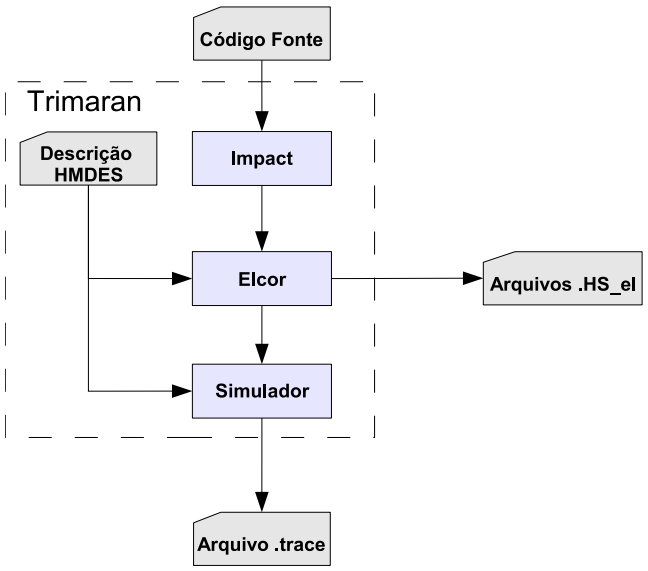


Figura 4.2: Fluxo de execução do Trimaran

Os arquivos **.trace** (Figura 4.4) armazenam o *trace* de execução dos programas, utilizados para a avaliação dinâmica. Nestes arquivos, os procedimentos e blocos são representados exatamente na ordem e número de vezes em que são executados durante o programa. Na Figura 4.4, a letra `p` indica um procedimento e a letra `c`, um bloco. Por exemplo, a segunda linha indica a execução de todas as operações do bloco 8, procedimento `_main`, enquanto que a sétima linha representa a execução das operações do bloco 76, procedimento `_read_min`.

prog.HS_el

```
proc _main 1 (
 bb 2
   op 304 SHL_W [br<1:i gpr 7>] [i<1> br<22:i gpr 8>] s_time(0)
   op 170 BRCT [] [br<55:b btr 2> br<56:p pr 2>] s_time(0)
)

proc _read_min 1 (
 bb 5
   op 20 (PBRR [br<2:b btr 2>] [l:g_abs<_$fn_malloc> i<1>] s_time(0)
)
```

Figura 4.3: Trecho de código um arquivo .HS_el

prog.trace

```
p _main
c 8
c 42
c 30
p _read_min
c 51
c 76
c 17
c 5
c 19
```

Figura 4.4: Trecho de código de um arquivo *.trace*

4.2.2 Escalonador 2D-VLIW

O Escalonador 2D-VLIW é a ferramenta responsável por organizar as operações em hiperblocos de até 16 operações escalonadas. Esta ferramenta recebe como entrada os arquivos .HS_el produzidos pelo Trimaran, realiza o escalonamento segundo algoritmos definidos para a arquitetura 2D-VLIW [36] e gera como saída os arquivos **output**. É importante ressaltar que a descrição HMDES já impõe restrições arquiteturais e que nesta ferramenta estas restrições são apenas seguidas. A Figura 4.5 ilustra o fluxo de execução e os arquivos utilizados e gerados pelo Escalonador 2D-VLIW.

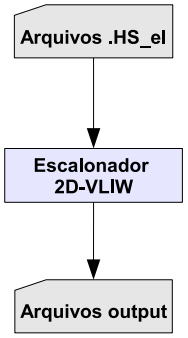


Figura 4.5: Fluxo de execução do Escalonador 2D-VLIW

Os arquivos *output* representam de forma simplificada os hiperblocos de até 16 operações através dos números das operações que são obtidas a partir dos arquivos .HS_el. Também

estão presentes neste arquivo informações de UF indisponível devido às operações que necessitam mais que um ciclo, além de instruções vazias (NOPs), necessárias para resolver problemas de dependência entre as operações.

Como o arquivo *output* contém as operações escalonadas de um determinado bloco (pertencente à um procedimento de um arquivo .HS_el), o nome do arquivo é composto pela palavra *output*, seguida pelo nome do procedimento (*proc*) e pelo número do bloco (*num*), ou seja, output_*proc*_*num*.

A Figura 4.6 representa um trecho de operações escalonadas, referentes ao bloco `34` do procedimento `_read_min`. Na primeira linha, as operações 304, 170, 137, 303 e 82 foram alocadas respectivamente para UF_1, UF_2, UF_3, UF_5 e UF_7, enquanto que as outras UFs estão ociosas (caracter "sublinhado"). Na segunda linha todas as UFs estão ociosas (caracter "sublinhado") com exceção da UF_5 que está ocupada (caracter "menos") devido a uma operação executada anteriormente e que precisa de mais de um ciclo para terminar. Na terceira linha apenas as UF_1 e UF_2 estão em atividade. Na última linha da figura, as UF_1, UF_3 e UF_5 possuem operações alocadas, a UF_7 está ocupada e todas as outras UFs ociosas.

output_read_min_34

```
==========================================
304 170 137 _ 303 _ 82 _ _ _ _ _ _ _ _ _ _
==========================================
_ _ _ _ - _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
==========================================
151 150 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
==========================================
544 _ 3 _ 61 _ - _ _ _ _ _ _ _ _ _ _ _ _ _ _
==========================================
```

Figura 4.6: Trecho de um arquivo *output*

4.2.3 Gerador Estático PBIW

O Gerador Estático PBIW é responsável por complementar os arquivos de saída do Escalonador 2D-VLIW (arquivos *output*) com informações das operações (*opcodes* e operandos)

obtidas nos arquivos de saída do Trimaran (arquivos .HS_el). Para cada arquivo *output*, esta ferramenta procura nos arquivos .HS_el o bloco *num* do procedimento *proc* (o nome do *output* é output_*proc_num*). A Figura 4.7 ilustra o fluxo de execução e os arquivos utilizados e gerados pelo Gerador Estático PBIW.

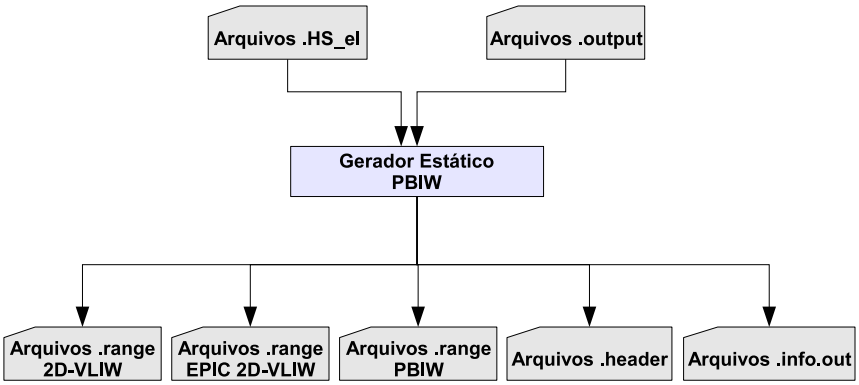


Figura 4.7: Fluxo de execução do Gerador Estático PBIW

Após identificar o bloco *num* dentro do procedimento *proc*, a ferramenta busca pelas operações com números equivalentes aos números presentes no arquivo de saída do Escalonador 2D-VLIW (arquivos *output*) e cria um arquivo output_*proc_num*.info.out, contendo os mesmos números de operações e, além disso, os *opcodes* e operandos de cada operação.

Os arquivos de saída (output_*proc_num*.info.out) do Gerador Estático PBIW representam o código estático completo de um programa escalonado, com registradores alocados e representados em grupos de até 16 operações que podem ser codificadas em uma única instrução PBIW. Para isso, o Gerador Estático PBIW conta com a implementação simplificada (sem os passos referentes à criação/utilização de padrões) do Algoritmo 2 (Codificação) que permite avaliar se um conjunto de operações pode formar uma única instrução PBIW ou se será necessário dividir este conjunto de operações em mais de uma instrução codificada. Desta forma, o número de linhas com grupos de operações pode aumentar com relação às linhas dos arquivos *output* utilizados na entrada da ferramenta. A Figura 4.8 ilustra um trecho de um arquivo .info.out.

Além dos arquivos .info.out, esta ferramenta gera arquivos .header e .range referen-

output_proc_num.info.out

```
================================================================
304 170 137 _ 303 _ 82 _ _ _ _ _ _ _ _ _
op 304 SHL_W [br<1:i gpr 7>] [i<1> br<22:i gpr 8>]
op 170 BRCT [] [br<55:b btr 2> br<56:p pr 2>]
op 137 S_B_C1 [] [br<8:i gpr 5> i<1>]
op 303 S_W_C1 [] [m<int_p2> br<26:i gpr 2>]
op 82 L_H_C1_C1 [br<1:i gpr 3>] [br<31:i gpr 6>]
================================================================
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
================================================================
151 150 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
op 151 PBRR [br<53:b btr 2>] [b<7> i<0>]
op 150 CMPP_W_EQ_UN_UN [br<54:p pr 2> u<>] [br<19:i gpr 7> i<0>]
================================================================
544 _ 3 _ 61 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
op 544 S_B_C1 [] [br<8:i gpr 8> i<21>]
op 3 PBRR [br<57:b btr 2>] [b<7> i<110>]
op 61 L_H_C1_C1 [br<1:i gpr 13>] [br<31:i gpr 27>]
================================================================
```

Figura 4.8: Trecho de um arquivo .info.out

tes à codificação PBIW, que são utilizados na análise dinâmica. Os arquivos **.header** são praticamente iguais aos arquivos *output* porém um pouco mais simplificados (sem os separadores de instruções). O nome de um arquivo .header é representado por output_*proc_num*.header. A Figura 4.9 ilustra um trecho de um arquivo .header.

output_proc_num.header

```
304 170 137 _ 303 _ 82 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
151 150 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
544 _ 3 _ 61 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
```

Figura 4.9: Trecho de um arquivo .header

Os arquivos **.range** são compostos por números seqüenciais onde cada um deles representa uma instrução. Estes números vão sendo incrementados até o fim do processamento de todos os arquivos *output* fazendo com que um determinado número apareça uma única vez e somente em um único arquivo. A Figura 4.10 apresenta o conteúdo do arquivo output_*proc_num*.range, onde o bloco *num* do procedimento *proc* possui apenas quatro

instruções.

Também são gerados arquivos **.range** para 2D-VLIW e para EPIC baseado em 2D-VLIW. Cada instrução recebe um número seqüencial único que é armazenado no arquivo referente ao bloco de um determinado procedimento. Estes arquivos são utilizados na análise dinâmica ao montar o trace de execução do programa.

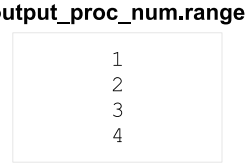


Figura 4.10: Trecho de um arquivo .range

4.2.4 Gerador Estático VLIW / EPIC / IA64

O Gerador Estático VLIW / EPIC / IA64 é responsável por empacotar operações segundo critério de empacotamento da codificação VLIW, EPIC convencional ou IA64 (descritos na Seção 2.1 do Capítulo 2). A escolha de qual codificação deve ser utilizada é realizada via passagem de parâmetro para a ferramenta. As entradas para esta ferramenta são os arquivos de saída do Trimaran (.HS_el).

A partir destes arquivos é gerada uma representação estática do programa (considerando todos os procedimentos do programa) e, a partir desta representação, são geradas as instruções (conjunto de operações empacotadas de acordo com o parâmetro que indica o esquema de codificação) que são representadas apenas por números seqüencias em arquivos .range. Estes arquivos são semelhantes aos gerados pela ferramenta de geração estática PBIW. A Figura 4.11 ilustra o fluxo de execução e os arquivos utilizados e gerados pelo Gerador Estático VLIW / EPIC / IA64.

Não há necessidade de gerar código completo (número da operação, *opcode* e operandos) como é feito na ferramenta para PBIW pois, para estas codificações (EPIC, VLIW e IA64), basta apenas saber o número de instruções geradas. A partir do número de ins-

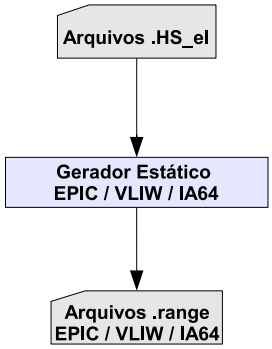


Figura 4.11: Fluxo de execução do Gerador Estático VLIW / EPIC / IA64

truções é possível obter o tamanho do programa e compará-lo ao tamanho do programa obtido pela codificação PBIW.

4.2.5 Analisador Estático PBIW

O Analisador Estático PBIW é a principal ferramenta utilizada nos experimentos estáticos. Esta ferramenta é responsável por gerar instruções PBIW e os respectivos padrões, além de estatísticas de utilização das palavras, número de instruções, número de junções por linha e por coluna, total de padrões após as junções, além de detalhar informações de ocupação de UF e utilização da palavra para cada instrução codificada.

Os Algoritmos 2 (codificação) e 3 (junção de padrões), apresentados no capítulo anterior, são executados por esta ferramenta durante a codificação das instruções. A Figura 4.12 ilustra o fluxo de execução e os arquivos utilizados e gerados pelo Analisador Estático PBIW.

Os arquivos de saída do Gerador Estático PBIW (arquivos .info.out) são utilizados como entrada e uma das saídas é o arquivo **.info** que contém informações sobre a codificação para cada instrução (Figura 4.13), além de totalizações resumidas no fim do arquivo (Figura 4.14).

Para cada instrução, são apresentados os números das operações que a compõe, o número da palavra codificada, os registradores de leitura, os registradores de escrita, o

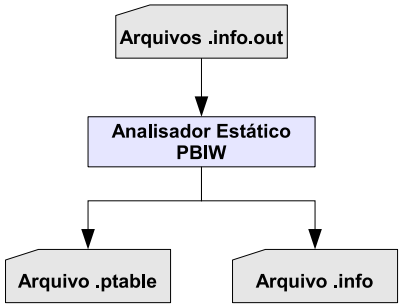


Figura 4.12: Fluxo de execução do Analisador Estático PBIW

pegwit.info

```
============================================================
_ _ 466 _ _ _ 125 893 _ _ _ _ _ _ _ _

Palavra: 10
Leitura: R27
Escrita: R16 R1 R0 R25
Posições de leitura ocupadas: 1/8
Posições de escrita/outros ocupadas: 4/13
Total de posições ocupadas: 5/21
Numero de UFs utilizadas: 3/16
============================================================
```

Figura 4.13: Arquivo .info do programa pegwit - Informações sobre a instrução codificada

número de posições de leitura ocupadas, o número de posições de escrita ocupadas, o total de posições ocupadas e número de UFs que seriam utilizadas para a execução completa da instrução.

Além do arquivo .info, também é gerado um arquivo **.ptable**, que contém o mapeamento de cada instrução para o índice de seu respectivo padrão. Este arquivo é utilizado na geração do *trace* de padrões, que é utilizado na avaliação dinâmica da P-*cache*. Na Figura 4.15 a segunda e a sexta instruções utilizam o mesmo padrão (numerado como 31). A mesma situação acontece entre a terceira e a quarta instrução (padrão 25) e também entre a quinta e a sétima instrução (padrão 28).

pegwit.info

```
================================================================
Total de linhas no programa: 13649
Total de linhas da PCache: 2186
Total de linhas da PCache unidas por linhas: 421
Total de linhas da PCache unidas por colunas: 262
Total de linhas da PCache: 1503
```

Figura 4.14: Arquivo .info do programa pegwit - Resumo da codificação do programa

pegwit.ptable

```
177 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ | 76
378 _ _ _ 379 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ | 31
_ _ _ _ _ _ _ _ _ 178 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ | 25
_ 175 176 181 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ | 25
_ _ 382 _ _ _ 383 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ | 28
_ _ _ _ _ _ _ _ _ 182 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ | 31
_ _ _ 179 180 183 _ _ _ 184 _ _ _ _ _ _ _ _ _ _ _ _ _ _ | 28
```

Figura 4.15: Trecho do arquivo .ptable referente ao programa pegwit

4.2.6 Gerador de Seqüência de Padrões

O Gerador de Seqüência de Padrões é responsável por gerar arquivos .range (semelhantes aos já descritos na Seção 4.2.3) referentes aos padrões. Estes arquivos são utilizados na avaliação dinâmica da P-*cache*. A entrada para esta ferramenta são os arquivos .header (gerado pelo Gerador Estático PBIW - Seção 4.2.3) e o arquivo .ptable (gerado pelo Analisador Estático PBIW - Seção 4.2.5). A Figura 4.16 ilustra o fluxo de execução e os arquivos utilizados e gerados pelo Gerador de Seqüência de Padrões.

Os arquivos .range, neste caso, representam os padrões gerados estaticamente e possuem nomenclatura output_*proc_num*.range, assim como os outros tipos de arquivos .range já apresentados. Dentro destes arquivos existem números seqüenciais e, cada um destes números, representa unicamente um padrão gerado para o programa avaliado. Em outras palavras, dentro do arquivo output_*proc_num*.range estão os números dos padrões utilizados pelas instruções cujas operações pertencem ao bloco *num* do procedimento *proc*.

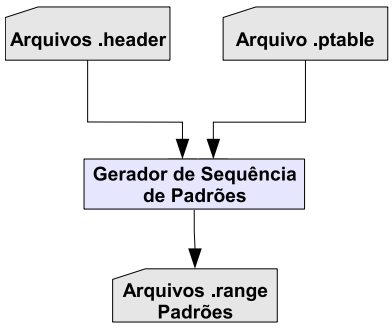


Figura 4.16: Fluxo de execução do Gerador de Seqüência de Padrões

4.2.7 Gerador de Traces Dinâmicos

O Gerador de Traces Dinâmicos é responsável por gerar uma representação resumida da execução de um programa que posteriormente é utilizada como entrada para o analisador dinâmico. A entrada para esta ferramenta são os arquivos .trace (gerados pelo Trimaran) e arquivos .range.

É importante ressaltar que o mesmo arquivo .trace é utilizado para geração de *traces* dinâmicos 2D-VLIW, EPIC baseado em 2D-VLIW, PBIW e também dos Padrões, porém, para cada um destes casos é utilizado um arquivo .range específico, gerado por alguma das ferramentas descritas nas seções anteriores. O arquivo de saída **.din** representa o acesso às instruções na I-*cache* e, no caso dos padrões, o acesso aos padrões armazenados na P-*cache*, durante a execução do programa.

A saída do gerador de *trace* dinâmico é simplesmente uma tradução do *trace* gerado pelo Trimaran para os números equivalentes das instruções codificadas. Ou seja, os procedimentos e blocos representados no arquivo de *trace* de entrada são substituídos pelos números das instruções que compõem estes blocos. A Figura 4.17 ilustra o fluxo de execução e os arquivos utilizados e gerados pelo Gerador de Traces Dinâmicos.

Para cada bloco *num* de um procedimento *proc* encontrado no arquivo .trace, é aberto o arquivo output_*proc*_*num*.range e o conteúdo deste arquivo é concatenado no arquivo de saída (.din). A Figura 4.18 mostra a formação de um arquivo .din, a partir de um *trace* e arquivos .range.

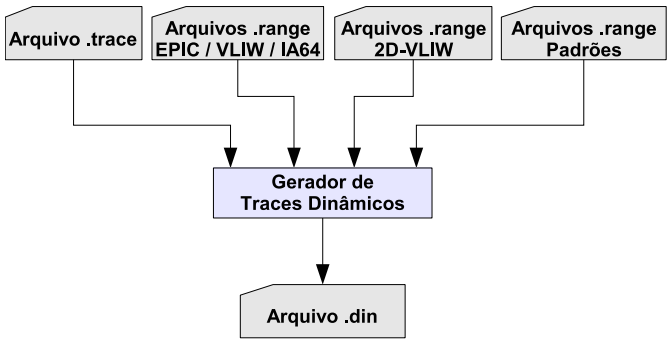


Figura 4.17: Fluxo de execução do Gerador de Traces Dinâmicos

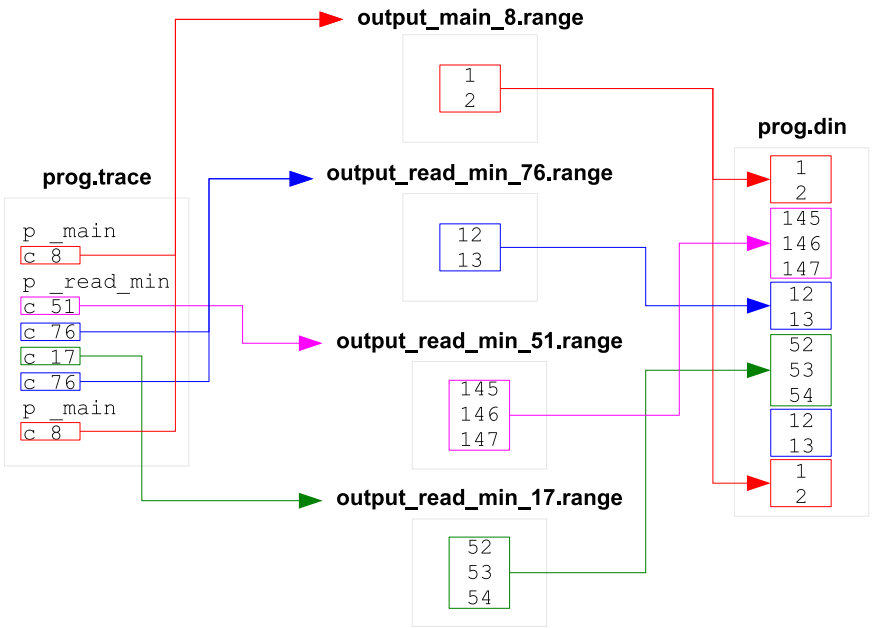


Figura 4.18: Formação do arquivo .din a partir dos arquivos .trace e .range

4.2.8 Analisador Dinâmico

O Analisador Dinâmico é responsável por criar arquivos de entrada para a ferramenta Dinero e executar chamadas para esta ferramenta, variando as configurações da *cache* como associatividade, tamanho e número de palavras por bloco. Este analisador é a principal ferramenta da análise dinâmica e os arquivos de saída gerados contém o número total de acessos à *cache* e número de *misses* para cada configuração de *cache*.

Quando se trata da análise da P-*cache*, a diferença é a execução do Algoritmo 1 (Alocação de Linhas na P-*cache*), utilizado para organizar os dados de forma otimizada na P-*cache*. A Figura 4.19 ilustra o fluxo de execução e os arquivos utilizados e gerados pelo Analisador Dinâmico

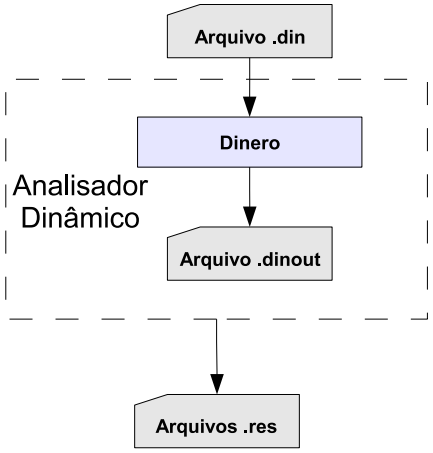


Figura 4.19: Fluxo de execução do Analisador Dinâmico

A entrada para esta ferramenta é o arquivo de saída do Gerador de Traces Dinâmicos (arquivo .din) e a saída é o arquivo **.res** com um resumo do número de *misses* por tamanho de *cache* avaliada. O nome de um arquivo .res, é composto pelo nome do programa (*prog*), associatividade (N) e o número de palavras por bloco (M). A Figura 4.20 mostra o conteúdo de um arquivo .res.

A primeira linha deste arquivo representa o número de acessos à *cache* (*fetches*), que é igual ao número de instruções executadas. Cada uma das outras linhas representa o

prog-*N*-way-*M*-word.res

```
1.184.209.559

  543.697.664
  543.697.664
  474.955.648
  114.399.408
        2.897
        2.897
        2.897
```

Figura 4.20: Exemplo de conteúdo de um arquivo .res

número *misses* que ocorreram para o tamanho de *cache* avaliado. Os valores repetidos nas últimas três linhas do exemplo apresentado na Figura 4.20 indicam que a partir do quinto tamanho de *cache* passaram a ocorrem somente *misses* compulsórios.

Na próxima seção é apresentado o conjunto de experimentos e resultados da avaliação estática, que demonstram a capacidade de redução no tamanho do código, através do reuso de padrões, proposto pela codificação PBIW.

4.3 Avaliação Estática

Parte das ferramentas apresentadas na seção anterior foram desenvolvidas exclusivamente para analisar o comportamento estático dos programas e os ganhos que a codificação PBIW pode oferecer. A avaliação estática mostra o impacto desta técnica sobre o tamanho do programa comparado à outras estratégias de codificação. A codificação das instruções e seus respectivos padrões são obtidos de acordo com as definições apresentadas no capítulo anterior e considerando uma configuração 2D-VLIW com 4 × 4 UFs.

Este experimento foi realizado para comprovar a sobrejeção entre instruções e padrões declarada no capítulo anterior, através da medida do Reuso (descrita na Seção 4.1). Conseqüentemente, a avaliação estática mostra qual o Fator de Redução (também descrito na Seção 4.1) que a codificação PBIW proporciona ao comparar o tamanho dos programas codificados nesta técnica com o de outras técnicas de codificação

A Tabela 4.2 mostra o número de instruções obtidas codificando programas *SPEC* e

MediaBench através dos esquemas de codificação VLIW (I_v), EPIC convencional (derivado de VLIW) (I_{ev}), IA64 (I_i) (as três codificações estão descritas na Seção 2.1 do Capítulo 2), 2D-VLIW (I_{2d}) (descrito na Seção 2.5 do Capítulo 2) e EPIC derivado de 2D-VLIW (I_{e2d}) (descrito no início do Capítulo 3).

As instruções VLIW foram obtidas agrupando operações independentes (paralelas) e utilizando operações vazias (NOPs) quando o número de operações independentes não era suficiente para compor uma instrução completa. As instruções EPIC, baseadas em VLIW, foram obtidas agrupando operações dependentes ou independentes, sendo que as operações dependentes são indicadas por um conjunto separado de bits na instrução. As dependências em uma instrução IA64 são indicadas da mesma forma que nas instruções EPIC. A principal diferença entre EPIC e IA64 é o número de operações que podem compor uma instrução e a quantidade dos bits que indicam as dependências.

As instruções 2D-VLIW foram obtidas através do escalonamento baseado em isomorfismo de subgrafos [36] e as instruções EPIC, baseadas em 2D-VLIW, foram obtidas da mesma forma que as instruções EPIC, baseadas em VLIW. A principal diferença entre os dois modelos EPIC é que o último utiliza operações previamente escalonadas pelo mesmo algoritmo aplicado às instruções 2D-VLIW. Além disso, o número de bits considerado para indicar dependências é de 16 no primeiro modelo e 64 no último.

Pode-se observar na Tabela 4.2 que o número de instruções 2D-VLIW (I_{2d}) é, em média, 43% menor que o número de instruções VLIW tradicionais (I_v). Porém, o número de instruções EPIC derivadas de VLIW (I_{ev}) é, em média, praticamente igual ao número de instruções EPIC derivadas de 2D-VLIW (I_{e2d}). Sobre um outro aspecto, a diferença entre o número de instruções 2D-VLIW e EPIC baseado em 2D-VLIW é muito menor do que a diferença entre VLIW e EPIC baseado em VLIW. Isto se explica pela ineficiência do modelo VLIW ao agrupar operações. Enquanto que a codificação VLIW obriga a criação de uma nova instrução a cada operação dependente, a codificação 2D-VLIW (devido ao seu modelo de execução matricial) permite que algumas operações dependentes sejam agrupadas na mesma instrução escalonando-as de forma que sejam executadas em ciclos diferentes e que seja possível utilizar os resultados entre as operações dependentes, através da comunicação existente entre as UFs na matriz de execução.

Programa	I_v	I_{ev}	I_i	I_{2d}	I_{e2d}
168wupwise	13,27	1,53	7,10	7,66	1,80
175vpr	72,64	10,07	44,35	40,18	10,04
179art	14,44	1,75	8,38	9,07	2,05
181mcf	13,94	1,80	8,15	5,85	1,53
183equake	13,81	1,72	8,08	8,47	2,01
197parser	74,76	11,48	49,83	41,69	11,39
255vortex	121,77	25,45	89,94	79,60	25,92
256bzip2	28,11	3,99	17,82	17,57	4,13
300twolf	116,33	16,13	66,94	66,20	15,34
epic	5,13	1,17	3,52	3,03	1,21
g721decode	3,06	0,50	1,87	1,64	0,53
g721encode	3,02	0,51	1,87	1,65	0,53
gsmdecode	18,30	2,47	9,20	9,91	2,65
gsmencode	24,32	2,97	12,31	13,01	3,20
pegwit	32,85	3,12	14,49	13,64	3,12
Média	37,05	5,64	22,92	21,28	5,70

Tabela 4.2: Número de instruções (× mil) sobre programas *SPEC* e *MediaBench*

A Tabela 4.3 mostra os resultados da estratégia de codificação PBIW, utilizando instruções de 64 bits. As primeiras colunas, I_{2d}, I_{e2d}, I_p, e P, mostram o número de instruções codificadas no formato 2D-VLIW, EPIC baseado em 2D-VLIW, PBIW de 64 bits e padrões para cada programa (após a otimização realizada pelas junções).

A coluna *Reuso* é a taxa de reuso dos padrões P, por parte das instruções PBIW I_p. Em outras palavras, os valores nesta coluna mostram quantas instruções PBIW, em média, utilizam o mesmo padrão. As colunas F_x indicam o fator de redução (%) de um programa que utiliza instruções PBIW quando comparado ao mesmo programa codificado em outras técnicas. Por exemplo, o programa 175vpr codificado com PBIW é 77% menor do que sua versão 2D-VLIW (F_{2d}) e 18% menor do que sua versão EPIC baseado em 2D-VLIW (F_{e2d}).

Os valores das colunas Reuso e Fator de Redução são calculados de acordo com as Equações 4.1 e 4.2, respectivamente.

$$\textbf{Reuso} = \frac{I_p}{P} \tag{4.1}$$

$$\textbf{Fator de Redução} = 1 - \frac{((I_p \times \alpha) + (P \times \beta))}{(I_x \times \gamma)} \tag{4.2}$$

onde:

- α é o tamanho da instrução PBIW. No exemplo considerado neste experimento, cada instrução possui 64 ou 128 bits.
- β é o tamanho de cada padrão de instrução. Cada padrão possui 368 bits (instrução de 64 bits) ou 528 bits (instrução de 128 bits).
- I_x é a quantidade de instruções codificadas por alguma das outras técnicas (2D-VLIW ou EPIC baseado em 2D-VLIW).
- γ é o tamanho da instrução codificada por alguma das outras técnicas (2D-VLIW ou EPIC baseado em 2D-VLIW). As instruções 2D-VLIW utilizam 512 bits para armazenar até 16 operações de 32 bits. Uma instrução EPIC baseada em 2D-VLIW utiliza 576 bits, sendo que 512 são utilizados para armazenar até 16 operações de 32 bits e os outros 64 para indicar dependências entre as operações (Capítulo 3).

Os resultados da Tabela 4.3 confirmam algumas premissas declaradas no capítulo anterior. Os valores da coluna Reuso representam a sobrejeção entre instruções PBIW e seus padrões. Por exemplo, uma sobrejeção de $14,05$ (programa pegwit) indica que, em média, mais de 14 instruções PBIW utilizam o mesmo padrão.

Como já mencionado, a sobrejeção é uma condição necessária para decrementar o tamanho do programa, uma vez que o tamanho total de uma instrução PBIW (instrução codificada + padrão) pode ser maior que o tamanho de uma instrução codificada em outra técnica. Além disso, a sobrejeção reduz as possibilidades de perdas de desempenho devido à freqüência de *misses* simultâneos na *cache* de instruções e de padrões.

A Tabela 4.3 também revela outro resultado interessante: o impacto das restrições arquiteturais (uso restrito de portas de leitura e escrita nos registradores globais) e do

Programa	I_{2d}	I_{e2d}	I_p	P	*Reuso*	F_{2d}	F_{e2d}
168wupwise	7,66	1,80	10,03	0,89	11,29	75	6
175vpr	40,18	10,04	53,32	3,56	14,99	77	18
179art	9,07	2,05	11,71	1,00	11,67	76	5
181mcf	5,85	1,53	7,69	0,80	9,57	74	11
183equake	8,47	2,01	11,11	1,09	10,20	74	4
197parser	41,69	11,39	55,55	3,93	14,15	77	24
255vortex	79,60	25,92	110,41	2,88	38,39	80	46
256bzip2	17,57	4,13	22,13	1,70	13,00	77	14
300twolf	66,20	15,34	78,47	4,14	18,95	81	26
epic	3,03	1,21	4,04	0,57	7,11	70	33
g721decode	1,64	0,53	2,19	0,37	5,91	67	9
g721encode	1,65	0,53	2,20	0,37	6,02	67	10
gsmdecode	9,91	2,65	11,80	0,75	15,75	80	33
gsmencode	13,01	3,20	15,56	1,07	14,48	79	25
pegwit	13,64	3,12	16,90	1,20	14,05	78	15
Média	21,28	5,70	27,54	1,62	13,70	75	19

Tabela 4.3: Número de instruções (× mil) e Fator de Redução (%) sobre programas *SPEC* e *MediaBench* utilizando instruções PBIW de 64 bits

algoritmo de codificação (menor número de operações com imediatos) causou um aumento significativo no número de instruções codificadas PBIW (coluna I_p) quando comparado ao número de instruções codificadas segundo a estratégia 2D-VLIW (coluna I_{2d}).

O número de instruções PBIW de 64 bits aumentou de 19% (programa gsmdecode) à 39% (programa 255vortex), tendo um aumento médio de 29%. Contudo, como pode ser observado pela coluna Fator de Redução, esse aumento não foi significativo a ponto de impedir a redução no tamanho do programa. A codificação PBIW produz programas até 81% (programa 300twolf) menores do que a codificação 2D-VLIW e no pior caso, os programas são 67% menores (programa g721encode), sendo que em média, são 75% menores.

A codificação PBIW (coluna I_p) apresentou um grande aumento no número de instruções quando comparada à EPIC baseado em 2D-VLIW (coluna I_{e2d}), sendo de 234% (programa epic) à 471% (programa 179art), com uma média de 387%. Isto era esperado

pois não existem *NOPs* para resolver dependências em instruções EPIC. Neste modelo de codificação os *stalls* referentes às dependências entre as operações ocorrem durante a execução de acordo com a análise dos bits de template que compõem a instrução. Por outro lado, as instruções PBIW sobre a arquitetura 2D-VLIW utilizam o recurso de instruções *NOP* para que o hardware não precise fazer qualquer avaliação quanto à dependência entre operações.

Essa diferença entre o número de instruções não foi suficiente para a codificação EPIC conseguir programas menores que PBIW. A codificação PBIW de 64 bits produz programas até 46% (programa 255vortex) menores do que a codificação EPIC e no pior caso, os programas são 4% maiores (programa 183equake), sendo que em média, são 19% menores.

A Tabela 4.4 mostra os resultados da estratégia de codificação PBIW, utilizando instruções de 128 bits. As primeiras colunas, I_{2d}, I_{e2d}, I_p, e P, mostram o número de instruções codificadas no formato 2D-VLIW, EPIC baseado em 2D-VLIW, PBIW de 128 bits e padrões para cada programa (após a otimização realizada pelas junções).

Para instruções PBIW de 128 bits, o número de instruções aumentou bem menos em relação às instruções 2D-VLIW. A taxa de aumento foi de 0% (programa gsmdecode) à 3% (programa 168wupwise), tendo um aumento médio de 1%. Porém, da mesma forma como ocorreu com PBIW de 64 bits, esse aumento também não foi suficiente para 2D-VLIW superar PBIW. A codificação PBIW de 128 bits produz programas até 80% (programa 255vortex) menores do que a codificação 2D-VLIW e no pior caso, os programas são 63% menores (programa g721decode), sendo que em média, são 73% menores.

A codificação PBIW (coluna I_p) também apresentou aumento no número de instruções quando comparada à EPIC baseado em 2D-VLIW (coluna I_{e2d}), sendo de 154% (programa epic) à 354% (programa 179art), com uma média de 283%. O motivo é o mesmo já descrito na análise de PBIW com 64 bits.

Da mesma forma como ocorreu para as instruções PBIW com 64 bits, essa diferença entre o número de instruções não foi o bastante para impedir que a codificação PBIW de 128 bits conseguisse produzir programas menores que EPIC baseado em 2D-VLIW na maior parte dos programas. A codificação PBIW de 128 bits produz programas até 45%

Programa	I_{2d}	I_{e2d}	I_p	P	*Reuso*	F_{2d}	F_{e2d}
168wupwise	7,66	1,80	7,90	1,06	11,01	73	-3
175vpr	40,18	10,04	40,63	4,91	14,91	75	10
179art	9,07	2,05	9,32	1,22	11,47	73	-5
181mcf	5,85	1,53	5,90	0,98	9,46	70	-2
183equake	8,47	2,01	8,65	1,30	9,99	71	-7
197parser	41,69	11,39	41,95	5,36	14,08	74	16
255vortex	79,60	25,92	80,12	5,92	37,40	80	45
256bzip2	17,57	4,13	17,71	2,12	12,99	75	5
300twolf	66,20	15,34	66,35	5,52	18,92	79	19
epic	3,03	1,21	3,08	0,67	6,90	65	21
g721decode	1,64	0,53	1,64	0,39	5,94	63	-2
g721encode	1,65	0,53	1,65	0,39	6,02	63	-1
gsmdecode	9,91	2,65	9,91	0,93	15,32	78	26
gsmencode	13,01	3,20	13,02	1,30	14,21	77	18
pegwit	13,64	3,12	13,65	1,50	13,83	76	7
Média	21,28	5,70	21,43	2,24	13,50	73	10

Tabela 4.4: Número de instruções (× mil) e Fator de Redução (%) sobre programas *SPEC* e *MediaBench* utilizando instruções PBIW de 128 bits

(programa 255vortex) menores do que a codificação EPIC baseado em 2D-VLIW e no pior caso, os programas são 7% maiores (programa 183equake), e, em média, os programa são 10% menores.

Conforme já mencionado no início do Capítulo 3 e apresentado na Figura 3.1 deste mesmo capítulo, a codificação EPIC baseada em 2D-VLIW, com 64 bits para indicar dependências, é a forma mais trivial de se representar uma instrução EPIC tendo como referência o modelo de execução 2D-VLIW.

Teoricamente é possível obter codificações com menos bits para indicar dependências mas, utilizar apenas 16 bits seria praticamente inviável pois, esses bits precisam ser suficientes para indicar a posição (uma das 16 UFs) de cada uma das 16 operações na matriz de execução. De qualquer forma, as colunas F_{eN} das Tabelas 4.5 e 4.6 indicam o fator de redução de PBIW sobre EPIC baseado em 2D-VLIW que utiliza N bits para indicar dependência entre as operações. Nesta tabela os valores de N são: 64, 48, 32 e 16.

Programa	F_{e64}	F_{e48}	F_{e32}	F_{e16}
168wupwise	6	4	1	-2
175vpr	18	16	14	11
179art	5	3	0	-3
181mcf	11	8	5	3
183equake	4	1	-2	-5
197parser	24	22	19	17
255vortex	46	44	42	41
256bzip2	14	12	9	6
300twolf	26	24	22	19
epic	33	31	29	27
g721decode	9	7	4	1
g721encode	10	8	5	2
gsmdecode	33	31	29	26
gsmencode	25	22	20	18
pegwit	15	13	10	7
Média	19	16	14	11

Tabela 4.5: Fatores de Redução (%) de PBIW com 64 bits sobre EPIC baseado em 2D-VLIW variando o número de bits de template de EPIC

Os resultados da Tabela 4.5 mostram que mesmo chegando ao limite mínimo teórico, a codificação EPIC baseada em 2D-VLIW não consegue superar a codificação PBIW. A codificação PBIW consegue em média programas 11% menores. Apenas 3 (168wupwise, 179art e 183equake) dos 15 programas avaliados tornam-se maiores em F_{e16}. Porém, conforme já mencionado, é bastante improvável que se consiga uma representação EPIC com apenas 16 bits para indicar a posição das operações dentro da matriz de execução.

A Tabela 4.6 mostra que a codificação PBIW de 128 bits também é mais eficiente do que EPIC com relação ao tamanho dos programas codificados. Mesmo utilizando a quantidade mínima de bits para instruções EPIC, PBIW ainda ganha (2%) considerando a média dos fatores de redução dos programas avaliados.

É possível notar uma pequena vantagem de PBIW com 64 bits sobre a codificação de 128 bits. Enquanto na primeira o número de padrões e o tamanho de cada padrão são menores, na segunda, o número de palavras é menor. Com os dados apresentados

Programa	F_{e64}	F_{e48}	F_{e32}	F_{e16}
168wupwise	-3	-6	-9	-13
175vpr	10	8	5	2
179art	-5	-8	-11	-15
181mcf	-2	-4	-8	-11
183equake	-7	-10	-13	-16
197parser	16	13	11	8
255vortex	45	43	41	40
256bzip2	5	3	0	-3
300twolf	19	17	14	12
epic	21	19	16	14
g721decode	-2	-5	-8	-11
g721encode	-1	-4	-7	-10
gsmdecode	26	24	22	20
gsmencode	18	15	13	10
pegwit	7	5	2	-1
Média	10	7	5	2

Tabela 4.6: Fatores de Redução (%) de PBIW com 128 bits sobre EPIC baseado em 2D-VLIW variando o número de bits de template de EPIC

nesta seção, é possível concluir que, estaticamente, ou seja, considerando o tamanho dos programas, PBIW de 64 bits é mais eficiente do que PBIW de 128 bits.

Na próxima seção são apresentados os experimentos executados durante a avaliação dinâmica. Os resultados comprovam que um *miss* na I-*cache* não implica sempre em um *miss* na P-*cache* e que a combinação I-*cache* + P-*cache* com a codificação PBIW pode ser uma alternativa mais eficiente do que apenas uma I-*cache* utilizada por outras codificações.

4.4 Avaliação Dinâmica

Boa parte dos dados gerados nos experimentos descritos na seção anterior foram utilizados como entrada para as ferramentas de avaliação dinâmica, cujo objetivo é mostrar que a codificação PBIW pode melhorar o desempenho (número estimado de ciclos para execução) dos programas, quando comparado ao desempenho dos mesmos programas codificados

com instruções 2D-VLIW ou EPIC baseado em 2D-VLIW.

Assim como na avaliação estática, a codificação das instruções e seus respectivos padrões são obtidos de acordo com as definições apresentadas no capítulo anterior e considerando uma configuração 2D-VLIW com 4 × 4 UFs.

A avaliação dinâmica visa medir o impacto da utilização de duas *caches* (I-*cache* + P-*cache*) durante a busca e execução das instruções, comparando com a utilização de apenas uma I-*cache* nas outras técnicas de codificação, através da simulação dos programas. Nestes experimentos, foram considerados apenas 10, dos 15 programas da Tabela 4.1. Os programas 181mcf, 255vortex, 300twolf, epic e gsmencode não foram utilizados pois, seus respectivos *traces* gerados pelo Trimaran, não representavam uma execução completa do programa. Ao executar estes cinco programas no Trimaran ocorreram erros na simulação, impedindo que o programa executasse até o final.

Considerando que cada instrução leva 1 ciclo para executar no *pipeline* caso sua busca resulte em *hit*, neste caso não há nenhum *overhead* de tempo, além da própria execução, pois não é necessário acessar a memória. Em outras palavras, o Tempo de Execução do programa poderia ser calculado conforme a Equação 4.3.

$$\textbf{Tempo de Execução} = (\text{Instruções} \times 1) + \textit{Misses}\ \text{Compulsórios} \tag{4.3}$$

Esta situação não é muito usual pois, neste caso, a *cache* provavelmente é maior do que o necessário e, por isso, gera desperdício de espaço no processador. Normalmente, algumas das instruções não estão disponíveis na *cache* durante a execução do programa (*miss*) e portanto precisam ser buscada na memória. Desta forma, este *overhead* (acesso à memória) precisa ser contabilizado no Tempo de Execução do programa.

O valor "5" apresentado na Equação 4.4 foi utilizado nos experimentos envolvendo desempenho de *cache*, descritos em [16] para contabilizar o custo dos *misses*. Esta equação é utilizada como base para o cálculo de custo de *miss* nos experimentos dinâmicos apresentados nesta seção.

$$\textbf{Custo de Miss} = 5 \times \frac{\alpha}{\beta} \tag{4.4}$$

onde:

- α é o tamanho da palavra em memória (em bytes)
- β é o número de bytes por acesso à *cache*

Considerando que nesta fórmula já está sendo contabilizada a própria execução da instrução e que esta gasta apenas 1 ciclo, pode-se então calcular o *overhead* causado por um *miss* conforme a Equação 4.5.

$$\textbf{Overhead por Miss} = \left(5 \times \frac{\alpha}{\beta}\right) - 1 \tag{4.5}$$

Dois parâmetros foram considerados na avaliação dinâmica: *Overhead* Total e Tempo de Execução, os quais podem ser calculados de acordo com as Equações 4.6 e 4.7. O *overhead* total para a estratégia PBIW foi calculado pela soma dos *overheads* da I-*cache* e da P-*cache*.

$$\textbf{Overhead Total} = Misses \times Overhead\ por\ Miss \times Palavras\ por\ Bloco \tag{4.6}$$

$$\textbf{Tempo de Execução} = (\text{Instruções} \times 1) + Overhead\ Total \tag{4.7}$$

A única observação com relação à Equação 4.7 refere-se à forma de utilizá-la no cálculo para as instruções EPIC baseadas em 2D-VLIW. Neste caso, ao invés de utilizar a quantidade de instruções EPIC, deve-se utilizar o respectivo número de instruções 2D-VLIW. Isto é justificado pelo fato de que dentro do *datapath*, o número de ciclos gastos por um processador 2D-VLIW é exatamente igual o número de ciclos gasto por um processador EPIC equivalente.

Todos os experimentos foram realizados com tamanho de *cache* variando de 4KB à 256KB, associatividade variando de mapeamento direto à 4-*way set associative* e, o número de palavras por bloco, variando de 1 a 4.

O tamanho utilizado para palavras EPIC e linhas da P-*cache* foi de 512 bits durante a execução da ferramenta *dinero*, que aceita apenas potências de 2. Porém, ao comparar tamanho de *cache* entre as codificações, foram utilizados os tamanhos reais.

O tamanho da palavra considerado para 2D-VLIW foi de 512 bits e para EPIC, 576 bits. De forma equivalente à análise estática, foram realizadas simulações para palavras PBIW de 64 e 128 bits. Os tamanhos utilizados para uma linha na P-*cache* foram de 368 bits (PBIW 64 bits) e 528 bits (PBIW 128 bits).

Para cada programa, foram avaliadas todas as combinações de tamanho de *cache*, associatividade e número de palavras por bloco. A política de substituição foi LRU em todos os casos e a taxa de transferência variou entre 8 bytes (avaliação de PBIW com 64 bits) e 16 bytes (avaliação de PBIW com 128 bits) por acesso.

Nas avaliações da P-*cache*, foi utilizada uma estratégia de alocação de endereços diferenciada, de acordo com o Algoritmo 1 (Alocação de Linhas na P-*cache*), para minimizar conflitos entre os padrões mais freqüentes. Inicialmente, o endereço de cada padrão foi atribuído em ordem inversa com relação às sua freqüências, ou seja, o padrão mais utilizado recebeu o primeiro endereço, o segundo mais utilizado o segundo endereço e assim sucessivamente. Isto diminui a possibilidade do compartilhamento de uma única linha por padrões muito utilizados. Após alocar um padrão para cada linha, os próximos padrões foram colocados nas linhas onde não existia intersecção de ocorrência (Figura 3.7 da Seção 3.3). Depois disso, caso ainda existissem padrões não atribuídos à linhas, estes eram alocados em ordem seqüencial.

Depois de obter o número de *misses* por tamanho e configuração de *cache* para cada programa, através do Analisador Dinâmico (Seção 4.2.8), foi realizada uma análise para definir a faixa de tamanho válida a ser considerada para cada programa durante as comparações entre as técnicas de codificação.

Foi definido como valor final válido, o tamanho da *cache* onde aparece pela primeira vez apenas *misses* compulsórios, independente da estratégia de codificação. O tamanho inicial considerado deve ser o imediatamente anterior ao tamanho final, dentre os tamanhos avaliados.

As Tabelas de 4.7 à 4.16 apresentam os valores reais de *misses*, que foram utilizados para definir os tamanhos de *caches* considerados e também para calcular o tempo de execução dos programas.

Os valores apresentados nestas tabelas representam o número mínimo de *misses* para cada tamanho de *cache*, considerando os *misses* gerados para cada combinação de associatividade (1-*way*, 2-*way* ou 4-*way*) e quantidade de palavras por bloco (1, 2 ou 4).

A coluna *Cache* representa o tamanho (considerado na execução do *dinero*) da *cache* em que ocorreram os *misses*. As colunas M_{2d}, M_{e2d}, M_{i64}, M_{i128}, M_{p64} e M_{p128} representam, respectivamente, a quantidade de *misses* para as codificações 2D-VLIW, EPIC baseado em 2D-VLIW, I-*cache* PBIW de 64 bits, I-*cache* PBIW de 128 bits, P-*cache* PBIW de 64 bits e P-*cache* PBIW de 128 bits.

Para escolher a faixa considerada na comparação entre PBIW de 64 bits e outras técnicas, foram avaliadas as colunas M_{2d}, M_{e2d}, M_{i64} e M_{p64}. Na comparação com PBIW de 128 bits, foram avaliadas as colunas M_{2d}, M_{e2d}, M_{i128} e M_{p128}.

No programa 168*wupwise* (Tabela 4.7), a faixa escolhida para PBIW de 64 bits e 128 bits foi a mesma, 16KB a 32KB. Neste programa é possível observar que o primeiro tamanho de *cache* onde começa a ocorrer apenas *misses* compulsórios é 32KB.

No caso do programa 179*art* (Tabela 4.9) é possível observar que a faixa para PBIW de 64 bits é de 16KB a 32 KB, enquanto que para PBIW de 128 bits é de 32KB a 64 KB.

Cache	M_{2d}	M_{e2d}	M_{i64}	M_{i128}	M_{p64}	M_{p128}
4	474.986,05	98.957,26	586.875,01	452.798,37	227.115,01	142.320,66
8	445.034,08	98.957,26	529.116,54	449.010,94	143.891,95	116.811,89
16	445.034,08	60.808,74	179.165,42	362.854,37	89.308,89	76.987,18
32	440.678,53	0,53	3,14	2,63	0,53	0,54
64	354.522,05	0,53	3,14	2,34	0,53	0,54
128	2,31	0,53	3,14	2,34	0,53	0,54
256	2,31	0,53	3,14	2,34	0,53	0,54

Tabela 4.7: *Misses* (× mil) por tamanho de *cache* (KB) no programa 168wupwise

$Cache$	M_{2d}	M_{e2d}	M_{i64}	M_{i128}	M_{p64}	M_{p128}
4	3.330,1	536,6	592,7	1.054,1	2.588,2	1.447,6
8	2.337,1	68,2	25,4	195,0	976,9	389,4
16	1.112,6	2,7	7,5	5,8	64,9	3,2
32	154,8	1,3	7,2	5,2	1,8	1,4
64	6,3	1,3	7,2	5,2	1,2	1,2
128	5,1	1,3	7,2	5,2	1,2	1,2
256	5,1	1,3	7,2	5,2	1,2	1,2

Tabela 4.8: $Misses$ ($\times$ mil) por tamanho de *cache* (KB) no programa 175vpr

$Cache$	M_{2d}	M_{e2d}	M_{i64}	M_{i128}	M_{p64}	M_{p128}
4	335.080,26	1.642,52	440,94	115.131,82	43.130,84	26.726,96
8	317.542,27	62,74	331,90	363,43	4.967,99	2.141,86
16	103.334,71	23,91	126,20	206,85	278,47	186,71
32	351,86	10,68	5,78	73,27	2,25	3,53
64	186,87	0,98	5,78	4,58	0,71	0,78
128	66,96	0,98	5,78	4,58	0,71	0,78
256	4,48	0,98	5,78	4,58	0,71	0,78

Tabela 4.9: $Misses$ ($\times$ mil) por tamanho de *cache* (KB) no programa 179art

$Cache$	M_{2d}	M_{e2d}	M_{i64}	M_{i128}	M_{p64}	M_{p128}
4	12.653,66	1.423,44	7.643,54	8.656,68	3.624,64	2.180,00
8	11.670,97	138,33	6,57	3.604,75	1.690,40	869,12
16	8.574,78	1,13	6,50	5,02	99,33	1,02
32	3.462,22	1,12	6,50	4,97	0,88	0,90
64	4,85	1,12	6,50	4,97	0,81	0,87
128	4,85	1,12	6,50	4,97	0,81	0,87
256	4,85	1,12	6,50	4,97	0,81	0,87

Tabela 4.10: $Misses$ ($\times$ mil) por tamanho de *cache* (KB) no programa 183equake

$Cache$	M_{2d}	M_{e2d}	M_{i64}	M_{i128}	M_{p64}	M_{p128}
4	69.797,42	7.330,36	9.329,43	17.267,37	65.122,01	26.334,17
8	34.838,60	1.665,07	2.686,50	2.831,32	19.611,78	10.985,74
16	17.267,37	405,08	175,51	553,91	4.074,53	1.873,50
32	2.831,32	23,35	3,96	17,20	354,95	160,03
64	553,91	0,89	3,96	2,78	0,98	0,95
128	17,20	0,89	3,96	2,78	0,98	0,95
256	2,78	0,89	3,96	2,78	0,98	0,95

Tabela 4.11: $Misses$ ($\times$ mil) por tamanho de *cache* (KB) no programa 197parser

$Cache$	M_{2d}	M_{e2d}	M_{i64}	M_{i128}	M_{p64}	M_{p128}
4	1.479.249,92	102.099,10	107.007,90	1.155.583,87	376.280,64	185.069,50
8	1.407.663,49	7.818,46	10.681,03	73.018,35	55.422,74	29.642,79
16	1.120.167,94	950,71	1.415,35	6.550,75	5.437,96	3.041,65
32	70.804,62	11,26	11,41	399,71	180,86	112,41
64	6.426,01	2,01	11,25	8,99	1,31	1,44
128	380,63	1,91	10,77	8,71	1,21	1,34
256	8,90	1,91	10,77	8,71	1,21	1,34

Tabela 4.12: $Misses$ ($\times$ mil) por tamanho de *cache* (KB) no programa 256bzip2

$Cache$	M_{2d}	M_{e2d}	M_{i64}	M_{i128}	M_{p64}	M_{p128}
4	127.464,14	26.272,01	134.780,90	110.589,62	48.434,65	41.811,36
8	114.650,26	17.168,91	9.564,01	76.153,71	31.344,23	29.153,60
16	110.143,55	0,25	1,32	1,02	0,34	0,33
32	75.565,01	0,25	1,32	1,02	0,29	0,28
64	1,02	0,25	1,32	1,02	0,29	0,28
128	1,02	0,25	1,32	1,02	0,29	0,28
256	1,02	0,25	1,32	1,02	0,29	0,28

Tabela 4.13: $Misses$ ($\times$ mil) por tamanho de *cache* (KB) no programa g721decode

Cache	M_{2d}	M_{e2d}	M_{i64}	M_{i128}	M_{p64}	M_{p128}
4	131.101,32	26.054,09	137.578,94	112.906,82	51.127,92	48.341,05
8	115.761,74	18.438,61	15.878,96	77.478,88	29.200,68	28.942,31
16	112.574,44	0,26	1,35	1,05	0,29	0,29
32	76.889,26	0,26	1,35	1,05	0,28	0,29
64	1,05	0,26	1,35	1,05	0,28	0,29
128	1,05	0,26	1,35	1,05	0,28	0,29
256	1,05	0,26	1,35	1,05	0,28	0,29

Tabela 4.14: *Misses* (× mil) por tamanho de *cache* (KB) no programa g721encode

Cache	M_{2d}	M_{e2d}	M_{i64}	M_{i128}	M_{p64}	M_{p128}
4	206.229,54	24.920,48	122.336,03	205.887,34	10.772,57	6.780,18
8	205.884,86	1.878,39	8.448,17	102.079,22	1.595,05	1.419,86
16	205.870,75	1.528,03	3.941,77	6.353,12	768,72	687,58
32	102.063,29	739,09	176,05	2.798,00	0,59	0,64
64	6.325,44	1,12	4,96	4,10	0,58	0,63
128	2.787,34	1,12	4,96	4,10	0,58	0,63
256	4,10	1,12	4,96	4,10	0,58	0,63

Tabela 4.15: *Misses* (× mil) por tamanho de *cache* (KB) no programa gsmdecode

Cache	M_{2d}	M_{e2d}	M_{i64}	M_{i128}	M_{p64}	M_{p128}
4	47.950,76	3.886,34	23.322,93	27.388,88	8.936,63	8.043,59
8	40.845,58	3.172,41	18.662,50	19.486,82	3.354,88	2.119,40
16	27.367,88	1.916,31	5.308,72	16.088,04	705,92	535,28
32	19.464,66	2,57	12,14	1.340,38	32,54	61,12
64	16.085,05	2,04	11,56	9,42	1,05	1,23
128	1.339,74	2,00	11,56	9,26	1,05	1,21
256	9,41	2,00	11,56	9,26	1,05	1,21

Tabela 4.16: *Misses* (× mil) por tamanho de *cache* (KB) no programa pegwit

A Tabela 4.17 apresenta o número total de *fetches* (ciclos de execução) considerados para cada programa, no cálculo do Tempo de Execução. Conforme já mencionado, o valor considerado para EPIC baseado em 2D-VLIW é o mesmo considerado para 2D-VLIW. As colunas $F_{2d/e2d}$, F_{p64} e F_{p128} representam, respectivamente, os ciclos gastos com execução para a codificação 2D-VLIW (e EPIC baseado em 2D-VLIW), PBIW de 64 bits e PBIW de 128 bits.

Programa	$F_{2d/e2d}$	F_{p64}	F_{p128}
168wupwise	767.594	1.367.931	784.574
175vpr	6.204	9.160	6.425
179art	416.701	530.894	442.402
183equake	15.609	19.016	15.749
197parser	187.459	268.849	187.459
256bzip2	1.836.112	2.209.810	1.852.876
g721decode	198.726	253.857	199.510
g721encode	212.523	271.540	213.466
gsmdecode	261.126	274.381	261.287
pegwit	48.390	54.356	48.404
Média	395.044	525.979	401.215

Tabela 4.17: Número de *fetches* ou ciclos de execução (× mil) considerados no cálculo do Tempo de Execução

Para cada programa, foram escolhidos os tamanhos de P-*cache* e I-*cache* que apresentaram melhores valores de Tempo de Execução, dentro do intervalo de tamanho considerado válido. Como os tamanhos de *cache* não são exatamente iguais entre as codificações devido à diferença no tamanho das palavras, cada configuração PBIW escolhida foi comparada a um tamanho de *cache* menor e também a um tamanho maior nas outras técnicas de codificação.

As Tabelas 4.18 e 4.19 mostram os resultados dinâmicos para cada programa em todas as codificações avaliadas, considerando PBIW de 64 bits. As colunas C_{2d} e C_{e2d} representam os tamanhos das *cache* 2D-VLIW e EPIC baseado em 2D-VLIW. A coluna C_p representa o tamanho da *cache* PBIW. Este valor é a soma das colunas I_p e P_p que representam, respectivamente, os tamanhos da I-*cache* e da P-*cache* PBIW.

As colunas T_{2d}, T_{e2d} e T_p representam o Tempo de Execução, em ciclos, respectivamente para as estratégias de codificação 2D-VLIW, EPIC baseado em 2D-VLIW e PBIW e, as colunas S_{2d}, S_{e2d} representam o *speedup* (Equação 4.8) alcançado pela técnica PBIW com relação a 2D-VLIW e EPIC baseado em 2D-VLIW respectivamente. Os valores apresentados na tabela foram arredondados para melhor visualização, porém os cálculos foram realizados com os valores originais.

$$\textbf{Speedup} = 1 - \frac{T_p}{T_x} \tag{4.8}$$

onde x pode assumir um dos seguintes valores:

- $2d$: Tempo de Execução com instruções 2D-VLIW
- $e2d$: Tempo de Execução com instruções EPIC baseado em 2D-VLIW

Programa	C_{2d}	C_{e2d}	C_p	I_p	P_p	T_{2d}	T_{e2d}	T_p	S_{2d}	S_{e2d}
168wupwise	8,0	9,0	13,8	8,0	5,8	18.124	5.122	3.484	81	32
175vpr	8,0	9,0	10,9	8,0	2,9	97	9	9	90	-1
179art	8,0	9,0	13,8	8,0	5,8	12.801	419	532	96	-27
183equake	8,0	9,0	10,9	8,0	2,9	471	22	19	96	12
197parser	8,0	9,0	13,8	8,0	5,8	1.546	261	280	82	-7
256bzip2	8,0	9,0	13,8	8,0	5,8	56.735	2.180	2.253	96	-3
g721decode	8,0	9,0	10,9	8,0	2,9	4.670	954	292	94	69
g721encode	8,0	9,0	10,9	8,0	2,9	4.727	1.024	335	93	67
gsmdecode	32,0	18,0	34,9	32,0	2,9	4.242	328	275	94	16
pegwit	32,0	18,0	34,9	32,0	2,9	808	133	54	93	59
Média	12,8	10,8	16,8	12,8	4,0	10.422	1.045	753	91	22

Tabela 4.18: Tempo de Execução (milhões de ciclos) e *speedup* (%) considerando *caches* (KB) menores do que a *cache* PBIW e palavras de 64 bits

A Tabela 4.18 mostra que a estratégia de codificação PBIW obteve o melhor desempenho (menor tempo de execução) em todos os casos quando comparada à estratégia de codificação 2D-VLIW. O *speedup* variou de 81% (programa 168wupwise) à 96% (programa 183equake) e a média foi de 91%. As principais razões para este ganho são o tamanho da

instrução (64 bits *versus* 512 bits) e o custo por *miss* (4 ou 28 ciclos *versus* 39 ciclos ou no pior caso 4 + 28 *versus* 39 ciclos).

Da mesma forma, a estratégia PBIW obteve melhor desempenho do que EPIC baseado em 2D-VLIW para a maior parte dos programas. O *speedup* variou de -27% (programa 179art) à 69% (programa g721decode), e a média foi de 22%.

Apesar do número de instruções ser bem maior na técnica PBIW de 64 bits do que em EPIC baseado em 2D-VLIW, o tamanho de uma instrução PBIW é 9× menor (64 *versus* 576 bits) e o reuso de padrões, considerando os 10 programas envolvidos na análise dinâmica, é bastante alto (chegando até a quase 16 instruções, em média, por padrão - programa gsmdecode na Tabela 4.3). Além disso, o custo de *miss* de PBIW (4 ou 28 ciclos e no pior caso 4 + 28 ciclos) é menor do que o custo de um *miss* de EPIC (44 ciclos).

Programa	C_{2d}	C_{e2d}	C_p	I_p	P_p	T_{2d}	T_{e2d}	T_p	S_{2d}	S_{e2d}
168wupwise	16,0	18,0	13,8	8,0	5,8	18.124	3.443	3.484	81	-1
175vpr	16,0	18,0	10,9	8,0	2,9	50	6	9	81	-46
179art	16,0	18,0	13,8	8,0	5,8	4.447	418	532	88	-27
183equake	16,0	18,0	10,9	8,0	2,9	350	16	19	95	-22
197parser	16,0	18,0	13,8	8,0	5,8	861	205	280	68	-36
256bzip2	16,0	18,0	13,8	8,0	5,8	45.523	1.878	2.253	95	-20
g721decode	16,0	18,0	10,9	8,0	2,9	4.494	199	292	94	-47
g721encode	16,0	18,0	10,9	8,0	2,9	4.603	213	335	93	-58
gsmdecode	64,0	36,0	34,9	32,0	2,9	508	294	275	46	6
pegwit	64,0	36,0	34,9	32,0	2,9	676	49	54	92	-12
Média	25,6	21,6	16,8	12,8	4,0	7.963	672	753	83	-26

Tabela 4.19: Tempo de Execução (milhões de ciclos) e *speedup* (%) considerando *caches* (KB) maiores do que a *cache* PBIW e palavras de 64 bits

Na Tabela 4.19 é possível observar que, mesmo comparando com *caches* maiores, a codificação PBIW de 64 bits supera a codificação 2D-VLIW em todos os programas. O *speedup* variou de 46% (programa gsmdecode) à 95% (programa 183equake) e a média foi de 83%.

Ao comparar com *caches* maiores EPIC, a codificação PBIW de 64 bits apresentou melhor desempenho em apenas alguns programas. O *speedup* variou de -58% (programa g721encode) à 6% (programa gsmdecode), e a média foi de -26%.

As Tabelas 4.20 e 4.21 apresentam os resultados dinâmicos para todos os programas em todas as estratégias de codificação avaliadas, considerando palavras PBIW de 128 bits.

Programa	C_{2d}	C_{e2d}	C_p	I_p	P_p	T_{2d}	T_{e2d}	T_p	S_{2d}	S_{e2d}
168wupwise	8,0	9,0	12,1	8,0	4,1	9.223	2.895	2.581	72	11
175vpr	8,0	9,0	12,1	8,0	4,1	51	8	7	86	6
179art	16,0	18,0	24,5	8,0	16,5	2.380	417	444	81	-6
183equake	16,0	18,0	20,1	16,0	4,1	179	16	16	91	-1
197parser	32,0	36,0	36,1	32,0	4,1	241	188	190	21	-1
256bzip2	32,0	36,0	36,1	32,0	4,1	3.181	1.836	1.854	42	-1
g721decode	8,0	9,0	12,1	8,0	4,1	2.377	568	504	79	11
g721encode	8,0	9,0	12,1	8,0	4,1	2.412	609	523	78	14
gsmdecode	16,0	18,0	20,1	16,0	4,1	4.173	294	287	93	2
pegwit	32,0	36,0	36,1	32,0	4,1	418	48	54	87	-11
Média	17,6	19,8	22,2	16,8	5,4	2.464	688	646	73	2

Tabela 4.20: Tempo de Execução (milhões de ciclos) e *speedup* (%) considerando *caches* (KB) menores do que a *cache* PBIW e palavras de 128 bits

A estratégia de codificação PBIW de 128 bits obteve melhor desempenho em todos os casos quando comparada à estratégia de codificação 2D-VLIW, conforme pode ser observado na Tabela 4.20. O *speedup* variou de 21% (programa 197parser) à 93% (programa gsmdecode) e a média foi de 73%. As principais razões para este ganho são o tamanho da instrução (128 bits *versus* 512 bits) e o custo por *miss* que, quando ocorre apenas na I-*cache* (4 ciclos) é bem menor do que o custo de um *miss* na I-*cache* 2D-VLIW. Diferente de PBIW com 64 bits, quando ocorre um *miss* na P-*cache* ou em ambas (I-*cache* e P-*cache*) simultaneamente, o custo de *miss* passa a ser maior (20 ou 4 + 20) do que apenas o *miss* da I-*cache* 2D-VLIW (19 ciclos).

Da mesma forma, a estratégia PBIW obteve melhor desempenho do que EPIC baseado em 2D-VLIW para a maior parte dos programas. O *speedup* variou de -11% (programa

pegwit) à 14% (programa g721encode), e a média foi de 2%.

O número de *fetches* (ciclos de execução) não é muito maior (Tabela 4.17) do que em EPIC baseado em 2D-VLIW, o tamanho da instrução PBIW é $4,5\times$ menor (128 *versus* 576 bits) e o reuso de padrões, considerando os 10 programas envolvidos na análise dinâmica, é bastante alto (chegando até aproximadamente 15 instruções, em média, por padrão - programa gsmdecode na Tabela 4.4). Além disso, o custo de *miss* de PBIW (4 ou 20 ciclos e no pior caso 4 + 20 ciclos) é menor do que o custo de um *miss* de EPIC (22 ciclos) nas situações onde ocorre apenas *miss* na I-*cache* ou na P-*cache*. Todos estes fatores em conjunto justificam o bom desempenho da técnica PBIW de 128 bits, sobre EPIC baseado em 2D-VLIW.

Programa	C_{2d}	C_{e2d}	C_p	I_p	P_p	T_{2d}	T_{e2d}	T_p	S_{2d}	S_{e2d}
168wupwise	16,0	18,0	12,1	8,0	4,1	9.223	2.075	2.581	72	-24
175vpr	16,0	18,0	12,1	8,0	4,1	27	6	7	74	-15
179art	32,0	36,0	24,5	8,0	16,5	423	417	444	-5	-6
183equake	32,0	36,0	20,1	16,0	4,1	81	16	16	81	-1
197parser	64,0	72,0	36,1	32,0	4,1	198	187	190	4	-1
256bzip2	64,0	72,0	36,1	32,0	4,1	1.958	1.836	1.854	5	-1
g721decode	16,0	18,0	12,1	8,0	4,1	2.291	199	504	78	-154
g721encode	16,0	18,0	12,1	8,0	4,1	2.351	213	523	78	-146
gsmdecode	32,0	36,0	20,1	16,0	4,1	2.200	277	287	87	-3
pegwit	64,0	72,0	36,1	32,0	4,1	354	48	54	85	-11
Média	35,2	39,6	22,2	16,8	5,4	1.911	527	646	56	-36

Tabela 4.21: Tempo de Execução (milhões de ciclos) e *speedup* (%) considerando *caches* (KB) maiores do que a *cache* PBIW e palavras de 128 bits

Na Tabela 4.21 é possível observar que, mesmo comparando com *caches* maiores, a codificação PBIW de 128 bits supera a codificação 2D-VLIW em quase todos os programas (a única exceção foi o programa 179art). O *speedup* variou de -5% (programa 179art) à 87% (programa gsmdecode) e a média foi de 56%.

Ao comparar com *caches* maiores EPIC, a codificação PBIW de 128 bits não obteve um bom desempenho. O *speedup* variou de -154% (programa g721decode) à -1% (programa

183equake), e a média foi de -36%.

Conforme já mencionado, os tamanhos de I-*cache* e P-*cache* foram escolhidos respeitando a faixa considerada válida e também onde PBIW obteve melhores resultados. é possível observar nas Tabelas 4.18 e 4.19 que o tamanho mais adequado para P-*cache* é de 2,9KB e para I-*cache*, na maior parte dos programas, o melhor tamanho é de 8KB. Nas Tabelas 4.20 e 4.21 observa-se que para P-*cache*, o tamanho mais utilizado foi de 4,1 KB e para I-*cache* foi de 8KB, mas neste caso existem maior variação de valores do que para 64 bits.

Da mesma forma como foi observado na análise estática na seção anterior, pode-se concluir que dinamicamente, a codificação PBIW de 64 bits supera a de 128 bits. Mesmo tendo menos instruções (*fetches*) para executar (Tabela 4.17) do que a codificação com 64 bits, o número de *misses* para PBIW de 128 bits é maior, na maioria dos casos, do que o número de *misses* para PBIW de 64 bits, conforme pode ser observado nas Tabelas 4.7 à 4.16. Isto é explicado pelo fato de que em uma *cache* com um determinado tamanho, pode-se armazenar o dobro de instruções PBIW de 64 bits com relação a instruções com 128 bits.

Na próxima seção são apresentadas as considerações finais deste capítulo. São destacados os principais resultados e os pontos mais relevantes com relação aos experimentos executados para comprovar a eficiência do esquema de codificação PBIW.

4.5 Considerações Finais

Neste capítulo foram apresentados vários experimentos envolvendo os conceitos e proposições abordados ao longo desta dissertação. Além de mostrar os resultados dos experimentos, procurou-se detalhar a metodologia adotada assim como as ferramentas e critérios utilizados.

As principais medidas utilizadas foram Reuso, Fator de Redução, *Overhead* de *Misses* e Tempo de Execução. As duas primeiras são medidas relativas à análise estática enquanto que as duas últimas são referentes à análise dinâmica.

Dentre todas as ferramentas utilizadas nos experimentos, duas são bastante conhecidas: Trimaran e Dinero. A primeira é utilizada para geração de código intermediário a partir de um código fonte e simulações. A segunda é um simulador de *caches* capaz de informar a quantidade de *misses* baseada em uma determinada configuração de tamanho, associatividade, número de palavras por bloco e também política de substituição. Além destas duas ferramentas foram desenvolvidas várias outras.

O objetivo da análise estática foi comprovar que realmente existe reuso de padrões e que isto implica na redução do código dos programas. Este experimento mostrou que a técnica de codificação PBIW é capaz de reduzir drasticamente o tamanho de um programa. A grande maioria dos programas apresentou redução de código quando comparados a outras técnicas de codificação como 2D-VLIW e EPIC baseado em 2D-VLIW.

A avaliação dinâmica comprovou que boa parte dos programas PBIW tende a ser mais rápida do que quando codificados nas outras técnicas de codificação avaliadas. Contribui com isso o fato de que os *misses* da P-*cache* são independentes dos *misses* da I-*cache*.

O próximo capítulo retoma as motivações e objetivos principais que nortearam o desenvolvimento desta dissertação e apresenta diversas sugestões para o desenvolvimento de trabalhos futuros.

Capítulo 5

Conclusões

Nesta dissertação foi apresentada a técnica PBIW para codificação de instruções. Ao codificar um programa utilizando esta estratégia é possível reduzir o seu tamanho em memória e, portanto, minimizar a latência na busca de instruções.

A estratégia utilizada na codificação PBIW consiste em fatorar as operações de um programa em instruções codificadas e padrões. As instruções codificadas são armazenadas em uma I-*cache* enquanto que os padrões são armazenados na *cache* de padrões, denominada de P-*cache*. Durante o estágio de decodificação, os dados da P-*cache* são utilizados em conjunto com os dados recuperados da memória (I-*cache*) para compor a instrução completa que é utilizada no estágio de execução.

O esquema de codificação PBIW pode ser aplicado em arquiteturas que recuperam instruções grandes da memória como processadores EPIC, arquiteturas reconfiguráveis com várias unidades funcionais e arquiteturas de alto desempenho baseada em matriz de unidades funcionais.

Os resultados do Capítulo 4 mostraram que um programa PBIW pode ser até 69% (média de 22%) mais rápido do que o mesmo programa codificado como EPIC e até 96% (média de 91%) mais rápido do que 2D-VLIW (Tabela 4.18). Também é possível observar reduções de até 46% (média de 19%) no tamanho de código quando a técnica PBIW é comparada à EPIC e até 81% (média de 75%) quando comparada à 2D-VLIW (Tabela 4.3). Além disso, o grande reuso de padrões faz a utilização da P-*cache* uma alternativa viável para minimizar o gargalo na busca de instruções grandes da memória.

5.1 Contribuições desta Dissertação

A principal contribuição desta dissertação é a introdução do esquema de codificação PBIW, baseado na fatoração de padrões, que monta a instrução completa no estágio de decodificação, enquanto realiza a leitura de registradores, excluindo a necessidade de adicionar mais um estágio de pipeline. Além disso foi realizada a comprovação empírica, através de experimentos, dos benefícios relacionados à redução de código e aumento de desempenho ao utilizar a codificação PBIW.

Para apresentar esta nova técnica de codificação foi elaborado o seguinte relatório técnico:

R. Batistella, R. Santos and R. Azevedo. A New Technique for Instruction Encoding in High Performance Architecture. Technical Report IC-07-027. Institute of Computing, State University of Campinas, 2007, Campinas-São Paulo, Brazil.

Além deste trabalho, existe um artigo em elaboração que será submetido ao evento WSCAD 2008.

5.2 Propostas de Trabalhos Futuros

Apesar dos bons resultados com relação à redução de código e melhora de desempenho apresentados no Capítulo 4, existem alguns pontos que podem ser explorados com maior profundidade no sentido de ampliar ainda mais a vantagem da técnica de codificação PBIW sobre outras abordagens e avaliar outras variáveis para comprovar a viabilidade deste modelo. A listagem a seguir fornece algumas sugestões para desenvolvimento de pesquisas futuras:

- Analisar outras formas de junção de padrões;
  A junção de padrões utilizada nos experimentos avalia apenas a intersecção entre linhas e/ou colunas. Poderia ser analisada a possibilidade de junção por colunas onde a soma das colunas utilizadas seja menor ou igual ao total de colunas da matriz de execução. Assim, um dos dois padrões teria alguma de suas colunas deslocadas para a direita ou para a esquerda para que pudesse se unir ao outro padrão.

- Utilizar reuso após junções;
  Após a junção de dois padrões, avaliar se o padrão resultante é exatamente igual a algum outro padrão já existente. Se isso ocorrer, um dos dois podem ser descartados reduzindo o número total de padrões.

- Reordenar de dados para aumentar reuso;
  Uma das premissas para que duas instruções compartilhem o mesmo padrão é que todos os *opcodes* das operações devem ser idênticos entre as duas instruções. Dados 2 padrões com mesmos opcodes, pode-se tentar alterar a ordem dos campos na palavra codificada para provocar a criação de padrões iguais. Isto também ajuda a reduzir o número de padrões.

- Avaliar a possibilidade de representar imediatos com apenas um ponteiro na linha da P-*cache*;
  O objetivo disto é reduzir o tamanho de uma linha da P-*cache* e conseqüentemente melhorar o Fator de Redução dos programas. O único ponteiro na linha da P-*cache* indicaria o campo inicial dos n campos que são necessários para representar um imediato. Junto a este problema está a questão de como utilizar de forma otimizada os campos da palavra para representar o máximo de imediatos e operandos possível.

- Estudar a divisão da P-*cache* em partes ideais para evitar conflitos entre instruções muito utilizadas;
  O Capítulo 3 apresentou um algoritmo para alocação de linhas na P-*cache* com o objetivo de diminuir a concorrência entre as instruções mais utilizadas. Este estudo consiste em conseguir agrupar o maior número de instruções possíveis em uma única linha da P-*cache*, sendo estas instruções livres de intersecção de ocorrência, ou seja, a primeira ocorrência de uma delas só aparece depois da última ocorrência de todas as outras.

- Realizar um estudo com relação à alocação de linhas na P-*cache* baseada em informações estáticas do programa;
  A avaliação dinâmica apresentada nesta dissertação teve por objetivo medir o teto do desempenho da codificação PBIW. Porém, em muitas situações não é viável

utilizar informações dinâmicas para alocar linhas em uma *cache* pois senão seria necessário executar o programa pelo menos uma vez antes de fazer a alocação. Desta forma, ao utilizar informações obtidas em tempo de compilação para minimizar os conflitos entre instruções muito utilizadas, existe uma possibilidade real de adotar tal estratégia. Por exemplo, as instruções dentro de um laço não deveriam utilizar a mesma linha na P-*cache*.

- Verificar a possibilidade de duas P-*caches* sendo uma apenas para *opcodes* e outra para os operandos;
  Isto se justifica pelo fato de que opcodes e operandos não se repetem da mesma forma. Precisa ser verificado se a latência de acesso a mais uma *cache* é compensada pela diminuição do número de *misses* ocasionado pelo maior reuso individual de cada P-*cache*.

- Investigar a viabilidade/adaptação da técnica PBIW para palavras de tamanho variável;
  Todo o estudo da técnica PBIW foi voltado para palavras de tamanho fixo. Seria interessante averiguar o quão esta técnica pode ser útil e/ou que tipo de alterações deveria sofrer para se adequar a palavras de tamanho variável. Por exemplo, poderiam haver mudanças com relação ao formato da palavra, e/ou estrutura da linha na P-*cache* e até mesmo alterações no circuito decodificador.

- Realizar experimentos com palavras de 64 bits e palavras de 128 bits juntas;
  Os experimentos realizados para mostrar a eficiência da técnica PBIW foram realizados considerando todas as palavras com 64 bits ou todas com 128 bits. Apesar de que a maior parte dos grupos de operações escalonados para 2D-VLIW puderam ser codificados com 64 bits, alguns destes grupos de operações precisaram ser divididos em dois ou mais grupos devido à quantidade e diversidade de seus operandos. Este fator gerou um aumento considerável no número de palavras (em relação à codificação com 128 bits), o que pode prejudicar o tempo de execução devido ao maior número de buscas realizadas. Desta forma, para reduzir o número de palavras e não prejudicar tanto o tamanho final do programa pode-se codificar em 64 bits os

grupos de operações que puderem ser codificados com este tamanho e para os que não puderem, utilizar 128 bits.

- Comparar o *overhead* do algoritmo de codificação ao *overhead* da compressão de código em tempo de compilação;
  Este estudo visa obter a comparação entre o tempo gasto durante a codificação PBIW e o tempo gasto com alguma técnica de compressão mais recente. Com estes dados é possível ter mais um parâmetro para avaliar as situações em que a técnica PBIW pode ser mais adequada do que um esquema de compressão.

- Comparar o *overhead* do mecanismo de decodificação ao *overhead* da descompressão de código;
  O objetivo desta comparação é avaliar se realmente o circuito decodificador da técnica PBIW é eficiente e não apresenta desempenho inferior quando comparado à circuitos descompressores de técnicas de compressão de código.

- Avaliar questões de área e consumo de energia;
  A técnica PBIW conta com um decodificador específico capaz de compor uma instrução completa baseado em informações da palavra em memória e da P-*cache*. É muito importante realizar uma avaliação referente ao consumo de energia e área ocupada por este circuito a fim de comparar com outras técnicas, principalmente as de compressão, que utilizam um descompressor para descompactar a instrução no interior do processador.

- Realizar estudo sobre a viabilidade de aplicar técnicas de Reuso de Instruções junto à codificação PBIW;
  Neste estudo o principal objetivo é averiguar se é possível obter algum ganho utilizando estruturas como um *buffer* de reuso, em complemento à codificação PBIW.

- Realizar estudo sobre a aplicação de Agrupamento de Instruções como uma etapa anterior à codificação;
  Este estudo visa avaliar se a eficiência da codificação PBIW aumenta caso as operações iniciais estivessem agrupadas em instruções considerando os operandos temporários.

Antes de executar o algoritmo de codificação, seria aplicado o algoritmo de agrupamento de instruções e então, as instruções agrupadas seriam a entrada para a codificação PBIW.

- Implementar o circuito decodificador PBIW em hardware;
  Mesmo realizando diversos tipos de simulação, não é possível obter dados tão reais quanto aos que poderiam ser obtidos em testes com uma implementação real de um circuito. Através da implementação do circuito decodificador PBIW será possível obter com maior precisão dados como área, consumo de energia e tempo gasto com a decodificação.

Referências Bibliográficas

[1] Andrew Allison. A Brief History of Risc. http://www.aallison.com/history.htm.

[2] G. Araujo, P. Centoducatte, M. Cortes, and R. Pannain. Code Compression Based on Operand Factorization. In *Proceedings of the* 31^{st} *International Symposium on Microarchitecture (MICRO)*, pages 194–201. IEEE Computer Press, 1998.

[3] E. Billo, R. Azevedo, G. Araujo, P. Centoducatte, and E. Wanderley Netto. Design of a decompressor engine on a sparc processor. In *SBCCI '05: Proceedings of the 18th annual symposium on Integrated circuits and system design*, pages 110–114, Florianópolis, SC, Brazil, 2005.

[4] J. Bradford and R. Quong. An empirical study on how program layout affects cache miss rates. *ACM SIGMETRICS Performance Evaluation Review*, 27(3):28–42, May 1999.

[5] L. N. Chakrapani, J. Gyllenhaal, W. Mei, W. Hwu, S. A. Mahlke, K. V. Palem, and R. M. Rabbah. Trimaran - An Infrastructure for Research in Instruction-Level Parallelism. *Lecture Notes in Computer Science*, 3602:32–41, 2004.

[6] D. Citron and D. G. Feitelson. Revisiting Instruction Level Reuse. In *Proceedings of the Workshop on Duplication, Deconstructing and Debunking (WDDD)*, pages 62–70, May 2002.

[7] C. Dulong. The IA-64 Architecture at Work. *IEEE Computer*, 31(7):24–32, July 1998.

[8] J. Edler and M. D. Hill. Dinero IV Trace-Driven Uniprocessor Cache Simulator. [online], 1995. http://www.cs.wisc.edu/∼markhill/DineroIV/.

[9] J. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting. Code Compression. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 358–365. ACM, 1997.

[10] J. A. Fisher. Very Long Instruction Word Architectures and the ELI-512. In *Proceedings of the* 10^{th} *International Symposium on Computer Architecture (ISCA)*, pages 140–150, Los Alamitos, 1983. Computer Society Press.

[11] M. Franz and K. Thomas. Slim binaries. *Communications of the ACM*, 40(2):87–94, 1997.

[12] J. C. Gyllenhaal, W. W. Hwu, and B. R. Rau. HMDES Version 2.0 Specification. Technical Report IMPACT-96-3, Center for Reliable and High-Performance Computing - University of Illinois at Urbana-Champaign, Urbana-Champaign-Illinois, 1996.

[13] J. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millenium. *IEEE Computer*, 33(7):28–35, 2000.

[14] Texas Instruments. TMS320C6000 CPU and Instruction Set Reference Guide. Technical Report SPUR189F, Texas Instruments Inc., October 2000.

[15] Quinn Jacobson and James E. Smith. Instruction pre-processing in trace processors. In *High Performance Computer Architecture (HPCA)*, pages 125–129, 1999.

[16] L. John and R. Radhakrishnan. A Selective Caching Technique. http://citeseer.ist.psu.edu/115017.html.

[17] K. Kissell. MIPS16: High-density MIPS for the Embedded Market. Technical report, Silicon Graphics MIPS Group, 1997.

[18] Michael Kozuch and Andrew Wolfe. Compression of embedded system programs. In *ICCS '94: Proceedings of the1994 IEEE International Conference on Computer Design: VLSI in Computer & Processors*, pages 270–277, Washington, DC, USA, 1994. IEEE Computer Society.

[19] C. Lee, M. Potkonjak, and W. H. Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the* 30^{th} *Annual International Symposium on Microarchitecture (MICRO)*, pages 330–335, Research Triangle Park - North Carolina, 1997. IEEE Computer Society.

[20] Maksim Len and Ilya Vaitsman. VLIW: old architecture of the new generation. http://www.digit-life.com/articles2/vliw.

[21] Advanced RISC Machines Ltd. http://www.arm.com.

[22] Advanced RISC Machines Ltd. An Introduction to Thumb., March 1995.

[23] S. A. McKee. Reflections on the Memory Wall. In *Proceedings of the* 1^{st} *Conference on Computing Frontiers*, pages 162–167. ACM, April 2004.

[24] S. K. Menon and P. Shankar. Space/Time Tradeoffs in Code Compression for the TMS320C62x Processor. Technical Report IISc-CSA-TR-2004-4, Indian Institute of Science, India, 2004.

[25] Carlos Molina, Antonio Gonzalez, and Jordi Tubella. Dynamic removal of redundant computations. In *International Conference on Supercomputing*, pages 474–481, 1999.

[26] Ros Montserrat and Peter Sutton. Compiler Optimization and Ordering Effects on VLIW Code Compression. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 95–103. ACM, November 2003.

[27] S-J. Nam, I-C. Park, and C-H. Kyung. Improving Dictionary-Based Code Compression in VLIW Architectures. *IEICE Transactions on Fundamentals*, E82-A(11):2318–2324, November 1999.

[28] David A. Patterson. Reduced instruction set computers. *Commun. ACM*, 28(1):8–21, 1985.

[29] David A. Patterson and David R. Ditzel. The case for the reduced instruction set computer. *Readings in computer architecture*, pages 135–143, 2000.

[30] R. Phelan. Improving ARM Code Density and Performance, June 2003.

[31] J. Prakash, C. Sandeep, P. Shankar, and Y. N. Srikant. Experiments with a New Dictionary Based Code-Compression Tool on a VLIW Processor. Technical Report IISc-CSA-TR-2004-5, Indian Institute of Science, India, 2004.

[32] S. Prybylski, M. Horowitz, and J. Hennessy. Performance tradeoffs in cache design. In *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture*, pages 290–298, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[33] Stephen E. Richardson. Exploiting Trivial and Redundant Computation. In *Proceedings of the* 11^{st} *Symposium on Computer Arithmetic*, pages 220–227, July 1993.

[34] M. Ros and P. Sutton. Code Compression Based on Operand-Factorization for VLIW Processors. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, pages 559–569. ACM Press, September 2004.

[35] S. Sadasivan. An Introduction to the ARM Cortex-M3 Processor, October 2006. http://www.luminarymicro.com/products/white_papers.html.

[36] R. Santos. *2D-VLIW: Uma Arquitetura de Processador Baseada na Geometria da Computação*. PhD thesis, Unicamp, Brazil, Junho 2007.

[37] R. Santos, R. Azevedo, and G. Araujo. 2D-VLIW: An Architecture Based on the Geometry of the Computation. In *Proceedings of the* 17^{th} *IEEE International Symposium on Application-Specific, Architectures and Processors(ASAP)*, Steamboat Springs - Colorado, 2006. IEEE Computer Society.

[38] R. Santos, R. Azevedo, and G. Araujo. Exploiting Dynamic Reconfiguration Techniques: The 2D-VLIW Approach. In *Proceedings of the* 13^{rd} *IEEE International Reconfigurable Architectures Workshop (RAW)*, Rhodes Island - Greece, 2006. IEEE Computer Society.

[39] R. Santos, R. Azevedo, and G. Araujo. 2D-VLIW: An Architecture Based on the Geometry of the Computation. Technical Report IC-06-06, Unicamp, Brazil, 2007.

[40] Peter G. Sassone and D. Scott Wills. Dynamic strands: Collapsing speculative dependence chains for reducing pipeline communication. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 7–17, Washington, DC, USA, 2004. IEEE Computer Society.

[41] Peter G. Sassone, D. Scott Wills, and Gabriel H. Loh. Static strands: Safely collapsing dependence chains for increasing embedded power efficiency. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 127–136, New York, NY, USA, 2005. ACM Press.

[42] Toshinori Sato and Akihiro Chiyonobu. A preliminary evaluation of timing-speculative instruction collapsing. In *Proc. of 1st Workshop on Introspective Architectures*, 2006.

[43] A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the Memory Wall: The Case for Processor/Memory Integration. In *Proceedings of the* 23^{rd} *International Symposium on Computer Architecture (ISCA)*, pages 90–101, Philadelphia, October 1996. ACM.

[44] M. S. Schlansker and B. R. Rau. EPIC: An Architecture for Instruction-Level Parallel Processors. Technical Report 99-111, Hewlett Packard Laboratories Palo Alto, February 2000.

[45] M. S. Schlansker and B. R. Rau. EPIC: Explicitly Parallel Instruction Computing. *IEEE Computer*, 33(2):37–45, February 2000.

[46] A. Sodani and G. S. Sohi. Dynamic Instruction Reuse. In *Proceedings of the* 24^{th} *International Symposium on Computer Architecture (ISCA)*, pages 194–205. ACM, 1997.

[47] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM SigArch*, 23(1):20–24, March 1995.

[48] Y. Xie, W. Wolf, and H. Lekatsas. A Code Decompression Architecture for VLIW Processors. In *Proceedings of the* 34^{th} *IEEE/ACM International Symposium on Microarchitecture*, pages 66–75. IEEE Computer Press, 2001.

[49] Y. Xie, W. Wolf, and H. Lekatsas. Compression Ratio and Decompression Overhead Tradeoffs in Code Compression for VLIW Architectures. In *Proceedings of the* 4^{th} *International Conference on ASIC*, pages 337–341, October 2001.

[50] Y. Xie, W. Wolf, and H. Lekatsas. Code Compression for VLIW Processors Using Variable-to-Fixed Coding. In *Proceedings of the* 15^{th} *IEEE/ACM International Symposium on System Synthesis*, pages 138–143. IEEE Computer Press, 2002.