NÉLITON ANTÔNIO CAMPOS

UM PROCESSADOR RECONFIGURÁVEL COM JANELA DE INSTRUÇÕES DINÂMICA E BANCO DE REGISTRADORES DISTRIBUÍDO

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

VIÇOSA
MINAS GERAIS – BRASIL
2016

Ficha catalográfica preparada pela Biblioteca Central da Universidade Federal de Viçosa - Câmpus Viçosa

T
C198p
2016

Campos, Néliton Antônio, 1976-
Um processador reconfigurável com janela de instruções dinâmica e banco de registradores distribuído / Néliton Antônio Campos. – Viçosa, MG, 2016.
xiv, 91f. : il. (algumas color.) ; 29 cm.

Inclui anexo.
Inclui apêndice.
Orientador: Ricardo dos Santos Ferreira.
Dissertação (mestrado) - Universidade Federal de Viçosa.
Referências bibliográficas: f.62-64.

1. Microprocessadores. 2. Processamento paralelo (Computação). 3. Banco de registradores. 4. Memória de instruções. I. Universidade Federal de Viçosa. Departamento de Informática. Programa de Pós-graduação em Ciência da Computação. II. Título.

CDD 22. ed. 004.165

NÉLITON ANTÔNIO CAMPOS

UM PROCESSADOR RECONFIGURÁVEL COM JANELA DE INSTRUÇÕES DINÂMICA E BANCO DE REGISTRADORES DISTRIBUÍDO

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

APROVADA: 28 de março de 2016.

José Augusto Miranda Nacif

Carlos de Castro Goulart

Ricardo dos Santos Ferreira
(Orientador)

Este trabalho

dedico

a vocês

que, direta ou indiretamente,

me deram força e esperança

para conduzi-lo.

A vocês,

colegas de curso e de trabalho

que me acompanharam em inúmeros momentos.

A vocês,

meus próximos e familiares

que deram carinho e consideração.

A vocês,

pequeninos João Marcos e Pedro Lucas,

fontes de alegria, carinho e motivação.

A você,

querida Ana Lúcia,

pelo apoio incontestável,

paciência

e boa vontade,

que me faz viver mais intensamente.

E a Você,

Deus,

por nos fazer existir.

"A ciência
se aproxima da verdade progressivamente
quando descreve mais fenômenos
e quando investiga camadas de mecanismos".
(Paul Thagard)

Agradecimentos

Mais que para qualificar simplesmente, a Universidade Federal de Viçosa (UFV), através de seus integrantes, possibilita a mais um funcionário ampliar seus conhecimentos e oportunidades. Agradeço a todos os envolvidos que se propõem a apoiar a instrução e valorização dos servidores nessa instituição e os que estão sempre de prontidão para auxiliar nas mais diversas tarefas.

Conduzir um curso de mestrado em conjunto com atividades laborais e dando apoio à família é um grande desafio. Mas esse desafio se torna mais superável com o apoio de uma condução. Agradeço ao Carlos Antônio Bastos, Chefe da Divisão de Sistemas de Informação da UFV, e à Michelini Lopes da Mota, Diretora da Diretoria de Tecnologia da Informação da UFV, por serem tolerantes e darem condições e apoio para a realização deste trabalho.

Sem uma luz não há como enxergar o caminho. Muito menos os obstáculos que nele tem. Meu agradecimento aos professores do Departamento de Informática da UFV que iluminaram meus passos rumo a um maior conhecimento, tanto técnico quanto pessoal, e sem os quais não seria possível este trabalho. Um agradecimento especial ao meu orientador, Ricardo dos Santos Ferreira, mais que um orientador, um colega inspirador.

Sumário

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ALU	Unidade Lógica e Aritmética (*Arithmetic and Logical Unit*)
BTMS	Escalonamento por Módulo na Tradução Binária (*Binary Translation Modulo Schedule*)
CFG	Grafo de Fluxo de Controle (*Control Flow Graph*)
CGRA	Arquitetura Reconfigurável de Grão Grosso (*Coarse-Grained Reconfigurable Architecture*)
CGRP	Processador Reconfigurável de Grão Grosso (*Coarse-Grained Reconfigurable Processor*)
CJPEG	Compressor [de arquivos de mídia] do Grupo de Especialistas Fotográficos Reunidos (*Compressor [of media files] of Joint Photographics Experts Group*)
Corelap	Processador Largo de Reconfiguração Grossa (*"Coarsed" Reconfigurable Large Processor*)
DFG	Grafo de Fluxo de Dados (*Data Flow Graph*)
EPI	Energia por Instrução (*Energy per Instruction*)
FPGA	Circuito Reconfigurável em Nível de *Bit* (*Field-Programming Gate Array*)
FU	Unidade Funcional (*Functional Unit*)
GOPS	Giga Operações por Segundo (*Giga Operations Per Second*)
GPP	Processador de Propósito Geral (*General Purpose Processor*)
GPU	Unidade de Processamento Gráfico (*Graphic Processing Unit*)
II	Intervalo de Iniciação (*Initiation Interval*)
ILP	Paralelismo em Nível de Instrução (*Instruction Level Parallelism*)
IPC	Instrução Por Ciclo (*Instruction Per Cycle*)
ISA	Arquitetura de Conjunto de Instruções (*Instruction Set Architecture*)

JIT	No Tempo Certo (*Just-In-Time*)
MIN	Rede de Interconexão Multiestágio (*Multistage Interconnection Network*)
MIPS	Microprocessador Sem Estágios Intertravados de *Pipeline* (***M**icroprocessor Without **I**nterlocked **P**ipeline **S**tages*)
MOPS	Mega Operações Por Segundo (*Mega Operations Per Second*)
MS	Escalonamento por Módulo (*Modulo Scheduling*)
NOP	Operação Nula (***No Op**eration*)
NP	Polinomial Não Determinístico (***N**on-Deterministic **P**olynomial*)
PC	Contador de Programa (*Program Counter*)
PCI	Interconector de Componentes Periféricos (*Peripheral Component Interconnect*)
PE	Elemento de Processamento (*Processing Element*)
RISC	Computador de Conjunto Reduzido de Instruções (*Reduced Instruction Set Computer*)
RSIW	Largura de Busca Escalável no Tempo Corrente (***R**un-Time **S**calable **I**ssue **W**idth*)
SIMD	Instrução Única e Dados Múltiplos (*Single Instruction, Multiple Data*)
VLIW	Palavras de Instrução Muito Longas (*Very Long Instruction Word*)
x86	Família de processadores baseados no processador Intel 8086

Lista de símbolos

ρ	Rô
$\equiv$	Equivalente a
$\emptyset$	Variável nula ou conjunto vazio
$\cdots \pmod{\cdots}$	Módulo

Resumo

CAMPOS, Néliton Antônio, M.Sc., Universidade Federal de Viçosa, março de 2016. **Um processador reconfigurável com janela de instruções dinâmica e banco de registradores distribuído**. Orientador: Ricardo dos Santos Ferreira.

A demanda por desempenho computacional é crescente, visto que a variedade das aplicações exige soluções com arquiteturas heterogêneas. A necessidade de conciliar alto desempenho com eficiência energética desafia os desenvolvedores, pelos quais a alternativa mais utilizada é o paralelismo. O presente trabalho é baseado nas CGRAs (*Coarse-Grained Reconfigurable Architectures*) e nos processadores VLIW (*Very Large Instruction Word*), que são arquiteturas paralelas em nível de instrução. Muitos dos processadores VLIW atuais possuem uma janela fixa de instruções, que degrada a utilização da memória. Além disso, o banco de registradores dos processadores VLIW limita o tamanho máximo da janela, afetando a exploração do paralelismo. Este trabalho apresenta uma fusão entre a CGRA e o VLIW em um único processador com janela variável de instruções e registradores distribuídos. A janela variável aproveita os possíveis espaços vazios no final das linhas de memória, melhorando a ocupação; e os registradores e interconexões da CGRA substituem o banco de registradores centralizado de alto custo. O trabalho em questão comprova a viabilidade da proposta com dois estudos de caso. Os resultados das simulações desses exemplos no compilador VEX apresentam um ganho médio de 4,61× em ocupação de memória.

Abstract

CAMPOS, Néliton Antônio, M.Sc., Universidade Federal de Viçosa, March, 2016. **A reconfigurable processor with dynamic instruction window and distributed registers bank**. Adviser: Ricardo dos Santos Ferreira.

The demand for compute performance is increasing, as the variety of applications requires solutions with heterogeneous architectures. The need to combine high performance with energy efficiency challenges developers, in which the most widely used alternative is the parallelism. This work is based on CGRAs (Coarse-Grained Reconfigurable Architectures) and VLIW (Very Large Instruction Word) processors, which are instruction-level parallel architectures. Many of the current VLIW processors have a fixed instruction window, which degrades the memory usage. Additionally, the register file of the VLIW processor limits the maximum size of the window, affecting the parallelism exploitation. This work presents a fusion of the CGRA and the VLIW in a single processor with variable instruction window and distributed registers. The variable window takes advantage from the possible empty spaces at the end of memory lines, improving the occupation; the CGRA registers and interconnections replace the high cost and centralized registers bank. The present work proves the feasibility of the proposal with two case studies. Simulation results of these examples in the VEX compiler have an average gain of 4.61× in memory occupation.

1 Introdução

1.1 Motivação

A demanda por desempenho computacional continua crescendo. As aplicações são heterogêneas, atualmente, e novas soluções computacionais vêm surgindo para atender essa demanda. Um dos maiores desafios dos desenvolvedores é conciliar o alto desempenho com o consumo de energia. A exploração do paralelismo em vários níveis vem sendo utilizada para alcançar o alto desempenho.

Dentre as arquiteturas de processamento paralelo, podemos destacar os processadores superescalares, os processadores de Palavras de Instrução Muito Longas (*Very Long Instruction Word*—VLIW), as Unidades de Processamento Gráfico (*Graphic Processing Units—GPUs*), os circuitos reconfiguráveis em nível de *bits* (*Field-Programming Gate Arrays—FPGAs*) e as Arquiteturas Reconfiguráveis de Grão Grosso (*Coarse-Grained Reconfigurable Architectures—CGRAs*). Cada uma delas tem suas vantagens e desvantagens.

O processador superescalar é a solução mais adotada, visto que segue o padrão da indústria e explora o paralelismo em nível de instruções *assembly* de forma transparente e em tempo de execução. Os compiladores não precisam modificar os códigos binários já que estes têm compatibilidade em nível de *bits*, o que garante a presença dos superescalares no mercado. Porém, o alto consumo de energia e a complexidade dessas arquiteturas são suas maiores limitações.

O processador VLIW transfere a tarefa de detectar e explorar o paralelismo para o compilador, ou seja, faz em *software* para simplificar o *hardware*. Essa transferência simplifica as verificações no nível da arquitetura e pode aumentar a eficiência energética do dispositivo. Porém, a maior parte dos códigos está disponível para superescalares e deve ser recompilada para o VLIW. Outras limitações do VLIW são a janela de instruções ser fixa e o banco de registradores ser pouco escalável.

As GPUs se popularizaram nos últimos anos. Uma GPU é um acelerador que trabalha acoplado ao processador. Ela tem alto desempenho para processamento vetorial com grande volume de dados. Mas uns dos maiores gargalos são o acoplamento fraco ao processador através do barramento Interconector de Componentes Periféricos (*Peripheral Component Interconnect*—PCI) e o modelo de execução baseado na arquitetura Instrução Única e Dados Múltiplos (*Single Instruction, Multiple Data—SIMD*). A arquitetura SIMD executa a mesma instrução para vários dados, o que dificulta a solução de problemas heterogêneos. Os pontos fortes da GPU são o

compilador e a popularização da solução. Existem muitos códigos e problemas já modelados que exploram essa arquitetura extensivamente.

O FPGA é um circuito reconfigurável que aceita programação após a sua fabricação. Consiste em um arranjo de blocos lógicos reprogramáveis e um sistema de interconexão programável. O FPGA pode acomodar uma solução específica para um determinado problema, com exploração do paralelismo. Porém, o tempo para o desenvolvimento do projeto e a falta de ferramentas automatizadas de finalidade geral são seus maiores gargalos.

Uma solução reconfigurável intermediária são as CGRAs. Enquanto é possível reconfigurar os FPGAs em nível de *bits* somente, a CGRA aceita programação em nível de palavras. Não existem soluções comerciais explorando essa arquitetura nos melhores conhecimentos disponíveis apesar da solução da CGRA ser promissora. Os FPGAs (TESSIER; POCEK; DEHON, 2015), as GPUs (KUMAR; JHA; SINGH, 2014), os superescalares (SHEN; LIPASTI, 2013), os VLIWs (IMEC, 2016) e outras arquiteturas já estão presentes no mercado.

A CGRA é dependente do compilador, assim como o VLIW. No entanto, soluções com tradução binária (FERREIRA et al., 2013; FERREIRA et al., 2014) estão disponíveis recentemente para evitar a recompilação do código. É possível configurar a CGRA para executar um ou mais blocos do código, tirando proveito do paralelismo temporal e espacial. Outros trabalhos mostram relações promissoras da CGRA com eficiência energética e desempenho (HAMZEH; SHRIVASTAVA; VRUDHULA, 2013).

1.2 Objetivo

Existem várias possibilidades de arquiteturas paralelas, ou seja, soluções heterogêneas, com vantagens e desvantagens. Este trabalho busca agregar vantagens de duas soluções. Apresenta uma proposta de uma nova arquitetura com escalabilidade o suficiente para suportar um maior número de elementos de processamento e com melhores custos de área de processamento e de memória de instruções, e que mantém ou até mesmo melhora o desempenho computacional frente às arquiteturas VLIW atuais. Trata-se, portanto, de uma fusão de um processador VLIW com um acelerador CGRA.

1.3 Justificativa

Este trabalho busca preencher lacunas presentes em soluções anteriores (HAMZEH; SHRIVASTAVA; VRUDHULA, 2013; MEI et al., 2003; FERREIRA et al.,

2014; JOST, 2014) para compor uma nova proposta. Há a possibilidade de reduzir o custo ao eliminar processadores externos. Os compiladores podem alocar um menor espaço na memória de instruções dos processadores VLIW. Um melhor aproveitamento na codificação dos programas pode reduzir essa exigência de tamanho. Existe ainda a viabilidade de se alcançar desempenho e custo melhores evitando o acesso centralizado aos registradores.

Ao mesclar soluções, este trabalho exibe uma nova arquitetura, mostrando um processador VLIW reconfigurável com janela de execução variável e banco de registradores distribuído, em uma só proposta. Trata-se de um processador CGRA heterogêneo adaptado para realizar as funções de um processador VLIW. A janela de execução variável possibilita um melhor aproveitamento da memória de instruções. E a distribuição dos registradores reduz o custo.

Os compiladores é que devem explorar o Paralelismo em Nível de Instruções (*Instruction Level Parallelism*—ILP) dos códigos, buscando extrair um melhor desempenho. Somente a partir daí a arquitetura irá acelerar a execução dos blocos de código. Os compiladores podem utilizar as técnicas de *software pipelining* e *loop unrolling* (WARTER et al., 1992; MEI et al., 2003) para compor esses blocos. As próximas seções apresentam uma revisão mais detalhada das vantagens e desvantagens do VLIW e da CGRA.

1.3.1 VLIW

Os processadores VLIW (LAM, 1988), como já mencionado, transferem para o compilador a tarefa de extrair o ILP do código fonte. Tais processadores geralmente têm uma janela fixa de execução. Suponha o código com instruções de Computador de Conjunto Reduzido de Instruções (*Reduced Instruction Set Computer*—RISC) da Figura 1a e o código gerado por um compilador VLIW da Figura 1b, com influência da sintaxe do Microprocessador Sem Estágios Intertravados de *Pipeline* (*Microprocessor Without Interlocked Pipeline Stages*—MIPS). As linhas horizontais agrupam instruções RISC em uma mesma instrução, chamada de instrução longa. A arquitetura VLIW executa uma instrução longa em apenas um passo. Suponha nesse exemplo que o processador possa executar até 4 instruções RISC por vez, ou seja, a janela de execução do VLIW tem tamanho 4. O compilador irá reordenar as instruções RISC da Figura 1a sem alterar a semântica do código para extrair o máximo de paralelismo. O compilador pode ainda renomear um ou mais registradores e/ou usar outras técnicas.

Observando a Figura 1a, ocorre a execução de apenas uma instrução RISC por ciclo por todo o código. Cada iteração do laço L_2 (linhas 6–21) desse exemplo contém 8 instruções RISC e se completa após cada 8 passos. Isso representa um valor

```
1  SUM:
2    move r2, r3
3   ------------------------------
4    move r4, -32
5   ------------------------------
6  L2:
7    lw r3, 0(r2)
8   ------------------------------
9    slt b0, r4, r0
10  ------------------------------
11   add r4, r4, 1
12  ------------------------------
13   add r3, r3, 7
14  ------------------------------
15   ble, b0, 0, L3
16  ------------------------------
17   sw 0(r2), r3
18  ------------------------------
19   add r2, r2, 4
20  ------------------------------
21   j L2
22  ------------------------------
23 L3:
24   move r3, r0
25  ------------------------------
26   jr ra
```

(a) Código RISC sem extração de paralelismo.

```
1  SUM:
2    move r2, r3
3    move r4, -32
4   ------------------------------
5  L2:
6    slt b0, r4, r0
7    lw r3, 0(r2)
8    add r4, r4, 1
9   ------------------------------
10   ble b0, 0, L3
11  ------------------------------
12   add r3, r3, 7
13  ------------------------------
14   sw 0(r2), r3
15   add r2, r2, 4
16   j L2
17  ------------------------------
18 L3:
19   jr ra
20   move r3, r0
```

(b) Código VLIW RISC com extração de paralelismo.

Figura 1 – Comparação entre o código RISC e o código VLIW.

de Instruções Por Ciclo (*Instruction Per Cycle*—IPC) de 1 $\frac{[\text{Instrução RISC}]}{[Ciclo]}$. Algumas instruções RISC executam concomitantemente — ou em paralelo — com outras, no caso do código reordenado na Figura 1b; e essa execução ocorre mantendo o mesmo resultado que o código da Figura 1a gera. O processador chega a utilizar até 3 operações simultâneas das 4 possíveis em várias instruções longas na Figura 1b. Cada iteração no laço L_2 da Figura 1b (linhas 5–16), também com 8 instruções RISC, agrupadas em 4 instruções longas (separadas por linhas horizontais), precisa de menos passos dessa vez para completar, de 4 passos. O resultado em IPC é então, nesse caso, de 2 $\frac{[\text{Instruções RISC}]}{[Ciclo]}$. Isso representa um ganho de 2× em IPC em comparação com o laço L_2 do primeiro exemplo. Ganhos nos laços são importantes, porque o tempo que os laços levam para processar domina o tempo total de execução, usualmente.

Alguns outros tipos de arquiteturas obtêm vantagens de ILP extraindo paralelismo em tempo real, diferentemente dos processadores VLIW. Processadores superescalares são exemplos desses tipos de arquitetura. A Figura 2 mostra a comparação das duas abordagens. O superscalar detecta o paralelismo em *hardware* com um custo de complexidade adicional para o processador. O processador VLIW transfere

essa complexidade para o compilador por outro lado.

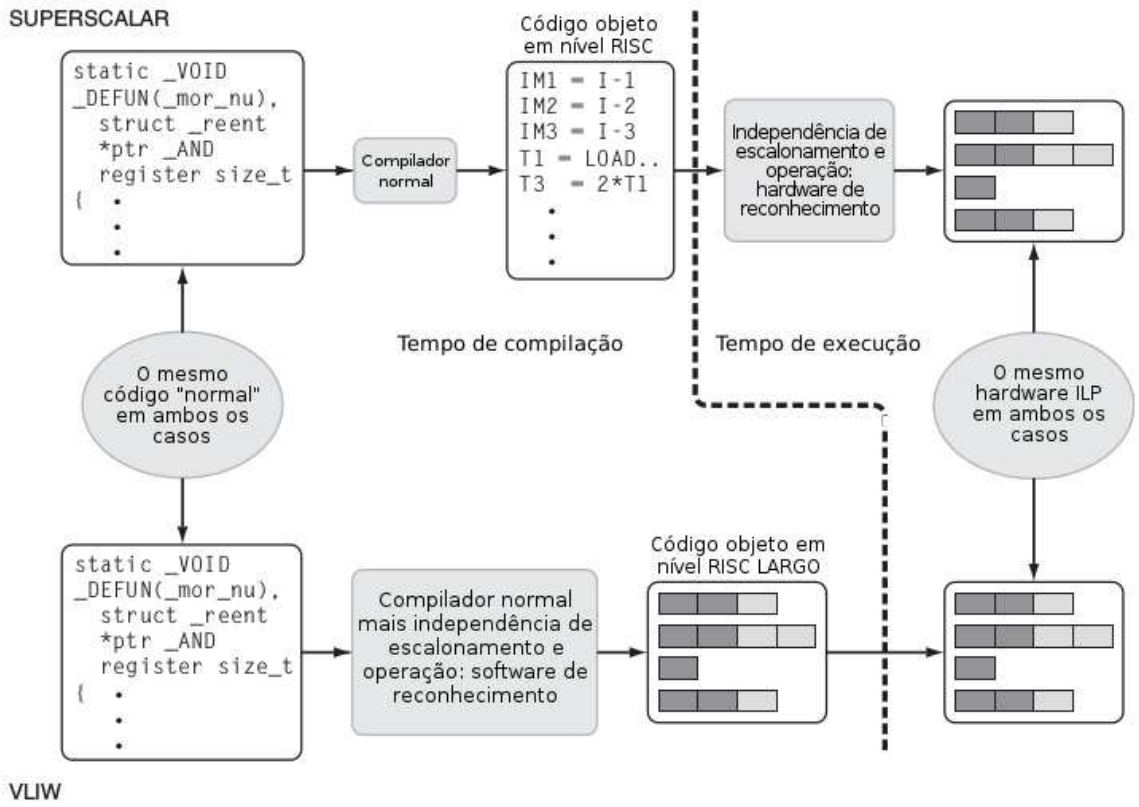


Figura 2 – VLIW *versus* superscalar.

Fonte: Fisher, Faraboschi e Young (2005).

Pode acontecer do código não ter muito paralelismo em vários trechos como ilustra a Figura 3. O compilador não consegue tirar proveito do potencial paralelo da arquitetura mesmo com uma janela de tamanho 4, nesse caso. O laço L_6 (linhas 2–28) desse exemplo da Figura 3 apresenta um menor IPC de $\frac{17}{10} = 1,7 \frac{\text{Instrução RISC}}{Ciclo}$. O código é mais sequencial. Os códigos com mais paralelismo podem aproveitar melhor os processadores de janelas maiores. Mas uma grande janela fixa pode gerar, nos trechos sequenciais, uma subalocação ou um desperdício de memória e de elementos de processamento.

Uma alternativa é aplicar técnicas para minimizar o desperdício de memória que as janelas de busca fixas causam. É possível fazer com que a janela de busca seja de tamanho variável. Os desenvolvedores podem adotar um tamanho de janela diferente para cada aplicação específica. O ρ-VEX (WONG; AS; BROWN, 2008) é um processador VLIW de janela extensível capaz de parametrizar o tamanho da janela de busca, dentre outros aspectos arquiteturais. Wong, As e Brown (2008) apresenta o êxito de configurar uma janela de busca de 1, 2 ou até 4 palavras em tempo de síntese do circuito reconfigurável. Mas ainda é possível atribuir mais flexibilidade na

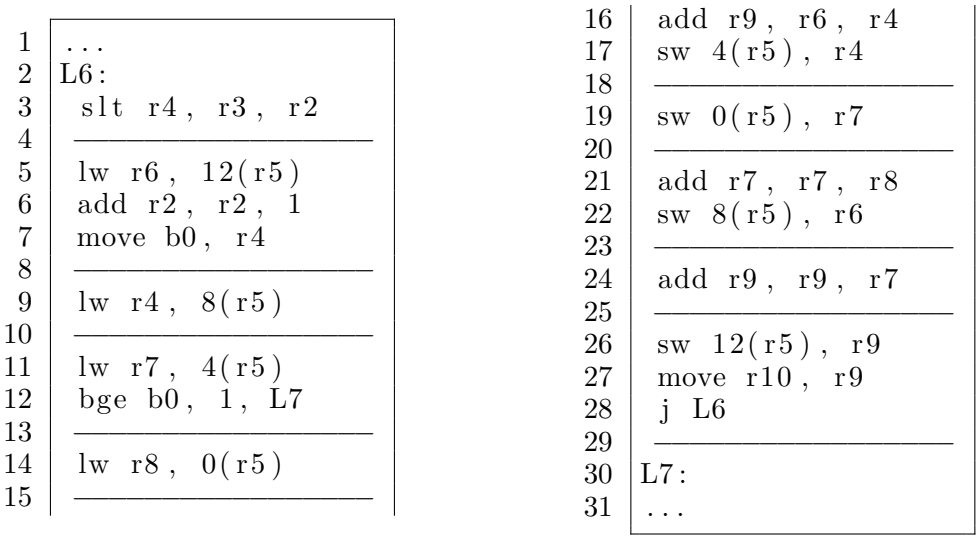


Figura 3 – Código VLIW baseado em RISC de uma série Fibonacci de 4 passos, representando uma baixa extração de paralelismo.

variação de janela para um melhor aproveitamento de memória.

Um banco de registradores com múltiplas portas é um dos princípios de projeto de um processador VLIW (FISHER; FARABOSCHI; YOUNG, 2005). Instruções RISC paralelas realizam acessos concorrentes de leitura e escrita nos registradores. Então, são necessárias portas em quantidade suficiente para permitir vários acessos simultâneos de leitura e escrita. A Figura 4 apresenta um caminho de dados de um processador VLIW genérico de 8 palavras, por exemplo. Nele há um banco de registradores interconectado ao restante da arquitetura por um circuito lógico de transposição. Essa arquitetura precisa de até 8 portas de escrita e de até 16 portas de leitura concorrentes no banco de registradores. Há uma instrução RISC paralela associada a cada Unidade Lógica Aritmética (*Arithmethic and Logical Unit*—ALU) da Figura 4. As ALUs e os dispositivos de leitura e escrita em memória devem então ter acessos independentes e simultâneos a qualquer registrador desse banco de registradores para o ILP ocorrer.

O banco de registradores é um dos recursos mais caros dos processadores VLIW (BENINI et al., 2001) e pode até inviabilizar o projeto, em função do seu tamanho. O processador VLIW reconfigurável ρ-VEX (WONG; AS; BROWN, 2008) alcança apenas o máximo de 8 janelas por causa desse custo, por exemplo.

Uma alternativa é acoplar uma CGRA com uma janela de 16 unidades. Ferreira et al. (2014) e Ferreira et al. (2015) mostram essa viabilidade, em que uma CGRA de 16 unidades funcionais é acoplada a um VEX de 4 palavras. Esses trabalhos apresentam a possibilidade de execução com uma janela maior dos trechos paralelos. No entanto necessitam da CGRA como um circuito externo acoplado ao processador

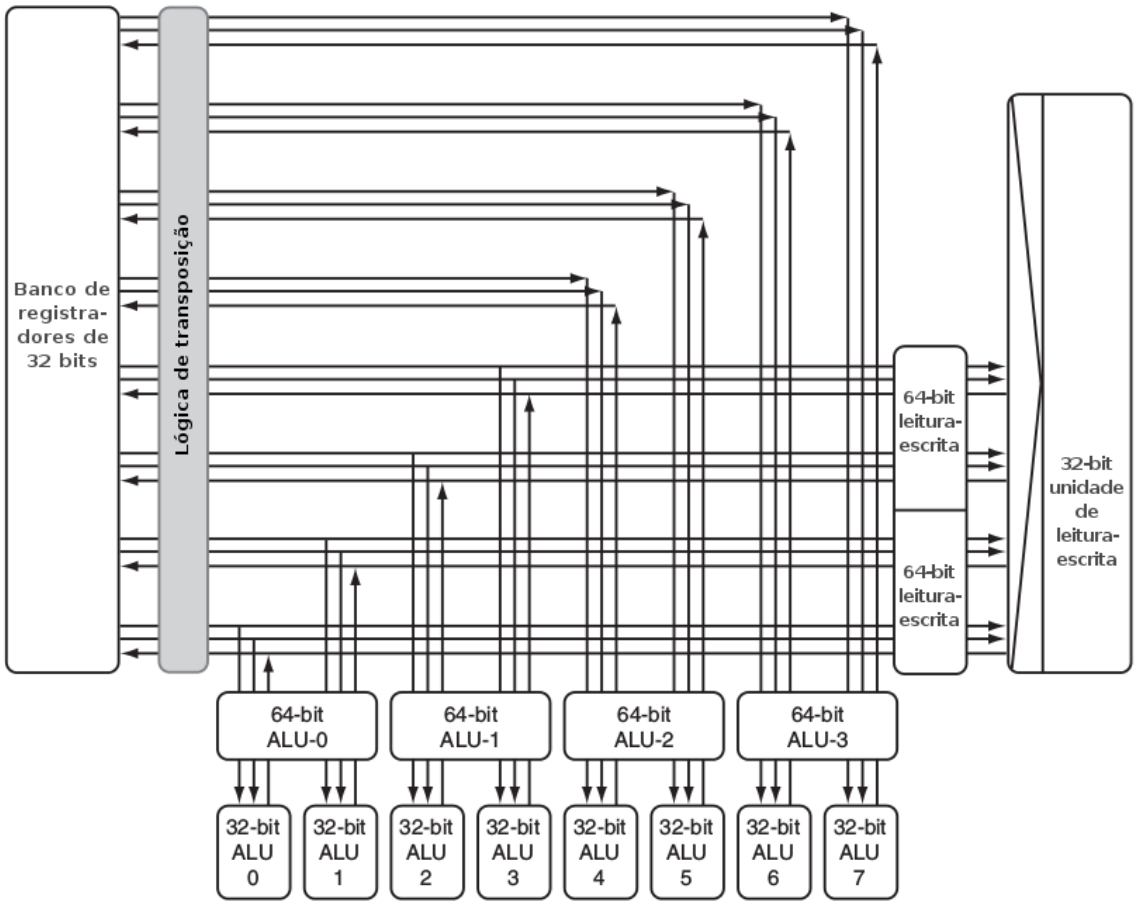


Figura 4 – Caminho de dados de uma arquitetura VLIW genérica de 8 palavras.

Fonte: Fisher, Faraboschi e Young (2005).

VLIW.

1.3.2 CGRA

A CGRA é uma arquitetura de computador reconfigurável capaz de realizar múltiplas operações em palavras ou subpalavras de *bits* (MEI et al., 2003). Computadores reconfiguráveis atribuem um pouco da flexibilidade do *software* ao *hardware* para alcançar desempenho. Uma arquitetura reconfigurável abriga uma variedade de sequências de instruções longas em sua memória de configuração. Um computador reconfigurável pode buscar tais instruções da memória para configurar seu caminho de dados e/ou seu fluxo de controle internos. Ele realiza essa configuração em tempo de execução. Tal configuração de *hardware* em tempo de execução é capaz de produzir ganhos de desempenho computacional.

É importante considerar a granularidade de configuração dos computadores reconfiguráveis. Um computador reconfigurável pode oferecer configuração indo desde um componente computacional inteiro de múltiplos *bytes* até à configuração de um

único *bit* de uma porta lógica, por exemplo. Essa granularidade de configuração depende da concepção de computador reconfigurável sendo abordada. O FPGA é uma concepção de computador reconfigurável capaz de realizar configurações em granularidade fina, de portas lógicas. O FPGA tem muita flexibilidade na configuração.

Cada porta do FPGA requer vários *bits* da memória de configuração, entretanto. O FPGA também necessita configurar o mecanismo de conexão além das portas para interligar essas próprias portas. Isso requer ainda mais *bits* de configuração. O espaço que esses *bits* ocupam provoca uma memória de configuração de alto custo. E ter muitos *bits* para configurar requer um considerável tempo para o processo de geração NP-completo e configuração do dispositivo.

Minimizando o volume de conexões, o tamanho da memória de configuração e o tempo de configuração, e buscando manter o desempenho, a CGRA oferece uma maior granularidade na configuração. Os compiladores precisam configurar componentes em nível de palavras na CGRA, somente. A CGRA requer uma memória de configuração bem menor que o FPGA como resultado. Com isso, a CGRA exige menos tempo de configuração e simplifica o problema da geração ou compilação dos circuitos, apesar de perder em flexibilidade.

Existem muitos tipos de CGRAs. Sua organização contém um arranjo de Unidades Funcionais (*Functional Units*—FUs) ou de Elementos de Processamento (*Processing Elements*—*PEs*), usualmente, que realizam as operações com palavras e subpalavras. A Figura 5 apresenta uma organização típica de uma CGRA, proposta em Hamzeh, Shrivastava e Vrudhula (2013). Trata-se de uma CGRA com um arranjo de 16 PEs.

Cada PE abriga uma FU, na qual as operações lógicas e aritméticas ocorrem de maneira concorrente. A memória de instrução configura os PEs através de um barramento de instruções. Por sua vez, os PEs têm comunicação com a memória de dados por um barramento de dados. A interconexão entre os PEs acontece pelos barramentos específicos que existem entre eles. Essa interconexão torna os compiladores capazes de criar inúmeros caminhos de dados e fluxos de controle entre os próprios PEs. A interconexão segue a topologia do tipo malha, nesse caso, em que cada PE se comunica com seu vizinho.

1.3.3 Coprocessadores CGRA

Alguns trabalhos (MEI et al., 2003; SINGH et al., 2000; CONG et al., 2014; FERREIRA et al., 2013) combinam a CGRA com os Processadores de Propósito Geral (*General Purpose Processors*—GPPs), aproveitando as vantagens em desempenho e

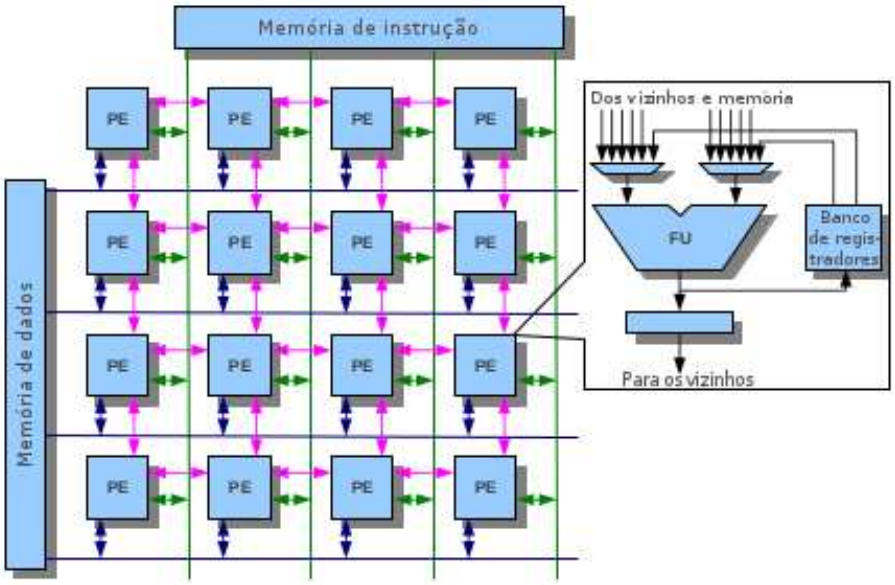


Figura 5 – Uma organização típica de uma CGRA: a arquitetura Regimap.

Fonte: Hamzeh, Shrivastava e Vrudhula (2013).

em variedade de operações. A CGRA é capaz, com essa variedade de processamento, de substituir muitas das operações que um GPP realiza. Os citados trabalhos acoplam, com isso, a CGRA para operar como um coprocessador junto à GPP.

Um coprocessador CGRA atribui maior desempenho a um GPP. Os compiladores paralelizam, geralmente, instruções de partes de execução lenta do código para o coprocessador CGRA executar. O coprocessador CGRA é capaz de capturar e executar o paralelismo melhor que o GPP, gerando ganho em desempenho. Mas o funcionamento com acoplamento requer necessariamente um *hardware* extra para esse coprocessador.

Hardwares extras ocupam mais área e têm o potencial de provocar perda em eficiência energética. Adicionalmente, exceto em alguns exemplos como no paradigma superescalar, os compiladores precisam inserir instruções adicionais para a atuação do coprocessador CGRA. Essas instruções extras geram um custo adicional também.

1.4 Contribuições da dissertação

Este trabalho propõe uma nova organização baseada nas arquiteturas dos processadores VLIW e na CGRA. Primeiro, o compilador gera o código com paralelismo explícito, como os compiladores VLIW. Porém, a janela é de tamanho variável. O formato das instruções é regular para simplificar a codificação e a transferência das informações de controle para a arquitetura. O banco de registradores é distribuído de modo semelhante às CGRAs, entretanto, não é necessário ter o coprocessador

CGRA acoplado ao VLIW. Assim, a proposta desta dissertação é ter uma estrutura única que é um processador VLIW e uma CGRA ao mesmo tempo, se ajustando a cada instrução com uma janela variável.

1.5 Organização da dissertação

Na sequência, o Capítulo 2 contém os trabalhos relacionados que envolvem CGRAs acopladas a processadores e a proposta de uma CGRA operando como um processador VLIW. Possui, ainda, as metodologias de compilação e as estruturas de interconexão das CGRAs.

Detalhes da arquitetura do novo processador estão no Capítulo 3. Esse capítulo mostra a estrutura de interconexão, o funcionamento e alocação dos registradores distribuídos; apresenta o formato das instruções também; detalha o mecanismo de busca de instruções com a janela variável; apresenta, posteriormente, o modelo de compilação baseado na alocação dos registradores destino, que permite uma nova abordagem de renomeação e simplifica um dos maiores gargalos dos processadores VLIW ao substituir o banco centralizado com múltiplas escritas por um banco de registradores distribuído.

O Capítulo 4 apresenta um estudo de caso para validar a nova arquitetura e uma mostra do potencial da solução proposta. O Capítulo 5 expõe os resultados de simulações retirados do trabalho. E, por último, há a descrição das principais conclusões e sugestões de trabalhos futuros.

2 Revisão de literatura

2.1 Introdução

Este capítulo aborda os trabalhos correlatos. Como já mencionado, essa dissertação apresenta uma nova organização para o processador, que é derivada dos processadores VLIW e das CGRAs. Apresenta esses dois temas inicialmente. E mostra uma proposta de fusão entre as arquiteturas VLIW e CGRA.

2.2 VLIW

Um processador VLIW dispara uma janela de N instruções para executá-las em paralelo a cada ciclo. O processador não verifica se tem dependência de dados entre as instruções. O compilador é o responsável por essa tarefa de escalonar o código e resolver os conflitos de dados para explorar ao máximo o paralelismo (FISHER; FARABOSCHI; YOUNG, 2005). Esta seção aborda três pontos: banco de registradores, janela de instruções e simulação e compilação.

2.2.1 Banco de registradores

Um dos problemas do processador VLIW é o consumo em área e energia do banco de registradores (BENINI et al., 2001). Trabalhos recentes (WONG; ANJAM; NADEEM, 2010) para síntese de VLIW reconfiguráveis mostram que é possível ter um banco de 4 ou 8 acessos simultâneos. Porém, com 16, é inviável. Ferreira et al. (2013) apresenta a possibilidade de acoplar um acelerador com 16 ou mais unidades ao processador, no caso de códigos com mais paralelismo. No entanto, existem os custos do acelerador, da recompilação e da tradução binária (FERREIRA et al., 2014) adicionados ao custo do processador VLIW.

2.2.2 Janela de instruções

A maioria dos VLIW tem uma janela de N instruções RISC (FISHER; FARABOSCHI; YOUNG, 2005). Alguns processadores usam uma técnica de compactar o código para reduzir o espaço ocupado pelas instruções RISC de Operação Nula (*No OPeration*—NOP) (LIN; XIE; WOLF, 2004) já que nem sempre é possível extrair N instruções.

Alguns processadores têm uma janela dinâmica como o ρ-VEX (WONG; AS; BROWN, 2008; KOENIG et al., 2011; NAGARAJAN et al., 2004). A janela dinâmica

do ρ-VEX permite que um código compilado para uma janela de tamanho 2 execute com tamanho 1, 4 ou 8 (WONG; AS; BROWN, 2008). O processador pode ajustar, entre esses tamanhos, a largura da janela em função do paralelismo que existe em cada trecho do programa.

Outros trabalhos acoplam uma CGRA para executar código com janelas maiores. Exemplos são a arquitetura Edge (LIN; XIE; WOLF, 2004), a arquitetura do Adres (MEI et al., 2003) e a arquitetura de Largura de Busca Escalável no Tempo Corrente (*Run-Time Scalable Issue Width*—RSIW) (KOENIG et al., 2011). Também com janelas maiores há a CGRA virtual proposto em Ferreira et al. (2013) com compilação No Tempo Certo (*Just-In-Time*—JIT) ou tradução binária (FERREIRA et al., 2014).

2.2.3 Simulação e compilação

Esta dissertação adota arquitetura VEX (FISHER; FARABOSCHI; YOUNG, 2005) para a simulação e compilação. A VEX tem um compilador configurável, além da descrição da arquitetura. Tal compilador pode emular de forma eficiente a execução de um processador VLIW em processadores x86. A ferramenta também oferece recursos de *profile* e acesso aos valores tanto dos registradores quanto da memória.

Um arquivo de configuração define o tamanho da janela, sua composição — que pode ser heterogênea — e as latências das instruções, como ilustra a Figura 6. Nesse exemplo, as linhas 1–7 representam a quantidade de recursos que a arquitetura possui. As linhas 8–10 dizem ao simulador VEX qual é a latência da operação indicada. E as linhas 11–12 indicam a quantidade disponível de registradores.

```
1  RES: IssueWidth 16
2  RES: MemLoad 2
3  RES: MemStore 2
4  RES: IssueWidth.0 16
5  RES: Alu.0 16
6  RES: Mpy.0 4
7  RES: Memory.0 2
8  DEL: Multiply.0 1
9  DEL: Load.0 1
10 DEL: Store.0 0
11 REG: $r0 16
12 REG: $b0 8
```

Figura 6 – Arquivo de configuração do simulador VEX de 16 palavras e 2 acessos simultâneos à memória.

Um arquivo mostra os principais resultados (Figura 7), de forma resumida, após a execução da simulação. É possível obter o número de ciclos que a arquitetura leva para executar o código, dentre outras informações, como mostra as linhas 1–4 da Figura 7. Esse número já desconsidera os ciclos provenientes de interrupções que o processador x86 despende. O número de ciclos é importante para verificar o desempenho do código.

```
1  Total Cycles:                                    34087
     ↪ (0.068174 msec)
2  Execution Cycles:                                22008 (
     ↪  64.56%)
3  Stall Cycles:                                    12079 (
     ↪  35.44%)
4  Nops:                                             3377 (
     ↪   9.91%)
5  Executed operations:                             31113
6
7  Executed branches:                                4562 (
     ↪  14.66% ops)(20.73% insts)
8  Not taken branches:                               1025 (
     ↪   3.29% ops)( 4.66% insts)(22.47% br)
9  Taken branches:                                   3537 (
     ↪  11.37% ops)(16.07% insts)(77.53% br)
10   Taken uncond branches:                          1931 (
       ↪   6.21% ops)( 8.77% insts)(42.33% br)
11   Taken cond branches:                            1606 (
       ↪   5.16% ops)( 7.30% insts)(35.20% br)
12 Size of Loaded Code:                              28096
     ↪ Bytes
13
14 ...
15
16 Percentage Bus Bandwidth Consumed:                24.55%
17
18 Avg. IPC (no stalls):   1.41
19 Avg. IPC (with stalls): 0.91
20
21 ...
22
23 Simulation time  =     0.0105 s
24 Simulation speed =     2.9595 MOPS
25
26 ...
```

Figura 7 – Parte do conteúdo do arquivo de resultado de simulação do compilador VEX.

O arquivo resultante da simulação apresenta outras medidas de desempenho também, como o número de instruções executadas, na linha 5, e o número de IPC, nas linhas 18–19. Esse arquivo apresenta tempo e velocidade de execução também, como exemplifica as linhas 23–24.

2.3 CGRA

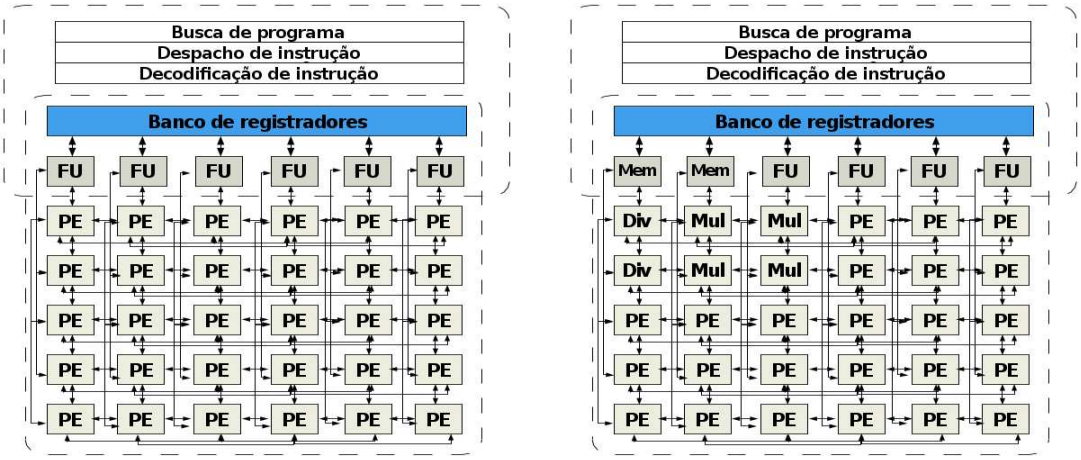
As CGRAs podem ser bem eficientes em termos enérgicos quando comparadas a processadores superescalares. Mei et al. (2003) mostra que as CGRAs possuem uma eficiência energética, medida em Energia Por Instrução (*Energy Per Instruction*— EPI), de 24 pW/Instrução. Isso com um desempenho operacional aproximado, medida em IPC, de 3,3 GOPS. Por contraste, segundo o trabalho, o processador Intel Core2 possui uma EPI de 6 nW/Instrução, com uma IPC de 5,2 GOPS, considerando suas duas janelas de instruções. As CGRAs possuem, assim, um ganho máximo de 250× em eficiência energética, que ocorre nos trechos de código em que os compiladores conseguem preencher todas as janelas das arquiteturas.

Alguns trabalhos como Adres (MEI et al., 2003), Trips (SANKARALINGAM et al., 2003) e Morphosys (LEE et al., 2000) propõem diversas CGRAs. Eles utilizam as CGRAs na busca de reduzir o *overhead* de comunicação, reduzir dificuldades de programação e alcançar desempenho ao combinar *hardwares* reconfiguráveis com GPPs. A CGRA, no Adres e no Morphosys, atua como um acelerador ILP acoplado a um processador central. E, no Trips, a CGRA age como um próprio processador VLIW. Hartenstein (2001) apresenta uma revisão detalhada do histórico e das características das CGRAs.

As CGRAs, em geral, têm uma arquitetura em malha bidimensional, como o Adres (MEI et al., 2005), ilustrado na Figura 8, e o Regimap (HAMZEH; SHRIVASTAVA; VRUDHULA, 2013), ilustrado na Figura 5. A CGRA é um conjunto de M×N PEs ou FUs. Cada PE possui uma FU. O PE pode ter registradores nas entradas e na saída ou mesmo um banco de registradores local (Figura 5). A CGRA pode ser homogênea (Figura 8a), na qual todos os PEs fazem todas as operações. Ou pode ser heterogênea (Figura 8b), a partir da qual algumas operações, por exemplo, fazem apenas a multiplicação, outras operam sobre a memória e ainda outras realizam operações lógicas e aritméticas.

Outro ponto relevante é a conexão com o banco de registradores do processador — acoplamento forte — ou a comunicação através da memória — acoplamento fraco. O número máximo de operações simultâneas com a memória é um ponto de destaque, assim como a estrutura de interconexão entre os PEs. Algumas CGRAs só se comunicam com os vizinhos diretos (Figura 5), outras podem ter acesso a todos os PEs de uma linha ou coluna. Quanto maior a conectividade, maior será o custo da CGRA.

Entretanto, CGRAs com malha bidimensional dificultam o trabalho do compilador devido à complexidade do posicionamento e roteamento das instruções. Ferreira et al. (2011), um trabalho recente, introduz uma arquitetura baseada em rede global,



(a) Homogênea (b) Heterogênea

Figura 8 – Caminho de dados da arquitetura Adres, em configuração homogênea e heterogênea.

Fonte: Mei et al. (2005) (Figura a).

o que simplifica o processo. Tal rede global permite a incorporação de técnicas dinâmicas, seja com compilação JIT (FERREIRA et al., 2013) ou tradução binária (FERREIRA et al., 2014).

A Figura 9 mostra a arquitetura CGRA com rede global, que consiste de duas Redes de Interconexão Multiestágio (*Multistage Interconnection Networks*—MINs) em paralelo. Nelas, cada FU acessa, em um ciclo, qualquer outra FU. Ferreira et al. (2011) propõe a rede global com custo de $O(n \log n)$ e latência máxima de $2\log(n) - 1$ estágios. É uma rede reduzida comparada a propostas similares (FERREIRA et al., 2011).

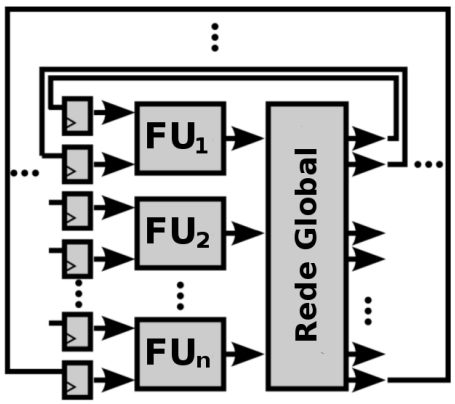
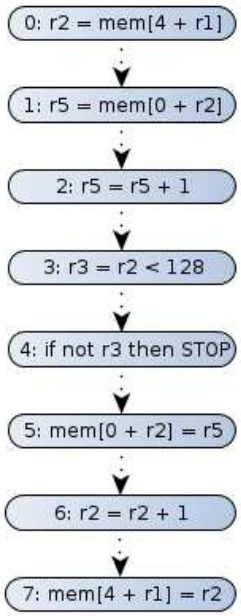


Figura 9 – CGRA com rede de interconexão global.

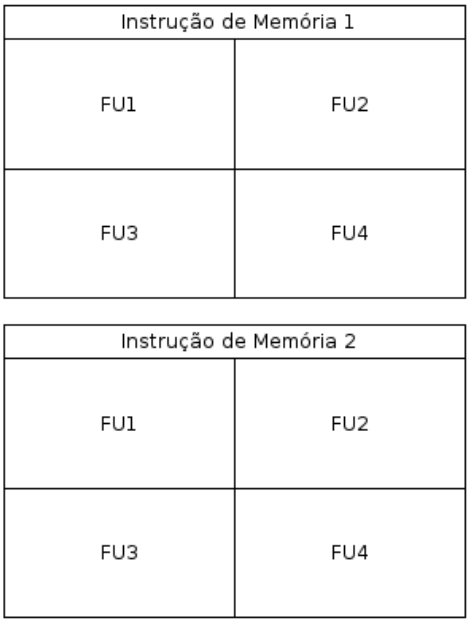
Fonte: Ferreira et al. (2011).

2.3.1 Escalonamento por Módulo

As CGRAs são adequadas para executar os laços. A arquitetura Adres introduziu a técnica de Escalonamento por Módulo (*Modulo Scheduling*—MS) nas CGRAs (MEI et al., 2003). Descrevendo o MS, a Figura 10a mostra o Grafo de Fluxo de Controle (*Control Flow Graph*—CFG) de um laço com 8 instruções. A técnica consiste em detectar as instruções paralelas no CFG e escaloná-las na memória de instruções de um processador paralelo, como a CGRA. A técnica agrupa instruções paralelas entre si em uma mesma instrução de memória, preferencialmente, para ocorrer paralelismo.



(a) CFG de um laço.

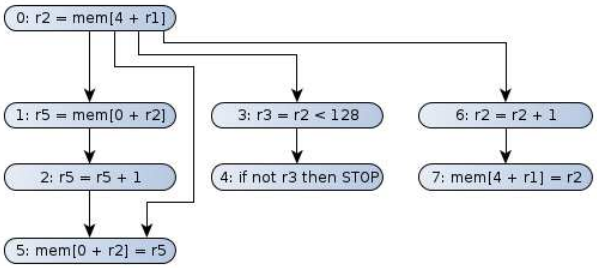


(b) 2 instruções vazias na memória de configuração, de 4 FUs cada.

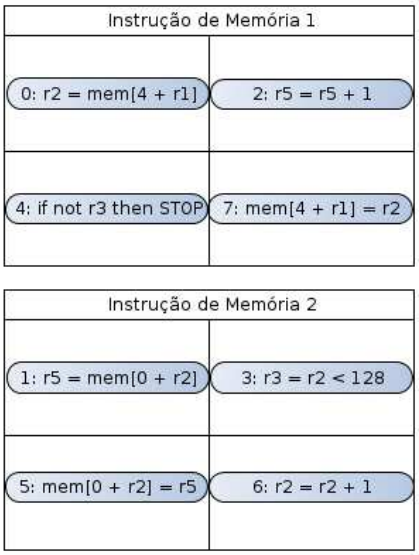
Figura 10 – Escalonamento de um laço por *Modulo Scheduling*.

São necessárias, no mínimo, 2 instruções de memória para escalonar todas as instruções paralelas do CFG do exemplo, considerando um processador paralelo de 4 palavras. Esse número de instruções de memória é chamado de Intervalo de Iniciação (*Initiation Interval*—II). A Figura 10b mostra essas instruções de memória de configuração antes do escalonamento, que devem ser em quantidade igual ao valor do II.

A Figura 11a apresenta o Grafo de Fluxo de Dados (*Data Flow Graph*—DFG) equivalente ao CFG da Figura 10a. A altura h_i de uma instrução é o número de arestas entre uma dada instrução e a instrução independente, no primeiro nível do DFG (instrução 0 do exemplo). Assim, a altura da instrução 3 nas Figura 10 é $h_3 = 1$, da instrução 5 é $h_5 = 3$, e assim por diante.



(a) DFG do laço.



(b) Instruções do laço escalonadas na memória de instruções.

Figura 11 – Escalonamento de um laço por *Modulo Scheduling* (continuação).

O MS determina que todas as instruções p_i que possuírem o mesmo módulo de sua altura no DFG em relação ao II, na forma $p_i = h_i \pmod{II}$, são instruções paralelas. O escalonamento final do CFG fica conforme a Figura 11b, dessa forma. Todas as instruções paralelas ficam agrupadas em suas respectivas instruções de memória de configuração, finalmente, de acordo com seus módulos. O laço, que antes apresentava uma IPC de $\frac{8}{8} = 1\frac{\text{Instrução Paralela}}{Ciclo}$ na Figura 10a, agora apresenta uma IPC de $\frac{8}{2} = 4\frac{\text{Instruções Paralelas}}{Ciclo}$ na Figura 10.

A vazão que o MS gera é de um novo resultado a cada 2 ciclos apesar do grafo de dependência ter a latência de 4 ciclos. O código é executado com uma sobreposição de duas iterações. O MS é uma técnica de *software pipelining* que realiza um *loop unrolling* dinamicamente ao sobrepor as iterações.

É necessário ainda encaminhar o resultado da instrução 0 para a instrução 5 consumi-la, de acordo com o DFG da Figura 11a. É preciso adicionar algumas instruções antes e depois do laço também, para tratar a inconsistência que algumas instruções podem produzir por executarem fora de ordem. Há a latência das instruções de memória também, que as figuras não apresentam. Mais detalhes da técnica estão fora do escopo desta descrição.

2.4 Instruções paralelas

Uma ideia geral dos compiladores ILP é reunir o máximo de instruções RISC paralelas e alocá-las em um mínimo de instruções longas para haver execução simultânea dessas instruções RISC reunidas e assim alcançar desempenho. Duas ou mais instruções RISC são paralelas quando estão de acordo com a Condição de Bernstein (BERNSTEIN, 1966) e quando não possuem dependência de controle entre si.

Dado um processo u, considerando $M(u)$ a lista de recursos de leitura e escrita que u modifica, como registradores ou linhas de memória, e $L(u)$ a lista de recursos que u lê. E dado um outro processo v, com as mesmas considerações feitas para o processo u. A Condição de Bernstein determina que esses dois processos u e v podem ser reordenados sem modificar os resultados do programa quando $M(u) \cap M(v) = M(u) \cap L(v) = L(u) \cap M(v) = \emptyset$. Dessa forma, nenhum processo que atende a essa condição modifica valores utilizados por outros e nenhum processo sucessivo utiliza dados que ambos modificam. É o que ocorre dentro das instruções de memória da Figura 11b e das instruções longas da Figura 1b.

Além disso, dois processos u e v possuem dependência de controle entre si quando v executa depois de u, possivelmente, e a execução de u determina se v irá ou não executar. Essa dependência acontece entre instruções de desvio e suas instruções sucessivas. Um exemplo são as instruções longas das linhas 10 e 12 da Figura 1b. Essas linhas estão em instruções longas separadas por terem dependência de controle entre si, apesar de satisfazerem a Condição de Bernstein.

2.5 A CGRA como um processador VLIW

É possível projetar uma CGRA para realizar, não apenas algumas, mas todas as operações que um GPP realiza. Um GPP realiza operações lógicas, aritméticas, de memória e de busca de instruções, com ou sem desvios, normalmente, e possui um caminho de dados para encaminhar resultados de operações para aquelas que os consomem. Muitos trabalhos implementam FUs de CGRA já capazes de realizar a maioria dessas operações. O que falta em uma CGRA para operar como um processador é um mecanismo de busca de instruções. Esse mecanismo pode capacitar a CGRA a realizar operações de desvio.

Seguindo essa ideia, as arquiteturas Trips (SANKARALINGAM et al., 2003) e Refree-MIPS (JOST, 2014) mostram a CGRA agindo como um processador VLIW. No caso do Refree-MIPS, o termo "Refree" vem da ideia de "ficar livre" do banco de registradores, do inglês "*register free*", tendo outra alternativa para ele.

A Figura 12 ilustra a disposição dos PEs de uma CGRA junto ao mecanismo de interconexão do Refree-MIPS. Trata-se de uma CGRA em rede global, como Ferreira et al. (2011) sugere (Figura 9). Jost (2014) propõe um caminho de dados alinhado entre as instruções RISC em uma memória de instrução longa e os PEs da Figura 12. Um outro mecanismo, não ilustrado aqui, possibilita a realização de operações de busca na memória de instruções. Dessa forma, o Refree-MIPS consegue realizar operações de um GPP VLIW sem necessitar de *hardware* extra. O Refree-MIPS apresenta uma arquitetura computacional mais simples com menos *hardware* extra. Seguindo ideia semelhante, mas utilizando redes de interconexão de duas dimensões ao invés da rede global, Sankaralingam et al. (2003) realiza operações de um GPP VLIW sem *hardware* extra com a arquitetura Trips também.

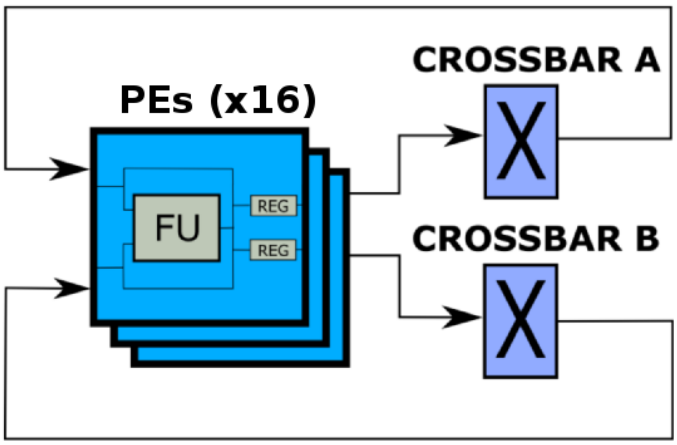


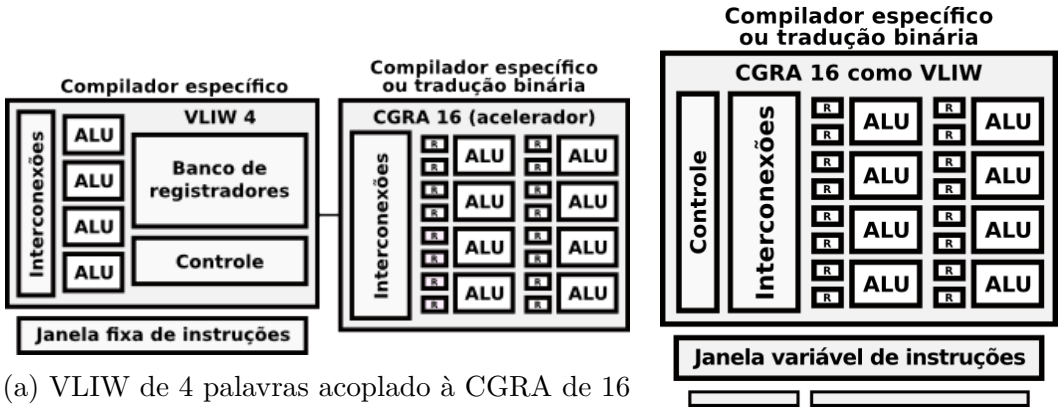
Figura 12 – Topologia de interconexão do Refree-MIPS.

Fonte: Jost (2014).

Cada PE da CGRA contém um ou mais registradores internos para realizar suas operações, normalmente. Uma operação aritmética deve ter um local para armazenar seus resultados ou seus operandos, temporariamente, por exemplo. A Figura 12 e a Figura 5 mostram registradores acoplados na saída de seus PEs com essa função. Jost (2014) então aproveita esses registradores para substituir o banco de registradores dos VLIW.

Para observar a união da arquitetura VLIW com a CGRA, a Figura 13 compara uma arquitetura VLIW de 4 palavras acoplada a um acelerador CGRA de 16 palavras com a proposta que substitui esse acoplamento em uma arquitetura VLIW única de 16 palavras construída sobre a CGRA. A CGRA não realiza operações de desvio na Figura 13a, mas, na Figura 13b, incorpora uma unidade de controle.

A arquitetura com acoplamento (Figura 13a) apresenta dois mecanismos de interconexão: um para os PEs do VLIW e outro para os da CGRA. Existe uma única rede de interconexão na arquitetura unificada (Figura 13b). A primeira arquitetura



(a) VLIW de 4 palavras acoplado à CGRA de 16 palavras.

(b) CGRA de 16 palavras como um VLIW.

Figura 13 – Comparação entre a arquitetura VLIW de 4 palavras acoplada a um acelerador CGRA de 16 palavras com a proposta da CGRA de 16 palavras com janela variável operando como um VLIW.

(Figura 13a) utiliza 20 PEs para uma janela máxima de 16 palavras também. A outra (Figura 13b) necessita de 16 PEs somente, para uma mesma janela máxima, um PE para cada palavra.

Tem-se ainda que a arquitetura unificada substitui todo o mecanismo de banco de registradores pelos registradores R dos PEs da CGRA de 16 palavras (Figura 13). Por fim, lançada na proposta deste trabalho, a arquitetura unificada utiliza uma janela de instruções que possui um tamanho variável (Figura 13b), em contraste com o tamanho fixo de janela da arquitetura da Figura 13a.

Encontra-se um compartilhamento de mecanismos de interconexão e de alguns PEs para uma mesma capacidade funcional no modelo da Figura 13b comparado ao da Figura 13a. Além disso, o banco de registradores é eliminado. Há, portanto, uma diminuição de complexidade entre o modelo de arquitetura VLIW acoplada a uma CGRA — apresentada na Figura 13a — e o modelo de unificação dessas arquiteturas apresentado na Figura 13b.

Arquiteturas VLIW com maior largura tendem a permitir maior aproveitamento de paralelismo dos códigos que elas executam. A Figura 14 demonstra o ganho de desempenho que uma arquitetura de 16 palavras tem sobre as arquiteturas de 4 e 8 palavras. O ganho está em porcentagem de ILP de exemplos em relação a seus ILPs máximos ideais. O Escalonamento por Módulo na Tradução Binária (*Binary Translation Modulo Schedule*—BTMS) (FERREIRA et al., 2015) constitui-se de uma arquitetura VLIW de 4 PEs acoplada a uma CGRA de 16 PEs, similar ao modelo na Figura 13a. O VLIW 16 encontra-se somente em simulação, atualmente, nos melhores conhecimentos disponíveis.

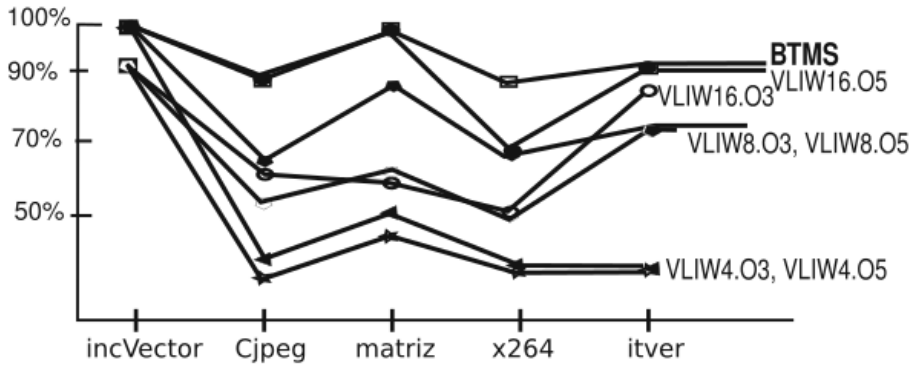


Figura 14 – ILPs de alguns exemplos em relação aos seus ILPs máximos de CGRAs de 4, 8 e 16 PEs e 2 acessos consecutivos na memória.

Fonte: Ferreira et al. (2015).

Sankaralingam et al. (2003), Jost (2014) e outros trabalhos recentes já citados apresentam arquiteturas VLIW que atingem seus objetivos de corretude, desempenho e custo. No entanto, a janela de instruções longas desses trabalhos é de tamanho fixo. Janelas fixas de instruções longas degradam a ocupação de memória de instrução, como mencionado. Isso por causa da presença das instruções paralelas RISC nulas no final da memória de instruções.

2.6 Considerações finais

O banco de registradores impõe alto custo nos processadores VLIW. A distribuição de registradores pela utilização dos PEs interconectados possibilita reduzir esse custo.

As instruções RISC nulas presentes no fim das instruções longas desperdiçam espaço na memória de instrução. Há a possibilidade de evitar o desperdício. O compilador pode alocar apenas instruções RISC úteis na memória. O processador VLIW pode reduzir a expansão de código na memória de instrução ao evitar instruções RISC nulas.

A eficiência do simulador VEX provê ferramentas capazes de comprovar os modelos de arquiteturas VLIW. O simulador disponibiliza detalhes suficientes para verificar a corretude e o desempenho de uma nova proposta.

Jost (2014) apresenta comprovações de que as CGRAs de rede global (FERREIRA et al., 2011) são adequadas para a construção de uma arquitetura VLIW. Outros trabalhos (FERREIRA et al., 2014; FERREIRA et al., 2015) confirmam a viabilidade de aproveitar esses dispositivos com rede global com até 16 FUs. E as CGRAs permitem a aplicação de técnicas de ILP, como o MS. Porém, requerem

a presença de um processador VLIW além da CGRA para agir como um GPP. A eficiência energética reforça o potencial de utilização das CGRAs.

Sankaralingam et al. (2003) e Jost (2014) comprovam a possibilidade da fusão de uma CGRA com o VLIW em um mesmo dispositivo. Essa fusão possibilita o compartilhamento da rede de interconexão e dos registradores dos PEs. Tal compartilhamento serve para a adoção do mecanismo de registradores distribuídos em uma mesma arquitetura sem dispositivos acoplados, mas existem potenciais melhorias na ocupação da memória de instrução e na alocação de registradores com o auxílio do compilador.

Essas considerações motivam a proposta que este trabalho apresenta. Os próximos capítulos descrevem a proposta e utiliza um estudo de caso para testar seus conceitos.

3 Arquitetura do processador

Esta seção apresenta a proposta de um processador de instruções longas reconfigurável em nível de palavras (*"Coarsed" Reconfigurable Large Processor*— Corelap). Trata-se de um novo modelo de processador VLIW, baseado nas CGRAs. O Corelap é uma fusão das arquiteturas VLIW e CGRA.

3.1 Banco de registradores

O Corelap apresenta uma arquitetura em que os registradores ficam distribuídos juntos aos PEs. O banco de registradores possui, no modelo tradicional, decodificadores internos para selecionar quais registradores estarão ativos para escrita e/ou leitura, como ilustra a Figura 15. As saídas do banco ficam ligadas em uma ALU. Esse é um banco de registradores de uma escrita e duas leituras consecutivas. Um banco de N escritas e $2N$ leituras precisaria de N decodificadores de 1-para-2^N na escrita e $2N$ decodificadores de 2^N-para-1 na leitura.

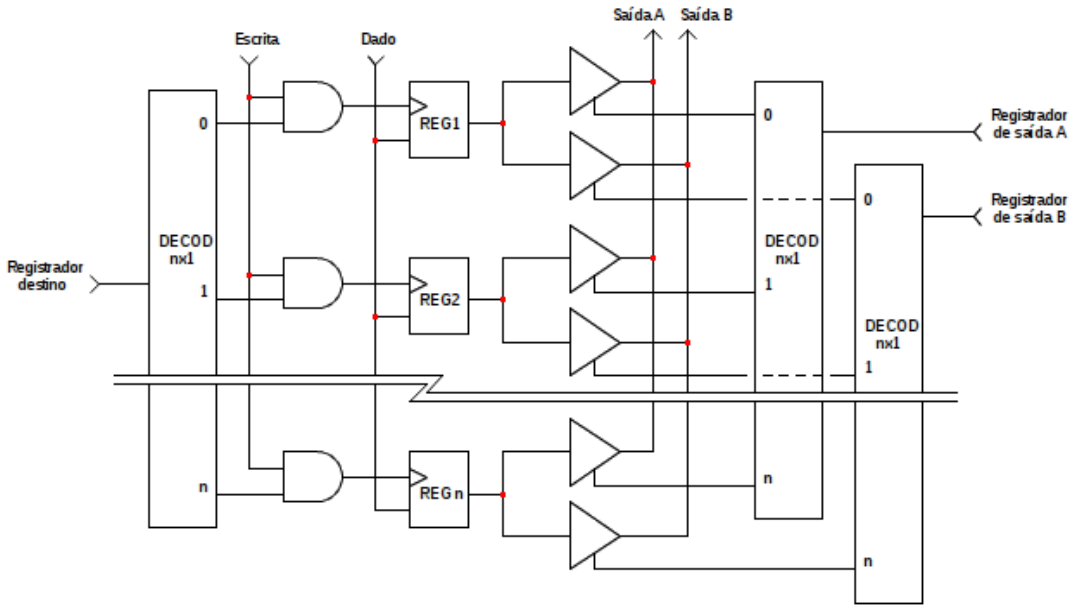


Figura 15 – Circuito de um banco de N registradores de uma escrita e duas leituras consecutivas.

A proposta com o banco distribuído não usa decodificadores. Os registradores ficam conectados aos PEs ou FUs através de redes de interconexão, que substituem os decodificadores, como ilustra a Figura 16. As N saídas A e B do exemplo são conectadas aos operandos dos N PEs. Os endereços das redes de interconexões servem

para selecionar os registradores em cada entrada e saída de dados. O banco realiza N escritas e $2N$ leituras, consecutivamente.

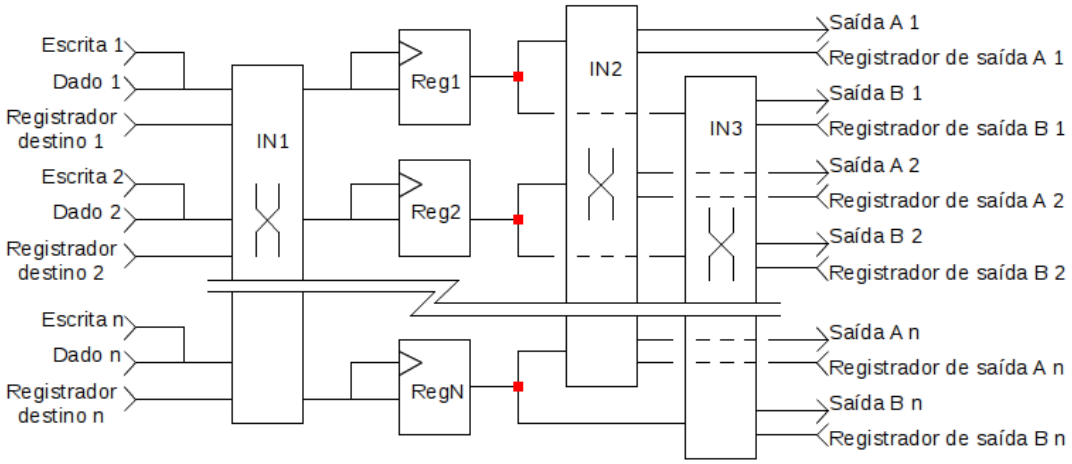


Figura 16 – Banco de registradores distribuído de N escritas e $2N$ leituras consecutivas.

A rede possibilita a transferência do resultado de um cálculo para qualquer outro PE, ficando o resultado disponível no próximo ciclo de relógio. O formato Arquitetura de Conjunto de Instruções (*Instruction Set Architecture*—ISA) de uma arquitetura RISC tradicional tem campos para especificar o endereço dos dois registradores fonte e um campo para o registrador destino. Supondo haver 32 registradores, são então necessários 5 *bits* para comandar os decodificadores em um RISC, como ilustra a Figura 17. O campo RS significa *registrador fonte* (*source register*) e RT significa *registrador após o registrador fonte*, com "T" sendo uma letra após a letra "S", simplesmente. Eles correspondem aos endereços dos dois registradores fonte do PE. O campo RD corresponde ao endereço do registrador destino (*register [of type] destination*).

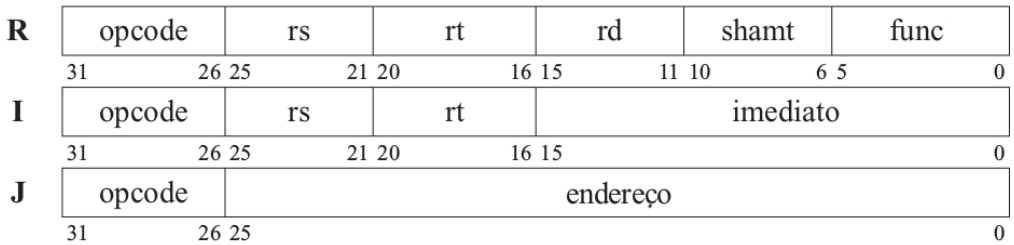


Figura 17 – Formato de instruções básicas do MIPS, um processador RISC.

Fonte: Patterson e Hennessy (2013).

Esta dissertação apresenta uma nova proposta de formato ISA. Não é necessário especificar o destino, apenas os registradores fonte, que controlam a rede de

interconexão. Pode-se dizer que os decodificadores substituem os multiplexadores, com o controle direto para cada PE, como ilustra a Figura 18, simbolicamente. Vê-se parte da ISA com os 16 campos Opcode, que definem as operações dos PEs, e os 32 campos de registradores fonte RS e RT. Há uma instrução RISC para cada PE e para cada campo da ISA. Como um detalhe importante, o registrador de saída é fixo, ligado na ALU da FU diretamente. O formato ISA completo está na seção 3.4.

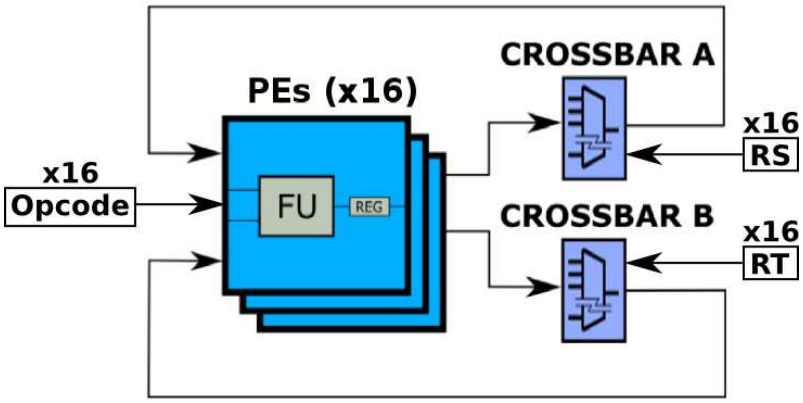


Figura 18 – Caminho de dados do Corelap com parte do formato de instruções.

Essa é a primeira contribuição desta dissertação, que apresenta uma arquitetura com o registrador de destino fixo. Esse conceito elimina a necessidade de um banco de registradores com estrutura para permitir múltiplas escritas, que é um dos pontos que aumenta, significativamente, o custo dos processadores VLIW e limita o tamanho máximo da janela.

Entretanto, a simplificação do registrador de saída gera um trabalho extra para o compilador, como a seção 3.6 detalha. Além disso, é necessário adicionar registradores temporários para a movimentação dos dados que resolve conflitos provenientes do registrador fixo de saída. Mas há casos em que o compilador pode gerar código sem a necessidade de uso dos registradores de movimentação.

3.2 Janela de instruções variável

Ampliar o tamanho da janela pode não ser vantajoso em uma arquitetura de janela fixa, apesar de contribuir para o desempenho. Isso porque o paralelismo disponível varia ao longo da execução do programa. Os trechos com pouco paralelismo geram instruções longas com poucas instruções RISC paralelas. Mas cada linha da memória de instruções tem capacidade fixa de registrar uma instrução RISC para cada PE. Como a janela fixa utiliza apenas uma instrução longa em cada linha, ocorre uma subutilização da memória.

A Figura 19 ilustra uma alocação de instruções longas de vários tamanhos em uma arquitetura de janela fixa de 4 PEs. Essa ilustração é equivalente à alocação de instruções do código da Figura 1b. Fica uma instrução longa em cada linha. Há espaços nos finais das linhas de memória que o compilador não pode alocar instruções RISC efetivas, no caso de um aproveitamento parcial (Figura 19b). Somente a última linha, talvez, contenha espaços vazios no aproveitamento total (Figura 19c).

(a) Janela fixa.

(b) Janela variável com pre-enchimento parcial.

(c) Janela variável com pre-enchimento total.

Figura 19 – Instruções RISC alocadas nas linhas de memória em uma arquitetura de 4 PEs.

Já o formato ISA do Corelap é uma instrução longa VLIW de comprimento variável. O número de instruções RISC determina o comprimento da instrução longa. A instrução longa anterior é que especifica o tamanho da próxima instrução. Dessa maneira, o compilador pode utilizar uma linha de memória para alocar mais de uma instrução longa (Figura 19b e Figura 19c). Durante a execução, o Corelap ativa somente as instruções RISC que a instrução longa utiliza naquele ciclo.

A Figura 19b ilustra a janela variável equivalente ao código da Figura 1b. A segunda e a terceira linhas possuem, agora, instruções longas que não aparecem na janela fixa. Essas duas linhas estão ocupadas totalmente. A janela fixa utiliza 6 linhas de memória para alocar esse trecho de código. A janela variável utiliza apenas 4, um ganho de 1,5× em ocupação da memória de instruções.

A primeira linha de memória da Figura 19b ainda continua com espaços vazios no final da linha porque a próxima instrução longa ocupa um espaço maior que o disponível. O compilador considera que a instrução longa dessa primeira linha tem tamanho 4, nesse caso, e preenche tais espaços vazios com instruções RISC nulas, como faz na janela fixa.

A memória ocupa todos os espaços vazios no final de todas as linhas de memória, no caso da Figura 19c. A janela variável ocupa 3 linhas de memória, com um ganho de 2,0×, no exemplo em questão.

Esta dissertação propõe o processador Corelap em nível ISA e arquitetural. Em paralelo com o trabalho, como uma consequência direta da proposta da dissertação,

e de forma complementar, Oliveira Júnior (2015) detalha o formato das instruções e a especificação em linguagem de *hardware* para extração de estimativas de custos. Ou seja, esta dissertação apresenta o Corelap enquanto o trabalho paralelo proposto em Oliveira Júnior (2015) auxilia a validação da proposta com a descrição detalhada de partes da arquitetura. Um dos pontos detalhados é uma estrutura de interconexão para a memória de instruções. Essa estrutura permite o acesso a uma janela variável com conexão direta aos PEs e à rede de interconexão que faz o mapeamento dos registradores.

Esta dissertação especifica, além da definição em nível arquitetural, as tarefas que o compilador deve realizar para viabilizar a proposta. A próxima seção mostra alguns detalhes da arquitetura e aborda as tarefas do compilador posteriormente.

3.3 Arquitetura alvo

Um processador VLIW deve ser capaz de utilizar códigos com ILP explícito para executar instruções RISC paralelas em um único ciclo. Possibilitando esse único ciclo, esta proposta apresenta um processador com um alinhamento entre, de um lado, a memória de instruções do VLIW e, de outro, o arranjo de FUs e o fluxo de controle da CGRA. A Figura 20 mostra o núcleo do caminho de dados básico do Corelap. O conjunto FU_I, composto de N unidades funcionais FU_i ($i \in I$, $I \equiv \{1, 2, \cdots, N\}$, $N \equiv$ *Número de unidades funcionais*), representa o arranjo de FUs da CGRA. Os conectores IMM_I na Figura 20 e todos os sinais de controle da instrução longa VLIW estão designados à saída de uma única memória de instrução diretamente. As redes de interconexão IN_1 e IN_2 consomem os registradores internos R^{FU_I} e os registradores externos R_I e alimentam o primeiro e segundo operandos das FUs, respectivamente. A rede IN_3 consome os registradores internos R^{FU_I} somente, e alimenta os registradores externos R_I somente. Um mecanismo de busca, descrito na seção 3.5, é capaz de dispensar todas as instruções RISC paralelas de uma instrução longa VLIW em um único ciclo. E isso de forma alinhada a todas as N entradas paralelas.

Como já foi mencionado, com essa busca de memória de instrução de um único passo em alinhamento com as FUs da CGRA, o Corelap é capaz de executar todas as instruções RISC paralelas de cada uma dessas instruções longas VLIW dispensadas em um único ciclo.

Pode-se observar as principais características da arquitetura do novo processador nesse exemplo ilustrativo com apenas 4 unidades. Os sinais de controle do Corelap, designados aos seus PEs e às redes de interconexão IN_1, IN_2 e IN_3 da Figura 20, contribuem para a exposição de ILP. O compilador irá utilizar essa

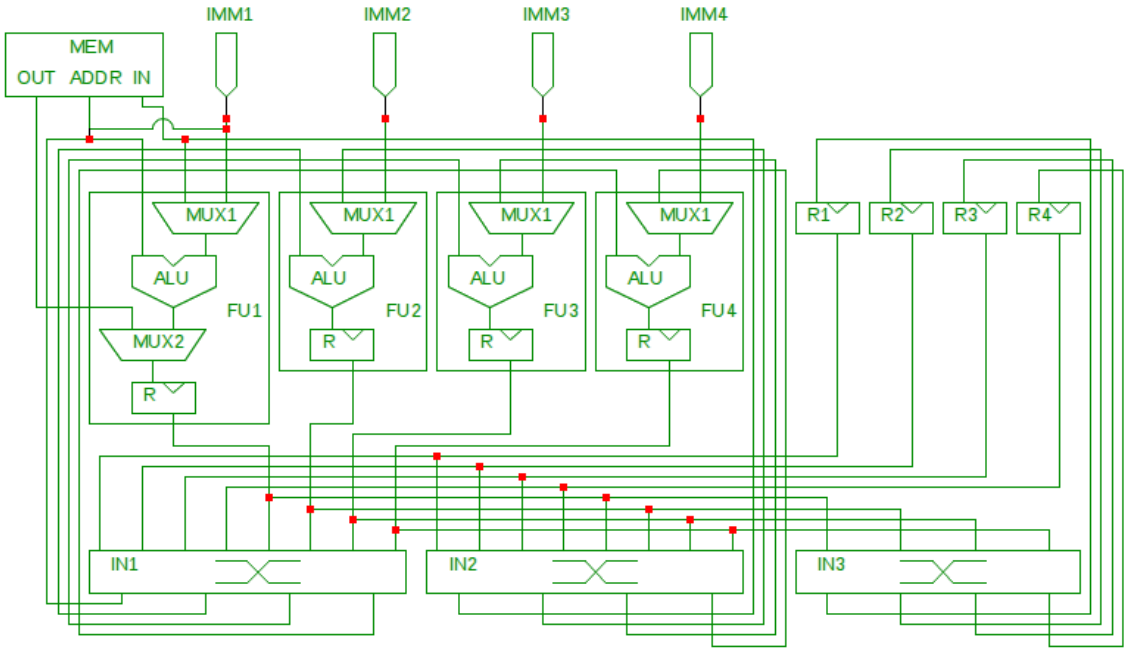


Figura 20 – Núcleo do caminho de dados do Corelap de 4 FUs and 1 operação concorrente de leitura e escrita em memória.

explicitação para fazer a atribuição paralela dos registradores e a movimentação paralela dos dados.

3.3.1 Caminho de dados

O caminho de dados do Corelap é um caminho circular em que o processador recebe, opera, registra e retorna os dados, realizando o processamento. As três redes IN_1, IN_2 e IN_3 são responsáveis pela intercomunicação dos PEs. As entradas da IN_1 recebem os registradores de destino das FUs do ciclo anterior e os registradores de movimentação (ou auxiliares). A rede IN_3 serve para implementar uma movimentação nos registradores auxiliares. O compilador é quem faz a alocação e, na proposta desta dissertação, a movimentação dos registradores através da programação das redes de interconexão. É o equivalente à atribuição de registradores que os processadores RISC e VLIW realizam, como já destacado.

A unidade funcional especial FU_1 precisa de um operando extra para inserir dados da memória para dentro do núcleo do caminho de dados. Esse operando tem a responsabilidade de permitir suporte à memória no caminho de dados do Corelap. O Corelap conecta esse operando à entrada do multiplexador $MUX_2^{FU_I}$ da Figura 20, permitindo essa inserção de dados. O fato da unidade funcional FU_1 ser diferente de outras FUs caracteriza o Corelap como uma arquitetura heterogênea.

Ocorre um problema quando os registradores internos R^{FU_I} da Figura 20

pegam os resultados da ALU^{FU_I} de cada ciclo. O Corelap permite que instruções RISC paralelas consumam dados de registradores vindos de uma instrução longa VLIW de um ciclo anterior. Entretanto, o Corelap sobrescreve os dados de cada registrador interno R^{FU_i} a cada ciclo.

A Figura 21 mostra quatro casos de sequenciamento de instruções longas na memória de instruções. Considerando que uma instrução longa sucessora (ou precedente) seja a instrução que aparece depois (ou antes) de uma dada instrução (Figura 21). Considerando que uma instrução longa adjacente seja uma instrução que aparece imediatamente depois ou imediatamente antes (Figura 21). Não há problemas quando uma instrução longa adjacente consome os dados da instrução longa corrente. É o que ocorre entre a primeira e segunda instruções longas para o registrador r_1 da Figura 21, por exemplo. Mas uma instrução longa sucessora não adjacente pode falhar se ela precisar consumir um dado produzido pela instrução longa corrente. É o caso da relação entre a primeira e a terceira instruções longas para o registrador r_3 se forem executadas seguindo o modelo de registrador destino fixo, como ilustra a Figura 21.

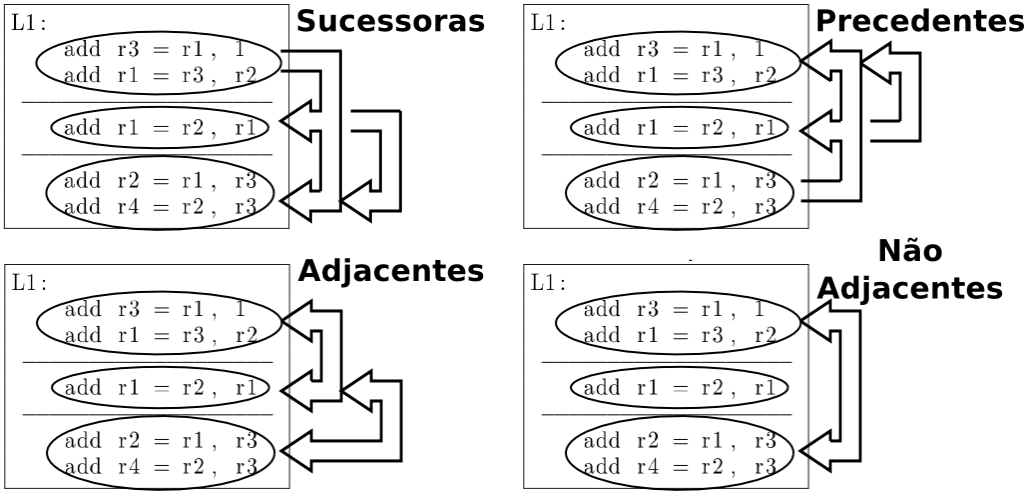


Figura 21 – Sequenciamento de instruções longas na memória de instruções: sucessão, precedência, adjacência e não adjacência de instruções longas.

O Corelap deve prover um mecanismo capaz de garantir a persistência de todos os dados dos registradores internos R^{FU_i} entre instruções longas não adjacentes. A solução proposta é utilizar os registradores externos R_I da Figura 20 para permitir a passagem de dados entre instruções longas não adjacentes. O Corelap preserva o dado desse registrador em um registrador externo R_i enquanto uma instrução longa está sobrescrevendo um registrador interno R^{FU_i}.

Existe um dado útil em cada unidade funcional FU_i, no pior caso. Por causa disso, o Corelap tem um registrador externo R_i para deixar vivo cada dado de cada

unidade funcional FU_i. Mas nem sempre um dado de um registrador externo R_i está na mesma ordem que os registradores internos R^{FU_I} estão na organização da arquitetura. O compilador que fica responsável por gerar N^N conjuntos diferentes de permutações — permutação de N com repetição — de uma sequência de dados de resultados destino no conjunto de registradores internos R^{FU_I}. O compilador tem informações adicionais para saber quais os valores deve preservar. Como já mencionado, minimizar o número de registradores vivos é um problema NP-completo. Se o conjunto de registradores não é suficiente, o compilador gera instruções de *load* e *store* para armazenar os valores na memória, temporariamente, que é conhecido com o problema de *espalhamento de registradores* (*register spilling*). Nos exemplos que usamos para validar o Corelap, não foi necessário gerar *register spilling*.

Tratando isso, a rede de interconexão IN_3 da Figura 20 faz a intermediação entre os registradores internos R^{FU_I} e o registradores externos R_I para suportar todos esses conjuntos de permutação. O Corelap conecta alguns sinais de controle de instruções RISC paralelas, diretamente às respectivas entradas de endereçamento de IN_3. Dessa maneira, os compiladores são capazes de utilizar qualquer registrador externo R_i para tornar vivos quaisquer dados dos registradores internos R^{FU_i}. Esse aparato de tornar dados vivos possibilita que cada dado dos registradores internos R^{FU_i}, que a arquitetura sobrescreve de forma indesejada, se torne vivo entre quaisquer instruções longas não adjacentes.

Caminhos críticos em circuitos lógicos sequenciais são o maior caminho que existe entre uma porta de entrada e uma porta de saída de qualquer componente sequencial em um circuito. Por esse conceito, a serialização com as redes de interconexão IN_1 e IN_2, o multiplexador $MUX_1^{FU_1}$, a unidade ALU^{FU_1} e o multiplexador $MUX_2^{FU_1}$ representa o caminho crítico básico do Corelap. O caminho crítico é uma característica importante, porque determina a frequência máxima de operação que um circuito é capaz de suportar. Quanto maior for a frequência que um circuito opera, mais desempenho ele tem capacidade de apresentar.

3.4 Formato da instrução

Nesta seção iremos detalhar um formato de instrução que foi proposto no trabalho de Oliveira Júnior (2015), dando continuidade à proposta de arquitetura apresentada nesta dissertação. A ideia foi gerar um primeiro modelo para poder detalhar uma implementação física do processador. O objetivo desta dissertação é mostrar o potencial da solução no nível arquitetural. Esta seção mostra que é possível ter um formato regular de instrução, semelhante às arquiteturas RISC e VLIW tradicionais. Como estamos propondo um processador com janela variável,

cada janela será composta por um bloco de 1 até N instruções nos formatos descritos a seguir. Pode-se observar que existem campos não usados.

A Figura 22 ilustra esse formato. Cada instrução longa tem um formato idêntico para cada instrução RISC paralela de 32 *bits*. Os números acima de cada bloco são as posições dos *bits* das unidades de formato. Esse formato representa um formato RISC, com apenas 8 tipos de instruções simples.

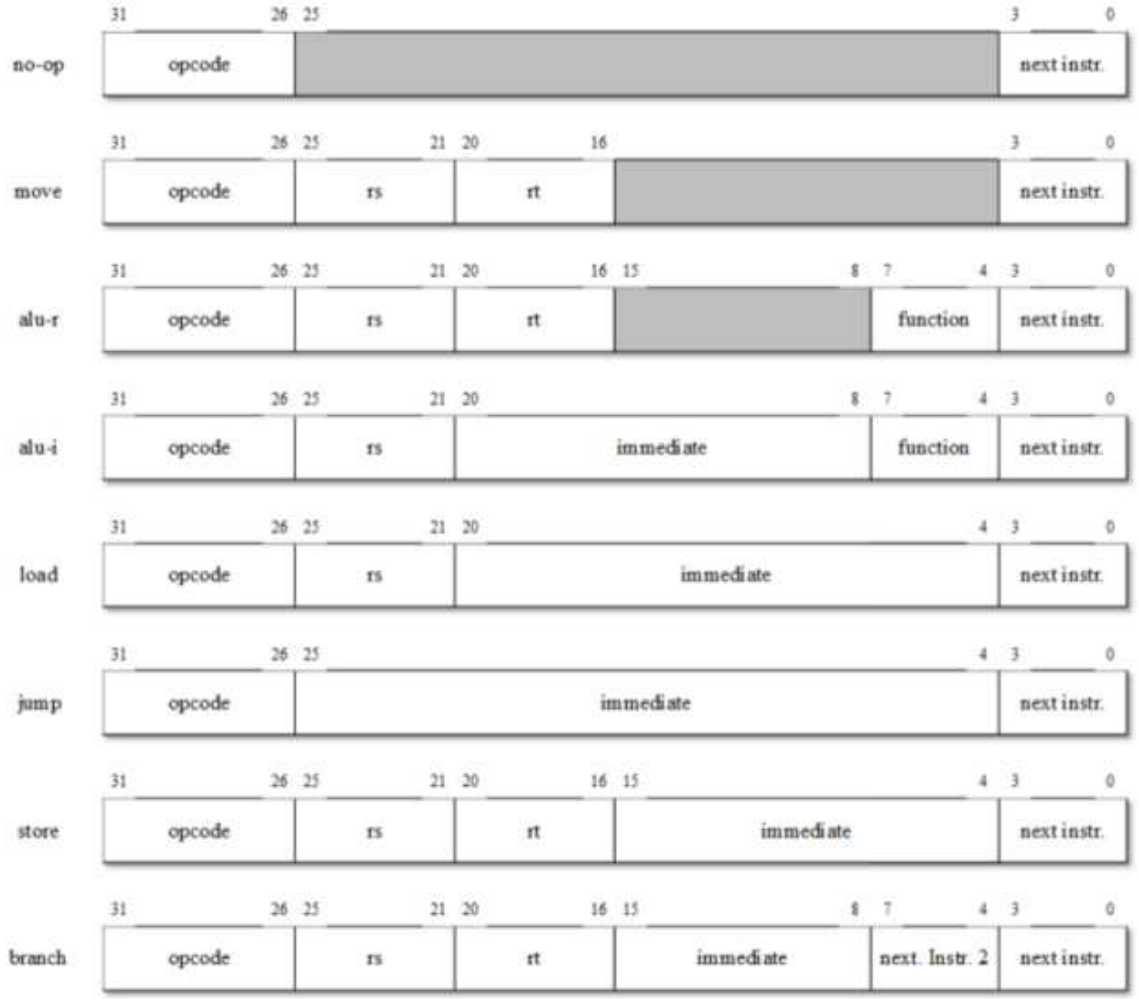


Figura 22 – Formato de instruções RISC paralelas do Corelap.

Fonte: Oliveira Júnior (2015).

A unidade de formato OPCODE da Figura 22 representa a operação que o Corelap deve realizar. Podem ser operações lógicas, de memória ou de controle. Os formatos RS e RT representam o primeiro e o segundo operandos das FUs, respectivamente. RT representa um operando de memória também. RS significa *registrador fonte* (*source register*) e RT significa *registrador após o registrador RS*, como dito. A unidade de formato IMMEDIATE da Figura 22 representam o operando imediato de instrução RISC, o qual é conectado, por exemplo, a cada entrada IMM_i

do Corelap diretamente. A unidade FUNCTION é a operação aritmética que a ALU deve executar. Finalmente, a unidade NEXT-INSTR-1 representa o número de instruções RISC paralelas que uma instrução longa sucessivamente adjacente em tempo real de execução possui. NEXT-INSTR-2 tem o mesmo significado de NEXT-INSTR-1, porém equivale a instruções longas adjacentes nas quais os desvios podem efetivamente ser tomados em tempo de execução.

De acordo com o número de *bits*, esse formato suporta até 32 operações diferentes, até 32 posições de registradores e até 16 funções diferentes de ALU; suporta, também, até 16 unidades funcionais e dados imediatos de 8 a 22 *bits*.

Instruções de formato NO-OP são instruções nas quais o Corelap não executa nenhuma operação, apenas realiza uma busca pela próxima instrução longa. O Corelap move o valor de um registrador interno R^{FU_i} da Figura 20 para um registrador externo R_i, nos formatos MOVE. Instruções de formato ALU-R realizam as operações das unidades ALU^{FU_I} que precisam de dois registradores como operando. O formato ALU-I realiza operações das unidades ALU^{FU_I}, de dois operandos também, mas um é o registrador como operando e o outro um imediato como operando.

Os formatos de instrução LOAD e STORE representam as operações de memória. A unidade de formato RT representa o deslocamento (*offset*) de endereço em ambos os formatos, nesse caso. E a unidade de formato RS representa os registradores internos R^{FU_i} ou externos R_i dos quais a memória deve pegar o valor para armazenar na operação de STORE.

JUMP e BRANCH representam desvios incondicionais e condicionais, respectivamente. O Corelap é capaz de tomar um desvio de acordo com a comparação entre dois registradores operandos nos desvios condicionais, e o formato IMMEDIATE dos formatos de instrução do tipo desvio representa o deslocamento da memória de instrução que o Corelap deve proceder no caso de um desvio ser tomado.

Algumas unidades de formato da Figura 22 não têm a mesma extensão de *bits* entre os formatos de instrução. Essa característica degrada a regularidade da arquitetura, mas é necessária para alocar um número máximo de funcionalidades em uma mesma instrução longa de 32 *bits* de largura.

3.5 Busca com janela variável

O Corelap é um processador reconfigurável que pode executar de 1 a N instruções RISC paralelas, permitindo variar o número de instruções em intervalos múltiplos de 1 (uma) instrução paralela. Se o bloco de instruções for pequeno, se comporta como um processador VLIW e, se o bloco for grande, terá o comportamento

de uma CGRA. O Corelap integra tudo em uma única arquitetura. Essa é a maior diferença em relação aos trabalhos anteriores, como já mencionado, nos quais o sistema tinha dois componentes (um VLIW e uma CGRA).

Semelhante à seção 3.4, iremos detalhar uma proposta derivada desta dissertação para o mecanismo de busca que Oliveira Júnior (2015) apresenta. A ideia básica é especificar o tamanho da janela do próximo bloco na instrução corrente e permitir instruções de desvios condicionais e incondicionais também.

A Figura 23 ilustra o modelo de mecanismo de busca do Corelap. Oliveira Júnior (2015) apresenta uma implementação desse mecanismo.

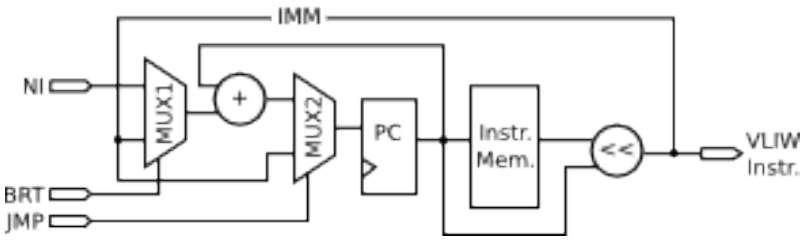


Figura 23 – Mecanismo básico de busca de instruções do Corelap.

A busca do Corelap oferece três alternativas: endereçamento *diretamente imediato*, *relativamente imediato* e *relativamente variado*. No endereçamento *diretamente imediato*, o controle de fluxo do Corelap ativa o sinal de entrada JUMP para forçar o MUX_2 a colocar o valor IMM para dentro do registrador PC (Figura 23). O valor IMM se encontra na primeira instrução RISC paralela, presente no sinal de saída VLIW-INSTR da Figura 23. Esse endereçamento ocorre em operações de desvio somente, seja condicional ou incondicional.

A alternativa de endereçamento *relativamente imediato* da busca do Corelap é similar ao endereçamento diretamente imediato. Ambos utilizam o barramento IMM da Figura 23 para calcular o endereço da memória. Mas o endereçamento relativamente imediato utiliza o valor do barramento IMM como um fator para, com o valor do registrador PC sendo o outro fator, realizar uma operação de soma aritmética para compor o valor final do PC (Figura 23). O endereçamento relativamente imediato ocorre em operações de desvio somente, como no endereçamento diretamente imediato.

O endereçamento *relativamente variável* ocorre quando não há uma operação de desvio, condicional ou incondicional, na instrução longa em execução. Ou quando um desvio condicional está em execução, mas não está sendo tomado. Outras arquiteturas, como a do MIPS (PATTERSON; HENNESSY, 2013), aumenta o valor do PC por um valor constante nessas instruções sem desvio. O Corelap, por sua vez, incrementa o PC, em valores múltiplos de 1, com a janela variável da instrução anterior.

3.5.1 Memória com janela variável

O mecanismo de busca inclui uma nova proposta para a arquitetura da memória de instruções. As memórias têm uma estrutura 2-D com decodificador de linha e coluna, normalmente. Multiplexadores fazem a seleção do subconjunto de linha corrente nas saídas. Propomos uma rede com entrada de tamanho $2N$, em que N é o tamanho máximo da janela, para ter uma janela variável de instruções. Essa rede pode utilizar a topologia multiestágio e o padrão Ômega (GRAMMATIKAKIS; HSU; KRAETZL, 2000). Sua função é deslocar a janela sobre o espaço $2N$ para gerar uma saída de M instruções, no qual $1 < M < N$.

Suponha $N = 4$ e uma memória com 8 colunas como exemplo da proposta da memória de instruções de janela variável, como ilustra a Figura 24. Cada coluna representa uma instrução RISC que a arquitetura suporta em cada palavra. As colunas de sufixo "a" da Figura 24 estão ligadas à primeira partição da Rede Ômega (entradas de 0 a $N-1$) e as colunas de sufixo "b" estão na segunda, e última, partição (entradas N a $2N-1$) da rede.

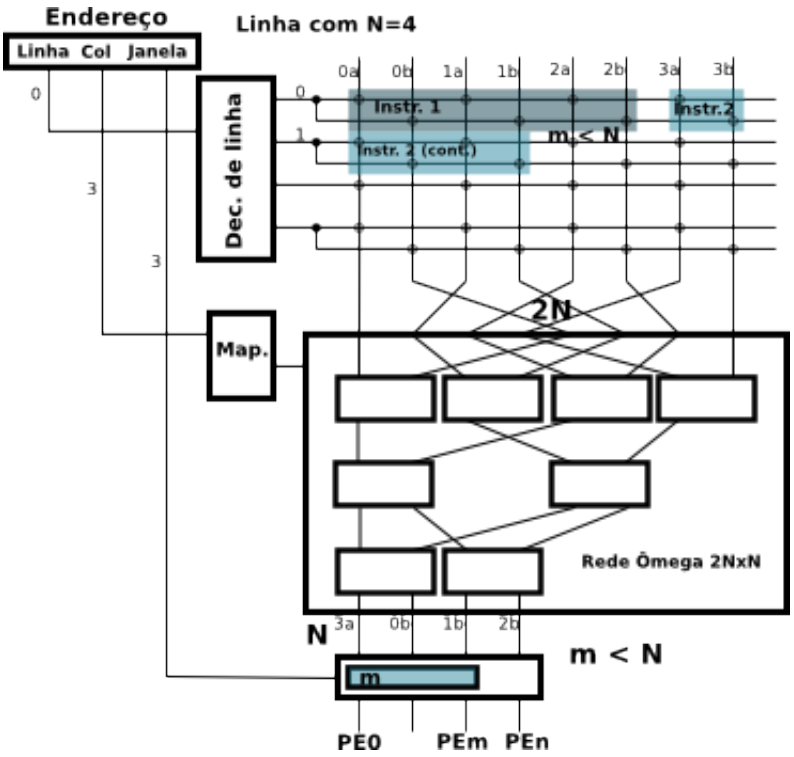


Figura 24 – Memória com janela variável de 4 colunas.

O decodificador de linhas seleciona uma ou duas linhas de acordo com a posição da primeira instrução RISC da instrução longa atual e das instruções longas sucessivas. O decodificador de linha ativa as linhas de memória, nas quais tais instruções longas se encontram, se acaso uma instrução RISC da instrução longa atual ou a primeira instrução RISC da instrução longa sucessiva estiverem, ao mesmo

tempo, em outra linha de memória. Todas as instruções RISC da instrução longa "Instr. 1" estão posicionadas na linha 0 do decodificador de linha na Figura 24. E a arquitetura precisa de todas as instruções RISC em sua entrada para a efetiva execução VLIW. Então, apenas a linha 0 precisa estar ativa para entregar as instruções RISC da "Instr. 1".

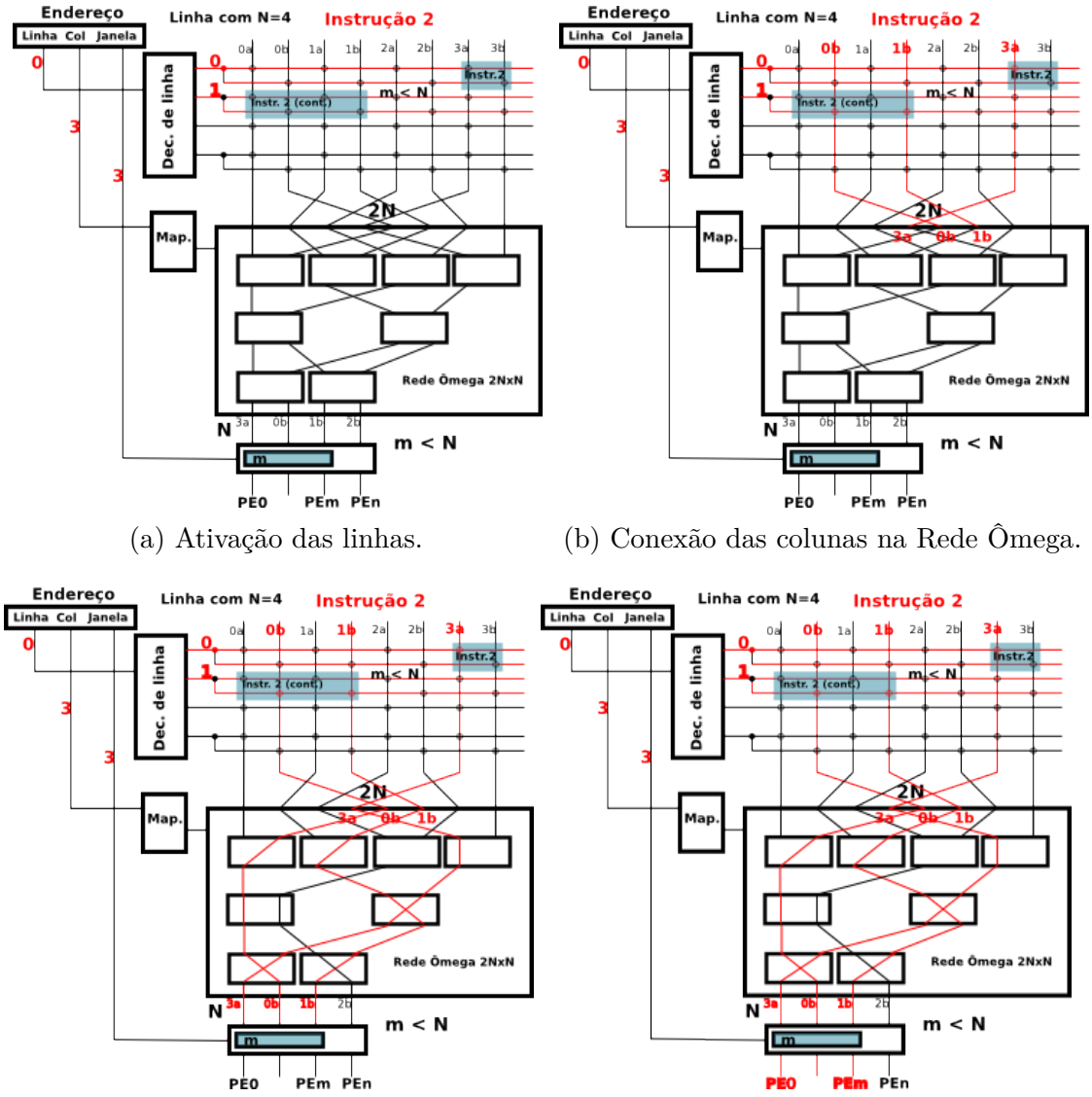
Já a instrução longa "Instr. 2" da Figura 24 tem algumas de suas instruções RISC posicionadas na linha 0 e outras na linha 1 do decodificador de linha. O decodificador tem que ativar ambas as linhas para entregar todas as instruções RISC da "Instr. 2" na entrada da Rede Ômega, nesse caso.

As linhas de mesmo número na Figura 24 contêm dados idênticos, necessitando então de uma duplicação de dados ao gravar a memória. A rede tem acesso às 4 colunas da linha que contêm a instrução longa atual e às 4 colunas da linha que contêm a próxima instrução longa (ou o restante de instruções RISC da instrução atual). Isso porque as colunas de sufixo "a" estão em uma partição da Rede Ômega e as de sufixo "b" em outra. A rede pode deslizar as colunas dessa forma para transferir a coluna que contém a instrução longa atual para a primeira coluna de sua saída, o que a arquitetura requer para funcionar. Um outro decodificador ativa a janela de tamanho M na última saída.

A Figura 25 mostra a memória variável da Figura 24 buscando a instrução longa "Instr. 2", sombreada na figura. A "Instr. 2" possui uma janela de tamanho 3, portanto composta de 3 instruções RISC. A primeira instrução RISC da "Instr. 2" está nas colunas 3a e 3b da linha 0 da memória. O restante das instruções RISC dessa instrução longa estão nas colunas 0a(b), 1a(b) e 2a(b) da linha 1, nessa sequência. Como a "Instr. 2" está nas linhas 0 e 1 da memória, o decodificador tem que ativar essas duas linhas para buscá-la (Figura 25a). Todas as instruções RISC da instrução longa "Instr. 2" aparecem na entrada da Rede Ômega, como ilustra a Figura 25b.

As instruções RISC da "Instr. 2" aparecem na sequência esperada (3a, 0b, 1b), na entrada da Rede Ômega (Figura 25b), por causa da topologia que define as conexões entre as colunas da memória e as entradas da Rede Ômega e pelo fato das colunas de sufixo "a" e "b" possuírem o mesmo dado em suas respectivas colunas. Mas essas instruções RISC estão desalinhadas já que a primeira instrução RISC não está na primeira posição, mas na entrada 2. Vale notar que a primeira instrução RISC, 3a, gravada na coluna de sufixo "a", está na primeira partição da Rede Ômega (entradas 0 a $N-1$), e as instruções RISC 0b e 1b, de sufixos "b", na última partição (entradas N a $2N-1$).

A arquitetura requer que as instruções RISC de uma instrução longa estejam na sequência correta e que a primeira instrução RISC esteja alinhada no primeiro PE da arquitetura. A Rede Ômega desliza as instruções RISC para a esquerda com o



(a) Ativação das linhas.

(b) Conexão das colunas na Rede Ômega.

(c) Roteamento das colunas.

(d) Seleção das colunas na saída.

Figura 25 – Exemplo de busca na memória variável, com destaques em vermelho, de uma instrução longa com janela de 3 instruções RISC situada nas linhas 0 e 1 da memória de instruções e com a primeira instrução RISC situada na coluna 3.

cálculo da coluna da primeira instrução RISC que o mecanismo de busca do Corelap (Figura 23) realiza, alinhando-as com a arquitetura. É o que exemplifica a Figura 25c, que desliza as instruções RISC desalinhadas da instrução longa “Instr. 2” em 3 entradas para a esquerda. A “Instr. 2” aparece na sequência correta a partir da coluna 0 na saída da Rede Ômega, dessa forma.

A arquitetura requer a definição de quais instruções RISC na saída da Rede Ômega estão ativas. Isso depende do tamanho da janela. A janela da instrução longa “Instr. 2” é de tamanho 3 no caso da Figura 25d. Assim, o decodificador na saída da Rede Ômega ativa as três primeiras instruções RISC somente, concluindo a execução

do ciclo de busca na memória variável.

Este trabalho considera, por definição, uma *janela de busca* como sendo o número de instruções RISC paralelas que um processador VLIW é capaz de buscar em cada ciclo de execução. A Figura 26 compara a janela de busca do Corelap com a de outras arquiteturas. O processador MIPS (PATTERSON; HENNESSY, 2013), não sendo um processador VLIW, tem uma janela de busca de tamanho 1. Então ele busca apenas uma instrução por ciclo. O ρ-VEX (WONG; AS; BROWN, 2008), um processador VLIW, é capaz de buscar mais de uma instrução em um ciclo. Mas os desenvolvedores devem determinar suas janelas de busca em tempo de sintetização. Por outro lado, o Processador Reconfigurável de Grão Grosso (*Coarse-Grained Reconfigurable Processor*—CGRP) (OLIVEIRA JÚNIOR, 2015), uma implementação do Corelap, busca instruções variando de 1 a 16 instruções em tempo de execução.

Arquitetura	Características de janela de busca		
	Tamanho	Variabilidade	Configurabilidade
MIPS	1	Fixo	Tempo de sintetização
ρ-VEX	1, 2, ou 4	Variável	Tempo de sintetização
CGRP (Corelap)	1, 2, 3, ..., 16	Variável	Tempo de execução

Figura 26 – Comparação de janelas de busca.

O Corelap tem a capacidade de melhorar a utilização de memória com sua janela variável. Outro ponto é que a arquitetura pode desligar as FUs que não são usadas para reduzir o consumo de energia. Comparada com as outras arquiteturas VLIW que possuem uma CGRA acoplada, a fusão da CGRA com o VLIW reduz também a área em silício e o consumo.

3.6 Compilação

Os compiladores têm muita influência no desempenho dos processadores VLIW. A disposição entre os registradores internos R^{FU_I} e os registradores externos R_I no caminho de dados (Figura 20) adiciona restrições na compilação VLIW no caso do Corelap. Esses registradores preservam, entre ciclos, todos os resultados das FUs. E ambos suprem cada um dos $2N$ operandos das ALUs e as N entradas dos registradores externos R_I através de redes de interconexão. Os registradores externos R_I preservam esses valores resultantes por tantos ciclos quanto os códigos requerem. Mas os registradores internos R^{FU_I} preservam valores de um ciclo para o

outro apenas. O valor ficará preservado, exceto quando a FU não é usada nos ciclos subsequentes.

Os registradores internos R^{FU_I} constituem-se de uma camada entre os registradores externos R_I e as ALUs. Por causa disso, o Corelap permite a atualização dos valores dos registradores externos R_I apenas em um ciclo após as ALUs os gerarem.

O Corelap pode ser homogêneo ou heterogêneo. Ele reserva as primeiras FUs para operações de memória e apenas a primeira instrução RISC paralela para operações de desvio, no caso em que as FUs são heterogêneas. Os compiladores então devem alocar essas instruções nessas devidas posições dentro de uma instrução longa.

Os compiladores podem utilizar muitas técnicas de ILP para incrementar o desempenho do código no Corelap, como fazem muitos trabalhos com CGRAs. Podemos citar: o escalonamento de blocos básicos, de laços ou global, *software pipelining* e/ou *modulo scheduling*. Esta seção descreve, mais adiante, uma aplicação de escalonamento ILP sobre o Corelap.

A janela variável do Corelap impõe apenas uma pequena restrição ao compilador. A instrução longa corrente tem que informar o tamanho da janela da próxima instrução longa. Tal janela facilita outros pontos, por outro lado, como a não necessidade de gerar instruções RISC nulas para completar a janela fixa. O Corelap oferece também a oportunidade de explorar mais ILP, pois pode ter até 16 unidades. Um VLIW é limitado ao máximo de 8, devido ao banco de registradores unificado.

Outra restrição vem do fato do registrador de destino da FU ser fixo. O Corelap é uma fusão das arquiteturas VLIW e CGRA, como mencionado. Além disso, pode-se dizer que o Corelap herda uma característica do escalonamento de instruções, introduzida pelo algoritmo Tomasulo (TOMASULO, 1967). Esse escalonamento renomeia a dependência de dados em função da unidade na qual a operação está escalonada no lugar de usar o registrador. O Corelap exige que os compiladores redirecionem as dependências para a FU ao fixar o registrador destino à FU. Mas o compilador é que será responsável pela alocação dos registradores.

Alocar registradores é um problema importante e clássico de compiladores (HACK; GOOS, 2006). Não é escopo desta dissertação propor uma técnica eficiente para alocação de registradores. Nosso objetivo é mostrar que a restrição do registrador destino fixo em *hardware* não inviabiliza a geração de código no Corelap.

Suponha o código RISC abaixo:

```
add R3, R2, R1; # R3 = R2 + R1
sub R2, R3, 5;  # R2 = R3 - 5
add R1, R1, 3;  # R1 = R1 + 2
```

Um compilador VLIW irá verificar as dependências para agrupar e escalonar as instruções em paralelo. Podemos observar que a primeira e a terceira instruções podem ser agrupadas. A segunda instrução deve executar em um novo bloco devido à dependência em R3. Portanto, o compilador irá gerar:

```
# bloco 1
add R3, R2, R1 # R3 = R2 + R1 - soma
add R1, R1, 3; # R1 = R1 + 2  - contador
# bloco 2
sub R2, R3, 5; # R2 = R3 - 5  - subtração
```

Iremos rotular as três operações como *soma*, *contador* e *subtração* para facilitar a explicação. O VLIW escreve em R1 no final do ciclo. Então não gera problema executar as duas instruções *add*. Suponha uma arquitetura com 4 unidades para ilustrar o compilador para o Corelap. Os registradores de saída serão R1, ..., R4 para as unidades 1, ..., 4, respectivamente. Suponha que os valores iniciais estão nos registradores R1, R2, R3, como ilustra o exemplo do VLIW. Para o Corelap serão gerados dois blocos também. Vamos usar a notação cujas instruções no bloco são alocadas para as unidades 1, 2, 3 e 4, sequencialmente. Uma solução gerada para o Corelap seria:

```
# bloco 1
add R1, R2, R1 # R1 = R2 + R1 - soma     - Executa na unidade 1.
add R2, R1, 3; # R2 = R1 + 2  - contador - Executa na unidade 2.
# bloco 2
sub R1, R1, 5; # R1 = R1 - 5  - subtração - Executa na unidade 1.
```

O registrador R1 passa a armazenar a soma R2 + R1, que antes tinha sido alocada para o registrador R3. A instrução *sub* do bloco 2 irá buscar a dependência de dados através do valor proveniente da unidade 1, agora, através de R1, consequentemente. Já a segunda instrução *add* foi alocada na unidade 2. Portanto, seu resultado será armazenado em R2. Não há necessidade de armazenar R2 pelo fato do valor de R2 já estar sobrescrito no código original após ser lido para fazer R2 + R1. O compilador apenas renomeia registradores. As operações de soma, contador e subtração foram armazenadas nos registradores R3, R1 e R2, respectivamente, no código VLIW. O contador agora está em R2 e a subtração em R1 no Corelap. Porém, o valor da soma não foi preservado. Aqui, temos duas situações que o compilador, com sua visão global, pode resolver.

A primeira situação é quando o valor da soma não é mais consumido pelo restante do código e, portanto, não precisa ser preservado. A segunda situação

ocorre quando é necessário preservar a soma que será consumida por uma instrução posterior. Falha a solução que o Corelap apresenta, nesse caso. A solução proposta nesta dissertação é incluir as instruções de *move* e os registradores adicionais. Ou seja, a simplificação da conexão direta do registrador destino é transferida em parte para o *software* (compilador) e parte para o *hardware* (registradores adicionais).

Já podemos verificar que 16 registradores são poucos se formos avaliar o custo adicional em *hardware* para um Corelap com 16 unidades. E o banco deve ter pelo menos 32 registradores, como qualquer outro processador, seja RISC ou VLIW. Ou seja, o Corelap terá pelo menos 32 registradores, destes 16 são conectados às unidades e 16 são usados como registradores adicionais. A diferença no Corelap é que o banco está distribuído. O outro custo envolvido é a rede de interconexão para permitir a movimentação. O terceiro ponto é a instrução *move*. A codificação é simples e pode-se encapsular pelo menos 2 *move* em uma única sílaba de instrução de 32 *bits*.

O algoritmo de alocação de registradores deve levar em conta essa restrição, do ponto de vista de *software*. Suponha um Corelap com 4 unidades e 8 registradores, em que R1, ..., R4 são dedicados às unidades e R5, ..., R8 são os registradores adicionais para ilustrar uma solução. Uma solução gerada para o Corelap para incluir o armazenamento da soma seria:

```
# bloco 1
add  R1, R2, R1 # R1 = R2 + R1 - soma     - Executa na unidade 1.
add  R2, R1, 3; # R2 = R1 + 2  - contador - Executa na unidade 2.
# bloco 2
sub  R1, R1, 5  # R1 = R1 - 5  - subtração - Executa na unidade 1
move R1, R5;    #                          - Preserva o resultado
                #                            da operação de soma.
```

Teríamos duas instruções apesar do bloco 2 estar com uma operação a mais em um VLIW tradicional mesmo com janela de tamanho 2. O código irá economizar espaço no Corelap com a janela ajustável, nos trechos com pouco paralelismo. Nos trechos com paralelismo, os blocos grandes podem aproveitar o ILP.

A instrução *move* se faz necessária quando precisamos preservar o valor para ser consumido em dois ou mais ciclos de relógio após ser gerado.

Vamos agora ver um outro exemplo em que há operações de desvio e de memória. Suponha o código VLIW:

```
# bloco 1
L1:
```

```
 add  R1, R2, R1  # R1 = R2 + R1  - soma
 add  R2, R1, 3   # R2 = R1 + 3   - soma
 brgt R2, 40, L2; # Se R2 > 40,
                  # então goto L2 - desvio
# bloco 2
 sub  R1, R1, 5;  # R1 = R1 - 5   - subtração
# bloco 3
L2:
 sub R2, R2, 7    # R2 = R2 - 7   - subtração
 stw 0[R2], R1;   # Mem[R2] = R1  - escrita
```

O Corelap aceita operações de memória apenas nas primeiras instruções RISC das instruções longas e operações de desvio apenas na primeira instrução RISC. O compilador deve então ordenar as operações de desvio (bloco 1) e escrita (bloco 3). O código fica assim, já com a renomeação do *hardware*:

```
# bloco 1
L1:
 brgt R2, 40, L2  # Se R2 > 40,
                  # então goto L2 - desvio - Reordenada
 add  R2, R2, R1  # R1 = R2 + R1  - soma   - R1 renomeado pelo
                  #                         hardware.
 add  R3, R1, 3;  # R2 = R1 + 3   - soma   - R2 renomeado pelo
                  #                         hardware.
# bloco 2
 sub  R1, R1, 5;  # R1 = R1 - 5   - subtração
# bloco 3
L2:
 stw 0[R2], R1    # Mem[R2] = R1   - escrita   - Reordenada
 sub R2, R2, 7;   # R2 = R2 - 7    - subtração
```

O *hardware* renomeia a soma e a subtração dos respectivos blocos a partir do segundo registrador fixo. Isso porque o desvio do bloco 1 e a escrita do bloco 3 já ocupam a primeira instrução RISC paralela.

Resta ao compilador completar a renomeação que o *hardware* força e preservar os valores que as instruções longas não adjacentes consomem. Para tanto, o compilador pode montar uma lista de registradores de entrada para cada instrução longa. Essa lista é trivial no caso do bloco 2, já que a única instrução longa que a precede é a do bloco 1. A lista do bloco 2 então é $B2 = \{\emptyset, R1, R2\}$. O símbolo $\emptyset$ significa que

nenhuma instrução RISC utiliza o registrador fixo, mas ocupa o espaço na instrução longa. É o caso do desvio no bloco 1.

Já o bloco 3, por ter etiqueta (*label*), está sujeito a ter uma lista conflituosa de registradores de entrada. A lista da instrução longa desse bloco não se trata dos destinos do bloco 2 somente. Ela tem influência dos destinos das instruções longas do bloco 1 também. Unindo, então, os registradores destino dos blocos 1 e 2 e eliminando as repetições, a lista de registradores de entrada do bloco 3 fica $B_3 = \{\emptyset, R1, R2\} \cup \{R1\} = \{\emptyset, R1, R2\}$. O bloco 1 tem etiqueta também, mas, no exemplo, não tem nenhum desvio que leve a ele.

O compilador pode renomear os registradores fonte das instruções longas e adicionar as instruções *move* para preservar valores. Isso porque ele tem todas as listas de registradores de entrada definidas. Há uma instrução RISC *move* para cada item da lista de entrada de cada instrução longa. E as listas de entrada determinam a renomeação de cada registrador fonte das instruções longas, conforme mostrado a seguir:

```
# bloco 1
L1:
 brgt R2, 40, L2  # Se R2 > 40,
                  # então goto L2 - desvio
 add  R2, R2, R1  # R1 = R2 + R1  - soma
 add  R3, R1, 3;  # R2 = R1 + 3   - soma
# bloco 2
 nop              # nula          - Alocação para fazer correspon-
                  #                 dência à lista de entrada do
                  #                 bloco 3.
 sub  R2, R2, 5   # R1 = R1 - 5   - subtração - R1 destino renomeado
                  #                           - R1 fonte renomeado
 move R2, R3;     # R2            - Preserva o resultado
                  #                 da soma (bloco 1).
# bloco 3
L2:
 stw 0[R2], R1    # Mem[R2] = R1   - escrita
 sub R2, R2, 7    # R2 = R2 - 7    - subtração
 move R1, R2      # R1             - Preserva o resultado
                  #                  da soma (bloco 1).
                  #                  ou da subtração (bloco 2).
 move R2, R3;     # R2             - Preserva o resultado
                  #                  da soma (bloco 1).
```

É importante notar que a ordem das instruções RISC paralelas das instruções longas de desvio devem seguir a ordem das listas de registradores de entrada que elas produzem em suas respectivas instruções longas etiquetadas. Pode ocorrer dessa lista conter instruções RISC que não registram valores em registradores, como o desvio do bloco 1. As instruções longas de desvio devem, mesmo assim, manter a correspondência com a lista que produzem. Elas devem passar a conter ou uma outra instrução RISC que não gere valor ou uma instrução nula na posição das instruções RISC que não registram valores, como representa a instrução nula do bloco 2. Caso contrário, o *hardware* pode causar conflitos ao renomear os registradores destino.

3.7 Qualidades em potencial e restrições

A característica RISC e VLIW do Corelap, junto com a característica CGRA e sua exposição de ILP, permitem aos compiladores terem um melhor potencial de exploração de ILP de seus códigos fonte. A janela variável de busca, nesta proposta, pode reduzir a subalocação de memória e o consumo de energia ao desativar as FUs que não são usadas.

As características RISC e VLIW do Corelap contribuem para sua simplicidade. O Corelap é um processador ILP por si só. Ele não precisa de aceleradores externos para exploração de ILP.

O Corelap utiliza CGRA heterogênea em seu caminho de dados. Uma CGRA com características heterogêneas reduz o custo em relação às CGRAs homogêneas.

O Corelap adiciona algumas restrições aos compiladores, por fim. A Figura 27 apresenta um resumo dessas restrições. Este trabalho descreve detalhes de cada uma dessas restrições, em mais detalhes, nas seções anteriores.

Nesse capítulo mostramos dois exemplos simples para ilustrar uma situação nova gerada pela restrição do registrador fixo. O próximo capítulo apresenta um estudo de caso, expondo mais detalhes.

Restrição	**Descrição**	**Implicações**
Destino fixo	O Corelap tem registradores de destino fixos de resultados das instruções RISC paralelas. Cada FU tem seu próprio registrador destino fixo em tempo real de execução	O compilador deve sempre se referir a esses registradores quando está capturando um valor que uma instrução RISC paralela gera em tempo real. Um compilador deve ter atenção especial às instruções longas etiquetadas, as quais possuem uma lista variável de registradores de entrada em tempo real
Destino volátil	O Corelap sobrescreve os valores destino dos registradores das FUs por um novo valor a cada ciclo. O valor resultante de uma instrução longa precedente, mas não adjacente, será perdido	O compilador pode necessitar desse valor sendo sobrescrito. Se for assim, ele deve então salvar esse valor volátil nos registradores externos, ou mesmo salvá-lo em memória
Heterogêneo	Apenas as primeiras instruções RISC paralelas ativas realizam operações de memória. Somente a primeira e a segunda FUs realizam operações de memória em uma sintetização de memória com 2 operações concorrentes por exemplo. Adicionalmente, apenas a primeira FU realiza operações de desvio	Os compiladores devem alocar as operações de desvio sempre na primeira instrução RISC paralela de uma instrução longa, e alocar operações de memória sempre nas primeiras instruções RISC paralelas de uma instrução longa

Figura 27 – Restrições do modelo Corelap aos compiladores.

4 Estudo de caso

Esta seção apresenta um estudo de caso para mostrar que é possível compilar trechos completos de código, respeitando as novas restrições que o Corelap impõe; trata-se de um laço que realiza uma soma de vetores, um exemplo que abrange todos os casos de restrições de compilação da arquitetura proposta e que, ao mesmo tempo, é pequeno, de facilitada descrição.

A metodologia consiste em gerar o código com ILP com o compilador VEX e propor uma tradução desse código para o equivalente que atenda às restrições do Corelap, simulando o papel do compilador. O código VEX serve de base para comparação e avaliação da solução Corelap também, pois já é uma solução paralela que um compilador VLIW eficiente gera.

4.1 Soma de vetores

A Figura 28 mostra um código em Linguagem C para exemplificar o escalonamento ILP sobre o Corelap. Esse código consiste de uma soma de valores de cada linha de dois vetores em um laço e o armazenamento do resultado em um terceiro vetor na forma $C[i] = A[i] + B[i]$.

```
1 void vecsum (int* a, int* b, int* c, int n)
2 {
3     int i;
4     for (i = 0; i < n; i++)
5         c[i] = a[i] + b[i];
6 }
```

Figura 28 – Código em Linguagem C da soma de vetores.

A Figura 29a mostra um código *assembly* que o compilador VEX gera a partir do código da Figura 28. É o resultado completo de uma compilação da função "sum", da Figura 28, pelo simulador VEX com todas as suas otimizações escalares, mas sem *loop unrolling*. O *loop unrolling* geraria um *assembly* de maior desempenho. No entanto, apresentaria um código mais extenso, dificultando a descrição.

O compilador considera janelas de 16 palavras. Considera, também, duas leituras e/ou escritas consecutivas de memória com latências de dois ciclos. Os pares de dois pontos na Figura 29 separam as instruções longas, enquanto as novas linhas separam as instruções RISC paralelas dentro das instruções longas. O valor do II é 5

```
1  vecsum::
2      sub r6 = r0, r6
3      mov r7 = r3
4      mov r3 = r4
5      mov r4 = r0
6  ;;
7
8
9
10     mov r2 = r6
11     mov r8 = r5
12     mov r5 = r3
13 ;;
14
15
16
17
18 L0:
19     sh2add r6 = r4, r5
20     sh2add r9 = r4, r7
21 ;;
22
23
24
25
26     ldw.d r6 = 0[r6]
27     ldw.d r9 = 0[r9]
28 ;;
29     sh2add r10 = r4, r8
30 ;;
31
32
33     bcmpge r2, r0, L1
34     add r6 = r6, r9
35
36 ;;
37     add r4 = r4, 1
38     add r2 = r2, 1
39     stw 0[r10] = r6
40     goto L0
41 ;;
42
43
44
45 L1:
46     return r1 = r1, 0, l0
47
48 ;;
```

(a) Código *assembly* VEX.

```
1  vecsum::
2      sub I1 = r0, r6
3      mov I2 = r3
4      mov I3 = r4
5      mov I4 = r0
6  ;;
7      nop
8      nop
9      add I3 = I4, 0
10     mov I4 = I1
11     mov I5 = r5
12     mov I6 = I3
13     mov r6 = I1
14     mov r7 = I2
15     mov r3 = I3
16     mov r4 = I3
17 ;;
18 L0:
19     sh2add I1 = I2, I3
20     sh2add I2 = I2, r7
21     mov r4 = I3
22     mov r2 = I4
23     mov r8 = I5
24     mov r5 = I6
25 ;;
26     ldw.d I1 = 0[I1]
27     ldw.d I2 = 0[I2]
28 ;;
29     sh2add I1 = r4, r8
30     mov r6 = I1
31     mov r9 = I2
32 ;;
33     bcmpge r2, r0, L1
34     add I2 = r6, r9
35     mov r10 = I1
36 ;;
37     goto L0
38     stw 0[r10] = I2
39     add I3 = r4, 1
40     add I4 = r2, 1
41     add I4 = r8, 0
42     add I5 = r5, 0
43     mov r6 = I2
44 ;;
45 L1:
46     return r1 = r1, 0, l0
47     mov r6 = I2
48 ;;
```

(b) Simulação do código *assembly* Corelap.

Figura 29 – Tradução do código *assembly* VEX de uma soma de vetores para o código *assembly* Corelap equivalente.

já que iteração do laço L_0 do código contém 5 instruções longas. O laço processa 11 instruções RISC a cada iteração.

A tradução do *assembly* VEX para o *assembly* Corelap deve seguir as restrições dessa segunda arquitetura. A Figura 29b mostra um *assembly* com as restrições do Corelap equivalente ao *assembly* VEX da Figura 29a. O Corelap exige registradores destino apenas em instruções "move". Mas a Figura 29b simula a escrita dos registradores internos aos PEs com registradores anotados pela letra "I" para simulação no

compilador VEX e melhor descrição do exemplo. Vale enfatizar que o Corelap realiza a escrita dos registradores internos em *hardware*, ocultando essa funcionalidade do compilador. Um código *assembly* Corelap real utilizaria registradores destino — e operadores "=" — apenas em instruções "move".

As FUs do Corelap sobrescrevem os valores dos registradores internos aos PEs a cada ciclo. O código da Figura 29b numera os registradores "I", de acordo com suas posições dentro da instrução longa, para simular a operação dos registradores internos. Por exemplo, os registradores destino I_1 e I_2, nas linhas 26 e 27 da Figura 29b, destacadas na Figura 30, correspondem aos registradores destino r_6 e r_9 nas mesmas linhas da Figura 29a, respectivamente. O Corelap renomeia, em *hardware*, esses registradores destino r_6 e r_9, compulsoriamente.

```
18 L0:
19     sh2add r6 = r4, r5
20     sh2add r9 = r4, r7
21 ;;
22
23
24
25
26     ldw.d r6 = 0[r6]
27     ldw.d r9 = 0[r9]
28 ;;
29     sh2add r10 = r4, r8
30 ;;
31
32
```

```
18 L0:
19     sh2add I1 = I2, I3
20     sh2add I2 = I2, r7
21     mov r4 = I3
22     mov r2 = I4
23     mov r8 = I5
24     mov r5 = I6
25 ;;
26     ldw.d I1 = 0[I1]
27     ldw.d I2 = 0[I2]
28 ;;
29     sh2add I1 = r4, r8
30     mov r6 = I1
31     mov r9 = I2
32 ;;
```

Figura 30 – Destaque da simulação da renomeação em *hardware* dos registradores internos.

Os compiladores, para a arquitetura executar o código corretamente, devem renomear os registradores fonte que referenciam os registradores destino, renomeados pelo Corelap em *hardware*. As instruções RISC paralelas podem requerer um valor que um PE gera no ciclo anterior. Por exemplo, o elemento PE_2 produz e salva um valor que corresponde ao registrador r_6 na linha 34 do código VEX da Figura 29a, com destaque na Figura 31. A instrução na linha 39 do código VEX consome esse valor. Mas o Corelap renomeia o registrador r_6, que o código Corelap simula com o registrador I_2 da linha 34, compulsoriamente. O Corelap renomeia o registrador r_6 na linha 34 do código Corelap. Então, a instrução da linha 38 do código Corelap deve consumir o valor do registrador interno I_2, no qual está o real valor do registrador r_6. Os compiladores devem renomear registradores fonte, dessa forma, toda vez que uma instrução RISC puder consumir um valor que outra instrução RISC produz em um ciclo imediatamente anterior.

Os compiladores podem salvar os valores dos registradores internos do Corelap nos seus registradores externos. Isso se forem utilizar os resultados dos PEs de uma instrução longa precedente, mas não adjacente, a qual ocorre em dois ou mais ciclos

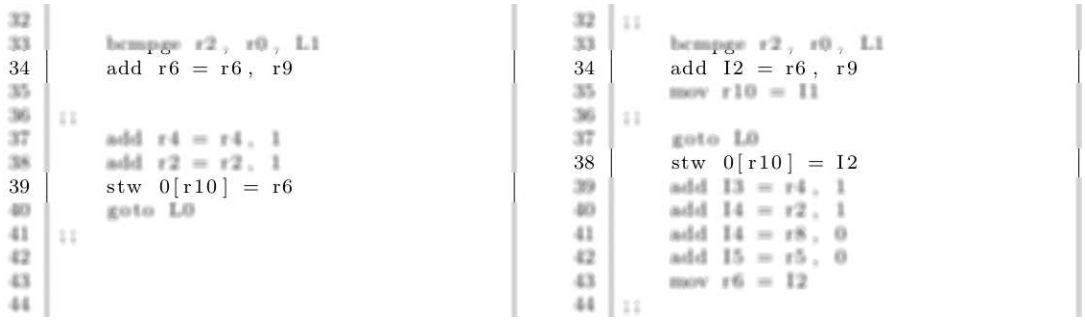


Figura 31 – Destaque da simulação da renomeação em *software* dos registradores fonte referentes os registradores internos.

após a produção do valor. A Figura 29b representa os registradores externos com a letra "r". Os compiladores devem utilizar a operação "move" para salvar esses valores. O Corelap permite que apenas operações do tipo "move" movam os valores dos registradores internos para os registradores externos. No exemplo, os "moves" das linhas 30 e 31 do código Corelap da Figura 29b, destacado na Figura 32, salvam os valores referentes aos registradores r_6 e r_9 das linhas 26 e 27 do código VEX, agora presentes nos registradores I_1 e I_2, nas mesmas linhas do código Corelap, respectivamente. Os compiladores podem utilizar os registradores regulares "r", após salvar, para consumir um valor que uma instrução RISC produz em uma instrução longa precedente, mas não adjacente; isso sem qualquer outro esforço de renomeação. Mas podem evitar de salvar esses valores caso eles não os utilizem mais em ciclos futuros.

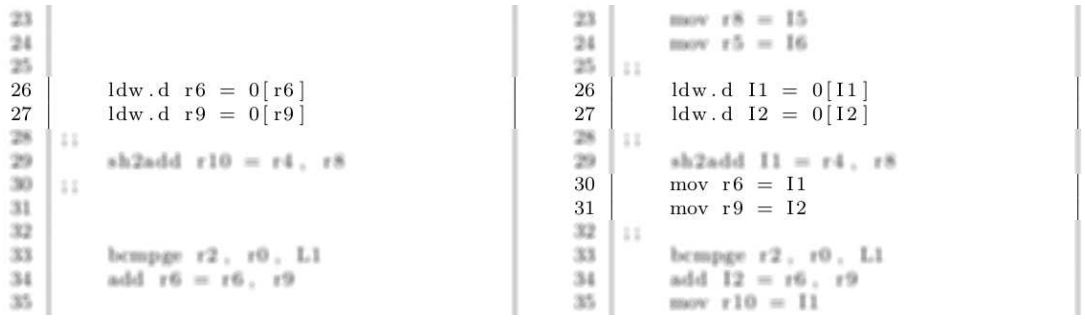


Figura 32 – Destaque da simulação da persistência de valores nos registradores externos.

É importante notar que esses valores podem ficar desatualizados apesar do compilador poder registrar valores nos registradores "r" para utilizar em dois ou mais ciclos posteriores. Por exemplo, o compilador Corelap preserva o valor correspondente ao registrador destino r_6 da linha 34 do código VEX da Figura 29a — renomeado para o registrador I_2 do código Corelap na Figura 29b — no registrador r_6 com a instrução "move" da linha 43, com mostra o destaque na Figura 33. O *assembly* VEX

utiliza esse valor na linha 19 do código Corelap. Esse valor já está desatualizado no próximo ciclo (linha 19) apesar do registrador r_6 preservar o valor na linha 43 do código Corelap. O valor atual do registrador r_6 está no registrador I_1 da linha 19 do código Corelap, e não mais em r_6. Resta ao compilador Corelap continuar a utilizar, na linha 19 do código Corelap, não o registrador r_6, mas o registrador I_1. Trata-se de consumir valores produzidos em um ciclo anterior, já definido nesta seção. O compilador inclusive pode eliminar a instrução "move" da linha 43 sem prejuízo para a corretude da execução nesse tipo de situação em específico. É uma situação em que um mesmo registrador se repete na lista de registradores destino de dois ou mais instruções longas consecutivamente adjacentes.

```
18 L0:
19     sh2add r6 = r4, r5
20     sh2add r9 = r4, r7
21 ;;
22
23
24
25
26     ldw.d r6 = 0[r6]
27     ldw.d r9 = 0[r9]
28 ;;
29     sh2add r10 = r4, r8
30 ;;
31
32
33     bcmpge r2, r0, L1
34     add r6 = r6, r9
35
36 ;;
37     add r4 = r4, 1
38     add r2 = r2, 1
39     stw 0[r10] = r6
40     goto L0
41 ;;
42
43
44
45 L1:
```

```
18 L0:
19     sh2add I1 = I2, I3
20     sh2add I2 = I2, r7
21     mov r4 = I3
22     mov r2 = I4
23     mov r8 = I5
24     mov r5 = I6
25 ;;
26     ldw.d I1 = 0[I1]
27     ldw.d I2 = 0[I2]
28 ;;
29     sh2add I1 = r4, r8
30     mov r6 = I1
31     mov r9 = I2
32 ;;
33     bcmpge r2, r0, L1
34     add I2 = r6, r9
35     mov r10 = I1
36 ;;
37     goto L0
38     stw 0[r10] = I2
39     add I3 = r4, 1
40     add I4 = r2, 1
41     add I4 = r8, 0
42     add I5 = r5, 0
43     mov r6 = I2
44 ;;
45 L1:
```

Figura 33 – Destaque de registrador preservado, mas desatualizado.

A heterogeneidade, pelas operações de desvio e operações de memória, é outra restrição que o Corelap impõe aos compiladores. O Corelap permite esses tipos de operações apenas nas primeiras instruções RISC paralelas de uma instrução longa. No caso do desvio, a primeira instrução RISC paralela possui essa ação, apenas. Os compiladores devem alocar essas instruções RISC nas instruções longas do Corelap, adequadamente. A Figura 29b, com destaque na Figura 34, mostra essa alocação sobre operações de desvio e memória, nas linhas 26, 27, 33, 37, 38 e 46. Vale notar que a tradução coloca a instrução de desvio na linha 37 na primeira posição da instrução longa compreendida nas linha 37–43 da Figura 29b. Resta à instrução de memória da linha 38 ficar na próxima posição.

Uma observação importante é que as instruções "move" são além das 16 instruções que os PEs podem executar em um trecho com muito paralelismo. Ou

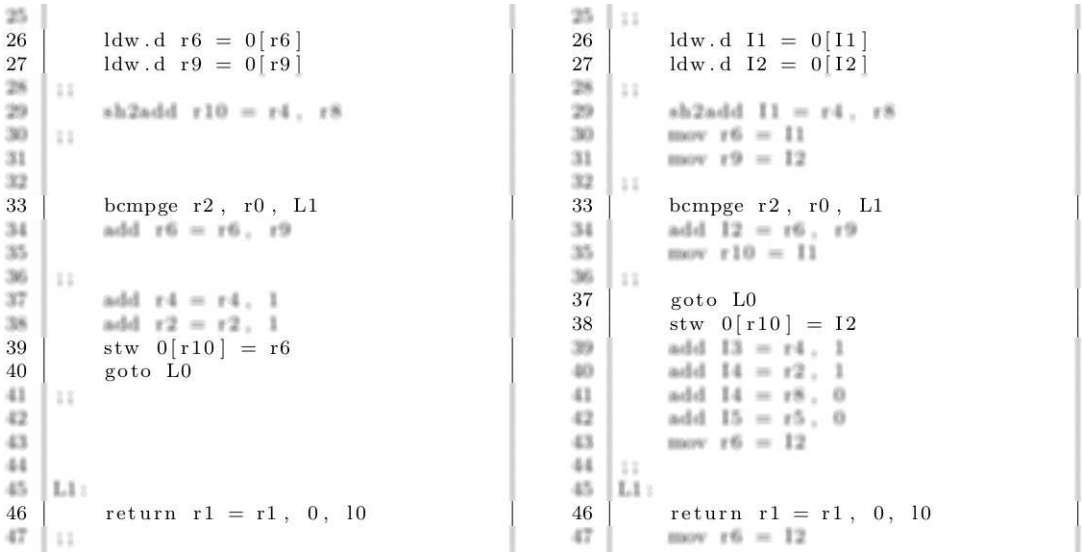


Figura 34 – Destaque no posicionamento das instruções de desvio e memória: heterogeneidade.

seja, o Corelap pode executar 16 operações em PEs e ainda 16 moves com uma única instrução. Portanto, apesar do Corelap gerar restrições, ele também gera novas possibilidades para os compiladores explorarem o paralelismo.

A tradução do código VEX da Figura 29a para o código Corelap da Figura 29b mostra que é possível compilar com o registrador destino fixo. Porém, o binário VEX é ponto de partida. É possível dizer então que a metodologia é mais voltada para implementar um mecanismo de tradução binária já usando o código legado para VEX do que recompilar um código fonte C para o Corelap. Ambos os códigos têm 5 instruções e não há ganho de desempenho com o Corelap, para esse exemplo. O problema é que seria necessário fazer uso de *loop unrolling* ou *modulo scheduling* para gerar ganho de desempenho e tirar mais proveito do paralelismo. Mas isso deixaria o código muito grande para explicar uma opção de geração de código Corelap.

É possível otimizar o exemplo de soma de vetores se for aplicado o *unrolling* do laço e, ou *modulo scheduling*, tanto para o VLIW quanto para o Corelap. Outo ponto é a restrição de operações de memória. Esse exemplo terá ganhos se for possível realizar mais de um acesso de leitura e escrita em paralelo.

4.1.1 Lista de registradores destino de instruções longas VLIW etiquetadas

Como um outro item sobre as restrições de tradução de código *assembly* no Corelap, os compiladores devem resolver um problema de correspondência entre instruções longas que possuem etiquetas (*labels*) e suas instruções longas preceden-

temente adjacentes, em tempo real. O código VEX da Figura 29a apresenta três instruções longas etiquetadas, por exemplo, duas com as etiquetas simples L_0 e L_1, nas linhas 18–20 e 45–46, respectivamente, e a outra com a etiqueta de procedimento "vecsum", nas linhas 1–5.

Instruções longas precedentemente adjacentes de uma instrução longa etiquetada correspondem a instruções que podem buscar essa instrução etiquetada na execução do código. Podem ser as instruções longas de desvio que referenciam a instrução etiquetada ou podem ser as instruções longas que não tenham desvio incondicional para outras instruções, mas que estão alocadas imediatamente anterior à instrução longa etiquetada. Como exemplo, a instrução longa etiquetada L_0, nas linhas 18–20 do código VEX da Figura 29a, melhor ilustrado na Figura 35, possui duas instruções precedentemente adjacentes. Uma é a instrução longa das linhas 37–40 do código VEX. Há o desvio da linha 40 que a busca. Esse é um desvio incondicional que referencia a instrução etiquetada L_0, mas poderia ser um desvio condicional com referência à etiqueta L_0 também. A outra instrução é a das linhas 10–12 do código VEX. Essa instrução não é um desvio incondicional que busca outra instrução, mas está em uma posição imediatamente anterior à instrução etiquetada L_0.

```
5      mov r4 = r0
6   ;;
7
8
9
10     mov r2 = r6
11     mov r8 = r5
12     mov r5 = r3
13  ;;
14
15
16
17
18  L0:
19     sh2add r6 = r4, r5
20     sh2add r9 = r4, r7
21  ;;
22
23
24
25
```

```
5      mov I4 = r0
6   ;;
7      nop
8      nop
9      add I3 = I4, 0
10     mov I4 = I1
11     mov I5 = r5
12     mov I6 = I3
13     mov r6 = I1
14     mov r7 = I2
15     mov r3 = I3
16     mov r4 = I3
17  ;;
18  L0:
19     sh2add I1 = I2, I3
20     sh2add I2 = I2, r7
21     mov r4 = I3
22     mov r2 = I4
23     mov r8 = I5
24     mov r5 = I6
25  ;;
```

```
35
36  ;;
37     add r4 = r4, 1
38     add r2 = r2, 1
39     stw 0[r10] = r6
40     goto L0
41  ;;
42
43
44
45  L1:
```

```
35     mov r10 = I1
36  ;;
37     goto L0
38     stw 0[r10] = I2
39     add I3 = r4, 1
40     add I4 = r2, 1
41     add I4 = r8, 0
42     add I5 = r5, 0
43     mov r6 = I2
44  ;;
45  L1:
```

Figura 35 – Destaque de instruções longas etiquetadas e suas precedentemente adjacentes.

Cada instrução longa possui uma lista de registradores destino, como toda

instrução. Trata-se de todos os registradores nos quais uma instrução longa escreve. Analogamente, a lista de registradores fonte corresponde a todos os registradores que uma instrução longa lê. Por exemplo, a lista de registradores destino da instrução longa nas linhas 37–40 do código VEX (Figura 35) corresponde aos registradores $\{r_4, r_2\}$ e a lista de registradores fonte é $\{r_4, r_2, r_6, r_{10}\}$.

Como este trabalho descreve em seções anteriores, os registradores I são uma restrição da arquitetura Corelap e os compiladores não os podem modificar. O Corelap define as posições dos registradores I de acordo com a posição da instrução RISC paralela dentro da instrução longa VLIW. Então, a primeira instrução RISC paralela pode salvar seu valor no registrador I_1 somente, a segunda, no registrador I_2, e assim por diante. O Corelap força todas as operações a serem uma renomeação de registradores em cada ciclo.

Então, a lista de registradores destino corresponde aos registradores internos, no caso do Corelap, que seus PEs escrevem durante uma operação. Deve sempre existir uma determinação entre a lista de registradores destino das instruções longas precedentemente adjacentes do *assembly* VEX e essa lista de registradores destino que o Corelap renomeia compulsoriamente. Essa determinação é necessária para renomear os registradores fonte, que irão consumir os registradores que o Corelap renomeia. Como a lista de registradores destino da instrução longa das linhas 37–40 do código VEX (Figura 35) é $\{r_4, r_2\}$, por exemplo, o Corelap transforma essa lista em $\{I_3, I_4\}$. Essa última lista é a posição de cada elemento na lista de registradores destino do código VEX após o posicionamento dos operadores de desvio e memória. O compilador reserva as posições 1 e 2 do código Corelap para a operação de desvio da linha 40 e a operação de memória da linha 39 do código VEX.

A correspondência das listas de registradores destino entre o código VEX e o que o Corelap força é trivial quando a renomeação de registradores acontece em uma instrução longa não etiquetada. Instruções longas não etiquetadas possuem uma única instrução longa precedentemente adjacente, mesmo em tempo real de execução. Elas então possuem uma lista constante de registradores destino para consumir da única instrução longa que a busca, a sua instrução precedentemente adjacente. Os compiladores podem determinar a renomeação dos registradores fonte dessas instruções, facilmente, pois já estão com essa lista constante. A instrução longa da linha 29 não possui etiqueta, no exemplo da Figura 29a. Então ela precisa somente da lista de registradores destino da instrução das linhas 26–27, que o Corelap transforma de $\{r_6, r_9\}$ para $\{I_1, I_2\}$, para renomear seus registradores fonte $\{r_4, r_8\}$. Nenhuma renomeação de registrador fonte é necessária, nesse caso, pois a lista de registradores fonte da instrução da linha 29 não tem interseção com a lista de registradores destino das linhas 26–27.

No entanto, essa trivialidade não ocorre nas instruções longas etiquetadas. As instruções longas etiquetadas possuem mais de uma instrução longa precedentemente adjacente em tempo real de execução, geralmente. Então a instrução longa etiquetada se depara com mais de uma lista de registradores destino renomeados, obrigatoriamente, uma para cada instrução longa precedentemente adjacente. Por exemplo, a instrução das linhas 18–20 da Figura 35 é etiquetada com L_0. Ela possui duas instruções longas precedentemente adjacentes, a das linhas 10–12 e a das linhas 37–40. As listas de registradores destino dessas duas instruções são $\{r_2, r_8, r_5\}$ e $\{r_4, r_2\}$, respectivamente. Mas essas listas se transformam, respectivamente, em $\{I_1, I_2, I_3\}$ e $\{I_1, I_2\}$ no Corelap.

Os compiladores podem se deparar com uma indeterminação de buscar, em tempo de compilação, o registrador correto I_i de uma instrução longa etiquetada precedentemente adjacente. Por exemplo, o Corelap renomearia o registrador r_2 da linha 10 para o registrador I_1, caso não houvesse reordenação do código VEX, na Figura 35. Mas renomearia r_2 da linha 38 para o registrador I_2 ao mesmo tempo. Não é o caso da instrução etiquetada das linhas 18–20, mas se ela consumisse o valor r_2, o compilador não saberia determinar se o valor de r_2 estaria no registrador I_1 ou no registrador I_2. Os compiladores precisam dessa determinação para realizar a necessária renomeação de registradores fonte.

Uma maneira intuitiva de resolver esse problema em tempo de compilação é sempre manter constante a lista de registradores destino de todas as instruções longas precedentemente adjacentes das instruções etiquetadas. Os compiladores podem então unir as listas de registradores destino de todas as possíveis instruções longas precedentemente adjacentes de tempo real de execução para compor uma única e constante lista de registradores destino para cada instrução longa etiquetada. Por exemplo, as linhas 19–20 do código VEX da Figura 35 apresentam a instrução longa etiquetada identificada por L_0. Como dito, ambas as instruções longas nas linhas 10–12 e nas linhas 37–40 do código VEX podem fluir para essa instrução longa de etiqueta L_0 em tempo de execução.

A instrução longa de etiqueta L_0 do código VEX da Figura 35 tem, após aplicar as restrições de posição de operações de memória e de desvio do Corelap sobre esse código, duas listas de registradores destino para consumir de suas instruções longas precedentemente adjacentes. Essas listas são $U = \{r_2, r_8, r_5\}$ e $V = \{\emptyset, \emptyset, r_4, r_2\}$, as quais a tradução extrai das linhas 10–12 e 37–40 da Figura 29a, respectivamente. O símbolo $\emptyset$ corresponde às operações de desvio e de memória, estas não escrevem nos registradores internos do Corelap, mas ocupam espaço de instruções RISC paralelas nas instruções longas. Uma união dos conjuntos U e V produz um conjunto de registradores destino que deve ser comum a cada lista de registradores destino das

instruções longas etiquetadas envolvidas. Então, o compilador pode produzir um conjunto como o $W = U \cup V = \{\emptyset, \emptyset, r_4, r_2, r_8, r_5\}$ a partir da Figura 35. Não é necessário e nem correto repetir os registradores destino no conjunto W, pois o compilador VEX utiliza apenas uma referência de cada destino dentro das instruções longas para renomear os registradores fonte.

Os compiladores podem, com uma constante lista de registradores destino para consumir, reordenar e ajustar as instruções RISC paralelas dessas instruções longas precedentemente adjacentes. É necessário que todas as instruções longas precedentemente adjacentes às instruções longas etiquetadas tenham a mesma ordem de registradores destino entre si. De outra forma, a lista de registradores destino que o Corelap força e que as instruções longas etiquetadas precisam consumir deixam de ser iguais. As linhas 7–16 e 37–43 do código Corelap da Figura 35 exemplifica essa reordenação e ajuste. Essas linhas apresentam a mesma lista de registradores destino, representada pelo conjunto W acima. As operações "nop" nas linhas 7–8 representam os valores vazios do conjunto W, que necessitam de correspondência também.

As operações "add" das linhas 41 e 42 do código Corelap da Figura 35 apenas movimentam valores dos registradores r_8 e r_5, respectivamente, para os registradores internos que correspondem a eles no conjunto W. Esses registradores não fazem parte da instrução longa correspondente no código VEX, mas o compilador necessita deles para compor a lista de registradores destino correspondente ao conjunto W. O mesmo ocorre com a operação "add" da linha 9. Como mencionado, o Corelap reserva suas operações de "move" apenas para movimentações entre os registradores internos e os registradores externos. Por isso a opção de utilizar a operação "add" para mover os valores nas linhas 9, 41 e 42.

Os compiladores são capazes de renomear os registradores fonte e preservar os registradores nas instruções longas etiquetadas com a construção dessa lista de registradores destino constante. As linhas 19–24 do código Corelap da Figura 35 exemplificam essa renomeação e preservação finais.

O Apêndice A possui o exemplo completo da simulação da soma de vetores em código *assembly* VEX, o qual este trabalho adapta para o Corelap. Há o CJPEG também no Apêndice A, um outro exemplo. O trabalho extrai esses exemplos do código C do Vecsum e do CJPEG que o Anexo A mostra. A simulação dos códigos Vecsum e CJPEG no Apêndice A estão livres de erros de compilação. E eles geram os mesmos resultados que os exemplos de código Vecsum e CJPEG no Anexo A geram.

Finalmente, o compilador poderia ter uma visão global se ele fosse desenvolvido para o Corelap partindo de uma representação intermediária ao invés de fazer uma tradução do código VEX binário para o código Corelap, como a seção anterior mostra.

Nessa visão global, poderia haver um mecanismo de alocação de registradores já considerando a restrição do registrador destino fixo e utilizando o potencial de ILP do Corelap, por exemplo, com o *modulo scheduling*. Com isso, seria possível explorar melhor o paralelismo e gerar um código eficiente. Portanto, é possível executar códigos com paralelismo no Corelap e trabalhos futuros podem mostrar que ainda é possível simplificar mais o código gerado e explorar o paralelismo com um compilador para Corelap.

5 Resultados de simulação

Este capítulo apresenta os resultados das simulações realizadas sobre o Corelap. São medidas de desempenho, de ocupação e de exigências de operação extraídos de dois exemplos: Vecsum e CJPEG.

A Tabela 1 mostra o número de instruções RISC e o desempenho em IPC dos exemplos que este trabalho explora. Esses dados vêm da compilação e simulação dos programas do Anexo A pelo VEX com configurações de otimização "-H0" e "-O3", que aplicam todas as otimizações escalares que o VEX oferece, mas sem *loop unrolling*. A janela é de tamanho 16, as operações de leitura em memória possuem latência de dois ciclos e a memória aceita duas leituras e duas escritas consecutivas. A opção de não aplicar o *loop unrolling* é pelo fato do código *assembly* VEX ficar extenso para uma tradução semimanual do código *assembly* VEX para o código *assembly* Corelap simulado.

O número de ciclos para executar todo o código no Corelap se mantém estável com relação ao número de ciclos no VEX, com um aumento de 0,001% para o Vecsum e de 0,279% para o CJPEG. Essa estabilidade indica que o Corelap possui uma equivalência de tempo com o VEX para resolver os mesmos dois problemas. Tal equivalência se explica pelo fato de ambas as soluções conterem o mesmo número de instruções longas em cada parte dos códigos em *assembly*. A Tabela 2 indica o número total de instruções longas dos dois exemplos, 52 em cada.

O IPC do Corelap apresenta um aumento de 1,010% e 0,943% em comparação com o VEX, com o Vecsum e o CJPEG, respectivamente, com referência à Tabela 1. O aumento do IPC, junto da manutenção de ciclos entre os exemplos, indicam um

Tabela 1 – Número de instruções RISC e ciclos executados e Instruções por Ciclo (IPC) medidos em simulação de exemplos.

Exemplo	Instruções	Ciclos	IPC
	VEX		
Vecsum	4.911.869	3.308.412	1,485
CJPEG	212.926	167.392	1,272
	Corelap		
Vecsum	4.961.060	3.308.457	1,500
CJPEG	215.613	167.859	1,284

Tabela 2 – Número de instruções longas e número de instruções RISC em exemplos simulados com janela de tamanho 16, indicando o tamanho dos códigos *assembly*.

Exemplo	Instruções longas	Instruções RISC	Média de janelas[1]	Máximo de janelas
		VEX		
Vecsum				
Completo	52	82	1,577	4
Laço	6	13	2,167	4
CJPEG				
Completo	162	311	1,920	4
Laço 1	33	83	2,515	4
Laço 2	32	82	2,563	4
		Corelap		
Vecsum				
Completo	52	131	2,519	8
Laço	6	28	4,667	8
CJPEG				
Completo	162	496	3,062	7
Laço 1	33	135	4,091	7
Laço 2	32	136	4,250	7

(1) Média de janelas obtida da divisão do número de instruções longas pelo número de instruções RISC.

aumento no número de instruções RISC executadas. De fato, há um aumento de 0,010% e 0,278% do total de instruções RISC executadas do Corelap em relação ao VEX, para o Vecsum e o CJPEG, respectivamente. Esse aumento de instruções executadas equivale às instruções RISC "move", necessárias para corrigir a renomeação que o Corelap força na execução. Essas instruções possuem características paralelas. Então, não reduzem o IPC.

No entanto, as instruções "move" provocam um aumento na janela de instruções. A Tabela 2 mostra que o VEX apresenta uma janela máxima de 4 em todas as situações. Já o Corelap apresenta um máximo de janelas que vão de 8 a 7 instruções RISC, quase o dobro, em média, do VEX em janela fixa. A arquitetura tem largura de 16 palavras, no caso dos exemplos, o que não afeta o desempenho. Mas esse aumento

Tabela 3 – Linhas estimadas de memória de instrução a partir da simulação dos códigos *assembly* de exemplos, com memória variável de largura de 16 instruções.

Exemplo	Linhas de memória	Ganho sobre o VEX
	VEX	
Vecsum	52	×
CJPEG	162	×
	Corelap	
Preenchimento parcial[1]		
Vecsum	11	4,727
CJPEG	36	4,500
Preenchimento total[2]		
Vecsum	4	13,00
CJPEG	11	14,73

(1) Linhas estimadas de memória ao utilizar a técnica de preenchimento parcial de espaços em branco no final das linhas de memória.
(2) Linhas estimadas de memória ao utilizar a técnica de preenchimento total de espaços em branco no final das linhas de memória.

de janela pode diminuir o IPC das soluções em arquiteturas de menos palavras ou em códigos Corelap — com as instruções "move" — que podem exigir mais de 16 instruções RISC paralelas.

A Tabela 2 surge a partir dos códigos *assembly* dos exemplos. Os processadores de arquivo texto extraem o número de instruções RISC em cada instrução longa desses códigos *assembly*. A partir desses dados é possível sintetizar o restante das informações. Ela apresenta o tamanho dos códigos *assembly* tanto do total dos exemplos quanto de seus principais laços.

Quanto à ocupação da memória de instruções, a simulação do Corelap indica um ganho médio de 4,614× com um aproveitamento parcial das linhas de memória e de 13,87× com um aproveitamento total em comparação com a simulação do VEX com janela fixa, de acordo com a Tabela 3.

A Tabela 3 apresenta a quantidade de instruções longas e de instruções RISC alcançada em simulações com os exemplos Vecsum e CJPEG para arquiteturas VEX e Corelap de 16 PEs. Traz ainda a quantidade média e máxima de janelas entre todas as instruções longas que esses exemplos requerem para funcionar nas arquiteturas.

O ganho em ocupação de memória acontece mesmo com a adição das instruções "move" e indica o potencial que a janela dinâmica do Corelap possui para reduzir os requerimentos de tamanho de memória. A janela variável parcial é capaz de utilizar os espaços vazios no final da maioria das linhas de memória e a janela variável total todos os espaços vazios.

6 Conclusão

A combinação da exposição de paralelismo das arquiteturas VLIW com a distribuição de registradores das arquiteturas CGRA possibilita um modelo de processador VLIW de janela variável de instruções, de um *hardware* simples e sem necessidade de um coprocessador, que busca a exploração do paralelismo e a redução de consumo de energia.

Os registradores das CGRAs, interligados por redes de interconexão, são capazes de substituir o tradicional banco de registradores centralizado que, devido às escritas e leituras concorrentes, são de alto custo e sem escalabilidade. Este trabalho apresenta uma alternativa descentralizada e mais escalável. O novo processador pode ser classificado como um VLIW com largura ajustável de até 16 palavras.

Há destaque, além da proposta do banco de registradores distribuídos em uma CGRA, para a definição arquitetural com um registrador fixo na saída das unidades funcionais. Essa definição simplifica a arquitetura ao eliminar coprocessadores, apesar de aumentar as restrições de compilação. No entanto, os compiladores podem contornar essas restrições, nas quais os registradores adicionais de movimentação auxiliam a tarefa. Este trabalho apresenta, ao introduzir a nova arquitetura, soluções para essas novas restrições que trabalhos futuros podem trabalhar durante a compilação e a eliminação de coprocessadores aliada ao paralelismo das CGRAs trazem possibilidades de um processador VLIW de maior eficiência energética sem perda de desempenho.

Um mecanismo eficiente de busca de instruções, que comporte instruções longas de tamanho variável e sem a necessidade de alinhamento, implica em uma redução do consumo e possibilita a desativação de unidades ociosas na execução. Esse mecanismo diminui os requerimentos de memórias de instruções sem a necessidade de compactar o código.

Os experimentos deste trabalho mostram que o novo processador Corelap pode alcançar uma redução média de 4,614× no tamanho exigido em memória em comparação com um processador VLIW tradicional, o qual possui janela fixa.

Trabalhos futuros

Apesar da possibilidade de um compilador Corelap utilizar os operadores "move", tais operadores ocupam espaço na memória de instruções. Além disso, as instruções "move" não são contabilizadas na janela máxima de 16 instruções. Uma outra proposta de instrução longa pode reduzir o espaço ocupado.

Dentre um conjunto de características de projeto VLIW, Fisher, Faraboschi e Young (2005) apresenta uma que diz para "expor o poder e esconder a fraqueza". Então, no meio de tantas alternativas, uma seria o *hardware* realizar essas operações "move" de maneira transparente, com um pequeno custo adicional. Análises preliminares indicam que, para isso, seriam necessários uma inclusão de $N + 1$ registradores no mecanismo de controle do Corelap e de um campo de $log(N) + 1$ *bits* no formato ISA de instruções para endereçar os registradores destino. A falta das operações de "move" tem pouco potencial para interferir na exposição de ILP.

Os compiladores são essenciais para a utilização das arquiteturas VLIW. Este trabalho apresenta sugestões de um compilador para o Corelap. Mas o Corelap carece de uma implementação completa de um compilador, ou um módulo de um, para uma validação mais abrangente e para possíveis aplicações em outros projetos. Este trabalho indica as principais restrições para guiar a construção desse compilador.

Há a possibilidade de dispor as redes de interconexão da CGRA de maneiras diferentes e continuar aproveitando da distribuição de registradores. Por exemplo, ao invés de utilizar os registradores externos junto com a rede de interconexão IN_3 da Figura 20 para preservar valores dos registradores, poderia haver uma rede para rotear as instruções RISC já nas entradas de arranjo de FUs, homogêneas ou heterogêneas. Dessa maneira, cada instrução RISC ficaria alocada na FU que corresponderia ao seu registrador destino, o que evitaria a renomeação em *hardware*, evitando assim a necessidade de registradores externos e diminuindo o esforço de compilação VLIW.

Referências

BENINI, L. et al. A power modeling and estimation framework for vliw-based embedded systems. In: CITESEER. *Proc. Int. Workshop on Power And Timing Modeling, Optimization and Simulation PATMOS*. [S.l.], 2001. v. 1, p. 2–3. Citado 2 vezes nas páginas 6 e 11.

BERNSTEIN, A. J. Analysis of programs for parallel processing. *Electronic Computers, IEEE Transactions on*, IEEE, n. 5, p. 757–763, 1966. Citado na página 17.

CONG, J. et al. A fully pipelined and dynamically composable architecture of cgra. In: IEEE. *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*. [S.l.], 2014. p. 9–16. Citado na página 9.

FERREIRA, R. et al. A run-time modulo scheduling by using a binary translation mechanism. In: IEEE. *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*. [S.l.], 2014. p. 75–82. Citado 6 vezes nas páginas 2, 6, 11, 12, 15 e 21.

FERREIRA, R. et al. A dynamic modulo scheduling with binary translation: Loop optimization with software compatibility. *Journal of Signal Processing Systems*, Springer, p. 1–22, 2015. Citado 3 vezes nas páginas 6, 20 e 21.

FERREIRA, R. et al. A just-in-time modulo scheduling for virtual coarse-grained reconfigurable architectures. In: IEEE. *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*. [S.l.], 2013. p. 188–195. Citado 5 vezes nas páginas 2, 9, 11, 12 e 15.

FERREIRA, R. et al. An fpga-based heterogeneous coarse-grained dynamically reconfigurable architecture. In: ACM. *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*. [S.l.], 2011. p. 195–204. Citado 3 vezes nas páginas 15, 18 e 21.

FISHER, J. A.; FARABOSCHI, P.; YOUNG, C. *Embedded computing: a VLIW approach to architecture, compilers and tools*. [S.l.]: Elsevier, 2005. Citado 5 vezes nas páginas 5, 7, 11, 12 e 60.

GRAMMATIKAKIS, M. D.; HSU, D. F.; KRAETZL, M. *Parallel system interconnections and communications*. [S.l.]: CRC press, 2000. Citado na página 33.

HACK, S.; GOOS, G. Optimal register allocation for ssa-form programs in polynomial time. *Information Processing Letters*, Elsevier, v. 98, n. 4, p. 150–155, 2006. Citado na página 37.

HAMZEH, M.; SHRIVASTAVA, A.; VRUDHULA, S. Regimap: Register-aware application mapping on coarse-grained reconfigurable architectures (cgras). In: ACM. *Proceedings of the 50th Annual Design Automation Conference*. [S.l.], 2013. p. 18. Citado 3 vezes nas páginas 2, 8 e 14.

HARTENSTEIN, R. A decade of reconfigurable computing: a visionary retrospective. In: IEEE PRESS. *Proceedings of the conference on Design, automation and test in Europe*. [S.l.], 2001. p. 642–649. Citado na página 14.

IMEC. *Adres Processor*. 2016. Disponível em: <http://www2.imec.be/content/user/File/Brochures/GR2011_Leaflet%20ADRES.pdf>. Acesso em: 12 de maio de 2016. Citado na página 2.

JOST, T. T. *RefreeMIPS*: A cgra-based mips architecture. 2014. 51 p. Monografia (Dissertação Graduação) — UFRGS, Porto Alegre-RS, 2014. Citado 5 vezes nas páginas 2, 18, 19, 20 e 21.

KOENIG, R. et al. A scalable microarchitecture design that enables dynamic code execution for variable-issue clustered processors. In: IEEE. *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. [S.l.], 2011. p. 150–157. Citado 2 vezes nas páginas 11 e 12.

KUMAR, S.; JHA, S. B.; SINGH, R. K. Graphical process unit a new era. In: JETIR. *Journal of Emerging Technologies and Innovative Research*. [S.l.], 2014. v. 1, n. 6 (November-2014). Citado na página 2.

LAM, M. Software pipelining: An effective scheduling technique for vliw machines. In: ACM. *ACM Sigplan Notices*. [S.l.], 1988. v. 23, n. 7, p. 318–328. Citado na página 3.

LEE, M.-H. et al. Design and implementation of the morphosys reconfigurable computing processor. *Journal of VLSI signal processing systems for signal, image and video technology*, Springer, v. 24, n. 2-3, p. 147–164, 2000. Citado na página 14.

LIN, C. H.; XIE, Y.; WOLF, W. Lzw-based code compression for vliw embedded systems. In: IEEE. *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*. [S.l.], 2004. v. 3, p. 76–81. Citado 2 vezes nas páginas 11 e 12.

MEI, B. et al. Architecture exploration for a reconfigurable architecture template. *Design & Test of Computers, IEEE*, IEEE, v. 22, n. 2, p. 90–101, 2005. Citado na página 14.

MEI, B. et al. Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In: *Field Programmable Logic and Application*. [S.l.]: Springer, 2003. p. 61–70. Citado 8 vezes nas páginas 2, 3, 6, 9, 12, 13, 14 e 15.

NAGARAJAN, R. et al. Static placement, dynamic issue (spdi) scheduling for edge architectures. In: IEEE COMPUTER SOCIETY. *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. [S.l.], 2004. p. 74–84. Citado na página 11.

OLIVEIRA JÚNIOR, G. F. de. *VLIW Based Coarse-grained Reconfigurable Processor (CGRP)*: An instruction set implementation. 2015. 25 p. Monografia (Dissertação Graduação) — UFV, Viçosa-MG, 2015. Citado 5 vezes nas páginas 26, 29, 30, 32 e 36.

PATTERSON, D. A.; HENNESSY, J. L. *Computer organization and design: the hardware/software interface*. [S.l.]: Newnes, 2013. Citado 3 vezes nas páginas 23, 32 e 36.

SANKARALINGAM, K. et al. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. In: IEEE. *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*. [S.l.], 2003. p. 422–433. Citado 4 vezes nas páginas 14, 18, 20 e 21.

SHEN, J. P.; LIPASTI, M. H. *Modern processor design: fundamentals of superscalar processors*. [S.l.]: Waveland Press, 2013. Citado na página 2.

SINGH, H. et al. Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *Computers, IEEE Transactions on*, IEEE, v. 49, n. 5, p. 465–481, 2000. Citado na página 9.

TESSIER, R.; POCEK, K.; DEHON, A. Reconfigurable computing architectures. *Proceedings of the IEEE*, IEEE, v. 103, n. 3, p. 332–354, 2015. Citado na página 2.

TOMASULO, R. M. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, IBM, v. 11, n. 1, p. 25–33, 1967. Citado na página 37.

WARTER, N. J. et al. Enhanced modulo scheduling for loops with conditional branches. In: IEEE COMPUTER SOCIETY PRESS. *ACM SIGMICRO Newsletter*. [S.l.], 1992. v. 23, n. 1-2, p. 170–179. Citado na página 3.

WONG, S.; ANJAM, F.; NADEEM, F. Dynamically reconfigurable register file for a softcore vliw processor. In: EUROPEAN DESIGN AND AUTOMATION ASSOCIATION. *Proceedings of the Conference on Design, Automation and Test in Europe*. [S.l.], 2010. p. 969–972. Citado na página 11.

WONG, S.; AS, T. V.; BROWN, G. ρ-vex: A reconfigurable and extensible softcore vliw processor. In: IEEE. *ICECE Technology, 2008. FPT 2008. International Conference on*. [S.l.], 2008. p. 369–372. Citado 4 vezes nas páginas 5, 6, 11 e 36.

Apêndices

APÊNDICE A – Exemplos de simulação de código Corelap

A.1 Simulação de código Corelap sobre o simulador VEX: Vecsum

```
1  .comment ""
2  .comment "Copyright (C) 1990-2010 Hewlett-Packard Company"
3  .comment "VEX C compiler version 3.43 (20110131 release)"
4  .comment ""
5  .comment "-dir /opt/vex -DVERBOSE -O1 -ms -DVERBOSE -fmm=../local-
       ↪ benchmark/vex1-16reg.mm"
6  .sversion 3.43
7  .rta 2
8  .section .bss
9  .align 32
10 .section .data
11 .align 32
12 .equ _?1TEMPLATEPACKET.9, 0x0
13 .equ _?1TEMPLATEPACKET.18, 0x0
14 .section .text
15 .proc
16 .entry caller, sp=$r0.1, rl=$l0.0, asize=-64, arg($r0.3:u32,$r0.4:
       ↪ s32)
17 .endp
18 .section .bss
19 .section .data
20 _?1STRINGPACKET.1:
21      .data1 37
22      .data1 100
23      .data1 32
24      .data1 0
25 _?1STRINGPACKET.2:
26      .data1 10
27      .data1 0
28 .equ ?2.1?2scratch.0, 0x0
29 .equ ?2.1?2ras_p, 0x10
30 .equ ?2.1?2spill_p, 0x14
31 .section .data
32 .section .text
33 .equ ?2.1?2auto_size, 0x40
34 .section .text
35 .proc
36 .entry caller, sp=$r0.1, rl=$l0.0, asize=0, arg($r0.3:u32,$r0.4:
       ↪ u32,$r0.5:u32,$r0.6:s32)
37 vecsum::
38 .trace 3
39          c0    sub $r1.1 = $r0.0, $r0.6 # d=6
40          c0    mov $r1.2 = $r0.3 # d=7
41          c0    mov $r1.3 = $r0.4 # d=3
42          c0    mov $r1.4 = $r0.0 # d=4
43 ;;
44 # s = { 6, 7, 3, 4 }
45 # d = { null, null, 2, 4, 5, 8 }
46 #             nop
47 #             nop
48          c0    mov $r1.3 = $r1.1 # d=2 s=6
49 #             nop
50          c0    mov $r1.5 = $r1.3 # d=5 s=3
51          c0    mov $r1.6 = $r0.5 # d=8
52
53          c0    mov $r0.6 = $r1.1
```

```
54          c0    mov $r0.7 = $r1.2
55          c0    mov $r0.3 = $r1.3
56          c0    mov $r0.4 = $r1.4
57  ;;
58  .trace 1
59  L2?3:
60  # s1 = { 2, 8, 5 }
61  # s2 = { null, null, 4, 2 }
62  # => s = { null, null, 2, 4, 5, 8 }
63          c0    cmplt $r1.1 = $r1.3, $r0.0 # d=3 s=2
64          c0    sh2add $r1.2 = $r1.4, $r1.5 # d=6 s=4 s=5
65          c0    sh2add $r1.3 = $r1.4, $r0.7 # d=9 s=4
66
67          c0    mov $r0.2 = $r1.3
68          c0    mov $r0.4 = $r1.4
69          c0    mov $r0.5 = $r1.5
70          c0    mov $r0.8 = $r1.6
71  ;;
72  # s = { 3, 6, 9 }
73          c0    ldw.d $r1.1 = 0[$r1.2] # d=6 s=6(waw)
74          c0    ldw.d $r1.2 = 0[$r1.3] # d=9 s=9(waw)
75
76          c0    mov $r0.3 = $r1.1
77          c0    mov $r0.6 = $r1.2
78          c0    mov $r0.9 = $r1.3
79  ;;
80  # s = { 6, 9 }
81          c0    sh2add $r1.1 = $r0.4, $r0.8 # d=10
82
83          c0    mov $r0.6 = $r1.1
84          c0    mov $r0.9 = $r1.2
85  ;;
86  # s = { 10 }
87          c0    brf $r0.3, L3?3 # d=null
88          c0    add $r1.2 = $r0.6, $r0.9 # d=6
89
90          c0    mov $r0.10 = $r1.1
91  ;;
92  # s = { null, 6 }
93  # d = { null, null, 2, 4, 5, 8 }
94          c0    goto L2?3 # d=null
95          c0    stw 0[$r0.10] = $r1.2 # d=null s=6
96          c0    add $r1.3 = $r0.2, 1 # d=2
97          c0    add $r1.4 = $r0.4, 1 # d=4
98          c0    add $r1.5 = $r0.5, $r0.0 #! d=5
99          c0    add $r1.6 = $r0.8, $r0.0 #! d=8
100
101         c0    mov $r0.6 = $r1.2
102 ;;
103 .trace 4
104 L3?3:
105 .return ret()
106 # s = { null, 6 }
107         c0    return $r0.1 = $r0.1, (0x0), $l0.0
108
109         c0    mov $r0.6 = $r1.2
110 ;;
111 .endp
112 .section .bss
113 .section .data
114 .section .data
115 .section .text
116 .equ ?2.2?2auto_size, 0x0
117 .section .text
118 .proc
119 .entry caller, sp=$r0.1, rl=$l0.0, asize=-8224, arg()
120 main::
121 .trace 1
122         c0    add $r0.1 = $r0.1, (-0x2020)
123 ;;
124         c0    stw 0x2010[$r0.1] = $l0.0
125         c0    add $r0.5 = $r0.1, 0x10
126         c0    mov $r0.6 = 2048
127 ;;
```

```
128         c0    stw 0x2014[$r0.1] = $r0.5
129         c0    mov $r0.4 = (B + 0)
130         c0    mov $r0.3 = (A + 0)
131 ;;
132 .call vecsum, caller, arg($r0.3:u32,$r0.4:u32,$r0.5:u32,$r0.6:s32)
        ↪ , ret()
133         c0    call $l0.0 = vecsum
134 ;;
135         c0    ldw $r0.3 = 0x2014[$r0.1]
136         c0    mov $r0.4 = 2048
137 ;;
138         c0    ldw $l0.0 = 0x2010[$r0.1]
139 ;;
140         c0    mov $r0.3 = $r0.0
141               xnop 2
142 ;;
143 .return ret($r0.3:s32)
144         c0    return $r0.1 = $r0.1, (0x2020), $l0.0
145 ;;
146 .endp
147 .section .bss
148 .section .data
149 .equ ?2.3?2scratch.0, 0x0
150 .equ _?1PACKET.3, 0x10
151 .equ ?2.3?2ras_p, 0x2010
152 .equ ?2.3?2spill_p, 0x2014
153 .section .data
154 .section .text
155 .equ ?2.3?2auto_size, 0x2020
156 .section .bss
157 .section .data
158 .skip 2
159 A::
160 ...
161 B::
162 ...
163 .section .data
164 .import _impure_ptr
165 .type _impure_ptr,@object
166 .section .text
167 .import vecsum
168 .type vecsum,@function
169 .import vecsum_print
170 .type vecsum_print,@function
171 .import fprintf
172 .type fprintf,@function
```

A.2 Simulação de código Corelap sobre o simulador VEX: CJPEG

```
1  .comment ""
2  .comment "Copyright (C) 1990-2010 Hewlett-Packard Company"
3  .comment "VEX C compiler version 3.43 (20110131 release)"
4  .comment ""
5  .comment "-dir /opt/vex -DVERBOSE -H0 -O3 -ms -fmm=../local-
       ↪ benchmark/vex4-16reg.mm"
6  .sversion 3.43
7  .rta 2
8  .section .bss
9  .align 32
10 .section .data
11 .align 32
12 .equ _?1TEMPLATEPACKET.18, 0x0
13 .section .text
14 .proc
15 .entry caller, sp=$r0.1, rl=$l0.0, asize=-64, arg($r0.3:u32)
16 # _____boundary_____
17 print_data::
```

```
18 .trace 6
19         c0    add $r1.1 = $r0.1, (-0x40)
20         c0    mov $r1.2 = (~0x7)
21 ;;
22         c0    stw 0x10[$r1.1] = $l0.0
23
24         c0    mov $r0.1 = $r1.1 # Renaming adjustment
25         c0    mov $r0.2 = $r1.2 # Renaming adjustment
26 ;;
27         c0    stw 0x38[$r0.1] = $r0.57
28 ;;
29         c0    stw 0x34[$r0.1] = $r0.58
30 ;;
31         c0    stw 0x30[$r0.1] = $r0.59
32 ;;
33         c0    stw 0x2c[$r0.1] = $r0.60
34 ;;
35         c0    stw 0x28[$r0.1] = $r0.61
36 ;;
37         c0    stw 0x24[$r0.1] = $r0.62
38 ;;
39         c0    stw 0x20[$r0.1] = $r0.63
40 ;;
41         c0    stw 0x1c[$r0.1] = $r0.3
42 ;;
43         c0    stw 0x18[$r0.1] = $r0.2
44 ;;
45 L0?3:
46 .trace 3
47         c0    cmplt $r1.1 = $r0.2, $r0.0
48         c0    mov $r1.2 = (~0x7)
49         c0    mov $r1.3 = $r0.3
50 ;;
51         c0    stw 0x14[$r0.1] = $r1.3
52
53         c0    mov $b0.0 = $r1.1 # Renaming adjustment
54         c0    mov $r0.7 = $r1.2 # Renaming adjustment
55         c0    mov $r0.6 = $r1.3 # Renaming adjustment
56 ;;
57         c0    brf $b0.0, L1?3
58         c0    stw 0x3c[$r0.1] = $r0.7
59 ;;
60 L2?3:
61 .trace 1
62         c0    ldw.d $r1.1 = ((_impure_ptr + 0) + 0)[$r0.0]
63         c0    cmplt $r1.2 = $r0.7, $r0.0
64 ;;
65         c0    ldw.d $r1.1 = 0[$r0.6]
66         c0    mov $r1.2 = (_?1STRINGPACKET.1 + 0)
67
68         c0    mov $r0.2 = $r1.1 # Renaming adjustment
69         c0    mov $b0.0 = $r1.2 # Renaming adjustment
70 ;;
71         c0    brf $b0.0, L3?3
72
73         c0    mov $r0.5 = $r1.1 # Renaming adjustment
74         c0    mov $r0.4 = $r1.2 # Renaming adjustment
75 ;;
76         c0    ldw $r1.1 = 8[$r0.2]
77               xnop 1
78 ;;
79 .call fprintf, caller, arg($r0.3:u32,$r0.4:u32,$r0.5:s32), ret($r0
   ↪ .3:s32)
80         c0    call $l0.0 = fprintf
81
82         c0    mov $r0.3 = $r1.1 # Renaming adjustment
83 ;;
84         c0    ldw $r1.1 = 0x14[$r0.1]
85 ;;
86         c0    ldw $r1.1 = 0x3c[$r0.1]
87               xnop 1
88
89         c0    mov $r0.6 = $r1.1 # Renaming adjustment
90 ;;
```

```
91          c0    add $r1.1 = $r0.6, 4
92
93          c0    mov $r0.7 = $r1.1 # Renaming adjustment
94    ;;
95          c0    add $r1.1 = $r0.7, 1
96
97          c0    mov $r0.6 = $r1.1 # Renaming adjustment
98    ;;
99    .trace 2
100         c0    stw 0x3c[$r0.1] = $r1.1
101
102         c0    mov $r0.7 = $r1.1 # Renaming adjustment
103   ;;
104         c0    goto L2?3
105         c0    stw 0x14[$r0.1] = $r0.6
106   ;;
107   L3?3:
108   .trace 4
109         c0    ldw $r1.1 = ((_impure_ptr + 0) + 0)[$r0.0]
110         c0    mov $r1.2 = (_?1STRINGPACKET.2 + 0)
111               xnop 2
112   ;;
113         c0    ldw $r1.1 = 8[$r1.1]
114               xnop 1
115
116         c0    mov $r0.6 = $r1.1 # Renaming adjustment
117         c0    mov $r0.4 = $r1.2 # Renaming adjustment
118   ;;
119   .call fprintf, caller, arg($r0.3:u32,$r0.4:u32), ret($r0.3:s32)
120         c0    call $l0.0 = fprintf
121
122         c0    mov $r0.3 = $r1.1 # Renaming adjustment
123   ;;
124         c0    ldw $r1.1 = 0x1c[$r0.1]
125   ;;
126         c0    ldw $r1.1 = 0x18[$r0.1]
127               xnop 1
128
129         c0    mov $r0.3 = $r1.1 # Renaming adjustment
130   ;;
131         c0    add $r1.1 = $r0.3, 32
132
133         c0    mov $r0.2 = $r1.1 # Renaming adjustment
134   ;;
135         c0    stw 0x1c[$r0.1] = $r1.1
136         c0    add $r1.2 = $r0.2, 1
137
138         c0    mov $r0.3 = $r1.1 # Renaming adjustment
139   ;;
140         c0    goto L0?3
141         c0    stw 0x18[$r0.1] = $r1.2
142
143         c0    mov $r0.2 = $r1.2 # Renaming adjustment
144   ;;
145   L1?3:
146   .trace 5
147         c0    ldw $r1.1 = ((_impure_ptr + 0) + 0)[$r0.0]
148         c0    mov $r1.2 = (_?1STRINGPACKET.3 + 0)
149               xnop 2
150   ;;
151         c0    ldw $r1.1 = 8[$r1.1]
152               xnop 1
153
154         c0    mov $r0.2 = $r1.1 # Renaming adjustment
155         c0    mov $r0.4 = $r1.2 # Renaming adjustment
156   ;;
157   .call fprintf, caller, arg($r0.3:u32,$r0.4:u32), ret($r0.3:s32)
158         c0    call $l0.0 = fprintf
159
160         c0    mov $r0.3 = $r1.1 # Renaming adjustment
161   ;;
162         c0    ldw $l0.0 = 0x10[$r0.1]
163   ;;
164         c0    ldw $r1.1 = 0x20[$r0.1]
```

```
;;
        c0    ldw $r1.1 = 0x24[$r0.1]

        c0    mov $r0.63 = $r1.1 # Renaming adjustment
;;
        c0    ldw $r1.1 = 0x28[$r0.1]

        c0    mov $r0.62 = $r1.1 # Renaming adjustment
;;
        c0    ldw $r1.1 = 0x2c[$r0.1]

        c0    mov $r0.61 = $r1.1 # Renaming adjustment
;;
        c0    ldw $r1.1 = 0x30[$r0.1]

        c0    mov $r0.60 = $r1.1 # Renaming adjustment
;;
        c0    ldw $r1.1 = 0x34[$r0.1]

        c0    mov $r0.59 = $r1.1 # Renaming adjustment
;;
        c0    ldw $r1.1 = 0x38[$r0.1]
              xnop 1

        c0    mov $r0.58 = $r1.1 # Renaming adjustment
;;
.return ret()
        c0    return $r0.1 = $r0.1, (0x40), $l0.0

        c0    mov $r0.57 = $r1.1 # Renaming adjustment
;;
.endp
.section .bss
.section .data
_?1STRINGPACKET.2:
    .data1 10
    .data1 0
.skip 2
_?1STRINGPACKET.1:
    .data1 37
    .data1 100
    .data1 32
    .data1 0
_?1STRINGPACKET.3:
    .data1 10
    .data1 0
.equ ?2.1?2scratch.0, 0x0
.equ ?2.1?2ras_p, 0x10
.equ ?2.1?2spill_p, 0x14
.equ _?1TEMPLATEPACKET.19, 0x0
.section .data
.section .text
.equ ?2.1?2auto_size, 0x40
.section .text
.proc
.entry caller, sp=$r0.1, rl=$l0.0, asize=-288, arg($r0.3:u32)
# _____boundary_____
fill_data::
.trace 5
        c0    add $r1.1 = $r0.1, (-0x120)
        c0    mov $r1.2 = 256
;;
        c0    stw 0x110[$r1.1] = $l0.0
        c0    add $r1.2 = $r1.1, 0x10

        c0    mov $r0.1 = $r1.1 # Renaming adjustment
        c0    mov $r0.5 = $r1.2 # Renaming adjustment
;;
        c0    stw 0x114[$r0.1] = $r0.3

        c0    mov $r0.4 = $r1.2 # Renaming adjustment
;;
        c0    stw 0x118[$r0.1] = $r0.4
        c0    mov $r1.2 = (_?1PACKET.4 + 0)
```

```
239 ;;
240 .call _bcopy, caller, arg($r0.3:u32,$r0.4:u32,$r0.5:s32), ret()
241         c0    call $l0.0 = _bcopy
242
243         c0    mov $r0.3 = $r1.2 # Renaming adjustment
244 ;;
245         c0    ldw $r1.1 = 0x114[$r0.1]
246         c0    mov $r1.2 = $r0.0
247         c0    mov $r1.3 = (~0x7)
248 ;;
249         c0    ldw $r1.1 = 0x118[$r0.1]
250
251         c0    mov $r0.3 = $r1.1 # Renaming adjustment
252         c0    mov $r0.6 = $r1.2 # Renaming adjustment
253         c0    mov $r0.4 = $r1.3 # Renaming adjustment
254 ;;
255         c0    ldw $r1.1 = 0x110[$r0.1]
256
257         c0    mov $r0.8 = $r1.1 # Renaming adjustment
258 ;;
259         c0    add $r1.1 = $r0.3, 32
260
261         c0    mov $r0.10 = $r1.1 # Renaming adjustment
262 ;;
263 #             nop # @labeled
264         c0    add $r1.2 = $r0.4, 0 # @labeled
265         c0    add $r1.3 = $r1.1, 0 # @labeled !! @move
266 #       c0    goto L4?3 # @labeled
267 #       c0    add $r0.4 = $r0.7, 1 # @labeled
268 #       c0    mov $r0.5 = $r1.1 # @labeled
269
270         c0    mov $r0.3 = $r1.1 # Renaming adjustment
271 ;;
272 L4?3:
273 .trace 3
274         c0    cmplt $r1.1 = $r1.2, $r0.0 # @labeled
275         c0    add $r1.2 = $r0.6, 32
276         c0    mov $r1.3 = (~0x7)
277         c0    mov $r1.4 = $r0.8
278 #       c0    goto L4?3 # @labeled
279 #       c0    add $r0.4 = $r0.7, 1 # @labeled
280 #       c0    mov $r0.5 = $r1.1 # @labeled
281
282         c0    mov $r0.4 = $r1.2 # Renaming adjustment
283         c0    mov $r0.5 = $r1.3 # Renaming adjustment @labeled
284 ;;
285         c0    add $r1.1 = $r1.2, $r0.8
286         c0    mov $r1.2 = $r0.4
287         c0    mov $r1.3 = $r0.6
288
289         c0    mov $b0.0 = $r1.1 # Renaming adjustment
290         c0    mov $r0.2 = $r1.3 # Renaming adjustment
291         c0    mov $r0.9 = $r1.4 # Renaming adjustment
292 ;;
293         c0    brf $b0.0, L5?3
294
295         c0    mov $r0.3 = $r1.1 # Renaming adjustment
296         c0    mov $r0.7 = $r1.2 # Renaming adjustment
297         c0    mov $r0.8 = $r1.3 # Renaming adjustment
298 ;;
299 L6?3:
300 .trace 1
301         c0    cmplt $r1.1 = $r0.2, $r0.0
302         c0    sh2add $r1.2 = $r0.2, $r0.3
303         c0    sh2add $r1.3 = $r0.2, $r0.5
304         c0    add $r1.4 = $r0.2, 1
305 ;;
306         c0    ldw.d $r1.1 = 0[$r1.2]
307
308         c0    mov $b0.0 = $r1.1 # Renaming adjustment
309         c0    mov $r0.6 = $r1.3 # Renaming adjustment
310         c0    mov $r0.2 = $r1.4 # Renaming adjustment
311 ;;
312         c0    brf $b0.0, L7?3
```

```
313 |
314 |         c0    mov $r0.4 = $r1.1 # Renaming adjustment
315 | ;;
316 | ;;
317 |         c0    goto L6?3
318 |         c0    stw 0[$r0.6] = $r0.4
319 | ;;
320 | L7?3:
321 | .trace 4
322 |         c0    add $r1.1 = $r0.5, 32
323 |         c0    add $r1.2 = $r0.8, 32
324 |         c0    mov $r1.3 = $r0.9
325 | ;;
326 |         c0    goto L4?3
327 |         c0    add $r1.2 = $r0.7, 1 # d=4 @labeled
328 |         c0    add $r1.3 = $r1.1, 0 # d=5 @labeled
329 | #       c0    goto L4?3 # @labeled
330 | #       c0    add $r0.4 = $r0.7, 1 # @labeled
331 | #       c0    mov $r0.5 = $r1.1 # @labeled
332 |
333 |         c0    mov $r0.5 = $r1.1 # Renaming adjustment
334 |         c0    mov $r0.6 = $r1.2 # Renaming adjustment
335 |         c0    mov $r0.8 = $r1.3 # Renaming adjustment
336 | ;;
337 | L5?3:
338 | .trace 6
339 |         c0    stw 0x110[$r0.1] = $r0.10
340 |         c0    mov $l0.0 = $r0.10
341 |               xnop 2
342 | ;;
343 | .return ret()
344 |         c0    return $r1.1 = $r0.1, (0x120), $l0.0 # d=1
345 | ;;
346 | .endp
347 | .section .bss
348 | .section .data
349 | .skip 2
350 | _?1PACKET.4:
351 |     .data4 -20238
352 |     .data4 22081
353 |     .data4 -859
354 |     .data4 -20826
355 |     .data4 20738
356 |     .data4 30280
357 |     .data4 -16590
358 |     .data4 -17599
359 |     .data4 17079
360 |     .data4 -15435
361 |     .data4 -27179
362 |     .data4 10282
363 |     .data4 28835
364 |     .data4 24754
365 |     .data4 9158
366 |     .data4 22156
367 |     .data4 20955
368 |     .data4 -16750
369 |     .data4 6772
370 |     .data4 30121
371 |     .data4 9521
372 |     .data4 2303
373 |     .data4 -8329
374 |     .data4 -21892
375 |     .data4 9092
376 |     .data4 22700
377 |     .data4 2507
378 |     .data4 -32688
379 |     .data4 16912
380 |     .data4 -3186
381 |     .data4 1649
382 |     .data4 29442
383 |     .data4 18897
384 |     .data4 -31978
385 |     .data4 -24152
386 |     .data4 6867
```

```
387       .data4 -1698
388       .data4 24796
389       .data4 22036
390       .data4 15382
391       .data4 -23407
392       .data4 -5143
393       .data4 -7105
394       .data4 5429
395       .data4 -13157
396       .data4 -30714
397       .data4 -5183
398       .data4 7799
399       .data4 -14696
400       .data4 1590
401       .data4 5153
402       .data4 -5175
403       .data4 -28875
404       .data4 -3176
405       .data4 -27065
406       .data4 -19783
407       .data4 -13244
408       .data4 8210
409       .data4 13066
410       .data4 -29100
411       .data4 5025
412       .data4 14715
413       .data4 344
414       .data4 23922
415   .equ ?2.2?2scratch.0, 0x0
416   .equ _?1PACKET.3, 0x10
417   .equ ?2.2?2ras_p, 0x110
418   .equ ?2.2?2spill_p, 0x114
419   .equ _?1TEMPLATEPACKET.20, 0x0
420   .section .data
421   .section .text
422   .equ ?2.2?2auto_size, 0x120
423   .equ _?1TEMPLATEPACKET.9, 0x0
424   .section .text
425   .proc
426   .entry caller, sp=$r0.1, rl=$l0.0, asize=-64, arg($r0.3:u32)
427   # _____boundary_____
428   jpeg_fdct_islow::
429   .trace 7
430           c0    add $r1.1 = $r0.1, (-0x40)
431           c0    mov $r1.2 = $r0.3
432           c0    mov $r1.3 = 7
433   ;;
434           c0    stw 0xc[$r1.1] = $r0.57
435
436           c0    mov $r0.1 = $r1.1 # Renaming adjustment
437           c0    mov $r0.2 = $r1.2 # Renaming adjustment
438           c0    mov $r0.5 = $r1.3 # Renaming adjustment
439   ;;
440           c0    stw 0x10[$r0.1] = $r0.58
441   ;;
442           c0    stw 0x14[$r0.1] = $r0.59
443   ;;
444           c0    stw 0x18[$r0.1] = $r0.60
445   ;;
446           c0    stw 0x1c[$r0.1] = $r0.61
447   ;;
448           c0    stw 0x20[$r0.1] = $r0.62
449   ;;
450           c0    stw 0x24[$r0.1] = $r0.63
451   ;;
452           c0    stw 0x8[$r0.1] = $r0.3
453    #      c0    stw 0x8[$r0.1] = $r0.3 # @labeled
454    #      c0    goto L8?3 # @labeled
455   ;;
456   L8?3:
457   .trace 1
458           c0    ldw.d $r1.1 = 0[$r0.2]
459           c0    cmpge $r1.2 = $r0.5, $r0.0
460           c0    add $r1.3 = $r0.5, -1
```

```
461 #       c0    stw 0x8[$r0.1] = $r0.3 # @labeled
462 #       c0    goto L8?3 # @labeled
463 ;;
464         c0    ldw.d $r1.1 = 28[$r0.2]
465
466         c0    mov $r0.3 = $r1.1 # Renaming adjustment
467         c0    mov $b0.0 = $r1.2 # Renaming adjustment
468         c0    mov $r0.5 = $r1.3 # Renaming adjustment
469 ;;
470         c0    brf $b0.0, L9?3
471         c0    ldw.d $r0.7 = 12[$r0.2]
472 #       c0    brf $b0.0, L9?3 # @labeled
473 #       c0    ldw.d $r0.7 = 12[$r0.2] # @labeled
474
475         c0    mov $r0.6 = $r1.1 # Renaming adjustment
476 ;;
477         c0    ldw $r1.1 = 16[$r0.2]
478 ;;
479         c0    ldw $r1.1 = 4[$r0.2]
480         c0    add $r1.2 = $r0.6, $r0.3
481         c0    sub $r1.3 = $r0.3, $r0.6
482
483         c0    mov $r0.8 = $r1.1 # Renaming adjustment
484 ;;
485         c0    ldw $r1.1 = 24[$r0.2]
486         c0    mpyl $r1.2 = $r1.3, 12299
487
488         c0    mov $r0.10 = $r1.1 # Renaming adjustment
489         c0    mov $r0.6 = $r1.2 # Renaming adjustment
490         c0    mov $r0.9 = $r1.3 # Renaming adjustment
491 ;;
492         c0    ldw $r1.1 = 8[$r0.2]
493         c0    add $r1.2 = $r0.8, $r0.7
494         c0    sub $r1.3 = $r0.7, $r0.8
495
496         c0    mov $r0.3 = $r1.1 # Renaming adjustment
497         c0    mov $r0.11 = $r1.2 # Renaming adjustment
498 ;;
499         c0    ldw $r1.1 = 20[$r0.2]
500         c0    add $r1.2 = $r0.9, $r1.3
501         c0    mpyl $r1.3 = $r1.3, 2446
502
503         c0    mov $r0.13 = $r1.1 # Renaming adjustment
504         c0    mov $r0.8 = $r1.2 # Renaming adjustment
505         c0    mov $r0.12 = $r1.3 # Renaming adjustment
506 ;;
507         c0    add $r1.1 = $r0.3, $r0.10
508         c0    sub $r1.2 = $r0.10, $r0.3
509         c0    sub $r1.3 = $r0.6, $r0.8
510         c0    add $r1.4 = $r0.9, $r0.12
511
512         c0    mov $r0.14 = $r1.1 # Renaming adjustment
513         c0    mov $r0.7 = $r1.2 # Renaming adjustment
514         c0    mov $r0.15 = $r1.3 # Renaming adjustment
515 ;;
516         c0    add $r1.1 = $r0.6, $r0.8
517         c0    add $r1.2 = $r1.2, $r0.12
518         c0    mpyl $r1.3 = $r1.4, -7373
519
520         c0    mov $r0.3 = $r1.1 # Renaming adjustment
521         c0    mov $r0.16 = $r1.2 # Renaming adjustment
522         c0    mov $r0.10 = $r1.3 # Renaming adjustment
523 ;;
524         c0    add $r1.1 = $r0.14, $r0.13
525         c0    sub $r1.2 = $r0.13, $r0.14
526         c0    mpyl $r1.3 = $r1.2, -16069
527
528         c0    mov $r0.6 = $r1.1 # Renaming adjustment
529         c0    mov $r0.4 = $r1.3 # Renaming adjustment
530 ;;
531         c0    stw 0x0[$r0.1] = $r0.5
532         c0    add $r1.2 = $r0.16, $r1.2
533         c0    add $r1.3 = $r0.4, 2048
534
```

```
535           c0    mov $r0.14 = $r1.1 # Renaming adjustment
536           c0    mov $r0.8 = $r1.2 # Renaming adjustment
537           c0    mov $r0.12 = $r1.3 # Renaming adjustment
538 ;;
539           c0    add $r1.1 = $r0.16, $r0.8
540           c0    add $r1.2 = $r0.7, $r1.2
541           c0    mpyl $r1.3 = $r0.16, 25172
542
543           c0    mov $r0.4 = $r1.3 # Renaming adjustment
544 ;;
545           c0    mpyl $r1.1 = $r1.2, 9633
546           c0    mpyl $r1.2 = $r1.1, -20995
547
548           c0    mov $r0.13 = $r1.3 # Renaming adjustment
549 ;;
550           c0    sub $r1.1 = $r0.3, $r0.14
551           c0    add $r1.2 = $r0.9, $r0.8
552           c0    mpyl $r1.3 = $r0.8, 16819
553
554           c0    mov $r0.7 = $r1.1 # Renaming adjustment
555           c0    mov $r0.16 = $r1.2 # Renaming adjustment
556 ;;
557           c0    stw 0x4[$r0.1] = $r0.6
558           c0    add $r1.2 = $r0.16, 2048
559           c0    add $r1.3 = $r0.7, $r0.12
560
561           c0    mov $r0.5 = $r1.1 # Renaming adjustment
562           c0    mov $r0.9 = $r1.2 # Renaming adjustment
563           c0    mov $r0.8 = $r1.3 # Renaming adjustment
564 ;;
565           c0    add $r1.1 = $r0.15, $r1.3
566           c0    add $r1.2 = $r0.13, $r1.3
567           c0    mpyl $r1.3 = $r0.9, -3196
568
569           c0    mov $r0.16 = $r1.2 # Renaming adjustment
570           c0    mov $r0.12 = $r1.3 # Renaming adjustment
571 ;;
572           c0    add $r1.1 = $r0.3, $r0.14
573           c0    add $r1.2 = $r0.5, $r0.10
574           c0    add $r1.3 = $r1.1, $r0.4
575           c0    add $r1.4 = $r1.2, $r0.16
576
577           c0    mov $r0.9 = $r1.3 # Renaming adjustment
578 ;;
579           c0    ldw $r1.1 = 0x4[$r0.1]
580           c0    shr $r1.2 = $r1.3, 12
581           c0    shr $r1.3 = $r1.4, 12
582           c0    add $r1.4 = $r0.7, $r0.9
583
584           c0    mov $r0.3 = $r1.1 # Renaming adjustment
585           c0    mov $r0.6 = $r1.2 # Renaming adjustment
586 ;;
587           c0    mpyl $r1.1 = $r0.6, 4433
588           c0    add $r1.2 = $r0.8, $r1.4
589           c0    add $r1.3 = $r0.11, $r1.4
590
591           c0    mov $r0.9 = $r1.1 # Renaming adjustment
592           c0    mov $r0.15 = $r1.2 # Renaming adjustment
593           c0    mov $r0.13 = $r1.3 # Renaming adjustment
594           c0    mov $r0.7 = $r1.4 # Renaming adjustment
595 ;;
596           c0    mpyl $r1.1 = $r0.10, 6270
597           c0    add $r1.2 = $r1.2, $r0.16
598           c0    add $r1.3 = $r1.3, $r0.4
599
600           c0    mov $r0.6 = $r1.1 # Renaming adjustment
601 ;;
602           c0    add $r1.1 = $r0.6, 2048
603           c0    shr $r1.2 = $r1.2, 12
604           c0    shr $r1.3 = $r1.3, 12
605
606           c0    mov $r0.10 = $r1.1 # Renaming adjustment
607 ;;
608           c0    add $r1.1 = $r0.3, $r0.9
```

```
609          c0    add $r1.2 = $r0.10, $r1.1
610          c0    mpyl $r1.3 = $r0.5, -15137
611
612          c0    mov $r0.6 = $r1.1 # Renaming adjustment
613          c0    mov $r0.8 = $r1.2 # Renaming adjustment
614          c0    mov $r0.11 = $r1.3 # Renaming adjustment
615 ;;
616          c0    stw 28[$r0.2] = $r0.15
617          c0    sub $r1.2 = $r0.9, $r0.3
618          c0    shl $r1.3 = $r1.1, 1
619          c0    shr $r1.4 = $r1.2, 12
620
621          c0    mov $r0.5 = $r1.3 # Renaming adjustment
622 ;;
623          c0    stw 12[$r0.2] = $r0.13
624          c0    shl $r1.2 = $r1.2, 1
625          c0    add $r1.3 = $r0.6, $r0.5
626
627          c0    mov $r0.4 = $r1.3 # Renaming adjustment
628          c0    mov $r0.10 = $r1.4 # Renaming adjustment
629 ;;
630          c0    stw 20[$r0.2] = $r0.8
631          c0    shr $r1.2 = $r1.3, 12
632
633          c0    mov $r0.9 = $r1.2 # Renaming adjustment
634 ;;
635          c0    stw 4[$r0.2] = $r0.11
636
637          c0    mov $r0.6 = $r1.2 # Renaming adjustment
638 ;;
639          c0    stw 8[$r0.2] = $r0.10
640 ;;
641          c0    stw 24[$r0.2] = $r0.6
642 ;;
643          c0    stw 0[$r0.2] = $r0.4
644 ;;
645          c0    stw 16[$r0.2] = $r0.9
646          c0    add $r1.2 = $r0.2, 32
647 ;;
648          c0    ldw $r1.1 = 0x0[$r0.1]
649                xnop 1
650
651          c0    mov $r0.2 = $r1.2 # Renaming adjustment
652 ;;
653          c0    goto L8?3
654  #       c0    stw 0x8[$r0.1] = $r0.3 # @labeled
655  #       c0    goto L8?3 # @labeled
656
657          c0    mov $r0.5 = $r1.1 # Renaming adjustment
658 ;;
659 L9?3:
660 .trace 5
661          c0    ldw $r1.1 = 0x8[$r0.1]
662          c0    mov $r1.2 = 7
663                xnop 2
664  #       c0    brf $b0.0, L9?3 # @labeled
665  #       c0    ldw.d $r0.7 = 12[$r0.2] # @labeled
666 ;;
667  #             nop
668  #             nop
669          c0    mov $r1.3 = $r1.1 # d=2
670  #       c0    goto L10?3 # @labeled
671  #       c0    stw 192[$r0.2] = $r0.6 # @labeled
672  #       c0    mov $r0.2 = $r1.1  # @labeled
673  #       c0    add $r0.2 = $r0.2, 4  # @labeled
674
675          c0    mov $r0.3 = $r1.1 # Renaming adjustment
676          c0    mov $r0.5 = $r1.2 # Renaming adjustment
677 ;;
678 L10?3:
679 .trace 2
680          c0    ldw.d $r1.1 = 0[$r1.3] # s=2
681          c0    cmpge $r1.2 = $r0.5, $r0.0
682          c0    add $r1.3 = $r0.5, -1
```

```
683 #       c0    goto L10?3 # @labeled
684 #       c0    stw 192[$r0.2] = $r0.6 # @labeled
685 #       c0    mov $r0.2 = $r1.1  # @labeled
686 #       c0    add $r0.2 = $r0.2, 4  # @labeled
687
688         c0    mov $r0.2 = $r1.3 # Renaming adjustment @labeled
689 ;;
690         c0    ldw.d $r1.1 = 224[$r0.2]
691
692         c0    mov $r0.3 = $r1.1 # Renaming adjustment
693         c0    mov $b0.0 = $r1.2 # Renaming adjustment
694         c0    mov $r0.5 = $r1.3 # Renaming adjustment
695 ;;
696         c0    brf $b0.0, L11?3
697         c0    ldw.d $r1.2 = 96[$r0.2] # d=7
698 #       c0    brf $b0.0, L11?3  # @labeled
699 #       c0    ldw.d $r0.7 = 96[$r0.2]  # @labeled
700
701         c0    mov $r0.6 = $r1.1 # Renaming adjustment
702 ;;
703         c0    ldw $r1.1 = 128[$r0.2]
704 #       c0    brf $b0.0, L11?3  # @labeled
705 #       c0    ldw.d $r0.7 = 96[$r0.2]  # @labeled
706
707         c0    mov $r0.7 = $r1.2 # Renaming adjustment
708 ;;
709         c0    ldw $r1.1 = 32[$r0.2]
710         c0    add $r1.2 = $r0.6, $r0.3
711         c0    sub $r1.3 = $r0.3, $r0.6
712
713         c0    mov $r0.8 = $r1.1 # Renaming adjustment
714 ;;
715         c0    ldw $r1.1 = 192[$r0.2]
716         c0    add $r1.2 = $r1.2, 1
717         c0    mpyl $r1.3 = $r1.3, 12299
718
719         c0    mov $r0.10 = $r1.1 # Renaming adjustment
720         c0    mov $r0.6 = $r1.2 # Renaming adjustment
721         c0    mov $r0.9 = $r1.3 # Renaming adjustment
722 ;;
723         c0    ldw $r1.1 = 64[$r0.2]
724         c0    add $r1.2 = $r0.8, $r0.7
725         c0    sub $r1.3 = $r0.7, $r0.8
726
727         c0    mov $r0.3 = $r1.1 # Renaming adjustment
728         c0    mov $r0.12 = $r1.2 # Renaming adjustment
729         c0    mov $r0.11 = $r1.3 # Renaming adjustment
730 ;;
731         c0    ldw $r1.1 = 160[$r0.2]
732         c0    add $r1.2 = $r0.9, $r1.3
733         c0    mpyl $r1.3 = $r1.3, 2446
734
735         c0    mov $r0.14 = $r1.1 # Renaming adjustment
736         c0    mov $r0.8 = $r1.2 # Renaming adjustment
737         c0    mov $r0.13 = $r1.3 # Renaming adjustment
738 ;;
739         c0    sub $r1.1 = $r0.10, $r0.3
740         c0    sub $r1.2 = $r0.6, $r0.8
741         c0    add $r1.3 = $r0.9, $r0.13
742         c0    add $r1.4 = $r0.12, $r0.8
743
744         c0    mov $r0.15 = $r1.1 # Renaming adjustment
745         c0    mov $r0.7 = $r1.2 # Renaming adjustment
746         c0    mov $r0.16 = $r1.3 # Renaming adjustment
747 ;;
748         c0    add $r1.1 = $r1.1, $r0.13
749         c0    mpyl $r1.2 = $r1.3, -7373
750
751         c0    mov $r0.4 = $r1.1 # Renaming adjustment
752         c0    mov $r0.6 = $r1.2 # Renaming adjustment
753         c0    mov $r0.12 = $r1.4 # Renaming adjustment
754 ;;
755         c0    sub $r1.1 = $r0.14, $r0.15
756         c0    mpyl $r1.2 = $r1.1, -16069
```

```
757
758         c0    mov $r0.3 = $r1.2 # Renaming adjustment
759 ;;
760         c0    add $r1.1 = $r0.4, $r1.1
761         c0    add $r1.2 = $r0.4, $r1.1
762         c0    add $r1.3 = $r0.3, 8192
763
764         c0    mov $r0.8 = $r1.1 # Renaming adjustment
765         c0    mov $r0.13 = $r1.2 # Renaming adjustment
766 ;;
767         c0    add $r1.1 = $r0.9, $r0.8
768         c0    add $r1.2 = $r0.7, $r1.2
769         c0    mpyl $r1.3 = $r0.4, 25172
770
771         c0    mov $r0.10 = $r1.1 # Renaming adjustment
772         c0    mov $r0.14 = $r1.2 # Renaming adjustment
773         c0    mov $r0.3 = $r1.3 # Renaming adjustment
774 ;;
775         c0    mpyl $r1.1 = $r1.2, 9633
776         c0    mpyl $r1.2 = $r0.10, -20995
777
778         c0    mov $r0.9 = $r1.1 # Renaming adjustment
779         c0    mov $r0.4 = $r1.3 # Renaming adjustment
780 ;;
781         c0    mpyl $r1.1 = $r0.8, 16819
782         c0    mpyl $r1.2 = $r0.9, -3196
783
784         c0    mov $r0.7 = $r1.1 # Renaming adjustment
785         c0    mov $r0.10 = $r1.2 # Renaming adjustment
786 ;;
787         c0    ldw $r1.1 = 32[$r0.2]
788         c0    add $r1.2 = $r0.10, 8192
789         c0    add $r1.3 = $r0.7, $r0.13
790
791         c0    mov $r0.8 = $r1.1 # Renaming adjustment
792         c0    mov $r0.9 = $r1.2 # Renaming adjustment
793 ;;
794         c0    ldw $r1.1 = 192[$r0.2]
795         c0    add $r1.2 = $r0.16, $r1.3
796         c0    add $r1.3 = $r0.4, $r1.3
797         c0    add $r1.4 = $r0.7, $r0.9
798
799         c0    mov $r0.14 = $r1.1 # Renaming adjustment
800         c0    mov $r0.10 = $r1.2 # Renaming adjustment
801         c0    mov $r0.13 = $r1.3 # Renaming adjustment
802 ;;
803         c0    add $r1.1 = $r1.2, $r0.3
804         c0    add $r1.2 = $r1.3, $r0.10
805         c0    add $r1.3 = $r0.8, $r1.4
806         c0    add $r1.4 = $r0.11, $r1.4
807
808         c0    mov $r0.9 = $r1.1 # Renaming adjustment
809         c0    mov $r0.7 = $r1.4 # Renaming adjustment
810 ;;
811         c0    shr $r1.1 = $r1.1, 14
812         c0    shr $r1.2 = $r1.2, 14
813         c0    add $r1.3 = $r1.3, $r0.10
814         c0    add $r1.4 = $r1.4, $r0.3
815 ;;
816         c0    ldw $r1.1 = 64[$r0.2]
817         c0    add $r1.2 = $r0.9, $r0.14
818         c0    shr $r1.3 = $r1.3, 14
819         c0    shr $r1.4 = $r1.4, 14
820
821         c0    mov $r0.16 = $r1.1 # Renaming adjustment
822         c0    mov $r0.4 = $r1.2 # Renaming adjustment
823 ;;
824         c0    stw 224[$r0.2] = $r0.16
825         c0    mpyl $r1.2 = $r0.6, 6270
826
827         c0    mov $r0.14 = $r1.1 # Renaming adjustment
828         c0    mov $r0.9 = $r1.2 # Renaming adjustment
829         c0    mov $r0.8 = $r1.3 # Renaming adjustment
830         c0    mov $r0.11 = $r1.4 # Renaming adjustment
```

```
831 ;;
832         c0    stw 96[$r0.2] = $r0.4
833
834         c0    mov $r0.3 = $r1.2 # Renaming adjustment
835 ;;
836         c0    stw 160[$r0.2] = $r0.8
837         c0    add $r1.2 = $r0.15, $r0.14
838 ;;
839         c0    stw 32[$r0.2] = $r0.11
840         c0    add $r1.2 = $r0.9, $r1.2
841         c0    sub $r1.3 = $r0.9, $r1.2
842
843         c0    mov $r0.15 = $r1.2 # Renaming adjustment
844 ;;
845         c0    add $r1.1 = $r1.3, $r0.6
846         c0    add $r1.2 = $r1.2, $r0.12
847         c0    mpyl $r1.3 = $r1.3, -15137
848
849         c0    mov $r0.9 = $r1.2 # Renaming adjustment
850 ;;
851         c0    shr $r1.1 = $r1.2, 1
852         c0    sub $r1.2 = $r0.12, $r0.9
853         c0    mpyl $r1.3 = $r1.1, 4433
854
855         c0    mov $r0.4 = $r1.3 # Renaming adjustment
856 ;;
857         c0    stw 0[$r0.2] = $r1.1
858         c0    shr $r1.2 = $r1.2, 1
859
860         c0    mov $r0.7 = $r1.1 # Renaming adjustment
861         c0    mov $r0.6 = $r1.3 # Renaming adjustment
862 ;;
863         c0    stw 128[$r0.2] = $r1.2
864         c0    add $r1.2 = $r0.6, 8192
865
866         c0    mov $r0.12 = $r1.2 # Renaming adjustment
867 ;;
868         c0    add $r1.1 = $r0.3, $r1.2
869         c0    add $r1.2 = $r1.2, $r0.4
870 ;;
871         c0    shr $r1.1 = $r1.1, 14
872         c0    shr $r1.2 = $r1.2, 14
873 ;;
874         c0    stw 64[$r0.2] = $r1.1
875
876         c0    mov $r0.3 = $r1.1 # Renaming adjustment
877         c0    mov $r0.6 = $r1.2 # Renaming adjustment
878 ;;
879         c0    goto L10?3
880         c0    stw 192[$r0.2] = $r0.6
881         c0    add $r1.3 = $r0.2, 4 # d=2
882 #       c0    goto L10?3 # @labeled
883 #       c0    stw 192[$r0.2] = $r0.6 # @labeled
884 #       c0    mov $r0.2 = $r1.1  # @labeled
885 #       c0    add $r0.2 = $r0.2, 0  # @labeled
886 ;;
887 L11?3:
888 .trace 6
889         c0    ldw $r1.1 = 0xc[$r0.1]
890 #       c0    brf $b0.0, L11?3  # @labeled
891 #       c0    ldw.d $r0.7 = 96[$r0.2]  # @labeled
892
893         c0    mov $r0.7 = $r1.2 # Renaming adjustment @labeled
894 ;;
895         c0    ldw $r1.1 = 0x10[$r0.1]
896
897         c0    mov $r0.57 = $r1.1 # Renaming adjustment
898 ;;
899         c0    ldw $r1.1 = 0x14[$r0.1]
900
901         c0    mov $r0.58 = $r1.1 # Renaming adjustment
902 ;;
903         c0    ldw $r1.1 = 0x18[$r0.1]
904
```

```
905           c0    mov $r0.59 = $r1.1 # Renaming adjustment
906   ;;
907           c0    ldw $r1.1 = 0x1c[$r0.1]
908
909           c0    mov $r0.60 = $r1.1 # Renaming adjustment
910   ;;
911           c0    ldw $r1.1 = 0x20[$r0.1]
912
913           c0    mov $r0.61 = $r1.1 # Renaming adjustment
914   ;;
915           c0    ldw $r1.1 = 0x24[$r0.1]
916                 xnop 1
917
918           c0    mov $r0.62 = $r1.1 # Renaming adjustment
919   ;;
920   .return ret()
921           c0    return $r1.1 = $r0.1, (0x40), $l0.0 # d=1
922
923           c0    mov $r0.63 = $r1.1 # Renaming adjustment
924   ;;
925   .endp
926   .section .bss
927   .section .data
928   .equ ?2.3?2spill_p, 0x0
929   .section .data
930   .section .text
931   .equ ?2.3?2auto_size, 0x40
932   .section .text
933   .proc
934   .entry caller, sp=$r0.1, rl=$l0.0, asize=-288, arg()
935   # _____boundary_____
936   main::
937   .trace 1
938           c0    add $r1.1 = $r0.1, (-0x120)
939   ;;
940           c0    stw 0x110[$r1.1] = $l0.0
941           c0    add $r1.2 = $r1.1, 0x10
942
943           c0    mov $r0.1 = $r1.1 # Renaming adjustment
944   ;;
945   .call fill_data, caller, arg($r0.3:u32), ret()
946           c0    call $l0.0 = fill_data
947           c0    stw 0x114[$r0.1] = $r1.2
948
949           c0    mov $r0.3 = $r1.2 # Renaming adjustment
950   ;;
951           c0    ldw $r1.1 = 0x114[$r1.1] # s=1
952                 xnop 1
953           c0    mov $r0.1 = $r1.1 # Renaming adjustment
954   ;;
955   .call print_data, caller, arg($r0.3:u32), ret()
956           c0    call $l0.0 = print_data
957
958           c0    mov $r0.3 = $r1.1 # Renaming adjustment
959   ;;
960           c0    ldw $r1.1 = 0x114[$r0.1]
961                 xnop 1
962   ;;
963   .call jpeg_fdct_islow, caller, arg($r0.3:u32), ret()
964           c0    call $l0.0 = jpeg_fdct_islow
965
966           c0    mov $r0.3 = $r1.1 # Renaming adjustment
967   ;;
968           c0    ldw $r1.1 = 0x114[$r1.1] # s=1
969                 xnop 1
970
971           c0    mov $r0.1 = $r1.1 # Renaming adjustment
972   ;;
973   .call print_data, caller, arg($r0.3:u32), ret()
974           c0    call $l0.0 = print_data
975
976           c0    mov $r0.3 = $r1.1 # Renaming adjustment
977   ;;
978           c0    ldw $l0.0 = 0x110[$r0.1]
```

```
979           c0     mov $r1.2 = $r0.0
980                  xnop 3
981   ;;
982   .return ret($r0.3:s32)
983           c0     return $r0.1 = $r0.1, (0x120), $l0.0
984
985           c0     mov $r0.3 = $r1.2 # Renaming adjustment
986   ;;
987   .endp
988   .section .bss
989   .section .data
990   .equ ?2.4?2scratch.0, 0x0
991   .equ _?1PACKET.26, 0x10
992   .equ ?2.4?2ras_p, 0x110
993   .equ ?2.4?2spill_p, 0x114
994   .section .data
995   .section .text
996   .equ ?2.4?2auto_size, 0x120
997   .section .bss
998   .section .data
999   # _____boundary_____
1000  a::
1001  # _____boundary_____
1002      .data4 0
1003  .section .data
1004  .import _impure_ptr
1005  .type _impure_ptr,@object
1006  .section .text
1007  .import jpeg_fdct_islow
1008  .type jpeg_fdct_islow,@function
1009  .import _bcopy
1010  .type _bcopy,@function
1011  .import fill_data
1012  .type fill_data,@function
1013  .import print_data
1014  .type print_data,@function
1015  .import fprintf
1016  .type fprintf,@function
```

Anexos

ANEXO A – Exemplos de código em Linguagem C da simulação de código Corelap

A.1 Código em Linguagem C do Vecsum

```
1  #define N 2048
2
3  #ifdef VERBOSE
4  #include <stdio.h>
5  #endif
6
7  // Seed 0
8  int A[N] = {
9     // define data here
10 };
11
12
13 // Seed 1
14 int B[N] = {
15     // define data here
16 };
17
18 #ifdef VERBOSE
19 void vecsum_print(int* a, int n)
20 {
21     int i;
22     for (i = 0; i < n; i++)
23         fprintf(stdout, "%d ", a[i]);
24     fprintf(stdout, "\n");
25 }
26 #endif
27
28 void vecsum(int* A, int* B, int* C, int n)
29 {
30     int i;
31     for (i = 0; i < n; i++)
32         C[i] = A[i] + B[i];
33 }
34
35 int main()
36 {
37     int C[N];
38     vecsum(A, B, C, N);
39 #ifdef VERBOSE
40     vecsum_print(C, N);
41 #endif
42     return 0;
43 }
```

A.2 Código em Linguagem C do CJPEG

```
1  /*
2   * jfdctint.c
3   *
4   * Copyright (C) 1991-1996, Thomas G. Lane.
5   * This file is part of the Independent JPEG Group's software.
```

```
6   * For conditions of distribution and use, see the accompanying
        ↪ README file.
7   *
8   * This file contains a slow-but-accurate integer implementation
        ↪ of the
9   * forward DCT (Discrete Cosine Transform).
10  *
11  * A 2-D DCT can be done by 1-D DCT on each row followed by 1-D
        ↪ DCT
12  * on each column.  Direct algorithms are also available, but they
        ↪  are
13  * much more complex and seem not to be any faster when reduced to
        ↪  code.
14  *
15  * This implementation is based on an algorithm described in
16  *   C. Loeffler, A. Ligtenberg and G. Moschytz, "Practical Fast
        ↪ 1-D DCT
17  *   Algorithms with 11 Multiplications", Proc. Int'l. Conf. on
        ↪ Acoustics,
18  *   Speech, and Signal Processing 1989 (ICASSP '89), pp. 988-991.
19  * The primary algorithm described there uses 11 multiplies and 29
        ↪  adds.
20  * We use their alternate method with 12 multiplies and 32 adds.
21  * The advantage of this method is that no data path contains more
        ↪  than one
22  * multiplication; this allows a very simple and accurate
        ↪ implementation in
23  * scaled fixed-point arithmetic, with a minimal number of shifts.
24  */
25
26  #ifdef VERBOSE
27  #include <stdio.h>
28  #else
29  typedef unsigned int size_t;
30  #endif
31
32  #define DCTSIZE             8   /* The basic DCT block is 8x8
        ↪ samples */
33  #define DCTSIZE2            64  /* DCTSIZE squared; # of elements
        ↪ in a block */
34  #define NUM_QUANT_TBLS      4   /* Quantization tables are
        ↪ numbered 0..3 */
35  #define NUM_HUFF_TBLS       4   /* Huffman tables are numbered
        ↪ 0..3 */
36  #define NUM_ARITH_TBLS      16  /* Arith-coding tables are
        ↪ numbered 0..15 */
37  #define MAX_COMPS_IN_SCAN   4   /* JPEG limit on # of components
        ↪ in one scan */
38  #define MAX_SAMP_FACTOR     4   /* JPEG limit on sampling factors
        ↪ */
39
40  typedef long INT32;
41
42  #if BITS_IN_JSAMPLE == 8
43  typedef int DCTELEM;            /* 16 or 32 bits is fine */
44  #else
45  typedef INT32 DCTELEM;          /* must have 32 bits */
46  #endif
47
48  /*
49   * This module is specialized to the case DCTSIZE = 8.
50   */
51  #define DCTSIZE 8
52  #if DCTSIZE != 8
53  Sorry, this code only copes with 8x8 DCTs. /* deliberate syntax
        ↪ err */
54  #endif
55
56  /*
57   * The poop on this scaling stuff is as follows:
58   *
59   * Each 1-D DCT step produces outputs which are a factor of sqrt(N
        ↪ )
```

```
60  * larger than the true DCT outputs.  The final outputs are
        ↪ therefore
61  * a factor of N larger than desired; since N=8 this can be cured
        ↪ by
62  * a simple right shift at the end of the algorithm.  The
        ↪ advantage of
63  * this arrangement is that we save two multiplications per 1-D
        ↪ DCT,
64  * because the y0 and y4 outputs need not be divided by sqrt(N).
65  * In the IJG code, this factor of 8 is removed by the
        ↪ quantization step
66  * (in jcdctmgr.c), NOT in this module.
67  *
68  * We have to do addition and subtraction of the integer inputs,
        ↪ which
69  * is no problem, and multiplication by fractional constants,
        ↪ which is
70  * a problem to do in integer arithmetic.  We multiply all the
        ↪ constants
71  * by CONST_SCALE and convert them to integer constants (thus
        ↪ retaining
72  * CONST_BITS bits of precision in the constants).  After doing a
73  * multiplication we have to divide the product by CONST_SCALE,
        ↪ with proper
74  * rounding, to produce the correct output.  This division can be
        ↪ done
75  * cheaply as a right shift of CONST_BITS bits.  We postpone
        ↪ shifting
76  * as long as possible so that partial sums can be added together
        ↪ with
77  * full fractional precision.
78  *
79  * The outputs of the first pass are scaled up by PASS1_BITS bits
        ↪ so that
80  * they are represented to better-than-integral precision.  These
        ↪ outputs
81  * require BITS_IN_JSAMPLE + PASS1_BITS + 3 bits; this fits in a
        ↪ 16-bit word
82  * with the recommended scaling.  (For 12-bit sample data, the
        ↪ intermediate
83  * array is INT32 anyway.)
84  *
85  * To avoid overflow of the 32-bit intermediate results in pass 2,
        ↪  we must
86  * have BITS_IN_JSAMPLE + CONST_BITS + PASS1_BITS <= 26.  Error
        ↪ analysis
87  * shows that the values given below are the most effective.
88  */
89
90 #if BITS_IN_JSAMPLE == 8
91 #define CONST_BITS  13
92 #define PASS1_BITS  2
93 #else
94 #define CONST_BITS  13
95 #define PASS1_BITS  1           /* lose a little precision to
        ↪ avoid overflow */
96 #endif
97
98 /* Some C compilers fail to reduce "FIX(constant)" at compile time
        ↪ , thus
99  * causing a lot of useless floating-point operations at run time.
100 * To get around this we use the following pre-calculated
        ↪ constants.
101 * If you change CONST_BITS you may want to add appropriate values
        ↪ .
102 * (With a reasonable C compiler, you can just rely on the FIX()
        ↪ macro...)
103 */
104
105 #if CONST_BITS == 13
106 #define FIX_0_298631336  ((INT32)  2446)        /* FIX
        ↪ (0.298631336) */
107 #define FIX_0_390180644  ((INT32)  3196)        /* FIX
        ↪ (0.390180644) */
```

```
108 #define FIX_0_541196100  ((INT32)  4433)        /* FIX
        ↪ (0.541196100) */
109 #define FIX_0_765366865  ((INT32)  6270)        /* FIX
        ↪ (0.765366865) */
110 #define FIX_0_899976223  ((INT32)  7373)        /* FIX
        ↪ (0.899976223) */
111 #define FIX_1_175875602  ((INT32)  9633)        /* FIX
        ↪ (1.175875602) */
112 #define FIX_1_501321110  ((INT32)  12299)       /* FIX
        ↪ (1.501321110) */
113 #define FIX_1_847759065  ((INT32)  15137)       /* FIX
        ↪ (1.847759065) */
114 #define FIX_1_961570560  ((INT32)  16069)       /* FIX
        ↪ (1.961570560) */
115 #define FIX_2_053119869  ((INT32)  16819)       /* FIX
        ↪ (2.053119869) */
116 #define FIX_2_562915447  ((INT32)  20995)       /* FIX
        ↪ (2.562915447) */
117 #define FIX_3_072711026  ((INT32)  25172)       /* FIX
        ↪ (3.072711026) */
118 #else
119 #define FIX_0_298631336  FIX(0.298631336)
120 #define FIX_0_390180644  FIX(0.390180644)
121 #define FIX_0_541196100  FIX(0.541196100)
122 #define FIX_0_765366865  FIX(0.765366865)
123 #define FIX_0_899976223  FIX(0.899976223)
124 #define FIX_1_175875602  FIX(1.175875602)
125 #define FIX_1_501321110  FIX(1.501321110)
126 #define FIX_1_847759065  FIX(1.847759065)
127 #define FIX_1_961570560  FIX(1.961570560)
128 #define FIX_2_053119869  FIX(2.053119869)
129 #define FIX_2_562915447  FIX(2.562915447)
130 #define FIX_3_072711026  FIX(3.072711026)
131 #endif
132
133 /* Multiply an INT32 variable by an INT32 constant to yield an
        ↪ INT32 result.
134  * For 8-bit samples with the recommended scaling, all the
        ↪ variable
135  * and constant values involved are no more than 16 bits wide, so
        ↪ a
136  * 16x16->32 bit multiply can be used instead of a full 32x32
        ↪ multiply.
137  * For 12-bit samples, a full 32-bit multiplication will be needed
        ↪ .
138  */
139
140 #if BITS_IN_JSAMPLE == 8
141 #define MULTIPLY(var,const)  MULTIPLY16C16(var,const)
142 #else
143 #define MULTIPLY(var,const)  ((var) * (const))
144 #endif
145
146 #define ONE      ((INT32) 1)
147 #define DESCALE(x,n)  (((x) + (ONE << ((n)-1))) >> n)
148
149 #ifdef VERBOSE
150
151 void print_data(DCTELEM data[8][8])
152 {
153     int i, j;
154     for (i = 0; i < 8; i++) {
155         for (j = 0; j < 8; j++)
156             fprintf(stdout, "%d ", (int)data[i][j]);
157         fprintf(stdout, "\n");
158     }
159     fprintf(stdout, "\n");
160 }
161
162 void fill_data(DCTELEM data[8][8])
163 {
164     /* Data example from rand() with seed(0) configuration. */
165     DCTELEM data2[8][8] = {
```

```
166         {-20238, 22081, -859, -20826, 20738, 30280, -16590,
                ↪ -17599},
167         {17079, -15435, -27179, 10282, 28835, 24754, 9158, 22156},
168         {20955, -16750, 6772, 30121, 9521, 2303, -8329, -21892},
169         {9092, 22700, 2507, -32688, 16912, -3186, 1649, 29442},
170         {18897, -31978, -24152, 6867, -1698, 24796, 22036, 15382},
171         {-23407, -5143, -7105, 5429, -13157, -30714, -5183, 7799},
172         {-14696, 1590, 5153, -5175, -28875, -3176, -27065,
                ↪ -19783},
173         {-13244, 8210, 13066, -29100, 5025, 14715, 344, 23922}
174     };
175     int i, j;
176     for (i = 0; i < 8; i++)
177         for (j = 0; j < 8; j++)
178             data[i][j] = data2[i][j];
179 }
180
181 #endif
182
183 /*
184  * Perform the forward DCT on one block of samples.
185  */
186 int a = 0;
187 void jpeg_fdct_islow (DCTELEM data[8][8])
188 {
189
190     INT32 tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7;
191     INT32 tmp10, tmp11, tmp12, tmp13;
192     INT32 z1, z2, z3, z4, z5;
193     DCTELEM *dataptr;
194     int ctr;
195
196     //SHIFT_TEMPS
197
198     /* Pass 1: process rows. */
199     /* Note results are scaled up by sqrt(8) compared to a true
            ↪ DCT; */
200     /* furthermore, we scale the results by 2**PASS1_BITS. */
201
202     dataptr = *data;
203     for (ctr = DCTSIZE-1; ctr >= 0; ctr--) {
204
205         tmp0 = dataptr[0] + dataptr[7];
206         tmp7 = dataptr[0] - dataptr[7];
207         tmp1 = dataptr[1] + dataptr[6];
208         tmp6 = dataptr[1] - dataptr[6];
209         tmp2 = dataptr[2] + dataptr[5];
210         tmp5 = dataptr[2] - dataptr[5];
211         tmp3 = dataptr[3] + dataptr[4];
212         tmp4 = dataptr[3] - dataptr[4];
213
214         /* Even part per LL&M figure 1 --- note that published
                ↪ figure is faulty;
215          * rotator "sqrt(2)*c1" should be "sqrt(2)*c6".
216          */
217
218         tmp10 = tmp0 + tmp3;
219         tmp13 = tmp0 - tmp3;
220         tmp11 = tmp1 + tmp2;
221         tmp12 = tmp1 - tmp2;
222
223         dataptr[0] = (DCTELEM) ((tmp10 + tmp11) << PASS1_BITS);
224         dataptr[4] = (DCTELEM) ((tmp10 - tmp11) << PASS1_BITS);
225
226         z1 = MULTIPLY(tmp12 + tmp13, FIX_0_541196100);
227         dataptr[2] = (DCTELEM) DESCALE(z1 + MULTIPLY(tmp13,
                ↪ FIX_0_765366865),
228                                        CONST_BITS-PASS1_BITS);
229         dataptr[6] = (DCTELEM) DESCALE(z1 + MULTIPLY(tmp12, -
                ↪ FIX_1_847759065),
230                                        CONST_BITS-PASS1_BITS);
231
232         /* Odd part per figure 8 --- note paper omits factor of
                ↪ sqrt(2).
```

```
233             * cK represents cos(K*pi/16).
234             * i0..i3 in the paper are tmp4..tmp7 here.
235             */
236
237            z1 = tmp4 + tmp7;
238            z2 = tmp5 + tmp6;
239            z3 = tmp4 + tmp6;
240            z4 = tmp5 + tmp7;
241            z5 = MULTIPLY(z3 + z4, FIX_1_175875602); /* sqrt(2) * c3
                   ↪ */
242
243            tmp4 = MULTIPLY(tmp4, FIX_0_298631336); /* sqrt(2) * (-c1+
                   ↪ c3+c5-c7) */
244            tmp5 = MULTIPLY(tmp5, FIX_2_053119869); /* sqrt(2) * ( c1+
                   ↪ c3-c5+c7) */
245            tmp6 = MULTIPLY(tmp6, FIX_3_072711026); /* sqrt(2) * ( c1+
                   ↪ c3+c5-c7) */
246            tmp7 = MULTIPLY(tmp7, FIX_1_501321110); /* sqrt(2) * ( c1+
                   ↪ c3-c5-c7) */
247            z1 = MULTIPLY(z1, - FIX_0_899976223); /* sqrt(2) * (c7-c3)
                   ↪  */
248            z2 = MULTIPLY(z2, - FIX_2_562915447); /* sqrt(2) * (-c1-c3
                   ↪ ) */
249            z3 = MULTIPLY(z3, - FIX_1_961570560); /* sqrt(2) * (-c3-c5
                   ↪ ) */
250            z4 = MULTIPLY(z4, - FIX_0_390180644); /* sqrt(2) * (c5-c3)
                   ↪  */
251
252            z3 += z5;
253            z4 += z5;
254
255            dataptr[7] = (DCTELEM) DESCALE(tmp4 + z1 + z3, CONST_BITS-
                   ↪ PASS1_BITS);
256            dataptr[5] = (DCTELEM) DESCALE(tmp5 + z2 + z4, CONST_BITS-
                   ↪ PASS1_BITS);
257            dataptr[3] = (DCTELEM) DESCALE(tmp6 + z2 + z3, CONST_BITS-
                   ↪ PASS1_BITS);
258            dataptr[1] = (DCTELEM) DESCALE(tmp7 + z1 + z4, CONST_BITS-
                   ↪ PASS1_BITS);
259
260            dataptr += DCTSIZE;             /* advance pointer to next
                   ↪  row */
261        }
262
263        /* Pass 2: process columns.
264         * We remove the PASS1_BITS scaling, but leave the results
                ↪ scaled up
265         * by an overall factor of 8.
266         */
267
268        dataptr = *data;
269        for (ctr = DCTSIZE-1; ctr >= 0; ctr--) {
270
271            tmp0 = dataptr[DCTSIZE*0] + dataptr[DCTSIZE*7];
272            tmp7 = dataptr[DCTSIZE*0] - dataptr[DCTSIZE*7];
273            tmp1 = dataptr[DCTSIZE*1] + dataptr[DCTSIZE*6];
274            tmp6 = dataptr[DCTSIZE*1] - dataptr[DCTSIZE*6];
275            tmp2 = dataptr[DCTSIZE*2] + dataptr[DCTSIZE*5];
276            tmp5 = dataptr[DCTSIZE*2] - dataptr[DCTSIZE*5];
277            tmp3 = dataptr[DCTSIZE*3] + dataptr[DCTSIZE*4];
278            tmp4 = dataptr[DCTSIZE*3] - dataptr[DCTSIZE*4];
279
280            /* Even part per LL&M figure 1 --- note that published
                   ↪ figure is faulty;
281             * rotator "sqrt(2)*c1" should be "sqrt(2)*c6".
282             */
283
284            tmp10 = tmp0 + tmp3;
285            tmp13 = tmp0 - tmp3;
286            tmp11 = tmp1 + tmp2;
287            tmp12 = tmp1 - tmp2;
288
289            dataptr[DCTSIZE*0] = (DCTELEM) DESCALE(tmp10 + tmp11,
                   ↪ PASS1_BITS);
```

```
290         dataptr[DCTSIZE*4] = (DCTELEM) DESCALE(tmp10 - tmp11,
                ↪ PASS1_BITS);
291
292         z1 = MULTIPLY(tmp12 + tmp13, FIX_0_541196100);
293         dataptr[DCTSIZE*2] = (DCTELEM) DESCALE(z1 + MULTIPLY(tmp13
                ↪ , FIX_0_765366865),
294                                                CONST_BITS+
                                                    ↪ PASS1_BITS);
295         dataptr[DCTSIZE*6] = (DCTELEM) DESCALE(z1 + MULTIPLY(tmp12
                ↪ , - FIX_1_847759065),
296                                                CONST_BITS+
                                                    ↪ PASS1_BITS);
297
298         /* Odd part per figure 8 --- note paper omits factor of
                ↪ sqrt(2).
299          * cK represents cos(K*pi/16).
300          * i0..i3 in the paper are tmp4..tmp7 here.
301          */
302
303         z1 = tmp4 + tmp7;
304         z2 = tmp5 + tmp6;
305         z3 = tmp4 + tmp6;
306         z4 = tmp5 + tmp7;
307         z5 = MULTIPLY(z3 + z4, FIX_1_175875602); /* sqrt(2) * c3
                ↪ */
308
309         tmp4 = MULTIPLY(tmp4, FIX_0_298631336); /* sqrt(2) * (-c1+
                ↪ c3+c5-c7) */
310         tmp5 = MULTIPLY(tmp5, FIX_2_053119869); /* sqrt(2) * ( c1+
                ↪ c3-c5+c7) */
311         tmp6 = MULTIPLY(tmp6, FIX_3_072711026); /* sqrt(2) * ( c1+
                ↪ c3+c5-c7) */
312         tmp7 = MULTIPLY(tmp7, FIX_1_501321110); /* sqrt(2) * ( c1+
                ↪ c3-c5-c7) */
313         z1 = MULTIPLY(z1, - FIX_0_899976223); /* sqrt(2) * (c7-c3)
                ↪  */
314         z2 = MULTIPLY(z2, - FIX_2_562915447); /* sqrt(2) * (-c1-c3
                ↪ ) */
315         z3 = MULTIPLY(z3, - FIX_1_961570560); /* sqrt(2) * (-c3-c5
                ↪ ) */
316         z4 = MULTIPLY(z4, - FIX_0_390180644); /* sqrt(2) * (c5-c3)
                ↪  */
317
318         z3 += z5;
319         z4 += z5;
320
321         dataptr[DCTSIZE*7] = (DCTELEM) DESCALE(tmp4 + z1 + z3,
322                                                CONST_BITS+
                                                    ↪ PASS1_BITS);
323         dataptr[DCTSIZE*5] = (DCTELEM) DESCALE(tmp5 + z2 + z4,
324                                                CONST_BITS+
                                                    ↪ PASS1_BITS);
325         dataptr[DCTSIZE*3] = (DCTELEM) DESCALE(tmp6 + z2 + z3,
326                                                CONST_BITS+
                                                    ↪ PASS1_BITS);
327         dataptr[DCTSIZE*1] = (DCTELEM) DESCALE(tmp7 + z1 + z4,
328                                                CONST_BITS+
                                                    ↪ PASS1_BITS);
329
330         dataptr++;                      /* advance pointer to next
                ↪  column */
331
332     }
333
334 }
335
336 int main(void)
337 {
338     DCTELEM data[8][8];
339 #ifdef VERBOSE
340     fill_data(data);
341     print_data(data);
342 #endif
343     jpeg_fdct_islow (data);
```

```
344 #ifdef VERBOSE
345     print_data(data);
346 #endif
347     return 0;
348 }
```