Hadley Magno da Costa Siqueira

Proposta de Arquitetura de Alto Desempenho para Sistemas de Tempo Real

Natal/RN – Brasil

Julho de 2020

Hadley Magno da Costa Siqueira

Proposta de Arquitetura de Alto Desempenho para Sistemas de Tempo Real

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Sistemas e Computação da Universidade Federal do Rio Grande do Norte, como parte dos requisitos necessários para a obtenção do Título de Doutor.

Universidade Federal do Rio Grande do Norte – UFRN

Departamento de Informática e Matemática Aplicada – DIMAp

Programa de Pós-Graduação em Sistemas e Computação – PPgSC

Orientador: Dr. Marcio Eduardo Kreutz

Natal/RN – Brasil

Julho de 2020

Universidade Federal do Rio Grande do Norte - UFRN
Sistema de Bibliotecas - SISBI
Catalogação de Publicação na Fonte. UFRN - Biblioteca Setorial Prof. Ronaldo Xavier de Arruda - CCET

Siqueira, Hadley Magno da Costa.
Proposta de arquitetura de alto desempenho para sistemas de tempo real / Hadley Magno da Costa Siqueira. - 2020.
122f.: il.

Tese (Doutorado) - Universidade Federal do Rio Grande do Norte, Centro de Ciências Exatas e da Terra, Departamento de Informática e Matemática Aplicada, Programa de Pós-Graduação em Sistemas e Computação. Natal, 2020.
Orientador: Márcio Eduardo Kreutz.

1. Computação - Tese. 2. PRET - Tese. 3. Multicore - Tese. 4. CGRA - Tese. 5. Sistemas de tempo real - Tese. 6. Cyber-físico - Tese. I. Kreutz, Márcio Eduardo. II. Título.

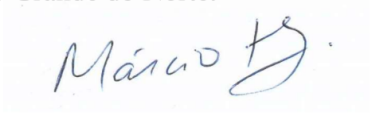
RN/UF/CCET CDU 004

Elaborado por Joseneide Ferreira Dantas - CRB-15/324

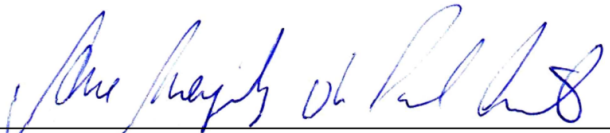
HADLEY MAGNO DA COSTA SIQUEIRA

"*Proposta de Arquitetura de Alto Desempenho para Sistemas de Tempo Real*"

Esta Tese foi julgada adequada para a obtenção do título de doutor em Ciência da Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte.



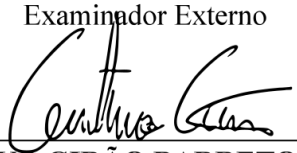
Dr. MÁRCIO EDUARDO KREUTZ (UFRN)
Presidente – Orientador



Profa. Dra. ANNE MAGALY DE PAULA CANUTO
Coordenadora do PPgSC

Banca Examinadora

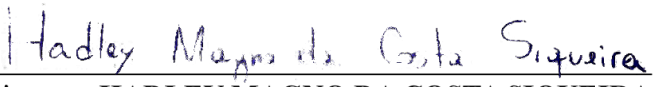
Dr. CÉSAR ALBENES ZEFERINO
Examinador Externo



Dr. GUSTAVO GIRÃO BARRETO DA SILVA
Examinador Externo

Dr. IVAN SARAIVA SILVA
Examinador Externo

Dra. MÔNICA MAGALHAES PEREIRA
Examinadora Interna



Discente: **HADLEY MAGNO DA COSTA SIQUEIRA**

Julho, 2020

Este trabalho é dedicado às crianças adultas que,
quando pequenas, sonharam em se tornar cientistas.

Agradecimentos

Em primeiro lugar, agradeço a Deus pela oportunidade de percorrer esse caminho e ter chegado até esse instante. Percebo Sua presença ao longo dessa jornada e os momentos que me ajudou. No âmbito acadêmico, agradeço primeiramente ao meu orientador, professor Dr. Marcio Eduardo Kreutz, que sempre teve paciência e respeito pela minha maneira de trabalhar e que meu deu total liberdade para escolher o tema do meu trabalho, sempre mostrando as várias alternativas possíveis. Também agradeço ao professor Kreutz pela paciência que teve comigo no que diz respeito a conciliação do doutorado e do meu emprego. Comecei o doutorado bolsista e no decorrer dele comecei a trabalhar, precisando deixar de me dedicar integralmente ao doutorado para passar a conciliar a pesquisa com o novo emprego. Nesse sentido meu orientador sempre foi compreensível e agradeço bastante ele por isso.

Agradeço também ao professor Dr. Ivan Sairava Silva que foi quem me colocou nessa área ainda na graduação e que sou eternamente grato, pois não me vejo sendo tão feliz na computação em outras áreas como sou nessa. Também agradeço a professora Dra. Monica Maglhães Pereira e ao professor Dr. Edgard de Faria Correa que me ajudaram em minha graduação e mestrado e na construção da base para andar com o doutorado.

No âmbito pessoal, agradeço a minha esposa Thayse, que sempre entendeu minhas necessidades acadêmicas e me ajudou e me motivou nos momentos difíceis. Agradeço aos meus dois filhos, Nicole e Johann, cujas vidas me motivam a pensar no futuro e continuar em frente por eles e para eles. Por fim, gostaria de agradecer ao amigo Junior que conheci na Universidade Federal do Piauí e que tive oportunidade de encontrar novamente em um congresso. Nossas conversas ajudaram a polir mais este trabalho.

"O SENHOR é o meu pastor, nada me faltará.
(Bíblia Sagrada, Salmos 23.1)

Resumo

Precision Timed Machines (PRET) são arquiteturas voltadas para o uso em sistemas embarcados de tempo real e sistemas cyber-físicos. A principal característica dessas arquiteturas é a capacidade de prover previsibilidade e repetibilidade para tarefas de tempo real, facilitando assim o desenvolvimento, análise e teste de sistemas de tempo real. O estado da arte, no momento dessa escrita, consiste em um conjunto de processadores baseados no conceito PRET. Esses processadores tem como uma das principais características o uso de threads em hardware com chaveamento das mesmas a cada ciclo de relógio. Essa estratégia provê um bom desempenho quando há paralelismo a nível de thread, porém induz a um baixo desempenho na falta desse paralelismo. Além disso, o chaveamento das threads a cada ciclo de relógio leva a uma latência alta. Essa latência alta pode inviabilizar a execução de tarefas que requeiram baixa latência. O presente trabalho contribui para o estado da arte de duas formas: a primeira é apresentando uma proposta de um array reconfigurável de grão grosso baseado no conceito PRET. O array proposto é acoplado a um processador PRET, provendo suporte para aceleração de trechos importantes de uma aplicação. O array foi projetado de tal forma que quando acoplado ao processador não faça este perder suas propriedades temporais originais. A segunda contribuição desta tese é a proposta e implementação de uma arquitetura *multicore*. Cada *core* é composto por um processador acoplado ao *array* proposto. Dessa forma, este trabalho procura apresentar uma arquitetura de alto desempenho voltado para sistemas embarcados de tempo real que tenham alta demanda de processamento tais como na aviônica, por exemplo. Resultados mostram que a arquitetura proposta é capaz de prover aceleração de mais de 10 vezes para alguns tipos de aplicação. Em termos de área, resultados de síntese para FPGA mostram que cada core ocupa menos da metade de um processador com execução fora de ordem. Além disso, possui área similar a outros arrays usados em sistemas embarcados low-power.

Palavras-chave: PRET, Multicore, CGRA, Sistemas de Tempo Real, Cyber-Físico

Abstract

Precision-Timed Machines (PRET) are architectures intended for use in real-time and cyber physical cyber systems. The main feature of these architectures is that they provide predictability and repeatability for real-time tasks, thus facilitating development, analysis and testing of these systems. The state of the art, at the time of this writing, consists of processors based on the PRET concept. These processors explores thread level parallelism by interleaving threads at a fine-grained leve, i.e. at each clock cycle.This strategy provides good performance when there is parallelism at the thread level, but induces a low performance in the absence of this parallelism. In addition, the switching of threads to each clock cycle leads to high latency. This high latency can make it impossible performing tasks that require low latency. The present work contributes for the state of the art in two ways: first by presenting a proposal for a reconfigurable coarsed-grain reconfigurable array based on the PRET concept. The proposed array is coupled to a PRET processor, providing support for accelerating important parts of an application. The array was designed in such a way that when coupled to the processor do not make the processor lose its original temporal properties. The second contribution of this thesis is the proposal and implementation of a *multicore* architecture. Each *core* is composed of a processor coupled to the proposed *array*. Thus, this work seeks to present a high performance architecture facing embedded real-time systems that have high demand for performance such as avionics, for example. Results show that the proposed architecture it is capable of providing acceleration of more than 10 times for some types of applications. In terms of area, synthesis results for FPGA show that each core occupies less than half of a processor running out of order. In addition, it has an area similar to other arrays used in low-power embedded systems.

Keywords: Precision-Timed Machines. Multicore, Coarse-Grained Reconfigurably Arrays, Cyber Physical, Real-Time Systems.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ULA	Unidade Lógico Aritmética
CGRA	Coarse-Grained Reconfigurable Array
AGU	Address Generation Unit
CGRA	Coarse-Grained Reconfigurable Array
CGRA	Coarse-Grained Reconfigurable Array
CPS	Cyber-Physical Systems
DFG	Data-Flow Graph
DMA	Direct Memory Access
DRAM	Dynamic RAM
DSP	Digital Signal Processor
EP	Elemento de Processamento
FGRA	Fine-Grained Reconfigurable Architecture
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
ILP	Instruction Level Parallelism
IP	Intellectual Property
ISA	Instruction Set Architecture
LE	Logical Element
LLVM	Low Level Virtual Machine
LUT	Look-Up Table
NoC	Network-on-Chip
OoO	Out-of-Order
PC-PO	Parte de Controle – Parte Operativa

PLL	Phase-Locked Loop
PRET	Precision-Timed Machines
RAM	Random Access Memory
RAM	Random Access Memory
RTL	Register Transfer Level
SDRAM	Synchronous Dynamic Random Access Memory
SoC	System-on-Chip
SPM	Scratchpad Memory
TDM	Time-Division Multiplexing
TLP	Thread Level Parallelism
TSMC	Taiwan Semiconductor Manufacturing Company
UF	Unidade Funcional
ULA	Unidade Lógico Aritmética
VHDL	VHSIC Hardware Description Language
VLIW	Very Long Instruction Word
WCET	Worst-Case Execution Time

Sumário

1 Introdução

Em sistemas cyber-físicos (Cyber-Physical Systems - CPS), a computação interage com processos físicos (LEE, 2008). Computadores embarcados, quase sempre conectados através de redes, realizam simultaneamente as tarefas de monitoramento e controle de processos físicos através do uso de sensores e atuadores. Áreas de aplicações tais como automotiva, aviônica, automação industrial e equipamentos médicos contém tais tipos de sistemas com diversos requerimentos. Essa variedade de aplicações impõe desafios para o design de um sistema que atenda toda essa gama de aplicações mas ao mesmo tempo apresenta o potencial para soluções de grande importância.

Diferente de sistemas embarcados de tempo real mais tradicionais (BUTTAZZO, 2011), sistemas cyber-físicos priorizam a interação entre computação e o processo físico que aquela controla. Em ambos os sistemas, a computação não é instantânea e esse tempo de processamento afeta o mundo físico. Dessa forma, o tempo que o software leva para executar passa a ser parte da corretude de uma computação (LEE, 2009). Isso difere do conceito de corretude de software de propósito geral que depende apenas de suas entradas e saídas e o tempo que se leva para realizar o processamento passa a ser apenas uma medida de desempenho. Muitas das técnicas usadas em hardware e software para acelerar computação de propósito geral – que abstrai o tempo de execução para facilitar a programação e melhorar o desempenho do caso mais comum de execução – causam imprevisibilidades temporais e não são aplicáveis diretamente a sistemas de tempo real (EDWARDS; LEE, 2007). Entretanto, desenvolvedores de sistemas ainda usam essas técnicas porque utilizar a infraestrutura já existente de hardware, compiladores, sistemas operacionais, linguagens de programação etc reduz custos e esforços de desenvolvimento.

Em geral, engenheiros, agencias regulatórias e consumidores demandam alta confiabilidade no tempo de execução e funcionalidade de sistemas cyber-físicos do que de sistemas de propósito geral. Muitas áreas de aplicação, como a automotiva e a aviônica, por exemplo, contém funcionalidades de segurança críticas onde uma falha tem consequências catastróficas. Ainda que não houvesse riscos de acidentes com humanos, bugs no software ou baixa confiabilidade causam danos custosos, recalls, processos jurídicos etc. Exemplos famosos são o caso de veículos da Toyota que apresentaram problemas de aceleração indevida e problemas causados por inversão de prioridade no robô Mars Pathfinder (BARR, 2013).

Alcançar alta confiabilidade no tempo de execução e funcionalidade de um software é um problema difícil. Testes empíricos às vezes não suficientes porque o espaço de exploração é muito grande para testar todas as possibilidades. O processo de verificação

formal e técnicas de análise aumentam as garantias mas operação em abstrações do sistema (WILHELM et al., 2008). Essas abstrações e outras suposições nem sempre são o suficiente ou válidas. Em muitos sistemas, o tempo de execução não é especificado diretamente mas sim apenas uma consequência da implementação em uma plataforma de hardware específica, o que leva à implementações fortemente acopladas a uma dada plataforma de hardware. Mesmo com características temporais bem determinadas, o mundo físico é imprevisível e a implementação do software deve ser robustas para se adaptar a mudanças no ambiente.

Apesar dos desafios, sistemas cyber-físicos complexos são construídos hoje em dia e geralmente funcionam corretamente e de forma segura. Entretanto, com o aumento da complexidade desses sistemas, validação e verificação se tornarão tarefas cada vez mais complicadas. No campo automotivo, um carro já ultrapassa a quantidade de 100 milhões de linhas de código executadas em mais de 70 microprocessadores, com o software e hardware contribuindo com cerca de 35% a 40% do custo do veículo (CHARETTE, 2009). Com massivos investimentos em veículos autonômos, essa complexidade só vai aumentar. Mudanças serão necessárias na forma como se projeta software e hardware para esses sistemas cyber-físicos.

Uma das formas para atacar essa crescente complexidade é diminuir a quantidade de hardware necessário para executar o software, diminuindo assim restrições não-funcionais tais como custo, peso, tamanho, consumo de energia etc. Um único processador deve ser então capaz de executar múltiplas tarefas, cada uma com diferentes requisitos, criando assim um sistema criticalidade mista (BURNS; DAVIS, 2013). Esses requisitos geralmente são especificados usando níveis.

Requisitos temporais da ordem de milissegundos não são o suficiente para algumas aplicações, apesar de ser suficiente para várias outras. Aplicações que usam software para substituir funcionalidades geralmente providas por hardware ou que interajam com outros dispositivos requerem um controle preciso de operações de entrada e saída, com intervalos de tempo na ordem de nanossegundos, a mesma granularidade dos ciclos de relógios dos processadores atuais.

Uma arquitetura de processador consiste tanto do conjunto de instruções, que funciona como contrato entre o software e o hardware; e de sua microarquitetura, que determina como as instruções da ISA são executadas. Qualquer abordagem por software toma como premissa caraterísticas temporais em camadas de abstrações mais baixas. Dessa forma, o hardware que executa o software, do qual o processador costuma ser um elemento chave, deve expor suas características temporais para o software, ou pelo menos ser passível de análise temporal.

As abordagens atuais de software já fazem suposições sobre as características temporais do hardware subjacente. Por exemplo, mais de 30 anos de pesquisa em teoria

de escalonamento em tempo real assume que o tempo de computação de cada tarefa é conhecido ou limitado (BUTTAZZO, 2011). Na prática, esses limites são importantes, mas só podem ser garantidos para um subconjunto restrito de técnicas de programação e arquiteturas de processadores.

Uma tendência atual, tanto em software quanto em hardware de tempo real, é o esforço e o custo das fases de verificação e validação que supera as do design. Sem maior confiança na corretude - seja por expor mais informações nas camadas de abstração, seja por melhorar a análise - testando dificuldades de verificação limitarão a complexidade do sistema. Além disso, as arquiteturas devem ser eficientes em relação à área, potência ou desempenho; caso contrário, seria inadequado para a maioria das áreas de aplicação da indústria.

Em (EDWARDS; LEE, 2007) é proposto um novo conceito chamado *Precision-Timed Machines - PRET* que defende a construção de arquiteturas que exponham suas propriedades temporais para o software. Essas arquiteturas além de documentar a semântica do seu conjunto de instruções ou *Instruction Set Architecture - ISA*, também documenta as características temporais das instruções, permitindo que sejam feitas análises e estimativas mais precisas sobre o tempo de execução de uma aplicação.

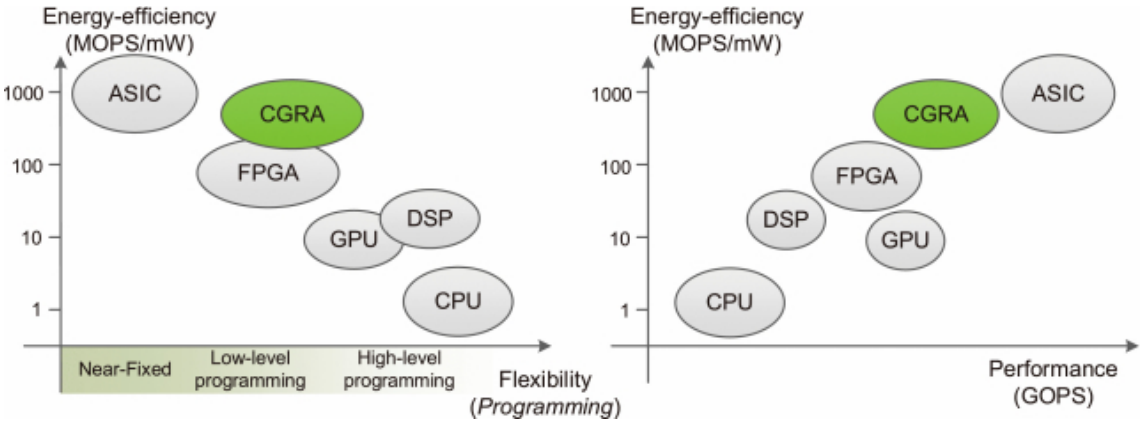
Arquiteturas PRET incorporam técnicas de desempenho que não atrapalhem as características temporais e descartam aquelas que o fazem. Por exemplo, processadores PRET executam threads intercaladas numa granularidade fina a fim de obter isolação temporal entre as threads. Por outro lado, elas não usam memórias caches porque estas atuam como fonte de indeterminismo, com ora o dado estando presente nela, ora não. Em seu lugar, arquiteturas PRET utilizam memórias de rascunho ou *scratchpads*, que são gerenciáveis via software.

O atual estado da arte das arquiteturas PRET consiste em processadores capazes de garantir previsibilidade temporal ao intercalar threads de tempo real em granularidade fina. As threads de tempo real são executadas de forma intercalada e escalonas segundo um esquema de *round-robin*. Essa estratégia permite atingir previsibilidade temporal e permite obter um bom desempenho como um todo. Entretanto, tem como desvantagem uma alta latência do ponto de vista de uma thread e sofre de baixo desempenho quando não há TLP suficiente para ser explorado.

Neste sentido, a presente tese se propõe a apresentar uma solução para melhorar o desempenho de arquiteturas PRET, principalmente do ponto de vista da thread. Da mesma forma que as primeiras arquiteturas PRET se basearam em conceito já existentes para a construção das arquiteturas, adaptando esses conceitos para garantir tempo real, está tese também se baseia em conceitos já existentes e se propõe a adaptar eles a fim de tornar compátivel com a filosofia PRET.

Em arquiteturas tradicionais, o aumento de desempenho advém da exploração de ILP e TLP. Uma das técnicas de aumento de desempenho tem sido através de arquiteturas reconfiguráveis de grão grosso. Essas arquiteturas se encontram em um meio termo entre arquiteturas totalmente reconfiguráveis como no caso dos FPGAs e arquiteturas de propósito geral como processadores convencionais como mostrado na Figura 1.

Figura 1 – CGRA comparado a outras arquiteturas

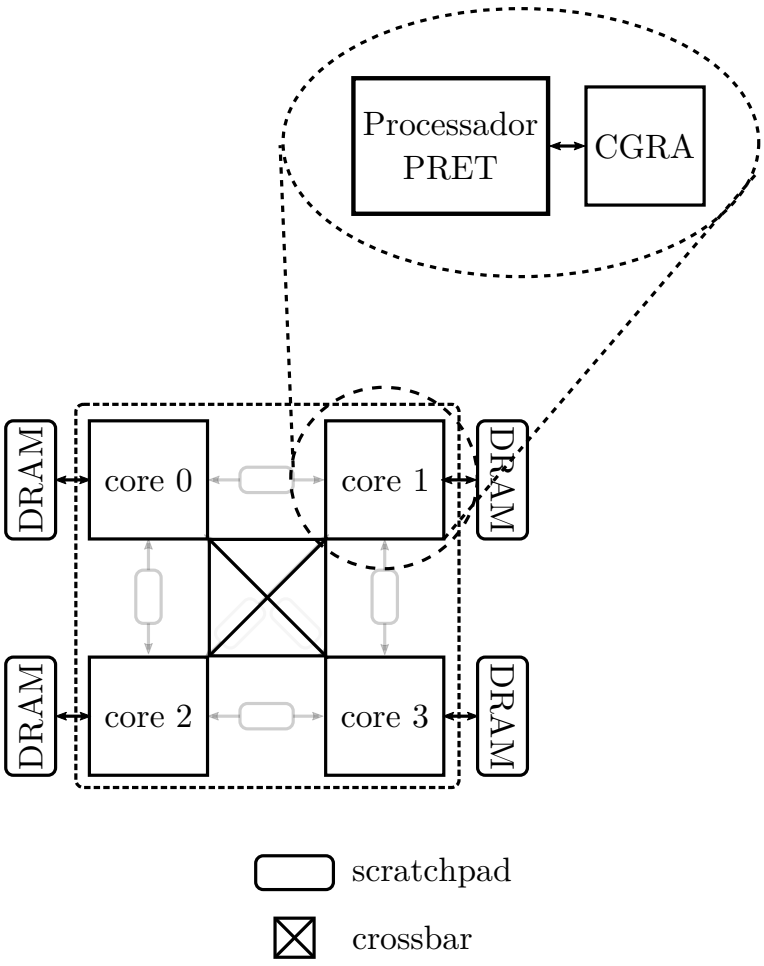


Fonte: (LIU et al., 2019)

Trabalhos como (RUTZIG; BECK; CARRO, 2013a) já mostraram que o melhor desempenho é obtido quando se explora tanto ILP quanto TLP. Assim, nessa tese é proposto um processador multicore, chamado de MultiHivek-RT, com cada núcleo composto por um processador PRET acoplado a um CGRA como mostrado na Figura 2. Arrays reconfiguráveis de grão grosso podem ser acoplados de várias maneiras ao processador. Nessa tese o array reconfigurável é acoplado fracamente ao processador, funcionando como uma espécie de co-processador. Essa maneira foi escolhida porque evita modificações profundas no processador, permitindo que as características temporais do processador sejam preservadas. Essa é uma constante preocupação em uma arquitetura PRET, uma vez que a integração de componentes deve preservar as características temporais dos componentes individuais.

Enquanto o array reconfigurável é responsável pela exploração de ILP na arquitetura proposta, a abordagem multicore é responsável pela exploração de TLP. No multicore proposto, cada núcleo é composto por um processador PRET acoplado ao array reconfigurável proposto. Na atual implementação da proposta, foi implementado um *quadcore* homogêneo, ou seja, todos os núcleos são compostos do mesmo processador PRET acoplado ao mesmo array. O processador PRET escolhido foi o processador Hivek-RT (SIQUEIRA, 2015). O processador Hivek-RT é processador SMT que segue os conceitos PRET.

Figura 2 – Visão geral da arquitetura proposta



Fonte: autoria própria

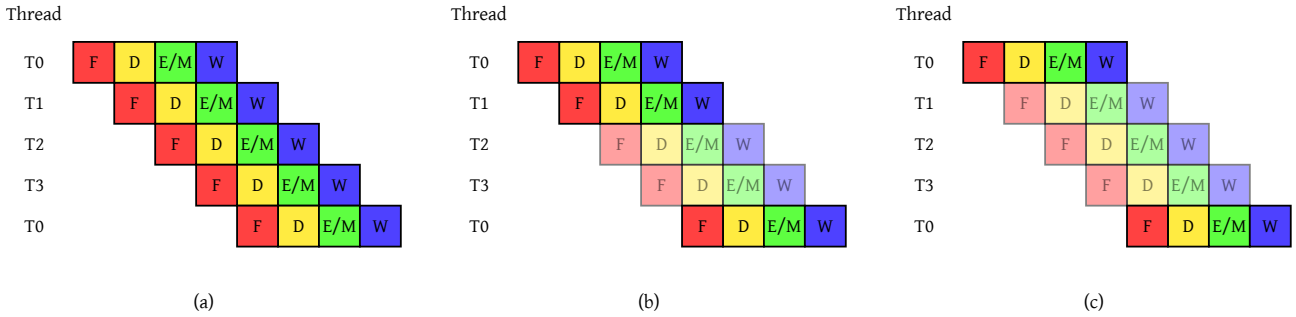
1.1 Problemática

O atual estado-da-arte dos processadores PRET, pode-se resumir aos trabalhos apresentados em (LICKLY et al., 2008), (LIU et al., 2012), (ZIMMER et al., 2014) e (SIQUEIRA; KREUTZ, 2018). Todos esses trabalhos usam o esquema de *fine-grained thread interleaving*, para conseguir o isolamento temporal entre as *threads*. A intercalação das *threads* a cada ciclo de relógio, é o que garante uma fatia de tempo fixa para que se possa realizar análises temporais em uma *thread*, ignorando as demais, já que a cada *thread* é garantido uma fatia de tempo que independe das demais. Além disso, também possibilita um hardware mais simples, uma vez que mecanismos como *bypass* e execução fora de ordem, são removidos.

No atual estado-da-arte, esse processo de intercalação é constante no sentido de que uma *Hard Real-Time Thread (HRTT)* não pode alocar mais ciclos para si, além dos já previamente alocados. Trabalhos como (ZIMMER et al., 2014) e (SIQUEIRA;

KREUTZ, 2018), apresentam processadores capazes de executar tanto *threads* de tempo real como *threads* normais. A estratégia usada é, quando as HRTTs estão impossibilitadas de executar alguma instrução por causa de um acesso a memória, ou porque os operandos não estão disponíveis ainda, instruções de *threads* normais são executadas no lugar. Nesses processadores, as HRTTs só podem ceder seus slots de tempo para outras *threads* normais, mas não podem alocar mais slots para si, caso assim desejem, e nem emprestar slots para outras HRTTs. Essa característica impõe uma limitação no desempenho das HRTTs no processador, como ilustrado na Figura 3.

Figura 3 – Subutilização do pipeline por falta de paralelismo



Fonte: autoria própria

Na Figura 3 é possível ver um pipeline de quatro estágios. Quando existe paralelismo suficiente a nível de *thread*, o processador não sofre de subutilização, pois, todos os estágios ficam ocupados. Entretanto, quando não há threads suficientes, o pipeline fica com bolhas, como é o caso dos itens (b) e (c) da Figura 3.

Quando o processador é capaz de executar instruções de outra *thread*, como é o caso do FlexPRET e Hivek-RT, o problema das bolhas pode ser diminuído. Porém, em casos como o PTARM, não há nada a ser feito. Isso porque não é possível alocar ou desalocar slots de tempos para as HRTTs, sem quebrar a temporização da arquitetura.

Essa falta de flexibilidade traz alguns problemas. O primeiro problema é que o processador corre o risco de ficar subutilizado: gasta-se energia para executar no-operations, a fim de manter a execução de instruções a uma taxa fixa do ponto de vista da *thread*. O segundo problema é a inflexibilidade na transferência de slots de tempo entre as *threads* de tempo real, ocasionando a limitação da capacidade de processamento do ponto de vista da thread. Em alguns casos, o programador pode resolver essa questão colocando *threads* para cooperar numa mesma tarefa, mas em outros casos, isso é impossível em virtude da falta de paralelismo a nível de *thread*.

A partir dessa atual falta de flexibilidade existente nos atuais processadores PRET, a presente tese se propõe a acoplar um array reconfigurável de grão grosso a um processador

PRET, a fim de aumentar o desempenho na execução de *threads HRTT*. A ideia é usar o array reconfigurável para explorar paralelismo a nível de instrução (*Instruction Level Parallelism - ILP*), e circundar, ou, ao menos amenizar o problema de subutilização do pipeline do processador. Ou seja, espera-se que, para um certo nicho de aplicações, o CGRA possa compensar a deficiência do processador host. A junção do processador com o array constitui um núcleo a ser usado para compor um processador multicore, com a arquitetura proposta sendo então capaz de explorar tanto ILP, através do array reconfigurável como TLP, através dos núcleos.

1.2 Objetivos e Contribuições

O atual estado da arte das arquiteturas PRET apresentam processadores com exploração básica de paralelismo a nível de threads ou *Thread Level Parallelism - TLP*. Essa tese apresenta a proposta e implementação de uma arquitetura PRET de alto desempenho quando comparada com o estado da arte. A arquitetura proposta consiste em um processador quadcore. Cada núcleo do processador é composto por um processador PRET e de um array configurável de grão grosso ou *Coarse-Grained Reconfigurable Array - CGRA*. A arquitetura proposta é capaz então de uma exploração de *Instruction Level Parallelism (ILP)* e TLP em uma escala bem maior do que o atualmente fornecido pelo estado da arte. A exploração de ILP é obtida pelo uso array reconfigurável enquanto que a exploração de TLP é obtida pelos núcleos do quadcore.

Em resumo, as contribuições são:

- Proposta e implementação de um array reconfigurável de grão grosso sendo os conceitos das arquiteturas PRET. Até onde se sabe, esse é o primeiro trabalho da literatura a propor um array reconfigurável para máquinas PRET.
- Proposta e implementação de um arquitetura multicore seguindo os conceitos PRET.

1.3 Organização da Tese

O restante desta tese encontra-se dividida da seguinte maneira

- No capítulo 2 é apresentado a fundamentação teórica deste trabalho. Nele é explicado o conceito das Precision-Timed Machines e dos arrays reconfiguráveis de grão grosso, que servem como base de construção para o presente trabalho.
- No capítulo 3 é realizado um levantamento na literatura do estado da arte. Primeiro uma breve revisão de arquiteturas de arrays reconfiguráveis clássicos. Em seguida, é feito um estudo acerca de arrays reconfiguráveis voltados para sistemas de tempo

real. Também é feito um estudo sobre as arquiteturas CReAMS e HyCUBE, cujos elementos foram utilizados durante o desenvolvimento deste trabalho. Por fim, esse capítulo realiza um breve levantamento de arquiteturas multicore diretamente ligadas com o presente trabalho

- O capítulo 4 apresenta a arquitetura proposta. O capítulo primeiro discute os detalhes que compõe cada núcleo e então parte para a explicação do multicore. Também é explicada a hierarquia de memória utilizada. Por fim, há uma breve explanação sobre uma proposta de modelo de programação para a arquitetura.

- O capítulo 5 apresenta os experimentos realizados e resultados obtidos. Nos experimentos foram utilizados os benchmarks da suíte ParMiBench. A arquitetura proposta foi comparada com um processador VLIW e com um processador Superescalar.

- Por fim, o capítulo 6 apresenta a conclusão e trabalhos futuros.

2 Fundamentação Teórica

Este capítulo é destinado a apresentação de conceitos necessários para o entendimento do restante deste trabalho. A primeira seção apresenta o conceito das *Precision Timed Machines*, bem como suas características. Trata-se de arquiteturas de processadores voltados para sistemas de tempo real.

A segunda seção é destinada a apresentação do conceito de arquiteturas reconfiguráveis, mostrando os tipos que existem, suas classificações e características. Por fim, a última seção é responsável por aprofundar o conceito de arquiteturas reconfiguráveis de grão grosso, foco do presente trabalho.

2.1 Precision Timed Machines

As *Precision Timed Machines (PRET)* (EDWARDS; LEE, 2007) foram proposta por Stephen Edwards e Edward Lee. Os autores defendiam a ideia de que era necessário repensar o uso de técnicas existentes para a construção de processadores. Inspirada pela revolução RISC, onde o elemento chave foi o emprego mais eficaz de técnicas já existentes, a filosofia PRET faz repensar a forma como são projetados os processadores para sistemas de tempo real, uma vez que, atualmente costuma-se usar técnicas de desempenho voltadas para a execução do caso mais comum de um programa, em detrimento de uma temporização precisa e que é necessária para sistemas de tempo real.

A filosofia PRET defende que a raiz do problema está na ausência de semânticas temporais nas camadas de abstrações. Quando a ideia das PRETs foi proposta, quase todas as abstrações em uso falhavam em prover uma semântica temporal: o conjunto de instruções, destinado a esconder os detalhes da microarquitetura do software, falhava porque não dava nenhuma garantia sobre o tempo de execução, decorrente do emprego de técnicas como execução especulativa e emprego de caches, por exemplo, que visam a aceleração em detrimento da temporização. Linguagens de programação de alto nível como C e Java, por exemplo, não possuem em suas semânticas quaisquer construções temporais, apenas lógicas. Sistemas operacionais de tempo real escondem dos programas o escalonamento, provendo apenas estimativas, e assim, várias outras camadas de abstração como interfaces de rede, por exemplo, não possuem uma semântica temporal bem definida. A passagem do tempo é, então, uma mera consequência e não algo tratado como prioridade.

A falta de uma semântica temporal teve consequências nos sistemas embarcados de tempo real. Na aviônica, por exemplo, a certificação é extremamente cara: não só o software, mas todo o hardware também precisam ser verificados e validados. Se o fabricante

espera que um modelo dure por décadas, é necessário manter um estoque dos componentes que compõe o sistema por décadas também, uma vez que mesmo pequenas melhorias podem destruir as propriedades temporais e todo o sistema precisaria ser revalidado e certificado. Para piorar, essas verificações não são exaustivas, não podendo se garantir a ausência total de problemas. Fica evidente que essa abordagem tem alto custo e é de difícil manutenção.

Mesmo em ambientes desktop, a ausência de semânticas temporais também causa problemas. A execução concorrente de várias aplicações sem uma semântica temporal de apoio e bem definida impede uma execução previsível e determinística, mesmo quando as condições iniciais são idênticas. Esta é a causa de vários *bugs* de sincronização em programas concorrentes e um dos motivos da necessidade de uso de semáforos, mutex e outras primitivas do gênero.

Atualmente, a ausência de semânticas temporais é mais grave ainda por causa da tendência de projetar arquiteturas e programas voltados para a execução concorrente. Sistemas modernos lidam com a concorrência de sistemas físicos com múltiplas tarefas, componentes e/ou subsistemas integrados. Para poder desenvolver de forma eficiente sistemas complexos, é necessário desenvolver e testar as partes do sistema de forma independente das demais. Essa modularidade é crucial para lidar com a crescente complexidade dos sistemas. Entretanto, se durante a integração com os demais componentes se perde as propriedades que existiam quando isolados, então os módulos não podem ser testados e projetados individualmente. Um sistema que pode ser projetado por partes e depois integrado é dito componível.

A componibilidade temporal se refere a habilidade de integrar componentes preservando suas características e propriedades temporais. A fim de preservar as propriedades dos componentes e módulos durante a fase da integração, arquiteturas modernas usam arquiteturas federadas (ou federativas). Uma arquitetura federativa (OBERMAISSER et al., 2009) contém funcionalidades implementadas em componentes físicos distintos e que são combinados para compor o sistema final, onde essa combinação se dá através do uso de interconexões e barramentos. Como a conexão se dá de forma limitada, a interferência entre os módulos é reduzida, garantindo a preservação de propriedades testadas individualmente.

As arquiteturas federadas são bastante usadas atualmente. Em automóveis, por exemplo, é bastante comum encontrar vários sistemas embarcados trabalhando em paralelo, cada um com uma função bem específica. Essa estratégia, entretanto, faz com que vários módulos fiquem ociosos no decorrer da execução (LUDWIG, 1991). Um exemplo disso é a ociosidade do mecanismo do freio ABS quando o carro se encontra em movimento acelerado. Outras desvantagens da abordagem federada são a grande quantidade de elementos que precisam ser desenvolvidos, custos energéticos, etc. Por causa disso, há uma tendência crescente no uso de arquiteturas integradas, onde várias funcionalidades são integradas em

um único módulo como apresentado em (WATKINS; WALTER, 2007) e (OBERMAISSER et al., 2009). Existem vários desafios nessa mudança, mas a principal delas é garantir que as propriedades temporais de uma funcionalidade continuarão a serem respeitadas quando for alocada junto com outras funcionalidades no mesmo chip. Só então o projeto e design de funcionalidades poderá ser projetado de forma modular como é feito atualmente.

As máquinas PRET são uma tentativa de uma construção de uma arquitetura integrada que forneça uma semântica temporal a nível de conjunto de instruções. A fim de integrar várias tarefas no mesmo processador, as máquinas PRET apostam no uso de multithreading. Cada thread é executada de forma isolada das demais em um processador virtualizado, onde o objetivo dessa virtualização é garantir tempos de acessos previsíveis aos recursos, permitindo que as threads sejam testadas individualmente e depois executadas em conjunto no mesmp chip sem sofrer interferências.

Dessa forma, o conceito PRET defende a construção de arquiteturas que expõem e garantem características temporais. Essas características são:

- **Previsibilidade**: a previsibilidade é a capacidade de uma arquitetura apresentar um comportamento previsível de forma que se possa realizar estimativas em cima desse comportamento. Para sistemas de tempo real, essa característica é importante pois permite predizer possíveis comportamentos da arquitetura diante de um evento, por exemplo.
- **Repetibilidade**: repetibilidade é a capacidade de uma arquitetura apresentar a mesma temporalidade para um mesmo conjunto de entrada. Para sistemas de tempo real, essa característica é de muita valia, uma vez que permite se ter um modelo padrão de comportamento o qual pode ser avaliado de maneira estática.

Os conceitos de previsibilidade e repetibilidade são os pilares das máquinas PRET. É importante entender que os dois conceitos são necessários. Uma arquitetura pode ser previsível e não ter repetibilidade e vice-versa.

Dentre os trabalhos relacionados às máquinas PRET, pode-se dizer que existem três trabalhos canônicos. O primeiro é o apresentado em (LICKLY et al., 2008) que consiste numa máquina PRET cujo conjunto de instrução *(Instruction Set Architecture - ISA)* é baseado na arquitetura SPARC. O segundo é um processador baseado na ISA arquitetura ARMv4, o PTARM (LIU et al., 2012). Por último, o FlexPRET (ZIMMER et al., 2014) é baseado na ISA do RISC-V.

Além das três principais arquiteturas PRET, também existe o Hivek-RT. O Hivek-RT (SIQUEIRA, 2015) é um processador PRET adaptado a partir de um processador VLIW, o Hivek. Essa adaptação acaba transformando o Hivek-RT num processador SMT capaz de executar até duas instruções em paralelo e é o processador PRET escolhido para

ser usado na atual proposta. Mais detalhes sobre o Hivek-RT podem ser encontrados no capítulo de Trabalhos Relacionados.

Apesar das diferenças devidas aos conjunto de instruções diferentes e demais peculiaridades de cada uma, todas usam dos mesmos princípios de funcionamento que são:

- Intercalação de threads em grão fino, ou do inglês *Fine-Grained Thread Interleaving*, de threads em hardware. As três arquiteturas são arquiteturas *multithreading* com o contexto de threads armazenadas em hardware;
- Hierarquia de memória exposta através do uso de memórias *scratchpad* bem como controladores DRAM que expõe a arquitetura interna de memórias DDR SDRAM.
- Instruções especializadas para a manipulação da passagem do tempo durante a execução de um programa.

2.1.1 Uma máquina PRET

A filosofia PRET defende o uso de uma arquitetura multithread, com o pipeline capaz de executar threads intercaladas em granularidade fina: a cada ciclo de relógio uma instrução de uma thread diferente entra no pipeline. Mais especificamente, o núcleo do processador possui um pipeline de N estágios com capacidade para manter o contexto de N threads de hardware. Essas threads são escalonadas por uma política de escalonamento *round-robin* que garante, além da implementação eficiente e simples, uma fatia de tempo igual e determinístico para todas as threads.

Arquiteturas multithread podem apresentar diferentes tempos de execução. Este tipo de comportamento é mais fácil de ser observado em arquiteturas multithread de granularidade grossa, cuja troca de threads se dá na ocorrência de uma instrução de acesso à memória ou de alguma operação multicíclica como multiplicações, por exemplo. Máquinas PRET fazem uso agressivo de multithreading de grão fino ao invés de grão grosso, escalonando uma thread diferente a cada ciclo de relógio ao invés de esporadicamente.

Juntamente com o escalonamento agressivo, uma máquina PRET possui uma quantidade de threads igual à quantidade de estágios no pipeline. O armazenamento do contexto das threads é feito por hardware dedicado a fim de permitir o rápido chaveamento entre as threads. O requisito de se ter uma quantidade de threads igual a quantidade de estágios do pipeline permite a eliminação de conflitos de dados, de controle e estruturais que aparecem em arquiteturas convencionais. Por fim, uma máquina PRET possui, em seu conjunto de instruções, instruções para manipulação da passagem do tempo. Essas instruções permitem ao programador configurar tempos mínimos e máximos de execução para um dado trecho de código de código.

Uma máquina PRET não é composta apenas pelo núcleo de processamento, mas também por sua hierarquia de memória, que é de onde o procesador obtém suas instruções e dados. A hierarquia de memória deve prover tempos de acesso previsível e determinístico ao processador, caso contrário o processador não é capaz de garantir às camadas mais abstratas garantias de tempo. A filosofia PRET defende o uso de uma hierarquia de memória exposta ao programador, dando ao programador a liberdade de escolher em qual tipo de memória ler ou escrever uma informação e qual será o tempo para realizar a ação. Essas características tornam as máquinas PRETs diferentes das arquiteturas tradicionais, que se não se preocupam em fornecer uma abstração temporal para o programador.

2.1.2 Conjunto de instruções

Além do comportamento previsível e repetível da arquitetura, as máquinas PRET fornecem semântica temporal à nível de conjunto de instruções. Alguns exemplos de instruções que são empregadas para fornecer esta semântica são: *get_time*, *delay_until*, *exception_on_expired* e *deactivate_exception*. Essas instruções fazem uso de um relógio interno ao processador que contabiliza a passagem do tempo na mesma frequência que as instruções são executadas.

A instrução *get_time* é usada para obter o valor atual do relógio, carregando o valor do relógio para um registrador de propósito geral, permitindo assim a manipulação posterior pelo programador pelo uso de instruções convencionais como somas, subtrações, etc. Geralmente ela é usada em conjunto com a instrução *delay_until* para determinar tempos mínimos de execução para um certo trecho de código. A instrução *delay_until* recebe como argumento um valor de tempo e tem o seguinte funcionamento: quando o processador encontra essa instrução, ele só prossegue na execução do programa caso o tempo atual seja maior do que o tempo passado como argumento para *delay_until*. Se o tempo já for maior ou igual, *delay_until* funciona como um NOP. Porém, se o tempo for menor, a thread fica bloqueada até se obter o tempo desejado. Dessa forma, essa instrução é usada para garantir um tempo mínimo de execução.

A instrução *exception_on_expired* é usada para especificar um tempo máximo de execução. Ela também recebe como argumento um *timestamp* e quando o processador encontra essa instrução, ele atribui à um contador o valor tempo atual + timestamp. Daí em diante, o valor é decrementado automaticamente até chegar em zero. Neste momento, é disparada uma exceção, o que permite ao programador executar uma ação para reparar a perda do deadline. Ou seja, essa instrução não garante que certo trecho será executado no tempo estipulado, mas sim que, caso o deadline não seja respeitado, o programador terá a chance de tentar reparar o erro, provavelmente usando um código de compensação. Por fim, *deactivate_exception* serve para desativar o contador inicializado pela instrução *exception_on_expired*, evitando assim o lançamento de uma exceção.

Essas são as quatro instruções atualmente que fornecem uma semântica temporal para as máquinas PRET. Apesar de serem simples, elas permitem manipulações de deadlines e de tempos mínimos de execução. Em outras palavras, elas apenas fornecem mecanismos ao programador para monitorar, detectar e interagir com a passagem do tempo via software. Essas instruções não impõem nenhuma política ao programador e nem impedem melhorias na microarquitetura e nem a adição de novas instruções, desde que as semânticas existentes atualmente não sejam violadas. Isso permite ao programador refletir sobre a passagem do tempo sem se preocupar com detalhes da microarquitetura.

2.1.3 Eliminação de conflitos

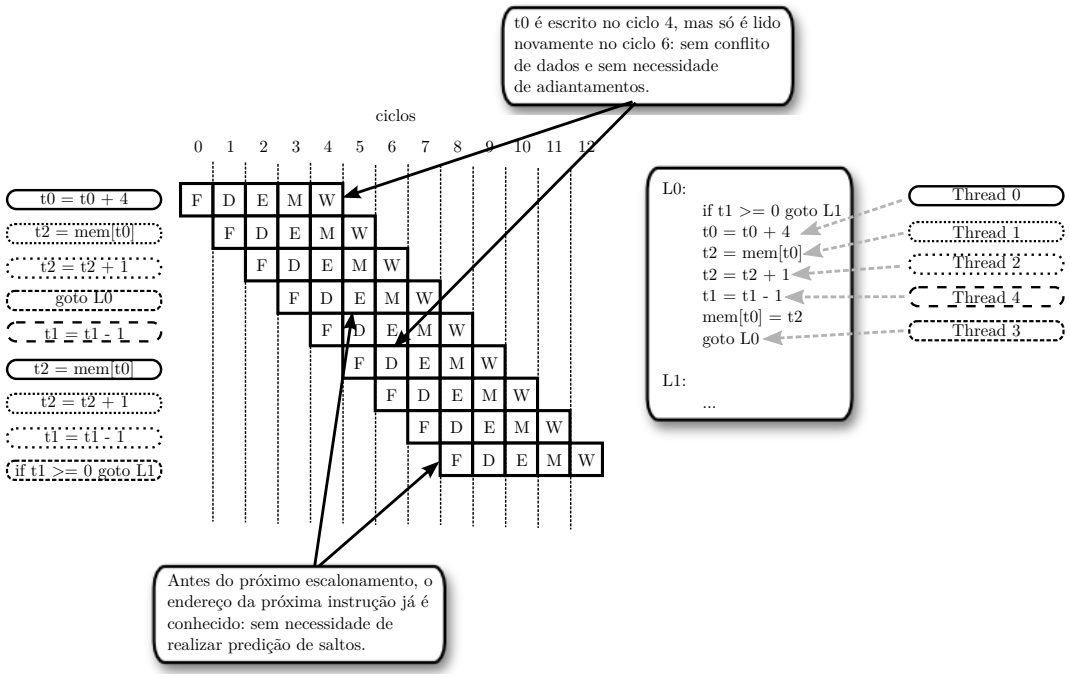
A eliminação de conflitos de dados e de controle em máquinas PRET se dá pela intercalação de uma quantidade de threads maior ou igual que o número de estágios do pipeline. A ideia é que ao intercalar tal quantidade de threads, cada estágio do pipeline estará executando uma instrução de uma thread diferente. Isto significa que não há a possibilidade de qualquer conflito de dados ou de controle no pipeline. Deve-se deixar claro que as threads aqui citadas não são as threads de software, mas sim threads em hardware. Por thread em hardware, quer-se dizer o conjunto de registradores, o estado da thread, valor atual do contador de programa e etc que é salvo por hardware dedicado. No restante deste texto, será usado o termo *thread* para designar uma thread em hardware, usando o termo thread em software explicitamente quando se quiser fazer referência a esta.

A Figura 4 exemplifica como a intercalação das threads elimina os conflitos. Nessa figura é apresentado um pipeline de cinco estágios executando cinco threads diferentes, onde a cada ciclo de relógio uma nova thread entra no pipeline. Essas threads possuem hardware dedicado para manter seus contextos e permitir uma troca rápida de contexto a cada ciclo. As threads são escalonadas sempre na mesma ordem por round-robin, garantindo uma fatia de tempo igual e previsível à todas as threads.

É possível perceber que nunca ocorre uma dependência de dados entre quaisquer instruções que estejam no pipeline, uma vez que estas pertencem à threads diferentes. Essa estratégia permite aumentar a vazão de instruções e elimina a necessidade de usar adiantamentos no pipeline, tornando o hardware mais simples de ser projetado. Entretanto, essa estratégia também tem desvantagens, já que pelo escalonamento fixo em round-robin, o pipeline fica subutilizado quando não há threads o suficiente para preencher o pipeline por completo, resultando em bolhas durante a execução como mostrado na Figura 5.

Os conflitos de controle também são eliminados por essa estratégia. Em arquiteturas convencionais, os conflitos de controle surgem porque o cálculo e tomada de decisão do salto se dá por vários estágios do pipeline, ao passo que a busca de novas instruções se dá a cada ciclo de relógio. Como o cálculo do próximo endereço em uma instrução de salto não consegue ser realizado tão rápido quanto a busca de instruções, o pipeline deve

Figura 4 – Exemplo de execução em um pipeline com execução intercalada de threads

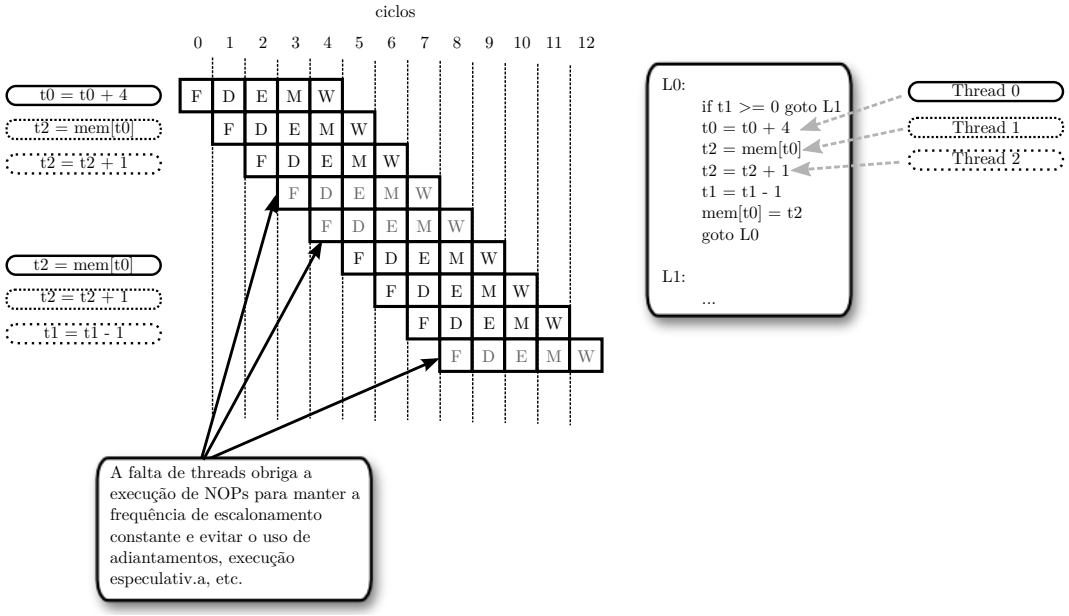


Fonte: (SIQUEIRA, 2015)

ficar ocioso esperando o cálculo ser finalizado. Já no caso de uma máquina PRET, a intercalação das threads amortiza o tempo do cálculo de endereço da instrução de salto executando outras instruções de outras threads nos demais estágios do pipeline. Quando a thread é escalonada novamente para ser executada, todas as dependências já se encontram resolvidas e o próximo endereço pronto para ser usado. A intercalação das threads permite, então, eliminar do processador circuitos como preditores de saltos. Esses preditores, além de aumentarem a área total do procesador, trabalham de forma estatística e dinâmica, impedindo uma execução determinística da thread.

A intercalação de várias threads por si só não é o suficiente para prover previsibilidade e repetibilidade. O pipeline deve ser projetado para que não exista interação entre as threads e assim garantir isolamento temporal entre elas. Na prática, isso significa que recursos compartilhados entre as threads não devem existir a fim de evitar disputa entre as threads pelos recursos. Portanto, juntando intercalação e ausência de compartilhamento de recursos, obtém-se previsibilidade e repetibilidade. A previsibilidade é adquirida quando cada thread sempre é escalonada a cada N ciclos pela política round-robind, garantindo uma fatia de tempo fixa para cada thread. Além disso, a ausência de interferência das demais threads cria a ilusão de que a thread é a única detentora do processador, permitindo obter também repetibilidade, uma vez que não existem interferências externas que possam ocasionar variação no tempo de execução.

Figura 5 – Execução de NOPs em um pipeline com execução intercalada de threads



Fonte: (SIQUEIRA, 2015)

As Precision Timed Machines certamente também tem desvantagens. A principal delas é o fato de que cada thread só possui o processador por 1/N do tempo. Aplicações que não possuem muito paralelismo à nível de thread ou que necessitem de latências muito baixas não são boas candidatas para serem executadas nesse tipo de arquitetura. Por outro lado, aplicações que exibam muito paralelismo a nível de thread e que requeiram uma alta vazão de instruções e dados como as aplicações gráficas, por exemplo, são ótimas candidatas para a execução em máquinas PRETs.

A ausência de compartilhamento de recursos também pode ser uma desvantagem em alguns cenários. Uma tarefa pode precisar de uma quantidade considerável de memória que não pode ser compartilhada com outras tarefas, levando à necessidade de fornecer outros módulos de memória para as demais tarefas e assim aumentando a área e consumo de energia.

2.2 Arrays Reconfiguráveis

Em virtude dos limites encontrados nas arquiteturas consideradas tradicionais, buscou-se formas de aumentar o desempenho utilizando arquiteturas não convencionais. Uma dessas arquiteturas são as reconfiguráveis, que proveem um conjunto de unidades programáveis, as quais podem ser programadas para executar uma série de tarefas, para satisfazer os requisitos computacionais de diferentes aplicações (SINGH et al., 2000). Essas arquiteturas exploram o paralelismo a nível de instruções existentes nas aplicações, mas

também podem explorar outros tipos de paralelismo como, por exemplo, paralelismo a nível de thread, ou do uso de recursos em si como o acesso à memória.

As arquiteturas reconfiguráveis podem estar acopladas a um processador de propósito geral, onde a arquitetura reconfigurável fica responsável pela execução das partes mais importantes do programa, as quais se deseja acelerar, e o processador de propósito geral com as demais partes do programa. Arquiteturas reconfiguráveis também podem trabalhar independentes de um processador, onde nesse caso a arquitetura fica totalmente responsável pela execução do programa.

As primeiras arquiteturas reconfiguráveis trabalhavam a nível de bits, ou seja, seus elementos processantes poderiam manipular bits individualmente. Os FPGAs hoje em dia são os melhores exemplos desse tipo de arquitetura. Com o passar dos tempos, viu-se que poderia se explorar melhor alguns cenários, caso a reconfiguração fosse feita a nível de palavras e operações mais abstratas, ao invés da manipulação bruta dos bits (WIJTVLIET; WAEIJEN; CORPORAAL, 2016).

Atualmente, as arquiteturas reconfiguráveis são classificadas segundo alguns critérios: i) granularidade, ii) acoplamento da arquitetura reconfigurável com o processador, iii) mecanismo de reconfiguração e iv) subsistema de interconexão. O restante desta seção é destinada a explicar esses conceitos.

2.2.1 Granularidade

Uma arquitetura reconfigurável é composta por unidades funcionais, as quais são passíveis de reconfiguração. Essas unidades funcionais, podem ser capazes de trabalharem com bits individuais, ou com um agregado maior como um byte, 16 bits, 32 bits etc. O grau com que uma unidade funcional consegue manipular o dado em si, é o que se chama de granularidade da arquitetura reconfigurável.

Hoje em dia é comum classificar as granularidades em dois tipos de grãos:

- Grão fino: são arquiteturas reconfiguráveis, cujas unidades funcionais podem ser reconfiguradas para trabalharem com bits individuais. O exemplo clássico desse tipo de arquitetura hoje em dia são os FPGAs.
- Grão grosso: são arquiteturas reconfiguráveis cujas unidades funcionais costumam operar em palavras ao invés de bits.

As arquiteturas de grão fino, possuem unidades reconfiguráveis que podem ser programadas para realizar funções a nível de bit, tais como: portas lógicas e flip-flops. Essas arquiteturas também são conhecidas pelo seu nome em inglês: *Fine Grained Reconfigurable Architectures (FGRA)*.

As arquiteturas de grão fino possuem unidades reconfiguráveis que podem ser programadas para realizar funções a nível de bit, tais como: portas lógicas, somadores, multiplexadores de 1 bit, flip-flops. Geralmente, as unidades funcionais desse tipo de arquitetura consistem em *look-up tables*, e/ou multiplexadores aliado a memórias locais e uma rede de conexão que permite a manipulação a nível de bits.

Após o surgimento das FGRAs, veio as arquiteturas de grão grosso, também conhecidas por *Coarse-Grained Reconfigurable Architectures*. Com o tempo, percebeu que várias aplicações não conseguiam utilizar de forma eficaz o grau de flexibilidade fornecidas pelas arquiteturas FGRA porque as operações dessas aplicações trabalhavam a nível de palavras e raramente manipulavam bits de forma individual. Dessa forma, boa parte dos esforços de mapeamento, roteamento e posicionamento, eram gastos para gerar unidades funcionais mais complexas, onde essas sim, seriam usadas pela aplicação em si. Os principais problemas disso eram: tempo alto de reconfiguração, maior gasto de energia e de área. Os CGRAs são uma resposta a esse problema, uma vez que, suas unidades reconfiguráveis não operam em bits individuais, mas sim em palavras, permitindo um menor *bitstream* de reconfiguração, o que leva a um tempo de reconfiguração menor. Em termos de área, os CGRAs também conseguem ser mais eficientes do que os FGRAs (WIJTVLIET; WAEIJEN; CORPORAAL, 2016).

Percebe-se hoje uma tendência à utilização de CGRAs, ao invés de FGRAs para alcançar alta performance e eficiência energética. Isso pode ser observado pela inserção de blocos IPs (Intellectual property) de granularidade grossa nas FPGAs, como DSP (Digital Signal Processor) de ponto flutuante e interfaces de memória externa (WIJTVLIET; WAEIJEN; CORPORAAL, 2016). Outro ponto que colabora para essa tendência, é o grande overhead no custo do roteamento, no tamanho da configuração e no tempo de reconfiguração do sistema, causado pelos FGRAs mencionados nos parágrafo anteriores.

2.2.2 Acoplamento

Uma arquitetura reconfigurável, pode trabalhar de forma independente ou ser comandada por algum outro circuito, onde esse outro circuito costuma ser um processador de propósito geral. O termo acoplamento é usado nesse contexto para determinar de que forma a arquitetura reconfigurável está trabalhando: se de forma independente, ou se acoplado a um processador, por exemplo.

Chama-se de arquitetura *standalone*, aquela que não necessita de outros circuitos para trabalhar de forma autônoma. Os FPGAs, por exemplo, são ótimos exemplos desse tipo de arquitetura. FPGAs podem ser configurados e, uma vez configurados, não dependem de um mecanismo externo para trabalharem. Normalmente, essas arquiteturas necessitam de ferramentas que possibilitem o mapeamento completo da aplicação na arquitetura.

Por outro lado, quando a arquitetura reconfigurável trabalha comandada por um circuito externo, como um processador, por exemplo, diz-se que ela está acoplada. Esse acoplamento pode ocorrer de forma forte ou fraca. Quando se diz que arquitetura reconfigurável está fortemente acoplada ao processador, isso significa que o processador e a arquitetura reconfigurável tendem a se misturar de tal forma que parecem ser um único circuito. Na prática isso significa que um processador pode ver, por exemplo, os registradores e unidades funcionais da arquitetura reconfigurável, e vice-versa. De modo geral, quanto mais próximo do processador a unidade reconfigurável estiver, mais eficiente é a comunicação entre elas.

Com relação as arquiteturas fracamente acopladas, estas podem se conectar ao processador das seguintes formas: como um co-processador, através de barramentos ou através de memória compartilhada. Na primeira abordagem, a arquitetura reconfigurável está mais próxima do processador e a comunicação é mais eficiente que as outras duas formas. No caso de uso do barramento ou de memória compartilhada, a velocidade de comunicação pode ser um gargalo significante no desempenho da arquitetura. Essas decisões de como o acoplamento será realizado devem ser levadas em consideração pois o ganho provido pela arquitetura reconfigurável depende da soma dos tempos de configuração, execução e comunicação do CGRA. Essa soma deve ser menor do que o tempo que o mesmo trecho de código levaria para ser executado no processador sem o auxílio do CGRA.

2.2.3 Mecanismo de Reconfiguração

Os mecanismos de reconfiguração dizem respeito a como a arquitetura se reconfigura para uma dada tarefa. A reconfiguração pode ocorrer de forma estática, com a configuração sendo gerada em tempo de compilação e carregada previamente ao início da execução da tarefa; ou pode ocorrer de forma dinâmica, com a arquitetura sofrendo reconfigurações a medida que a tarefa é executada. Também é possível ter um misto das duas, com partes da arquitetura usando uma configuração estática enquanto as demais usam uma reconfiguração dinâmica.

Na primeira abordagem estática, um compilador ou uma ferramenta de apoio será responsável por identificar as partes da aplicação que podem ser aceleradas pela arquitetura reconfigurável e gerar as configurações para esses trechos. A vantagem dessa abordagem é que a complexidade da geração da configuração está nas ferramentas, permitindo que várias técnicas de otimização possam ser implementadas tais como exploração de ILP e otimizações de loops, por exemplo. Além disso, possibilita que o hardware da arquitetura reconfigurável seja mais simples uma vez que retira a responsabilidade do hardware de prover tal lógica. Apesar dessas vantagens, na reconfiguração estática a configuração só pode ser carregada no sistema quando a arquitetura estiver inativa. Deste modo, todo o array é reconfigurado antes de passar a executar outro trecho de uma aplicação ou

uma nova aplicação, o que pode implicar em aumento do tempo de execução, quando comparado com o uso do mecanismo de configuração dinâmica, pois o sistema precisa ser desativado, reconfigurado e reativado. Por fim, a reconfiguração estática pode impedir a exploração de otimizações que só são possíveis em tempo de execução.

Já na reconfiguração dinâmica, a responsabilidade da geração das configurações é tirada do compilador e o próprio sistema irá se encarregar de fazê-lo em tempo de execução. Essa abordagem permite que a unidade reconfigurável possa ser reconfigurada com o sistema em operação. Desse modo, é possível que um bloco de hardware, acoplado à arquitetura reconfigurável e ao processador, gere as configurações em tempo de execução e, eventualmente, carregue uma nova configuração na arquitetura. Por serem geradas em tempo de execução, as configurações podem explorar características da aplicação. Por exemplo, é possível determinar os trechos de código que estão sendo executados com maior frequência e otimizar aquele trecho. Esse é um tipo de otimização que não é possível com a reconfiguração estática. Por outro lado, o hardware deve ser responsável por implementar tal lógica, o que pode levar a uma arquitetura complexa.

2.2.4 Subsistema de Interconexão

Uma arquitetura reconfigurável geralmente é composta por várias unidades funcionais as quais precisam se comunicar entre si. Os sistemas de interconexão dizem respeito a como essas unidades se conectam. O sistema de interconexão é o meio pelo qual os dados fluirão dentro da arquitetura e permitirá a comunicação e transferência de dados entre as unidades funcionais. As arquiteturas reconfiguráveis possuem quatro tipos de subsistema de interconexão: i) rede em malha, ii) rede de barramento, iii) crossbar e iv) conexão direta.

As redes em malha geralmente consistem numa matriz de unidades funcionais conectadas ponto a ponto com seus vizinhos. São mais comuns de serem encontradas em arquiteturas reconfiguráveis de grão grosso. Essa estrutura é muito utilizada pois possui alta escalabilidade.

Em arquiteturas que usem barramentos, os dados são enviados através de barramentos globais aos quais todas as unidades funcionais têm acesso ou através de uma hierarquia de barramentos. Geralmente esse sistema de conexão é usado em conjunto com algum outro, sendo os barramentos responsáveis pelas comunicações menos frequentes enquanto o outro tipo de conexão fica responsável pelas comunicações mais frequentes. Geralmente, esse mecanismo mais rápido é conseguido através do uso de crossbars. Na interconexão crossbar todas as unidades funcionais estão conectadas às unidades funcionais das camadas adjacentes. Contudo, essa abordagem não é escalável, portanto, não é recomendada para arquiteturas que possuam muitas unidades funcionais e por isso costumam ser usadas com os barramentos como complemento.

Por fim, a interconexão direta, onde cada unidade funcional é conectada diretamente às demais. Normalmente essa conexão é feita por meio da configuração do subsistema de interconexão que informará as interconexões das unidades funcionais. Esse sistema permite a maior velocidade de comunicação, porém tem o maior gasto de área, sendo por isso utilizada em locais onde a comunicação realmente precisa ser o mais rápida possível.

2.3 Arrays Reconfiguráveis de Grão Grosso

Nos últimos anos, arquiteturas reconfiguráveis têm se tornado mais populares por causa da habilidade de se adaptarem à aplicação. Ao fazer isso, essas arquiteturas puderam obter um desempenho por Watt muito melhor em uma variada gama de aplicações, quando comparado com processadores convencionais (HAMEED et al., 2010). Essa tendência se reflete na integração de arquiteturas convencionais com arquiteturas reconfiguráveis como, por exemplo, a Xilinx Zynq e os *System-on-Chip (SoC)* da Altera. Mesmo em arquiteturas de menor desempenho como o microcontrolador PIC16(L)F150X da Microchip (CONFIGURABLE. . . ,), já possuem células reconfiguráveis capazes de executar alguns tipos de operação. Para todas essas arquiteturas, o objetivo é aumentar o desempenho enquanto mantém o consumo energético tão baixo quanto possível, uma propriedade não só desejável em dispositivos móveis, mas também agora em *datacenters* visto os altos custos com refrigeração dos mesmos (PUTNAM et al., 2014).

A fim de obter um melhor desempenho por Watt, *Field Programmable Gate Arrays (FPGAs)* tem aumentando o uso de blocos lógicos de grão grosso. Exemplos de blocos de grão grosso são as unidades de *Digital Signal Processing (DSP)*, interfaces de memória externa e até mesmo processadores completos (MARTIN; CHANG, 2012). Estes blocos de lógica de grão grosso são usados para compensar os altos custos de desempenho e consumo energético do sistema de interconexão dos FPGAs (WIJTVLIET et al., 2016). Ao mesmo tempo, processadores de propósito geral, movem para uma arquitetura mais heterogênea e com unidades reconfiguráveis. Isto permite a essas arquiteturas alcançarem melhores desempenhos (CEBRIAN; NATVIG; MEYER, 2012).

O aumento da popularidade das arquiteturas reconfiguráveis, também é notado no aumento de ferramentas disponíveis, tais como OpenCL para FPGAs, por exemplo (CZAJKOWSKI et al., 2012). OpenCL permite que programadores especifiquem aplicações paralelas com maior facilidade, quando comparado à linguagens de descrição de hardware convencionais, tais como VHDL e Verilog, por exemplo. Além disso, requer um menor conhecimento de hardware por parte do programador. Essa facilidade, porém, não vem de graça, uma vez que OpenCL não permite descrições de operações a nível de bits individuais, de tal forma que a granularidade é bem maior, quando comparado com HDLs convencionais.

Baseado na adição de blocos de grão grosso em FPGAs e processadores de propósito geral, pode-se dizer que existe um movimento de crescimento no uso das arquiteturas de grão grosso, os CGRAs. Os CGRAs podem ser os elementos principais a estabilizar a conexão entre arquiteturas tradicionais e arquiteturas reconfiguráveis.

Na literatura é possível encontrar algumas definições do que seriam arquiteturas reconfiguráveis de grão grosso. Em (SUTTER; RAGHAVAN; LAMBRECHTS, 2019) CGRAs são definidas como arquiteturas voltadas para o processamento de instruções de um loop. Outros trabalhos descrevem CGRAs como arquiteturas capazes de operar numa granularidade a nível de palavras, utilizando unidades funcionais reconfiguráveis (HARTENSTEIN, 2001). Entrentanto, essas definições não são totalmente corretas, pois, a primeira exclui CGRAs capazes de executar uma aplicação por completo, enquanto que a segunda é capaz de incluir FPGAs que estejam agregados à unidades DSP, por exemplo.

Uma definição mais genérica que é usada, diz que arquiteturas reconfiguráveis de granularidade grossa usam um hardware flexível, a fim de adaptar o caminho de dados em tempo de execução, provendo assim, uma arquitetura programável (TESSIER; POCEK; DEHON, 2015). Isso significa que, antes da computação ser realizada, alguma forma de configuração foi carregada previamente na arquitetura. Essa configuração geralmente é estática dentro de um intervalo de tempo como, por exemplo, o tempo de execução de uma aplicação. Entretanto, virtualmente qualquer circuito capaz de executar instruções como um processador, por exemplo, possui esse tipo de configuração em algum grau. No caso, um processador pode ser visto como um circuito que a cada ciclo de relógio é configurado através das instruções que recebe da memória. Até mesmo circuitos ASICs podem ter algum grau de configuração.

Dessa forma, uma definição precisa do que é um CGRA é um pouco complicada de ser dada, porque quase todas as arquiteturas são configuráveis dependendo da forma que se enxergue. Para entender isso melhor, pode-ser pensar em uma arquitetura como uma abstração onde se recebe uma entrada I, uma configuração C e usa a combinação desses dois juntamente com um estado interno S, para produzir uma saída O (WIJTVLIET; WAEIJEN; CORPORAAL, 2016). A parte importante aqui é que essa abstração podem ser vista de vários níveis. Se o caminho de dados de um processador de propósito geral for considerado, então I seriam os operandos, C as instruções a serem executadas, O os dados gerados ao executar as instruções. Porém, se o processador for considerado como um todo, então a entrada I passaria a ser um dado mais complexo, como uma imagem, por exemplo, enquanto que a configuração C passaria a ser o código do executável. Para o usuário, a funcionalidade continua sendo a mesma, pois, a granularidade de operação é escondida do usuário.

É interessante notar que a classificação do que é uma entrada, uma configuração e um estado são convenções legadas dos primeiros projetos de computadores. Geralmente,

essas definições podem ser consideradas intercambiáveis em alguns casos. Como exemplo, podemos citar o caso de uma Look-Up Table (LUT) em um FPGA, que ora pode ser configurada para realizar uma operação binária, e ora serve como elemento de memória, passando a funcionar então, como parte do estado. Outro exemplo é o de processadores que executam código, cujos dados ora são interpretados como instruções, ora como valores.

De um outro ponto de vista, as arquiteturas podem ter seus hardwares configurados em diferentes níveis e em diferentes granularidades temporais e espaciais. O tamanho dos blocos em hardware que são passíveis de configuração a nível de sistema consitituem a granularidade espacial. Já a granularidade temporal diz respeito, por exemplo, por quanto tempo uma configuração fica ativa até que outra tome o seu lugar. Um processador, por exemplo, reconfigura sua ULA (granularidade espacial) a cada ciclo de relógio, enquanto que um FPGA configura suas LUTs (granularidade espacial) antes da execução da aplicação, e assim permanece até o fim da aplicação.

Considerando a granularidade da reconfiguração tanto no domínio espacial quanto temporal, em (WIJTVLIET; WAEIJEN; CORPORAAL, 2016) define-se CGRAs como arquiteturas nos quais podem ser observadas as seguintes características: i) uma reconfiguração espacial a nível de unidade funcional ou acima e; ii) uma reconfiguração temporal a nível de blocos básicos ou acima.

Já que pode haver vários níveis de reconfiguração em um mesmo sistema, os autores recomendam que o foco deve ser dado no nível dominante que está sendo levado em consideração. Considera-se o nível dominante aquele em que ocorre a reconfiguração de maior influência do sistema.

Este capítulo apresentou os conceitos das Precision-Timed Machines e dos arrays reconfiguráveis de grão grosso. O próximo capítulo é destinado a apresentação dos trabalhos relacionados, elencando arquiteturas de arrays reconfiguráveis de grão grosso e de processadores multicore.

3 Trabalhos Relacionados

Desde o surgimento dos CGRAs, várias propostas de arquitetura surgiram na literatura. Este capítulo destina-se a apresentar algumas dessas arquiteturas. Primeiro é feito um breve resumo das arquiteturas mais conhecidas. Em seguida, são apresentadas algumas arquiteturas voltadas para sistemas de tempo real e, por fim, são apresentadas duas arquiteturas que serviram de inspiração para esse trabalho.

A primeira seção deste capítulo contém um resumo de várias arquiteturas CGRAs ao longo dos anos. A fonte desse resumo vem, principalmente, do *survey* apresentado em (WIJTVLIET; WAEIJEN; CORPORAAL, 2016). Já com relação aos trabalhos de CGRAs voltados para tempo real, não se achou muitos trabalhos relacionados que fossem compatíveis com o que é apresentado neste trabalho.

Em seguida são apresentados os trabalhos sobre a arquitetura HyCube e CREaMs. A primeira serviu de base para as versões iniciais da arquitetura proposta. Apesar da arquitetura proposta ter sofrido várias modificações durante sua implementação, alguns conceitos originais do HyCube ainda persistiram tais como quantidades de elementos de processamento, por exemplo. Já a arquitetura CREaMs contribuiu servindo como referencial teórico para alguns resultados de síntese no que diz respeito a quantidade de elementos de processamento capazes de existir num array reconfigurável.

Por fim, o capítulo apresenta as arquiteturas Hivek-RT, RIDECORE e ρ-VEX. O Hivek-RT é o processador PRET usado na arquitetura proposta. É um processador *Simultaneous Multithreading* capaz de executar até duas instruções em paralelo e foi escolhido por causa da familiaridade do autor com a arquitetura bem como pelo código-fonte contendo a descrição da arquitetura estar disponível. De forma analoga, as arquiteturas RIDECORE e ρ-VEX foram escolhidas em virtude da disponibilidade do código-fonte dessas arquiteturas, disponibilizados pelos respectivos autores. O processador RIDECORE é um processador Superescalar com execução fora de ordem enquanto que o ρ-VEX é um processador VLIW. Essas duas arquiteturas são utilizadas para fins de comparação com a arquitetura proposta, visto que elas representam bem ambientes de alto desempenho tradicionais.

3.1 CGRAs clássicos

A arquitetura XPuter (SCHMIDT et al., 1991) consiste em um array reconfigurável de unidades funcionais homogêneas, que são conectadas por uma mesh 2D. A arquitetura se distingue dos demais CGRAs, por causa da forma como trata o fluxo de controle. Ao invés

de usar um contador de programa convencional, o XPuter usa um *data sequencer* e *tags* de controle associadas ao *stream* de dados. O acesso a memória é explícito e controlador pelo sequenciador.

A arquitetura PADDI (CHEN; RABAEY, 1992), assim como a XPuter, também é constituída de várias unidades funcionais homogêneas, porém, usa uma rede de conexão basead em *crossbar*. Cada uma das unidades funcionais possuem uma ULA e uma memória de controle responsável pela configuração. A memória de instrução utiliza palavras de 53 bits com oito entradas. Cada entrada pode ser diferente para cada uma das unidades funcionais, e funcionam de forma similar a uma instrução VLIW. As instruções ativas são selecionadas por um sinal de 3 bits, que funciona como indexador. A *crossbar* é configurada de forma estática a cada ciclo de relógio.

A arquitetura rDPA/KressArray (HARTENSTEIN; KRESS, 1995) é baseada na arquitetura XPuter. Essa arquitetura usa o mesmo tipo de unidade funcional, porém difere em termos de organização. A rDPA consiste de um *array* reconfigurável, que além de permitir a conexão local entre vizinhos, dispõe de um barramento global que permite a comunicação entre todas as unidades funcionais. O rDPA possui suporte de compilador na linguagem C, e o posicionamento e roteamento é realizado através de *simulated annealing*.

Colt (BITTNER; ATHANAS; MUSGROVE, 1996) é uma arquitetura de *array* homogêneo de unidades funcionais, que trabalham com palavras de 16 bits. Essas unidades funcionais são conectadas através de uma *mesh* em forma de toroide, e uma *crossbar* simplificada conecta essa *mesh* para as portas de dados do chip. Cada unidade funcional é capaz de executar operações lógico-aritméticas. Unidades funcionais podem ser combinadas para suportar palavras maiores. A arquitetura Colt é configurada estaticamente, e as unidades funcionais podem introduzir *delays*, a fim de sincronizar as operações. Essa arquitetura tem três modos de operação para execuções condicionais, dependendo das características do salto um dos três modos é escolhido. A ideia é manter o *overhead* de controle dentro de limites aceitáveis para saltos simples, mas manter um desempenho razoável para programas que tenham estruturas de controle mais complexas.

A arquitetura MATRIX (MIRSKY; DEHON et al., 1996) consiste de um *array* de unidades homogêneas, com cada uma contendo uma ULA, memórias locais, unidade de controle e várias portas de comunicação. A rede de comunicação possui três níveis, com cada nível num layout 2D. Cada nível garante um grau de comunicação diferente, indo do mais local até a comunicação global. Esse sistema de comunicação pode operar tanto de forma dinâmica como estática.

A arquitetura GARP (HAUSER; WAWRZYNEK, 1997) combina um processador MIPS com uma espécie de FPGA. O processador MIPS controla a configuração do *array* através de instruções dedicadas. O *array* consiste em unidades funcionais que operam com valores de 2 bits, e que podem ser combinadas para realizarem funções mais complexas.

Dessa forma, o MIPS era capaz de delegar tarefas mais complexas para o *array*.

A arquitetura RAW (WAINGOLD et al., 1997) é um *array* 2D, que contém unidades funcionais com pipelines semelhantes aos usados em processadores RISC. Cada unidade funcional possui seu próprio conjunto de instruções e memórias locais. Cada unidade funcional pode executar independente das demais, e podem se comunicar através de uma *mesh*. As unidades funcionais trabalham a nível de *bytes*, e são combinadas para realizarem funções mais complexas.

A arquitetura RaPID (CRONQUIST et al., 1999) é um *array* reconfigurável, cujas unidades funcionais estão dispostas em uma dimensão apenas (1D), lembrando bastante um array sistólico, porém, com a capacidade de reconfiguração das interconexões e operações das unidades funcionais. O fato de ser 1D ocasionava longas palavras de controle e, para contornar isso, a RaPID definia dois tipos de configuração: uma maior, porém mais estável, que era usada durante toda a aplicação, e uma menor, porém mais dinâmica, que permite reconfiguração durante a execução. Além disso, RaPID tinha suporte para para a execução de laços aninhados de forma eficiente.

A arquitetura REMARC (MIYAMORI; OLUKOTUN, 1999) funciona como um co-processador de um processador MIPS, tendo a comunicação entre eles através de um barramento. O *array* consiste em uma grade contendo 64 unidades funcionais de 16 bits. Uma unidade controle gerencia de forma global os dados, e controle do array. As unidades funcionais numa mesma linha ou coluna, podem ser configuradas para trabalhar em modo SIMD.

A arquitetura CHESS (MARSHALL et al., 1999) contém uma grade de unidades funcionais e de roteamento alocadas de tal forma, que lembra um tabuleiro de xadrez. Esse layout permite uma comunicação eficiente entre as unidades funcionais. Uma outra característica da arquitetura CHESS, é que as unidades de roteamento podem ser configuradas para funcionarem como pequenas memórias RAM locais, transformando assim as unidades funcionais em pequenos mini processadores.

A arquitetura PipeRench (GOLDSTEIN et al., 2000) consiste em várias linhas de unidades funcionais, cada uma dessas linhas é denominada de *stripes*. Unidades funcionais numa mesma linha se comunicam através de uma rede local, e entre diferentes linhas é usado uma espécie de *crossbar*. A arquitetura PipeRench faz uso de um pipeline entre as linhas que permite o mapeamento de um *loop* para as unidades funcionais das *stripes*. Caso o loop não possa ser mapeado por completo de uma só vez, as *stripes* vão sendo reconfiguradas em tempo de execução para executar diferentes estágios do *loop*.

A arquitetura DReaM (ALSOLAIM et al., 2000) é uma arquitetura reconfigurável voltada para o processamento de protocolos *wireless* usados em dispositivos móveis. As unidades funcionais são agrupadas em *clusters*, com cada *clusters* contendo quatro

unidades funcionais. A comunicação intra-cluster é realizada ponto a ponto, enquanto que a comunicação entre *clusters* diferentes se dá por memória compartilhada. Por fim, cada unidade funcional é dotada de uma memória local.

A arquitetura Pleiades (ZHANG et al., 2000) define um template para arquiteturas reconfiguráveis. A arquitetura é composta de um processador host, e de unidades funcionais heterogêneas (chamadas de satelites), que são conectadas através de uma rede reconfigurável. Pleiades dá suporte para a execução vetorial de operações.

MorphoSys (SINGH et al., 2000) é um processador reconfigurável conectado a um processador host através de um barramento. O processador reconfigurável é composto por mesh 8x8 de unidades funcionais, cada uma contendo uma ULA, banco de registradores e um multiplicador. Cada unidade é configurada por uma palavra de 32 bits. Os contextos das configurações são armazenadas em uma memória de contexto e podem ser replicadas via multicast para linhas ou colunas da mesh a fim de simular um SIMD. A comunicação consiste de quatro quadrantes que são conectadas em linhas e colunas. Dentro de cada quadrante uma estrutura mais complexa de conexão é utilizada.

A arquitetura Chimaera (YE et al., 2000) leva a um novo patamar o que se pode considerar uma arquitetura reconfigurável de grão grosso devido ao fato da arquitetura utilizar unidades reconfiguráveis que estão integradas ao pipeline do processador host. Internamente, as unidades funcionais são configuradas ao nível de bit, porém para o processador host apresenta uma interface de trabalho a nível de palavras. Um banco de registradores permite a leitura de até nove operandos, permitindo a definição de operações customizadas no processador host. A memória de configuração é armazenada na memória local do processador host e pode ser armazenada na cache para garantir maior velocidade de configuração.

A arquitetura SmartMemories (MAI et al., 2000) consiste de múltiplos processadores que se comunicam através pacotes em uma rede. Quatro processadores são agrupados em um cluster com comunicação interna rápida. As memórias podem ser conectadas aos processadores de diversas formas através de uma crossbar a fim de se adaptar ao padrão de acesso da aplicação.

A arquitetura Imagine (KAPASI et al., 2002) é composta por oito clusters de unidades VLIWs que processam dados em modo SIMD. Cada cluster contém uma rede de conexão que permite conexões diretas entre as ULAs e seus operandos bem como memórias scratchpads de forma similar a arquitetura Chameleon. As unidades funcionais executam assim que seus operandos se tornam disponíveis, configurando assim uma arquitetura *dataflow*. O suporte a loops é dado através de um processador de controle.

A arquitetura ADRES (MEI et al., 2003) consiste em um processador VLIW fortemente acoplado ao CGRA. O CGRA compartilha unidades funcionais e registradores

com o processador host e assim pode executar tanto em modo processador como em modo CGRA. Quando executando em modo CGRA, o array pode acessar diretamente o banco de dados do processador diminuindo o tempo de transferência de dados.

A arquitetura DART (DAVID et al., 2003) trabalha com o conceito de clusters de caminho de dados reconfiguráveis chamados de DPR. Cada DPR consiste de unidades funcionais conectadas através de vários barramentos configuráveis. Dessa forma, tanto as unidades funcionais como as conexões entre elas podem ser configuradas, o que resulta em longas palavras de configuração (122 bits por DPR). O objetivo da arquitetura DART é manter a configuração estática pela maior quantidade de tempo possível porém, quando necessário, uma estratégia similar a usada pelo RaPID é utilizada. A DART permite uma total reconfiguração da arquitetura em um intervalo de 4 a 9 ciclos de relógio, mas também permite reconfigurações parciais em um único ciclo de relógio. Vale ressaltar que só as unidades funcionais é que podem ser reconfiguradas parcialmente em tempo de execução.

A arquitetura PACT XPP (BAUMGARTE et al., 2003) define dois tipos de elementos processantes (chamados de PAE): um para computação e outro para acesso a memória. Os PAE são conectados através de uma rede baseada em pacotes e o processamento dos dados é baseada em eventos. Tanto os PAEs como a rede de conexão são reconfiguráveis, o que resulta em uma grande quantidade de bits de configuração. O modelo de execução baseada em eventos significa que o controle de fluxo é realizado de forma distribuída. Para dar suporte a computações irregulares que demandem uma maior frequência de reconfiguração, o PACT XPP usa duas técnicas. A primeira é realizar caching das configurações mais usadas enquanto que a segunda é permitir máscaras de bits capazes de realizar reconfiguração diferencial. A arquitetura usa um modelo de computação baseado em *dataflow* e é transparente ao usuário.

A arquitetura TRIPS (SANKARALINGAM et al., 2004) difere da maioria dos CGRAs por causa de sua execução fora de ordem. A arquitetura contém vários processadores que podem se comunicar através de uma rede mesh. Cada processador aloca, dinamicamente, frames de instruções para serem executados na arquitetura. Os frames são armazenados em caches.

Em (KIM et al., 2004) é proposto uma arquitetura reconfigurável baseada na MorphoSys, porém com a diferença de que essa arquitetura é capaz de outros modos de operação além do SIMD suportado pela MorphoSys. A arquitetura também possui suporte para loops de forma similar ao PipeRench. Como consequência, as configurações não podem mais ser compatilhadas entre as unidades mas sim cada uma armazenar a sua própria, já que agora elas podem estar executando operações diferentes das demais.

A arquitetura STA (CICHON et al., 2004) consiste de vários clusters de unidades funcionais, com cada um desses clusters operam em modo SIMD. Dentro de cada cluster, a saída de cada unidade funcional é conectada a todas as outras unidades. A cada ciclo de

relógio uma instrução VLIW ativa uma ou mais unidades funcionais e determina como será feito o roteamento dos operandos e dos resultados permitindo um mapeamento espacial da aplicação nas unidades funcionais.

A arquitetura apresentada em (GALANIS et al., 2005) define um caminho de dados reconfigurável que é integrado em um *System-on-Chip (SoC)* junto com um processador host e um FPGA. As unidades funcionais do caminho de dados contém ULAs e multiplicadores cujos operandos podem vir de outras unidades funcionais ou de um banco de registradores. A interconexão entre as unidades é realizada através de uma crossbar completa ou uma árvore obesa caso ocorram problemas de escalabilidade por causa da crossbar. A organização das unidades funcionais facilita o mapeamento de grafos DFG da aplicação para as unidades. A configuração da arquitetura é gerada pelo FPGA, provendo flexibilidade para o suporte a execução de laços.

A arquitetura Astra (DANILIN; BENNEBROEK; SAWITZKI, 2006) apresenta um array reconfigurável com unidades funcionais trabalhando a nível de bytes. Essas unidades podem ser combinadas para trabalharem em palavras maiores. As unidades funcionais tem capacidade de armazenar várias configurações que podem ser alternadas em tempo de execução. A troca de contexto é realizada por uma lógica de controle separada como um processador, por exemplo. As unidades são conectadas com seus vizinhos em uma malha. Também existe a possibilidade de comunicação entre unidades distantes.

A arquitetura Montium (SMIT et al., 2007) é constituída de várias unidades funcionais que lembram processadores VLIW. Cada unidade funcional recebe os dados por via de uma rede consistindo de crossbars. As configurações são armazenadas na memória local de instruções, e são escalonadas em tempo de compilação.

A arquitetura WaveScalar (SWANSON et al., 2007) é uma arquitetura dataflow cujas unidades reconfiguráveis ficam em uma fila de espera. Essas unidades ficam a espera de instruções vindas de uma cache de trabalho de forma que as instruções são alocadas dinamicamente para as unidades funcionais. As alocações das instruções lembram uma execução fora de ordem, mas as dependências são levadas em consideração baseadas numa descrição dataflow da aplicação em questão. A arquitetura contém quatro camadas de redes de conexão que variam desde conexão ponto a ponto entre as unidades até conexões entre clusters de unidades funcionais.

A arquitetura TCPA (DUTTA et al., 2009) consiste de um array de unidades funcionais heterogêneas que executam instruções VLIW. As unidades são conectadas aos seus vizinhos e a conexão é configurada de forma estática. As unidades podem ser configuradas de forma a trabalhar como uma arquitetura SIMD. Diferentemente de um VLIW tradicional, o banco de registrador é controlado explicitamente.

A arquitetura PPA (PARK; PARK; MAHLKE, 2009a) apresenta um pipeline

polimórfico composto por clusters, com cada cluster sendo composto por quatro unidades funcionais. Dentro do cluster, as unidades funcionais compartilham uma cache de instrução, um buffer para loops e um banco de registradores de predicados. Além disso, cada unidade funcional apresenta seu próprio banco de registradores e ULA. Num cluster, uma das quatro unidades é escolhida para prover uma unidade de multiplicação. A comunicação intra-cluster é realizada por uma mesh. Já para a comunicação entre os clusters, a comunicação se dá através de registradores.

A arquitetura CGRA express (PARK; PARK; MAHLKE, 2009b) é uma extensão da arquitetura ADRES. Ela adiciona a habilidade de comunicação assíncrona entre as unidades funcionais a fim de permitir a construção de operações mais complexas que são capazes de serem executadas em um único ciclo de relógio. Para conseguir isso, as unidades funcionais são conectadas por uma rede de mecanismos de bypass. Um algoritmo de modulo scheduling é adaptado a fim de reconhcer subgrafos na aplicação e combinar essas operações em uma única operação.

A arquitetura EGRA (ANSALONI; BONZINI; POZZI, 2011) é construída como um array de clusters compostos por ULAs chamados de RACs. Esses RACs contém uma estrutra 2D compostas por essas ULAs as quais são conectadas através de uma rede formada por switchs. A comunicação se dá entre linhas. Aliado a isso, existem mais dois níveis de comunicação: um a nível local entre linhas da malha 2D e um outro nível de comunicação entre clusters. Cada cluster possui memórias locais e acesso à memória principal.

A arquitetura DySER (GOVINDARAJU et al., 2012) apresenta um processador com suporte para execução fora de ordem. As unidades reconfiguráveis são inseridas no pipeline do processador. No DySER um estágio extra é adicionado ao pipeline próximo ao estágio de execução. Esse estágio contém uma FIFO que é responsável por transformar um conjunto de instruções sequenciais em um novo conjunto de instruções capazes de serem executadas em paralelo. Essas novas instruções são, então, processadas pelas unidades funcionais. Para a execução de loops, o processador host deve dar suporte ao CGRA.

A arquitetura FPCA (CONG et al., 2014) tem uma malha 2D de clusters. Cada cluster contém várias unidades funcionais, memórias locais e uma comunicação interna provida por crossbar. Cada elemento processante executa o mesmo tipo de operação (similar aos módulos DSP usados em FPGA). A operação de fato é especficada por constantes. Essas constantes e as entradas são roteadas pelos elementos processantes no cluster através da crossbar lembrando uma espécie de processador *Transport Triggered Architecture (TTA)*. A comunicação entre clusters se dá por meio de uma rede ponto-a-ponto. Um escalonador executa uma alocação dinâmica de recursos e tenta preencher a arquitetura com o máximo possível de instruções da aplicação.

A arquitetura HARTMP (SOUZA et al., 2016) apresenta uma matriz reconfigu-

rável dentro de um processador RISC. Instruções são dinamicamente mapeadas para a matriz através de um tradutor binário. As configurações geradas dinamicamente ficam armazenadas em uma cache para uso futuro. Quando a arquitetura detecta que um dado trecho de código já possui uma configuração salva em cache, essa configuração é reusada. Como as configurações são geradas em tempo de execução e de forma transparente para o programa, não é necessário suporte especial por parte de ferramentas de compilação.

3.2 Arrays Reconfiguráveis para Sistemas de Tempo Real

Apesar de existirem várias arquiteturas CGRAs na literatura, um número relativamente pequeno dessas arquiteturas são destinadas para sistemas de tempo real.

Em (LOPES; SOUSA; FERREIRA, 2017) é prosposta uma análise da implementação de uma arquitetura para análise em tempo real de sinais vitais. A arquitetura usada consiste em uma malha 4x4 de unidades funcionais que podem se comunicar com os vizinhos na vertical, horizontal e diagonal. O trabalho se preocupa em levantar as características das aplicações e propõe um sistema semi-automatizado de geração de CGRA para as aplicações alvo. As arquiteturas geradas trabalham em cooperação com um processador host com o processador ficando responsável pelo controle de fluxo da aplicação enquanto que o CGRA fica responsável pela parte dataflow da aplicação. Não são levados em consideração aspectos temporais como nas máquinas PRET, ficando apenas a arquitetura responsável por prover poder computacional suficiente para conseguir processar dentro dos limites estabelecidos.

Em (KIM et al., 2012) é apresentado um CGRA fortemente acoplado com um processador VLIW. As unidades funcionais do CGRA são integradas no pipeline do VLIW e arquitetura ora pode funcionar como um VLIW, ora como um CGRA. O CGRA é capaz de operar numa configuração 2x2 ou 3x3 de unidades funcionais, além de atuar como processador VLIW. Essas configurações tem como objetivo garantir um baixo consumo de potência do que alguma característica de tempo real. A preocupação com potência vem do fato que a arquitetura tem como escopo alvo aplicações biomédicas móveis. Apesar da arquitetura ser voltada para sistemas de tempo real, monitorando sinais vitais, ela considera que apenas essa aplicação estará sendo executada pelo sistema.

3.3 As arquiteturas HyCUBE e CReAMS

Após uma visão geral dos tipos de CGRAs existentes na literatura, esta seção se destina a apresentar os dois principais trabalhos que serviram de base para esse trabalho: o HyCUBE e o CReAMS. O HyCUBE foi o que teve maior influência no trabalho propost. Ele é um CGRA onde uma das principais características é a comunicação

assíncrona que o mesmo possui. De forma similar, o CReAMS usa um CGRA que possui também comunicação entre as unidades de forma combinacional. Essas características são importantes para o trabalho proposto porque permitem a execução de instruções em paralelo de forma combinacional sem degradar o desempenho como um todo. Além disso, dá a esse trabalho o amparo de que ambos os trabalhos foram sintetizados e informações de frequência de operação e área foram obtidos. Dessa forma, como o trabalho proposto foi primeiro implementado em SystemC, isso dá a confiança que quando a hora de obter resultados de área, potência etc, a arquitetura proposta tem potencial para dar resultados satisfatórios. O restante dessa seção é apresentar, então, os dois trabalhos citados e suas características. Primeiro será descrito o HyCUBE e então o CReAMS.

3.3.1 HyCUBE

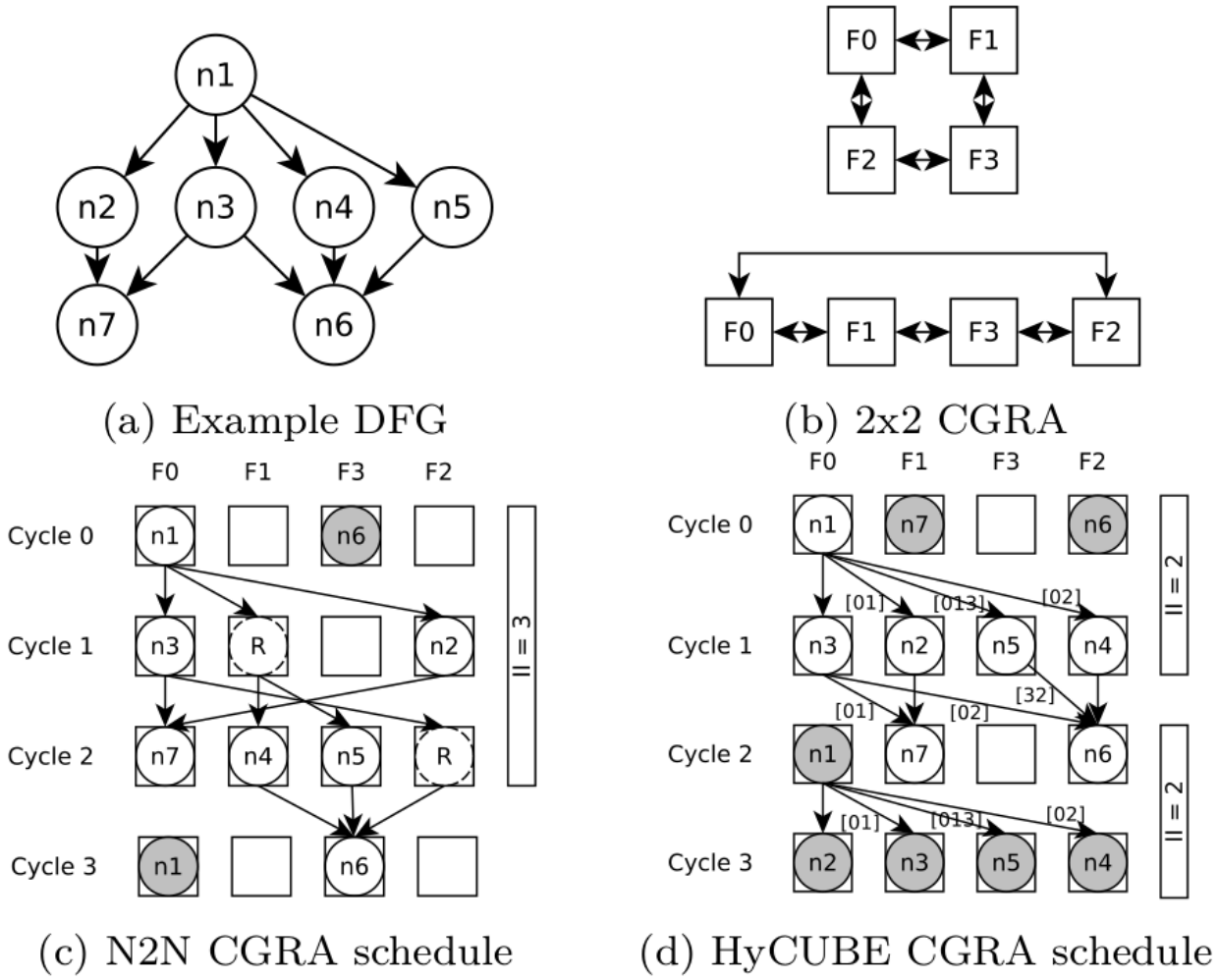
O HyCUBE (KARUNARATNE et al., 2017) é um CGRA com suporte a reconfiguração não só das unidades funcionais como também do sistema de comunicação, que é capaz de prover comunicação entre unidades distantes em um único ciclo de relógio. Essa capacidade de comunicação em um único ciclo de relógio permite uma maior capacidade exploração do mapeamento de instruções para as unidades reconfiguráveis por gerar uma espécie de vizinhança virtual.

Como exemplo de motivação do trabalho, os autores apresentam o pequeno exemplo ilustrado na Figura 6. Nessa figura é ilustrado o mapeamento de um *Data Flow Graph (DFG)* para um array tradicional e para o HyCUBE. Considera-se um array tradicional aquele que provê uma mesh 2D de unidades funcionais que se comunicam ponto a ponto enquanto que o HyCUBE provê comunicação entre unidades funcionais distantes entre si. Nesse exemplo é considerado um pequeno array de 2x2. No item (c) é possível ver o mapeamento das operações do DFG nas quatro unidades do array convencional alocadas no tempo. Enquanto que no ciclo 1 no array convencional a operação n4 é roteada para ser escalonada no ciclo seguinte, no HyCUBE ela é escalonada no ciclo 1 porque existe a capacidade de se enviar o dado para a unidade funcional de destino. Essa capacidade permite que o array seja melhor utilizado e permite intervalos de iniciação menores.

A arquitetura do HyCUBE consiste em unidades funcionais conectadas em uma *mesh* 2D como mostrado na Figura 7. A principal característica que distingue o HyCUBE de outros trabalhos é a rede de conexão usada para conectar as unidades funcionais. Essa rede permite que unidades funcionais distantes uma das outras na mesh 2D possam se comunicar entre si em um único ciclo de relógio e o escalonamento dessas comunicações é gerado de forma estática. Ou seja, o compilador é quem determina a configuração da rede e das unidades funcionais a cada ciclo de relógio.

Cada unidade funcional do HyCUBE é capaz de executar operações lógico-aritméticas. Além disso, as unidades funcionais da primeira coluna à esquerda também possuem unida-

Figura 6 – Exemplo de mapeamento no HyCUBE



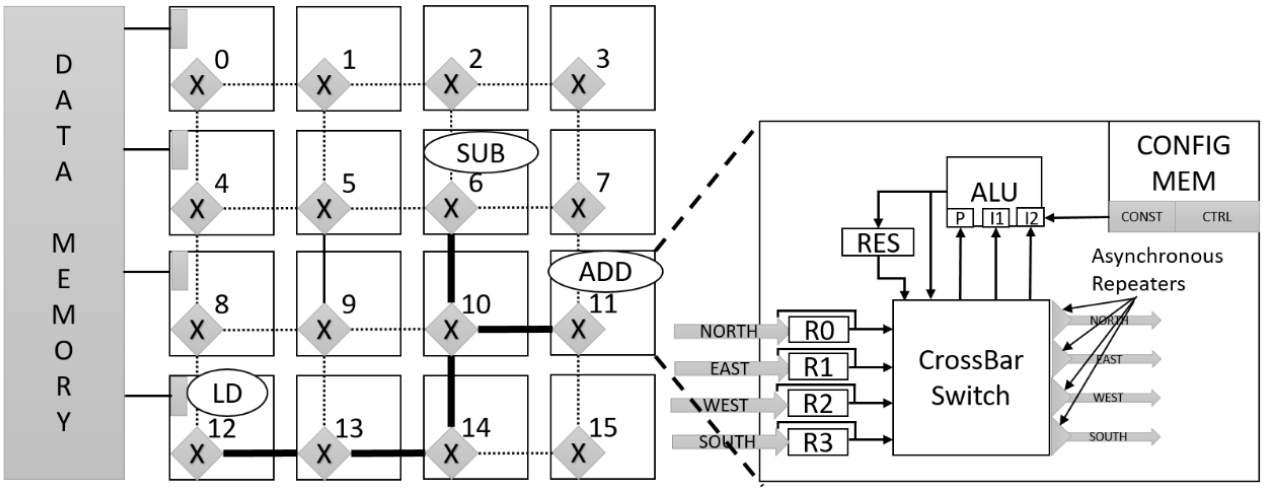
Fonte: (KARUNARATNE et al., 2017)

des de acesso à memória. Em cada unidade também se encontra a memória de configuração e uma crossbar que é a responsável pela comunicação entre as unidades.

A crossbar usada no HyCUBE é programável. Cada saída da crossbar possui um repetidor assíncrono que funciona como buffer temporário. Cada uma dessas saídas podem deixar o sinal passar para o próximo vizinho, ou optar por receber o dado e repassá-lo para a ULA. Isso permite que os dados possam ser enviados para várias unidades em único ciclo. Além disso, essa abordagem é escalável pois, na ocasião de um aumento da rede, a distância entre os vizinhos se mantém constante. A ideia de usar repetidores assíncronos veio da SMART NoC. Neste trabalho, foi demonstrado que tais repetidores conseguem perfazer 11 *hops* em 1ns numa tecnologia de 32nm.

Por causa da comunicação assíncrona, a conexão entre as unidades funcionais do HyCUBE consistem apenas na crossbar e não precisam de hardware para realizar

Figura 7 – Diagrama simplificado da arquitetura HyCUBE



Fonte: (KARUNARATNE et al., 2017)

roteamento ou controle de fluxo. Também permite o uso de apenas um registrador por porta ao invés de filas ajudando assim a reduzir os custos com potência e área que são inerentes aos CGRAs. A configuração de cada crossbar faz parte da instrução de configuração do HyCUBE que determina, a cada ciclo de relógio que conexões serão realizadas.

Além da eliminição dos *queue buffers* das portas da crossbar, a comunicação assíncrona também permite a remoção de banco de registradores da arquitetura. Em CGRAs tradicionais, o banco de registradores é usado para armazenar dados temporariamente que serão usados em ciclos futuros. Esse endereçamento de registradores bem como a leitura e a escrita dos mesmos cria um custo adicional tanto de tempo, quanto de área e potência. Entretanto, no HyCUBE, os registradores são dispensáveis pois a comunicação assíncrona permite que o dado seja roteado e consumido no mesmo ciclo de relógio, dispensando assim a necessidade de armazenamento temporário.

Tendo sido sintetizado para uma tecnologia de 28nm da TSMC, a arquitetura mostrou um melhor desempenho e consumo energético quando comparado com o estado da arte. Esses resultados aliados a arquitetura simples do HyCUBE é o que motivou o uso dessa arquitetura para o presente trabalho. Mais ainda: o fato do HyCUBE ter mostrado resultados sintetizáveis de que a comunicação assíncrona é possível sem degradação de perfomance. Esse resultado é de extrema importância para o presente trabalho pois permite ganhos de desempenho ao permitir que várias instruções sejam executadas em único ciclo de relógio.

3.3.2 CReAMS

A arquitetura CReAMS (RUTZIG; BECK; CARRO, 2013b), apresentada na Figura 8, é um processador multicore cujos núcleos possuem uma unidade CGRA acoplada. Uma das principais características dessa arquitetura é que o CGRA possui uma longa cadeira de lógica combinacional. implementando as unidades funcionais do CGRA.

As unidades funcionais do CReAMS são configuradas em tempo de execução. Um módulo de tradução binária realiza, em tempo de execução, um mapeamento das instruções sendo executadas pelo processador para unidades funcionais do CGRA. As configurações que a arquitetura achar conveniente são armazenadas em uma cache para posterior execução quando ocorrer do mesmo trecho de código ser executado.

O trabalho aqui proposto em quase nada se utiliza da arquitetura do CReAMS e, mais especificamente, do seu CGRA. Porém, uma característica de extrema importância para o presente trabalho é que o CReAMS apresenta resultados, assim como o HyCUBE, de uma capacidade de integrar uma grande quantidade de lógica combinacional sem degradar a frequência de operação do processador host. Os autores do CReAMS alegam frequências de operação na casa dos 600MHz sem degradação do processador host. Esse resultado juntamente com os do HyCUBE constrói uma base para o uso de comunicação assíncrona que será usada nesse trabalho

3.4 Arquiteturas Multiprocessadas

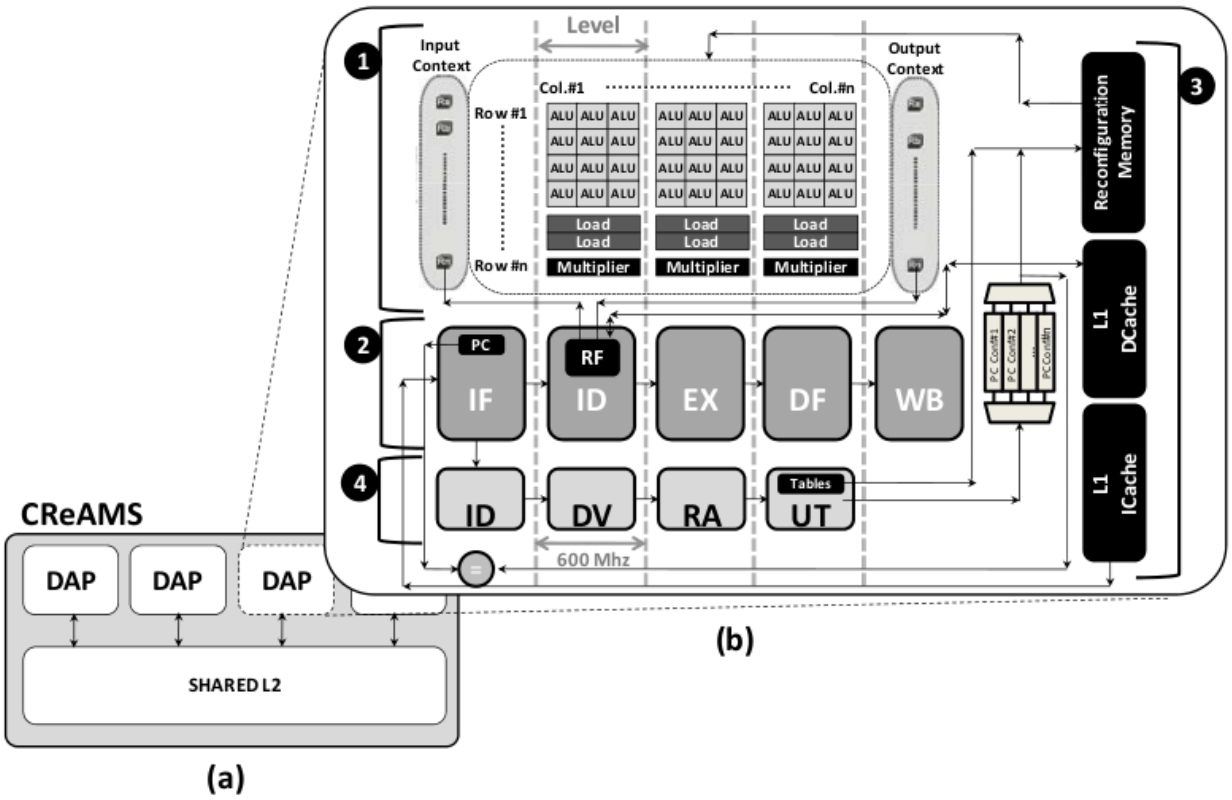
Já existe na literatura propostas de arquiteturas multicores para sistemas de tempo real. O parMERASA (UNGERER et al., 2013) propõe uma arquitetura multicore com cada núcleo consistindo em um processador superescalar. Cada núcleo executa uma única thread de tempo real ou então threads sem requisitos de tempo real. Essa arquitetura entretanto não possui as mesmas características de arquiteturas PRET. O parMERASA apresenta características de previsibilidade porém não apresenta as características de repetibilidade das máquinas PRET.

Outra arquitetura multicore é o T-CREST (SCHOEBERL et al., 2015). É uma arquitetura multicore onde cada núcleo consiste em um processador VLIW chamado PATMOS (SCHOEBERL et al., 2018). Os núcleos são interconectados através de uma rede em chip chamada Argo (KASAPAKI et al., 2015). Essa rede transmite as informações usando a estratégia de multiplexação no tempo.

Um trabalho mais recente é apresentado em (HILI; GIRAULT; JENN, 2019) e que consiste em um multicore denominado MultiPRET, composto por processadores FlexPRET conectados através de um barramento multiplexado no tempo.

Trabalhos como o parMERASA e o T-CREST apresentam exploração de ILP e

Figura 8 – Diagrama simplificado da arquitetura CReAMS



Fonte: (RUTZIG; BECK; CARRO, 2013b)

TLP porém apresenta apenas a característica de previsibilidade. Uma máquina PRET deve apresentar tanto previsibilidade quanto repetibilidade. Já a arquitetura MultiPRET, por ser uma arquitetura PRET, apresenta previsibilidade e repetibilidade, porém só é capaz de explorar TLP. Dessa forma, o presente trabalho se propõe a apresentar uma arquitetura PRET capaz de explorar tanto ILP quanto TLP.

3.5 Os processadores Hivek-RT, RIDECORE e ρ-VEX

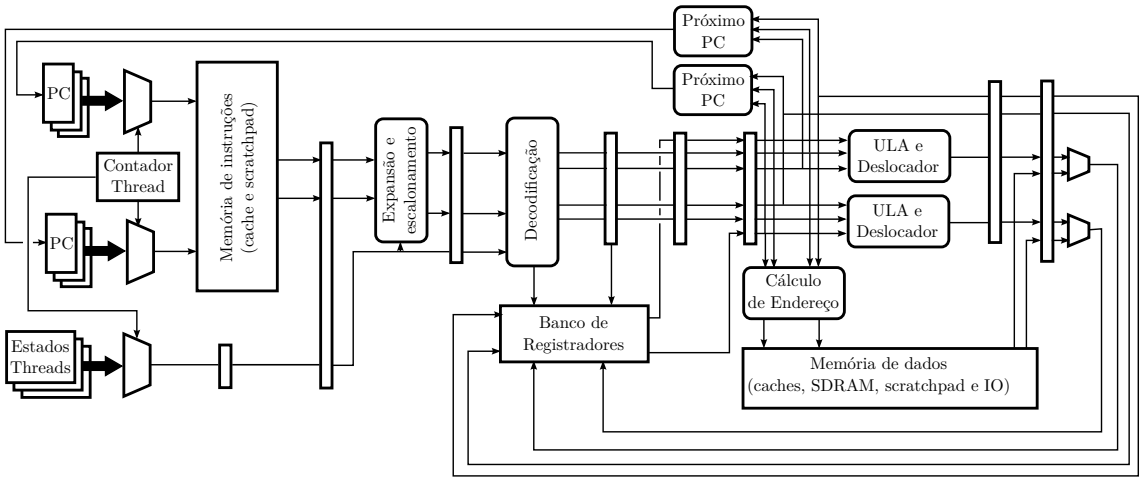
Esta seção é destinada a explicar três arquiteturas de processadores: Hivek-RT, RIDECORE e ρ-VEX. Essas arquiteturas são usadas no escopo deste trabalho para a realização de experimentos confome explicados no capítulo de Experimentos e Resultados. As três arquiteturas foram descritas em nível RTL sendo o Hivek-RT e RIDECORE descritos em Verilog e o ρ-VEX em VHDL. Isso possibilita uma comparação direta e precisa a nível de ciclo de relógio entre as arquiteturas. Como a arquitetura proposta, também descrita em Verilog, utiliza o Hivek-RT como processador base, isso permite uma comparação da arquitetura proposta com arquiteturas clássicas como o RIDECORE,

que aqui está representando as arquiteturas superescalares com execução fora de ordem comumente encontradas em ambientes de alto desempenho e; com o ρ-VEX, que está aqui representando arquiteturas VLIW que geralmente são encontradas em sistemas embarcados que necessitam de um melhor desempenho.

3.5.1 O processador Hivek-RT

O processador Hivek-RT (SIQUEIRA, 2015) é o processador PRET utilizado na arquitetura proposta. O processador Hivek-RT é um processador *Simultaneous Multithreading* implementado seguindo o conceito das Precision-Timed Machines. Ele é capaz de executar até duas operações em paralelo. Possui hardware para armazenar o contexto de um total de 16 threads, sendo estas particionadas em dois grupos. O primeiro grupo contém oito threads de tempo real enquanto que o segundo grupo contém 8 threads sem requisitos de tempo real. A Figura 9 apresenta o pipeline do processador Hivek-RT.

Figura 9 – Pipeline do processador Hivek-RT



Fonte: autoria própria

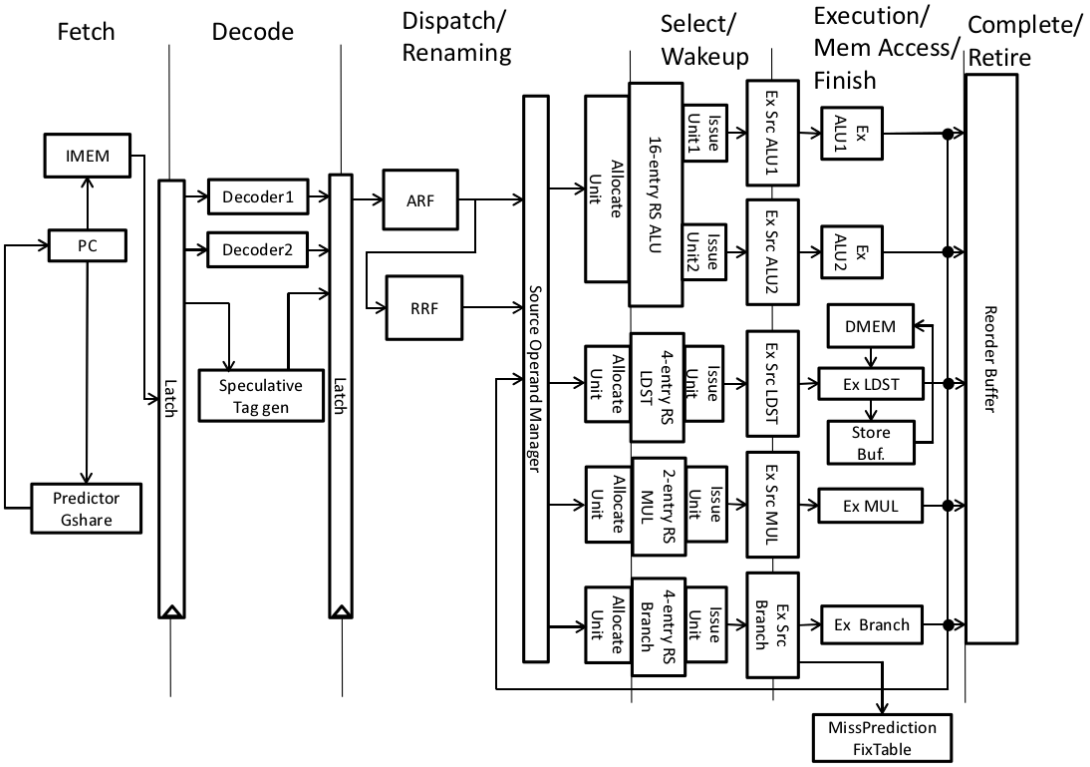
A cada ciclo de relógio, o processador busca duas instruções de uma thread de tempo real e duas instruções de uma thread sem requisitos de tempo real. Um pequeno circuito combinacional verifica quais instruções podem ser executadas em paralelo e as envia para os demais estágios do pipeline. Caso se verifique que as duas instruções da thread de tempo real podem ser executadas em paralelo, as instruções da outra thread são descartadas da execução naquele instante. Já se não houver ILP a ser explorado na thread de tempo real, uma instrução da thread normal é escalonada para ser executada junto com a instrução da thread de tempo real. Por fim, caso a thread de tempo real não possua instruções para serem executadas naquele ciclo, o pipeline é utilizado apenas pela outra thread.

O Hivek-RT foi desenvolvido para sistemas embarcados de tempo real mas que também executem tarefas sem requisitos de tempo real como um sistema de log, por exemplo. As threads de tempo real sempre tem prioridade no uso do pipeline, com as demais threads executando nos períodos ociosos das threads de tempo real. Dessa forma, o Hivek-RT é capaz de explorar tanto ILP quanto TLP.

3.5.2 O Processador RIDECORE

O processador RIDECORE (RIsc-v Dynamic Execution CORE) é um processador com capacidade de execução fora de ordem que executa a ISA RISC-V (FUJINAMI et al.,). Sua microarquitetura é baseada na microarquitetura apresentada em (SHEN; LIPASTI, 2013). Ele possui um pipeline de seis estágios e é capaz de buscar duas instruções por ciclo de clock. Possui duas unidades lógico-aritméticas, uma unidade de load/store, uma unidade de salto e um multiplicador sendo assim possível executar até 5 instruções em paralelo. A Figura 10 apresenta o pipeline do RIDECORE.

Figura 10 – Pipeline do Processador RIDECORE

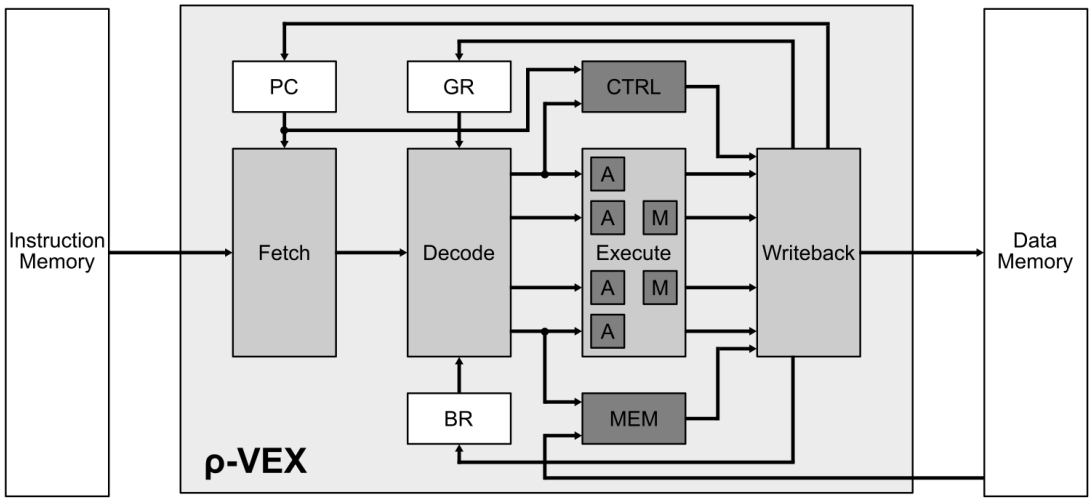


Fonte: (FUJINAMI et al.,)

3.5.3 O Processador ρ-VEX

O processador ρ-VEX é um processador VLIW com suporte para instruções reconfiguráveis (WONG; AS; BROWN, 2008). Em sua configuração padrão, ele é capaz de executar até quatro instruções em paralelo. Possui quatro unidades lógico-aritméticas, dois multiplicadores, uma unidade de salto e uma unidade de load/store. Seu pipeline pode ser visto na Figura 11.

Figura 11 – Pipeline do Processador ρ-VEX



Fonte: (HOOZEMANS; WONG; AL-ARS, 2015)

4 Arquitetura Proposta

A arquitetura proposta é um *multicore* composto por quatro núcleos, ou seja, é um processador *quadcore*. Cada núcleo consiste em um processador PRET acoplado com um array reconfigurável de grão grosso.

O restante deste capítulo está dividido da seguinte maneira: primeiro a arquitetura do núcleo é explicada, começando pelo processador e em seguida pelo array reconfigurável. Depois da explicação do núcleo, é explicado o controlador de memória RAM utilizado e, por fim, vem a explicação sobre o *multicore*.

4.1 Arquitetura do Núcleo

Cada núcleo ou *core* do *multicore* é composto por um processador PRET acoplado ao array reconfigurável. O array é fracamente acoplado ao processador e a comunicação se dá quase toda via as memórias de rascunho compartilhadas entre ambos. O array funciona similar a um coprocessador, executando apenas quando instruído pelo processador.

Uma unidade de Acesso Direto à Memória ou *Direct Memory Access (DMA)* também faz parte do núcleo e é responsável por tratar as transferências de dados entre as memórias de rascunho e a memória principal.

4.1.1 Arquitetura do Processador

Neste trabalho, o Hivek-RT original foi modificado para se adequar melhor às propostas deste trabalho. Como o objetivo é uma arquitetura de alto desempenho para sistemas de tempo real, foi tirado o suporte para a execução de threads sem requisitos de tempo real. Ou seja, a versão modificada agora contém apenas hardware para manter o contexto de threads com requisitos de tempo real. Além disso, enquanto que no processador original havia um total de 8 threads de tempo real, neste trabalho essa quantidade foi reduzida para quatro threads.

As justificativas para essas mudanças são:

- **Redução de área**: ao reduzir a quantidade de threads com contexto salvo em hardware, a área do processador diminui.
- **Diminuição na Latência**: como havia um total de oito threads de tempo real, isso significava que cada thread executava a uma frequência 1/8 do relógio. Ao reduzir a quantidade para quatro threads, cada thread agora passa a executar a uma

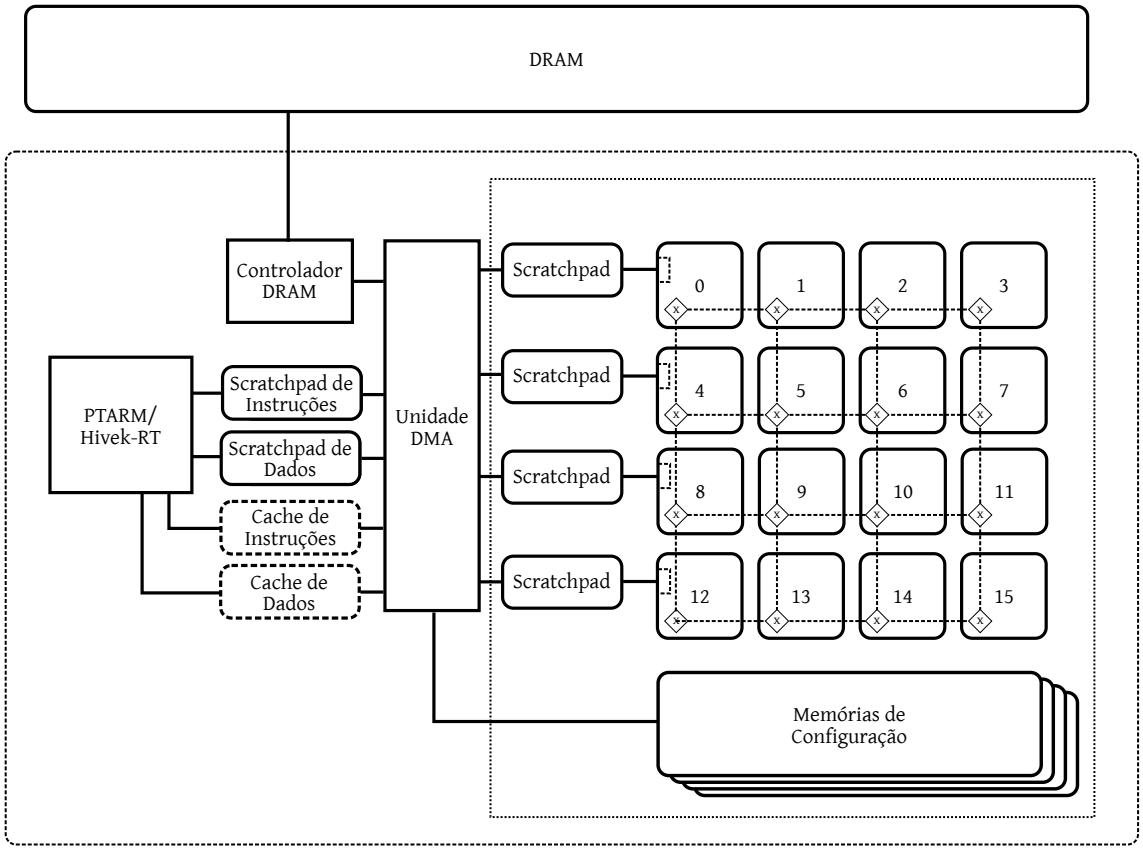
frequência de 1/4 do relógio. A redução de 8 para 4 threads também permite reduzir a quantidade de estágios no pipeline, já que a quantidade de estágios de pipeline deve ser proporcional à quantidade de threads.

- **Arquitetura Multicore**: a construção de uma arquitetura multicore por si só traz uma capacidade muito maior de exploração de TLP do que o inicialmente fornecido apenas utilizando o Hivek-RT. Assim, optou-se por reduzir o tamanho do processador para compensar o aumento de área em virtude do multicore.

4.1.2 Arquitetura do CGRA

A arquitetura do CGRA proposto sofreu modificações profundas no decorrer do desenvolvimento deste trabalho. A primeira versão do CGRA foi baseada no HyCube. Consistia em uma matriz de unidades funcionais cuja principal característica era a capacidade de percorrer vários nós em único ciclo de relógio. A Figura 12 apresenta a primeira proposta do núcleo, apresentando o array em conjunto com os demais elementos do núcleo.

Figura 12 – Diagrama em blocos da primeira versão proposta do núcleo



Fonte: autoria própria

A capacidade de atravessar vários nós da matriz era considerada importante pois permitia uma alocação mais eficaz das operações nas unidades funcionais do CGRA, além

de permitir que operações com dependência de dados fossem executadas em um único ciclo de relógio. Entretanto, essa ideia foi abandonada porque não se conseguiu reproduzir os resultados de síntese. O trabalho original do HyCube sintetizou o mesmo para uma tecnologia de 28nm usando ferramentas as quais o presente trabalho não tem acesso o que impossibilita a replicação desses resultados. Diante disso, a implementação do CGRA inicialmente proposto e que havia sido escrito em SystemC, foi reescrito para Verilog e sintetizado para FPGA, onde se constatou que de fato seria difícil obter a comunicação assíncrona entre unidades funcionais distantes.

Por causa disso, a arquitetura do CGRA proposto foi profundamente modificada. Ao invés de usar uma matriz de unidades funcionais como usado pelo HyCube, optou-se por usar uma abordagem mais heterogênea.

O conceito que sempre guiou o desenvolvimento do CGRA, tanto nas primeiras versões quanto na atual, foi o de que processadores PRET expõem sua arquitetura para o software no sentido de que o hardware não faz nada que não seja do conhecimento do software. Em outras palavras, diferente de outras arquiteturas que muitas vezes escondem detalhes da microarquitetura do software, em uma máquina PRET se defende que o software tenha conhecimento e faça uso de detalhes da microarquitetura, permitindo obter previsibilidade na execução. Por exemplo, enquanto em arquiteturas tradicionais a quantidade de estágios de pipeline é um detalhe da microarquitetura para o programador, em máquinas PRET essa quantidade é importante para realizar cálculos de WCET.

A nova implementação do CGRA foi inspirada no conceito de Parte de Controle e Parte Operativa. Essa metodologia de projeto defende que o hardware deve ser particionado em uma parte responsável pelo processamento dos dados, ou seja, a parte operativa, enquanto que a outra parte fica responsável por controlar a primeira, ou seja, a parte de controle. Um hardware projetado dessa forma apresenta boas características de previsibilidade. Por exemplo, existe o formalismo de máquina de estados finitos que pode ser usado para projetar a parte de controle. Esse formalismo é passível de ser analisado de forma estática o que facilita a previsibilidade. O fato de vários hardwares desenvolvidos sobre a metodologia PC-PO serem descritos a nível de RTL também ajuda no comportamento previsível. Uma descrição em nível RTL ajuda a definir o tempo de propagação de um dado de um local do hardware para outro. Esse tempo nada mais é do que o tempo necessário para atravessar os registradores no meio do caminho entre o registrador origem e o registrador destino.

De forma simplificada, um hardware desenvolvido sobre a metodologia PC-PO pode ser representado como ilustrado na Figura 14. A parte operativa pode ser abstraída como um conjunto de registradores que são alimentados por unidades de computação tais como somadores e subtratores, por exemplo. Um multiplexador decide qual resultado será efetivamente salvo no registrador.

Os sinais de controle responsáveis por controlar as unidades de computação, multiplexadores e registradores são comandados pela parte de controle. A parte de controle pode ser modelada como uma máquina de estados finito. Uma FSM transita entre os estados quando condições de transições são satisfeitas. Essas condições, na prática, consistem muitas vezes de comparações entre os valores contidos nos registradores da parte operativa.

Como exemplo, considere a Figura 13 que apresenta um simples programa e uma possível implementação em hardware seguindo a metodologia PC-PO na Figura 14. É possível perceber os elementos descritos anteriormente. As variáveis são mapeadas para registradores que por sua vez são alimentados por uma estrutura de roteamento formada por multiplexadores. A operação de comparação do laço é usada para alimentar a parte de controle que por sua vez controla os sinais de controle da parte operativa.

Figura 13 – Exemplo de código para realizar multiplicação

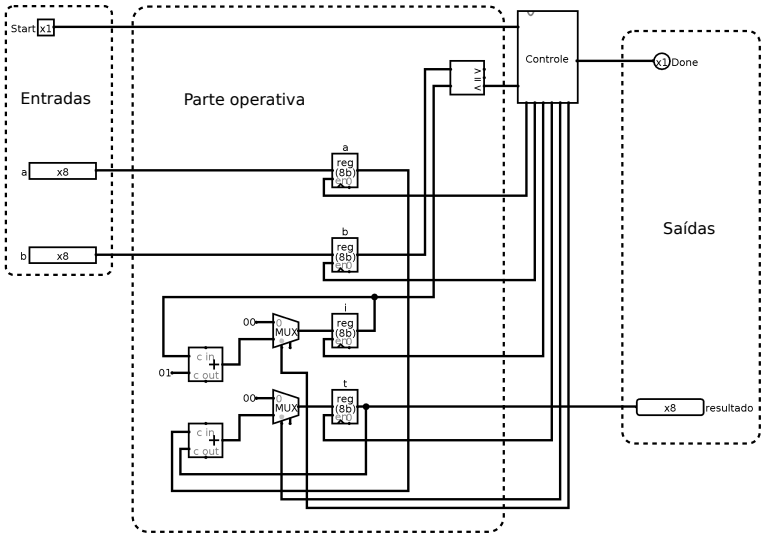
```
int mult(int a, int b) {
    int t = 0;

    for (int i = 0; i < b; ++i) {
        t = t + a;
    }

    return t;
}
```

Fonte: autoria própria

Figura 14 – Parte de Controle e Parte Operativa de um circuito multiplicador



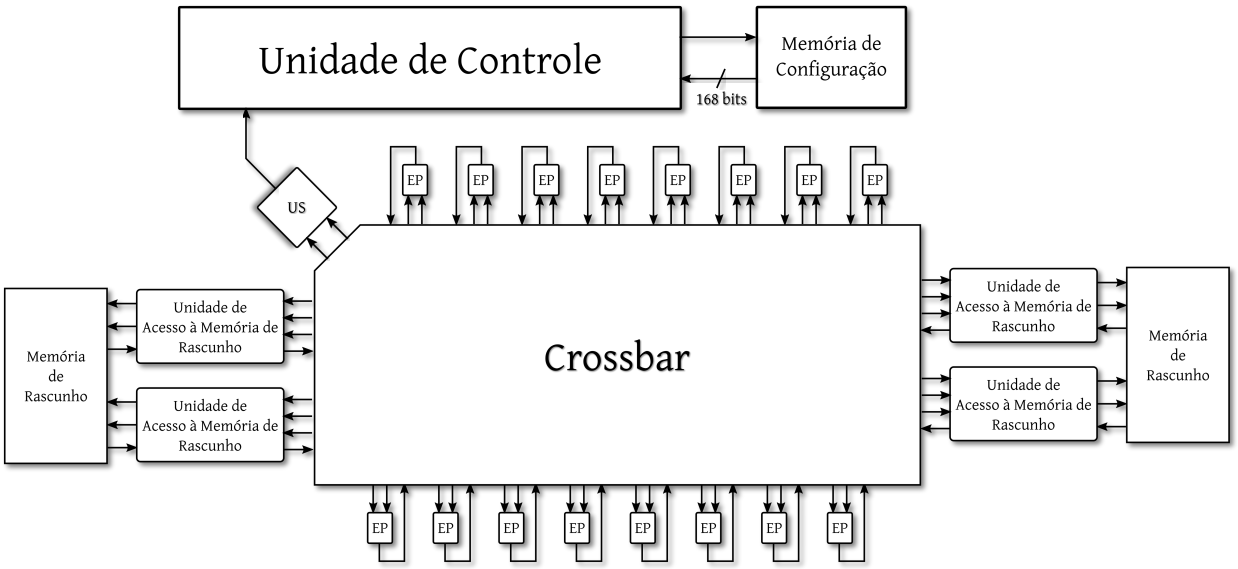
Fonte: autoria própria

O código do exemplo Figura 13 mostra um mapeamento entre a implementação de um algoritmo e uma possível representação em hardware. Apesar de ser um código simples, ele contém boa parte das estruturas de programação que é usada atualmente. Apesar das linguagens de programação terem evoluído bastante nos últimos anos, elas ainda continuam tendo como base estruturas tais como:

- Estruturas de controle tais como laços de repetição e condicionais
- Expressões para geração de resultados
- Variáveis para armazenar resultados de expressões

A arquitetura do CGRA proposto foi desenvolvida tendo como base o que foi apresentado até agora sobre a metodologia PC-PO. A Figura 15 apresenta um diagrama em blocos do CGRA proposto. Como o CGRA deve ser capaz de executar uma gama variada de operações, unidades simples como somadores usados no PC-PO do exemplo da Figura 14 se transformam em Unidades Lógico-Aritméticas (ULA). As estruturas de roteamento, formadas pelos multiplexadores, formam uma espécie de crossbar, permitindo assim a comunicação de um registrador com os demais.

Figura 15 – Diagrama do CGRA proposto



Fonte: autoria própria

Na Figura 16 é mostrado com mais detalhes do array proposto, apresentando uma versão menor do mesmo. Nessa figura é possível identificar quatro Elementos de Processamento (EP). Um EP é composto por uma ULA que realiza a operação propriamente

dita, registradores para armazenar o resultado da operação calculado pela ULA e de multiplexadores que selecionam quais serão os dados de entrada da ULA.

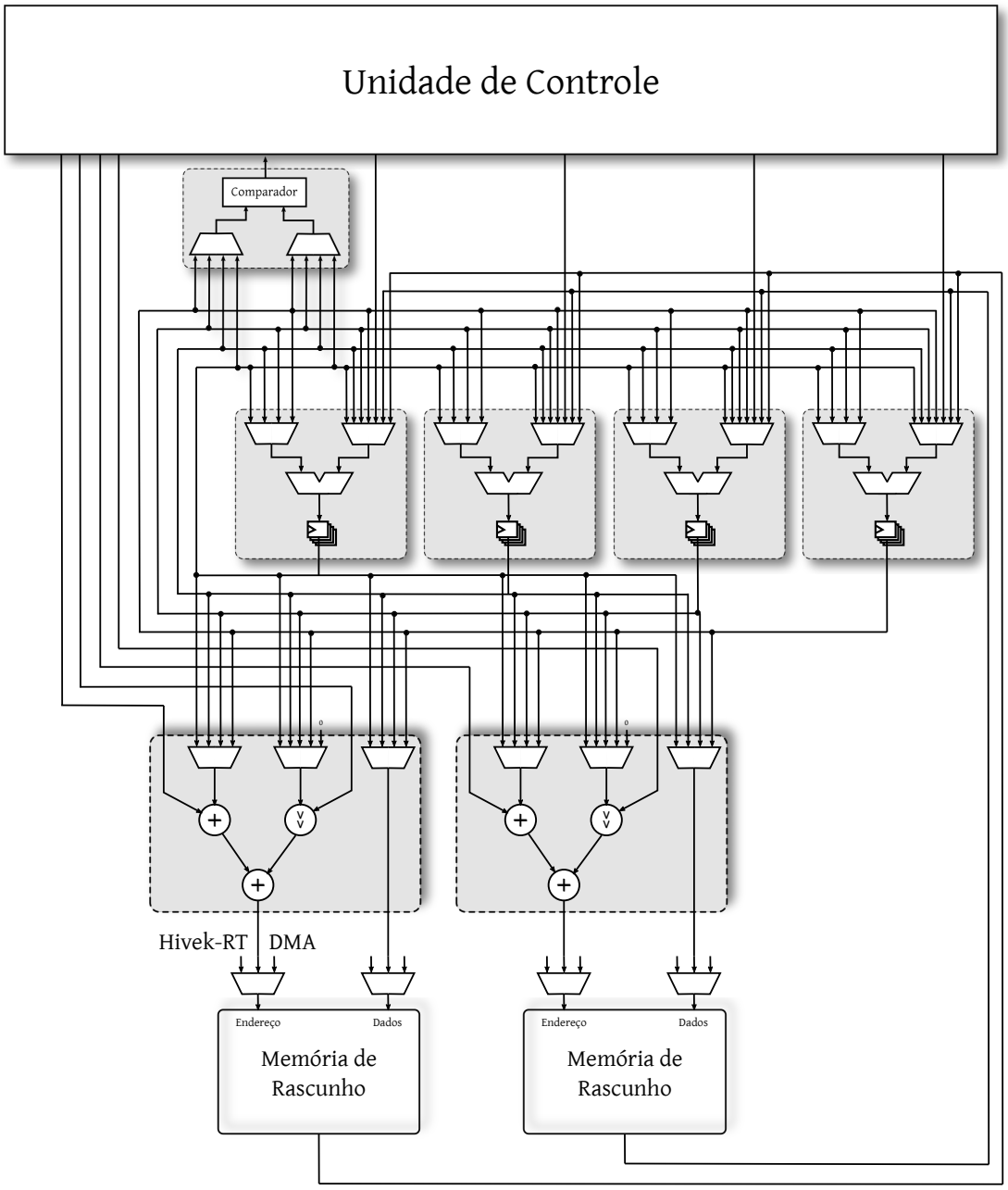
Na parte inferior da Figura 16 encontra-se duas unidades de cálculo de endereço usado por operações de acesso à memória. Um acesso a memória nesse contexto significa um acesso à memória de rascunho. O array (e o processador também) nunca acessam diretamente a memória RAM principal, mas somente as memórias de rascunho. A unidade de Acesso Direto à Memória é quem fica responsável por transferir dados entre as memórias de rascunho e a memória principal.

Na parte superior da Figura 16 encontra-se a unidade de controle e a unidade de salto. A unidade de salto nada mais é do que um datapath para realizar comparações entre registradores. Ela é similar a um EP, com a diferença que ao invés de uma ULA completa tem-se apenas circuitos comparadores. De forma similar ao EP, as entradas do comparador vem de multiplexadores. O resultado gerado pela unidade de salto (que pode ser chamada também de unidade de comparação) é usado pela unidade de controle para decidir qual a próxima palavra de controle a ser executada.

Na implementação desenvolvida nessa tese, foram implementados um total de 16 registradores para representar variáveis. Esse valor foi escolhido por ser um meio termo entre área e programação. Foi levado em consideração que arquiteturas como ARMv8 (WAFAA, 2013), MIPS (SWEETMAN, 2010), SPARC (INC; WEAVER, 1994), RISC-V (WATERMAN et al., 2014) entre outras apresentam 32 registradores de propósito geral. Desses 32 registradores, alguns são usados pelo sistema operacional e outros para tarefas tais como: auxiliar o assembler, topo de pilha, passagem de parâmetros entre outros, de tal forma que a quantidade de registradores de propósito geral que podem ser usados livremente fica na faixa de 20 a 25 registradores. A arquitetura MIPS, por exemplo, considerando os registradores de passagem de argumentos, retorno de função e temporários totalizam 24 registradores. Se considerarmos apenas os temporários, esse número cai para 18 registradores. Já a arquitetura ARM 32 bits usa 16 registradores de propósito geral, estando aí inclusos registradores para ponteiro de instrução, ponteiro de pilha entre outros. Dessa forma, vê-se que 16 registradores é uma quantidade razoável para armazenar variáveis do ponto de vista de conjunto de instruções.

Um outro argumento para a quantidade de 16 registradores são os resultados de síntese. Para essa quantidade de registradores, a ferramenta de síntese já apresentou dificuldades de realizar o roteamento de forma que as restrições de frequência de operação fossem respeitadas. Aumentar a quantidade de registradores implicaria em uma maior quantidade de roteamento o que levaria à falha de respeitar as restrições impostas de frequência de operação, onde nessa implementação tinha-se uma frequência alvo de 50 MHz. Essa frequência foi determinada de duas formas: é uma fração da frequência da velocidade da memória DRAM disponível para prototipagem em hardware, onde essa memória roda

Figura 16 – Versão reduzida do array para fins de ilustração



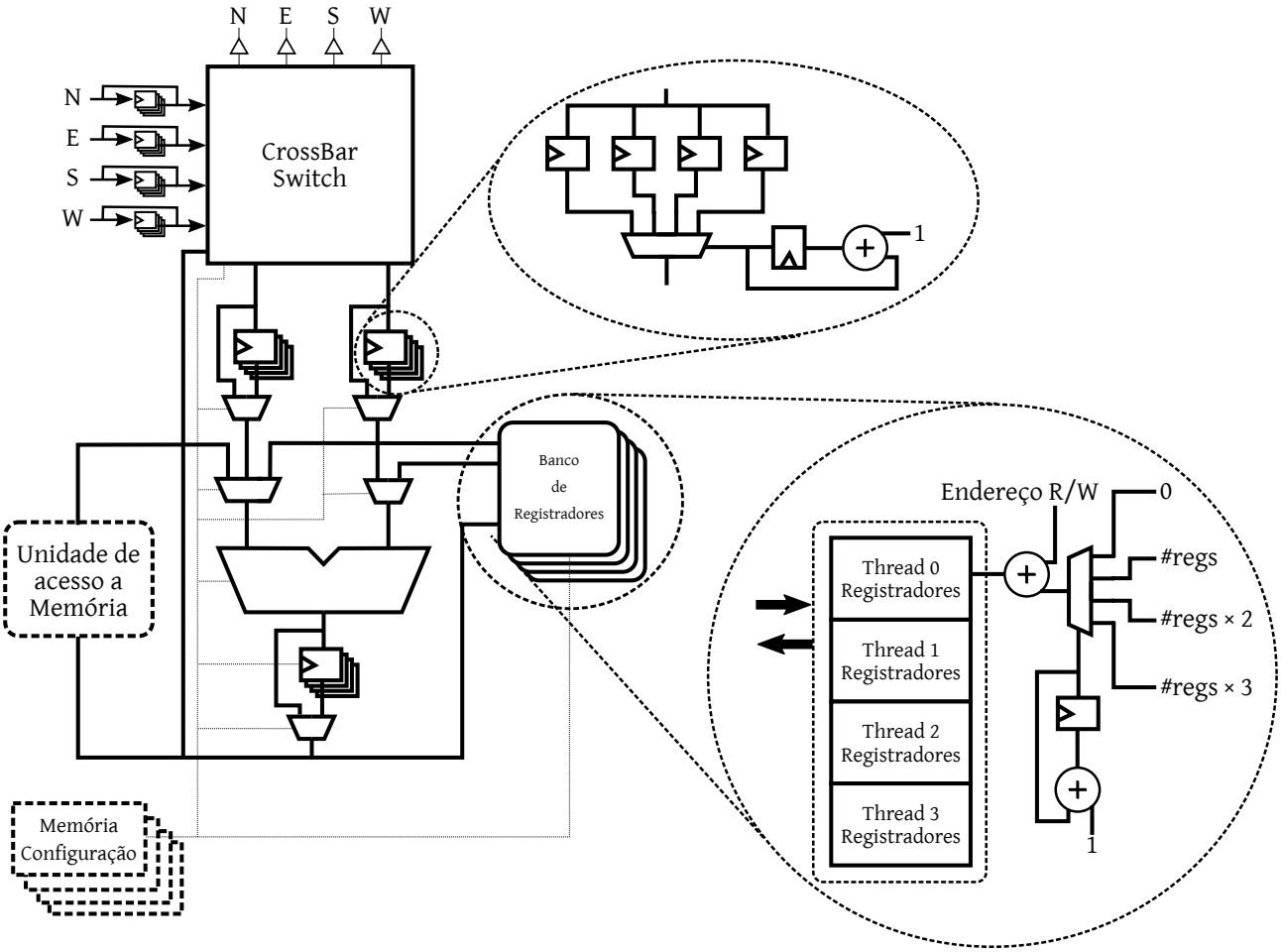
Fonte: autoria própria

a 100 MHz e; é próxima da frequência obtida em prototipação do processador Hivek-RT original. Além disso, sabe-se que um roteamento em crossbar cresce de forma quadrática, de tal forma que a área do CGRA cresceria bastante.

4.1.2.1 Elemento de Processamento

O Elemento de Processamento (EP) é o circuito responsável por realizar uma operação lógico-aritmética e armazenar em seu respectivo registrador. A primeira versão do EP é apresentada na Figura 17. Ela foi desenvolvida com mais registradores, incluindo aí um banco de registradores. Ela tinha mais registradores para que se pudesse testar formas de programação e ver quais registradores deveriam ser mantidos e quais poderiam ser removidos por não agregar conveniência na programação.

Figura 17 – Elemento de Processamento original

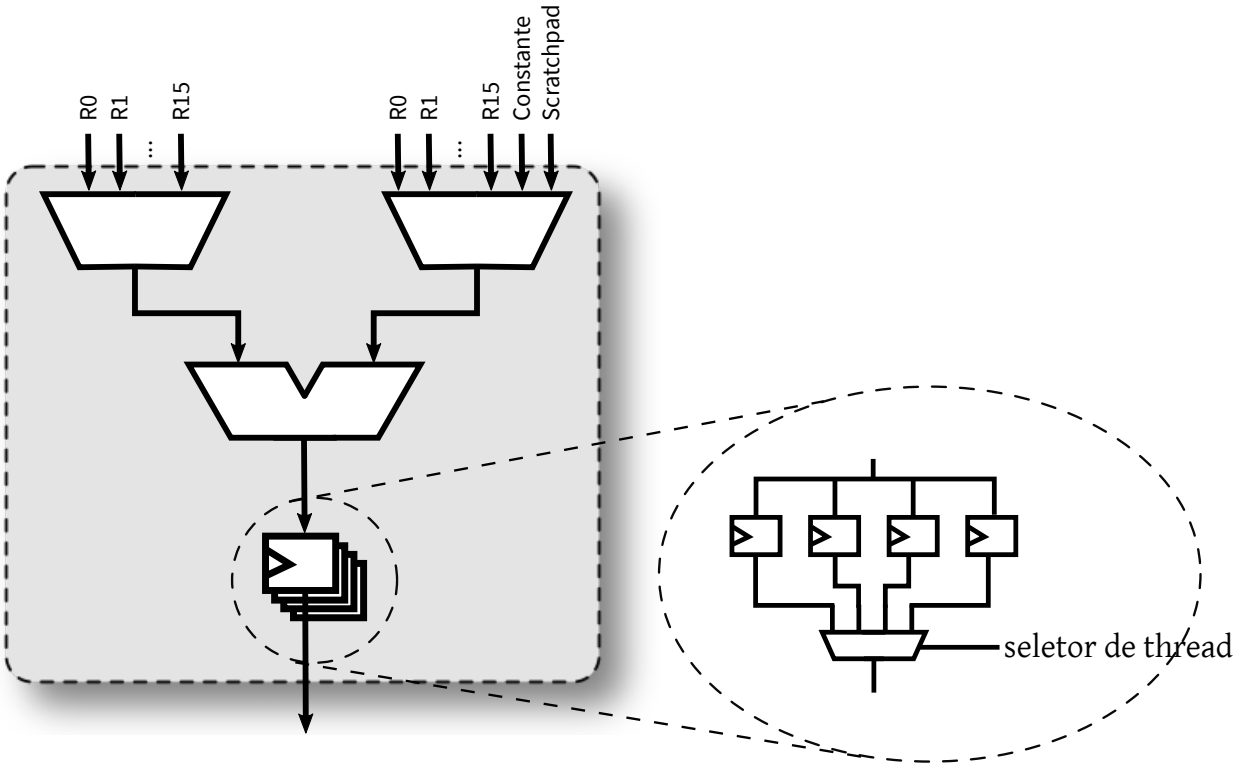


Fonte: autoria própria

Não foi possível explorar muito a primeira versão da arquitetura porque os resultados de síntese impediram que ela fosse utilizável na prática. Diante disso e da modelagem baseada em PC-PO como descrito anteriormente, a nova unidade funcional é apresentada na Figura 18. Ela consiste de um registrador de thread alimentado por uma Unidade Lógico-Aritmética (ULA) que por sua vez recebe seus operandos de uma estrutura de roteamento formada por multiplexadores. Essa estrutura de roteamento formada por

multiplexadores forma uma espécie de *crossbar* ou uma rede mesh totalmente conectada, permitindo que uma unidade funcional se comunique com todas as demais.

Figura 18 – Novo Elemento de Processamento



Fonte: autoria própria

Comparando-se a arquitetura da primeira versão da unidade funcional baseada no HyCube com a de agora, é fácil notar que a atual é bem mais simples. Na versão atual o banco de registradores foi descartado, representando uma economia significativa em termos de área. Também foram removidos registradores de bufferização que também ajuda a diminuir a área ocupada.

Ainda com relação a Figura 18, ela mostra alguns detalhes destacados: o registrador de thread e os multiplexadores. O registrador de thread foi o nome dado ao conjunto de registradores usados para armazenar o contexto das threads que executam no CGRA. Observe que os quatros registradores recebem a mesma entrada e um multiplexador decide qual registrador vai para a saída. Esse multiplexador é controlado por um sinal de controle que decide qual thread está em execução no ciclo de relógio atual. Como cada thread possui seu próprio registrador, não há disputa por esse recurso e como esses registradores são chaveados a cada ciclo de relógio em *round-robin*, isso permite que as características temporais de uma thread não interfira com as demais. Cada thread possui uma fatia de tempo garantida para sua execução e que ocorre a uma frequência regular. Esse é o mesmo

mecanismo usado pelos processadores PRET para isolar uma thread das demais.

A ULA é capaz de executar operações lógico-aritméticas. A Tabela 1 apresenta as uma lista não exaustiva das operações que a atual implementação é capaz de executar. Note que o suporte para instruções de multiplicação e divisão de inteiros nessa implementação é opcional, ou seja, durante a síntese é possível selecionar se haverá ou não unidades de multiplicação e divisão. Por serem unidades custosas em termos de área, essas unidades foram implementadas sem pipeline e com um algoritmo de soma e deslocamento que requer menos portas lógicas. O algoritmo de soma e deslocamento não é o mais rápido porém tem a vantagem de ser executado em número fixo de ciclos de relógio, o que torna o tempo de execução dessas instruções previsível assim com as demais operações que são realizadas na ULA. É importante mencionar que as implementações dos multiplicadores e divisores possuem registradores de thread embutidos a fim de evitar disputa de recursos. Por fim, caso a tecnologia utilizada possua bastante área disponível, nada impede o uso de algoritmos mais rápidos que o de soma e deslocamento, desde que se atente para o fato de que é preciso lidar com o tempo de execução da operação a fim de não perder as características temporais de uma máquina PRET.

Na atual implementação, existe um total de 16 elementos de processamento. Essa quantidade foi decidida através de alguns fatores:

- A primeira versão do array foi baseado no HyCUBe que, por sua vez, utiliza uma matriz de 4x4 de elementos processantes, totalizando 16 elementos de processamento. Dessa forma, a atual implementação do array continua tendo a mesma quantidade de elementos de processamento que a primeira versão do array e do HyCUBE.

- Arquiteturas consolidadas na industria tais como ARM, x86-64, RISC-V, por exemplo, apresentam cerca de 32 a 64 registradores de propósito geral expostos na ISA. Desse total, um grupo desses registradores é para armazenamento de informações tais como ponteiro de pilha e contador de instruções, por exemplo, outros são usados para auxiliar em computações e, por fim, um grupo é usado para efetivamente guardar valores de variáveis. Em análise de códigos assembly com a atual tecnologia de compiladores, foi possível perceber que uma pequena quantidade de registradores é utilizada de fato para o armazenamento de variáveis, mesmo em arquiteturas como as VLIW, que dispõe de mais registradores que o comum. Dessa forma, foi escolhido um total de 16 elementos de processamento que, no caso do array proposto, também significa um total de 16 registradores para armazenamento de variáveis. Os registradores adicionais encontrados em arquiteturas tradicionais que são usados para armazenamento de resultados temporários se tornam menos relevante no array proposto visto que as ULAs são capazes de realizar operações mais complexas que eliminam a necessidade de registradores para armazenamento de resultados

temporários.

Tabela 1 – Lista de operações suportadas pelo array

Categoria	Instruções		Unidade Responsável
Aritméticas	A = B + C A = B – C A = B * C A = B / C	A = B + imediato A = B – imediato A = B * imediato A = B / imediato	Elemento de Processamento
Lógicas	A = B & C A = B | C A = B ^C A = ∼B	A = B & imediato A = B | imediato A = B ^imediato	Elemento de Processamento

Saltos	if (B == C) goto Endereço
	if (B != C) goto Endereço
	if (B <C) goto Endereço
	if (B >= C) goto Endereço
	if (B == 0) goto Endereço
	if (B >0) goto Endereço
	if (B <0) goto Endereço
	if (B != 0) goto Endereço
	goto Endereço
	call Endereço
			Unidade de Decisão de Salto

Deslocamento de Bits	A = B <<C A = B >>C A = B >>>C	A = B <<imediato A = B >>imediato A = B >>>imediato	Elemento de Processamento
Movimentações condicionais	A = B if C == 0 A = B if C != 0 A = 1 if B <C else 0 A = 1 if B >= C else 0		Elemento de Processamento
Acesso à memória De rascunho	A = mem[B + imediato] A = mem[B + C + imediato] A = mem[B + C <<1 + imediato] A = mem[B + C <<2 + imediato] A = mem[B + C <<3 + imediato] mem[B + imediato] = A mem[B + C + imediato] = A mem[B + C <<1 + imediato] = A mem[B + C <<2 + imediato] = A mem[B + C <<3 + imediato] = A		Unidade Acesso à memória

Fonte: autoria própria

4.1.2.2 Unidade de Decisão de Salto

A Unidade de Decisão de Salto (US) é responsável por decidir por qual próxima instrução da configuração será executada. Ela recebe dois operandos através dos multiplexadores que são então usados por um circuito de comparação como mostrado na Figura 16. O resultado dessa comparação é então utilizado por uma Unidade de Controle para decidir qual a próxima instrução a ser executada ou então retornar o controle para o processador.

4.1.2.3 Unidade de Controle

A Unidade de Controle (UC) é responsável por coordenar as demais unidades e acessar a Memória de Configuração através da execução de uma Máquina de Estados Finito (*Finite State Machine - FSM*).

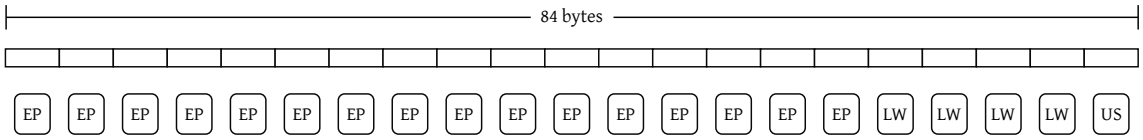
Ao ligar a arquitetura, o CGRA entra em um estado de espera e fica aguardando uma ordem de *start* vinda do processador. Ao receber a ordem de *start*, o CGRA inicia sua execução começando pela primeira palavra de instrução que vem como argumento na ordem de *start*. O CGRA passa então para um *loop* formado pelo estado de execução. Esse loop só é quebrado quando aparece alguma instrução de retorno do controle para o procesador.

Na unidade de controle existem um total de 21 decodificadores de instruções sendo: 16 decodificadores para decodificar as instruções dos 16 elementos de processamento, 4 decodificadores para controlar as unidades de acesso à memória de rascunho e uma para controlar a unidade de salto.

4.1.2.4 Memória de Configuração

A Memória de Configuração é responsável por armazenar as operações a serem realizadas em cada uma das unidades contidas no array. Como cada unidade é controlada por uma palavra de 32 bits e existem 16 Unidades Funcionais, 4 unidades de Cálculo de Endereço e 1 Unidade de Salto, totalizando 21 unidades, são então necessários 672 bits para configurar as unidades ou 84 bytes. Esses 84 bytes podem ser vistos como formando uma espécie de instrução VLIW (*Very Long Instruction Word*) e que devem ser fornecidos a cada ciclo de relógio para instruir o que cada unidade deve realizar como mostrado na Figura 19.

Figura 19 – Palavra de configuração com 84 bytes



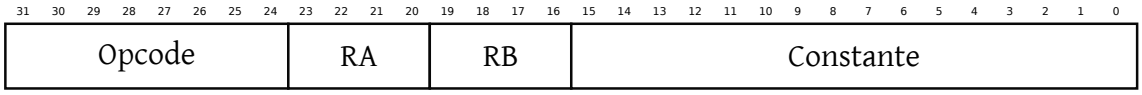
Fonte: autoria própria

Do ponto de vista de interface, a Memória de Configuração é, então, uma memória com largura de barramento de 84 bytes. Do ponto de vista de implementação, essa memória pode ser de fato uma única memória com largura de barramento de 84 bytes ou então memórias de largura de barramento menores combinadas para totalizar os 84 bytes e sendo controladas pelo mesmo registrador de endereço. Nessa tese, a descrição em Verilog da Memória de Configuração é descrita como uma única memória e é deixado para a ferramenta de síntese decidir se a mesma vai inferir uma única memória de barramento de 84 bytes ou se vai combinar memórias menores.

Existem apenas dois formatos de instrução. O primeiro formato, que pode ser visto na Figura 20, tem como campos o opcode da operação a ser realizado, os registradores RA

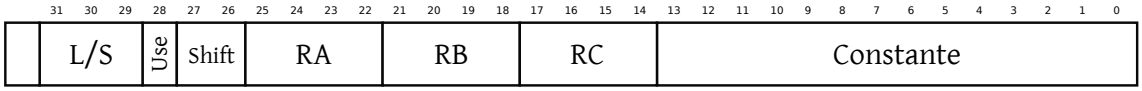
e RB que vão servir de fonte e uma constante de 16 bits. RA e RB podem se referir a um dois 16 registradores ou a uma das quatro memórias de rascunho dependendo do opcode. O segundo formato, que pode ser visto na Figura 21, é usado para configurar as unidades de acesso à memória.

Figura 20 – Palavra de configuração de um Elemento de Processamento



Fonte: autoria própria

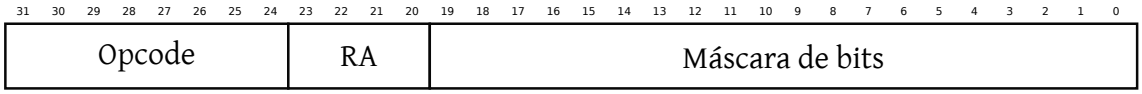
Figura 21 – Palavra de configuração para operações Load/Store



Fonte: autoria própria

O primeiro formato de instrução também é usado pela unidade de salto e possui algumas peculiaridades. Além de fornecer operações tradicionais de salto condicional, ela também é usada para realizar operações com atraso. O campo RB é interpretado junto com o campo Constante como uma máscara de bits como mostrado na Figura 22. Cada bit corresponde a um dos 16 EPs e das quatro unidades de acesso à memória. O exemplo da Figura 23 mostra um grafo representando instruções e suas dependências. Logo abaixo encontra-se uma máscara de bits. Instruções com bit 0 são executadas no ciclo corrente, enquanto que as instruções com bit 1 são executadas no próximo escalonamento. Isso permite reduzir o tamanho das configurações armazenadas na memória de configuração. Em casos onde se exista pouco ILP, uma mesma palavra pode ser utilizada para codificar dois ciclos de execução.

Figura 22 – Palavra de configuração para operações com atraso

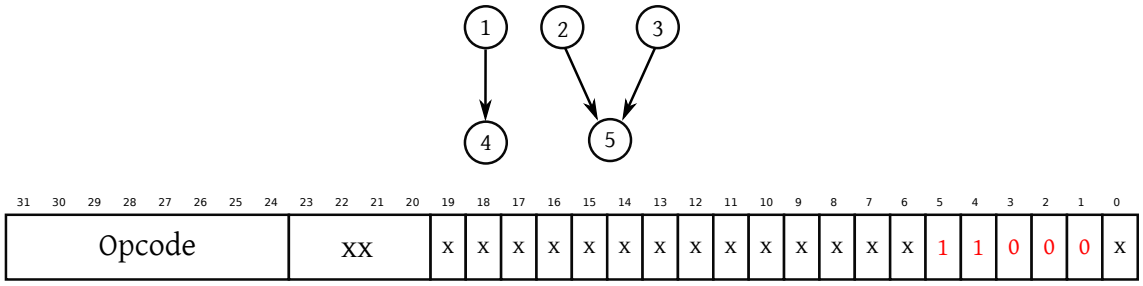


Fonte: autoria própria

4.1.2.5 Unidade de Cálculo de Endereço

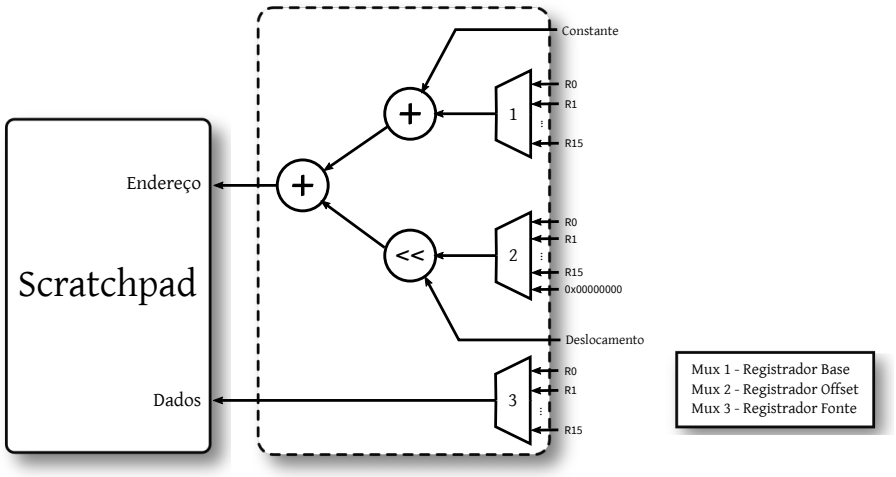
O CGRA possui quatro unidades de cálculo de endereço (em inglês, Address Generation Unit - AGU). Essas unidades são usadas para calcular endereços para as

Figura 23 – Exemplo de execução com atraso



Fonte: autoria própria

Figura 24 – Unidade de Acesso à Memória

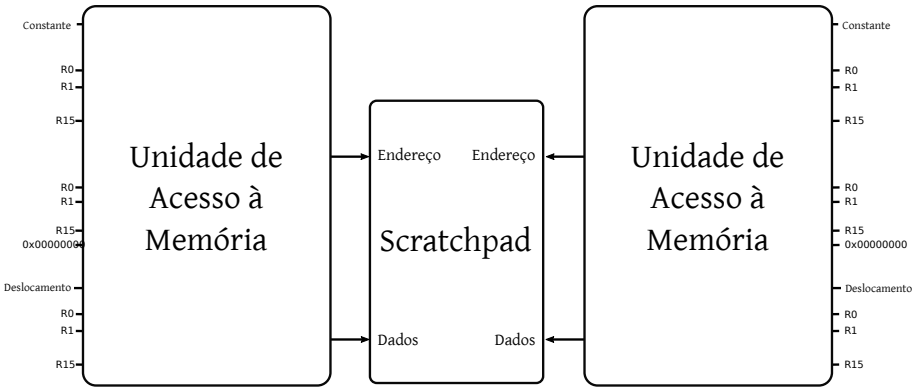


Fonte: autoria própria

operações de *load* e *store* executadas no CGRA. Por serem quatro AGUs, é possível realizar quatro operações de acesso à memória simultaneamente. Por acesso à memória, deve-se entender uma escrita ou leitura em uma *scratchpad*.

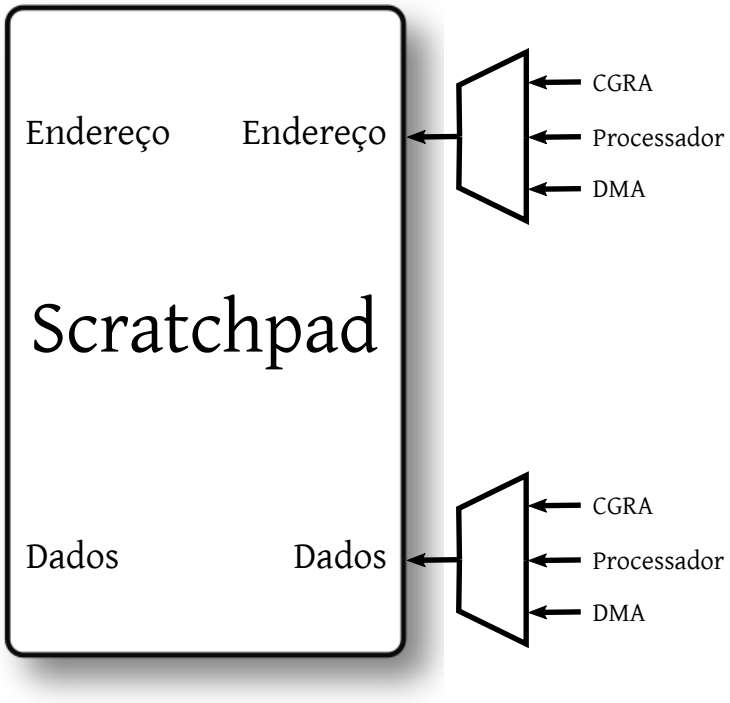
Cada AGU recebe como operandos um registrador que funciona como registrador base; um registrador que funciona como um offset e que pode sofrer deslocamentos de 0 até 3 bits; e uma constante de offset. As AGUs foram projetadas dessa forma para poder prover aceleração no cálculo de endereço do acesso bem como diminuir a quantidade de instruções a serem executadas no CGRA, levando a um bitstream de configuração menor. Isso é possível porque vários dos cálculos de endereços gerados pelos compiladores podem ser resumidos na Equação 4.1. Essa equação diz que um endereço é composto por um registrador de base somado a um registrador de *offset* (que pode ou não conter um

Figura 25 – Scratchpad Dual Port



Fonte: autoria própria

Figura 26 – Origem dos endereços para a scratchpad



Fonte: autoria própria

deslocamento) somado a uma constante fixa.

$$endereço = r\_base + r\_offset << immd + offset \tag{4.1}$$

Como exemplo, considere o simples laço de repetição apresentado abaixo:

```
int zero(int* p, int n) {
    for (int i = 0; i < n; ++i) {
        p[i] = 0;
    }
}
```

Cujo laço de repetição pode ter a seguinte tradução em assembly (extraído do gcc usando a flag -S):

```
$L3:
    lw $2,4($fp)     ; $2 = i
    sll $2,$2,2      ; $2 = i * 4
    lw $3,16($fp)    ; $3 = p
    addu $2,$3,$2    ; $2 = p + i * 4
    sw $0,0($2)      ; mem[p + i * 4] = 0
    lw $2,4($fp)
    addiu $2,$2,1
    sw $2,4($fp)
```

$$endereço = p + i << 2 + 0 \tag{4.2}$$

O cerne do cálculo de endereço dessa operação consiste no deslocamento usando a instrução *sll* que faz um deslocamento à esquerda combinado com uma instrução de soma para se obter o endereço final. Esse endereço é então usado por uma operação de acesso à memória. Usando a Equação 4.1 como base, o cálculo do endereço do exemplo ficaria como mostrado na Equação 4.2, onde p é o registrador base, i é o registrador de *offset* deslocado em duas vezes (multiplicado por quatro) com uma constante de *offset* igual a zero.

A partir desse padrão de acesso é que se chegou a configuração mostrado na Figura 24. Dessa forma, as três instruções necessárias para a escrita ou leitura de um dado são executadas em uma única etapa em virtude da estrutura combinacional do circuito. Sabendo-se que existe quatro unidades de acesso à memória, existe um potencial de se paralelizar um total de 12 instruções que em um processador *single-issue* se levaria, no mínimo, 12 ciclos de relógio.

A Figura 26 mostra que, na verdade, a scratchpad do CGRA recebe seu endereço e valor a partir de multiplexadores. O multiplexador conectado à porta de endereço multiplexa entre endereços vindos do processador, do CGRA e do controlador de memória RAM. O multiplexador é controlado por um circuito de mapeamento, já que cada scratchpad consiste em um espaço de endereço diferente. Dessa forma, o processador e o controlador de

memória RAM podem alimentar o CGRA com dados e vice-versa. E como cada scratchpad consiste em um espaço de endereçamento diferente, é possível ocorrer leituras e escritas em paralelo. Por exemplo, a unidade DMA pode estar escrevendo em uma das scratchpads enquanto o processador lê de outra. A Figura 26 apresenta como o CGRA, o processador e o controlador de memória RAM interagem através da scratchpad.

4.2 Controlador de Memória

Em arquiteturas PRET, a hierarquia de memória é um ponto de atenção uma vez que essa tem influência direta nas características do processador. Por exemplo, se o acesso a um dado armazenado na hierarquia de memória apresenta latência variável, isso degrada a previsibilidade e repetibilidade do processador. No presente trabalho, a hierarquia de memória é constituída por dois níveis. No nível mais próximo do processador e do CGRA se encontram memórias scratchpads. Como já discutido, esse tipo de memória apresenta comportamento previsível, com latências de acesso bem conhecidas e constantes. No segundo nível, encontra-se uma memória principal utilizando tecnologia *Synchronous Dynamic Random-Access Memory (SDRAM)*.

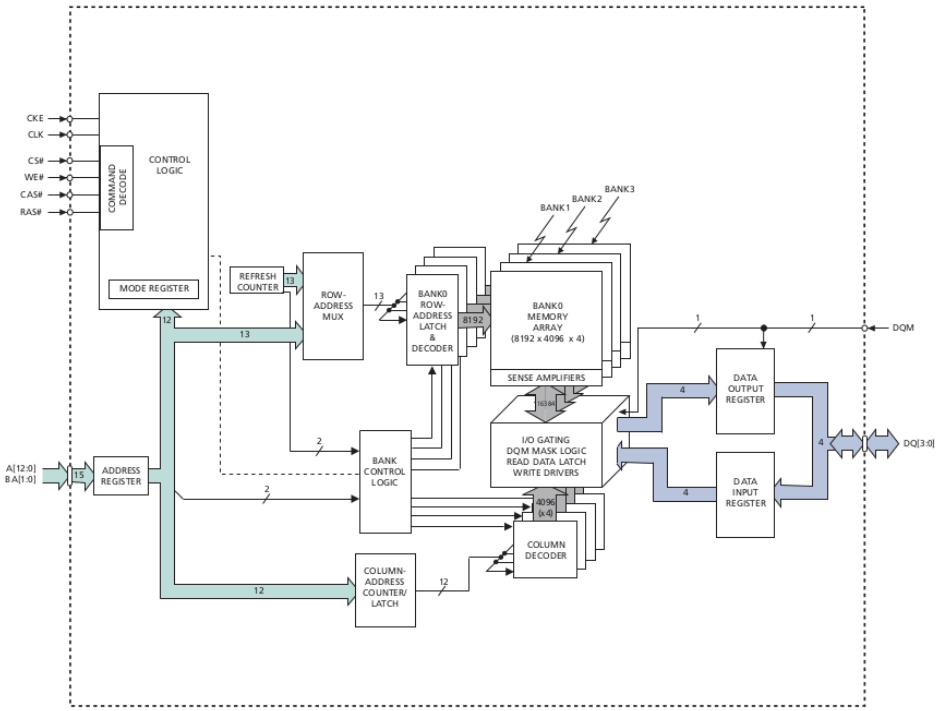
Processadores convencionais trabalham com uma hierarquia de memória onde a memória principal utiliza tecnologia SDRAM, sendo a versão mais comum as com tecnologia Double Data Rate (DDR), sendo esta a sucessora das memórias *Single Data Rate (SDR)*.

Memórias SDRAM, sejam elas SDR ou DDR, costumam ter uma arquitetura como a apresentadas na Figura 27. Internamente, uma memória SDRAM é divida em bancos que podem ser operados em paralelo. Esses bancos compartilham os barramentos de controle e de dados. Diferença entre as versões de memórias DDR e SDR, por exemplo, consistem em: diferentes larguras de barramento de dados, quantidade de bancos, tensões de operação entre outras. O ponto principal, porém, é que em ambas tecnologias usam o mesmo conceito de bancos paralelos que dividem os barramentos de controle e dados.

O componente responsável por realizar a interface entre o processador e uma memória SDRAM é conhecido por Controlador de Memória. A função do controlador de memória é traduzir endereços e dados vindos do processador para o formato esperado pela memória SDRAM e vice-versa. Além disso, ele também é responsável por gerenciar o estado da SDRAM. Memórias SDRAM usam capacitores para o armazenamento dos bits. Esses capacitores precisam ser recarregados de tempos em tempos e controlador de memória fica responsável por esse gerenciamento de recarga dos capacitores.

Controladores convencionais tratam a memória SDRAM como sendo um dispositivo monolítico. Em outras palavras, esses controladores enxergam os bancos que compõem a memória SDRAM como consistindo de um único espaço de endereçamento. Por causa disso,

Figura 27 – Arquitetura de uma memória SDR SDRAM



Fonte: Edwards e Lee (2007, p. 24)

disputas pelo acesso à memória SDRAM é uma consequência em arquiteturas multicore.

O controlador de memória SDRAM desenvolvido nesse trabalho usa uma abordagem similar ao apresentado pelo controlador apresentado em (LIU et al., 2012). Ao invés de enxergar a memória como um recurso monolítico, ele expõe os bancos que compõem a memória. Na atual implementação, o controlador foi implementado para trabalhar com uma memória SDR SDRAM. A escolha dessa memória foi feita porque se tinha acesso a uma placa de prototipação com FPGA e a memória SDR SDRAM em questão.

O acesso à memória é feito através de multiplexação por divisão de tempo (*Time-Division Multiplexing - TDM*). Cada banco é visto como um recurso independente dos demais e que compartilha o barramento de controle e dados com os demais bancos.

De forma resumida, cada banco é visto como uma matriz composta por linhas e colunas. Para realizar uma operação de escrita ou leitura, deve-se escolher uma linha da matriz. Essa linha é carregada para um conjunto de *latches* sobre os quais se realizam de fato a leitura ou escrita. Após o fim da operação, os valores carregados nos *latches* são transferidos de volta para a linha original. Uma operação de carga de uma linha para os latches tem o nome de ACTIVE, operações de leitura e escrita realizada em cima dos latches tem o nome de READ e WRITE, respectivamente. Uma operação de

carga dos latches para a linha da matriz é conhecida por PRECHARGE. Assim, enquanto que do ponto de vista do processador um simples operação de load ou store consiste em acessar uma posição, essa instrução deve ser traduzida pelo controlador de memória em um conjunto de operações ACTIVE, READ/WRITE e PRECHARGE, esse é o funcionamento de forma resumida de um controlador de memória SDRAM, seja ele qual for.

Além desses comandos, também existem comandos para manutenção da memória. Como os capacitores das SDRAMs se descarregam com o passar do tempo por causa de fuga de corrente, eles precisam ser recarregados de tempos em tempos. Uma recarga consiste em ler uma linha para os latches e em seguida de volta para a linha. A essa operação se dá o nome de REFRESH. Esse processo deve ser repetido a uma dada frequência para todas as linhas a fim de evitar corrupção dos dados. As operações de REFRESH realizam as mesmas etapas de uma operação READ, a diferença é que a operação de REFRESH nenhum dado é enviado para o processador e tem como principal função a de recarregar os capacitores. Já uma operação de READ envia os dados de volta para o processador e a recarga dos capacitores é apenas uma consequência.

A memória SDR SDRAM utilizada possui quatro bancos que são multiplexados a cada 16 ciclos de relógio. O valor 16 foi escolhido por ser um múltiplo da quantidade de threads e que ao mesmo tempo satisfaz as restrições de tempo impostas pela tecnologia SDR SDRAM. Por exemplo, existe um parâmetro conhecido por tRRD, que é a quantidade de ciclos mínima entre um comando ACTIVE realizado ao banco A e o comando ACTIVE realizado ao banco B. Como cada banco é controlado por uma thread diferente e cada thread pode iniciar uma operação de acesso à memória em paralelo com as demais, deve-se respeitar o tempo definido por tRRD a fim de não ocorrer corrupção de dados. Na prática, isso significa que o controlador de memória reserva 16 ciclos de relógio para cada banco e as operações de escrita e leitura ocorrem em múltiplos dessa quantidade.

Por serem intercalados a cada 16 ciclos com a memória RAM e o controlador rodando a 100 MHz e o processador rodando a 50 MHz com quatro threads, isso cria uma sincronização de que a cada 64 ciclos uma thread executando no processador coincide com seu respectivo banco sendo acessado.

4.2.1 Operação de Refresh

As operações de REFRESH são realizadas de tempos em tempos pelo controlador. Num controlador de memória RAM convencional, as operações de REFRESH são relizadas automaticamente através do comando AUTO REFRESH. As memórias SDRAM apresentam internamente circuitos dedicados à manutenção das operações de REFRESH, bastando apenas que o controlador faça um pedido de execução ao módulo de memória SDRAM.

Nesse trabalho o controlador não usa o comando AUTO REFRESH. Ao invés disso, o próprio controlador mantém seu monitoramento de quando é preciso ou não fazer uma operação de REFRESH. O controlador implementa uma pequena lógica: primeiro ele verifica se está na hora ou não de realizar uma operação de REFRESH e caso esteja, ele executa. Se por acaso não for o instante de fazer uma operação de REFRESH, ele verifica se há alguma solicitação pendente de leitura ou escrita, contabilizando quantos dados serão transferidos por aquela solicitação. Ele então faz um pequeno cálculo matemático para ver se o tempo da transferência irá colidir com o tempo da operação da execução da próxima operação REFRESH. Caso isso aconteça, o controlador funciona de forma conservadora, executando um comando de REFRESH extra. Isso significa que o controlador proposto realiza muito mais operações de REFRESH do que um controlador convencional. Em contrapartida, se obtém um melhor previsibilidade do tempo de acesso à memória.

4.2.2 Operação de Leitura

Uma operação de leitura se dá como apresentado na Figura 28, que apresenta o instante da rajada de dados. Os primeiros 16 ciclos são dedicados a realizar uma operação de REFRESH. Em seguida, é executado um comando de ACTIVE seguido por um comando READ que dá início a rajada de dados. Os próximos ciclos são usados para continuar a rajada de dados. Por fim, é executado um comando de precharge deixando o banco preparado para iniciar uma nova operação, seja ela de escrita ou leitura.

O tempo de execução de uma operação de leitura pode ser calculado pelas equações 4.2 e 4.3. Nessas equações, um ciclo é considerado o intervalo de tempo dedicado de arbitragem de cada banco, que nesse caso corresponde a 16 ciclos de relógio. Os dois primeiros números 1 que aparecem na equação é referente ao ciclo de REFRESH e do ciclo de ativação da linha. O último número 1 na equação é em virtude do comando PRECHARGE.

$$ciclos = 1 + 1 + \left\lceil \frac{bytes}{largura \times 12} \right\rceil + 1 \tag{4.3}$$

$$tempo = ciclos \times 64 \times período \tag{4.4}$$

Como exemplo, considere uma memória SDR SDRAM rodando a 100 MHz com barramento de 16 bits (2 bytes) e que se deseja ler 1024 bytes. Assim, $período = 1/100MHz = 10ns$, $bytes = 1024$ e substituindo os valores nas equações resulta em

$$ciclos = 1 + 1 + \left\lceil \frac{1024}{2 \times 12} \right\rceil + 1 \tag{4.5}$$

$$ciclos = 46 \tag{4.6}$$

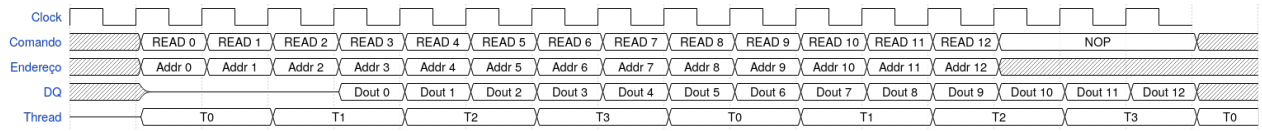
$$tempo = 46 \times 64 \times 10ns \tag{4.7}$$

$$tempo = 29440ns \tag{4.8}$$

Dessa forma, uma thread que requisite 512 bytes à memória sabe que essa transferência levará 29.440 ns para ser executada.

A Figura 28 apresenta as formas de onda durante uma operação de leitura sendo realizada pela thread 0. A cada ciclo de relógio é enviado um comando de leitura e o respectivo endereço de acesso. Depois de um atraso de dois ciclos de relógio, os dados começam a aparecer no barramento de dados. O fim do período de leitura consiste em comandos NOP para respeitar os tempos de intervalo entre uma thread e outra.

Figura 28 – Exemplo de leitura na memória RAM



Fonte: autoria própria

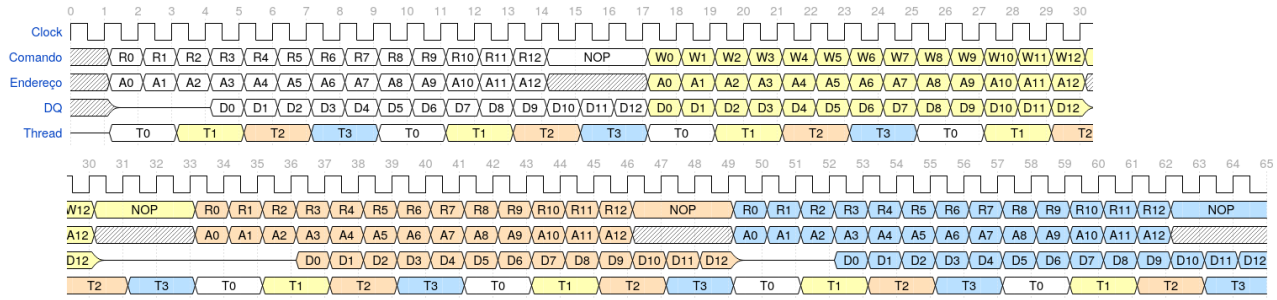
4.2.3 Operação de Escrita

A operação de escrita se dá de maneira similar às operações de leitura. Um processo de escrita se inicia da mesma forma: com um comando primeiro de REFRESH seguido de outro comando da ativação de linha. Em seguida tem-se o início da rajada de dados e por fim o comando de PRECHARGE.

A Figura 29 apresenta um exemplo das quatro threads acessando a memória RAM. A thread 1 está realizando uma operação de escrita enquanto as demais estão realizando operações de leitura. Note que a diferença de uma operação de leitura para uma operação de escrita é onde os NOPs são inseridos e o instante que os dados aparecem no barramento. Nesse exemplo, um total de 96 bytes são transferidos a cada 65 ciclos de relógio, o que representa uma taxa de transferência de $96bytes/65ciclos \times 100MHz = 147MB/s$. Considerando que o limite teórico para a memória utilizada é de 200 MB/s, isso significa

que o controlador consegue atingir 73% da taxa de transferência máxima teórica e que cada thread tem $147MB/s/4 = 36MB/s$ a sua disposição para ser utilizado.

Figura 29 – Exemplo de múltiplos acessos à memória RAM



Fonte: autoria própria

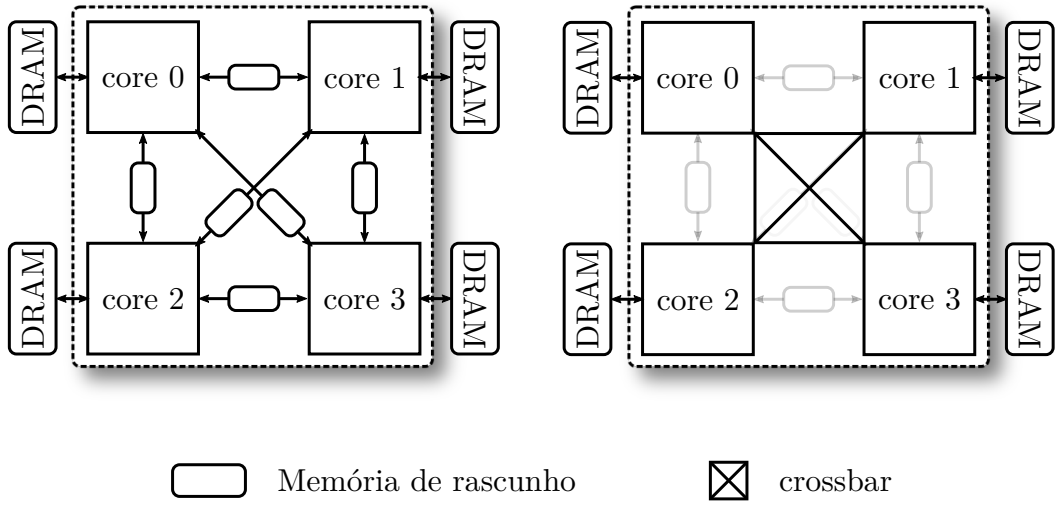
4.3 Arquitetura do Multicore

A arquitetura do processador *multicore* é relativamente simples e pode ser vista na Figura 30. Ela é composta por quatro núcleos (um *quadcore*) onde cada núcleo tem as características apresentadas anteriormente. Como a quantidade núcleos é pequena (apenas quatro), a comunicação entre os mesmos é realizada através de uma *crossbar*. A vantagem da *crossbar* aqui é que ela permite uma comunicação ponto-a-ponto entre os núcleos e portanto não há disputa por recursos como seria no caso de um barramento, por exemplo. Isso é importante pois ao evitar a disputa por um recurso, facilita a implementação dos conceitos das arquiteturas PRET. Uma *crossbar* também apresenta uma implementação mais simples do que uma rede em chip, por exemplo, uma vez que a quantidade de elementos se comunicando é pequena e não é preciso se preocupar com estruturas como roteadores, por exemplo.

Além da *crossbar*, também existem seis memórias *scratchpad* ligando os núcleos ponto a ponto a fim de permitir uma maior transferência de dados. Na atual implementação, essas memórias *scratchpads* são implementadas usando memórias *block RAM* comumente encontradas em FPGAs. Block RAMs costumam possuir duas portas de acesso, de tal forma que uma porta fica com um núcleo e a outra porta com outro núcleo. Dessa forma, configurações como produtor-consumidor são facilmente implementadas, com um núcleo produzindo valores e escrevendo na *scratchpado* enquanto que o núcleo na outra ponta consome esses dados.

Como cada núcleo possui uma hierarquia de memória isolada dos demais núcleos, foi implementado um circuito que permite a troca de dados entre as hierarquias de memórias

Figura 30 – Arquitetura do *multicore*



Fonte: autoria própria

através de comandos explícitos executados pelo processador. Após uma breve negociação entre os núcleos, é possível realizar transmissões de dados em *broadcast*. Por exemplo, um dos experimentos realizados neste trabalho foi a multiplicação de matrizes. Ao invés de transferir a mesma matriz três vezes, uma vez para cada núcleo, a matriz é transferida uma única vez via broadcast. Também é possível transferir de uma scratchpad de um núcleo diretamente para a scratchpad de um outro núcleo sem a necessidade do processador ficar intervindo, sendo portanto um mecanismo semelhante ao *Direct Memory Access - DMA*. O processador só precisa configurar o destino e a quantidade de dados a serem transferidos e a unidade de DMA realiza o resto da transferência.

Cada núcleo possui acesso a uma memória DRAM privada. Essa foi uma decisão de projeto para que a arquitetura tivesse uma alta vazão de dados e a latência de acesso fosse diminuída. Caso existisse apenas uma única memória compartilhada entre os quatro núcleos, essa memória seria acessada via multiplexação no tempo, acarretando longas latências de acesso e reduzindo o desempenho.

A quantidade de memórias SDRAM foi escolhida baseando na atual tecnologia utilizada pelas placas-mães que possuem normalmente quatro slots de memória RAM, sendo portanto uma tecnologia já bem difundida no mercado. Como se desejava que cada núcleo tivesse acesso a uma memória SDRAM privada e como se optou por utilizar quatro memórias SDRAM em virtude dos quatro slots, a quantidade de núcleos foi definida tomando como base a quantidade de memórias SDRAM. É por isso que neste trabalho se implementou um processador quadcore.

4.4 Modelo de Programação

Existe um compilador em desenvolvimento que já é capaz de compilar alguns programas simples, porém que está longe de ser capaz de compilar programas mais complexos. A linguagem escolhida é uma linguagem própria inspirada em C e Python. De C é utilizado toda a semântica enquanto que de Python é utilizado a sintaxe. Além disso, a linguagem também é inspirada nas linguagens ForeC, PRETC, Lingua Franca e em CUDA. Todas essas linguagens são, na verdade, extensões da linguagem C. As três primeiras são extensões para o desenvolvimento de sistemas de tempo real enquanto que a última é um extensão para escrever kernels a serem executados em uma GPU.

A linguagem proposta aqui não traz nenhum conceito novo, apenas agrega os conceitos das linguagens mencionadas. De ForeC, PRETC e Lingua Franca é absorvido o conceito da escrita de programas para tempo real para serem executadas no processador enquanto que CUDA serve de inspiração para a escrita de código que será executado no array.

Os motivos pelos quais se optou por desenvolver uma nova linguagem ao invés de adaptar alguma existente são elencadas a seguir:

- Apesar da linguagem C ser bastante difundida, suas ferramentas e a própria linguagem pouco a pouco vão ficando para trás no sentido de auxiliar no desenvolvimento rápido de aplicações. Por exemplo, a linguagem C até hoje não tem um suporte nativo para inferência de tipos bem como para sobrecarga de funções a não ser através de extensões customizadas de compiladores. Essas extensões muitas vezes não são compatíveis entre diferentes compiladores. Outro exemplo, agora sobre as ferramentas, é a dificuldade para se gerenciar inclusão de arquivos tais como linguagens mais modernas o fazem.

- Prover uma sintaxe mais amigável. Esse é um conceito subjetivo, porém como a linguagem Python está se difundindo rapidamente por causa de aprendizado de máquina e um dos motivos para isso é sua sintaxe amigável, optou-se por basear boa parte da sintaxe da linguagem proposta na sintaxe de Python.

- Das linguagens Ruby e VHDL são usadas alguns syntatic sugars. Por exemplo, de Ruby foi utilizado a ideia de blocos enquanto que de VHDL foi utilizado a forma como parâmetros são declarados. Ainda com relação a VHDL, ressalta-se que apenas a sintaxe é usada como inspiração, não herdando nenhuma semântica da mesma.

- Ter liberdade para testar conceitos novos, uma vez que não existe a obrigação de se manter compatibilidade.

- A sintaxe da linguagem fica restrita apenas ao front-end do compilador. Dessa forma, através de uma representação intermediária tal como a usada pelo framework LLVM, é possível se aproveitar de toda a tecnologia desenvolvida até então para geração de código eficiente e que roda na parte back-end de um compilador.

Como mencionado anteriormente, a linguagem não traz nenhum conceito novo, apenas agrega conceitos já existentes de outras linguagens. Segue um resumo das funcionalidades esperadas para a linguagem:

- Suporte a construções clássicas tais como estruturas, enumerações, classes, laços de repetição, variáveis, funções e similares.
- Suporte a inferência de tipos, *closures*, *lambdas*, sobrecarga de funções, métodos e operadores.
- Suporte para as construções das linguagens ForeC, PRETC e Lingua Franca bem como para as usadas em CUDA.

A Figura 31 mostra um exemplo de programa na linguagem proposta. Pode-se perceber a forte inspiração que a linguagem Python tem na sintaxe da linguagem. Funções são iniciadas pela palavra reservada *def*. A função *sumOnArray* tem uma semântica própria. Ela é uma função que será executada no array através da marcação @kernel. No corpo dessa função, o acesso aos registradores do array é feito pelos nomes r0, r1, r2 e assim por diante até o r15. A diretiva step salva todos os passos executados até então em uma palavra de controle. Dessa forma é capaz de se programar o array indicando o que cada EP deve executar a cada ciclo de relógio. Labels são usadas também para prover alvos para operações de salto. No futuro espera-se que os kernels possam ser definidos como uma sintaxe de mais alto nível, gerando então o código de forma automatizada.

Esta seção finaliza o capítulo 4, o qual apresentou os conceitos e detalhes da arquitetura proposta. O próximo capítulo é destinado a apresentação dos experimentos realizados e resultados obtidos.

Figura 31 – Exemplo de Código

```
import std.io

@kernel
def sumOnArray : void
    r0 = r0 + r1
    r1 = r2 + r3
    step

L0:
    r1 = mem[r1 + r2 << 1 + 16]
    if r1 ==0 goto L1
    step

    r1 = r1 + 1
    goto L0
    step

L1:
    return

def sum : int
    @a : int
    @b : int

    return a + b

def times : int
    @a : int
    @b : int

    t = 0
    while a > 0:
        t += b

    return t

def main : int
    @args : String[]

    println('Hello, world')
    return 0
```

Fonte: autoria própria

5 Experimentos e Resultados

Este capítulo apresenta os experimentos realizados e os resultados obtidos. O capítulo inicia com uma descrição das ferramentas utilizadas para realizar os experimentos. Em seguida é apresentada a metodologia que foi utilizada e por último são mostrados os resultados obtidos.

5.1 Materiais

5.1.1 Ferramentas utilizadas

A arquitetura proposta foi descrita na linguagem Verilog (na versão 2001), sendo utilizado o simulador Icarus Verilog para realizar as simulações com precisão de ciclo de relógio. O simulador Icarus Verilog foi escolhido por ser uma ferramenta open-source, gratuito e apresentar bom suporte para a linguagem. O visualizador de formas de ondas utilizado foi o GtkWave, que também é open-source e gratuito.

Para os resultados de síntese, foi utilizado o Quartus versão 11.0. Esta versão possui suporte para o FPGA EP4CE115F29C8N da família Cyclone IV E, encontrado na placa de desenvolvimento DE2-115 da Terasic. Esse ambiente foi escolhido pois se tinha acesso à DE2-115 para realizar eventuais testes em hardware e pela familiaridade com a placa de desenvolvimento. Versões mais novas do Quartus já não fornecem mais suporte para a família Cyclone IV E, por isso a versão 11.0, que é antiga, foi utilizada.

Os experimentos foram realizados em um computador contendo um processador i7-8750H e 16GB de memória RAM. Essa configuração provavelmente não tem efeito na reprodução dos resultados aqui obtidos, influenciando apenas no tempo de execução das simulações resultados (a configuração citada possibilita simulações e sínteses mais rápidas). Os sistemas operacionais utilizados foram o Ubuntu 18.04 e Ubuntu 20.04.

Para auxiliar na compilação dos benchmarks, foi utilizado um *cross-compiler* MIPS, o gcc na versão 10 (gcc-10-mips-linux-gnu). O uso desse compilador é explicado na seção *Metodologia*. Por fim, pequenos programas auxiliares foram escritos no decorrer da tese para extração de paralelismo das instruções geradas pelo *cross-compiler* MIPS.

5.1.2 Benchmarks

Os benchmarks selecionados foram extraidos do MiBench (GUTHAUS et al., 2001) e foram selecionados por representar diferentes cenários e ao mesmo tempo possuir um código fácil de se trabalhar. Os benchmarks escolhidos são descritos brevemente:

- **BitCount**: verifica a capacidade de manipulação de bits da arquitetura ao contar bits utilizando diversas estratégias
- **Susan**: realiza um processamento de imagens, aplicando um filtro de suavização e realizando detecção de bordas e cantos
- **SHA-1**: realiza o processamento de geração de uma hash baseado no SHA-1
- **Patricia**: monta uma árvore Patricia para a busca de IPs
- **StringSearch**: realiza busca de uma string no conjunto de entradas

5.2 Metodologia

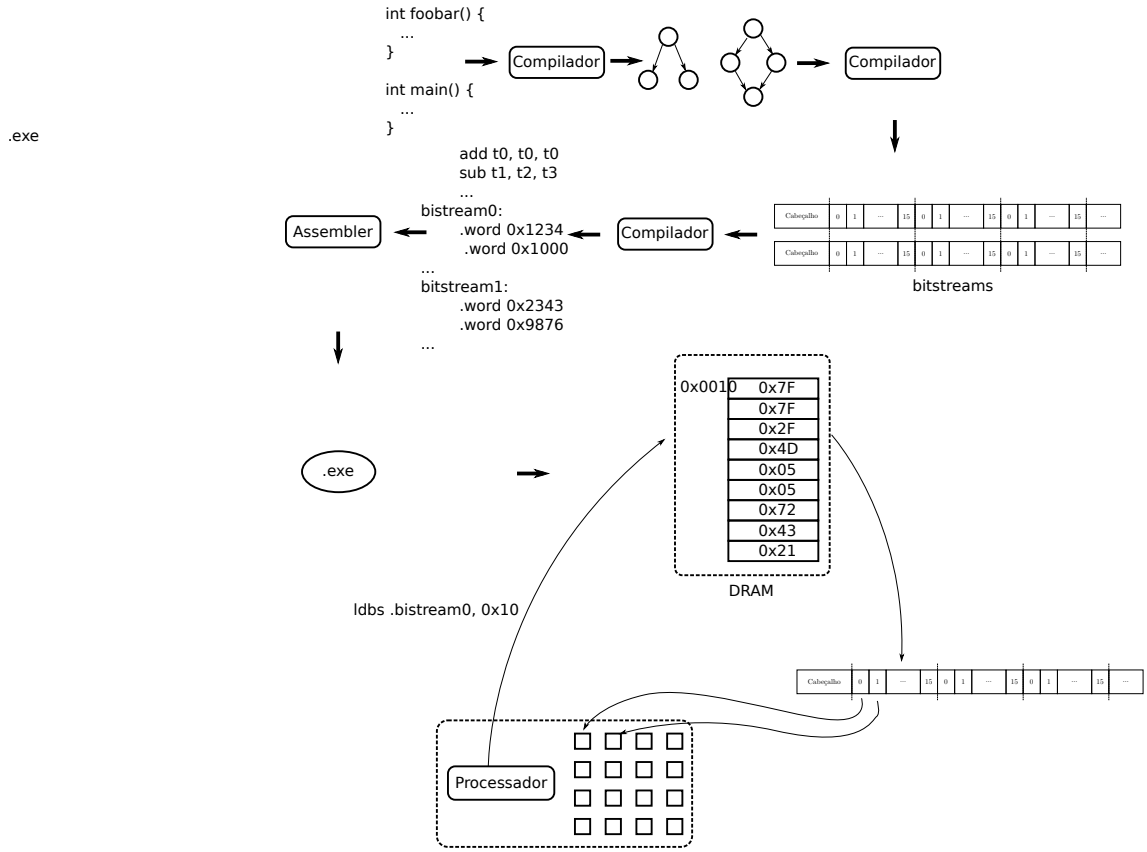
Como a arquitetura proposta ainda não possui o compilador pronto para uso e como o código dos benchmarks são muito complexos para serem implementados manualmente em assembly, foi utilizado o auxílio de um *cross-compiler gcc* para a arquitetura MIPS. O processo de codificação em assembly dos benchmarks está apresentado na Figura 32. Primeiro o código do benchmark é compilado usando o cross-compiler. Uma análise é feita em cima dos blocos básicos procurando identificar instruções que possam ser executadas em paralelo pelo CGRA. Essa análise é feita com o auxílio de pequeno programa desenvolvido que recebe como entrada um arquivo assembly e gera um grafo de dependências, a partir do qual se extrai as instruções que podem ser executadas em paralelo. Com relação ao RIDECORE e ao ρ-VEX, foi utilizado o cross-compiler de cada um respectivamente.

Em alguns casos melhorias manuais eram realizadas a fim de explorar melhor a arquitetura proposta. Por exemplo, no caso do benchmark BitCount onde era possível fazer um loop unrolling agressivo e paralelizar entre várias threads, essas otimizações foram feitas. Em outros benchmarks como o Susan, por exemplo, onde a lógica é mais complicada, foi utilizado apenas o ILP extraído a partir de blocos básicos, ou seja, não foi feito uma exploração de ILP agressiva. Ainda com relação ao Susan, como é um filtro em cima de uma imagem, o paralelismo a nível de thread foi obtido particionando a imagem com cada thread responsável por processar uma parte da imagem.

Como todas as arquiteturas estão rodando em *bare metal*, ou seja, sem o intermédio de um sistema operacional, chamadas de sistema e funções como *printf*, por exemplo, foram removidas dos benchmarks. Dessa forma, o código executado nos testes não correspondem 100% aos códigos originais dos benchmarks, porém todas as arquiteturas executam o mesmo código modificado.

Modificações nos dados de entrada também foram realizados. As modificações consistiram em basicamente diminuir a quantidade de entrada dos benchmarks originais, limitando a um máximo de 1MB de dados de entrada. Esse tamanho foi escolhido tendo

Figura 32 – Processo de compilação



Fonte: autoria própria

como base arquiteturas atuais de processador que já contém caches com esse tamanho, permitindo que todos os dados de entrada se encontrem em memória. Isso foi necessário porque como as simulações são executadas em nível RTL e as entradas originais são grandes, o tempo de simulação era bastante alto. Por exemplo, no Susan a imagem original foi redimensionada para um tamanho menor enquanto que no BitCount e Patricia foram escolhidas entradas aleatoriamente formando um novo conjunto de entrada menor baseado no original.

Os resultados de desempenho são mostrados sempre com relação ao processador Hivek-RT executando apenas uma thread. É importante mencionar o fato de que no Hivek-RT, como cada thread é intercalada no pipeline em *round-robin* e como são quatro threads, então cada thread recebe 1/4 do tempo total de execução. É por isso que em todos os benchmarks apresentados o Hivek-RT executando apenas uma thread tem o pior desempenho, já que ele está rodando com uma frequência 4 vezes mais lento que os demais. Isso significa que, por exemplo, se uma thread executa um programa em 1000 ciclos de thread, esses 1000 ciclos de thread representam 4000 ciclos de relógio.

No que diz respeito aos resultados de desempenho, cada benchmark foi executado

em cada uma das arquiteturas em simulações com precisão de ciclo de relógio no simulador *Icarus Verilog*. A quantidade total de ciclos de cada arquitetura foi anotada em uma planilha para posterior comparação entre as mesmas. Para evitar variações causadas por causa de diferentes hierarquias de memórias, os processadores RIDECORE e ρ-VEX foram equipados de uma cache que sempre apresenta *hit* de forma a se equiparar as *scratchpads*. As memórias foram *caches* e *scratchpads* foram feitas grandes o suficiente para que todo o *benchmark* coubesse nelas. Isso não chega a ser tão irreal porque se trata de benchmarks de poucos kilobytes, tamanho este que é totalmente viável dado as tecnologias atuais.

5.3 Resultados de Síntese

A arquitetura proposta foi comparada com outras arquiteturas para verificar o uso de área. Essas arquiteturas foram escolhidas porque possuem código fonte disponível em domínio público além de estarem descritos também em nível RTL, permitindo uma comparação justa com a arquitetura proposta que também foi descrita em nível RTL. Como não se tinha acesso a ferramentas de síntese para standard cells, realizou-se a síntese para FPGA da Intel. As características desse FPGA estão descritas na Tabela 2.

Tabela 2 – Características do FPGA alvo

Modelo	Elementos Lógicos	Bits de Memória	Pinos I/O	Multiplicadores	PLL
Cyclone IV EP4CE115F29C7	114.480	3.981.312	529	532	4

Fonte: autoria própria

Foram sintetizados o processador Hivek-RT, o processador ρ-VEX, o processador RIDECORE e a arquitetura proposta. O processador ρ-VEX é um processador VLIW capaz de executar até 6 instruções em paralelo sendo elas quatro lógico-aritméticas, duas multiplicações e uma de acesso à memória. Já o processador RIDECORE é um processador *Out-of-Order* que implementa a ISA RISC-V.

Tabela 3 – Resultados de Síntese

Arquitetura	Área (LEs)	Frequência	Frequência (alvo 50MHz)
Hivek-RT	5.399	62 MHz	62 MHz
ρ-VEX	16.163	66 MHz	66 MHz
RIDECORE	62.831	35 MHz	35 MHz
CGRA	29.087	51 MHz	61 MHz
Hivek-RT + CGRA	35.270	49 MHz	57 MHz

Fonte: autoria própria

A Tabela 3 apresenta os resultados de síntese de cada uma das arquiteturas. O processador Hivek-RT utilizado apresenta a menor área de todas, utilizando 5.399 Elementos Lógicos (*Logical Elements - LE*). Esse resultado é esperado porque de todas as arquiteturas, é a que menos explora ILP, uma vez que só é capaz de executar até duas instruções em paralelo. Além disso, por ser um processador PRET, não possui mecanismos tais como *bypass*, predição de salto entre outros.

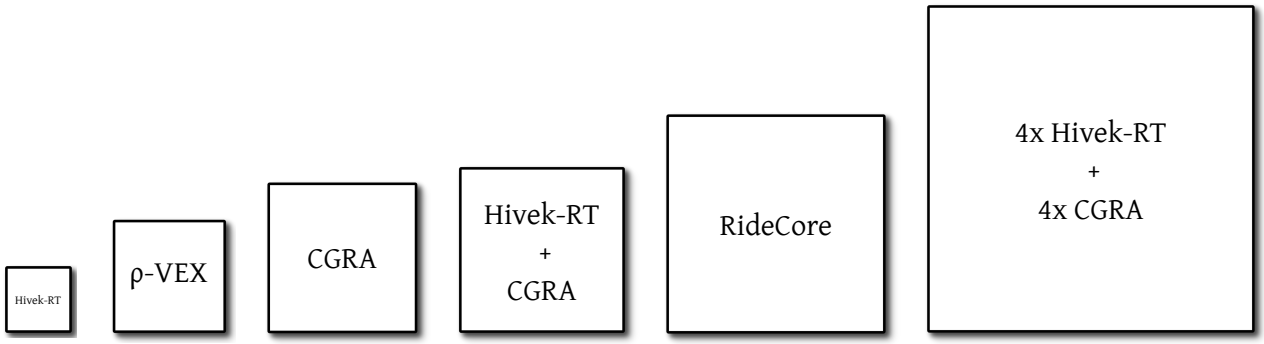
O processador ρ-VEX apresenta um consumo de 16.163 LEs, sua área é maior que o Hivek-RT uma vez que consegue executar mais instruções em paralelo. Já o processador RIDECORE é o processador que mais consome elementos lógicos, utilizando um total de 62.831 LEs.

O CGRA proposto consome uma área de 29.087 LEs. Combinado com o processador Hivek-RT essa área sobe para 35.270 LEs, sendo portanto 2.2x maior que o ρ-VEX e 43% menor que o processador RIDECORE. Em outras palavras, o núcleo da arquitetura proposta ocupa, aproximadamente, o dobro da área do processador VLIW ρ-VEX e metade da área do processador *Out-of-Order* RIDECORE.

A ferramenta de síntese não foi capaz de sintetizar o multicore por ultrapassar a quantidade de elementos lógicos do FPGA utilizado como alvo. Entretanto, é possível fazer uma estimativa grosseira do uso de área do multicore ao adicionar a área ocupada por cada núcleo. Como cada núcleo ocupa 35.270 LEs, é possível estimar que os quatro núcleos juntos ocupem, no mínimo, uma área de 141.080 LEs levando a uma área 2.25x maior que o processador RIDECORE, 8.72x maior que o processador ρ-VEX e 26.2x maior que o processador Hivek-RT. É importante ressaltar que são apenas estimativas, uma vez que ao juntar os quatro núcleos deve-se levar em consideração a área utilizada para a comunicação dos mesmos. De qualquer forma, essa estimativa grosseira permite ter uma idea de como as arquiteturas se relacionam. A Figura 33 mostra uma ilustração representativa das áreas de cada uma das arquiteturas.

Em termos de frequência de operação, a Tabela 3 apresenta os resultados obtidos para cada uma das arquiteturas. Primeiro foi feito uma síntese sem nenhuma restrição de frequência a fim de verificar qual seria a frequência máxima de operação de cada uma das arquiteturas. Em seguida, foi inserida uma restrição de frequência máxima de operação com alvo de 50 MHz. Quando uma restrição é imposta, a ferramenta de síntese se esforça mais para tentar atingir a frequência requisitada. Pela Tabela 3, é possível observar que o processador ρ-VEX obteve uma frequência de 66 MHz, a maior frequência de operação de todas as arquiteturas, apesar de que por uma margem pequena. O processador Hivek-RT apresentou uma frequência de operação de 62 MHz enquanto que o processador RIDECORE obteve apenas 35 MHz, sendo a menor frequência de todas. Já o CGRA em síntese sem restrição de frequência obteve 51 MHz e com a restrição de frequência a ferramenta conseguiu subir a frequência de operação para 61 MHz. O núcleo formado pelo

Figura 33 – Área das arquiteturas



Fonte: autoria própria

processador Hivek-RT e o CGRA obteve frequência de operação máxima de 57 MHz.

A frequência maior das arquiteturas VLIW (Hivek-RT e ρ-VEX) é esperada uma vez que sua microarquitetura é mais simples que as demais. Como o processador RIDECORE usa um pipeline de apenas 6 estágios e por ter o mecanismo de execução fora de ordem, a ferramenta de síntese provavelmente teve problemas para rotear e obter melhores frequências. Isso é diferente dos processadores VLIW que possuem quantidade de estágios similar porém com uma microarquitetura bem mais simples. Um pipeline maior e com estágios menores provavelmente levaria o RIDECORE a uma frequência de operação maior. Com relação ao CGRA, o mesmo conseguiu atingir a frequência alvo de operação de 50 MHz, tendo atingido uma frequência máxima de 61 MHz e sua junção com o processador Hivek-RT a fim de formar o núcleo ainda continuou acima da frequência alvo de 50 MHz, tendo atingido 57 MHz. Por fim, como não foi possível sintetizar o multicore por completo, não foi possível obter a frequência de operação. Porém como cada núcleo roda independente dos demais, espera-se que seja possível obter frequências similares.

5.4 Análise de Desempenho

As máquinas PRETs foram concebidas para prover características de tempo real e bom desempenho para sistemas embarcados de tempo real e ciber-físicos. Dessa forma, é importante que a arquitetura proposta apresente bom desempenho quando comparado a arquiteturas tradicionais tais como é o caso do ρ-VEX, que é um VLIW e o RIDECORE, que é um *Out-of-Order*. Além disso, máquinas PRET devem ser capazes de executar uma gama variada de aplicações, visto que tem por objetivo substituir arquiteturas federadas. Assim, para medir o desempenho da arquitetura proposta considerando um ambiente heterogêneo, foram utilizado os benchmarks MiBench (GUTHAUS et al., 2001) e

ParMibench (IQBAL; LIANG; GRAHN, 2010). O MiBench é um benchmark voltado para sistemas embarcados single-core, enquanto o ParMiBench é a versão paralela do MiBench para sistemas multicore. Além desses benchmarks, também é realizado um experimento utilizando multiplicação de matrizes.

5.4.1 Análise de Desemenho do Benchmark BitCount

O benchmark Bitcount mede a capacidade de manipulação de bits da arquitetura contando o número de bits usando diferentes estratégias de contagem. Ele possui oito sub-algoritmos cujas saídas são o número de bits nos dados de entrada cujo valor é 1. Cada um dos oito sub-algoritmos implementam diferentes estratégias de contagem de bits.

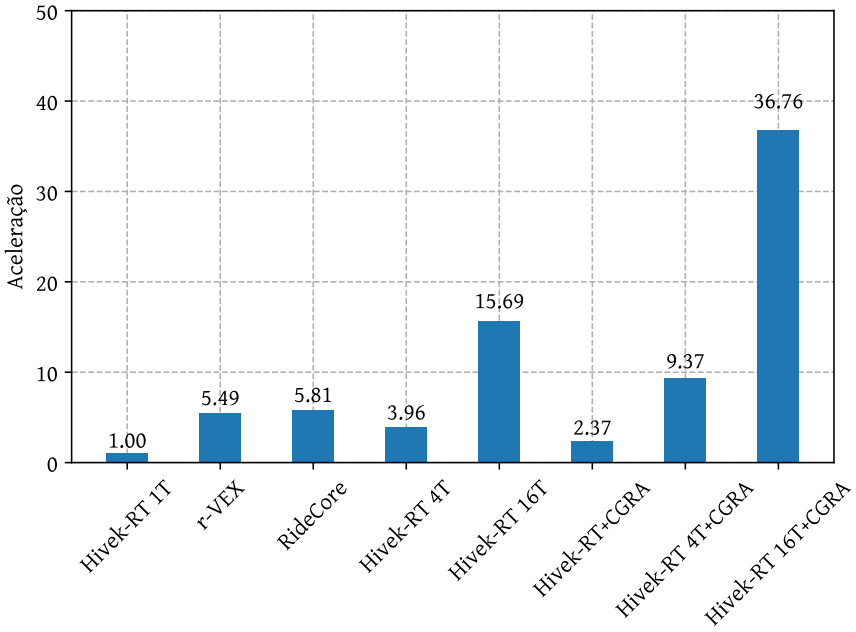
Na versão paralela do benchmark, cada um dos oito sub-algoritmos executam em uma thread diferente. O grau de comunicação entre as threads é extremamente baixo uma vez que a única comunicação necessária é que cada uma informe a quantidade de bits contados, o que pode ser realizado uma única vez ao fim da execução. Dessa forma esse benchmark apresenta um grande potencial de TLP. Já com relação ao ILP, a princípio esse benchmark apresenta blocos básicos pequenos. Porém, como as iterações são independentes entre si, é possível usar um loop unrolling agressivo. Em um dos sub-algoritmos, por exemplo, foi possível processar até quatro entradas em paralelo através da exploração de ILP sem considerar a exploração de TLP. Além do loop unrolling, alguns dos sub-algoritmos utilizam de *lookup tables* para realizar a contagem. Como o CGRA é capaz de executar até quatro operações de acesso à memória em paralelo, isso representa uma grande vantagem sobre o ρ-VEX e o RIDECORE que só possuem uma unidade de acesso à memória.

O gráfico apresentado na Figura 34 apresenta o desempenho de cada uma das arquiteturas ao executar todos os sub-algoritmos do BitCount.

O processador Hivek-RT executando o código em apenas uma thread é o mais lento de todos. Como explicado, isso se deve ao fato de que as threads no Hivek-RT são intercaladas a cada ciclo de relógio, o que leva cada thread a executar a 1/4 do tempo disponível. É preciso que cada thread esteja realizando trabalho útil para que o desempenho do Hivek-RT seja equivalente aos dos processadores ρ-VEX e RIDECORE. Isso fica evidenciado quando o Hivek-RT executando só com uma thread equivale a 18% do poder de processamento do processador ρ-VEX e 17% do processador RIDECORE como pode ser observado na Tabela 4. Já com as quatro threads sendo utilizadas, esses valores sobem para 72% e 68% respectivamente. Esses valores são esperados uma vez que o Hivek-RT só é capaz de executar até duas instruções em paralelo enquanto que o ρ-VEX pode chegar a um pico de quatro instruções em paralelo e o RIDECORE a um pico de 6 instruções em paralelo.

A adição do CGRA à uma thread aumenta o desempenho em 2.37x quando

Figura 34 – Aceleração das arquiteturas para o BitCount



Fonte: autoria própria

comparado com essa mesma thread sem o auxílio do CGRA. Com relação ao ρ-VEX e o RIDECORE, a thread é capaz de obter 43% do desempenho do ρ-VEX e 41% do desempenho do RIDECORE. Já com o Hivek-RT executando as quatro threads com o auxílio do CGRA, ele consegue ser 1.71 vezes mais rápido que o ρ-VEX e 1.61 vezes mais rápido que o RIDECORE. Considerando que a área do Hivek-RT+CGRA é um pouco maior que a área do ρ-VEX e menor que a área do RIDECORE, essa aceleração representa um bom resultado.

Quando levado em consideração o processador multicore, o desempenho das 16 threads executando em paralelo consegue ser 2.85 vezes mais rápido que ρ-VEX e 2.70 vezes mais rápido que o RIDECORE. Com o auxílio do CGRA, esses valores sobem para 6.69x e 6.33x respectivamente. Se comparado com a execução de uma única thread, o processador multicore com CGRA consegue ter uma aceleração de 36.76x quando comparado com a execução de uma única thread.

Quando se compara o multicore executando as 16 threads sem o auxílio do CGRA e com o auxílio do CGRA, pela Tabela 4 é possível constatar um aumento de velocidade de 2.3x, mostrando assim que para esse benchmark, a inclusão do CGRA traz ganhos expressivos.

5.4.2 Análise de Desempenho do Benchmark Susan

O Susan é um programa para tratamento de imagens. Ele consiste em três sub-programas: um para suavizar a imagem, um para detecção de cantos e um para a detecção de bordas. Nesse trabalho foi executado o algoritmo de detecção de bordas visto que era o que mais possuia capacidade de exploração de ILP.

Apesar do Susan apresentar uma grande quantidade de instruções para exploração de ILP, muitas das instruções possuem dependências de dados. O código apresentado na Figura 35 é um trecho da aplicação. O leitor poderá perceber que a operação $c = *(cp - *p++)$ é repetida várias vezes e foi o cerne de aceleração implementado. A paralelização desse trecho permitiu a execução de até quatro operações de *load* em paralelo em conjunto com outras instruções lógico-aritméticas.

Figura 35 – Trecho do código do Susan Edges

```
if (n>600)
{
  p=in + (i-3)*x_size + j - 1;
  x=0;y=0;

  c=*(cp-*p++);x-=c;y-=3*c;
  c=*(cp-*p++);y-=3*c;
  c=*(cp-*p);x+=c;y-=3*c;
  p+=x_size-3;

  c=*(cp-*p++);x-=2*c;y-=2*c;
  c=*(cp-*p++);x-=c;y-=2*c;
  c=*(cp-*p++);y-=2*c;
  c=*(cp-*p++);x+=c;y-=2*c;
  c=*(cp-*p);x+=2*c;y-=2*c;
  p+=x_size-5;
```

Fonte: autoria própria

A exploração de TLP foi obtida particionando a imagem a ser processada. Cada thread ficou responsável por processar uma parte da imagem.

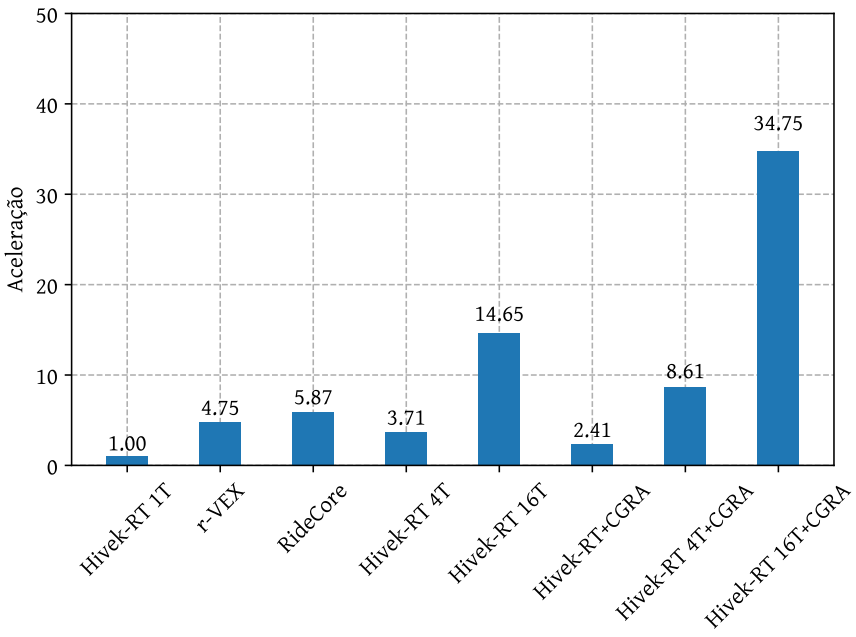
A Figura 36 apresenta os resultados de desempenho obtidos para esse benchmark. Mais uma vez o Hivek-RT executando apenas uma thread é o que apresenta o pior desempenho. O processador RIDECORE apresenta um desempenho melhor que o processador ρ-VEX, provavelmente por causa do seu mecanismo de especulação que consegue explorar a longa cadeia de dependências contidas no Susan Edges.

O Hivek-RT executando quatro threads sem o auxílio do CGRA consegue um desempenho equivalente a 78% do ρ-VEX e 63% do RIDECORE o que é compreensível

visto que o Hivek-RT tem uma capacidade de exploração de ILP menor. Trabalhando em conjunto com o CGRA, o Hivek-RT 4T consegue ser 1.81 vezes mais rápido que o ρ-VEX e 1.47 vezes mais rápido que o RIDECORE.

Explorando o máximo de TLP possível, a execução nas 16 threads do multicore leva a um aumento de 3.08 vezes se comparado ao ρ-VEX e 2.50 vezes se comparado ao RIDECORE. Se combinado como o CGRA, a aceleração sobe para 7.31 vezes se comparado ao ρ-VEX e 5.92 vezes se comparado ao RIDECORE. Por fim, se comparar o Hivek-RT 16T com o Hivek-RT 16T+CGRA, obtém-se uma aceleração de 2.37x.

Figura 36 – Aceleração das arquiteturas para Susan



Fonte: autoria própria

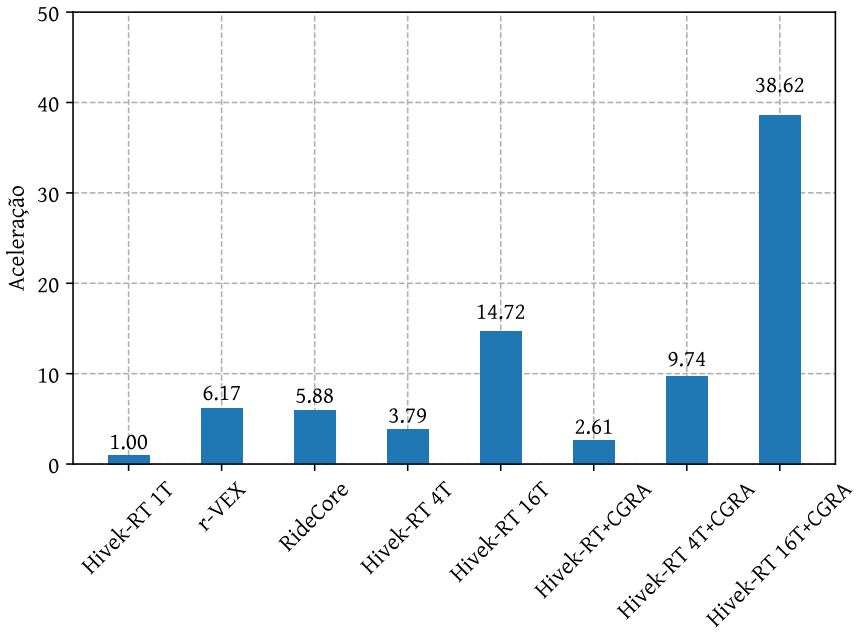
5.4.3 Análise de Desempenho do Benchmark SHA-1

O benchmark SHA-1, como o nome sugere, implementa o algoritmo SHA-1 para a geração de hashes. Esse é outro benchmark no qual é possível explorar de forma agressiva tanto ILP como TLP. Na exploração de ILP, esse benchmark realiza diversas operações *bitwise* e que podem ser paralelizadas no CGRA. Já a exploração de TLP é obtida mais uma vez dividindo as entradas entre as várias threads. Resultados de desempenho podem ser observados na Figura 37.

Esse benchmark apresenta ganhos parecidos com os benchmarks Susan e BitCount, no sentido de que acelerações maiores são obtidas quando combinado a exploração de ILP com o CGRA e a exploração de TLP com as várias threads. Por exemplo, o Hivek-RT

4T+CGRA consegue ser 1.58 vezes mais rápido que o processador ρ-VEX e 1.66 vezes mais rápido que o processador RIDECORE. Já sem o uso do CGRA esses resultados caem para 0.61 e 0.65, respectivamente. A exploração de apenas TLP usando as 16 threads leva a um aumento de desempenho de 2.38 vezes quando comparado ao ρ-VEX e 2.51 vezes quando comparado com o RIDECORE. Combinado com o CGRA, esse desempenho aumenta para 6.26x e 6.57x respectivamente, um ganho bastante expressivo.

Figura 37 – Aceleração das arquiteturas para o SHA-1



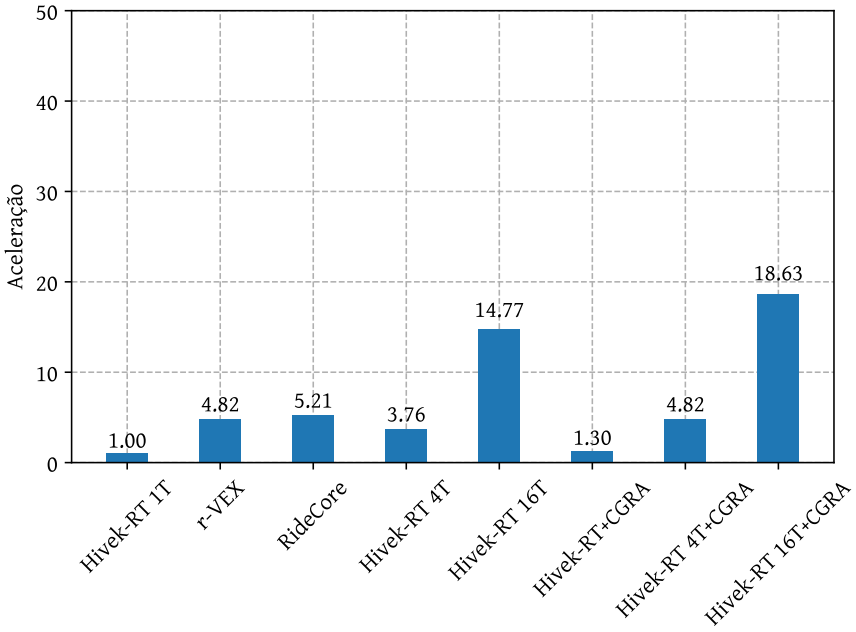
Fonte: autoria própria

5.4.4 Análise de Desempenho do Benchmark Patricia

O benchmark Patricia consiste na montagem de uma árvore de busca para endereços IP. Para esse benchmark, a arquitetura proposta apresenta desempenho pouco satisfatório. Esse benchmark tem uma quantidade elevada de saltos e blocos básicos pequenos quando comparado com os três últimos benchmarks discutidos até então. Por causa disso, a exploração de ILP é prejudicada visto que o overhead de se usar o CGRA para processar blocos pequenos não compensa.

O desempenho de cada arquitetura para esse benchmark pode ser visto na Figura 38. Mesmo com o auxílio do CGRA, o Hivek-RT 4T+CGRA não consegue ser melhor que os processadores ρ-VEX e RIDECORE. A aceleração desse benchmark vem, praticamente, apenas da exploração de TLP, necessitando o uso completo da arquitetura para ser melhor que os processadores ρ-VEX e RIDECORE.

Figura 38 – Aceleração das arquiteturas para o benchmark Patricia



Fonte: autoria própria

5.4.5 Análise de Desempenho do Benchmark StringSearch

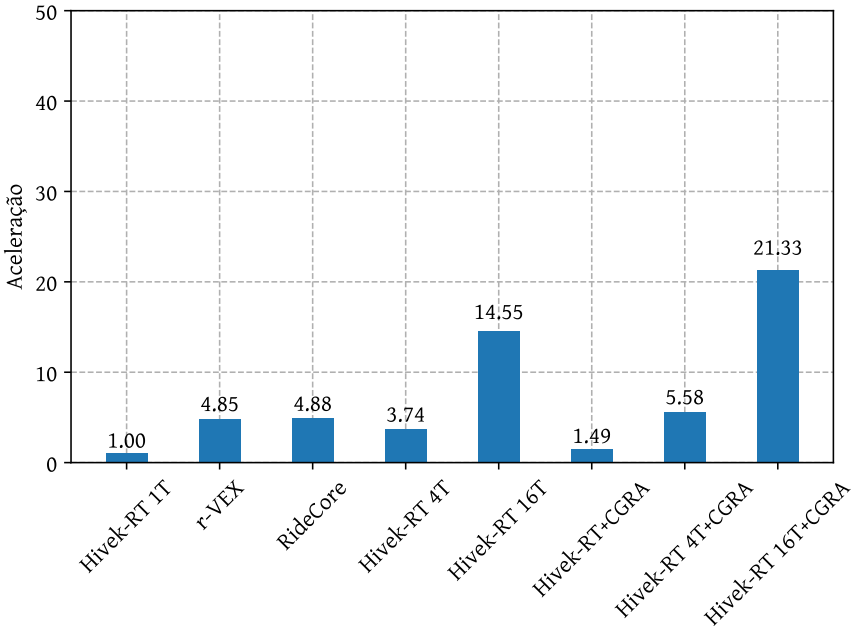
O benchmark StringSearch realiza a busca de padrões em strings de entrada. Ele é parecido com o benchmark Patricia no sentido de ser uma aplicação *control flow*. Apesar dos seus blocos básicos serem pequenos, eles ainda fornecem um pouco mais de ILP que o benchmark Patricia. Os resultados para esse benchmark podem ser conferidos na Figura 39 e na Tabela 4.

5.4.6 Análise de Desempenho do Benchmark Multiplicação de Matrizes

A multiplicação de matrizes foi escolhido como um dos benchmarks a ser realizado porque é uma aplicação que provê grandes oportunidades de exploração tanto de ILP como de TLP. Para esse teste foi utilizado o programa mostrado na Figura 40. Ele implementa uma multiplicação de matrizes de dimensões $m \times n$ por $n \times p$.

Para esse benchmark em especial, será dada uma explicação breve de como o algoritmo foi mapeado para o CGRA a fim de ilustrar o poder computacional do CGRA proposto. O código em C mostrado na Figura 40 é usado como base para montar o circuito mostrado na Figura 41, que consiste em um pipeline de multiplicação. Na parte inferior da imagem estão as unidades responsáveis pelo cálculo do endereço dos elementos a serem multiplicados. Esses endereços são enviados para as memórias scratchpads que por sua vez retornam os valores a serem multiplicados e os repassam adiante através de registradores.

Figura 39 – Aceleração das arquiteturas para o StringSearch



Fonte: autoria própria

A multiplicação é realizada e o resultado é adicionado a um acumulador. Uma vez que o pipeline esteja cheio, consegue-se obter uma nova multiplicação a cada ciclo de relógio.

Partindo desse circuito, foi realizado um mapeamento para o CGRA conforme mostrado na Figura 42. Apesar do CGRA em si não possuir um pipeline, a forma como se dá a conexão entre as unidades de processamento acaba gerando uma espécie de pipeline virtual. Essa configuração permitiu uma exploração agressiva de ILP, com todas as unidades do CGRA trabalhando em paralelo, porém com algumas funcionando como armazenamento e cálculo e outras apenas como armazenamento.

O gráfico da Figura 44 mostra a quantidade de ciclos de cada uma das arquiteturas para a multiplicação de matrizes de dimensões variáveis. Como esperado, o Hivek-RT executando em uma única thread apresenta o pior desempenho de todas as arquiteturas. Entretanto, um resultado muito interessante é obtido quando se combina um Hivek-RT 1T com o CGRA: ele consegue ser mais rápido que o processador ρ-VEX e que o processador RIDECORE. De todos os benchmarks, esse foi o único onde uma thread do Hivek-RT foi capaz de performar melhor que processadores tradicionais. Isso foi possível por causa da agressiva exploração de ILP obtida ao se executar em pipeline.

Um Hivek-RT 4T+CGRA é 4.08 vezes mais rápido que o ρ-VEX e 4.12 vezes mais rápido que o RIDECORE. Esse é um resultado bastante satisfatório considerando as áreas similares do Hivek-RT 4T+CGRA e ρ-VEX e a área menor que a do RIDECORE.

Figura 40 – Função de multiplicação de matrizes de dimensões variadas

```
// c = a * b, a com dimensões m x n e b com dimensões n x p
void mult(int* c, int* a, int* b, int m, int n, int p) {
    int acc;

    for (int i = m - 1; i >= 0; --i) {
        for (int j = p - 1; j >= 0; --j) {
            acc = 0;

            for (int k = 0; k < n; ++k) {
                acc = a[i * n + k] * b[k * p + j];
            }

            c[i][j] = acc;
        }
    }
}
```

Fonte: autoria própria

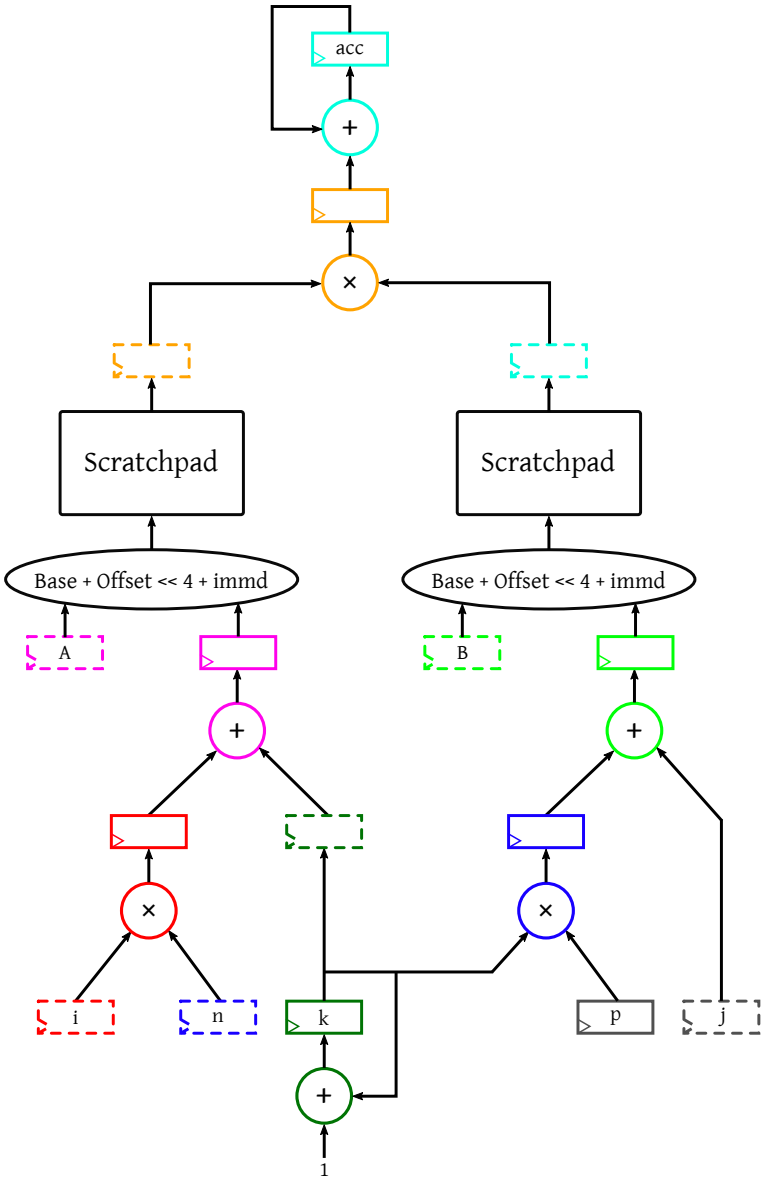
Entretanto, diferente do esperado, a exploração de TLP utilizando os quatro núcleos não trouxe ganhos significativos. Isso se deve por causa do overhead da transferência das matrizes a serem multiplicadas como ilustrado na Figura 43. É preciso realizar uma cópia das matrizes a serem multiplicadas para cada um dos núcleos e depois ler de volta os resultados. Esse overhead na transferência de dados acaba por degradar os resultados obtidos com o uso do CGRA. Dessa forma, ao invés de um ganho na ordem de 16x, obtém-se um ganho de apenas 7.23x quando comparado com o ρ-VEX e 7.30x quando comparado com o RIDECORE. O ganho na ordem de 16x seria possível num cenário onde a multiplicação de matrizes fossem independentes. Por exemplo, considere um cenário onde cada núcleo é responsável por processar uma imagem oriunda de uma câmera diferente. Como não seria necessário a cooperação entre os núcleos, mas sim cada núcleo ficaria responsável por sua própria matriz, nesse cenário seria possível o ganho de 16x.

A Tabela 4 apresenta um resumo das acelerações obtidas por cada uma das arquiteturas. A coluna *Aceleração Hivek-RT 1T* contém a aceleração das demais arquiteturas em relação ao Hivek-RT executando uma única thread. Já as colunas *Aceleração ρ-VEX* e *Aceleração RIDECORE* representam a aceleração das demais arquiteturas em relação a essas duas arquiteturas.

5.4.7 Resumo das Análises de Desempenho

Os experimentos discutidos anteriormente mostram que a adição do CGRA é fundamental para a obter melhor desempenho, mesmo em ambientes multiprocessados.

Figura 41 – Circuito para multiplicação de matrizes

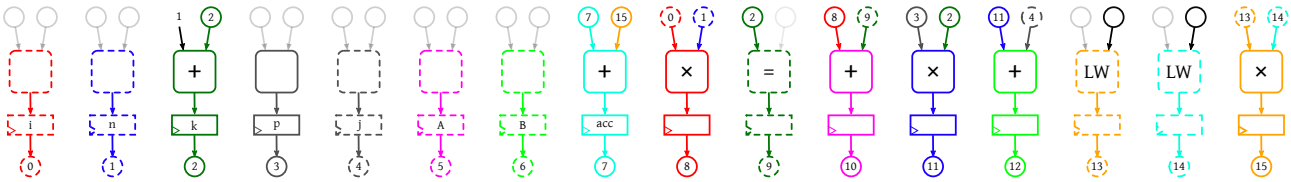


Fonte: autoria própria

Esses resultados corroboram com a discussão realizada em (RUTZIG; BECK; CARRO, 2013a) que defende que o melhor desempenho é conseguido por arquiteturas capazes de explorar simultaneamente TLP e ILP e que serviu de fundamento para o presente trabalho.

Apesar do CGRA ter acelerado em todas os benchmarks, é preciso mencionar que essa aceleração veio a um custo elevado na implementação desse trabalho. Como não havia um suporte maduro de ferramentas de desenvolvimento tais como compiladores e similares, houve bastante esforço na implementação dos benchmarks. Para cada benchmark foi preciso realizar uma análise dos graus de paralelismo que se poderia extrair, demandando um alto tempo de desenvolvimento. No escopo dessa pesquisa esse tempo é aceitável. Já

Figura 42 – Mapeamento da multiplicação de matrizes para o CGRA



Fonte: autoria própria

numa situação de desenvolvimento de produto comercial dificilmente esse tempo seria aceitável.

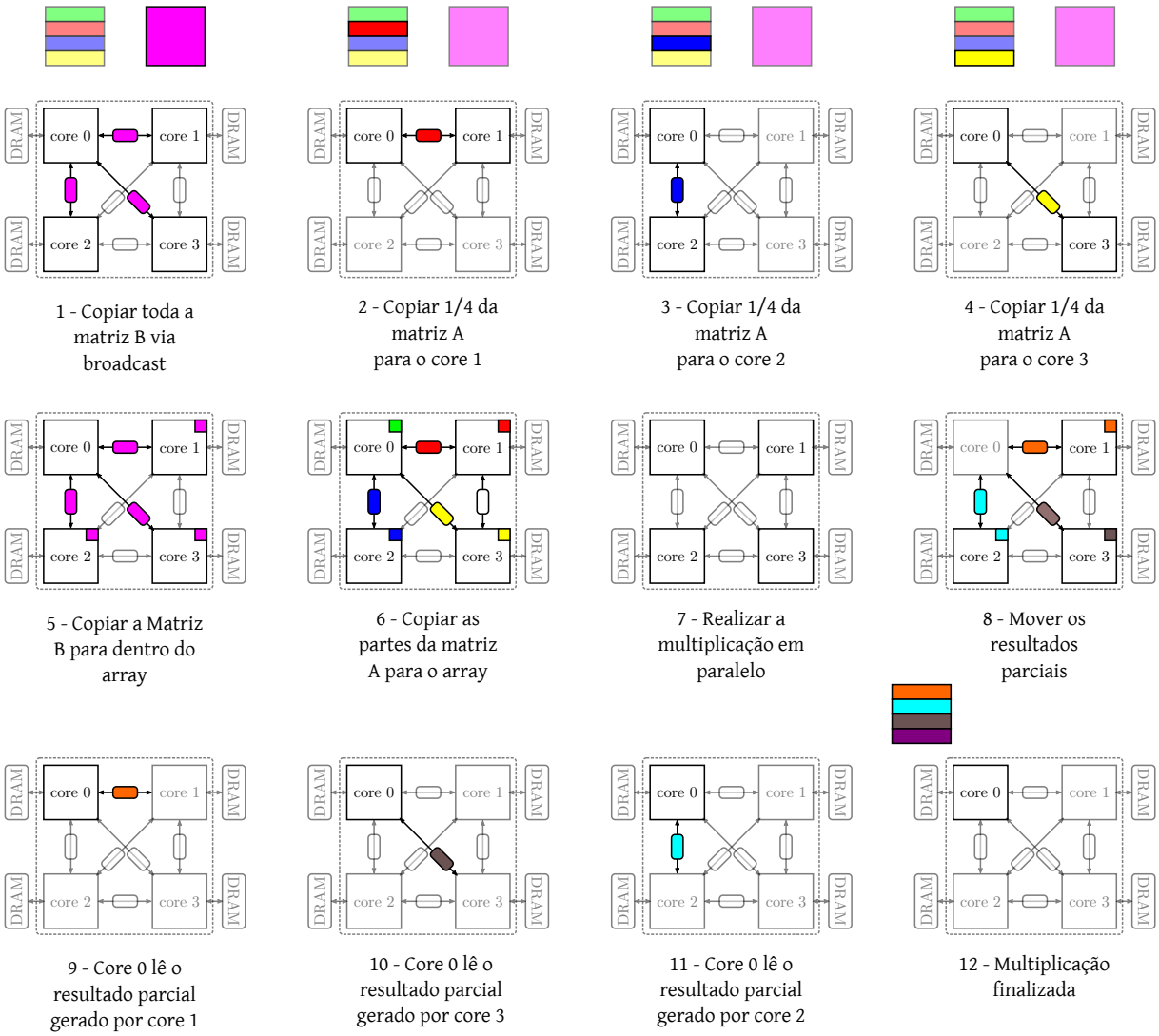
Por outro lado, as acelerações obtidas tem um ponto muito positivo e justificado quando se leva em consideração que são acelerações obtidas em uma arquitetura amigável para sistemas de tempo real. Por esse ponto de vista, pode-se dizer que a arquitetura teve seu objetivo alcançado: uma arquitetura de alto desempenho com previsibilidade e repetibilidade para sistemas de tempo real.

Como boa parte dos CGRAs, os melhores resultados vieram de aplicações com bastante computação. Benchmarks com bastante control flow como é o caso do Patricia sofrem bastante na arquitetura, uma vez que o CGRA não consegue ser utilizado de forma eficiente. Nesses casos arquiteturas Superescalares como é o caso do processador RIDECORE levam a vantagem já que são capazes de especularem de forma agressiva sem nenhum esforço por parte do programador. No caso da arquitetura proposta, essa é uma desvantagem pois a arquitetura proposta tem por base que o programador deve ser o responsável por extrair paralelismo. Sem o auxílio de ferramentas de desenvolvimento, como foi o caso dessa pesquisa, a arquitetura proposta é bastante penalizada. Esse é um ponto negativo que não pode ser negligenciado.

5.5 Análise das Características Temporais

As arquiteturas PRET surgiram para tentar prover alto desempenho combinado com um comportamento previsível e repetitivo a fim de facilitar o desenvolvimento de aplicações de tempo real. O objetivo desta seção é comparar a arquitetura desenvolvida com o processador ρ-VEX e o RIDECORE, que neste trabalho representam arquiteturas tradicionais de alto desempenho. O ρ-VEX e o RIDECORE não foram projetados tendo foco em sistemas de real, por isso é esperado que eles não tenham as mesmas características de uma arquitetura PRET projetada para esse fim. De toda forma, o objetivo aqui é apenas mostrar como as arquiteturas diferem em suas características temporais e apresentar motivos que justifiquem o uso das arquiteturas PRET.

Figura 43 – Etapas da Multiplicação de Matrizes usando todos os núcleos

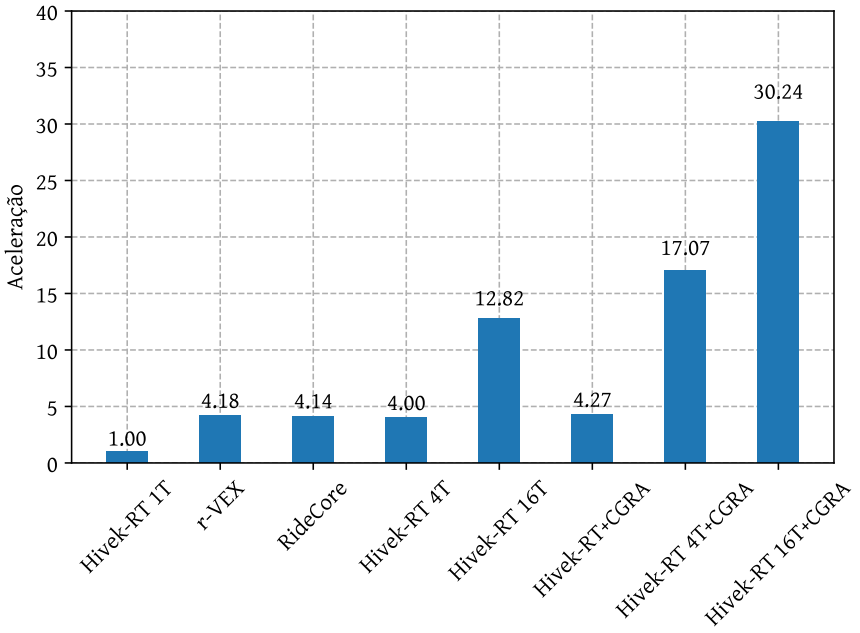


Fonte: autoria própria

Para os experimentos a seguir, o ρ-VEX e o RIDECORE foram modificados. Foi incluído um pequeno temporizador que dispara um sinal a cada 1 milissegundos a fim de simular uma troca de contexto entre threads. Dessa forma, a cada 1 milissegundos o ρ-VEX e o RIDECORE salva o contexto de uma thread e carrega a de outra. Como ambas arquiteturas testadas estão rodando em bare metal, a troca de contexto foi implementada por uma pequena rotina em assembly. Essa rotina escalona as threads em um simples esquema de *round-robin*. Além disso, é utilizado uma cache de instrução e de dados ambas de 32KB para o ρ-VEX e RIDECORE enquanto que para a arquitetura proposta é utilizado a hierarquia de memória discutida anteriormente e que consiste de scratchpads.

Como benchmarks, foram utilizados o *Susan* e o *BitCount*. O Susan foi escolhido porque possui bastante operações de acesso à memória o que vai levar, provavelmente, à

Figura 44 – Aceleração das arquiteturas para a multiplicação de matrizes de dimensões variáveis



Fonte: autoria própria

cache misses nos processadores ρ-VEX e RIDECORE. Já o BitCount foi escolhido por ter poucos acessos a memória e possuir uma computação mais intensiva. No caso da arquitetura proposta, os dois benchmarks foram paralelizados em threads a fim de utilizar a arquitetura como um todo.

Para o experimento, os benchmarks processam sempre as mesmas entradas em um loop infinito. A razão pela qual se processa sempre as mesmas entradas é para retirar a interferência temporal causada por diferentes entradas. Por exemplo, um dos algoritmos do BitCount utiliza um laço para a contagem de bits cuja quantidade de iterações é proporcial a quantidade de bits sendo contada. Dessa forma, usando sempre a mesma entrada, diferenças de tempo causadas por entradas diferentes são eliminadas e assim é possível verificar o grau de previsilibilidade e repetibilidade de cada arquitetura.

Cada uma das arquiteturas sofreu uma modificação com a inserção de um pequeno hardware para emitir um sinal em nível alto ao fim do processamento de uma entrada. Em outras palavras, a cada fim de uma iteração, o software executa a habilitação desse sinal com o auxílio do hardware, permitindo assim a verificação do momento exato que o processamento da iteração termina.

As arquiteturas foram simuladas em nível RTL e as formas de ondas geradas foram processadas por um pequeno programa em C++ que lê o arquivo de formas de onda no

Tabela 4 – Aceleração das Arquiteturas

Benchmark	Arquitetura	Ciclos	ILP Médio	Aceleração Hivek-RT 1T	Aceleração ρ-VEX	Aceleração RIDECORE
Bitcount	ρ-VEX	9.950.901	1,66	5,49	1,00	0,95
	RideCore	9.411.639	1,75	5,81	1,06	1,00
	Hivek-RT 1T	54.676.368	0,30	1,00	0,18	0,17
	Hivek-RT 4T	13.805.783	1,20	3,96	0,72	0,68
	Hivek-RT 16T	3.485.618	4,73	15,69	2,85	2,70
	Hivek-RT 1T + CGRA	23.104.896	0,71	2,37	0,43	0,41
	Hivek-RT 4T + CGRA	5.833.986	2,83	9,37	1,71	1,61
	Hivek-RT 16T + CGRA	1.487.378	11,09	36,76	6,69	6,33
Susan	ρ-VEX	22.040.099	1,42	4,75	1,00	0,81
	RideCore	17.838.764	1,75	5,87	1,24	1,00
	Hivek-RT 1T	104.753.328	0,30	1,00	0,21	0,17
	Hivek-RT 4T	28.231.022	1,11	3,71	0,78	0,63
	Hivek-RT 16T	7.149.415	4,37	14,65	3,08	2,50
	Hivek-RT 1T + CGRA	43.456.288	0,72	2,41	0,51	0,41
	Hivek-RT 4T + CGRA	12.167.761	2,57	8,61	1,81	1,47
	Hivek-RT 16T + CGRA	3.014.780	10,37	34,75	7,31	5,92
SHA-1	ρ-VEX	6.078.190	1,85	6,17	1,00	1,05
	RideCore	6.387.073	1,76	5,88	0,95	1,00
	Hivek-RT 1T	37.525.128	0,30	1,00	0,16	0,17
	Hivek-RT 4T	9.897.253	1,13	3,79	0,61	0,65
	Hivek-RT 16T	2.549.363	4,40	14,72	2,38	2,51
	Hivek-RT 1T + CGRA	14.368.064	0,78	2,61	0,42	0,44
	Hivek-RT 4T + CGRA	3.854.233	2,91	9,74	1,58	1,66
	Hivek-RT 16T + CGRA	971.640	11,55	38,62	6,26	6,57
Patricia	ρ-VEX	7.565.926	1,23	4,82	1,00	0,92
	RideCore	6.995.197	1,33	5,21	1,08	1,00
	Hivek-RT 1T	36.440.048	0,26	1,00	0,21	0,19
	Hivek-RT 4T	9.702.163	0,96	3,76	0,78	0,72
	Hivek-RT 16T	2.466.536	3,77	14,77	3,07	2,84
	Hivek-RT 1T + CGRA	27.949.500	0,33	1,30	0,27	0,25
	Hivek-RT 4T + CGRA	7.553.352	1,23	4,82	1,00	0,93
	Hivek-RT 16T + CGRA	1.956.465	4,76	18,63	3,87	3,58
StringSearch	ρ-VEX	23.331.704	1,34	4,85	1,00	0,99
	RideCore	23.168.789	1,35	4,88	1,01	1,00
	Hivek-RT 1T	113.147.076	0,28	1,00	0,21	0,20
	Hivek-RT 4T	30.266.843	1,03	3,74	0,77	0,77
	Hivek-RT 16T	7.778.861	4,02	14,55	3,00	2,98
	Hivek-RT 1T + CGRA	75.792.460	0,41	1,49	0,31	0,31
	Hivek-RT 4T + CGRA	20.274.483	1,54	5,58	1,15	1,14
	Hivek-RT 16T + CGRA	5.305.472	5,89	21,33	4,40	4,37
Matriz	ρ-VEX	186.663	1,45	4,18	1,00	1,01
	RideCore	188.510	1,44	4,14	0,99	1,00
	Hivek-RT 1T	780.400	0,35	1,00	0,24	0,24
	Hivek-RT 4T	195.080	1,39	4,00	0,96	0,97
	Hivek-RT 16T	60.884	4,46	12,82	3,07	3,10
	Hivek-RT 1T + CGRA	182.712	1,49	4,27	1,02	1,03
	Hivek-RT 4T + CGRA	45.720	5,94	17,07	4,08	4,12
	Hivek-RT 16T + CGRA	25.808	10,52	30,24	7,23	7,30

Fonte: autoria própria

formato VCD e identifica os instantes em que o sinal de fim de iteração é ativado. A partir desses valores foi gerado o gráfico apresentado na Figura 45. Ele mostra que na arquitetura proposta, os intervalos entre uma iteração e outra se mantém constantes, o que evidencia as características de repetibilidade da arquitetura proposta, diferente do ρ-VEX e do RIDECORE, onde os intervalos variam. Essas variações tem duas fontes: a memória cache e o mecanismo de troca de contexto. Como a memória cache é compartilhada entre

as threads no ρ-VEX e no RIDECORE, há momentos onde uma thread sobrescreve os dados da outra, o que interfere no tempo de execução. Já a troca de contexto interfere no tempo de execução da thread porque quando a thread está executando no ρ-VEX e no RIDECORE, ela não sabe a hora que será pre-emptada. Ela é retirada de forma brusca do pipeline e quando volta para o mesmo, o seu estado de execução é diferente de antes da pre-empção. Aqui deve-se entender estado de execução todo o estado da arquitetura, incluindo aqueles não controlados por software.

Essas mudanças de estado forçadas, seja por miss na cache ou por pre-empção, levam a diferentes tempos de execução no caso do ρ-VEX e RIDECORE, o que por sua vez levam a estimativas pessimistas de WCET. Já a arquitetura proposta, por outro lado, apresenta um intervalo de repetição preciso a nível de ciclo de relógio, o que permite uma estimativa de WCET muito mais otimista. A arquitetura proposta consegue esse resultado porque consegue isolar temporalmente uma thread das demais. Essa isolação é obtida projetando o hardware de tal forma que haja o mínimo possível de disputa por recursos entre as threads e garantindo uma fatia de tempo de execução para cada uma. Nos recursos em há disputa como a memória principal, por exemplo, há uma política e mecanismo de acesso que permite garantir previsibilidade e repetibilidade. No exemplo da memória RAM, é garantir uma latência conhecida a priori pelo software.

É importante ressaltar mais uma vez que esses resultados foram obtidos pelo processamento sempre das mesmas entradas. Entradas diferentes provavelmente levariam a tempos de execução diferentes em todas as arquiteturas. De qualquer forma, o objetivo aqui era mostrar que a arquitetura proposta é muito mais amigável para sistemas de tempo real do que uma arquitetura clássica como é o caso do ρ-VEX e do RIDECORE.

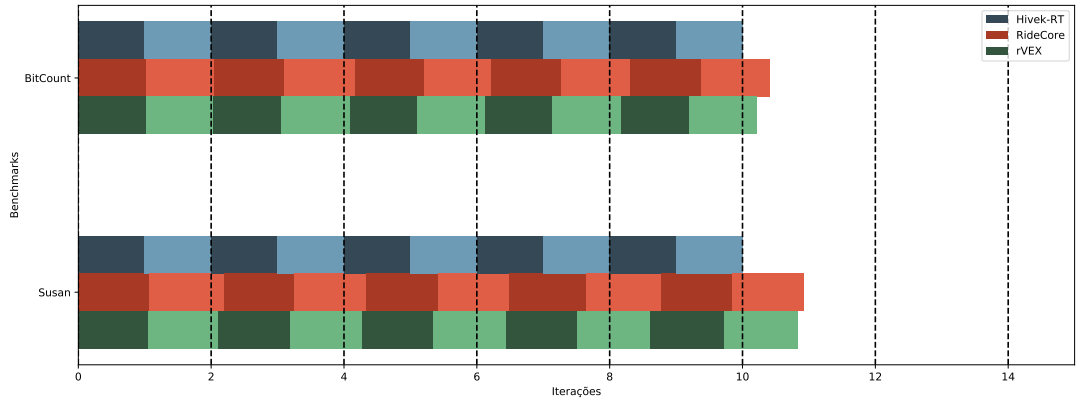
Esta seção finaliza o capítulo de experimentos e resultados. Em resumo, pode-se dizer que a arquitetura conseguiu obter resultados satisfatórios quando comparado a processadores tradicionais tais como VLIW e superescalar. O próximo capítulo é destinado a conclusão e futuros trabalhos.

5.6 Considerações Finais sobre Resultados Obtidos

A arquitetura proposta foi comparada com arquiteturas clássicas tais como processadores VLIW e superescalares. Essas arquiteturas foram escolhidas porque ou representam bem arquiteturas usadas em sistemas embarcados como é o caso do VLIW ou arquiteturas de alto desempenho em ambientes desktop como é o caso dos superescalares com execução fora de ordem. Foram comparados várias versões da arquitetura proposta (um núcleo, quatro núcleos, com ou sem CGRA etc) com os processadores ρ-VEX e RIDECORE.

No geral, um núcleo da arquitetura proposta consegue performar melhor do que o ρ-VEX e o RIDECORE. Considerando que as áreas de um núcleo da arquitetura proposta,

Figura 45 – Gráfico Temporal



Fonte: autoria própria

composto de processador e CGRA, consegue ser menor que o RIDECORE e só um pouco maior que o ρ-VEX e alcançar quase 2x o desempenho do ρ-VEX e do RIDECORE, pode-se dizer que nesse sentido o núcleo da arquitetura proposta consegue prover melhor desempenho com um mesmo *budget* de área e, provavelmente, de energia e potência. Esse resultado pode ser conferido analisando os resultados de desempenho do ρ-VEX e RIDECORE com a coluna *HIVEK-RT 4T + CGRA*.

Apesar da versão multicore não ser possível de comparar diretamente, uma vez que se trata de comparar uma arquitetura multicore com arquiteturas single-core como é o caso do ρ-VEX e RIDECORE, ficam os resultados documentados para trabalhos futuros. Todavia, como cada núcleo individual conseguiu performar melhor na maioria dos benchmarks, é possível levantar a hipótese de que um multicore ρ-VEX, por exemplo, tenha desempenho inferior ao da arquitetura proposta.

No caso do *benchmark* da multiplicação de matrizes, o emprego do CGRA é tão relevante que mesmo a versão *HIVEK-RT 1T + CGRA* consegue performar melhor que o ρ-VEX e o RIDECORE. Esse resultado é bastante satisfatório, pois é o equivalente a dizer que a arquitetura proposta, rodando com um clock com frequência 4x menor, ainda assim consegue performar melhor que arquiteturas clássicas. Obviamente, esse *benchmark* em especial foi projetado para extrair o máximo possível da arquitetura proposta, mesmo assim é um *benchmark* que consegue mostrar o potencial da arquitetura.

No que diz respeito aos resultados de análise temporal, foram escolhidos dois *benchmarks* para evidenciar as características de repetibilidade da arquitetura proposta. Através do gráfico apresentado na Figura 45, é possível perceber que a arquitetura não apresenta anomalias temporais (para um mesmo conjunto de entradas). A ausência dessas

anomalias permite uma análise de WCET mais simplificada e robusta, mostrando que a arquitetura proposta além de apresentar bom desempenho, também conseguiu respeitar os conceitos das *Precision-Timed Machines*, tornando-a apta para o uso em sistemas de tempo real.

Conclusão e Trabalhos Futuros

Esta tese apresentou uma proposta e implementação de uma arquitetura de alto desempenho voltada para sistemas de tempo real. A arquitetura consiste em um processador quad-core, com cada núcleo consistindo de um processador SMT trabalhando em conjunto com um *Coarse Grain Reconfigurable Array*. A arquitetura mostrou desempenho satisfatórios para todos os benchmarks testados.

5.7 Resultados

A arquitetura proposta foi comparada com duas arquiteturas convencionais: o RIDECORE, aqui representando processadores superescalares com execução fora de ordem e o ρ-VEX, aqui representando processadores VLIW. Resultados de desempenho mostraram que a arquitetura é capaz de performar melhor que o RIDECORE e o VLIW quando existe um nível mínimo de TLP. No caso da multiplicação de matrizes, mesmo na ausência de TLP, a arquitetura proposta conseguiu performar melhor que o RIDECORE e o ρ-VEX.

Além de performar melhor que arquiteturas clássicas, os testes temporais mostraram que a arquitetura proposta consegue prover repetibilidade e previsibilidade, características que podem ser usadas para facilitar bastante na estimativa de WCET e ajudar em geral no desenvolvimento de aplicações de tempo real. Entretanto, para poder explorar essa característica, a arquitetura demanda um novo modelo de programação, visto que é necessário particionar o que será executado no array reconfigurável e o que será executado no processador.

5.8 Contribuições

Para o conhecimento do autor desta tese, esse é o primeiro trabalho da literatura a propor um CGRA para arquiteturas PRET. As arquiteturas PRET são arquiteturas voltadas para sistemas de tempo real, tendo como foco principal sistemas cyber-físicos. Elas garantem como principal característica a previsibilidade e repetibilidade, ao passo que arquiteturas rivais se focam apenas no componente de previsibilidade. O estado-da-arte atual das máquinas PRET consiste de processadores single-core ou, mais recente, de um multicore, porém nenhuma delas explora ILP no grau do presente trabalho. Esse alto grau de exploração de ILP vem do uso de um array reconfigurável trabalhando em conjunto com o processador PRET e que servem de base para os núcleos de um multicore, aqui denominado de MultiHivek-RT. Como mostrado nos resultados, isso possibilita uma arquitetura de alto desempenho enquanto consegue atender aplicações de tempo real.

5.9 Trabalhos Futuros

Há uma gama variada de exploração que pode ser realizada na arquitetura proposta, bem como propostas de arquiteturas totalmente diferentes da apresentada aqui. No que diz respeito ao atual trabalho, exemplos de trabalhos futur

Questões energéticas continuam em aberto para máquinas PRET. Por causa do pipeline intercalado, é mais difícil hibernar o processador. Uma thread pode estar ociosa ou com baixa carga de trabalho, porém por causa de outra que está ativa, a frequência de operação não pode ser reduzida a fim de economizar energia. A intercalação das threads ocorrem em uma granularidade tão fina que não compensa hibernar o processador e depois voltar a execução normal.

No que diz respeito a hierarquia de memória, o uso apenas de scratchpads coloca sobre o programador o fardo de gerenciar manualmente a memória. Existe grande espaço para pesquisa sobre como integrar memórias caches à máquinas PRET a fim de facilitar a programação dessas arquiteturas. Além disso, a utilização de várias memórias scratchpads trabalhando em paralelo se mostrou bastante útil no sentido de que permitiu uma melhor exploração de paralelismo a nível de dados na memória. Um estudo mais detalhado sobre possíveis padrões de armazenamento e acesso a esses dados pode resultar em uma hierarquia mais otimizada e eficiente para a exploração de paralelismo.

Com relação ao CGRA em si, a estrutura proposta nesse trabalho foi suficiente para obter bons resultados, porém sabe-se que há espaços para melhorias. Por exemplo, em vários benchmarks os elementos de processamento não utilizavam suas ULAs, servindo apenas para armazenar variáveis temporárias. Pode-ser realizar um estudo para determinar um número melhor de registradores e de unidades funcionais. Já o sistema de comunicação que foi implementado usando crossbar, sabe-se que não é escalável porém facilita o escalonamento de operações nos elementos de processamento.

A problemática inicial que deu início a esse trabalho ainda persiste: reduzir a latência de cada thread a fim de obter bom desempenho em aplicações com baixo grau de paralelismo a nível de thread. A proposta inicial para se resolver esse problema era executar instruções de forma assíncrona em um único ciclo de relógio a fim de compensar o intercalamento das threads. Apesar de haver trabalhos como (RUTZIG; BECK; CARRO, 2013a) que obtiveram resultados satisfatórios com o uso de operações assíncronas, essa abordagem só não foi levada adiante por falta de acesso à ferramentas de síntese para validar propostas de arquitetura nesse sentido. Sem as ferramentas, acabou-se por propor a exploração de forma síncrona a qual era passível de validação.

O compilador iniciado neste trabalho é outra frente que pode ser abordada. Como máquinas PRET dependem de uma boa programação para obter desempenho satisfatório, um bom compilador é uma ferramenta essencial. Mais ainda: existe bastante espaço

para o estudo de algoritmos de exploração automática de ILP para gerar configurações automatizadas para o CGRA. Há possibilidades de trabalho também no desenvolvimento de ferramentas para fazer análises estáticas de WCET voltadas para arquiteturas PRET. Como arquiteturas PRET são mais previsíveis, espera-se que seja possível desenvolver ferramentas que se aproveitem dessa previsibilidade.

Referências

ALSOLAIM, A. et al. Architecture and application of a dynamically reconfigurable hardware array for future mobile communication systems. In: IEEE. *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*. [S.l.], 2000. p. 205–214. Citado na página 49.

ANSALONI, G.; BONZINI, P.; POZZI, L. Egra: A coarse grained reconfigurable architectural template. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, IEEE, v. 19, n. 6, p. 1062–1074, 2011. Citado na página 53.

BARR, M. Bookout vs. toyota. *case No. CJ-2008-7969, District Court of Oklahoma County, http://www. safetyresearch. net/Library/Bookout_v_Toyota_Barr_redacted. pdf, consultado el*, v. 10, 2013. Citado na página 23.

BAUMGARTE, V. et al. Pact xpp—a self-reconfigurable data processing architecture. *the Journal of Supercomputing*, Springer, v. 26, n. 2, p. 167–184, 2003. Citado na página 51.

BITTNER, R.; ATHANAS, P. M.; MUSGROVE, M. Colt: An experiment in wormhole run-time reconfiguration. In: INTERNATIONAL SOCIETY FOR OPTICS AND PHOTONICS. *High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic*. [S.l.], 1996. v. 2914, p. 187–195. Citado na página 48.

BURNS, A.; DAVIS, R. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep*, p. 1–69, 2013. Citado na página 24.

BUTTAZZO, G. C. *Hard real-time computing systems: predictable scheduling algorithms and applications*. [S.l.]: Springer Science & Business Media, 2011. v. 24. Citado 2 vezes nas páginas 23 e 25.

CEBRIAN, J. M.; NATVIG, L.; MEYER, J. C. Improving energy efficiency through parallelization and vectorization on intel core i5 and i7 processors. In: IEEE. *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. [S.l.], 2012. p. 675–684. Citado na página 43.

CHARETTE, R. N. This car runs on code. *IEEE spectrum*, IEEE, v. 46, n. 3, p. 3, 2009. Citado na página 24.

CHEN, D. C.; RABAEY, J. M. A reconfigurable multiprocessor ic for rapid prototyping of algorithmic-specific high-speed dsp data paths. *IEEE Journal of Solid-State Circuits*, IEEE, v. 27, n. 12, p. 1895–1904, 1992. Citado na página 48.

CICHON, G. et al. Synchronous transfer architecture (sta). In: SPRINGER. *International Workshop on Embedded Computer Systems*. [S.l.], 2004. p. 343–352. Citado na página 51.

CONFIGURABLE Logic Cell Tips n Tricks. <http://ww1.microchip.com/downloads/en/DeviceDoc/41631B.pdf>. Accesso em 30 de setembro de 2018. Citado na página 43.

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 53.

CRONQUIST, D. C. et al. Architecture design of reconfigurable pipelined datapaths. In: IEEE. *Advanced Research in VLSI, 1999. Proceedings. 20th Anniversary Conference on*. [S.l.], 1999. p. 23–40. Citado na página 49.

CZAJKOWSKI, T. S. et al. From opencl to high-performance hardware on fpgas. In: IEEE. *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. [S.l.], 2012. p. 531–534. Citado na página 43.

DANILIN, A.; BENNEBROEK, M.; SAWITZKI, S. Astra: An advanced space-time reconfigurable architecture. In: IEEE. *Field Programmable Logic and Applications, 2006. FPL'06. International Conference on*. [S.l.], 2006. p. 1–4. Citado na página 52.

DAVID, R. et al. Dart: a dynamically reconfigurable architecture dealing with future mobile telecommunications constraints. *Résumé*, 2003. Citado na página 51.

DUTTA, H. et al. A holistic approach for tightly coupled reconfigurable parallel processors. *Microprocessors and Microsystems*, Elsevier, v. 33, n. 1, p. 53–62, 2009. Citado na página 52.

EDWARDS, S. A.; LEE, E. A. The case for the precision timed (pret) machine. In: ACM. *Proceedings of the 44th annual Design Automation Conference*. [S.l.], 2007. p. 264–265. Citado 4 vezes nas páginas 23, 25, 31 e 80.

FUJINAMI, M. et al. *RIsc-v Dynamic Execution CORE*. <https://github.com/ridecore/ridecore>. Accesso em 29 de setembro 2020. Citado na página 61.

GALANIS, M. D. et al. A reconfigurable coarse-grain data-path for accelerating computational intensive kernels. *Journal of Circuits, Systems, and Computers*, World Scientific, v. 14, n. 04, p. 877–893, 2005. Citado na página 52.

GOLDSTEIN, S. C. et al. Piperench: A reconfigurable architecture and compiler. *Computer*, IEEE, v. 33, n. 4, p. 70–77, 2000. Citado na página 49.

GOVINDARAJU, V. et al. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*, IEEE, v. 32, n. 5, p. 38–51, 2012. Citado na página 53.

GUTHAUS, M. R. et al. Mibench: A free, commercially representative embedded benchmark suite. In: IEEE. *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*. [S.l.], 2001. p. 3–14. Citado 2 vezes nas páginas 89 e 94.

HAMEED, R. et al. Understanding sources of inefficiency in general-purpose chips. In: ACM. *ACM SIGARCH Computer Architecture News*. [S.l.], 2010. v. 38, n. 3, p. 37–47. Citado na página 43.

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 44.

HARTENSTEIN, R. W.; KRESS, R. A datapath synthesis system for the reconfigurable datapath architecture. In: IEEE. *Design Automation Conference, 1995. Proceedings of the ASP-DAC'95/CHDL'95/VLSI'95., IFIP International Conference on Hardware*

Description Languages. IFIP International Conference on Very Large Scal. [S.l.], 1995. p. 479–484. Citado na página 48.

HAUSER, J. R.; WAWRZYNEK, J. Garp: A mips processor with a reconfigurable coprocessor. In: IEEE. *Field-Programmable Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on.* [S.l.], 1997. p. 12–21. Citado na página 48.

HILI, N.; GIRAULT, A.; JENN, E. Worst-case reaction time optimization on deterministic multi-core architectures with synchronous languages. In: IEEE. *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA).* [S.l.], 2019. p. 1–11. Citado na página 58.

HOOZEMANS, J.; WONG, S.; AL-ARS, Z. Using vliw softcore processors for image processing applications. In: IEEE. *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS).* [S.l.], 2015. p. 315–318. Citado na página 62.

INC, S. I.; WEAVER, D. L. *The SPARC architecture manual.* [S.l.]: Prentice-Hall, 1994. Citado na página 68.

IQBAL, S. M. Z.; LIANG, Y.; GRAHN, H. Parmibench-an open-source benchmark for embedded multiprocessor systems. *IEEE Computer Architecture Letters*, IEEE, v. 9, n. 2, p. 45–48, 2010. Citado na página 95.

KAPASI, U. J. et al. The imagine stream processor. In: IEEE. *null.* [S.l.], 2002. p. 282. Citado na página 50.

KARUNARATNE, M. et al. Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect. In: IEEE. *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE.* [S.l.], 2017. p. 1–6. Citado 3 vezes nas páginas 55, 56 e 57.

KASAPAKI, E. et al. Argo: A real-time network-on-chip architecture with an efficient gals implementation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, IEEE, v. 24, n. 2, p. 479–492, 2015. Citado na página 58.

KIM, C. et al. Ulp-srp: Ultra low power samsung reconfigurable processor for biomedical applications. *2012 International Conference on Field-Programmable Technology*, p. 329–334, 2012. Citado na página 54.

KIM, Y. et al. Design and evaluation of a coarse-grained reconfigurable architecture. In: *Proc. of Int. SoC Design Conf.* [S.l.: s.n.], 2004. p. 227–230. Citado na página 51.

LEE, E. A. Cyber physical systems: Design challenges. In: IEEE. *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC).* [S.l.], 2008. p. 363–369. Citado na página 23.

LEE, E. A. Computing needs time. *Communications of the ACM*, ACM New York, NY, USA, v. 52, n. 5, p. 70–79, 2009. Citado na página 23.

LICKLY, B. et al. Predictable programming on a precision timed architecture. In: ACM. *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems.* [S.l.], 2008. p. 137–146. Citado 2 vezes nas páginas 27 e 33.

LIU, I. et al. A pret microarchitecture implementation with repeatable timing and competitive performance. In: IEEE. *Computer Design (ICCD), 2012 IEEE 30th International Conference on*. [S.l.], 2012. p. 87–93. Citado 3 vezes nas páginas 27, 33 e 80.

LIU, L. et al. A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 52, n. 6, p. 1–39, 2019. Citado na página 26.

LOPES, J.; SOUSA, D.; FERREIRA, J. C. Evaluation of cgra architecture for real-time processing of biological signals on wearable devices. *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, p. 1–7, 2017. Citado na página 54.

LUDWIG, G. L. Evolution of avionic systems architecture, from the 1950's to the present. *SAGARD*, DTIC Document, 1991. Citado na página 32.

MAI, K. et al. Smart memories: A modular reconfigurable architecture. *ACM SIGARCH Computer Architecture News*, ACM, v. 28, n. 2, p. 161–171, 2000. Citado na página 50.

MARSHALL, A. et al. A reconfigurable arithmetic array for multimedia applications. In: ACM. *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*. [S.l.], 1999. p. 135–143. Citado na página 49.

MARTIN, G.; CHANG, H. *Winning the SoC revolution: experiences in real design*. [S.l.]: Springer Science & Business Media, 2012. Citado na página 43.

MEI, B. et al. Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In: SPRINGER. *International Conference on Field Programmable Logic and Applications*. [S.l.], 2003. p. 61–70. Citado na página 50.

MIRSKY, E.; DEHON, A. et al. Matrix: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In: *FCCM*. [S.l.: s.n.], 1996. v. 96, p. 17–19. Citado na página 48.

MIYAMORI, T.; OLUKOTUN, K. Remarc: Reconfigurable multimedia array coprocessor. *IEICE Transactions on information and systems*, The Institute of Electronics, Information and Communication Engineers, v. 82, n. 2, p. 389–397, 1999. Citado na página 49.

OBERMAISSER, R. et al. From a federated to an integrated automotive architecture. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, v. 28, n. 7, p. 956–965, July 2009. ISSN 0278-0070. Citado 2 vezes nas páginas 32 e 33.

PARK, H.; PARK, Y.; MAHLKE, S. Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications. In: ACM. *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. [S.l.], 2009. p. 370–380. Citado na página 52.

PARK, Y.; PARK, H.; MAHLKE, S. Cgra express: accelerating execution using dynamic operation fusion. In: ACM. *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*. [S.l.], 2009. p. 271–280. Citado na página 53.

PUTNAM, A. et al. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News*, ACM, v. 42, n. 3, p. 13–24, 2014. Citado na página 43.

RUTZIG, M. B.; BECK, A. C. S.; CARRO, L. A transparent and energy aware reconfigurable multiprocessor platform for simultaneous ilp and tlp exploitation. In: EDA CONSORTIUM. *Proceedings of the Conference on Design, Automation and Test in Europe*. [S.l.], 2013. p. 1559–1564. Citado 3 vezes nas páginas 26, 103 e 112.

RUTZIG, M. B.; BECK, A. C. S.; CARRO, L. A transparent and energy aware reconfigurable multiprocessor platform for simultaneous ilp and tlp exploitation. In: EDA CONSORTIUM. *Proceedings of the Conference on Design, Automation and Test in Europe*. [S.l.], 2013. p. 1559–1564. Citado 2 vezes nas páginas 58 e 59.

SANKARALINGAM, K. et al. Trips: A polymorphous architecture for exploiting ilp, tlp, and dlp. *ACM Transactions on Architecture and Code Optimization (TACO)*, ACM, v. 1, n. 1, p. 62–93, 2004. Citado na página 51.

SCHMIDT, K. et al. A novel asic design approach based on a new machine paradigm. *J. SSC*, 1991. Citado na página 47.

SCHOEBERL, M. et al. T-crest: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, Elsevier, v. 61, n. 9, p. 449–471, 2015. Citado na página 58.

SCHOEBERL, M. et al. Patmos: A time-predictable microprocessor. *Real-Time Systems*, Springer, v. 54, n. 2, p. 389–423, 2018. Citado na página 58.

SHEN, J. P.; LIPASTI, M. H. *Modern processor design: fundamentals of superscalar processors*. [S.l.]: Waveland Press, 2013. Citado na página 61.

SINGH, H. et al. Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE transactions on computers*, IEEE, n. 5, p. 465–481, 2000. Citado 2 vezes nas páginas 38 e 50.

SIQUEIRA, H. *Proposta de um processador multithreading com caracteristicas de previsibilidade*. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Norte, 2015. Citado 5 vezes nas páginas 26, 33, 37, 38 e 60.

SIQUEIRA, H.; KREUTZ, M. A simultaneous multithreading processor architecture with predictable timing behavior. In: *Proceedings of the 2018 Brazilian Symposium on Computing Systems Engineering*. [S.l.: s.n.], 2018. Citado 2 vezes nas páginas 27 e 28.

SMIT, G. J. et al. The chameleon architecture for streaming dsp applications. *EURASIP Journal on Embedded Systems*, Springer, v. 2007, n. 1, p. 078082, 2007. Citado na página 52.

SOUZA, J. D. et al. A reconfigurable heterogeneous multicore with a homogeneous isa. In: IEEE. *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*. [S.l.], 2016. p. 1598–1603. Citado na página 53.

SUTTER, B. D.; RAGHAVAN, P.; LAMBRECHTS, A. Coarse-grained reconfigurable array architectures. In: *Handbook of signal processing systems*. [S.l.]: Springer, 2019. p. 427–472. Citado na página 44.

SWANSON, S. et al. The wavescalar architecture. *ACM Transactions on Computer Systems (TOCS)*, ACM, v. 25, n. 2, p. 4, 2007. Citado na página 52.

SWEETMAN, D. *See MIPS run*. [S.l.]: Elsevier, 2010. Citado na página 68.

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 44.

UNGERER, T. et al. parmerasa–multi-core execution of parallelised hard real-time applications supporting analysability. In: IEEE. *2013 Euromicro Conference on Digital System Design*. [S.l.], 2013. p. 363–370. Citado na página 58.

WAFAA, A. Introducing the 64-bit armv8 architecture. In: *Open Source Arm Ltd. EuroBSDCon conference, Malta*. [S.l.: s.n.], 2013. Citado na página 68.

WAINGOLD, E. et al. Baring it all to software: The raw machine. Massachusetts Institute of Technology, 1997. Citado na página 49.

WATERMAN, A. et al. *The risc-v instruction set manual. volume 1: User-level isa, version 2.0*. [S.l.], 2014. Citado na página 68.

WATKINS, C.; WALTER, R. Transitioning from federated avionics architectures to integrated modular avionics. In: *Digital Avionics Systems Conference, 2007. DASC '07. IEEE/AIAA 26th*. [S.l.: s.n.], 2007. p. 2.A.1–1–2.A.1–10. Citado na página 33.

WIJTVLIET, M. et al. Position paper: Reaching intrinsic compute efficiency requires adaptable microarchitectures. *Programmability and Architectures for Heterogeneous Multicores (MULTIPROG-2016)*, p. 25–31, 2016. Citado na página 43.

WIJTVLIET, M.; WAEIJEN, L.; CORPORAAL, H. Coarse grained reconfigurable architectures in the past 25 years: Overview and classification. In: . [S.l.: s.n.], 2016. p. 235–244. Citado 5 vezes nas páginas 39, 40, 44, 45 e 47.

WILHELM, R. et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, ACM New York, NY, USA, v. 7, n. 3, p. 1–53, 2008. Citado na página 24.

WONG, S.; AS, T. V.; BROWN, G. ρ-vex: A reconfigurable and extensible softcore vliw processor. In: IEEE. *2008 International Conference on Field-Programmable Technology*. [S.l.], 2008. p. 369–372. Citado na página 62.

YE, Z. A. et al. Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In: ACM. *ACM SIGARCH computer architecture news*. [S.l.], 2000. v. 28, n. 2, p. 225–235. Citado na página 50.

ZHANG, H. et al. A 1-v heterogeneous reconfigurable dsp ic for wireless baseband digital signal processing. *IEEE Journal of Solid-State Circuits*, IEEE, v. 35, n. 11, p. 1697–1704, 2000. Citado na página 50.

ZIMMER, M. et al. Flexpret: A processor platform for mixed-criticality systems. In: IEEE. *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*. [S.l.], 2014. p. 101–110. Citado 2 vezes nas páginas 27 e 33.