Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

NINA

Processador RISC-V para Aprendizado de Máquina com Recursos Limitados

Guilherme Moreno Silva

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Prof. Dr. Alfredo Goldman

São Paulo
2024

Resumo

Guilherme Moreno Silva. **NINA: *Processador RISC-V para Aprendizado de Máquina com Recursos Limitados***. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2024.

Processadores tem o crescimento constante de seu poder computacional há décadas. Todavia, a desaceleração da Lei de Moore e, consequentemente, dos avanços em processos de fabricação impõem um desafio para arquitetos de computadores, que contam com diminutas melhorias em desempenho e consumo energético outrora proporcionados por tais avanços. Nesse contexto, a abordagem de *hardware-software co-design*, surge como alternativa para a manutenção desse progresso, projetando o *hardware* otimizado para *softwares* específicos. Este trabalho apresenta o NINA, um processador RISC-V otimizado para aprendizado de máquina por meio de *co-design*. Partindo do algoritmo de inferência em redes neurais por multiplicação de matrizes, essa metodologia permite adaptar a organização do processador para mitigar situações de dependência entre instruções, responsáveis pela degradação de desempenho, com maior eficiência em comparação a abordagens tradicionais de otimização.

Palavras-chave: processador RISC-V. co-design. aprendizado de máquina. branch penalty. early branch resolution. inferência. microarquitetura.

Abstract

Guilherme Moreno Silva. **NINA:** ***A RISC-V Core for Resource-Constrained Machine Learning***. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2024.

Processors have maintained a steady increase in computational power over the decades. However, the deceleration of Moore's Law and, thus, advancements in lithographic fabrication processes impose a challenge for computer architects, who suffer from ever-smaller increases in performance and energy efficiency with every new process generation. In this context, hardware-software co-design offers a path for increasing performance by optimizing the hardware design for specific applications. This work introduces NINA, a RISC-V soft-core optimized for resource-constrained machine learning inference through co-design. This approach enables the implementation of a microarchitecture that more efficiently resolves pipeline hazards responsible for performance degradation and found on matrix multiplication algorithms used for inference, compared to traditional design approaches.

Keywords: RISC-V core. co-design. machine learning. branch penalty. early branch resolution. neural network inference. microarchitecture.

Lista de abreviaturas

ALU	*Arithmetic Logic Unit* (Unidade Lógica e Aritmética)
ASIC	*Application-specific Integrated Circuit* (Circuito Integrado de Aplicação Específica)
BHT	*Branch History Table* (Tabela de histórico de desvios)
CPI	*Cicles per Instruction* (Ciclos por Instrução)
CPU	*Central Processing Unit* (Unidade Central de Processamento)
DNN	*Deep Neural Network* (Rede Neural Profunda)
FPGA	*Field-Programmable Gate Array* (Matriz de Portas Programáveis em Campo)
GPU	*Graphics Processing Unit* (Unidade de Processamento Gráfico)
ILP	*Instruction-Level Parallelism* (Paralelismo em Nível de Instrução)
IR	*Instruction Register* (Registrador de Instrução)
ISA	*Instruction Set Architecture* (Arquitetura do Conjunto de Instruções)
LUT	*Lookup Table* (Tabela de Consulta)
PC	*Program Counter* (Contador do Programa)
PLD	*Programmable Logic Devices* (Dispositivo Lógico Programável)
PPA	*Power, Performance and Area* (Consumo, Desempenho e Área de Silício)
RTL	*Register-Transfer Level* (Nível de Transferência Entre Registradores)
SIMD	*Single Instruction, Multiple Data* (Única instrução operando em múltiplos dados)

Lista de figuras

Lista de tabelas

Lista de programas

Sumário

Capítulo 1

Introdução

> *[...] as análises [...] mostram que a história de um conceito não é, de forma alguma, a de seu refinamento progressivo, de sua racionalidade continuamente crescente, de seu gradiente de abstração, mas a de seus diversos campos de constituição e de validade, a de suas regras sucessivas de uso, a dos meios teóricos múltiplos em que foi realizada e concluída sua elaboração.*
>
> — Michel Foucault

1.1 Motivação

Nas últimas cinco décadas, os computadores tornaram-se componentes indissociáveis ao progresso tecnológico e social. A miniaturização dos circuitos eletrônicos permitiu que eles deixassem as poucas e enorme salas outrora habitadas para ocupar as mãos de cada cidadão, na forma de celulares, relógios e outros dispositivos inteligentes.

Concomitantemente, a evolução da capacidade computacional permite que se desenvolvam aplicações cada vez mais complexas, ou reciprocamente, que elas sejam executadas em dispositivos cada vez mais compactos. Neste sentido, com as novas tendências de computação ubíqua, a importância do desenvolvimento de processadores cada vez mais eficientes constitui um importante desafio das próximas cinco décadas.

1.2 Definição do Problema

Para arquitetos de computadores, os avanços na fabricação de semicondutores podem – e são – amplamente explorados como ferramentas no projeto de processadores. Seja aumentando a quantidade de transistores ou melhorando suas características, esses avanços possibilitam implementar processadores mais complexos e com melhor desempenho.

O problema fundamental é que a medida que o número de transistores em um circuito integrado cresce, também cresce a probabilidade de que existam unidades defeituosas. Em 1965, Gordon Moore notou que, para circuitos simples, o custo por transistor era quase inversamente proporcional à sua quantidade. Porém, a medida que o circuito se tornava

mais complexo, o aumento de unidades defeituosas acabava por compensar essa diferença. Assim, conforme o rendimento era aperfeiçoado, o fator econômico motivava transistores cada vez menores. Moore observou também que o número de transistores por circuito integrado dobrava a cada ano e, a partir disto, previu que esse ritmo se manteria pelos próximos anos (Moore, 1998). Tal obervação, a Lei de Moore – o tamanho do transistor mais economicamente vantajoso iria dobrar a cada cerca de 18 meses – definiu o futuro da computação.

Outro domínio importante para arquitetos é o consumo de energia. A maior densidade de transistores proporcionada pela Lei de Moore também representava um potencial maior consumo. Uma opção para aliviar esse problema poderia ser a diminuição da frequência de operação do circuito, fator proporcional ao consumo (Kim *et al.*, 2003). Entretanto, isso nem sempre é possível, pois a perda de desempenho pode ser superior aos ganhos de um desenho mais complexo. Uma solução conveniente é dada pela Escalabilidade de Dennard. Ao escalar por um mesmo fator, diga-se k, o tamanho do transistor, sua tensão e sua dopagem, a frequência de operação é aumentada pelo mesmo fator k, mantendo a densidade de potência (Dennard *et al.*, 1974). Isso permitia adicionar mais transistores, com maior frequência, sem afetar o consumo.

Por três décadas, a Lei de Moore ao lado da Escalabilidade de Dennard determinaram a cadência de inovação na fabricação de semicondutores, sendo responsáveis, por um lado, em garantir aos arquitetos de computadores um caminho pelo qual poderiam implementar processadores ainda mais complexos e, consequentemente, com *performance* superior. Por outro lado, o *software* se tornava constantemente mais rápido. Neste período, o desempenho *single-core* cresceu cerca de seis mil vezes (Hennessy e Patterson, 2017).

Na virada do milênio, a impossibilidade de continuar escalando a tensão limiar e, portanto, a tensão de operação dos transistores levou a uma divergência em relação ao modelo proposto por Dennard, desacelerando os ganhos em densidade de potência.

A energia dissipada por um transistor em operação, chamada potência dinâmica, é dada pela Equação 1.1 (Kim *et al.*, 2003):

$$P_d = CV^2 f \tag{1.1}$$

sendo C a capacitância de carga, V a tensão de entrada e f sua frequência de chaveamento. Como a frequência depende diretamente da tensão, não escalar a tensão limiar rapidamente se tornou um problema. Na prática, ela não apenas representava o aumento no consumo energético, mas também a impossibilidade de manter todos os transistores funcionando simultaneamente em um processador.

Além disso, conforme os transistores continuavam sendo miniaturizados, outras fontes de consumo outrora negligenciáveis passaram a ser fatores dominantes. Com um limite no total de energia dissipada, fenômeno intitulado *power wall*, houve um declínio acentuado na *performance single-thread* das novas gerações de processadores, como pode ser visto na Figura 1.1. Como resposta, os projetistas lançaram mão de diversas novas técnicas e abordagens.

Uma alternativa aos núcleos monolíticos com altas frequências é a utilização de vários núcleos com frequências mais baixas. Do ponto de vista de *software*, essa abordagem é

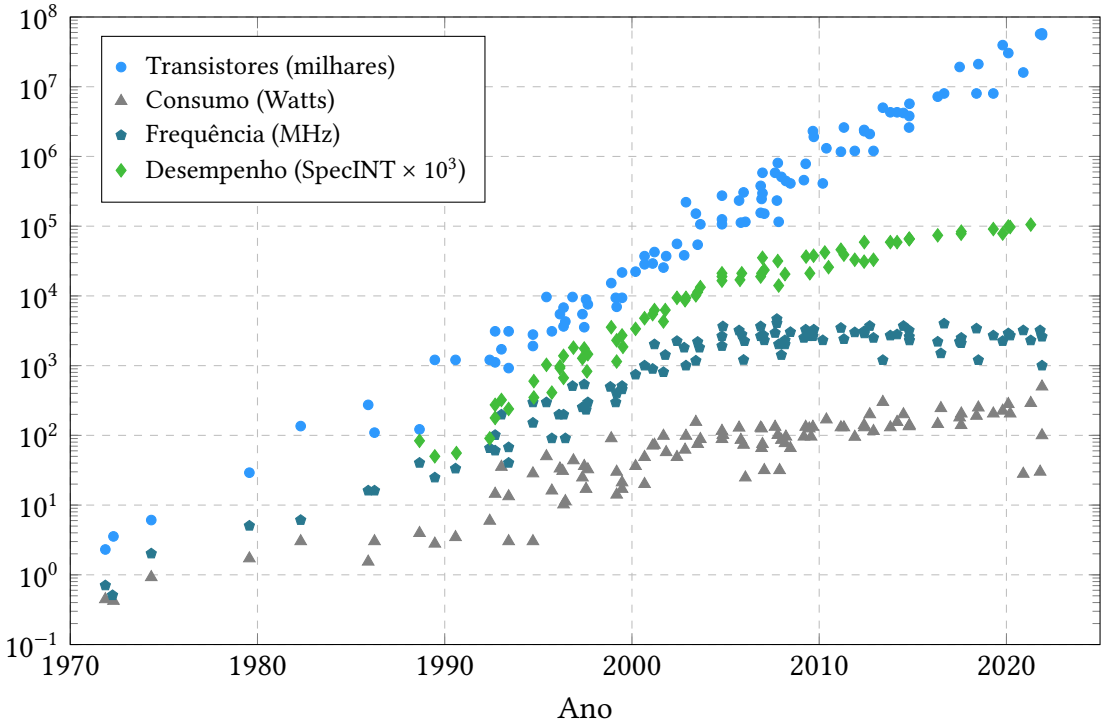

Figura 1.1: *O fim da Escalabilidade de Dennard no início dos anos 2000 colocou um limite no consumo típico, e consequentemente na dissipação de calor, que os arquitetos podem utilizar em seus processadores. Consequentemente, houve efeitos diretos na frequência e no desempenho single-thread nos anos subsequentes. Ainda que transistores continuem diminuindo, novos transistores vem sendo utilizado para outros fins que não na construção de núcleos monolíticos maiores, na busca por melhor desempenho.*
Dados disponíveis em: github.com/karlrupp/microprocessor-trend-data

eficiente ao explorar a capacidade de paralelização das tarefas. Já do *hardware*, a Lei de Moore garantiria novos transistores com o qual implementar esses núcleos sem afetar a estrutura de custo dos processadores. Embora conveniente, e ainda amplamente empregada (veja a Figura 1.2), aumentar arbitrariamente o número de núcleos não é uma solução viável. O fator de *speedup*, ou a relação entre o tempo de execução de uma tarefa com uma unidade de processamento e tempo de execução com n unidades, é limitado pela porção da tarefa que pode ser paralelizada. Esse resultado foi expresso por Gene Amdahl, naquilo que ficou conhecida como a Lei de Amdahl (Amdahl, 2007). Uma representação desse princípio pode ser visto na Equação 1.2, ele afirma que o *speedup* de uma tarefa com fração paralelizável f que recebe aumento S de recursos é dado por:

$$Speedup = \frac{1}{(1-f) + \frac{1}{S}} \tag{1.2}$$

Assim, o ganho potencial fica severamente restringido pela porção estritamente serial das tarefas, a despeito da quantidade de núcleos adicionados ao processador. Mesmo com arranjos mais avançados de múltiplos núcleos dos processadores modernos, a Lei de Amdahl ainda constitui uma barreira para a estratégia *multi-core* (Hill e Marty, 2008). Ademais, há um trabalho adicional necessário por parte de engenheiros de *software* para,

de fato, paralelizar os programas e realizar tais benefícios.

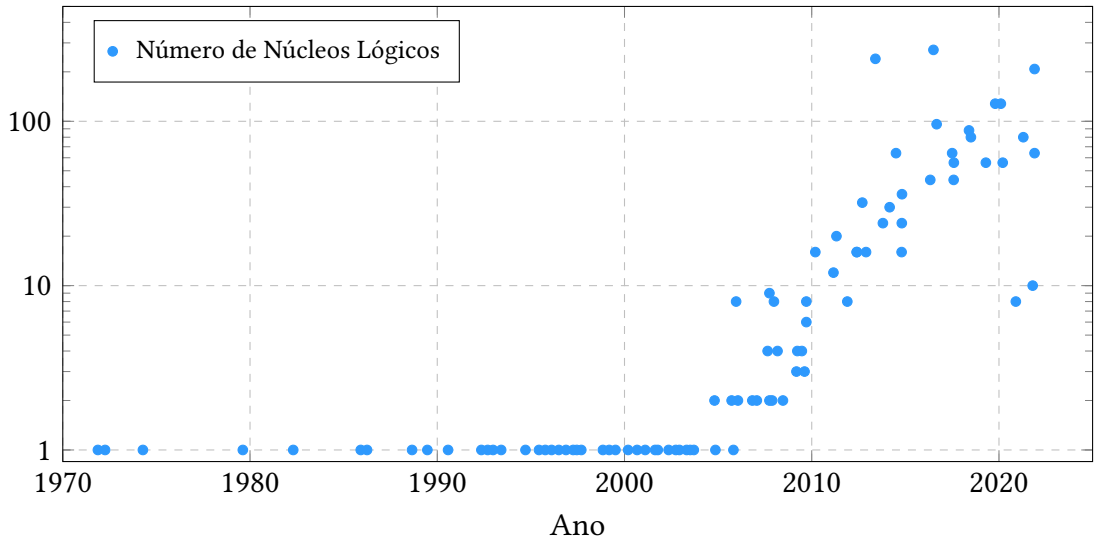

Figura 1.2: *O uso de múltiplos núcleos em processadores surgiu como solução para o power wall. Diminuindo-se a frequência e, consequentemente, a tensão de operação dos transistores, os projetistas exploraram a natureza da dissipação de energia (veja a Equação 1.1) para encapsular mais desempenho, por meio de paralelização, dentro do mesmo limite de dissipação.*
Dados disponíveis em: github.com/karlrupp/microprocessor-trend-data

Outra abordagem consiste em apoiar-se na especialização. Processadores de propósito geral amparam-se no efeito da escala de produção para serem mais competitivos economicamente, comparativamente a *Application Specific Integrated Circuits* (Circuito integrado de aplicação específica) (ASICs). Em contrapartida, os *trade-offs* gerados pela necessidade de conseguirem executar ampla gama de aplicações os fazem muito menos eficientes em termos de desempenho, consumo, e área de silício, ou PPA (do inglês *Power, Performance, Area*).

Por esse motivo, processadores de propósito geral vem gradualmente incorporando otimizações para cargas específicas, como, por exemplo, através da introdução de instruções especificas aos seus conjuntos de instruções, ou *Instruction Set Architecture* (Arquitetura do conjunto de instruções) (ISA). Essas pequenas otimizações, ainda que benéficas, tem eficácia reduzida em virtude da própria natureza das aplicações, que não raro fazem uso somente de um subconjunto dos recursos totais por vez (Hameed *et al.*, 2010).

Para dar continuidade ao crescimento de *performance*, os arquitetos voltaram-se à constante micro-otimização de seus processadores, somando pequenos ganhos. Foi necessário o desenvolvimento de uma metodologia ágil com a qual pudessem validar novas ideias e, principalmente, entregar novas microarquiteturas. A medida que aumenta a flexibilidade de projeto, surgem unidades de processamento cada vez mais especializadas, passando de uma filosofia com múltiplos núcleos de propósito geral, para uma filosofia de múltiplas unidades especializadas, os aceleradores, como resposta aos obstáculos apresentados (Hameed *et al.*, 2010; Wolf, 2003).

Nesse contexto, *hardware-software co-design*, a abordagem de desenvolver *hardware*

baseado nas características do *software* em específico a ser executado, e vice-versa, surge como método para unir os melhores atributos dos ASICs com a escala econômica dos processadores de propósito geral (Hennessy e Patterson, 2017; Hennessy e Patterson, 2018).

A pesquisa desenvolvida nesta tese possui, como resultado, caráter multidisciplinar, interligando diferentes áreas e conhecimentos:

- arquitetura e organização de computadores;
- arquitetura de conjuntos de instruções;
- circuitos eletrônicos;
- otimização de *hardware*;
- aprendizado de máquina.

A importância das quatro primeiras áreas foi estabelecida ao longo desta introdução. O aprendizado de máquina insere-se no contexto de *co-design*, os algoritmos das redes neurais artificiais contém propriedades particularmente desafiadoras para *Central Processing Units* (Unidades centrais de processamento) (CPUs) tradicionais.

1.3 Metodologia e Objetivos

O objetivo deste trabalho consiste em investigar e discutir a abordagem de *hardware-software co-design* enquanto controle do *trade-off* entre desempenho computacional e complexidade de implementação, aumentando o primeiro sem impactar significativamente o segundo.

Partimos da análise estática e dinâmica dos algoritmos associados ao aprendizado de máquina, especificamente as redes neurais artificiais, examinando os *bottlenecks* causados por uma implementação genérica de processador e investigando a viabilidade de adaptar esta implementação, a fim de eliminá-los. Essa adaptação está relacionada tão somente à modificação da organização interna do processador (microarquitetura).

A metodologia proposta se insere no âmbito de descrição no nível de transferência entre registradores (*Register-Transfer Level*) (RTL). Os circuitos digitais resultantes são posteriormente simulados via *software* ou implantados em *Programmable Logic Devices* (Dispositivo lógico programável) (PLDs), particularmente em *Field-Programmable Gate Array* (Matriz de portas programáveis em campo) (FPGA).

Nesse contexto, nos concentramos em ferramentas e conjuntos de instruções abertos. Inicialmente, a vasta disponibilidade de informações acelera o desenvolvimento da pesquisa. Ademais, as licenças abertas promovem a difusão e o aprimoramento da proposta e do conhecimento, bem como facilitam a reprodução dos resultados obtidos.

As questões que guiam este estudo são:

- Quais as condições para que a abordagem de *hardware-software co-design* seja viável?
- Em quais cenários ela consegue apresentar benefícios em relação a uma abordagem tradicional, com otimizações para o caso geral ou uma ampla gama de aplicações?

- Quais os ganhos de desempenho computacional obtidos por meio da aplicação dessa abordagem?

Dado o caráter da abordagem de *co-design*, para validar a proposta verificamos experimentalmente a variação no desempenho do algoritmo, medido através da quantidade de instruções executados por segundo.

É essencial ressaltar que a implementação proposta nesta pesquisa tem enfoque em sistema *embedded-class*, como, por exemplo, microcontroladores. Processadores *client-class*, usados em *desktops* e portáteis, ou *server-class*, empregados em grandes centros de dados, não fazem parte do escopo. Como discutido na subseção 1.2, esses circuitos possuem da ordem de bilhões de transistores (veja a Figura 1.1) e são desenvolvidos por centenas de profissionais ao longo de diversos anos.

Deve-se também considerar que os resultados deste estudo estão diretamente relacionados com as escolhas de algoritmos e, principalmente, de projeto da microarquitetura. Como efeito, os mecanismos específicos desenvolvidos não somente limitam a aplicabilidade em outros arranjos, como esperado pelo próprio princípio de *co-design*, mas também representam somente uma das possíveis configurações de solução.

Por fim, ambicionamos que este trabalho seja reprodutível. Assim, é necessário que as ferramentas utilizadas sejam acessíveis e estejam amplamente disponíveis. Por este motivo, fizemos uso extensivo de *software* livre, recorrendo a *software* comercial tão somente quando uma alternativa aberta fosse inexistente ou insuficiente, a exemplo *software* de síntese lógica e física do FPGA.

1.4 Organização do Trabalho

Este trabalho está organizado nos seguinte quatro capítulos:

No Capítulo 2 introduzimos os conceitos teóricos relativos à arquitetura e organização de computadores, arquitetura de conjunto de instruções e redes neurais artificiais.

No Capítulo 3 introduzimos a metodologia proposta apresentando o NINA, processador projetado aplicando *co-design*, a aplicação para qual ele foi otimizado e os resultados experimentais.

No Capítulo 4 retomamos as questões de pesquisa em face dos resultados. Também discutimos trabalhos futuros de pesquisa.

O Apêndice A contém o diagrama de *datapath* do NINA.

Capítulo 2

Conceitos

Neste capítulo revisaremos conceitos fundamentais no que diz respeito à arquitetura e organização de computadores, construindo o entendimento de microarquitetura, arquitetura do conjunto de instruções e paralelismo. Ao longo do capítulo serão apresentados os obstáculos e *trade-offs* de desempenho, bem como as principais soluções. Introduziremos a ideia de *throughput* (rendimento) e sua relação com a execução condicional de código. Concluímos o capítulo mostrando as características dos algoritmos de redes neurais artificiais e por que eles representam um desafio no *trade-off* entre o bom rendimento e complexidade de processadores tradicionais.

2.1 Arquiteturas do Conjunto de Instruções

Na transmissão de informações entre duas ou mais entidades, é necessário haver um sistema de comunicação estruturado e aceito por todas as partes. Na computação, as linguagens de programação são o meio pelo qual expressamos as informações que serão executadas pelos computadores. No entanto, é comum que tais linguagens lancem mão de variados níveis de abstrações, dando ao programador maiores graus de liberdade no modo como se comunica. Processadores, por outro lado, conseguem interpretar apenas um pequeno conjunto instruções.

A descrição das instruções que podem ser executadas por um processador, do ponto de vista do programador, é chamada de *arquitetura*. Mais especificamente, chamamos de arquitetura do conjunto de instruções (*Instruction-Set Architecture*) a linguagem entre *hardware* e *software*.

Assim como com linguagens de programação, existem várias arquiteturas do conjunto de instruções, como, por exemplo, 80x86, ARMv9 e RISC-V. Nesta seção, consideram-se as propriedades como definidas pela arquitetura RISC-V.

2.1.1 RISC-V

RISC-V é um *instruction-set architecture* aberto concebido originalmente como um ISA que apoiasse a pesquisa de arquitetura de computadores e, consequentemente, o desenvol-

vimento de novos processadores, mas também servisse de instrumento educacional. Ao longo do tempo, foi escolhido em diversos projetos comerciais. Apenas no ano de 2024, mais de dois bilhões de dispositivos usando núcleos RISC-V foram produzidos (RISC-V International, 2024).

Embora seja comum referir-se *à* arquitetura RISC-V, o termo trata de uma família de arquiteturas. Há quatro ISAs base, dos quais ao menos um deve estar presente em qualquer implementação, e são exclusivos, opcionalmente acrescidos de extensões. Cada ISA base é diferenciado pelo número de registradores e seu tamanho, além do espaço de endereços de memória disponível. Deste modo, diferentes bases atendem à diferentes classes de processadores, como microcontroladores e processadores *server-class* (Waterman e Asanović, 2019). Nesta pesquisa, consideraremos somente a base inteira de 32-*bit*, RV32I.

Cada base contém apenas um conjunto mínimo, contudo funcional, de instruções, seguindo os preceitos que dão nome ao ISA (Waterman e Asanović, 2019). Ainda que não haja uma definição formal, um *Reduced Instruction Set Computer* (Computador com conjunto reduzido de instruções) (RISC) é uma arquitetura cujo conjunto de instruções é composto por instruções simples, com uma única ação bem definida (Patterson, 1985). Ao todo, a RV32I contém 40 instruções.

2.1.2 Operações, Operandos e Formatos de Instruções

Instruções representam operações executadas por processadores. Deste modo, cada instrução deve dedicar alguns *bits* para definir sua operação ou a classe de operações que ela representa. A RV32I dedica os sete primeiros *bits* de cada instrução para este propósito, que recebem o nome de *opcode*. Dez *opcodes* são usados (veja a Tabela 2.1), que incluem operações lógico-aritméticas, de leitura e escrita dos registradores, desvios, e operações especiais. A seleção da operação pertencente a uma classe é feita por campos adicionais.

6	5	4	3	2	1	0	
0	0	1	0	0	1	1	Lógico-Aritméticas (Registrador)
0	1	1	0	0	1	1	Lógico-Aritméticas (Constante)
0	0	0	0	0	1	1	*Leituras*
0	1	0	0	0	1	1	*Escritas*
1	1	0	0	0	1	1	Desvios Condicionais
0	1	1	0	0	1	1	Desvio Não Condicional
1	1	0	1	1	1	1	Desvio Não Condicional (Reg.)
1	1	0	0	1	1	1	Carrega Constante (Registrador)
0	1	1	0	0	1	1	Carrega Constante (Contador)
1	1	1	0	0	1	1	*Environment Call/Break*

Tabela 2.1: *Na arquitetura RISC-V, os sete primeiros bits são dedicados para determinar a operação ou classe de operações, da instrução. Esse agrupamento simplifica a decodificação. A arquitetura base de 32-bit, RV32I, contempla um total de 40 instruções, agrupadas em dez opcodes. Os demais opcodes, que não estão representados, ficam disponíveis para extensões à RV32I.*

O campo *funct3*, seleciona o tipo de operação lógico-aritmética, por exemplo, adição ou disjunção exclusiva (*XOR*). Adicionalmente, o campo *funct7* determina a variação de uma

mesma operação, como, por exemplo, a subtração e deslocamento aritmético à direita, variações da adição e do deslocamento lógico à direita, respectivamente. Seus nomes referem-se ao fato de ocupar três e sete *bits*, respectivamente.

Como uma arquitetura *load-store*, há distinção entre instruções que fazem acesso à memória, de leitura e escrita, e instruções que operam sobre registradores. Consequentemente, não há instruções que operem diretamente sobre a memória. Operandos devem ser registradores ou valores constantes estipulados em tempo de compilação. Cada instrução contém até três operandos registradores, sendo um *rd* onde será escrito o resultado da operação, e dois registradores de entrada *rs1* e *rs2*, respectivamente.

Em arquiteturas do conjunto de instruções, o formato da instrução depende da combinação de campos necessários. A formatação pode ser realizada por meio de duas estratégias.

A primeira opção é adotar um tamanho de instrução variável. O tamanho da instrução é determinado, portanto, conforme os campos que a compõem. Essa opção tem potencial de aumentar a densidade de código, reduzindo o espaço de armazenamento utilizado pelos programas. Por outro lado, a decodificação da instrução pelo processador se torna mais complexa. A segunda opção para acomodar os requisitos de instruções é utilizar instruções de tamanho fixo e determinar campos comuns às diferentes classes de instruções. Essa abordagem é adotada pela RISC-V.

A base RV32I estabelece seis formatos de instruções, denominados R, I, S, B, U e J (veja a Tabela 2.2). Uma operação de adição, por exemplo, emprega o formato R, que dispõe dos dois operandos de entrada, um de saída e campos de especificação de operação. Já uma operação de escrita utiliza o formato I, que dispõe de apenas um operando de entrada, o registrador contendo o endereço base da escrita, um registrador de saída e um valor constante de deslocamento em relação ao endereço base. Na Seção 2.2, discutimos as implicações desse agrupamento no estágio de decodificação do processador.

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	Formato R
imm[11:0]		rs1	funct3	rd	opcode	Formato I
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	Formato S
imm[12 | 10:5]	rs2	rs1	funct3	imm[4:1 | 11]	opcode	Formato B
imm[31:12]				rd	opcode	Formato U
imm[20 | 10:1 | 11 | 19:12]				rd	opcode	Formato J

Tabela 2.2: *As quatro arquiteturas-base que estabelecem a família RISC-V possuem instruções com tamanho em bits fixos. Como a composição de campos necessários, como os operandos e outras informações adicionais, variam com cada operação, os projetistas buscam encontrar similaridades entre instruções e reduzir o número de formatos, simplificando a decodificação. Na base de 32-bit, RV32I, existem seis formatos de instruções.*

2.2 Microarquitetura de Processadores

Um processador é um componente que recebe uma sequência de instruções e executa as operações por elas estabelecidas. Na Seção 2.1 apresentamos o conceito de arquitetura, a descrição do sistema computacional e de suas capacidade do ponto de vista do programador. Reciprocamente, uma microarquitetura é a implementação de uma arquitetura. Nesta seção,

revisaremos a organização interna de processadores, discutindo algumas abordagens e *trade-offs* de desempenho.

O modelo de von Neumann descreve as características organizacionais essenciais de unidades de processamentos. Ele especifica as seguintes unidades funcionais:

- **unidade lógica e aritmética:** executa as operações;
- **unidade de controle:** mantém e controla o estado interno;
- **memória** para armazenar dados e instruções;
- **mecanismos de entrada/saída**.

Embora o modelo proposto por von Neumann continue sendo o principal paradigma computacional, o armazenamento conjunto de instruções e dados provou-se um fator limitante, conhecido como *von Neumann bottleneck* (Gargalo de von Neumann) (Backus, 1978). Em resposta, processadores modernos adotam o modelo Harvard, com memórias separadas (veja a Figura 2.1).

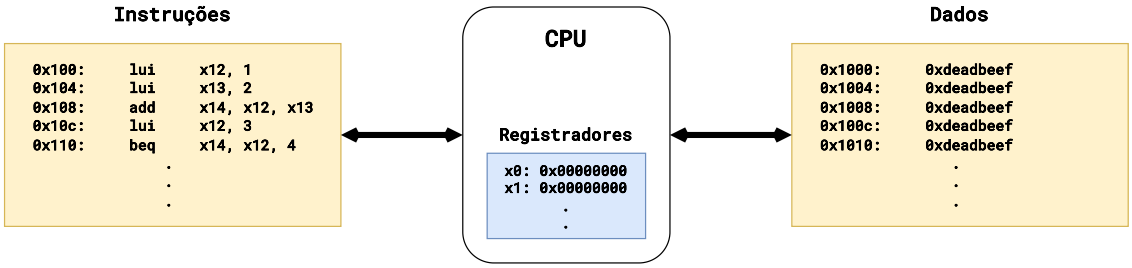

Figura 2.1: *Um processador decodifica uma sequência de instruções, realizando as operações indicadas. Em uma arquitetura load-store, como a RISC-V, o processador só tem acesso à memória através das instruções de load e store. Os dados da memória devem ser lidos e escritos em registradores antes de serem utilizados.*

O modelo também consolida algumas convenções entre programadores e projetistas de *hardware*:

1. um programa deve ser constituído por uma sequência de instruções válidas;
2. cada instrução é executada na ordem especificada pelo programa;
3. instruções são executadas uma por vez antes de passar para próxima. instrução

Para garantir a execução sequencial, surgem os dois últimos componentes primordiais das microarquiteturas: o *Program Counter* (Contador do programa) (PC) e o *Instruction Register* (Registrador de instrução) (IR). O primeiro registrador mantém o endereço da instrução sendo executada, enquanto o segundo mantém a instrução em si.

O fluxo de execução se inicia, portanto, no *program counter*, transmitindo o endereço nele contido para a memória de instruções (*Instruction Memory*) (IMEM), que lê a instrução e coloca os *bits* em sua saída. A instrução é então transmitida ao *instruction register* e também à unidade de decodificação. Identificada a operação, a unidade de decodificação

decompõe os campos conforme o formato, acessa os registradores necessários e repassa campos adicionais para suas respectivas unidades.

As etapas posteriores à decodificação variam conforme a operação. Operações de leitura e escrita, por exemplo, irão acessar a memória de dados (*Data Memory*) (DMEM), enquanto operações aritméticas não. Na Seção 2.1 apresentamos os seis formatos de instruções da arquitetura RV32I. Waterman e Asanović tinham como objetivo simplificar a implementação dos processadores com o reaproveitamento das unidades funcionais para a maior quantidade de instruções quanto possível (Waterman e Asanović, 2019). Deste modo, uma microarquitetura em conformidade com a RV32I contém um número fixo de campos para decodificar, enquanto uma instrução de acesso à memória, por exemplo, que não realiza operações lógico-aritméticas, consegue usar a *Arithmetic Logic Unit* (Unidade lógica e aritmética) (ALU) para calcular o endereço da memória que será acessado (Patterson e Hennessy, 2017).

A Figura 2.2 apresenta o diagrama em alto nível de microarquitetura com as unidades funcionais primordiais. A coleção dessas unidades, juntamente das ligações entre elas, é chamado *datapath* (caminho dos dados).

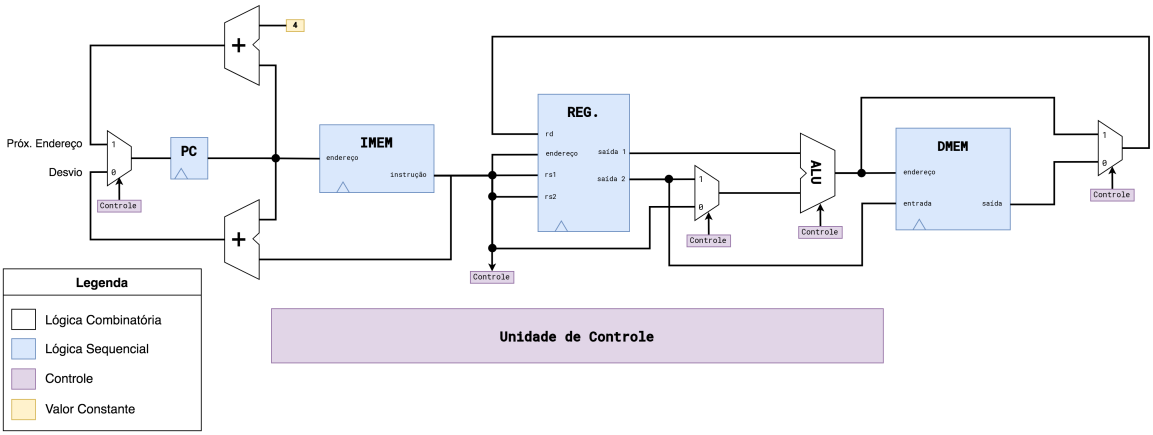

Figura 2.2: *A organização interna de um processador é formada por diversas unidades funcionais. Cada unidade desempenha função distinta na execução das operações. O início dessa execução ocorre com o endereço da instrução escrito no program counter (Contador do programa) (PC). O endereço é transmitido à memória de instruções (IMEM) e lida. A instrução é então decodificada, os registradores apropriados são lidos do banco de registradores, e os dados são transmitidos para as unidades pertinentes, como a arithmetic logic unit (Unidade Lógica e Aritmética) (ALU) e a memória de dados (DMEM). Finalmente, o endereço da próxima instrução é escrito no program counter e o processo se reinicia.*

Ao nível de implementação, algumas unidades funcionais dependem exclusivamente de sua entrada atual para gerar a saída correspondente. Uma ALU, por exemplo, produz o resultado de uma adição em sua saída ao conter os valores correspondentes em sua entrada. No entanto, certas unidades dependem também da manutenção de um estado. A memória de dados deve persistir o valor colocado em sua entrada ao longo do tempo, até que um novo valor seja sobrescrito. Outras unidades seguem similarmente. Chamamos os elementos lógicos que não dependem de estado de combinatórios e aqueles que dependem de sequenciais. As mudanças de estado em elementos lógicos sequenciais são controladas por um sinal de *clock* (relógio), ocorrendo em sincronia com o sinal.

Na prática, componentes lógicos incorrem em atrasos de propagação dos sinais pelo circuito. Deste modo, a frequência máxima para sinais de *clock* que atuam sobre elementos sequenciais são restritas pelo tempo necessário para que o sinal se propague por todos os elementos necessários para a execução da instrução. Este fato tem implicações no desempenho do processador, que é diretamente proporcional à frequência de operação.

2.2.1 Métricas de Desempenho

Mensurar e avaliar comparativamente a *performance* de processadores é um aspecto fundamental na escolha de *trade-offs* por parte dos arquitetos de computadores (Hennessy e Patterson, 2018). No entanto, quantificá-la é um problema complexo, conforme abordado na literatura (Gray, 1992; J. Dongarra *et al.*, 1987; Bard, 1973)

A abordagem mais eficaz na avaliação de desempenho consiste em analisar o tempo decorrido na execução de aplicação específica. Essa abordagem, no entanto, pode ser infactível sob diversos aspectos. Por um lado, pode não haver mais de uma aplicação que se deseje executar. Um computador pessoal, por exemplo, executará uma série de aplicações diferentes, dependendo de seu usuário. Por outro lado, durante a etapa de projeto, um arquiteto de computadores deve recorrer ao uso de simulações até que o processador seja finalizado. Tais simulações têm velocidades ordens de magnitude mais lentas que a execução real e, frequentemente, não dispõe da capacidade de executar aplicações de alto nível e sistemas operacionais completos.

Benchmarks sintéticos surgiram como uma possível solução para este problema. São algoritmos compactos, porém cuja composição de instruções sejam representativas das tarefas que aplicações usualmente realizam. Alguns exemplos de *benchmarks* são: Linpack, Dhrystone e SPEC.

O primeiro foi desenvolvido em 1976, por Jack Dongarra, e consistia em uma coleção de algoritmos de álgebra linear operados em uma matriz 100 por 100. Ele caracterizava as aplicações de computação científica que dominavam os *mainframes* do período (J. J. Dongarra, 1983).

O segundo, escrito por Reinhold Weicker em 1984, foi projetado para medir o desempenho de processadores com arquiteturas simples e era majoritariamente composto por funções de manipulação de texto (Gray, 1992). Ambos tinham como unidade de medida a quantidade de instruções executadas por segundo.

Em função da evolução dos sistemas computacionais, estes *benchmarks* tornaram-se obsoletos. Por um lado, eles não eram mais representativos da gama de tarefas para as quais os computadores eram empregados. Por outro, o surgimento e o aperfeiçoamento de arquiteturas não era explorado pela combinação de instruções empregadas nesses *benchmarks*. Essa discrepância era exacerbada pela métrica de quantidade de instruções executadas, que não capturava características como instruções dedicadas a operações específicas.

Em 1988 o *Standard Performance Evaluation Corporation* (SPEC), um consórcio fundado por 22 fornecedores de sistemas computacionais, surgiu visando padronizar avaliações de desempenho. O consórcio introduziu um conjunto de aplicações variadas, que estressavam

múltiplos componentes, para serem usados como *benchmarks*. Para os resultados serem reconhecidos, era necessário que as bases de código fossem otimizadas para as arquiteturas de cada processador onde a avaliação fosse conduzida, garantindo a uniformidade no aproveitamento de recursos exclusivos. Os resultados eram expressos como uma pontuação individual por aplicação, os SPECratios, calibrados usando como valor referencial os resultados do DEC VAX 780 (Gray, 1992).

Embora a abordagem de escore apresentada pelo SPEC tenha se mostrado mais eficiente na comparação de processadores com diferentes arquiteturas, para arquitetos de computadores a comparação de instruções executadas por segundo representa uma maneira simplificada, porém, eficiente de avaliar o rendimento da microarquitetura. O *throughput* é, portanto, a principal métrica empregada no projeto de processadores.

A equação 2.1 mostra a definição de *throughput*:

$$\text{Throughput} = \frac{\text{Frequência}}{\text{Ciclos por Instrução}} \tag{2.1}$$

onde o termo Ciclos por Instrução (CPI) é a quantidade de instruções completadas por ciclo de *clock* na execução da aplicação de referência. Matematicamente, o CPI é dado pela equação 2.2:

$$\text{CPI} = \frac{\text{Ciclos de } \textit{Clock}}{\text{Instruções Executadas}} \tag{2.2}$$

Busca-se, portanto, aumentar frequência de operação e diminuir o CPI. Todavia, a redução do número de ciclos por instrução está diretamente relacionado à quantidade de recursos que o processador dispõe. Porém, adicionar elementos lógicos incorre em maiores tempos de propagação dos sinais, afetando negativamente a frequência. Nesse sentido, existem várias abordagens de organização do *datapath* com objetivo de controlar o *trade-off* entre frequência, ciclos por instrução e complexidade de implementação.

2.2.2 Processadores de Ciclo Único

A primeira e mais elementar abordagem de implementação do *datapath* é a de ciclo único. Neste caso, todas as instruções serão executadas em somente um ciclo do sinal de *clock*. Para isso, alguns requisitos devem ser satisfeitos:

- as memórias de instruções e dados devem ser obrigatoriamente separadas;
- as memórias e o banco de registradores precisam conter duas portas, para leitura e escrita no mesmo ciclo;
- as memórias e o banco de registradores devem ser construídos com lógica combinatória.

O primeiro é necessário, pois operações de leitura e escrita precisam fazê-la no mesmo ciclo no qual a instrução é lida. A segunda deve-se ao fato de que a maioria das instruções lê registradores e escreve os resultados na mesma operação. O mesmo é válido para a memória de instruções. Por fim, circuitos sequenciais de memória e registradores levam

mais de um ciclo de *clock* para gerar suas saídas. A unidade de memória, por exemplo, pode exigir um ciclo para registrar o endereço e um segundo ciclo para a saída dos dados, impedindo o correto funcionamento dessa abordagem.

A frequência máxima será determinada pelo tempo exigido para o sinal sair do *program counter* e se propagar até a última unidade usada pela instrução. Instruções simples terão caminhos de propagação curtos, enquanto instruções complexas terão caminhos mais longos e, portanto, demorados. Entretanto, todas as instruções têm que ser executadas em um único ciclo. Logo, a frequência máxima será resultado do tempo gasto pela propagação do maior caminho, aquele da instrução mais complexa.

Em virtude dessas limitações, microarquiteturas de ciclo único são muito ineficientes e pouco aplicadas. Ainda que a quantidade de *Cycles per Instruction* (Ciclos por instrução) (CPI) seja um, o *trade-off* relativo à limitação da frequência máxima muitas vezes torna o *throughput* muito inferior às outras abordagens. Essa abordagem é vantajosa apenas para arquiteturas do conjunto de instruções reduzidas, principalmente aquelas onde não há instruções complexas como multiplicação e divisão ou suporte a ponto flutuante (Patterson e Hennessy, 2017).

2.2.3 Processadores Multiciclo

A abordagem de ciclo único tem como maior desvantagem a dificuldade em obter altas frequências de *clock*. Uma alternativa para aliviar esse problema é a de implementações multiciclo. A execução de operações é naturalmente composta por diversas subtarefas, como acessar o banco de registradores ou acessar a memória. Essa abordagem aproveita tal propriedade para reduzir o atraso de propagação. A cada ciclo de *clock* uma sub tarefa é executada. Ao final de cada ciclo, registradores guardam o estado dos elementos para a próxima etapa.

Em comparação com implementações de ciclo único, o *datapath* de microarquiteturas multiciclo apresenta as seguintes diferenças:

- as memórias de instruções e dados podem ser compartilhadas em uma única unidade funcional;
- um ou mais registradores são adicionados às saídas de cada unidade funcional.

além disso, empregando multiplexadores adicionais, é possível reutilizar unidades funcionais para mais tarefas. A Figura 2.3 apresenta, em alto nível, o diagrama de *datapath* multiciclo.

É importante ressaltar que, nessa abordagem, somente uma unidade funcional está executando a instrução por vez. A frequência de *clock* máxima é dada pelo tempo de propagação da unidade mais lenta, que será, obrigatoriamente, uma fração daquela obtida com ciclo único. Contudo, a quantidade de ciclos gastos por instrução será maior do que um.

Cada classe de instrução contém certa quantidade de subtarefas e, consequentemente, ciclos de *clock* necessário para sua finalização. O CPI de processadores multiciclo é dado pela média dos ciclos necessários, ponderado pela proporção de cada classe executada, variando segundo a aplicação. Uma microarquitetura e aplicação com as propriedades

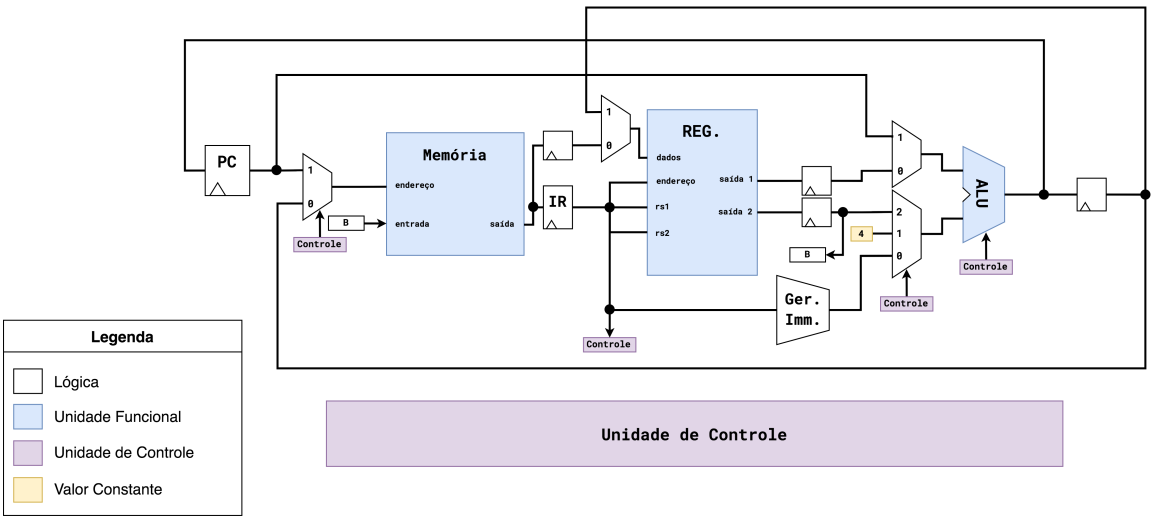

Figura 2.3: *Uma microarquitetura multiciclo implementa menos unidades funcionais comparativamente com a contrapartida de ciclo único. Essa abordagem explora às subtarefas comuns às instruções para compartilhar recursos. A Unidade Lógica e Aritmética, por exemplo, pode em calcular a soma de dois valores em um ciclo e o endereço da próxima instrução em outro. Para manter o estado entre ciclos, são adicionados registradores às saídas de cada unidade. O Instruction Register (IR) guarda a instrução que está sendo executada. Outra vantagem é a possibilidade de utilizar uma memória unificada para dados e instruções, seguindo o modelo de von Neumann. O diagrama demonstra uma visão de alto nível de uma implementação multiciclo.*

apresentadas na Tabela 2.3 terá CPI dado por (Patterson e Hennessy, 2017):

$$\begin{aligned} \text{CPI} &= \frac{\text{Ciclos de } \textit{Clock}}{\text{Instruções Executadas}} \\ &= \frac{\sum \text{Instruções}_i \times \text{CPI}_i}{\text{Instruções Executadas}} \\ &= \sum \frac{\text{Instruções}_i}{\text{Instruções Executadas}} \times \text{CPI}_i \end{aligned} \tag{2.3}$$

onde Instruções_i é a quantidade de instruções da i-ésima classe executada e CPI_i o respectivo CPI. As razões representam a proporção de cada classe no código. Logo, podemos substituir e obter:

$$\begin{aligned} \text{CPI} &= \sum p_i \times \text{CPI}_i \\ &= (4 \times 0,6) + (5 \times 0,2) + (4 \times 0,05) + (3 \times 0,15) \\ &= 2,4 + 1 + 0,2 + 0,45 \\ &= 4,05 \end{aligned} \tag{2.4}$$

onde p_i é a proporção de instruções executadas da classe i.

Ganhos de *throughput* baseiam-se no balanço entre o CPI por classe de instruções e a divisão eficiente do *datapath*. Organizações de *datapath* desproporcionais, com certas unidades funcionais incorrendo em atrasos muito superiores a outras, atenuam os ganhos em frequência de *clock* máxima. Na prática, a complexidade em implementar boas divisões

Classe	Ciclos	Proporção
Lógico Aritmética	4	60%
Leitura	5	20%
Escrita	4	5%
Desvio	3	15%

Tabela 2.3: *Ciclos necessários por classe de instruções e suas respectivas proporções em base de códigos hipotética.*

tornam essa abordagem pouco vantajosa.

2.2.4 Pipelining

Na abordagem multiciclo, somente uma unidade funcional fica ocupada por vez. Essa restrição determina um subaproveitamento de recursos computacionais, pois outras unidades poderiam ser ocupadas sem prejuízo. *Pipelining* (sobreposição) é uma técnica em que múltiplas instruções são executadas paralelamente por um mesmo processador sem a necessidade de aumento de recursos.

Em uma microarquitetura com *pipelining*, as instruções são executadas como em uma linha de montagem. Suponha que a execução de uma instrução envolva as seguintes etapas:

1. buscar a instrução na memória;
2. executar a instrução;
3. guardar os resultados.

Em um esquema de linha de montagem, a cada ciclo de *clock* uma nova instrução entrará na linha. Deste modo, no primeiro ciclo a instrução A entrará na etapa um, sendo acessada na memória. No segundo ciclo, ela passa para a etapa dois e é executada, enquanto uma nova instrução, B, entra na primeira etapa. No terceiro ciclo, a instrução A vai para a etapa três e tem seus resultados guardados. A instrução B é executada e uma nova instrução C entra na etapa um.

Essa abordagem é muito mais rápida em comparação com a de multiciclo, pois assim que a instrução termina de ser buscada na memória, uma próxima instrução é carregada. Além disso, essa técnica permite o pleno aproveitamento dos recursos do processador.

Arquiteturas RISC, tradicionalmente, decompõem o *pipeline* em cinco etapas, denominados estágios:

1. **Instruction Fetch (IF):** o endereço contido no *program counter* é transmitido para a memória de instruções, que lê a respectiva instrução;
2. **Instruction Decode (ID):** decodifica a instrução e lê os registradores apropriados;
3. **Execute (EX):** executa a operação ou calcula o endereço da memória/próxima instrução;
4. **Memory Access (MEM):** acessa a memória de dados, se necessários;

5. **Write-back (WB):** guarda os resultados no registrador de destino, se necessário.

A Figura 2.4 apresenta o diagrama de evolução das instruções ao longo dos ciclos. O melhor aproveitamento ocorre nos ciclos cinco e seis, quando todas as unidades funcionais estão ocupadas.

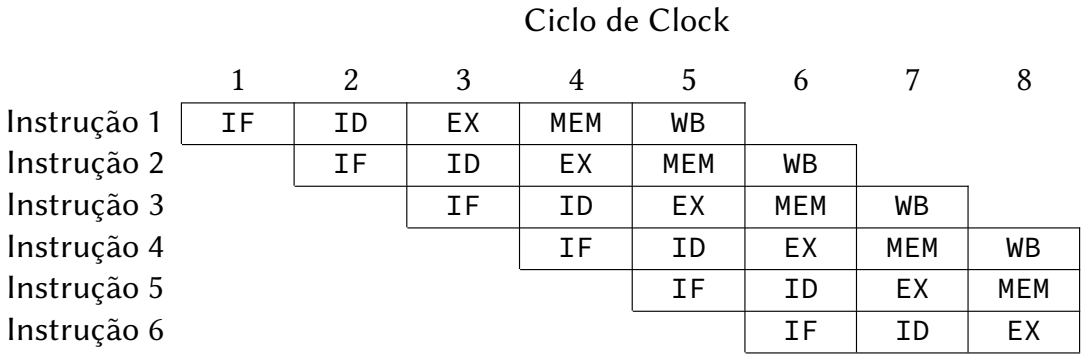

Figura 2.4: *Em uma microarquitetura com pipelining, a execução das instruções é sobreposta, de modo que cada uma use um subconjunto de unidades funcionais, denominados estágios. A cada ciclo de clock uma instrução entra no pipeline e começa sua execução no primeiro estágio, correspondente ao IF no diagrama, prosseguindo nos próximos ciclos até a completude. No quinto ciclo o processador tem aproveitamento máximo, pois todas as unidades estão ocupadas. Essa técnica aumenta o throughput em relação às microarquiteturas multiciclo.*

Embora *pipelining* não diminua o tempo de cada etapa e, portanto, não interfira na frequência máxima em relação ao processador multiciclo, essa técnica aumenta efetivamente o *throughput*, explorando *Instruction-Level Parallelism* (Paralelismo em nível de instrução) (ILP), o potencial de sobreposição de certas instruções. Assumindo que todos os estágios tenham tempos similares, o *speedup* potencial com uso de *pipelining* é igual ao número de estágios, caso todos estejam ocupados. Essa última condição nem sempre será satisfeita. Existem situações onde a próxima instrução não pode ser executada até que outra, predecessora, seja completada. Essas situações são chamadas *pipeline hazards* (conflitos).

2.2.5 Pipeline Hazards

No modelo de execução sequencial, instruções possuem dependências relativas à sua ordem. Os *pipeline hazards* são eventos que impedem a execução sobreposta, em decorrência de conflitos por dependência. *Hazards* são categorizados conforme a causa do conflito, em três grupos:

- conflitos estruturais;
- dependências de dados;
- dependências de controle.

A seguir, são discutidas as causas, bem como possíveis soluções para esses conflitos.

Conflitos Estruturais

O primeiro tipo de conflito são os *structural hazards* (conflitos estruturais). Eles ocorrem quando não há recursos computacionais suficientes no processador para sobrepor ambas as instruções e são comumente caracterizados por dois acessos simultâneos à memória. Em microarquiteturas com memórias de instruções e dados unificadas (veja a Figura 2.3), conflitos estruturais ocorrem sempre que uma instrução de leitura ou escrita atinge o estágio MEM, por ser necessário acessar a memória de dados ao mesmo tempo em que se lê a próxima instrução no estágio IF.

Ao nível de implementação, *structural hazards* podem ser evitados por meio da redundância das unidades funcionais e outros recursos. No entanto, é possível mitigar ou eliminá-los ao nível arquitetural. A arquitetura RISC-V foi projetada de modo que seja fácil evitar esse tipo de conflito (Waterman e Asanović, 2019; Patterson e Hennessy, 2017).

Dependências de Dados

Os *data hazards* (conflitos de dados) ocorrem quando uma instrução depende da conclusão de outra instrução mais a frente no *pipeline*. Como exemplo, a sequência de instruções a seguir possui dependência de dados:

```
addi    x1, x0, 1
addi    x2, x0, 2
sub     x3, x1, x2
```

O resultado da terceira instrução, subtrair o valor presente nos registradores `x1` e `x2`, necessita que primeiro sejam calculadas as somas nas duas instruções anteriores. A Figura 2.5 apresenta o diagrama do *pipeline* com conflito. Nota-se que a terceira instrução entra em estágio ID, quando os registradores são acessados, enquanto a anterior continua sendo executada e, portanto, não atualizou o banco de registradores.

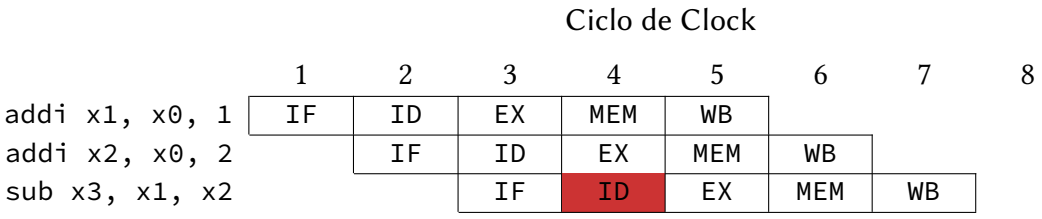

Figura 2.5: *Dependências de dados ocorrem quando o operando de uma instrução depende do resultado de outra instrução que continua no pipeline. A instrução de subtração depende dos resultados das duas adições anteriores, porém no momento em que ela atinge o estágio de decodificação e, consequentemente, de acesso ao banco de registradores, os valores dos operandos ainda não foram atualizados, pois as outras instruções encontram-se nos estágio EX e MEM, respectivamente.*

Existem duas possíveis soluções para *data hazards*: *stall* (parada) e *forwarding* (adiantamento).

A abordagem de *stall* consiste em parar o avanço da instrução dependente até que o conflito tenha sido resolvido, isto é, a dependência termine sua operação (veja a Figura 2.6).

Contudo, essa técnica aumenta o CPI, já que são gastos ciclos de *clock* onde unidades funcionais estão ociosas, afetando negativamente o *throughput*. Além disso, é preciso adicionar elementos lógicos capazes de identificar os conflitos.

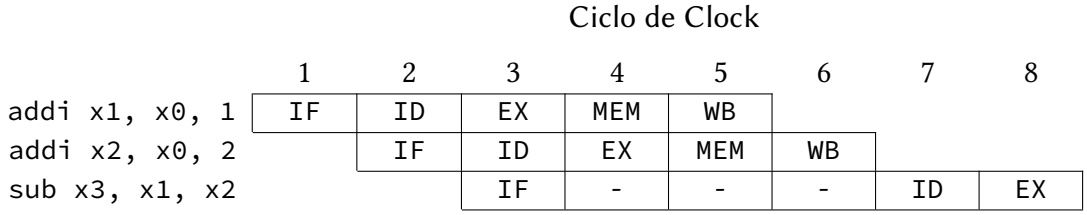

Figura 2.6: *Stalls impedem que instruções com dependências não resolvidas avancem no pipeline. A instrução de subtração, que depende da execução completa das duas anteriores, não entra no estágio ID até que a outras tenham sido completadas. Essa abordagem diminui o aproveitamento do processador, pois durante três ciclos existem unidades ociosas.*

Por outro lado, a abordagem de *forwarding* resolve os conflitos enquanto mitiga ou elimina o uso de *stalls*. Para isso, os dados são transmitidos da saída de um unidade funcional para a entrada de outra, antes que os registradores sejam atualizados. A Figura 2.7 apresenta a execução da sequência de instruções com *forwarding*. O resultado da Unidade Lógica e Aritmética, no estágio EX, é transmitido para a sua entrada no próximo ciclo. Desse modo, os valores carregados dos registradores no estágio ID, que ainda não refletem a instrução *addi x2, x0, 2*, serão substituídos pelos valores corretos.

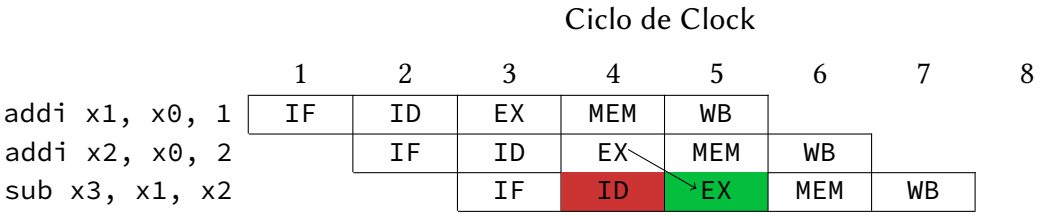

Figura 2.7: *O forwarding consiste em transmitir os dados de uma unidade funcional para outra cuja instrução necessite destes. Isso permite que instruções com dependências sejam executadas sem a necessidade de stalls. A subtração, que depende do resultado da anterior, recebe-o assim que a operação é executada, antes de atingir o último estágio, quando o banco de registradores é atualizado.*

Embora o adiantamento de dados atenue os efeitos dos *stalls* na quantidade de ciclos por instrução, o atraso de propagação do sinal entre a saída e entrada afeta negativamente a frequência máxima de *clock*. Ainda assim, essa abordagem é amplamente adotada e preferida em relação aos *stalls*.

Dependências de Controle

Dependências de controle ocorrem quando há desvios condicionais (*branches*). As operações de desvio condicional determinam a escolha da próxima instrução a ser executada. Com *pipelining*, o processador deve buscar uma nova instrução na memória a cada ciclo de *clock*. Deste modo, sempre que um *branch* entra no *pipeline*, é necessário que ele seja decidido, para a instrução seguinte ser determinada e lida no próximo ciclo. Por outro lado, o resultado do teste relacional entre dois operandos, que decide o desvio, só estará disponível após a execução da comparação, geralmente feita no estágio EX.

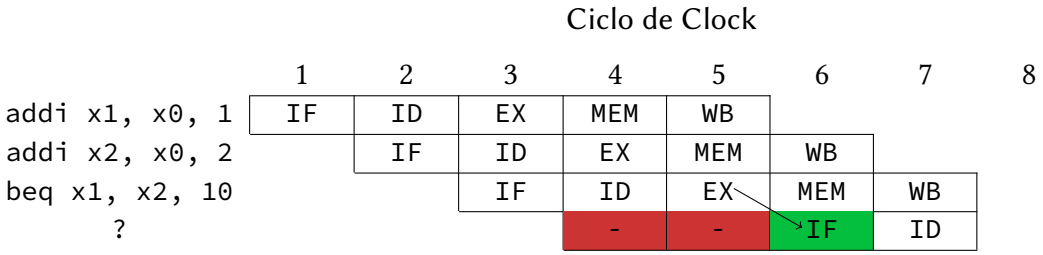

Figura 2.8: *Dependências de controle ocorrem quando há branches. Desvios condicionais são não determinísticos e, portanto, só são decididos após a execução. Assim, mesmo com uso de forwarding, é necessário o uso stalls até que o operando necessário esteja disponível.*

O impasse causado por dependências de controle é um dos maiores desafios de arquitetos em termos de *throughput* (Hennessy e Patterson, 2017). Dado que *branches* são não determinísticos, é necessário o uso de *stalls*, mesmo com *forwarding* (veja a Figura 2.8). Como discutido na Seção 2.2.5, essa abordagem é custosa ao desempenho do processador, sobretudo em *pipelines* que possuam muitos estágios.

Ainda que desvios condicionais não possam ser decididos antes de sua execução, é factível predizê-los. Existem várias estratégias de predição de desvios (*branch prediction*). A abordagem mais simples baseia-se na escolha de um resultado fixo, seja tomar ou não o desvio. Para certas tarefas, tais como laços, decisões constantes representam um *trade-off* razoável entre complexidade e eficácia.

No entanto, para atingir desempenho superior, abordagens mais sofisticadas são necessárias. A maioria das estratégias nesse sentido envolve a análise estatística dos desvios, isto é, guardam informações sobre as decisões tomadas anteriormente e as utilizam na predição.

A importância da implementação de bons *branch predictors* (preditores de desvios), unidade funcional responsável pela predição, encontra-se não somente na diminuição dos *stalls*, mas também pois, após uma predição errada, é preciso desfazer as instruções selecionadas incorretamente que estão no *pipeline*, processo que pode custar diversos ciclos de *clock*.

Devido à complexidade de tratá-los em microarquiteturas com *pipelining*, desvios condicionais tornaram-se um dos grandes limitantes de *performance*. Portanto, decisões de projeto como, por exemplo, número de estágios de *pipeline*, divisão dos recursos de cada estágio, implementação de *forwarding*, entre outros são essenciais no controle do impacto gerado por *branches*.

2.3 Aprendizado de Máquina

Esta seção apresenta uma revisão de algoritmos de aprendizado de máquina, mais especificamente de redes neurais artificiais, e sua relevância do ponto de vista de *hardware-software co-design*. O campo de aprendizado de máquina é formado por diversas abordagens. Apesar dessa diversidade, algumas se destacam por sua simplicidade e/ou eficiência.

As redes neurais artificiais são um modelo de aprendizado de máquina inspirado pelo cérebro animal e pelo funcionamento das redes de neurônios biológicos. O paradigma

de neurônios artificiais, embora proposto anteriormente, tornou-se mais amplamente difundido com a introdução do *perceptron*, introduzido por Rosenblatt (1958).

O *perceptron* modela um neurônio biológico, relacionando múltiplas entradas ponderadas com uma saída. A Figura 2.9 apresenta o esquema do *perceptron*. Assim como em neurônios, a presença ou não de uma saída, computacionalmente representada pelos valores binários 0 e 1, são determinados avaliando a soma das entradas em uma função denominada *função de ativação*. Essa relação é dada pelo seguinte par de equações (R. Valle Mesquita e Pimenta, 2005):

$$\tau(\mathbf{x}) = \sum_{i=1}^{n} x_i w_i \tag{2.5}$$

$$y(\mathbf{x}) = \varphi(\tau(\mathbf{x})) \tag{2.6}$$

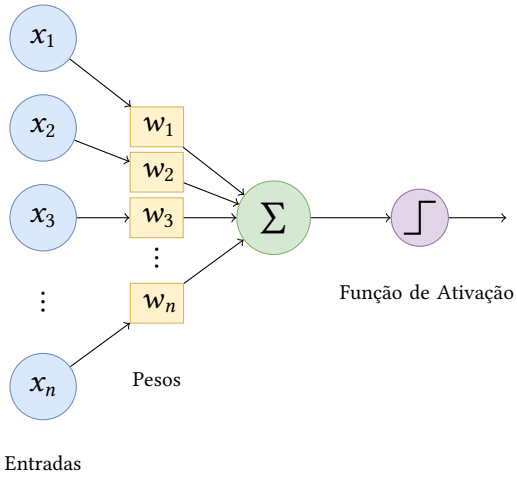

Figura 2.9: *O perceptron é um modelo de neurônio biológico, composto por n entradas e n pesos, além de uma função de saída. Os pesos atuam sobre o valor de cada entrada, determinando sua relevância. A saída é determinada por uma função, comumente função degrau, que realiza a classificação da entrada.*

onde $\mathbf{x} \in \mathbb{R}^n$ é o vetor de entradas, w_i o i-ésimo peso (constante), $\tau(\mathbf{x})$ a combinação linear, φ a função de ativação e $y(\mathbf{x})$ é a saída do neurônio. Além dos pesos, alguns modelos incluem a adição de um valor de *bias* (constate). Assim, transformam a Equação 2.5 obtendo:

$$\tau(\mathbf{x}) = \sum_{i=1}^{n} x_i w_i + b \tag{2.7}$$

onde $b \in \mathbb{R}$ é o *bias*.

Ainda que um único neurônio artificial seja limitado,[1] a construção de redes neurais com múltiplas camadas e múltiplos neurônios por camada mostra-se uma poderosa ferramenta na construção de sistemas inteligentes. A introdução da AlexNet, em 2012, popularizou

[1] A versão canônica do *perceptron*, com função degrau como ativação, funciona como classificador binário e, portanto, é incapaz de resolver problemas simples tais como o *problema XOR*.

o conceito de *Deep Neural Networks* (Redes neurais profundas) (DNNs), redes com três ou mais camadas, além daquelas de entrada e saída.

Matematicamente, podemos representar redes neurais *feedforward* (unidirecionais), isto é, aquelas onde os dados fluem somente na direção das entradas para as saídas, como matrizes. O processo de inferência, ou seja, a obtenção da saída é, consequentemente, definido pela multiplicação das matrizes de entradas, pesos e *biases*.

Os vetores de pesos da camada k formam a matriz W^k definida por:

$$W^k = \begin{bmatrix} w^k_{1,1} & w^k_{1,2} & \dots & w^k_{1,n} \\ w^k_{2,1} & w^k_{2,2} & \dots & w^k_{2,n} \\ \vdots & \vdots & \dots & \vdots \\ w^k_{m,1} & w^k_{m,1} & \dots & w^k_{m,n} \end{bmatrix} \tag{2.8}$$

Similarmente, os *biases* da camada k formam um vetor m-dimensional definido por:

$$b_k := (b_{x_1^{(k)}}, \dots, b_{x_m^{(k)}})^\top \tag{2.9}$$

Esses resultados permitem transformar as Equações 2.6 e 2.7, obtendo a saída da camada k com relação às saídas da camada $k-1$:

$$\mathbf{x}^k = y^k(\mathbf{x}^{k-1}) := \varphi(W^k \bullet \mathbf{x}^{(k-1)} + b^k) \tag{2.10}$$

onde x^{k-1} é a entrada (vetor de saídas da camada $k-1$) e y^k é o vetor de saídas da k-ésima camada. Alternativamente, podemos escrever a Equação 2.10 como:

$$y^k(\mathbf{x}^{k-1}) = \varphi\left(\begin{bmatrix} w^k_{1,1} & w^k_{1,2} & \dots & w^k_{1,n} \\ w^k_{2,1} & w^k_{2,2} & \dots & w^k_{2,n} \\ \vdots & \vdots & \dots & \vdots \\ w^k_{m,1} & w^k_{m,1} & \dots & w^k_{m,n} \end{bmatrix} \cdot \begin{bmatrix} x_{1^{k-1}} \\ x_{2^{k-1}} \\ \vdots \\ x_{n^{k-1}} \end{bmatrix} + \begin{bmatrix} b_{x_1{}^{(k)}} \\ b_{x_2{}^{(k)}} \\ \vdots \\ b_{x_m{}^{(k)}} \end{bmatrix}\right) \tag{2.11}$$

onde a função de ativação φ é aplicada sobre cada elemento do vetor de saída.

Um algoritmo para o cálculo das saídas decorre diretamente da Equação 2.11. A multiplicação das matrizes é, em essência, o aninhamento de laços. Como consequência, o algoritmo primordial de inferência sobre redes *feedforward* pode ser obtido repetindo o par de laços aninhados conforme o número de camadas. O pseudocódigo pode ser visto no Programa 2.1.

Do ponto de vista computacional, laços aninhados correspondem a desvios condicionais. Deste modo, o número de desvios condicionais na execução do algoritmo de inferência é diretamente proporcional ao número de entradas/pesos e saídas, uma vez que cada laço termina apenas quando o número de entradas/saídas é atingido.

Como discutido na Seção 2.2.5, desvios condicionais tem grande impacto no desempenho de CPUs. Por outro lado, a simplicidade do algoritmo, com abundância de multiplicações, e o potencial de paralelização, distribuindo parcelas da matriz, tornam redes neurais candidatas perfeitas para aceleração por instruções especiais do tipo *Single Instruction, Mul-*

Programa 2.1 Resumo do algoritmo de inferência em redes *feedforward* por multiplicação de matrizes.

```
1    ▷ x[]: cadeia das entradas
2    ▷ y[]: cadeia das ativações de saída
3    ▷ w[][]: matriz dos pesos da camada
4    ▷ b[]: cadeia dos biases da camada
5
6    ▷ ativação(x): função de ativação
7
8    para cada saída com índice i:
9      para cada entrada com índice j:
10       y[i] += (w[i][j] * x[j]);
11     fim
12     y[i] += b[i];
13     y[i] = ativação(y[i]);
14   fim
```

tiple Data (Única instrução operando em múltiplos dados) (SIMD) em CPUs. Contudo, ainda que essa abordagem alivie a penalidade causada por *branches*, ela não consegue resolvê-los, ao mesmo tempo que geralmente incorrem em aumento de trabalho aos programadores.

Com a crescente relevância das redes neurais, a pesquisa e o desenvolvimento de aceleradores que não sejam acometidos por esse problema, seja por meio de coprocessadores, GPUs, ASICs, entre outros, para essa finalidade domina os esforços da indústria e da academia (Reuther *et al.*, 2019). Entretanto, a maioria destas abordagens não endereçam o problema dos desvios condicionais em CPUs e, ao mesmo tempo, são inviáveis em dispositivos com recursos limitados, que dispõem tão somente de uma CPU.

Consequentemente, a metodologia proposta nesta pesquisa enfoca esse problema como objeto de estudo na abordagem de *co-design*.

Capítulo 3

Metodologia e Avaliação

O objetivo deste é trabalho avaliar a abordagem de *co-design* como metodologia para o controle do *trade-off* de *performance*. O foco desse trabalho concerne a inferência usando redes neurais em processadores de propósito geral e, por conseguinte, o custo computacional dos desvios condicionais. Mais especificamente, concentra-se nos processadores *embedded-class*, com recursos limitados.

A literatura sobre a implementação de redes neurais nessa classe de dispositivos é ampla. Shafique *et al.* (2021) discutem os principais desafios e soluções, sobretudo do ponto de vista de *software*. Garofalo *et al.* (2019) propõem uma biblioteca otimizada para um *cluster* (conjunto) de oito processadores RISC-V com extensão do conjunto de instruções voltadas para SIMD. Prakash *et al.* (2023) e Giri *et al.* (2020) introduzem ferramentas para geração de coprocessadores, com instruções customizadas, sob demanda. No entanto, carecem trabalhos com foco na modificação das *microarquiteturas* e dos *datapaths*, sem uso de instruções adicionais, em face dos algoritmos.

Nesse sentido, encontra-se em Wu et al. motivadores para otimizações desse espectro, em contrapartida às abordagens supracitadas. Por um lado, a heterogeneidade do campo dos coprocessadores, dificulta suportar diferentes plataformas. Por outro lado, a homogeneidade das CPUs, presentes em todos os *chips*, facilita a implementação dos modelos (Wu *et al.*, 2019).

No contexto desta pesquisa, a abordagem de *co-design* compreende uma série de análises e processos que guiam as decisões de projeto para o *hardware*, visando obter ganhos em comparação às decisões tomadas com outras abordagens. O procedimento compõe-se das seguintes etapas:

1. determinação do *software* para o qual se busca otimizar;
2. análise dinâmica do *software*;
3. análise da microarquitetura e possíveis otimizações;
4. implementação e avaliação do desempenho.

Para a análise da eficácia dessa abordagem, propõe-se três implementações do processador:

- **Implementação de Referência:** implementação usada como base, sem técnicas para mitigação ou eliminação dos impactos causados por *branches*;
- **Implementação Otimizada:** adiciona técnicas de *forwarding* e *branch prediction*, porém sem otimizações que considerem as particularidades do código;
- **NINA:** implementação modificada com base nas particularidades do algoritmo de inferência por multiplicação de matrizes.

Nas próximas seções, empregando o procedimento para *co-design*, será apresentado o algoritmo de *benchmark*, as estratégias de *branch prediction* e os detalhes das três implementações propostas.

3.1 Software

Nesta seção, introduzimos os componentes de *software* empregados nesta pesquisa, tanto do ponto de vista de *co-design*, quanto do ponto de vista de verificação da conformidade com a arquitetura.

Na Seção 2.3, apresentamos o algoritmo de multiplicação de matrizes para inferência com redes neurais. Consequentemente, como ponto de partida, adotamos a forma que pode ser vista no Programa 2.1. No entanto, a arquitetura das redes *feedforward*, como, por exemplo, o número de camadas ou a quantidade de neurônios por cada, varia conforme as necessidades do problema modelado.

Para não perder generalidade, ao mesmo tempo que mantemos a estrutura de laços aninhados e o procedimento de multiplicação de pesos com soma de *bias*, adota-se um algoritmo que simula e parametriza esse processo. Os parâmetros `DIM_SIZE` e `N` controlam o número de neurônios e entradas da camada, e, portanto, a dimensão da matriz, e a quantidade de repetições, representando o número de camadas, respectivamente.

A simulação das operações aritméticas de multiplicação e soma é feita usando-se os índices dos laços, nos permitindo verificar facilmente a corretude da execução. Além disso, é possível simular redes maiores, uma vez que não é necessário guardar na memória uma matriz de pesos completa e um vetor de *biases*, somente o vetor de ativações de saída.

O código-fonte completo pode ser visto no Programa 3.1.

Para verificar a conformidade com conjunto de instruções RV32I, adota-se a suíte de testes unitários `RISCV-TESTS`,[1] mantida pela RISC-V International.[2]

3.2 Hardware

A organização de computadores compõe a parte do *hardware* que implementa uma arquitetura do conjunto de instruções. O desenho de um processador passa, portanto, por decisões de organização que afetam *trade-offs* consoantes com os objetivos do projeto.

[1] github.com/riscv-software-src/riscv-tests

[2] Fundação internacional responsável pelo desenvolvimento da especificação.

Programa 3.1 Simulação de inferência por multiplicação de matrizes.

```
1   #define N 1
2   #define DIM_SIZE 1
3   #define EXPECTED_VALUE 0
4
5   static uint32_t Y[DIM_SIZE] = {0};
6
7   uint32_t matmul() {
8      uint32_t sum = 0;
9      uint32_t i, j;
10     for (i = 0; i < DIM_SIZE; i++) {
11        Y[i] = 0;
12        for (j = 0; j < DIM_SIZE; j++) {
13           /* Simula multiplicação dos pesos */
14           Y[i] += i * j;
15        }
16        /* Simula bias */
17        Y[i] += i;
18        sum += Y[i];
19     }
20     return (uint32_t) sum;
21  }
22
23  int main(int argc, char**argv) {
24     /* Simula N camadas */
25     uint32_t sum = 0;
26     uint8_t i;
27     for (i = 0; i < N; i++) sum = matmul();
28     if (sum == EXPECTED_VALUE) return 0;
29     return 1;
30  }
```

Esta seção explora o processo, as decisões tomadas e suas finalidades nas implementações propostas nesta pesquisa. Nesse contexto, empregamos uma abordagem incremental, partindo de uma microarquitetura base e refinando-a, atingindo a implementação projetada para o Programa 3.1.

3.3 Implementação de Referência

O conjunto de unidades funcionais e suas interconexões formam o *datapath*, elemento primário no desenho de processadores. A determinação do *datapath* é dividida nas seguintes etapas:

1. determinação das unidades funcionais;
2. determinação de interconexões, sinais de controle e unidades auxiliares;
3. determinação do uso de *pipeline*, quantidade de estágios e divisão dos estágios;
4. determinação de estratégias para mitigação e/ou eliminação de *hazards*.

Na implementação de referência, utilizam-se as unidades funcionais descritas no Capítulo 2 com memória separadas, seguindo a arquitetura Harvard, mais especificamente:

- memória de instruções;
- banco de registradores;
- unidade lógica e aritmética;
- memória de dados;
- unidade de controle.

Adicionalmente, empregam-se três unidades auxiliares:

- **Gerador de Máscara de Escrita (*Write Mask Generator*):** para escritas menores que uma *word* (palavra), ou 32 *bits*, gera máscara com valor 1 nos *bits* da respectiva metade ou quarto a ser escrito e 0 nos demais *bits*;
- **Gerador de Dados para Escrita (*Write Data Generator*):** para escritas menores que uma *word*, aplica a máscara de escrita nos dados;
- **Extensor de Dados de Leitura (*Load Extender*):** para leituras menores que uma *word*, estende os dados com valor 0 ou 1 os demais *bits*.

A organização dos elementos lógicos responsáveis por essas tarefas em unidades auxiliares, em contraste à integração como componentes discretos ao longo do *datapath*, não afeta diretamente o desempenho, como, por exemplo, incrementando o atraso de propagação dos sinais, ao mesmo tempo que melhora o gerenciamento do código RTL e a verificação de corretude.

3.3.1 Pipelining

No Capítulo 2, discutimos a importância de *pipelining* para explorar o *instruction-level parallelism* e, consequentemente, obter ganhos de *performance*. Por outro lado, essa abordagem incorre em *pipeline hazards*, eventos que impedem a execução sobreposta e, portanto, a exploração do potencial de ILP. Por esse motivo, as decisões relativas ao *pipeline*, como número de estágios, estão diretamente relacionadas ao *trade-off* entre desempenho e complexidade da implementação.

Como uma arquitetura que segue o conceito RISC, a RISC-V é projetada para uso de *pipelining*, mais especificamente o *pipeline* clássico de cinco estágios apresentado na Seção 2.2.4. Apesar disso, dispositivos *embedded-class* comumente implementam somente três estágios, como, por exemplo, núcleos ARM Cortex-M0, Cortex-M1, Cortex-M3, Cortex-M4, Cortex-M23, bem como núcleos RISC-V SiFive E21, entre outros.

O número ideal de estágios, no contexto de *performance*, é investigado por Hartstein e Puzak. Os pesquisadores demonstram que essa quantidade é inversamente proporcional ao número de *hazards* não resolvidos. Além disso, *pipelines* longos podem neutralizar os efeitos de *branch prediction* pouco eficazes (Hartstein e Puzak, 2002; Hartstein e Puzak, 2004).

Para a implementação de referência, usam-se três estágios. Por um lado, pois segue o padrão para dispositivos desta classe. Por outro lado, por atenuar o número de *hazards* não relacionados aos desvios condicionais, objeto deste trabalho.

A divisão dos estágios considera o atraso na propagação dos sinais gerados por cada unidade funcional, de modo que não exista estágio com atraso total discrepante e, como consequência, que limite a frequência de *clock* máxima. Os três estágios são:

1. **Fetch & Decode (Busca e decodificação):** o endereço contido no *program counter* é transmitido para a memória de instruções. A saída é decodificada e os registradores são acessados no banco de registradores;
2. **Execute (Execução):** a instrução é executada pela unidade lógica e aritmética;
3. **Memory Access & Write-back (Acesso à memória e retorno):** o valor dos registradores é atualizado e, para instruções de leitura e escrita, a memória é acessada neste estágio.

Nessa implementação, o tratamento de *hazards* é feito por meio de *stalls*, com custo de dois ciclos de *clock* para cada *hazard*. A Tabela 3.1 resume as características dessa implementação.

	Implementação de Referência
ISA	RV32I
Arquitetura	Harvard
Memória de Instruções	16 KB
Memória de Dados	16 KB
Tratamento de Hazards	*Stalls*
Contadores	Ciclos e Instruções Executadas
Linguagem de Descrição de *Hardware*	Verilog

Tabela 3.1: *Resumo das características da implementação de referência.*

3.4 Implementação Otimizada

A implementação otimizada constrói sobre a implementação de referência, com foco em mitigar os custos por conflitos e dependências. Adotam-se as estratégias de *forwarding* e *branch prediction*.

Em relação ao *forwarding*, são adicionados três caminhos, onde cada caminho apode adiantar tanto o valor do *rs1* quanto o valor do *rs2*:

- *write-back* para *execute*;
- *write-back* para *fetch & decode*.
- *execute* para *fetch & decode*.

Essa solução resolve os *data hazards*.

Existem duas abordagens para predição de desvios, apresentadas no Seção 2.2.5: estática e dinâmica. A primeira baseia-se na escolha de uma política fixa, seja tomar o desvio ou não.

A segunda baseia-se na análise de resultados anteriores, guardados em estruturas dedicadas, para a determinação do caminho do desvio, se ele é tomado ou não, e possivelmente o endereço, evitando a necessidade de calculá-lo. Em ambas as estratégias, é necessário desfazer instruções que estejam no *pipeline* em decorrência de uma predição equivocada.

A predição estática é, comumente, adotada em microarquiteturas com foco em simplicidade de implementação e/ou redução de área de silício ocupada. Embora pouco eficiente para cargas em geral, essa estratégia promove bons resultados em algoritmos com grande quantidade de laços, como o de multiplicação de matrizes. Neste sentido, a escolha de não tomar os desvios como política fixa é eficaz, uma vez que desvios por laços serão tomados ao máximo uma vez.

Por outro lado, a estratégia de predição dinâmica é tipicamente adotada em implementações com foco em *performance*, pois se adapta melhor a maior variedades de algoritmos. Deste modo, a implementação otimizada proposta incorpora predição dinâmica.

3.4.1 Branch Predictor

Nesta implementação, propõe-se a utilização de uma *Branch History Table* (Tabela de histórico de desvios) (BHT). A BHT guarda o resultado provável, baseado no histórico, para uma certa quantidade de desvios, identificados por seu endereço na memória.

Os valores da tabela são atualizados de acordo com um *2-bit Saturating Counter* (Contador de 2-*bit*). Os quatro valores que o contador assume são: fortemente não tomado, fracamente não tomado, fracamente tomado e fortemente tomado. Quando o valor é maior ou igual a um, o desvio é predito como tomado. Predições corretas somam ao contador, enquanto predições incorretas subtraem. A Figura 3.1 apresenta o esquema do contador.

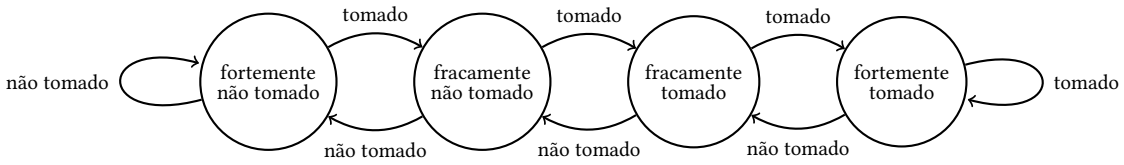

Figura 3.1: *Esquema de 2-bit saturating counter. Os desvios condicionais são preditos com base no valor do contador. Predições corretas incrementam o valor do contador, enquanto predições incorretas decrementam.*

Quando um desvio condicional é decodificado no estágio de *fetch & decode*, o seu endereço é enviado para a unidade de predição de desvios, que acessa a BHT e transmite o resultado predito. No próximo estágio, quando o resultado é validado, a saída é transmitida novamente à unidade para atualização da tabela.

3.5 NINA

Nesta seção apresentaremos o **N**INA **I**s **N**ot **A**RM (NINA), um processador RISC-V otimizado por *co-design* para aprendizado de máquina, mais especificamente inferência por redes neurais *feedforward*.

Nos Capítulos 1 e 2, apresentamos e discutimos o problema dos *pipeline hazards* em processadores de propósito geral, sobretudo aqueles causados por desvios condicionais, e sua relação com laços, como encontrado no algoritmo de inferência por multiplicação de matrizes (veja o Programa 2.1). Nesse contexto, discutimos no capítulo 3 duas implementações de processadores.

Para o projeto do NINA, parte-se da implementação apresentada na Seção 3.3, com objetivo de obter melhor desempenho que aquela apresentada na Seção 3.4.

Deste modo, são necessárias alterações na microarquitetura que, por um lado, diminuam o CPI e, por outro lado, mantenham ou aumentem a frequência máxima. Enfoca-se, portanto, técnicas alternativas à *branch prediction* por *2-bit saturating counter*.

Lalja (1988) propõe as seguintes abordagens para mitigação ou eliminação dos custos de *branches*:

- **Tabela de Branches:** discutida na Seção 3.4.1;
- ***Branch Delay Slot (Branch Atrasado):*** consiste em executar um número predeterminado de instruções subsequentes ao *branch* independentemente de seu resultado;
- **Execução Múltipla:** consiste em executar, simultaneamente, ambos os caminhos;
- **Fluxos de Execução Múltipla:** técnica que consiste em intercalar dois fluxos de instruções independentes simultaneamente, garantindo que o *pipeline* esteja ocupado com a instrução de um fluxo enquanto resolve o desvio condicional de outro;
- **Resolução Adiantada:** consiste em mover a resolução dos desvios para os estágios iniciais, diminuindo o custo de *stalls*.

A primeira técnica, embora eficaz, foi adotada na implementação otimizada e, portanto, não é adequada. A técnica de *branch atrasado*, ainda que com potencial de eliminar o uso de *stalls*, caso seja possível reordenar as instruções para que as dependências de controle estejam suficientemente distantes, efetivamente solucionando o problema dos *branches*, necessita de suporte dos compiladores. Como discutido na Seção 3.1, abordagens centradas em *software* não são consideradas neste trabalho.

Técnicas fundamentadas em execução múltipla são amplamente empregadas por processadores de alto desempenho. No entanto, o custo de complexidade de implementação e, sobretudo, área de silício as tornam incompatíveis com dispositivos *embedded-class*.

A resolução dos desvios nos estágios iniciais representa uma técnica pouco empregada. Por um lado, é preciso mover, ou adicionar em microarquiteturas que utilizam a ALU para executar *branches*, os elementos lógicos responsáveis pela decisão dos desvios, possivelmente desbalanceando o *pipeline* e afetando a frequência de *clock* máxima. Além disso, é necessário estender o *forwarding* para estágios anteriores, aumentando o atraso de propagação, com potencial impacto na frequência máxima. Por outro lado, algumas dependências não podem ser resolvidas por esta técnica, limitando a melhoria de CPI.

Aplicando *co-design*, a análise dinâmica dos algoritmos de inferência por multiplicação de matrizes, em particular o Programa 3.1, nos permite concluir que as dependências não resolvidas pela resolução adiantada para a configuração de *pipeline* proposta, como, por

exemplo, desvios dependentes de leituras imediatamente anteriores, uma vez que a leitura só será completada no último estágio e, portanto, após a decisão da próxima instrução a ser buscada, não ocorrem nesse programa. Consequentemente, pode-se simplificar a implementação sem perda de desempenho na execução da aplicação, removendo caminhos de *forwarding* e simplificando os critérios de detecção.

Em relação aos atrasos por *forwarding* ou adição de elementos lógicos para resolução dos desvios, nota-se que, para o *pipeline* proposto, é possível otimizar o tempo de propagação, e adicionalmente a área, ao substituir a solução da implementação otimizada pela resolução adiantada.

Conclui-se, portanto, que para a aplicação proposta, a resolução adiantada é técnica viável e adequada. Como consequência, emprega-se esta solução no NINA, adicionando uma unidade funcional para comparação de *branches* no estágio de *fetch & decode*. O diagrama com todos os detalhes acerca do *datapath* do NINA encontra-se no Apêndice A (veja a Figura A.1).

3.6 Avaliação Experimental

A fase de avaliação experimental tem por objetivo estabelecer a viabilidade da abordagem de *co-design*, proposta nesta pesquisa através do NINA, quantificando os deltas comparativamente às outras abordagens.

Os experimentos são conduzidos em dois ambientes: simulação e síntese em FPGA. O primeiro permite a automatização e reprodutibilidade, enquanto o segundo possibilita a avaliação de parâmetros relativos à implementação física, como, por exemplo, o uso de recursos lógicos.

O código-fonte completo, compreendendo o código RTL do NINA e a bancada de teste contendo o Programa 3.1, encontra-se disponível junto deste trabalho.

3.6.1 Ferramentas e Ambiente de Experimentação

O *software* usado na simulação do circuito foi o Icarus Verilog,[3] compilador Verilog de código-aberto. A mensuração da quantidade de ciclos decorridos e instruções executadas, para determinação de CPI, é realizada a partir dos respectivos contadores presentes em todas as implementações.

O conjunto de *hardware*, bem como as ferramentas de *software*, usadas na implementação física em FPGA foram:

- Plataforma Sipeed Tang Primer 20K;
- *Chip* FPGA Gowin GW2A-LV18PG256C8/I7;
- Ambiente de desenvolvimento integrado Gowin EDA.

[3] github.com/steveicarus/iverilog

3.6.2 Experimento e Resultados

O experimento para a avaliação de desempenho consistiu na execução do Programa 3.1 com diferentes valores para o parâmetro `DIM_SIZE`, que determina o tamanho da matriz. Adotaram-se os valores 16, 32, 64 e que representam boas aproximações de ambos os extremos, inferior e superior, de neurônios por camadas em modelos para dispositivos *embedded-class*, com recursos limitados. A métrica usada é a de *throughput*, em milhões de instruções por segundo (MIPS), conforme discutido na Seção 2.2.1. A Tabela 3.2 apresenta os resultados. Os resultados experimentais demonstram que, a abordagem de *co-design*,

Métrica	Referência	Otimizada	NINA
Ciclos por Instrução	1,18	1,1	1,0
Frequência Máxima	45 MHz	53 MHz	62,5 MHz
Throughput	38,13 MIPS	48,18 MIPS	62,5 MIPS
Recursos Lógicos	1642 LUTs	1913 LUTs	1729 LUTs

Tabela 3.2: *Desempenho das implementações propostas sob métricas de desempenho e uso de recursos lógicos.*

empregada no NINA, obteve êxito em melhorar o *trade-off* entre desempenho e complexidade de implementação. Por um lado, temos uma redução de CPI de 15, 25% e 9, 09% quando comparado às implementações de referência e otimizada, respectivamente. Por outro lado, a simplificação da microarquitetura, e, portanto, do *datapath*, resultaram em aumento da frequência máxima, na FPGA adotada para os experimentos, de 38, 88% e 17, 92%, respectivamente.

Deste modo, houve ganhos de 63, 91% e 29, 72%, respectivamente, em termos de instruções executadas por segundo, validando a eficácia da abordagem.

Com relação ao uso de recursos, embora o NINA apresente acréscimo de recursos lógicos da FPGA, as *Lookup Tables* (Tabelas de consulta) (LUTs), comparativamente à implementação de referência, houve redução em relação à implementação otimizada. Tal redução ocorre em consequência da adoção de um método para resolução de desvios condicionais mais simples, possibilitado pela análise da aplicação e verificação de que, a despeito de suas limitações, tal método era suficiente e eficaz para o Programa 3.1.

Capítulo 4

Conclusões

Esta pesquisa se inicia com a observação das tendências em fabricação de semicondutores, como a desaceleração da Lei de Moore e da Escalabilidade de Dennard, e seus efeitos nos avanços em arquitetura de computadores. Como consequência, arquitetos voltaram-se para a adoção de novos paradigmas e abordagens de projeto. *Hardware-software co-design*, abordagem de desenvolver *hardware* para um *software* específico e vice-versa, surge como uma dessas propostas.

Neste contexto, se colocaram as questões de pesquisa, sobretudo com enfoque na eficácia dessa metodologia em realizar o *trade-off* entre desempenho computacional e complexidade de implementação do processador, comparativamente ao método tradicional.

No Capítulo 3, apresentamos o NINA, um processador projetado por meio de *co-design*, e discutimos os resultados da avaliação experimental, comparando-o a outras implementações.

Neste capítulo, retomamos as questões de pesquisa, em face dos das discussões e resultados apresentados nos capítulos 1, 2 e 3, bem como sugerimos trabalhos futuros.

4.1 Questões de Pesquisa

As primeiras questões de pesquisa propostas abordam as condições necessárias para ser viável a aplicação de *hardware-software co-design*. No Capítulo 2, discutimos o processo de avaliação de desempenho de processadores. Neste sentido, a característica primordial de CPUs, em contraste aos aceleradores, é sua generalidade. Por esse motivo, um arquiteto deve conseguir controlar *trade-offs* que permitam à sua implementação ser competente em toda uma gama de aplicações esperadas para processadores de propósito geral.

No Capítulo 3, demonstramos que as decisões tomadas no desenho do NINA realizam *trade-offs* demasiadamente custosos para certas aplicações, porém irrelevantes no contexto da aplicação para a qual ele foi otimizado. Sendo assim, evidencia-se que a aplicação de *co-design* está condicionada a escopos de *software* reduzidos, com possível perda de eficácia caso contrário.

A terceira questão de pesquisa aborda os potenciais ganhos de desempenho computacional alcançados por meio de *co-design*. Como apresentado no capítulo 3, essa abordagem permitiu não apenas explorar uma maior gama de soluções para o problema de desvios condicionais, diminuindo efeitos negativos que afetam a *performance*, mas também identificar componentes que poderiam ser simplificados, como, por exemplo, a lógica de *forwarding*.

Consequentemente, o ganho de desempenho ocorreu nas duas dimensões, ciclos necessários por instrução e frequência de *clock*, resultando em ganhos percentuais de 29, 72%.

4.2 Sugestões para Trabalhos Futuros

A proposta desenvolvida neste trabalho constitui um estudo inicial da viabilidade de *hardware-software co-design* e, consequentemente, pode ser estendida e refinada. Neste sentido, destacam-se as seguintes sugestões para trabalhos futuros:

- para a inferência por multiplicação de matrizes, pode ser explorada a adição da extensão M ao conjunto de instruções do NINA, com instruções dedicadas para as operações de multiplicação;
- é possível explorar a aplicação de *co-design* para processadores superescalares, capazes de executar mais de uma instrução por ciclo de *clock*;
- é possível explorar, para além das otimizações de microarquitetura, o espaço de implementação física do processador.
- considera-se a exploração do método de *co-design* para outros algoritmos e aplicações.

Apêndice A

Datapath do NINA

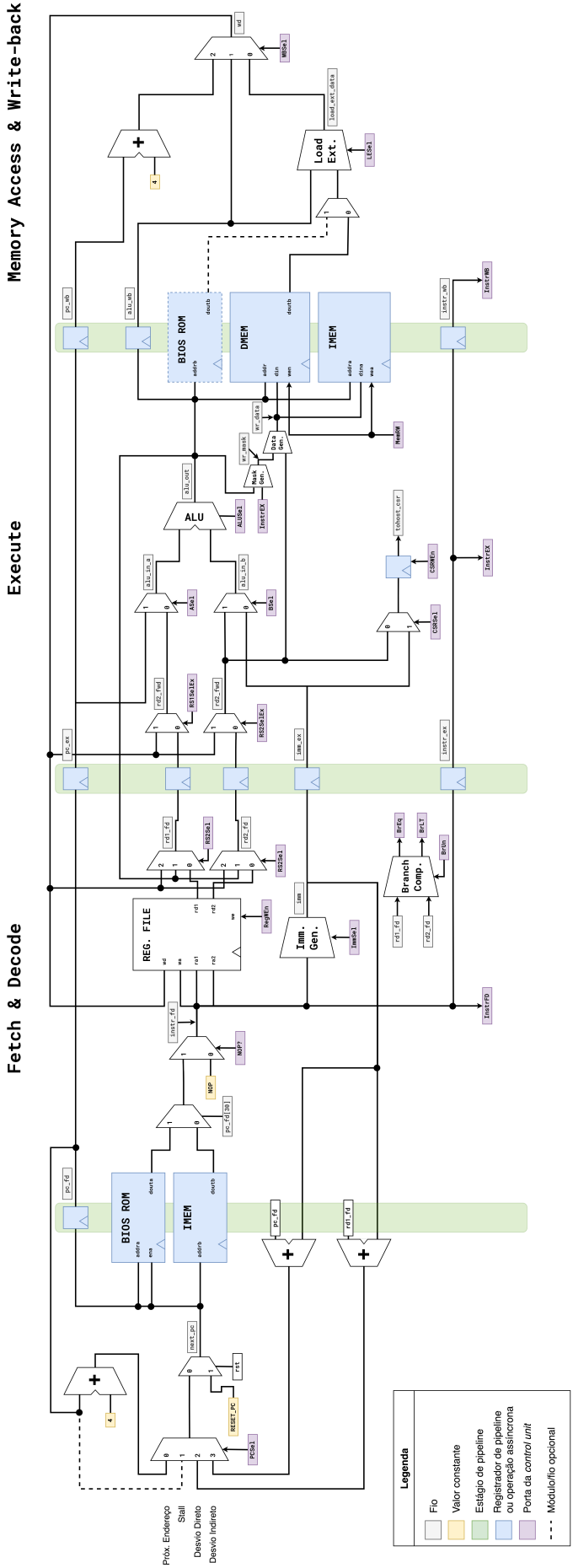

Figura A.1: *Diagrama de datapath do NINA.*

Referências

[Amdahl 2007] Gene M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities, reprinted from the afips conference proceedings, vol. 30 (atlantic city, n.j., apr. 18–20), afips press, reston, va., 1967, pp. 483–485, when dr. amdahl was at international business machines corporation, sunnyvale, california". *IEEE Solid-State Circuits Society Newsletter* 12.3 (2007), pp. 19–20. doi: 10.1109/N-SSC.2007.4785615 (citado na pg. 3).

[Backus 1978] John Backus. "Can programming be liberated from the von neumann style? a functional style and its algebra of programs". *Commun. ACM* 21.8 (ago. de 1978), pp. 613–641. issn: 0001-0782. doi: 10.1145/359576.359579. url: https://doi.org/10.1145/359576.359579 (citado na pg. 10).

[Bard 1973] Y. Bard. "Experimental evaluation of system performance". *IBM Systems Journal* 12.3 (1973), pp. 302–314. doi: 10.1147/sj.123.0302 (citado na pg. 12).

[Dennard *et al.* 1974] R.H. Dennard *et al.* "Design of ion-implanted mosfet's with very small physical dimensions". *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268. doi: 10.1109/JSSC.1974.1050511 (citado na pg. 2).

[J. Dongarra *et al.* 1987] Jack Dongarra, Joanne L. Martin e Jack Worlton. "Computer benchmarking: paths and pitfalls: the most popular way of rating computer performance can confuse as well as inform; avoid misunderstanding by asking just what the benchmark is measuring". *IEEE Spectrum* 24.7 (1987), pp. 38–43. doi: 10.1109/MSPEC.1987.6448963 (citado na pg. 12).

[J. J. Dongarra 1983] Jack J. Dongarra. "Performance of various computers using standard linear equations software in a fortran environment". *SIGARCH Comput. Archit. News* 11.5 (dez. de 1983), pp. 22–27. issn: 0163-5964. doi: 10.1145/859551.859555. url: https://doi.org/10.1145/859551.859555 (citado na pg. 12).

[Garofalo *et al.* 2019] Angelo Garofalo, Manuele Rusci, Francesco Conti, Davide Rossi e Luca Benini. "PULP-NN: accelerating quantized neural networks on parallel ultra-low-power RISC-V processors". *CoRR* abs/1908.11263 (2019). arXiv: 1908.11263. url: http://arxiv.org/abs/1908.11263 (citado na pg. 25).

[Giri *et al.* 2020] Davide Giri, Kuan-Lin Chiu, Giuseppe Di Guglielmo, Paolo Mantovani e Luca P. Carloni. "Esp4ml: platform-based design of systems-on-chip for embedded machine learning". In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2020, pp. 1049–1054. DOI: 10.23919/DATE48585.2020.9116317 (citado na pg. 25).

[Gray 1992] Jim Gray. *Benchmark Handbook: For Database and Transaction Processing Systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992. ISBN: 1558601597 (citado nas pgs. 12, 13).

[Hameed *et al.* 2010] Rehan Hameed *et al.* "Understanding sources of inefficiency in general-purpose chips". In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ISCA '10. Saint-Malo, France: Association for Computing Machinery, 2010, pp. 37–47. ISBN: 9781450300537. DOI: 10.1145/1815961.1815968. URL: https://doi.org/10.1145/1815961.1815968 (citado na pg. 4).

[Hartstein e Puzak 2002] A. Hartstein e Thomas R. Puzak. "The optimum pipeline depth for a microprocessor". In: *Proceedings 29th Annual International Symposium on Computer Architecture*. 2002, pp. 7–13. DOI: 10.1109/ISCA.2002.1003557 (citado na pg. 28).

[Hartstein e Puzak 2004] A. Hartstein e Thomas R. Puzak. "The optimum pipeline depth considering both power and performance". *ACM Trans. Archit. Code Optim.* 1.4 (dez. de 2004), pp. 369–388. ISSN: 1544-3566. DOI: 10.1145/1044823.1044824. URL: https://doi.org/10.1145/1044823.1044824 (citado na pg. 28).

[Hennessy e Patterson 2017] John L. Hennessy e David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN: 0128119055 (citado nas pgs. 2, 5, 20).

[Hennessy e Patterson 2018] John L. Hennessy e David A. Patterson. "A new golden age for computer architecture: domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 2018, pp. 27–29. DOI: 10.1109/ISCA.2018.00011 (citado nas pgs. 5, 12).

[Hill e Marty 2008] Mark D. Hill e Michael R. Marty. "Amdahl's law in the multicore era". *Computer* 41.7 (2008), pp. 33–38. DOI: 10.1109/MC.2008.209 (citado na pg. 3).

[Kim *et al.* 2003] N.S. Kim *et al.* "Leakage current: moore's law meets static power". *Computer* 36.12 (2003), pp. 68–75. DOI: 10.1109/MC.2003.1250885 (citado na pg. 2).

[Lalja 1988] D. J. Lalja. "Reducing the branch penalty in pipelined processors". *Computer* 21.7 (1988), pp. 47–55. DOI: 10.1109/2.68 (citado na pg. 31).

[Moore 1998] G.E. Moore. “Cramming more components onto integrated circuits”. *Proceedings of the IEEE* 86.1 (1998), pp. 82–85. doi: 10.1109/JPROC.1998.658762 (citado na pg. 2).

[Patterson 1985] David A. Patterson. “Reduced instruction set computers”. *Commun. ACM* 28.1 (jan. de 1985), pp. 8–21. issn: 0001-0782. doi: 10.1145/2465.214917. url: https://doi.org/10.1145/2465.214917 (citado na pg. 8).

[Patterson e Hennessy 2017] David A. Patterson e John L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. 1st. Morgan Kaufmann Publishers Inc., 2017 (citado nas pgs. 11, 14, 15, 18).

[Prakash *et al.* 2023] Shvetank Prakash *et al.* “Cfu playground: full-stack open-source framework for tiny machine learning (tinyml) acceleration on fpgas”. In: *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2023, pp. 157–167. doi: 10.1109/ISPASS57527.2023.00024 (citado na pg. 25).

[R. Valle Mesquita e Pimenta 2005] Marcos Eduardo R. Valle Mesquita e Márcio Annibal Pimenta. *Perceptrons Morfológico de Camada Única*. 2005. url: https://www.ime.unicamp.br/~valle/PDFfiles/SLMP_report.pdf (acesso em 27/11/2024) (citado na pg. 21).

[Reuther *et al.* 2019] Albert Reuther *et al.* “Survey and benchmarking of machine learning accelerators”. In: *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. 2019, pp. 1–9. doi: 10.1109/HPEC.2019.8916327 (citado na pg. 23).

[RISC-V International 2024] RISC-V International. *RISC-V Oozes Confidence with RVA23 Profile Ratification*. 2024. url: https://riscv.org/ecosystem-news/2024/10/risc-v-oozes-confidence-with-rva23-profile-ratification (acesso em 20/11/2024) (citado na pg. 8).

[Rosenblatt 1958] Frank Rosenblatt. “The perceptron: a probabilistic model for information storage and organization in the brain.” *Psychological review* 65 6 (1958), pp. 386–408 (citado na pg. 21).

[Shafique *et al.* 2021] Muhammad Shafique, Theocharis Theocharides, Vijay Janapa Reddy e Boris Murmann. “Tinyml: current progress, research challenges, and future roadmap”. In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 2021, pp. 1303–1306. doi: 10.1109/DAC18074.2021.9586232 (citado na pg. 25).

[Waterman e Asanović 2019] Andrew Waterman e Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. 13 de dez. de 2019. url: https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf (acesso em 10/11/2024) (citado nas pgs. 8, 11, 18).

[Wolf 2003] W. Wolf. “A decade of hardware/software codesign”. *Computer* 36.4 (2003), pp. 38–43. doi: 10.1109/MC.2003.1193227 (citado na pg. 4).

[Wu *et al.* 2019] Carole-Jean Wu *et al.* “Machine learning at facebook: understanding inference at the edge”. In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2019, pp. 331–344. DOI: 10.1109/HPCA.2019.00048 (citado na pg. 25).