**UNIVERSIDADE FEDERAL DO PAMPA**

**RENATO SAYYED DE SOUZA**

# PROPY: PROJETO E DESCRIÇÃO HDL DE UM PROCESSADOR PIPELINE NATIVO PARA EXECUÇÃO DE BYTECODES DE PYTHON

**Bagé**
**2025**

**RENATO SAYYED DE SOUZA**

# PROPY: PROJETO E DESCRIÇÃO HDL DE UM PROCESSADOR PIPELINE NATIVO PARA EXECUÇÃO DE BYTECODES DE PYTHON

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Engenharia de Computação como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação.

Orientador: Bruno Silveira Neves

**Bagé**
**2025**

Ficha catalográfica elaborada automaticamente com os dados fornecidos pelo(a) autor(a) através do Módulo de Biblioteca do Sistema GURI (Gestão Unificada de Recursos Institucionais).

S729 Souza, Renato Sayyed de

ProPy: Projeto e Descrição HDL de um Processador Pipeline Nativo para Execução de Bytecodes de Python / Renato Sayyed de Souza.

93 f.: il.

Orientador: Bruno Silveira Neves

Trabalho de Conclusão de Curso (Graduação) - Universidade Federal do Pampa, Engenharia de Computação, 2025.

1. Processador de bytecodes Python. 2. Processador personalizado. 3. Arquitetura pipeline. 4. Circuitos Integrados. 5. Sistemas embarcados. 6. Máquina virtual do Python. 7. HDL. I. Título.

**RENATO SAYYED DE SOUZA**

# PROPY: PROJETO E DESCRIÇÃO HDL DE UM PROCESSADOR PIPELINE NATIVO PARA EXECUÇÃO DE BYTECODES DE PYTHON

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Engenharia de Computação como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação.

Trabalho de Conclusão de Curso defendido e aprovado em: 24 de outubro de 2025.

Banca examinadora:

_____

Prof. Dr. Bruno Silveira Neves

Orientador

Unipampa

_____

Prof. Dr. Fábio Livi Ramos

Unipampa

_____

Prof. Dr. Fernando Gehm Moraes

PUCRS

Assinado eletronicamente por **Fernando Gehm Moraes**, **Usuário Externo**, em 29/10/2025, às 07:38, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **FABIO LUIS LIVI RAMOS**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 29/10/2025, às 10:33, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **BRUNO SILVEIRA NEVES**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 29/10/2025, às 14:20, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

A autenticidade deste documento pode ser conferida no site https://sei.unipampa.edu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **1877541** e o código CRC **2C66214A**.

Dedico este trabalho à minha família, especialmente aos meus pais e avó, que, com amor, esforço e dedicação, tornaram possível chegar até aqui

## AGRADECIMENTO

A trajetória até aqui foi longa! Lembro como se fosse ontem da minha chegada a Bagé, com um brilho gigante nos olhos para finalmente poder crescer. Achei que seria mais fácil, admito, mas não teria graça se fosse. São tantas as pessoas que tiveram influência em minha trajetória e que possibilitaram que eu chegasse onde estou que, infelizmente, não tenho como mencionar a todas.

Primeiramente, quero agradecer aos meus pais, Renato e Vitória, que sempre me apoiaram e permitiram que eu trilhasse uma história tão bonita até aqui. Quando tomei a decisão de fazer a graduação e me mudar para um lugar desconhecido, foram eles que me motivaram e me ajudaram a chegar lá. Também agradeço à minha avó, minha rainha e meu porto seguro. Faz anos que saí de casa, e todo dia ela me liga no mesmo horário para saber como estou.

Quero agradecer ao professor Fábio por ter sido o responsável por me fazer ver sentido na graduação e por ter me apresentado o mundo da microeletrônica. A primeira disciplina que tive prazer em me dedicar e explorar foi Técnicas Digitais, que, por sorte, estava sob a responsabilidade dele. Também quero agradecer ao professor Bruno pela paciência e por ter sido um grande parceiro nos últimos anos, sendo, por muitas vezes, meu confidente. Sem ele, eu não teria conseguido manter a cabeça no lugar e continuar me esforçando para chegar aqui. Ao professor Gerson, agradeço pela parceria durante a graduação. Demos muitas boas risadas nos "rolês de chopp".

Aos parceiros e amigos que a graduação me deu e que espero levar para a vida — Eduardo, Secco, Willian, Nicolas, Lucas V., Lucas B., Rodrigo, Heitor, Tomás e Gabriel —, obrigado por terem estado presentes em diversos momentos de alegria e tristeza. Sem vocês, nada disso faria sentido e não teria sido possível superar todas as dificuldades que surgiram. Aos “infortúnios” que passaram pela minha vida, muito obrigado por tudo; eu não teria conseguido crescer o suficiente sem ter passado por essas situações.

Por fim, agradeço a Bagé, o lugar ao qual sempre serei grato e do qual terei lembranças eternas. Lá me tornei adulto e me preparei para enfrentar o mundão, e disso nunca me esquecerei.

“If I have seen farther than others, it is because I stood on the shoulders of giants.”
— Sir Isaac Newton

## RESUMO

Este trabalho apresenta o desenvolvimento de uma arquitetura de hardware customizada, projetada para executar nativamente um subconjunto de bytecodes da Máquina Virtual do Python. A proposta segue uma metodologia de co-design na qual as responsabilidades de execução e gerenciamento são divididas entre dois componentes complementares: ProPy, o Processador Pipeline, um núcleo de três estágios focado na execução rápida de instruções, e o Gerenciador Dinâmico de Memória (GDM), um módulo de software encarregado da preparação de contextos (*frames*) e do acesso a variáveis. A arquitetura implementa uma ISA reduzida, capaz de executar as instruções básicas mais utilizadas da máquina virtual do Python (PVM). São duas as microarquiteturas apresentadas: uma em que o núcleo acessa dados em tempo de execução comunicando com o GDM e outra que utiliza do paradigma *carregar-executar-salvar*, restringindo a comunicação com o GDM em fronteiras de *frames*. A arquitetura é customizável, possuindo reconhecimento de diferentes tipos de dados, embora somente trate inteiros, fornecendo suporte futuro a modificações para tratamento interno de tipagem dinâmica. Além disso, é uma arquitetura parametrizável, suportando atualizações em tamanho de dados e instruções, capacidade interna de armazenamento e identificadores de instruções personalizados na ALU. A validação realizada com o caso de teste para o algoritmo *Bubble Sort* demonstra a capacidade da arquitetura em executar aplicações reais, desde que requeridos os bytecodes para compreender a ISA suportada. O caso de teste explicita a eficiência da arquitetura, apresentando resultados significativos de *speedup* em ambos os modelos de execução. Em comparação com a implementação de referência constituída pelo código Python do *Bubble Sort* exemplo, a microarquitetura que acessa dados comunicando com o GDM em tempo de execução (i) é 275 vezes mais rápida, e a arquitetura que acessa os dados internamente (ii) é 435 vezes mais rápida, sendo a (ii) 1.58 vezes mais rápida em comparação direta com a (i). A aceleração em relação à referência se dá, em grande parte, pelo suporte nativo aos *bytecodes* da PVM, uma vez que em software são necessárias 72 milhões de instruções para o caso de teste, enquanto no ProPy somente 542 são necessárias.



**Palavras-chave:** Processador de bytecodes Python. Processador personalizado. Arquitetura pipeline. Circuitos Integrados. Sistemas embarcados. Máquina virtual do Python. HDL.

## ABSTRACT

This work presents the development of a custom hardware architecture designed to natively execute a subset of Python Virtual Machine bytecodes. The proposal follows a co-design methodology in which execution and management responsibilities are divided between two complementary components: ProPy, the Pipeline Processor, a compact three-stage core focused on fast instruction execution, and the Dynamic Memory Manager (DMM), a software module tasked with context preparation (*frames*) and variable access. The architecture implements a reduced ISA, capable of executing the most commonly used basic instructions of the Python Virtual Machine (PVM). Two microarchitectures are presented: one in which the core accesses data at runtime by communicating with the GDM, and another that uses the *load-execute-store* paradigm, restricting communication with the GDM at *frame* boundaries. The architecture is customizable, with recognition of different data types, although it only handles integers, providing future support for modifications for internal handling of dynamic typing. It is also a parameterizable architecture, supporting updates in data and instruction widths, internal storage capacity, and custom instruction identifiers in the ALU. Validation carried out with the test case for the *Bubble Sort* algorithm demonstrates the architecture's ability to execute real applications, provided that the bytecodes are required to match the supported ISA. The test case makes the architecture's efficiency explicit, presenting significant *speedup* results in both execution models. Compared to the reference implementation consisting of the Python code from the *Bubble Sort* example, the microarchitecture that accesses data by communicating with the GDM at runtime (i) is 275 times faster, and the architecture that accesses the data internally (ii) is 435 times faster, with (ii) being 1.58 times faster in direct comparison with (i). The acceleration relative to the reference is largely due to native support for the PVM *bytecodes*, since in software 72 million instructions are required for the test case, whereas in ProPy only 542 are required.



**Keywords:** Python bytecode processor, Custom processor, Pipeline architecture, Integrated circuits, Embedded systems, Python virtual machine, HDL.

## LISTA DE FIGURAS

## LISTA DE TABELAS

## LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| ALU | Arithmetic Logic Unit |
| AOT | Ahead-of-Time |
| ASIC | Application-specific Integrated Circuit |
| CDC | Cross Domain Crossing |
| CISC | Complex Instruction Set Computer |
| CPI | Cycles Per Instruction |
| CPU | Central Processing Unit |
| DCC | Data Coherence Controller |
| DUT | Design Under Test |
| EDA | Eletronic Design Automation |
| FIFO | First-In-First-Out |
| FMAX | Maximum Frequency |
| FPGA | Field-Programmable Gate Array |
| GDM | Gerenciador Dinâmico de Memória |
| GIL | Global Interpreter Lock |
| GPU | Graphics Processing Unit |
| HDL | Hardware Description Language |
| HLS | High-Level Synthesis |
| IPC | Instructions Per Cycle |
| ISA | Instruction Set Architecture |
| JIT | Just-In-Time |
| JVM | Java Virtual Machine |
| LC | Logic Cell |
| LIFO | Last-In-First-Out |

| | |
|---|---|
| OO | Object Oriented |
| PC | Program Counter |
| PVM | Python Virtual Machine |
| RISC | Reduced Instruction Set Computer |
| TCC | Trabalho de Conclusão de Curso |
| TOS | Top of Stack |
| UNIPAMPA | Universidade Federal do Pampa |
| WCET | Worst-Case Execution Time |

**SUMÁRIO**

# 1 INTRODUÇÃO

O desenvolvimento da sociedade contemporânea está intrinsecamente ligado às capacidades tecnológicas disponíveis nas mais diversas áreas de conhecimento. Dentre estas, os sistemas computacionais formam o seu alicerce, simplificando tarefas e viabilizando a comunicação em escala global, por exemplo. Tais sistemas, por sua vez, são dependentes da programação, seja ela de alto ou baixo nível, para serem construídos e operados. Diante do vasto ecossistema de linguagens disponíveis, a escolha da linguagem mais adequada emerge como um fator crítico ao pensar na capacidade de inovação e produtividade destes sistemas, pois cada linguagem detém características próprias que influenciam no desempenho, segurança, manutenção e escalabilidade das aplicações.

Neste contexto, o Python é posicionado como uma das principais linguagens de programação, influenciando de forma pervasiva iniciativas acadêmicas tanto quanto o mercado tecnológico. A linguagem tornou-se o padrão para pesquisas exploratórias, interativas e orientadas pela computação desde o início da década de 2010 (MILLMAN; AIVAZIS, 2011). Índices recentes reafirmam a linguagem no topo da popularidade entre as diversas disponíves, como nos mais recentes índices *TIOBE* (TIOBE Software, 2025), levantamentos do *IEEE Spectrum* (CASS, 2024) e pesquisa de desenvolvedores do *Stack Overflow* (Stack Overflow, 2024).

O destaque obtido pela linguagem Python advém de características como a simplicidade semântica, próxima à linguagem natural que possibilita a execução de tarefas complexas com poucas linhas de código; o suporte a diferentes paradigmas de programação; a extensibilidade para uso em conjunto com outras linguagens; o uso de tipagem dinâmica, não limitando o uso de variáveis para somente um tipo de dado; a sua extensa biblioteca padrão; o acesso em código aberto, permitindo à comunidade de desenvolvedores aperfeiçoar o código de maneira conjunta; e a sua portabilidade, permitindo que o mesmo código possa ser executado em diferentes sistemas operacionais (CUTTING; STEPHEN, 2021).

As características da linguagem permitem a sua aplicação em diversos domínios, consolidando-a como uma plataforma de uso geral. Sua utilização é evidente no desenvolvimento de aplicações para a internet, automação de tarefas, automação em infraestruturas escaláveis e, em análise e ciência de dados (SAABITH; VINOTHRAJ; FAREEZ, 2020). Um exemplo é a *Netflix*, empresa de transmissão de vídeo atuante em todo o globo, que adota Python em componentes críticos de sua arquitetura, incluindo

serviços de monitoramento e balanceamento na rede de distribuição de conteúdo.

Áreas de vanguarda e alta complexidade, como o aprendizado de máquina e a inteligência artificial detêm o Python como sua linguagem padrão (RAYHAN, 2023). Isso se dá pela característica de extensibilidade, que permite a integração com bibliotecas de alta performance. Módulos computacionalmente intensivos são escritos em linguagens como C/C++ para obter o máximo de desempenho e, em seguida disponibilizados através de interfaces simples em Python.

Contudo, a própria necessidade de recorrer a extensões de baixo nível para tarefas intensivas expõe um paradoxo central da linguagem: o Python puro, por si só, enfrenta limitações de desempenho inerentes ao seu design. Essa característica deriva de seu conceito fundamental de ser uma linguagem interpretada (O'CONNOR; TREMBLAY, 1997). Em sua implementação padrão, o CPython (CANNON, 2005), o código fonte não é compilado diretamente para instruções da máquina hospedeira nativa; em vez disso, é traduzido dinamicamente, em tempo de execução, para um código intermediário (*bytecode*), que por sua vez é executado pela Máquina Virtual do Python (*PVM - Python Virtual Machine*).

O Java, linguagem interpretada consolidada no mercado à décadas, compartilha a mesma limitação de desempenho, porém utiliza a compilação estática, realizada previamente à execução, para a tradução do código fonte em *bytecodes*, além de dispor em sua implementação um compilador *JIT* (*Just-In-Time*), responsável por analisar e traduzir as instruções mais utilizadas em instruções nativas da máquina hospedeira (LINDHOLM et al., 2013), o qual não existe na implementação do Python.

É justamente este modelo de execução que garante a portabilidade das linguagens (WATTERS; ROSSUM; AHLSTROM, 1996), permitindo que o mesmo código execute em diferentes sistemas operacionais sem alteração. No entanto, essa flexibilidade impõe um custo em velocidade de processamento quando comparado a linguagens compiladas, justificando a constante busca por otimizações.

Endereçando as limitações de desempenho do CPython, implementações alternativas da linguagem foram desenvolvidas, propondo abordagens distintas para otimizar a execução do código Python. O Jython (The Jython Project, 2025), por exemplo, implementa a linguagem sobre a plataforma Java. Essa abordagem permite que o código Python seja executado na Máquina Virtual do Java (*JVM - Java Virtual Machine*), garantindo acesso direto a todo o ecossistema de bibliotecas Java. Contudo, seu desenvolvimento estagnou, oferecendo suporte apenas até a versão 2 do Python, o

que o tornou obsoleto para a maioria das aplicações modernas. Já o PyPy (The PyPy Team, 2025) emprega um compilador JIT, além de viabilizar a programação concorrente, pois não possui o Bloqueio Global do Interpretador (*GIL - Global Interpreter Lock*), uma limitação do CPython que impede a execução verdadeiramente paralela de múltiplas threads.

Apesar da existência dessas implementações alternativas, a estratégia predominante na comunidade para contornar as barreiras de desempenho intrínsecas à Máquina Virtual do Python consolidou-se no uso de bibliotecas com extensões em C/C++ e na delegação de tarefas pesadas para processamento em Unidades de Processamento Gráfico (*GPUs - Graphics Processing Unit*).

Essa abordagem se alinha a um dos princípios da computação: a aceleração por hardware. É um conceito bem estabelecido que um circuito dedicado a uma tarefa específica tende a executá-la com muito mais eficiência do que um algoritmo equivalente em software executando em um processador de uso geral (GREVERA; UDUPA; ODHNER, 2000).

Curiosamente, enquanto este princípio foi explorado em outras plataformas baseadas em máquinas virtuais, como o Java, essa lacuna é observada no ecossistema Python. Especificamente, identifica-se uma carência de trabalhos que investiguem a aceleração por hardware aplicada diretamente ao núcleo da Máquina Virtual do Python, sobretudo de análises quantitativas que explorem o potencial de ganho ao se acelerar os componentes centrais do interpretador.

Atuando diretamente sobre essa lacuna de pesquisa identificada, o trabalho de (BITENCOURT, 2019) apresentou a proposta inicial do projeto PamPy. Neste estudo, foi desenvolvida uma arquitetura de processador monociclo para a PVM, utilizando Linguagem de Descrição de Hardware (*HDL - Hardware Description Language*), focada em um subconjunto de instruções aritméticas, lógicas e de controle de fluxo. Embora a pesquisa tenha demonstrado a viabilidade de executar bytecodes Python em hardware dedicado, sua arquitetura monociclo se demonstrou insuficiente para a execução de programas, servindo, contudo, como um alicerce para a sua evolução.

Desta forma, o presente trabalho tem como objetivo principal dar continuidade ao projeto PamPy, evoluindo sua arquitetura para um processador pipeline descrito em HDL, posicionado como um co-processador para algoritmos escritos na linguagem Python. Ele é responsável pela execução nativa de *bytecodes* em nível de *frame*, enquanto a preparação e o gerenciamento dos contextos de execução ficam a cargo de um gerenciador

de memória externo, trabalho desenvolvido por (BENFICA, 2024) em seu TCC do curso de Engenharia de Computação .

A arquitetura proposta visa ser capaz de executar programas simples, porém projetada com foco em extensibilidade, permitindo a adição de suporte a novas instruções. No papel de co-processador, o módulo HDL é capaz de se comunicador com o sistema onde será integrado por meio de uma interface e de um protocolo *handshake* de quatro fases, garantindo o carregamento e escrita de retorno do contexto do *frame* executado e viabilizando a evolução para a execução de aplicações em um sistema completo.

A implementação de uma arquitetura pipeline representa um avanço significativo, acrescendo o paralelismo em nível de instrução, objetivando reduzir o tempo de execução. Ao final deste projeto entrega-se uma base de hardware validada e expansível, que em trabalhos futuros permitirá a adição de instruções para um suporte mais completo à PVM, incluindo funcionalidades complexas como o gerenciamento dinâmico de objetos e vetores, pavimentando o caminho para uma plataforma robusta e eficiente para a execução de algoritmos reais desenvolvidos em Python.

## 1.1 Objetivos

- Objetivo Geral: Desenvolvimento da arquitetura em hardware de um processador Python pipeline, descrito em HDL, capaz de executar aplicações da linguagem Python, compreendendo um subconjunto das instruções suportadas pela sua máquina virtual, além de disponibilizar uma interface e protocolo para comunicação com o gerenciador de memória (*GDM - Gerenciador Dinâmico de Memória*) desenvolvido no âmbito de um outro TCC do curso de Engenharia de Computação, sendo este gerenciador atualmente capaz de gerenciar as trocas contextos decorrentes das mudanças de *frames* (devido às chamadas de procedimentos) durante a execução dos programas Python.
- Objetivos Específicos
  - Planejamento e implementação da arquitetura para o processador pipeline, de forma a permitir fácil integração de novas instruções, além de otimizar o conjunto de instruções suportado pela arquitetura PamPy (BITENCOURT, 2019);
  - Planejamento e implementação do protocolo e interface de comunicação com

o gerenciador de memória;

- Projeto e desenvolvimento de uma cache para armazenamento do contexto atual de execução do programa Python;

## 1.2 Organização do trabalho

O Capítulo 1 apresenta uma breve contextualização do tema deste trabalho, abordando de forma geral a relevância da linguagem Python e os seus desafios de desempenho. A partir disso, é apresentada a justificativa da pesquisa, a delimitação do problema e, ao final, os objetivos que norteiam o desenvolvimento deste projeto.

O Capítulo 2 descreve a metodologia de pesquisa, detalhando os procedimentos sistemáticos adotados para o desenvolvimento da arquitetura de hardware proposta, bem como para a subsequente análise e validação dos resultados obtidos.

O Capítulo 3 estabelece as bases conceituais do trabalho, iniciando com uma análise aprofundada da linguagem Python, abordando o processo de execução de bytecodes e as estratégias existentes para otimização de desempenho. Em seguida, realiza-se uma breve descrição do conjunto de instruções da *PVM* em foco neste trabalho. Por fim, apresenta-se uma revisão de trabalhos correlatos sobre a aceleração de máquinas virtuais em hardware, estabelecendo um paralelo técnico para a proposta deste trabalho.

O Capítulo 4 detalha o planejamento e a implementação prática da arquitetura proposta. Nesta seção são descritos o projeto do processador pipeline e as decisões de arquitetura adotadas.

O Capítulo 5 apresenta o ambiente de validação (*testbench*, relógios e geração de estímulos), as métricas adotadas e os resultados para as duas variantes arquiteturais (acesso externo e *cache* de dados *on-chip*). Em seguida realiza a comparação contra a execução em software (CPython 3.11), evidenciando os impactos gerados pela execução nativa de bytecodes e, por fim, sintetiza os resultados de síntese lógica (área, frequência máxima e potência).

O Capítulo 6 sintetiza os resultados alcançados neste TCC, retomando os objetivos definidos na introdução, avaliando em que medida foram atingidos com o desenvolvimento do projeto. Na sequência são discutidas as limitações do trabalho e, por fim, são sugeridas propostas para trabalhos futuros que possam dar continuidade a esta linha de pesquisa.

# 2 PROCEDIMENTOS METODOLÓGICOS ADOTADOS NESTE TCC

Em virtude do caráter sistemático inerente a toda pesquisa científica, nesta seção apresenta-se a abordagem metodológica adotada para a elaboração desta monografia. Inicialmente, são definidas e classificadas as correntes metodológicas que fundamentam este estudo e, em seguida, são descritas as etapas realizadas para o desenvolvimento do trabalho.

## 2.1 Tipo e abordagem de pesquisa

Neste trabalho, adota-se um caráter predominantemente experimental, com suporte a métodos exploratórios e descritivos, orientados para avaliação quantitativa de desempenho. O objetivo experimental justifica-se pela necessidade de implementar em HDL a microarquitetura proposta e medir, de forma controlada, métricas como frequência máxima de relógio e área física ocupada pelo processador. Ao mesmo tempo, incorpora-se uma fase exploratória inicial, baseada em levantamento bibliográfico, para mapear arquiteturas existentes e identificar lacunas no suporte nativo a instruções da PVM. Complementarmente, aspectos descritivos permitem documentar detalhadamente as características do conjunto de instruções da Máquina Virtual do Python e o protocolo de interação com a memória, servindo de base para a implementação.

O estudo experimental consiste na construção de um protótipo em System Verilog, sintetizado e simulado utilizando ferramentas de automação de design eletrônico (*EDA - Eletronic Design Automation*), de modo a submeter o design a estímulos controlados. Essa abordagem possibilita isolar variáveis de projeto e observar diretamente seus efeitos sobre a performance da arquitetura. Os resultados quantificáveis obtidos são, então, comparados com os resultados apresentados em trabalhos similares.

Paralelamente, o levantamento bibliográfico atua como etapa exploratória, utilizando bases de dados especializadas (*Google Acadêmico*, *IEEE Xplore* e *ACM Digital Library*) e palavras-chave como “Python acceleration”, “pipeline microarchitecture” e “hardware acceleration”. Essa pesquisa preliminar fundamenta as prioridades de implementação e formas de tratamento, orienta a definição das fases de pipeline e embasa a justificativa teórica para o desenvolvimento de um processador dedicado a PVM.

A pesquisa também é quantitativa, pois focaliza medições numéricas e análise estatística de desempenho. Entretanto, adota elementos de pesquisa qualitativa ao

contrastar os dados quantitativos com os objetivos e as motivações para este projeto, além das decisões de microarquitetura, configurando-se, assim, como um estudo de caráter misto. Essa combinação assegura que o trabalho não apenas apresente dados objetivos de performance, mas também discuta criticamente as implicações de design e viabilidade prática de um processador nativo para execução de bytecodes de Python.

## 2.2 Etapas do TCC

A seguir estão elencadas as etapas realizadas para o cumprimento do objetivo geral do projeto.

1. **Estudo e Definição da Arquitetura Pipeline**

   A etapa inicial do projeto focou no estudo de técnicas de *pipelining* para embasar o planejamento da arquitetura de um processador que execute os bytecodes do Python, partindo do conjunto de instruções do projeto PamPy (BITENCOURT, 2019). O estudo convergiu para a adoção de uma proposta com ênfase na modularidade e extensibilidade, permitindo a adição de instruções customizadas para a PVM.

2. **Levantamento Bibliográfico e Análise da PVM**

   Esta etapa consistiu em um levantamento bibliográfico sistemático e uma análise aprofundada da Máquina Virtual do Python para embasar as decisões de projeto. A revisão da literatura foi conduzida em bases de dados como IEEE Xplore, ACM Digital Library e Google Acadêmico, com foco em termos como “Python acceleration”, “pipeline microarchitecture” e “hardware acceleration”, e incluiu o estudo detalhado de trabalhos correlatos. Concomitantemente, a análise da PVM focou em seu funcionamento interno, baseando-se na obra de referência “Inside The Python Virtual Machine” (IKE-NWOSU, 2015) e no uso prático da ferramenta nativa *dis* (*Disassembler*) para investigar a estrutura e o comportamento dos bytecodes.

   O trabalho prático com o disassembler (dis) foi conduzido de forma experimental para mapear a correspondência entre construções de alto nível da Linguagem

Python e os *bytecodes* gerados para a PVM. Para isso, foram criadas funções isoladas que implementavam operações fundamentais, como cálculos matemáticos, atribuições de variáveis e laços de repetição. Cada uma dessas funções foi então passada como argumento para o módulo dis, que retornava a sequência exata de bytecodes que a máquina virtual executaria. Essa análise sistemática permitiu uma compreensão detalhada da semântica de cada instrução e do fluxo de dados na pilha de avaliação, fornecendo uma base empírica crucial para a definição do conjunto de instruções a ser implementado em hardware.

3. **Projeto e Protocolo de Interação com o gerenciador de Memória**

A execução de um programa baseia-se na comunicação com um gerenciador de memória externo (BENFICA, 2024), desenvolvido em C++, que simula o comportamento da Máquina Virtual Python através da administração de contextos de execução, ou *frames*. Cada *frame* encapsula o estado completo de uma função, incluindo suas constantes, variáveis e instruções. O gerenciador é responsável por instanciar esses *frames*, mantendo um objeto mapeado em software, cujo conteúdo é atualizado dinamicamente durante a execução de um programa pelo processador, através de escritas de retorno (*write-back*).

O fluxo de execução é iniciado pelo gerenciador, que envia o *frame* principal (*main*) para o processador, até então ocioso, e comunica os *frames* aninhados conforme requisição do processador. A comunicação, com excessão do frame *main*, é realizada por demanda do processador através de um protocolo *handshake* de quatro etapas. Ao concluir a execução de um *frame*, o processador solicita ativamente ao gerenciador o próximo *frame* necessário para a continuidade do programa. Este ciclo de requisição e execução se repete até o término da aplicação.

Com o objetivo de analisar o impacto da comunicação entre o GDM e o processador, foram desenvolvidas duas arquiteturas: (a) uma em que o acesso a dados ocorre durante a execução do *frame* pelo processador e (b) outra em que os dados são completamente carregados, juntamente com as instruções, antes do início da execução do contexto.

a. **Modelo de Interação com o GDM com acesso à dados externos por**

**demanda**

Após o pré-carregamento das instruções do *frame main* pelo GDM, leituras e escritas de variáveis são realizadas sob demanda, por meio de transações externas controladas através de um protocolo *handshake* de 4 etapas. Essa opção elimina a necessidade de área física dedicada ao armazenamento local de dados, porém cada acesso a dados impõe paralisação ao pipeline até a conclusão do protocolo, introduzindo gargalo de execução e variabilidade de latência.

b. **Arquitetura Alvo com Cache de Dados *On-Chip* e Revisão do Protocolo (carregar–executar–salvar)**

Para mitigar o *overhead* de comunicação exposto no acesso à dados por demanda, a arquitetura evolui para incorporar uma cache de dados *on-chip* (memória do tipo *scratchpad*) e adotar o paradigma *carregar–executar–salvar*. No início de cada *frame*, o GDM realiza o pré-carregamento em bloco de instruções e dados do *frame*; durante a execução, o processador realiza instruções de acesso à dados sobre a cache de dados interna, reduzindo paralisações. O mapeamento é linear, segmentado por escopo (deslocamentos para variáveis locais, globais etc.), com dados etiquetados para suportar tipagem dinâmica. Acrescenta-se um estágio de *Write-Back* ao pipeline para coordenar a escrita de resultados na memória local e a posterior gravação em bloco dos dados modificados de volta ao GDM ao término do *frame*. O protocolo físico é revisado para interoperar com a cache, preservando a confiabilidade do *handshake* e retirando-o do caminho crítico.

4. **Integração e Verificação da Arquitetura Otimizada**

Concluído o desenvolvimento, esta etapa dedicou-se à construção do ambiente de simulação e à verificação das arquiteturas com acesso a dados externos sob demanda e com cache de dados interna. O algoritmo *Bubble Sort*, por ser amplamente conhecido e utilizado no ensino de programação, foi empregado como caso de teste, escrito em bytecode Python suportado pelo processador e traduzido em estímulos de simulação.

A verificação contemplou: (i) checagem funcional por comparação com

resultados esperados; (ii) monitoramento do protocolo de *handshake* e do fluxo *carregar–executar–salvar*; e (iii) análise de métricas de desempenho — como ciclos totais para a execução de um *frame* (CPI efetivo) — empregadas para quantificar o impacto do subsistema de memória e do estágio de *Write-Back*.

O ambiente de teste utilizou um *testbench* com geração de estímulos, monitores e *scoreboards*, além de rastreamento temporal para depuração. Os resultados foram comparados entre as duas arquiteturas, evidenciando a redução do *overhead* de comunicação e a mitigação de paralisações no pipeline durante a execução.

## 5. Análise de Desempenho

A avaliação comparativa considera duas frentes: (a) métricas de síntese lógica e (b) métricas de execução em simulação, ambas aplicadas às arquiteturas *sem cache* e *com cache* de dados, com comparação adicional a um *benchmark* em software executando o mesmo caso de teste.

a. *Síntese lógica* - são reportadas: (i) **área** (células lógicas para ASIC), (ii) **frequência máxima** ($F_{max}$) e (iii) **potência** obtidas em síntese lógica.

b. *Simulação* - são medidos: (i) **ciclos totais por *frame*** e (ii) **CPI efetivo**, além da fração de ciclos em *stall* por acesso a dados. Os estímulos são idênticos para ambas as arquiteturas. O mesmo algoritmo é executado em software (*benchmark* de referência) para aferição de: (a) **tempo total** e (b) **instruções/ciclos** no processador hospedeiro.

Os resultados são organizados em: (i) tabelas de área, $F_{max}$ e potência em síntese, (ii) tabelas de ciclos/CPI. Todas as métricas consideram a sobrecarga do estágio de *Write-Back* e da lógica da cache na arquitetura otimizada.

# 3 REFERENCIAL TEÓRICO

Este capítulo apresenta a fundamentação teórica que embasa o presente trabalho, construindo uma análise que parte da relevância da Linguagem Python até a identificação de uma lacuna específica na área de aceleração de hardware para linguagens interpretadas como Python. Inicialmente, é detalhado o contexto de amplo emprego da Linguagem Python no cenário tecnológico, industrial e acadêmico, e as características de design que impulsionam sua popularidade. Na sequência, a discussão aprofunda-se nos aspectos técnicos de sua arquitetura de execução, explicando o funcionamento da Máquina Virtual do Python como uma máquina de pilha e detalhando os principais grupos de instruções (*bytecodes*) que governam seu comportamento. A partir dessa base, o capítulo aborda os desafios de desempenho inerentes a este modelo e analisa as diversas estratégias de otimização propostas pela comunidade, desde implementações alternativas de software, como compiladores JIT e AOT, até a delegação de carga para hardware especializado, como GPUs. Por fim, através da análise de trabalhos correlatos — com destaque para os processadores dedicados à Máquina Virtual do Java —, demonstra-se a viabilidade e o potencial de aplicação de estratégias de aceleração análogas para Python. Esta revisão culmina na identificação de uma clara lacuna de pesquisa para a PVM, estabelecendo a justificativa e o alicerce conceitual para a arquitetura proposta neste trabalho.

## 3.1 Contextualização

A Linguagem Python, criada por Guido van Rossum e lançada em 1991, foi projetada com uma filosofia que prioriza a legibilidade e a simplicidade, conforme estabelecido no “Zen do Python” (PETTERS, 2004), permitindo que desenvolvedores expressem ideias complexas de forma concisa. Essa abordagem se provou bem-sucedida, levando a linguagem a uma posição de domínio no cenário tecnológico atual, sendo reconhecida como uma linguagem de uso geral. A sua proeminência é confirmada de forma quantitativa pelos principais índices da indústria: o Índice TIOBE (TIOBE Software, 2025) de maio de 2025 aponta o Python com uma participação de mercado recorde, liderando com mais de 15% de vantagem sobre o segundo colocado. Essa liderança é corroborada pelo ranking da IEEE Spectrum (CASS, 2024), que o considera a linguagem mais popular desde 2017, e pela pesquisa do Stack Overflow (Stack Overflow, 2024), que o identifica como a tecnologia mais “desejada” e “admirada” pela comunidade

de desenvolvedores.

Esse domínio não é acidental, mas fruto de um conjunto de características que a tornam poderosa, flexível e acessível. A seguir, as principais características da linguagem (CUTTING; STEPHEN, 2021; SAABITH; VINOTHRAJ; FAREEZ, 2020) são apresentadas:

- **Fácil Aprendizagem e Utilização**: A sintaxe do Python é considerada simples e direta, muito semelhante à Língua Inglesa. O código é fácil de ler e entender, e como a indentação define os blocos de código, não há necessidade de usar ponto e vírgula ou chaves. Por essas razões, é frequentemente recomendada como a primeira linguagem de programação para iniciantes.

- **Linguagem Expressiva**: Python é capaz de executar tarefas complexas com poucas linhas de código em comparação com outras linguagens. Um exemplo clássico é o programa "Hello World", que requer apenas uma única linha para ser executado.

- **Linguagem Interpretada**: O código Python é executado linha por linha, em vez de ser compilado de uma só vez. Essa abordagem torna o processo de depuração de erros muito mais fácil e eficiente. No entanto, essa característica também é um dos fatores que torna a execução do Python mais lenta que a de linguagens compiladas.

- **Orientação a Objetos**: A linguagem suporta plenamente a programação orientada a objetos. Isso ajuda os programadores a modelar entidades do mundo real e a escrever código reutilizável através de conceitos como herança e encapsulamento, permitindo o desenvolvimento de aplicações com menos código.

- **Tipagem Dinâmica**: O tipo de dado de uma variável não precisa ser especificado ao declará-la. O tipo é determinado em tempo de execução, quando um valor é atribuído à variável. Embora isso ofereça grande flexibilidade ao programador, exige que a máquina realize trabalho adicional, o que contribui para o desempenho reduzido da linguagem.

- **Extenso Repositório de Bibliotecas**: Python disponibiliza uma vasta gama de bibliotecas e módulos para os mais diversos campos, como aprendizado de máquina, desenvolvimento *web* e *scripting*. Isso permite que os desenvolvedores importem funcionalidades prontas em vez de ter que escrevê-las do zero.

- **Portátil e Multiplataforma**: Um programa escrito em Python pode ser executado da mesma forma em diferentes plataformas, como *Windows*, *Linux*, *UNIX* e *Macintosh*. Isso permite o desenvolvimento de software para múltiplos sistemas escrevendo o código apenas uma vez.

- **Extensível e Embarcável**: A propriedade de extensibilidade do Python permite que código escrito e compilado em outras linguagens, como C ou C++, seja utilizado dentro de um programa Python. De forma complementar, a propriedade de ser embarcável permite que o código Python seja integrado e executado dentro de aplicações escritas em outras linguagens.

- **Livre e de Código Aberto**: Python é totalmente gratuito e de código aberto, o que significa que seu código-fonte pode ser baixado, modificado e distribuído livremente. Essa abertura fomentou uma grande comunidade global que contribui ativamente para o desenvolvimento contínuo da linguagem e de suas bibliotecas.

Essa combinação de atributos impulsionou sua adoção em uma vasta gama de domínios, tornando-o cada vez mais presente no desenvolvimento *web*, desenvolvimento de jogos, e até mesmo em visão computacional e *web scraping*, além de ser a escolha padrão em áreas como a Inteligência Artificial e Ciência de Dados (RAYHAN, 2023). A sua relevância em infraestruturas de alta escala é demonstrada pela sua aplicação em componentes críticos em empresas como a Netflix, que a utiliza para gerenciar sua rede de entrega de conteúdo; Google, onde Python é uma das linguagens centrais e base para sua equipe de dados; e Facebook/Instagram, que a emprega na manutenção de sua infraestrutura e em aplicações de Inteligência Artificial.

Entretanto, as mesmas características que impulsionam a produtividade e a flexibilidade do Python são também a origem de seus principais desafios de desempenho. A chave para entender essa dualidade reside em sua arquitetura de execução como uma linguagem interpretada (WATTERS; ROSSUM; AHLSTROM, 1996).

## 3.2 Arquitetura e ambiente de execução do Python

A implementação de referência do Python, o CPython (CANNON, 2005), opera em um modelo de execução em duas etapas. Primeiramente, um compilador traduz o código-fonte para um formato intermediário e portátil chamado *bytecode*. Em seguida,

esses *bytecodes* são processados não pelo hardware nativo, mas por um componente de software conhecido como interpretador. O interpretador, por sua vez, funciona como um processador simulado que gerencia a execução das instruções, traduzindo-as em tempo de execução para as instruções nativas da máquina hospedeira (KWAME EZEKIEL MENSAH MARTEY, 2017).

A forma de funcionamento do interpretador se alinha à definição formal dada pelo *Dictionary of Computer Science, Engineering, And Technology* de que uma "Máquina virtual é um processo em um sistema multitarefa que se comporta como um computador independente, e não como parte de um sistema maior" (LAPLANTE et al., 2017). Sendo assim, o interpretador do Python é amplamente conhecido como Máquina Virtual do Python (*PVM - Python Virtual Machine*).

Essa arquitetura de máquina virtual é a chave para a portabilidade do Python, permitindo que o mesmo programa em *bytecode* rode em diferentes plataformas como Windows e Linux, desde que haja uma versão compatível da PVM em execução. Contudo, essa mesma camada de abstração é a raiz de seu principal desafio de desempenho. Como cada instrução precisa ser interpretada pela PVM antes de ser executada pelo hardware, cria-se uma sobrecarga computacional inerente que torna a linguagem mais lenta em comparação com alternativas compiladas (O'CONNOR; TREMBLAY, 1997).

Devido à sobrecarga de desempenho resultante do modelo de execução da Máquina Virtual do Python, diversas implementações alternativas ao interpretador da linguagem foram desenvolvidas, cada uma adotando estratégias próprias para otimizar a execução do código. Entre as mais relevantes, destacam-se:

- **Jython** (The Jython Project, 2025): Uma implementação da linguagem Python que compila o código-fonte para *bytecodes* Java. Isso permite que os programas rodem sobre a Máquina Virtual do Java (*JVM - Java Virtual Machine*), garantindo acesso direto e interoperabilidade com o vasto ecossistema de classes e bibliotecas da plataforma Java.

- **PyPy** (The PyPy Team, 2025): Uma implementação de alta performance focada em velocidade, sendo um dos projetos de compilação em tempo de execução (*JIT - Just-In-Time*) mais ativos para o Python. Sua principal característica é o uso de um compilador JIT, que analisa o código em execução e traduz os trechos mais utilizados para código da máquina nativa. Adicionalmente, o PyPy não possui o Bloqueio Global do Interpretador (*GIL - Global Interpreter Lock*), o que lhe

permite executar múltiplas threads em verdadeiro paralelismo, superando uma das principais limitações de concorrência do CPython.

- **IronPython** (The IronPython Team, 2025): Uma implementação de código aberto fortemente integrada com a plataforma .NET. O IronPython permite a utilização de bibliotecas .NET e Python de forma intercambiável, além de possibilitar que outras linguagens .NET utilizem código Python com a mesma facilidade.

- **Mamba** (PEREIRA; AYCOCK, 2002): Um projeto que propôs uma abordagem minimalista e baseada em registradores para a máquina virtual, em uma filosofia análoga à transição de processadores CISC para RISC. Diferente da PVM padrão, que é baseada em pilhas, o Mamba opera com um conjunto de instruções muito reduzido, onde os operandos são explicitamente nomeados em registradores virtuais.

- **Falcon** (POWER; RUBINSTEYN, 2013): Um interpretador de *bytecodes* de alta performance, projetado para ser totalmente compatível com o CPython e suas extensões em C. Diferente de outras alternativas, o Falcon não substitui o interpretador padrão, mas opera de maneira conjunta dele. Sua abordagem principal envolve a conversão de um subconjunto de funções do *bytecode* padrão, baseado em pilhas, para um formato otimizado baseado em registradores e então executado pela máquina virtual Falcon.

Uma das abordagens proeminentes para mitigar os desafios de desempenho do Python é a compilação em tempo de execução. Diferente de um interpretador puro, um sistema JIT monitora o código enquanto ele é executado e traduz dinamicamente os trechos mais utilizados para código da máquina nativa. Esse código nativo é então armazenado em cache para reutilização (NEWHALL; MILLER, 1998), eliminando a sobrecarga da interpretação para as partes críticas do programa. Como elencado anteriormente, o projeto PyPy é o principal expoente dessa técnica no ecossistema Python, oferecendo ganhos de velocidade significativos sobre a implementação CPython, sendo até 3 vezes mais rápido do que a implementação padrão.

Apesar de sua eficácia, a abordagem JIT possui seus próprios desafios e limitações. Primeiramente, ela introduz uma latência inicial, pois o sistema precisa de um tempo para identificar os caminhos críticos e realizar a primeira compilação. Em segundo lugar, como a compilação ocorre em tempo real, há um limite para a complexidade das

otimizações que podem ser aplicadas (MULLER et al., 1997), pois uma compilação lenta prejudicaria o desempenho geral. Por fim, os compiladores JIT exigem uma quantidade considerável de memória para armazenar o código compilado e os dados de perfilamento, tornando-os inadequados para sistemas com recursos limitados, como dispositivos embarcados (SCHOEBERL, 2008).

Uma abordagem alternativa à interpretação (seja ela pura, como no CPython, ou JIT, como no PyPy) é a compilação estática, também conhecida como *Ahead-of-Time* (*AOT*) ou compilação cruzada. Nesta estratégia, o código Python é traduzido diretamente para o conjunto de instruções da arquitetura (*ISA - Instruction Set Architecture*) de hardware alvo antes da execução (LAPLANTE et al., 2017). O resultado é um executável nativo que, em teoria, elimina a sobrecarga da máquina virtual e pode alcançar um desempenho superior.

No entanto, essa busca por desempenho impõe uma série de compromissos que limitam sua aplicabilidade prática no ecossistema Python. O primeiro e mais evidente é a perda de portabilidade. O código compilado nativamente é específico para uma única arquitetura de processador e sistema operacional, anulando uma das principais vantagens da linguagem. Adicionalmente, esta abordagem gera um alto custo de manutenção, pois o compilador precisa ser constantemente atualizado para acompanhar a rápida evolução das arquiteturas de hardware.

Outra limitação crítica é a perda de flexibilidade. Em ambientes dinâmicos, onde o código pode ser alterado frequentemente, a necessidade de um ciclo completo de recompilação para cada mudança é impraticável e interrompe a agilidade característica do desenvolvimento em Python.

Por fim, a compilação estática enfrenta um desafio fundamental de paradigma. A natureza altamente dinâmica e orientada a objetos do Python não se traduz de forma eficiente para as arquiteturas de processadores tradicionais, que não possuem suporte nativo para conceitos como herança ou polimorfismo (DYER; CHAUHAN, 2022).

Além das alternativas de implementação do interpretador, outras estratégias para melhorar o desempenho de aplicações Python focam em delegar a carga computacional para ambientes mais potentes ou especializados.

A computação em nuvem oferece um modelo flexível e escalável para a execução de código Python, com acesso sob demanda a recursos computacionais fornecidos remotamente (SURBIRYALA; RONG, 2019). Nesta modalidade, a complexidade da manutenção da infraestrutura local é abstraída, e os custos são associados diretamente

ao consumo.

Outra estratégia adotada é a delegação de tarefas computacionalmente intensivas para hardware especializado, principalmente as unidades de processamento gráfico (*GPUs - Graphics processing units*). Essa abordagem é padrão em domínios como simulações científicas (SCHOENHOLZ; CUBUK, 2019; BROUGH; WHEELER; KALIDINDI, 2017), aprendizado de máquina (MAIER et al., 2020) e *deep learning* (DOGARU; DOGARU, 2017), onde o paralelismo massivo das GPUs pode acelerar drasticamente os cálculos numéricos.

Todavia, a aceleração por GPU não é uma solução universal e introduz sua própria camada de complexidade. Primeiramente, exige do desenvolvedor habilidades avançadas em programação paralela. Além disso, a transferência de dados entre a memória da unidade central de processamento (*CPU - Central processing unit*) e da GPU frequentemente se torna um gargalo, podendo anular os ganhos de desempenho se o tempo de transferência for significativo (FERREIRA; MAGALHÃES; NACIF, 2019), problema quem em parte foi solucionado com as GPUs integradas diretamente aos processadores. Finalmente, a abordagem é limitada por questões de compatibilidade de hardware e software, alto consumo de energia e custo elevado, sendo inadequada para algoritmos de natureza inerentemente sequencial (MITTAL; VETTER, 2014).

Tanto a computação em nuvem quanto a aceleração por GPU representam soluções avançadas que contornam o problema de desempenho do Python ao invés de resolvê-lo em sua origem. Elas delegam a carga de trabalho para ambientes externos, mas não otimizam o fluxo de execução da própria Máquina Virtual do Python. Esse cenário evidencia a necessidade de explorar abordagens que atuem diretamente no núcleo do interpretador, como a aceleração por hardware dedicado à PVM.

Este princípio de aceleração por hardware é um conceito bem estabelecido que postula que arquiteturas físicas dedicadas executam algoritmos de forma mais eficiente do que implementações em software rodando em processadores de propósito geral (GREVERA; UDUPA; ODHNER, 2000). Essa eficiência superior advém da eliminação das camadas de abstração de software e da especialização das instruções para a tarefa em questão, o que reduz a latência e a sobrecarga computacional (BITENCOURT, 2019).

Apesar da clara vantagem da aceleração por hardware, constata-se uma notável lacuna na aplicação deste princípio para otimizar diretamente a execução da Máquina Virtual do Python. É verdade que o ecossistema Python se beneficia indiretamente da aceleração por hardware, mas isso ocorre através de um mecanismo de delegação. As

principais bibliotecas de computação científica e aprendizado de máquina, como NumPy, Pandas e Scikit-learn, são, em grande parte, invólucros para código de alta performance implementado em C/C++.

Essas bases de código em C/C++ podem, por sua vez, ser otimizadas para utilizar aceleradores de hardware como GPUs. Contudo, esta é uma aceleração do código da biblioteca, e não do Python em si. O código Python que orquestra essas chamadas — como laços de repetição, preparação de dados e a orquestração das chamadas — continua sendo executado e limitado pela sobrecarga do interpretador da PVM.

A distinção entre aceleração direta e indireta fica clara ao se comparar com a linguagem Java. Assim como o Python, Java é orientada a objetos e utiliza uma máquina virtual para executar bytecodes. No entanto, o ecossistema Java possui um relevante histórico de projetos de pesquisa focados no desenvolvimento de processadores dedicados à execução de seu *bytecode*, os quais serão discutidos na seção de trabalhos relacionados, validando a viabilidade da aceleração direta da máquina virtual.

A maturidade dessas arquiteturas para Java, em contraste com a dependência do Python em acelerar componentes críticos transcrevendo-os em C++, acentua a carência de iniciativas análogas para a PVM. A principal justificativa deste trabalho reside em endereçar essa lacuna, através da proposta de uma arquitetura de hardware dedicada à execução nativa de bytecodes, tendo como alvo o próprio modelo computacional da PVM.

O núcleo da Máquina Virtual Python opera com base em duas estruturas de pilha distintas, porém interligadas: a pilha de chamadas, que gerencia os contextos de função, e a pilha de avaliação, que serve como área de trabalho para a execução das operações. Enquanto a pilha de chamadas organiza a sequência hierárquica de funções ativas, a pilha de avaliação é onde as instruções de *bytecode* efetivamente manipulam os dados.

A execução de um programa é estruturada pela pilha de chamadas. A cada nova chamada de função, um *frame* de execução — contendo o escopo privado da função, como suas variáveis locais e referências — é criado e empilhado. Isso garante que cada função opere em seu próprio contexto. Embora os *frames* possam acessar escopos compartilhados (como variáveis globais), é importante notar que o GIL na implementação CPython impede a execução paralela de múltiplas threads, serializando a execução de bytecodes.

Dentro do contexto fornecido pelo *frame* no topo da pilha de chamadas, as operações de bytecode são realizadas na pilha de avaliação. Trata-se de uma estrutura de dados do tipo LIFO (*Last-In-First-Out*), onde operandos (como constantes e variáveis) são

empilhados para serem consumidos pelas instruções. Uma instrução típica desempilha seus operandos, executa a operação e empilha o resultado de volta. A manipulação precisa desta pilha, controlada por um ponteiro para seu topo (*TOS - Top of Stack*), é fundamental para a consistência das operações aritméticas, lógicas e de controle de fluxo do Python (IKE-NWOSU, 2015).

O suporte ao paradigma de orientação a objetos é uma das características centrais da linguagem Python. Em seu modelo de execução, praticamente todos os elementos — desde números e strings até funções e classes — são tratados como objetos. A Máquina Virtual do Python é a entidade responsável por orquestrar o ciclo de vida e a interação desses objetos, traduzindo os conceitos de alto nível do paradigma em operações concretas executadas a partir de seu conjunto de instruções (*bytecode*).

O processo de criação de objetos na PVM ocorre em duas fases distintas. Primeiramente, a definição de uma classe no código-fonte leva à construção de um "objeto-classe", que funciona como um molde ou protótipo. Este objeto-classe encapsula a estrutura e os comportamentos (métodos) definidos pelo programador. Em um segundo momento, a instanciação ocorre quando este objeto-classe é invocado, resultando na alocação de memória para um novo objeto de instância e na execução de seu construtor para inicializar seu estado.

A interação com os objetos é inerentemente dinâmica. O acesso a atributos e a chamada de métodos não são resolvidos em tempo de compilação, mas sim em tempo de execução pela PVM. Esse mecanismo é o que possibilita o polimorfismo: quando um método é invocado em um objeto, a PVM realiza uma busca dinâmica através da hierarquia de herança do objeto para encontrar a implementação correta a ser executada. Essa flexibilidade, embora poderosa, implica uma complexidade significativa no fluxo de execução, com múltiplas indireções e verificações de tipo ocorrendo dinamicamente.

Essa natureza dinâmica e centrada em objetos da PVM representa um desafio fundamental para uma implementação direta em hardware. Arquiteturas de processadores tradicionais são otimizadas para operações com tipos de dados de tamanho fixo, registradores e acesso previsível à memória — um modelo que contrasta fortemente com a gestão de objetos de tamanho variável, herança e resolução de atributos em tempo de execução. Uma tradução literal dessa semântica para um circuito de hardware resultaria em uma arquitetura excessivamente complexa e ineficiente. Diante disso, uma abordagem mais pragmática consiste em desacoplar as responsabilidades: projetar um processador pipeline focado na execução de operações primitivas e delegar a gestão de alto nível do

contexto dos objetos — como alocação, inicialização e resolução de atributos — a um gerenciador de memória dedicado, que atuaria como um coprocessador especializado.

### 3.2.1 Conjunto de instruções

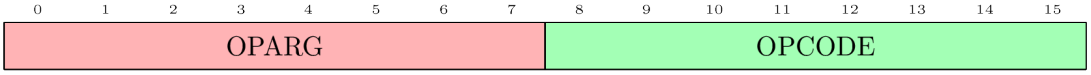
Como mencionado, o código Python não é executado diretamente, mas compilado para um formato intermediário chamado *bytecode*. Este *bytecode* é um conjunto de instruções de baixo nível projetado especificamente para a PVM.

É importante notar que estruturas de alto nível, como laços de repetição (*for*, *while*) ou mesmo chamadas de função, não possuem uma tradução direta para uma única instrução de *bytecode*. Em vez disso, o compilador as decompõe em uma sequência de instruções mais simples. Essa filosofia de design, que favorece instruções atômicas, simplifica a implementação da PVM e otimiza a performance, evitando os gargalos que poderiam ser causados por instruções excessivamente complexas. A ferramenta **dis** (*Disassembler*), nativa do Python, permite inspecionar a sequência de *bytecodes* gerada a partir de um código-fonte, tornando esse processo de decomposição observável.

Cada instrução (*bytecode*) da PVM é representada por uma unidade de 16 bits (2 bytes), dividida em dois campos principais, como na figura 1:

- **OpCode (8 bits)**: o código da operação, que define a ação a ser realizada (ex: LOAD_CONST, BINARY_OP).
- **OpArg (8 bits)**: o argumento da operação, que fornece um dado para a execução da operação descrita pelo *opcode*, como o índice de uma constante a ser carregada ou o endereço para uma instrução de desvio.

Figura 1 – Formato de uma instrução (2 bytes)

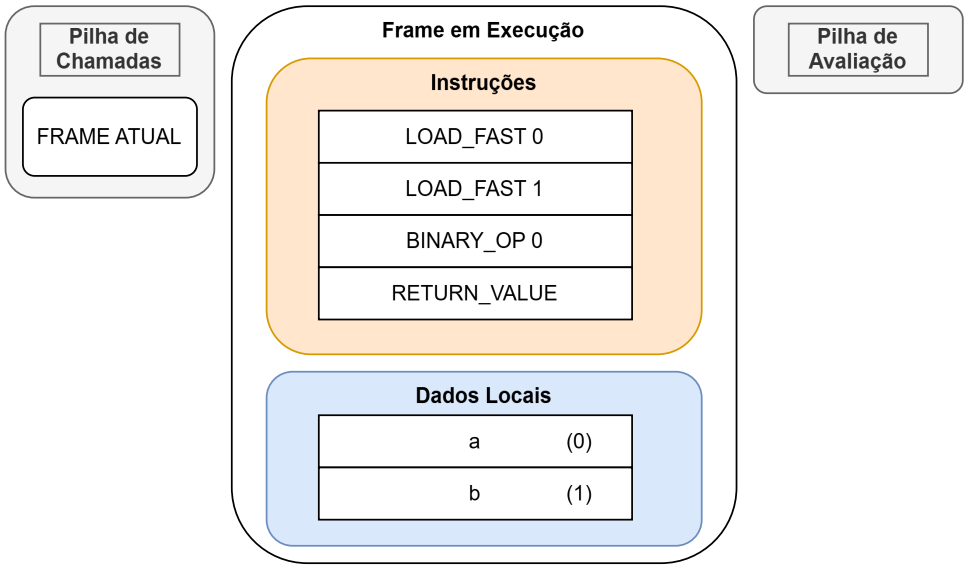



Fonte: (IKE-NWOSU, 2015)

Para ilustrar o funcionamento da máquina de pilha, considere a seguinte instrução Python: resultado = a + b. O compilador geraria uma sequência de bytecodes similar à seguinte, e a pilha de execução seria modificada a cada passo:

1. Carregamento do *frame*: O *frame* a ser executado, contendo a lista de instruções e variáveis, é carregado para a pilha de chamadas, como na figura 2.

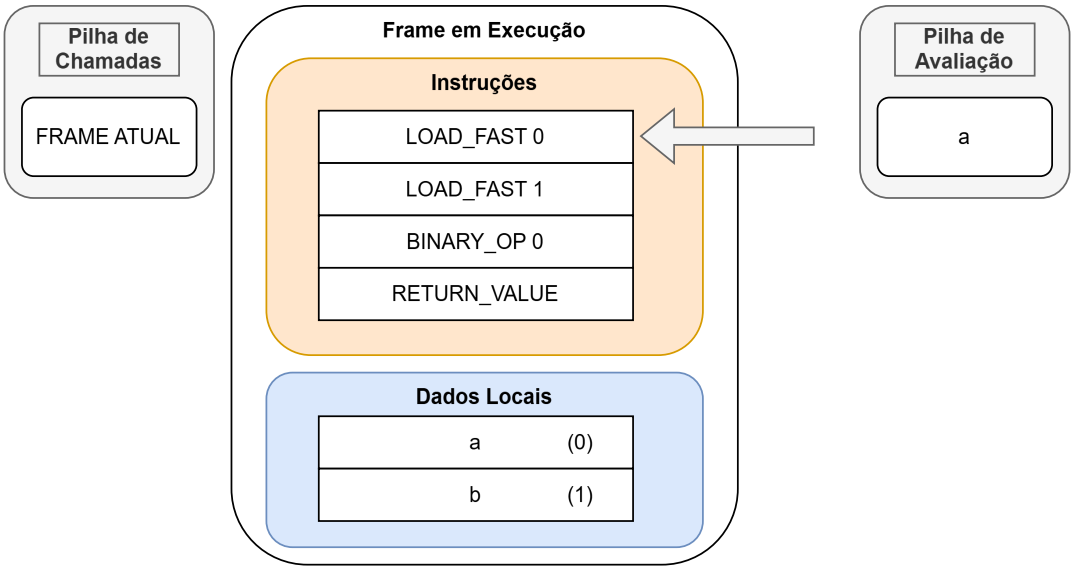
Figura 2 – *Frame* atual




Fonte: (Autor, 2025)

2. LOAD_FAST 0: O valor da variável local de índice 0 (a) é carregado e empilhado na pilha de avaliação, como na figura 3.

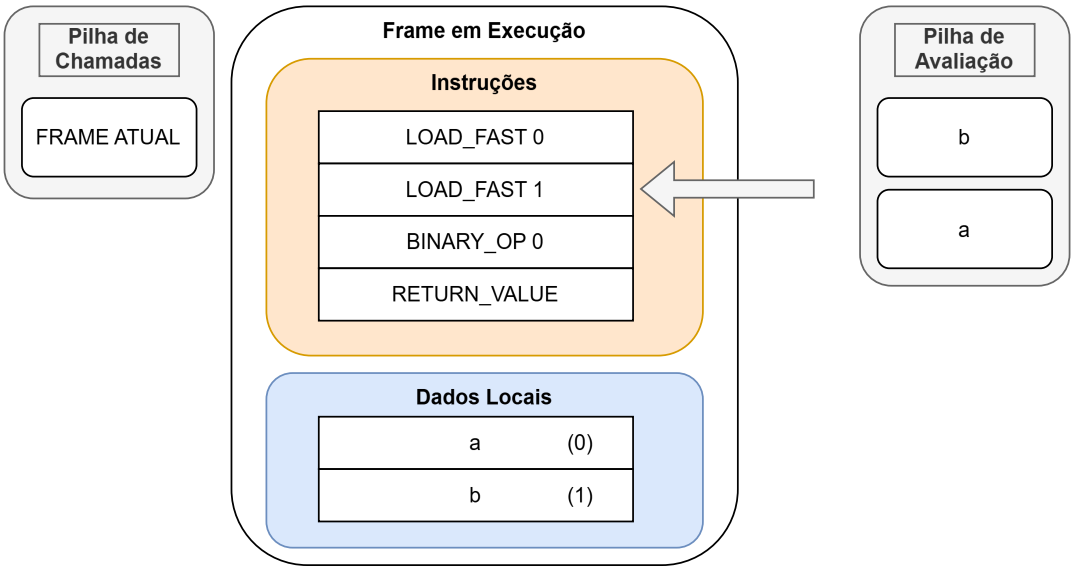
Figura 3 – Carregamento da variável local de índice 0




Fonte: (Autor, 2025)

3. LOAD_FAST 1: O valor da variável local de índice 1 (b) é carregado e empilhado na pilha de avaliação, como na figura 4.

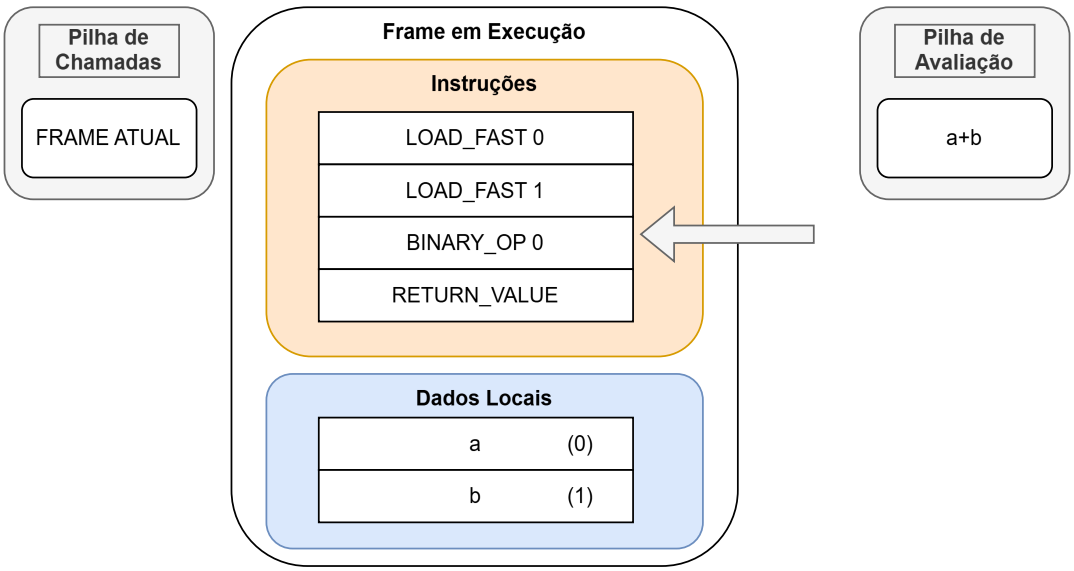
Figura 4 – Carregamento da variável local de índice 1




Fonte: (Autor, 2025)

4. BINARY_OP (0 - ADD): A instrução desempilha os dois valores do topo da pilha de avaliação (b e a), realiza a soma e empilha o resultado na pilha de avaliação, como na figura 5.

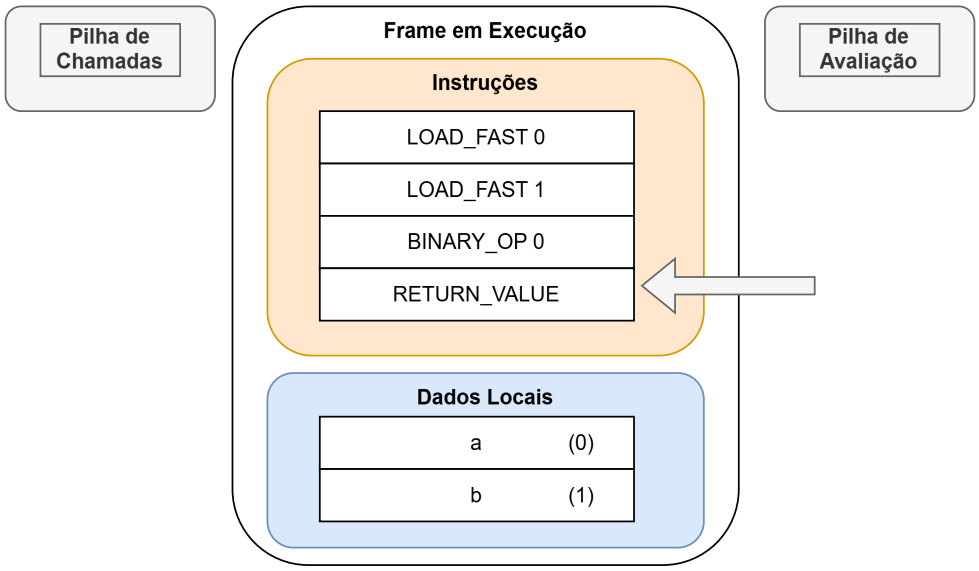
Figura 5 – Soma dos dois valores do topo da pilha de avaliação e reescrita do resultado




Fonte: (Autor, 2025)

5. RETURN_VALUE resultado: O valor no topo da pilha (soma) é desempilhado e devolvido para o GDM, como na figura 6.

Figura 6 – Retorno do valor no topo da pilha de avaliação




Fonte: (Autor, 2025)

As versões mais recentes do Python contam com mais de 200 instruções de *bytecodes* mapeadas. Embora atuem em diversos contextos, essas instruções operam sempre sobre a pilha de avaliação. Os grupos ou classes de instruções que desempenham um papel fundamental e de alicerce para a linguagem são: manipulação de pilha, operações aritméticas e lógicas, controle de fluxo, chamadas de função e manipulação de variáveis em memória, sendo estes os essenciais para o correto tratamento das funções mais básicas da linguagem (ROSSUM; DRAKE, 2009).

### 3.2.2 Instruções para manipulação de variáveis em memória

Este grupo de instruções é responsável pela movimentação de dados do *frame* atual, em execução, para a pilha de avaliação e vice-versa. Para que qualquer cálculo seja realizado, os operandos devem ser primeiro carregados na pilha. As instruções principais para esta tarefa são:

- **Instruções de carga** (*LOAD_CONST, LOAD_FAST, LOAD_GLOBAL, LOAD_NAME*): carregam um valor, seja uma constante ou o conteúdo de uma variável, do seu respectivo escopo para o topo da pilha de avaliação.
- **Instruções de armazenamento** (*STORE_FAST, STORE_GLOBAL, STORE_NAME*): realizam a operação inversa. Elas removem o valor que

está no topo da pilha de avaliação e o associam a um nome de variável no respectivo escopo.

- **Instruções de exclusão** (*DELETE_FAST, DELETE_GLOBAL, DELETE_NAME*): estas instruções removem a referência de uma variável em seu escopo. É crucial entender que elas não excluem o objeto da memória de forma imediata. Em vez disso, removem a ligação entre o nome e o objeto. Seu principal propósito está na gestão de memória: se a referência removida era a última existente para um objeto, seu contador de referências chega a zero, tornando-o elegível para ser desalocado pelo coletor de lixo. Adicionalmente, servem para gerenciar o escopo de forma explícita, prevenindo o uso acidental de uma variável que não é mais necessária ou válida no fluxo do programa.

### 3.2.3 Instruções para manipulação da pilha

Além da carga e armazenamento, um conjunto de instruções é dedicado a manipular diretamente a organização dos elementos na pilha de avaliação, otimizando o acesso aos dados para operações subsequentes.

- **Instruções de desempilhamento** (*POP_TOP*): descartam o valor que está no topo da pilha.
- **Instruções de rotação** (*ROT_TWO, ROT_THREE, ROT_FOUR*): reorganizam os elementos no topo da pilha, alterando sua ordem sem a necessidade de acessos à memória.
- **Instruções de duplicação** (*DUP_TOP*): duplicam o valor no topo da pilha.

### 3.2.4 Instruções aritméticas e lógicas

Estas instruções realizam cálculos e operações lógicas, operando implicitamente sobre os elementos no topo da pilha e substituindo os operandos pelo resultado.

- **Operações Binárias** (*BINARY_ADD, BINARY_SUBTRACT, BINARY_MULTIPLY, BINARY_AND, BINARY_XOR*, etc.): desempilham os dois valores do topo, realizam a operação correspondente e empilham o resultado.

- **Operações Unárias** (*UNARY_NOT, UNARY_NEGATIVE*): operam sobre o único valor no topo da pilha e o substituem pelo resultado (ex: negação lógica ou inversão de sinal).

### 3.2.5 Instruções para controle de fluxo

Essenciais para a implementação de laços e condicionais, estas instruções alteram a ordem sequencial de execução, modificando o contador de programa (PC).

- **Instrução de Comparação** (*COMPARE_OP*): desempilham dois valores, realiza uma comparação atrelada do *oparg* informado (ex: <, ==, >) e empilha o resultado booleano (Verdadeiro ou Falso).

- **Saltos Condicionais** (*POP_JUMP_IF_FALSE, POP_JUMP_IF_TRUE*): desempilham um valor booleano e, com base nele, decidem se devem ou não desviar a execução para outro ponto do código.

- **Saltos Incondicionais** (*JUMP_ABSOLUTE, JUMP_FORWARD*): realizam desvios obrigatórios no fluxo de execução para um novo endereço.

### 3.2.6 Instruções para chamada de função

Este grupo gerencia a criação e exclusão dos frames de execução na pilha de chamadas, controlando a execução modularizada do código.

- **Instrução de chamada** (*CALL_FUNCTION*): cria um novo *frame* de execução, utilizando os valores no topo da pilha de avaliação como argumentos para a função que está sendo chamada.

- **Instrução de retorno** (RETURN_VALUE): encerra o frame de execução atual, remove-o da pilha de chamadas e empilha o valor de retorno na pilha de avaliação do *frame* requerente.

## 3.3 Processadores em hardware para linguagens OO baseadas em pilha

A presente subseção analisa os trabalhos relacionados que fundamentam esta pesquisa, com os objetivos de contextualizar o estado da arte e identificar metodologias consolidadas para a aceleração de linguagens orientadas à objetos em hardware. O critério central para a seleção dos trabalhos foi a abordagem de implementar suporte nativo em hardware para a execução de bytecodes da linguagem de referência, em vez de estratégias que apenas traduzem ou substituem trechos de código, como as técnicas de JIT e AOT já mencionadas anteriormente.

A busca por pesquisas correlatas revelou uma notável escassez de trabalhos que endereçam especificamente a aceleração de hardware para a Máquina Virtual Python nos moldes propostos aqui. Diante da ausência de artigos tratando exatamente do mesmo tema, a estratégia adotada foi explorar projetos para uma linguagem conceitualmente similar: o Java. A Máquina Virtual Java compartilha muitas características com a PVM, entre as quais a implantação do paradigma da pilha e orientação a objetos torna as soluções para uma, em grande parte, aplicáveis à outra. A partir de trabalhos fundacionais encontrados para Java, foi realizado um levantamento de suas referências para delimitar a análise aos projetos com maior sinergia.

Essa análise de arquiteturas para Java mostra-se diretamente aplicável, pois as técnicas de otimização empregadas para a JVM — como caches de pilha dedicadas e modelos de execução híbridos (hardware/microcódigo) — resolvem desafios de arquitetura que são fundamentalmente os mesmos na PVM e, portanto, podem ser adaptadas. Diferenças importantes também existem, como a tipagem dinâmica, que no caso do Java é estática e explícita, enquanto que no Python é dinâmica, sendo estes aspectos também preponderantes tratados no escopo do presente estudo.

### 3.3.1 Hot & Spicy

Hot & Spicy é uma infraestrutura de código aberto (*Open-Source*) com um conjunto de ferramentas para a integração de aceleradores em FPGA em aplicações Python (SKALICKY et al., 2018). As ferramentas facilitam o empacotamento e integração de seus drives baseados nas linguagens C/C++ que podem ser acionados por aplicações em Python. São quatro as funcionalidades das ferramentas: (1) Traduzir funções em Python para funções em C/C++ que podem ser transcritas em HLS

(*High-Level Synthesis*); (2) Gerar interfaces de ligação entre código C e Python; (3) Automatizar o fluxo da ferramenta de EDA para FPGA; (4) Redirecionar o código fonte Python para utilizar bibliotecas de aceleração em C/C++.

A infraestrutura foi projetada para a aceleração seletiva de trechos de código em aplicações Python, partindo do princípio de que o desempenho geral é majoritariamente determinado por um pequeno conjunto de funções que constituem o caminho crítico. Ao focar a otimização nesses gargalos, obtêm-se os maiores ganhos de performance. As quatro funcionalidades do Hot & Spicy integram-se para oferecer um fluxo completo para essa finalidade. Em um teste com o algoritmo *Canny Edge Detector*, a metodologia demonstrou uma aceleração de 39.137 vezes em comparação com a execução puramente em software.

Apesar de sua relevância em demonstrar o potencial da aceleração por hardware, é fundamental notar que a abordagem do Hot & Spicy não acelera a Máquina Virtual do Python diretamente. Em vez disso, ele segue a estratégia de tradução de código: o trecho crítico em Python é convertido para C/C++, e é esta versão em C/C++ que é sintetizada em hardware. Essa metodologia, embora eficaz para caminhos críticos específicos, desvia-se da semântica de execução padrão da PVM e não otimiza o código Python que gerencia a aplicação. Ela contorna o problema de desempenho do interpretador ao substituir partes do código, em vez de propor uma solução para acelerar a execução dos bytecodes de forma nativa. Desta forma, o Hot & Spicy serve como um contraponto que reforça o potencial para exploração de um novo espaço de projeto, conforme previsto nos objetivos para o presente trabalho, cujo foco é a aceleração direta da PVM.

### 3.3.2 picoJava-I

O projeto picoJava-I, desenvolvido pela Sun Microsystems, é uma arquitetura de microprocessador que executa bytecodes da Máquina Virtual Java diretamente em hardware (MCGHAN; O'CONNOR, 1998; O'CONNOR; TREMBLAY, 1997). Essa abordagem foi projetada para superar as limitações das soluções de software, como a sobrecarga de interpretadores ou a alta demanda de memória de compiladores JIT, visando oferecer alto desempenho em sistemas de baixo custo e com recursos limitados, como os do mercado embarcado.

O núcleo da arquitetura é um pipeline de quatro estágios (Busca, Decodificação, Execução e Escrita), semelhante aos processadores RISC. Uma das otimizações mais

impactantes ocorre no estágio de Decodificação através do mecanismo de *folding*, que combina instruções sequenciais comuns — como um *load* de uma variável local seguido por uma operação que a consome. Essa técnica se mostrou extremamente eficaz, eliminando, em média, 28% do total de instruções que precisariam ser executadas. Para gerenciar a complexidade de um conjunto de instruções com mais de 300 opcodes, o picoJava-I adota uma estratégia de execução em três níveis: instruções simples são executadas em um único ciclo (hardwired), as moderadamente complexas são tratadas via microcódigo, e as mais raras ou que dependem do sistema operacional são emuladas em software, o que em parte também é a estratégia adotada no presente trabalho, ao relegar para GDM em software o gerenciamento das trocas de contexto e dos objetos Python.

Para lidar eficientemente com a natureza de pilha da JVM, o picoJava-I implementa uma cache de pilha de 64 entradas diretamente em hardware, gerenciada por um mecanismo de que move dados de e para a cache de dados principal de forma transparente, minimizando gargalos. Adicionalmente, a arquitetura foi estendida com 115 instruções customizadas que utilizam o espaço de opcode reservado da JVM. Essas instruções são cruciais para permitir a escrita de código de sistema (como acesso direto à memória) e para otimizar funcionalidades complexas, como o suporte a coleta de lixo. Todas essas decisões de design foram tomadas com o objetivo de manter um núcleo pequeno e de baixo custo, permitindo que a área do chip (*die area*) e a unidade de ponto flutuante fossem configuráveis para diferentes mercados.

O desempenho da arquitetura dedicada foi validado com benchmarks e os resultados demonstraram que o picoJava-I é de 15 a 20 vezes mais rápido que um processador 486 com um interpretador, e cinco vezes mais rápido que um Pentium com um compilador JIT, normalizando a frequência para 100 MHz. Não são expostos resultados de síntese como área utilizada e frequência máxima de relógio.

O projeto representa, portanto, uma prova de conceito fundamental: ele demonstra que a aceleração de uma máquina virtual baseada em pilha por meio de hardware dedicado é não apenas viável, mas capaz de oferecer ganhos de desempenho de ordens de magnitude. Os resultados e as técnicas empregadas validam a hipótese de que uma abordagem semelhante pode ser aplicada com sucesso para mitigar os desafios de desempenho da PVM, justificando a exploração de um processador dedicado para bytecodes Python.

### 3.3.3 picoJava-II

O picoJava-II, uma revisão da arquitetura original picoJava-I, lançada pela Sun Microsystems, representa um dos mais complexos e bem documentados processadores Java. Embora nunca tenha sido lançado comercialmente como um chip pela Sun, seu código-fonte em Verilog foi disponibilizado, permitindo implementações por terceiros (PUFFITSCH; SCHOEBERL, 2007).

A arquitetura do picoJava-II aprofunda os conceitos de seu predecessor, mantendo o modelo de execução em três níveis (hardwired, microcódigo e emulação) e otimizações como o instruction *folding* e uma cache de pilha de 64 entradas. Uma evolução notável em relação à primeira versão é a expansão do pipeline para seis estágios: Busca, Decodificação, Acesso a Registradores, Execução, Acesso à Cache e Escrita. O objetivo do projeto permaneceu o mesmo: executar bytecode Java diretamente em hardware de forma eficiente, mas com um design mais maduro.

Na implementação em um FPGA Altera Cyclone II, o núcleo do picoJava-II (com caches de 16 KB e sem unidade de ponto flutuante) consumiu 27.543 Células Lógicas (*LCs - Logic Cells)*. Este valor é consideravelmente superior — de sete a treze vezes maior — ao de outros processadores Java acadêmicos, evidenciando a complexidade do design industrial da Sun. A frequência de operação máxima alcançada na plataforma FPGA foi de 43 MHz, com os testes sendo realizados a 40 MHz. O principal fator limitante da frequência foi o atraso de interconexão na unidade de multiplicação e divisão inteira.

Apesar da frequência modesta e do alto uso de recursos, os testes de benchmark (Sieve e Kfl) demonstraram que o picoJava-II superou o desempenho de outros processadores Java acadêmicos, sendo quase 20% mais rápido que seu concorrente mais próximo JOP (SCHOEBERL, 2008). O estudo conclui que, embora o picoJava-II seja uma arquitetura complexa e de alto custo em termos de área para uma implementação em FPGA, suas otimizações de hardware, como o *folding* e a cache de pilha, são eficazes o suficiente para garantir um desempenho competitivo mesmo em frequências de relógio mais baixas. Este trabalho serve como uma referência importante sobre os desafios de implementação e os compromissos entre área, frequência e desempenho ao se portar uma arquitetura projetada para ASIC para uma plataforma FPGA.

### 3.3.4 JOP

O *Java Optimized Processor* (JOP) é uma arquitetura de processador de 32 bits, implementada como um softcore para FPGAs, com um foco estrito em sistemas embarcados de tempo real (SCHOEBERL, 2008). Diferente de outros processadores que visam o desempenho de caso médio, o principal objetivo de design do JOP é a previsibilidade temporal, ou seja, garantir que o tempo de execução de pior caso (*WCET - Worst-Case Execution Time*) das instruções possa ser analisado e garantido, uma exigência crítica para sistemas de tempo real.

Para alcançar a previsibilidade, a arquitetura do JOP adota um design simplificado. Ele utiliza um modelo de execução misto: bytecodes simples da JVM são traduzidos e executados diretamente em um único ciclo de relógio pelo hardware, enquanto instruções mais complexas são tratadas por um motor de microcódigo. O núcleo do processador possui um pipeline enxuto de três estágios (Busca, Decodificação, Execução) que opera sobre essas micro instruções. Essa abordagem evita otimizações complexas e de difícil previsão, como o instruction folding do picoJava (PUFFITSCH; SCHOEBERL, 2007).

Uma das otimizações mais notáveis do JOP é seu sistema de cache, estratégia também adotada no presente trabalho. Em vez de uma cache de dados tradicional, ele implementa uma cache de método, projetada para que falhas de cache ocorram apenas em instruções de chamada e retorno de métodos. Esse design torna o comportamento da cache altamente previsível, simplificando a análise de WCET.

A avaliação de desempenho do JOP, implementado em FPGAs com frequências de até 100 MHz, revela um clássico compromisso de engenharia quando comparado ao picoJava-II. Embora o JOP seja ligeiramente mais lento em testes de desempenho de caso médio, essa diferença é uma consequência direta de sua filosofia de projeto, que prioriza a previsibilidade temporal e, crucialmente, a eficiência de recursos. Essa eficiência é evidente na área de síntese: o JOP ocupa cerca de 2.271 LCs, um valor mais de 12 vezes inferior aos 27.543 LCs necessários para a implementação do picoJava-II. Portanto, o JOP representa uma abordagem de design muito mais enxuta e econômica, otimizada para o domínio de sistemas embarcados com restrições de custo e área.

A relevância do JOP para este trabalho não reside apenas no uso de um pipeline e cache, mas em sua abordagem de criar uma arquitetura de hardware co-projetada com a semântica da máquina virtual para atender a um domínio de aplicação específico (tempo real). Ele valida a tese de que é possível construir processadores eficientes e

especializados para linguagens de alto nível, servindo como uma importante referência sobre os compromissos de design entre previsibilidade e performance.

### 3.3.5 jHISC

O processador jHISC representa uma abordagem inovadora para a aceleração de hardware da JVM, com um foco particular na execução nativa de instruções orientadas a objetos (YIYU et al., 2006). Diferente de outras arquiteturas que tratam operações complexas com objetos via software traps ou microcódigo, o jHISC foi projetado para implementar a maior parte da semântica da JVM diretamente em hardware, alcançando uma cobertura de 94% de todos os bytecodes e, mais impressionantemente, 83% dos bytecodes relacionados a objetos.

A principal inovação arquitetural do jHISC é o uso de descritores de operandos, uma estrutura de dados de 32 bits interpretável por hardware que descreve as propriedades dos objetos, como tipo, permissões de acesso e atributos. Isso permite que o processador realize operações complexas, como acesso a campos de um objeto, de forma segura e eficiente em hardware, verificando permissões e tipos sem a necessidade de intervenção de software. Essa capacidade é suportada por um pipeline de seis estágios (*IFETCH, ITRANSLATION, IDECODE, DFETCH, EXEC, WBACK*), que inclui um estágio dedicado à tradução e ao *folding* de instruções para otimizar o fluxo de execução.

A eficácia dessa abordagem foi validada através de implementação em FPGA. Em termos de desempenho, o jHISC demonstrou uma performance superior aos seus contemporâneos, sendo 30% mais rápido que o picoJava-II e 183% mais rápido que o JOP em média de ciclos por bytecode, utilizando frequências de relógio normalizadas. Notavelmente, esse ganho de desempenho sobre o picoJava-II foi alcançado utilizando 15.803 LCs, demonstrando uma arquitetura mais eficiente tanto em velocidade quanto em área.

O jHISC, portanto, não apenas valida a aceleração de hardware para a JVM, mas prova que é viável implementar a semântica complexa de uma linguagem orientada a objetos diretamente em silício. Este projeto serve como uma referência fundamental, para a premissa de que uma arquitetura customizada, consciente da natureza dos objetos, pode oferecer ganhos significativos de desempenho e eficiência.

### 3.3.6 JAIP-MP

O JAIP-MP (*Java Application IP Multi-Processor*) é uma arquitetura de processador de quatro núcleos projetada para a execução de alto desempenho de aplicações Java, com um foco particular em implementar em hardware funções críticas que tradicionalmente pertencem a um sistema operacional ou a uma máquina virtual de software, como o gerenciamento de *threads* e a sincronização (TSAI et al., 2016; TSAI et al., 2015).

Cada núcleo JAIP individual opera com um pipeline de quatro estágios (Tradução, Busca, Decodificação e Execução) que implementa um processo de otimização em duas fases. Primeiro, no estágio de tradução, os bytecodes Java, que são complexos e de tamanho variável, são convertidos nas instruções nativas do processador, chamadas de j-codes. Essa tradução cria um fluxo de instruções mais simples e regular, adequado para o processamento em hardware. Em seguida, com as instruções já no formato de j-codes, o pipeline utiliza sua principal técnica de otimização, o *instruction folding*. O estágio de decodificação identifica pares de j-codes que ocorrem em sequência (como uma carga de variável seguida por uma operação aritmética) e os combina para serem executados como uma única operação em um ciclo de relógio. Essa fusão de instruções resulta em uma taxa de otimização de 10% a 40%, dependendo da aplicação.

Para otimizar o acesso à pilha de operandos, o JAIP emprega uma arquitetura de dois níveis com registradores dedicados para o topo da pilha e uma memória *on-chip* customizada, semelhante ao apresentado neste trabalho. Essa memória é organizada em uma configuração “ping-pong” com duas pilhas, permitindo que o contexto de uma *thread* seja carregado enquanto outra está em execução, o que possibilita trocas de contexto em um único ciclo de relógio. A integração dos quatro núcleos é gerenciada por um Controlador de Coerência de Dados (*DCC - Data Coherence Controller*) centralizado, que gerencia o balanceamento de carga de *threads* entre os núcleos e garante a consistência entre as caches de cada núcleo através de um protocolo de *snooping* com uma política de escrita *write-through*.

A arquitetura completa foi implementada e validada em um FPGA Xilinx Virtex-6, operando a uma frequência de 83,3 MHz. Os resultados dos benchmarks multi-thread demonstraram uma excelente escalabilidade, com um ganho de desempenho (speedup) de até 3,69 vezes ao utilizar quatro núcleos em comparação com a execução em um único núcleo. A frequência máxima alcançada em testes foi de 83.6 MHz e a

utilização de células lógicas por núcleo foi de 12.580 LCs, totalizando 50.983 LCs para os quatro núcleos junto ao DCC.

O projeto JAIP-MP serve como uma importante referência para o co-design de sistemas multi-core complexos para linguagens de alto nível, validando a eficácia de se implementar em hardware mecanismos avançados de gerenciamento de tarefas e sincronização.

### 3.3.7 PamPy

O projeto PamPy, trabalho precursor a esta pesquisa, representa a iniciativa pioneira na exploração de uma arquitetura de hardware dedicada à execução de bytecodes da Máquina Virtual do Python (BITENCOURT, 2019). O objetivo do projeto foi validar a premissa de que uma arquitetura de propósito específico, descrita em HDL, poderia executar algoritmos Python de forma mais eficiente do que o modelo tradicional baseado em software, que depende de múltiplas camadas de abstração (PVM, sistema operacional, processador de propósito geral).

A metodologia adotada consistiu primeiramente na definição de um subconjunto de instruções Python consideradas essenciais e presentes na maioria dos algoritmos, abrangendo operações de acesso à memória, aritmética, lógica e controle de fluxo. Com base nesse conjunto, foi projetada e implementada uma arquitetura de processador monociclo, desenvolvida de forma modular e integrada. Para viabilizar os testes e a carga de programas, foi criado também o conversor pyConv, uma ferramenta em C responsável por traduzir o código Assembly Python (gerado pelo disassembler da linguagem) para o formato binário executável pelo hardware.

O projeto resultou em uma arquitetura semi-funcional, validada em simulação, capaz de operar a uma frequência máxima de 84,35 MHz, ocupando uma área de 728 elementos lógicos e com uma potência total dissipada de 132,56 mW. Embora o PamPy tenha validado com sucesso a premissa de que é possível executar a semântica da PVM diretamente em hardware, sua natureza de processador monociclo e seu conjunto de instruções limitado o estabelecem como uma prova de conceito fundamental. Suas limitações de desempenho, inerentes a uma arquitetura não-pipeline, e a ausência de mecanismos para suporte à troca de contexto entre *frames* são precisamente as lacunas que o presente trabalho busca endereçar, propondo sua evolução para uma arquitetura pipeline mais robusta e eficiente.

### 3.3.8 Considerações sobre os trabalhos relacionados

A análise dos trabalhos correlatos revela um conjunto de princípios e desafios recorrentes no projeto de hardware para máquinas virtuais de alto nível. Fica evidente que o sucesso de tais arquiteturas não reside apenas na execução direta de bytecodes, mas na implementação de otimizações arquiteturais específicas, como pipelines eficientes e utilizações de caches para mitigar o gargalo de acesso à memória. Além disso, a abordagem de execução híbrida — combinando instruções executadas diretamente em hardware com microcódigo e emulação em software para as mais complexas — emerge como uma estratégia pragmática para equilibrar desempenho e complexidade.

Esses aprendizados servem como pilares para o planejamento e desenvolvimento da arquitetura proposta neste trabalho. Portanto, a análise de resultados, apresentada no capítulo 5, utiliza métricas análogas às encontradas nestes trabalhos de referência para avaliar a eficácia do processador desenvolvido.

A primeira etapa na avaliação da arquitetura proposta consiste em uma análise mais simples de sua eficiência enquanto implementação em hardware. Esta análise é quantificada por duas métricas principais: a área de síntese, medida em células lógicas e a frequência máxima de operação ($F_{max}$) alcançável em síntese. Para estabelecer uma base de comparação com o estado da arte, a tabela 1 consolida esses dados para os principais trabalhos correlatos analisados no referencial teórico. Os valores apresentados nesta tabela são indicativos e refletem condições específicas de cada fluxo. Para uma comparação efetiva entre trabalhos, é necessária a normalização para uma mesma tecnologia e, sem essa normalização, os números devem ser interpretados apenas como ordem de grandeza.

| Projeto | Estágios de Pipeline | Células Lógicas | Frequência de Operação | Cache |
|---|---|---|---|---|
| picoJava-II (2007) | 6 | 27.543 | 43 MHz | Dados |
| JOP (2008) | 3 | 2.271 | 100 MHz | Métodos |
| jHISC (2006) | 6 | 15.503 | 100 MHz | Dados |
| JAIP-MP (2016) | 4 | 12.580 | 83.6 MHz | Pilha |
| PamPy (2019) | 0 | 728 | 84.35 MHz | Pilha |
| ProPy | 3 | 36.582 | 200 MHz | Instruções |
| ProPy | 3 | 100.148 | 200 MHz | Dados + Instruções |

Tabela 1 – Consolidação das métricas de avaliação para os projetos

**Disclaimer:** Valores absolutos, sem normalização entre tecnologias utilizadas.

# 4 DESENVOLVIMENTO DO PROPY

Este capítulo detalha o processo de planejamento e implementação do processador ProPy, arquitetura de hardware com o objetivo de executar nativamente um subconjunto de instruções (*bytecodes*) da Máquina Virtual do Python. A arquitetura é concebida de maneira co-projetada, na qual as responsabilidades são divididas para desacoplar a execução de baixo nível da gestão de alto nível de programas. De um lado, o Processador ProPy, foco principal deste trabalho, consiste em um núcleo de hardware dedicado à execução nativa de bytecodes Python dentro de um determinado contexto (*frame*). Em paralelo, o Gerenciador Dinâmico de Memória (GDM) atua como um módulo de software que trata da gestão do estado do programa. Ele é responsável pela troca de *frames* e pelo acesso a variáveis, sendo projetado para ser executado como um co-processador na arquitetura.

A divisão de responsabilidades constitui o alicerce para a otimização do sistema. Considerando-se a limitação inerente aos recursos lógicos em hardware, a estratégia adotada consiste em alocar no ProPy unicamente as tarefas críticas ao desempenho de execução. Consequentemente, operações de gerenciamento, tais como a preparação de *frames* — caracterizadas por maior complexidade e menor frequência de execução —, são delegadas à camada de software do GDM, que oferece maior flexibilidade. Dessa forma, o projeto de hardware é preservado como um núcleo conciso, de alta performance e delimitado em sua função precípua.

Tal dissociação entre execução e gerenciamento é o elemento que garante à arquitetura seu caráter modular e expansível. A existência de uma interface de comunicação padrão entre o hardware de execução e o software de gerenciamento possibilita que os componentes evoluam de maneira independente. A adição de uma nova instrução, por exemplo, pode ser implementada de forma localizada no escopo do processador pipeline, sem demandar modificações sistêmicas na gestão de memória, o que corrobora a premissa de uma arquitetura planejada com potencial de aprimoramento.

As seções a seguir descrevem a trajetória de desenvolvimento da arquitetura ProPy. Iniciamos pela versão que realiza acesso a dados externos, concebida para validar os princípios centrais de co-design e fornecer uma base para análise crítica de suas capacidades e limitações. A partir dos aprendizados obtidos, detalhamos a metodologia utilizada para a evolução do projeto por meio da integração ao núcleo ProPy de uma cache interna de dados, projetada para ampliar a autonomia do processador.

## 4.1 Instruções suportadas

A definição do subconjunto de bytecodes suportados pelo processador, detalhados na Tabela 2, partiu de uma análise empírica focada nas operações mais fundamentais da Linguagem Python, versão 3.11. Utilizando a ferramenta de *disassembler* (dis), disponível na biblioteca padrão da linguagem, foram examinados os *bytecodes* gerados por *scripts* que implementavam tarefas comuns, como cálculos matemáticos, atribuições de variáveis e laços de repetição. Esta análise revelou que um conjunto recorrente de instruções constituía a base para a execução de todas essas operações essenciais. Portanto, a escolha recaiu sobre este conjunto específico, pois sua implementação em hardware garante que o processador possa executar nativamente os blocos de construção de uma vasta gama de algoritmos práticos, como operações lógicas de comparação, aritméticas de soma e subtração, desvio de fluxo condicionais e tratamento de variáveis, por exemplo. Ao conjunto de instruções foi adicionada a correspondência da função personalizada *EOF*, utilizada alternativamente para que o processador identifique que um *frame* foi completamente executado, sem a necessidade de requisitar um novo *frame* ou devolver uma resposta ao GDM. Essa abordagem pragmática assegura que o hardware projetado tenha aplicabilidade direta, focando os recursos de desenvolvimento naquilo que é mais frequentemente executado.

| Instrução | Grupo |
|---|---|
| POP_TOP | Manipulação de pilha |
| LOAD_CONST, LOAD_FAST, LOAD_GLOBAL | Acesso à memória |
| STORE_FAST, STORE_GLOBAL | Acesso à memória |
| DELETE_FAST, DELETE_GLOBAL | Acesso à memória |
| UNARY ($+, -, !, \sim$) | Aritmética |
| BINARY_OP (ADD, SUB, MUL, DIV, SHL, SHR) | Aritmética |
| BINARY_OP (AND, OR, XOR) | Aritmética / Lógica |
| COMPARE_OP ($<, \leq, =, \neq, \geq, >$) | Lógica |
| JUMP_FORWARD, JUMP_BACKWARD | Fluxo |
| JUMP_IF_FALSE_OR_POP, JUMP_IF_TRUE_OR_POP | Fluxo |
| POP_JUMP_FORWARD_IF_FALSE | Fluxo |
| POP_JUMP_FORWARD_IF_TRUE | Fluxo |
| POP_JUMP_BACKWARD_IF_FALSE | Fluxo |
| POP_JUMP_BACKWARD_IF_TRUE | Fluxo |
| POP_JUMP_IF_TRUE, POP_JUMP_IF_FALSE | Fluxo |
| CALL_FUNCTION, RETURN_VALUE | Função |
| EOF | Especial |

Tabela 2 – Instruções suportadas pelo processador ProPy

## 4.2 Conjunto mínimo de módulos do núcleo *ProPy*

Com base no subconjunto de bytecodes suportado e nas duas variantes arquiteturais avaliadas (arquitetura sem cache de dados, que realiza acesso a dados externos sob demanda e arquitetura com cache de dados interna), define-se o conjunto mínimo de módulos necessários para atingir os objetivos deste trabalho. Esses módulos estabelecem as garantias funcionais e de desempenho que permitem: (i) armazenar e prover instruções/dados com a taxa requerida pelo pipeline; (ii) sustentar o modelo de pilha da PVM; e (iii) executar e controlar o fluxo de instruções no processador.

O núcleo deve assegurar, por ciclo, a disponibilidade de pelo menos uma leitura e uma escrita em memória (para viabilizar instruções e dados), além de operações de pilha compatíveis com o padrão de avaliação da PVM (consumo de até dois operandos e armazenamento de um operando).

### 4.2.1 Memória cache para instruções e dados

O ProPy utiliza um módulo de memória *on-chip* com uma porta de escrita e uma porta de leitura síncronas, parametrizáveis em larguras de dados e de endereço, permitindo com que a arquitetura seja sintetizada com diferentes capacidades de armazenamento. A escrita e o endereço de leitura são registrados na borda de subida para garantir estabilidade do dado.

Em ambas as variantes do ProPy, o módulo sustenta, por ciclo, uma escrita e uma leitura concorrentes, desde que não sejam realizadas no mesmo endereço e tem o objetivo de ser instanciado como cache de instruções e cache de dados conforme necessidade. Na primeira variante da arquitetura a memória é instanciada uma única vez como cache de instruções, e na segunda, é instanciada duas vezes, uma como cache de instruções e outra como cache de dados.

No escopo deste trabalho, o termo "*cache*" é empregado apenas no sentido de explorar localidade — isto é, manter dados próximos aos elementos que os consomem —, sem adotar os mecanismos clássicos de *caches* convencionais (mapeamento, *tags*, detecção de *hit/miss*, políticas de substituição ou de escrita). Em termos estritos, as estruturas usadas são memórias *scratchpad*: o conteúdo é gerenciado de forma explícita pelo controle da arquitetura, com posicionamento e movimentação de dados determinísticos, sem lógica de coerência ou tratamento de falhas de acesso por hardware.

### 4.2.2 Pilha de dados

O ProPy emprega uma pilha *on-chip* com uma porta de escrita e duas portas de leitura síncronas, com controle de *pop*, parametrizável em largura de dados para concordância com o tamanho dos dados definidos para a arquitetura, e quantidade máxima de dados a serem manipulados. Leituras expõem, no mesmo ciclo, os dois dados do topo, enquanto a escrita ocorre na borda de subida do relógio, sempre no *TOS*. O controle oferece três operações: *push* (inserção de um elemento), *pop* (remoção de um elemento) e *pop_two* (remoção de dois elementos). O módulo possui elementos de controles para identificar *underflow* e *overflow*, bloqueando atualizações ilegais.

O uso é previsto para seguir o modelo de execução da PVM: instruções *load* empilham valores trazidos da memória para o *TOS*; instruções *store* consomem o *TOS* para gravação em memória; operações binárias (*binary_op*) e de comparação consomem *TOS* e *TOS - 1* e escrevem o resultado no topo.

### 4.2.3 Unidade lógica e aritmética

A *ALU* do ProPy é inteiramente combinacional e possui latência de um ciclo, permitindo que o resultado esteja disponível ao final do ciclo em que a instrução é decodificada. O módulo é parametrizável em largura de dados e identificador de instrução (i.e. ao instanciar o módulo da ALU pode-se definir que uma soma é representada pelo identificador 5 e em uma nova instância com o identificador 10), de forma a permitir atualizar rapidamente a correspondência de seleção para uma soma, por exemplo, sem a necessidade de modificar o RTL. O conjunto de operações é organizado por *opcode* principal e *oparg* (suboperação), compatível com os bytecodes mapeados (*BINARY_OP*, *COMPARE_OP*, operações unárias). As operações abrangem todas as operações lógicas e aritméticas presentes na tabela 2.

A *ALU* opera de forma alinhada ao modelo de pilha: operações binárias e de comparação consomem os dois dados do *TOS* e devolvem um único resultado, que é escrito na posição TOS pela lógica de controle, preservando a semântica da PVM. Essa organização favorece o acoplamento direto com a pilha de dados, minimizando controle adicional e mantendo o caminho crítico sob observação apenas no bloco combinacional.

### 4.2.4 Contextos de transferência e tipagem de dados

O protocolo de comunicação do ProPy utiliza *dados etiquetados* para transportar, em cada transferência, dois metadados: o *contexto* (escopo do dado ou instrução) e o *tipo* de dado. Essa codificação preserva a correspondência entre a organização de memória por *frame* e a semântica dinâmica da PVM, permitindo que processador e GDM operem de forma coerente.

O campo de contexto, definido na Tabela 3, segmenta os espaços lógicos de um *frame* e direciona o roteamento interno. A memória *on-chip* é organizada em dois blocos: cache de instruções e cache de dados. Na cache de dados, os contextos **CONST**, **FAST** e **GLOBAL** mapeiam para segmentos independentes, cada qual endereçado por *offset* relativo ao seu espaço lógico, armazenando dados de 32 bits. Somente são suportados dados individuais de 32 bits, sem correspondência em mais de uma posição para um mesmo dado, não sendo possível o armazenamento de variáveis complexas como objetos ou strings com mais de 32 bits.

No carregamento de cada *frame*, o GDM envia blocos rotulados pelo contexto correspondente em um barramento de 3 bits: **INST** preenche a cache de instruções; **CONST**, **FAST** e **GLOBAL** preenchem, respectivamente, os seus segmentos na cache de dados. Durante a execução de um *frame*, instruções de leitura e escrita utilizam o campo de contexto para selecionar o segmento apropriado e um *offset* para localizar a linha. A etiqueta de tipo acompanha o valor, no caso de dados, ao longo do pipeline, preservando a tipagem dinâmica do Python, sem interferir no roteamento por contexto.

Na finalização de um *frame*, quando há uma chamada aninhada, o processador emite **FUNC** para que o GDM carregue o novo *frame*. Quando a execução do *frame* atual termina sem chamadas pendentes, o processador emite **ANSWER**, indicando retorno de resultado; o GDM verifica se o *frame* concluído era aninhado e, se for o caso, prepara o *frame* predecessor para ser carregado assim que o processador entrar em modo ocioso.

Após o retorno da resposta ou requisição de um *frame* aninhado, na arquitetura com cache de dados, o processador executa o *write-back* dos valores modificados em execução. Nessa etapa são transferidos ao GDM apenas os dados alterados na cache de dados, delimitando o bloco com **CLOSE**. O protocolo concentra a comunicação externa nas fronteiras do *frame* e reduz paralisações do *pipeline* durante instruções de *load/store*.

| Contexto | Descrição |
|---|---|
| **INST** | Identifica a transação de uma instrução |
| **CONST** | Identifica um valor do espaço de constantes |
| **FAST** | Identifica um valor do espaço de variáveis locais |
| **GLOBAL** | Identifica um valor do espaço de variáveis globais |
| **FUNC** | Identifica a chamada de um novo *frame* pelo processador |
| **ANSWER** | Identifica a resposta para a execução de um frame pelo processador |
| **CLOSE** | Terminador de bloco/transação (delimitação de fronteiras). |

Tabela 3 – Contextos de transferência

Já a tipagem, conforme tabela 4, acompanha o dado em transferências e movimentação para com a pilha de avaliação, permitindo decisões locais (e.g., realizar a soma ou a concatenação entre dois operandos quando a instrução é *ADD*) e validações no GDM. Os tipos apresentados correspondem a um subconjunto dos tipos da PVM, sendo selecionados por representarem grupos básicos de uso frequente.

O tipo **DELETE** não pertence ao conjunto de tipos da PVM. Trata-se de uma extensão do protocolo para sinalizar que o item endereçado deve ser invalidado no escopo indicado pelo campo de contexto. A arquitetura provê suporte ao **DELETE** caso, durante a execução, um dado seja removido através do comando Python ***del***, fazendo com que o GDM necessite tratar dessa remoção também. Na PVM, o uso de ***del*** ocorre apenas quando explicitamente solicitado no código, com a finalidade de melhorar a legibilidade e deixar claro que, a partir daquele ponto, a variável não será mais utilizada.

Nas operações de leitura, o processador empilha o par {valor, tipo}, preservando a tipagem na pilha de avaliação. Nas operações de escrita, o processador emite {valor, tipo} extraídos do topo da pilha, garantindo que a semântica dinâmica seja mantida no destino, e juntamente comunica {contexto, índice} baseado na instrução em execução. Quando o tipo é **DELETE**, o processador emite {tipo = **DELETE**, contexto, índice} para provocar a invalidação do item correspondente, sem necessidade de transportar um valor.

| Tipo | Descrição |
|---|---|
| **INT** | Inteiro primitivo suportado no subconjunto de bytecodes |
| **FLOAT** | Ponto flutuante primitivo |
| **STRING** | Cadeia de caracteres |
| **BOOLEAN** | Valor lógico |
| **DELETE** | Marcador de remoção/invalidação lógica do item endereçado |

Tabela 4 – Tipagem de dados

A presença explícita do campo de contexto e do campo de tipo na carga útil das transações assegura que cada transferência tenha metadados suficientes para ser aplicada de forma independente no destino. Isso oferece suporte direto ao modelo de *frames*

da PVM e à execução orientada a pilha. Na variante de acesso a dados sob demanda, simplifica transações elementares de leitura e escrita. Já na variante com cache de dados, viabiliza o pré-carregamento por escopo, o rastreamento de modificações e a escrita em bloco ao final do *frame*.

Embora a arquitetura proposta neste trabalho reconheça e transporte diferentes tipos de dados por meio de *etiquetas*, o suporte operacional, nesta versão, restringe-se a operações inteiras e booleanas para dados de 32 bits, independente do tipo. Os tipos são registrados em um vetor auxiliar de 3 bits, indexado um pra um pelas linhas da *cache* de dados. Assim, toda linha válida na *cache* de dados possui, na mesma posição desse vetor, o código do seu tipo. Por exemplo, se a linha 10 da *cache* de dados está ocupada, então o vetor de tipo, na posição 10 (*type[10]*) armazena o identificador de tipo correspondente, como 3'b000 para inteiro, 3'b001 para ponto flutuante, e assim por diante para os demais tipos.

O desenho modular preserva a extensibilidade: futuras versões poderão implementar o tratamento de *FLOAT* e *STRING*, por exemplo, reaproveitando integralmente a infraestrutura de transferência e de contexto já definida, limitando-se à ampliação da lógica de execução e às regras semânticas de cada tipo.

### 4.2.5 Formato de payload

*Payload* é o conjunto de campos, carga útil, que acompanha cada transferência entre o processador e o GDM. O *payload* é formado por: (i) **contexto** — identifica o escopo lógico do dado ou instrução (tabela 3); (ii) **índice** — endereça a posição dentro do escopo selecionado; (iii) **tipo** — etiqueta de tipagem do valor (tabela 4); e (iv) **valor** — conteúdo efetivo transportado. Os campos são transmitidos em paralelo e preservam a semântica de *frame* e de tipagem dinâmica.

Para fins de nomenclatura de sinais no código HDL, adotam-se prefixos e sufixos padronizados. O prefixo `s_` identifica sinais emitidos pelo lado que inicia a transação (mestre/"iniciador"); o prefixo `d_` identifica sinais do lado que a recebe (escravo/"alvo"). Os sufixos são relativos ao módulo observado: `_i` denota entrada do módulo e `_o` denota saída do módulo. Assim, em uma mesma ligação, o que é `s_contexto_o` no processador corresponde a `d_contexto_i` no GDM.

Para evitar ambiguidade entre texto e código, quando abreviações e renomeações aparecerem em nomes de sinais: `contexto` pode surgir como `ctx`, `índice` como

`idx`, `tipo` como `type` e `valor` como `data`.

### 4.2.6 Sinais do protocolo handshake

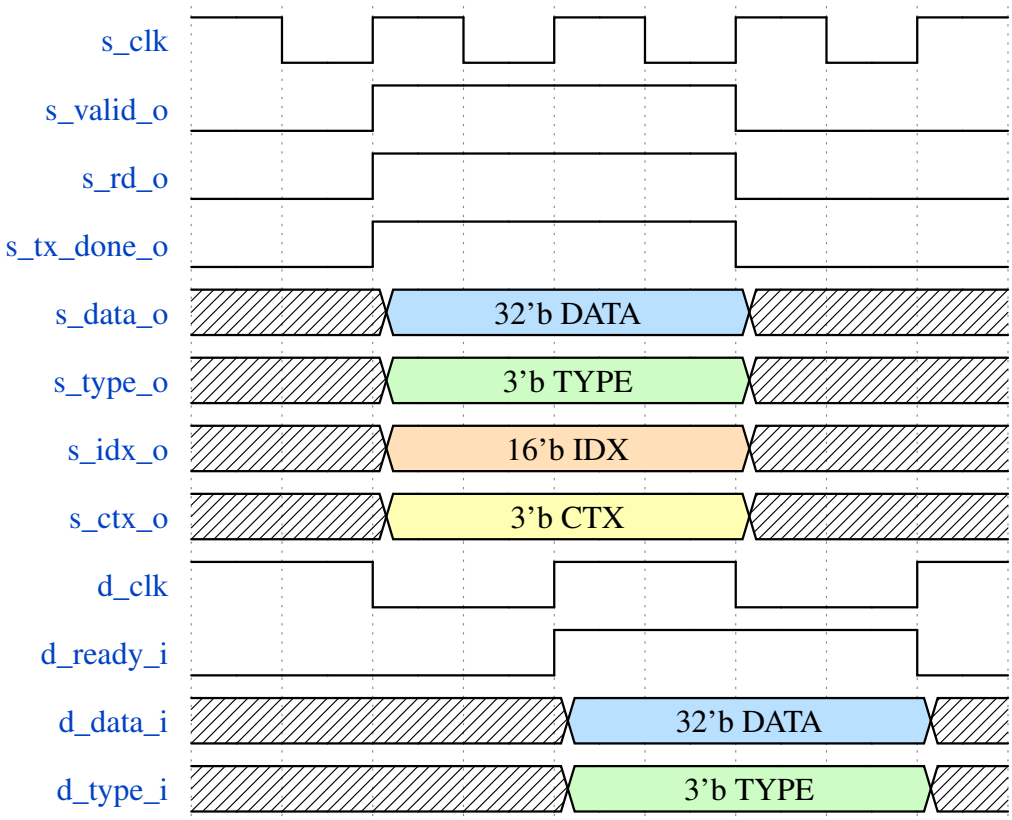
O ProPy utiliza uma interface de transação (troca de informação seguindo o protocolo especificado) com o GDM baseada em *handshake* de quatro fases. Cada transação transporta o *payload* (contexto, índice, tipo e valor), além de dois sinais de controle de fluxo: `valid` (emitido pelo mestre) e `ready` (emitido pelo escravo). Há ainda: `rd`, que indica operação de leitura quando ativo em nível alto e de escrita quando em nível baixo; e `tx_done`, que delimita o fim de bloco para enquadrar transferências múltiplas durante o carregamento.

A sequência do protocolo segue quatro fases: (1) **REQ↑** — o lado mestre apresenta o *payload* estável no barramento e ativa `valid`; (2) **ACK↑** — o lado escravo, após amostrar o *payload* com segurança, ativa `ready`; (3) **REQ↓** — o mestre desativa `valid`; (4) **ACK↓** — o escravo desativa `ready`, concluindo a transação. Em operações de leitura, a resposta (valor e tipo) percorre o caminho inverso do dado e é capturada pelo mestre na confirmação do escravo. No código HDL, aplica-se a convenção: `s_*_o` no módulo mestre corresponde a `d_*_i` no módulo escravo (e vice-versa), conforme a padronização de prefixos e sufixos apresentada no formato de *payload*.

A interface foi definida para integração entre domínios de relógio distintos. O *handshake* de quatro fases, aliado à sincronização por dois registradores em sequência (2 flip-flops) em sinais de controle, reduz o risco de metastabilidade sem impor relação de fase ou frequência entre relógios e tolera latências arbitrárias no escravo. Em contrapartida, cada item transacionado requer a conclusão do ciclo `valid`/`ready` (subida e descida), o que limita a vazão por ciclo quando comparado a canais síncronos ou a FIFOs assíncronos de alta largura.

A figura 7 apresenta a forma de onda para os sinais utilizados na interface *handshake*, vistos pela perspectiva do ProPy, com seus respectivos tamanhos de barramento, e nas seções seguintes, para cada arquitetura é detalhado como as interfaces são controladas para compreender os comportamentos esperados.

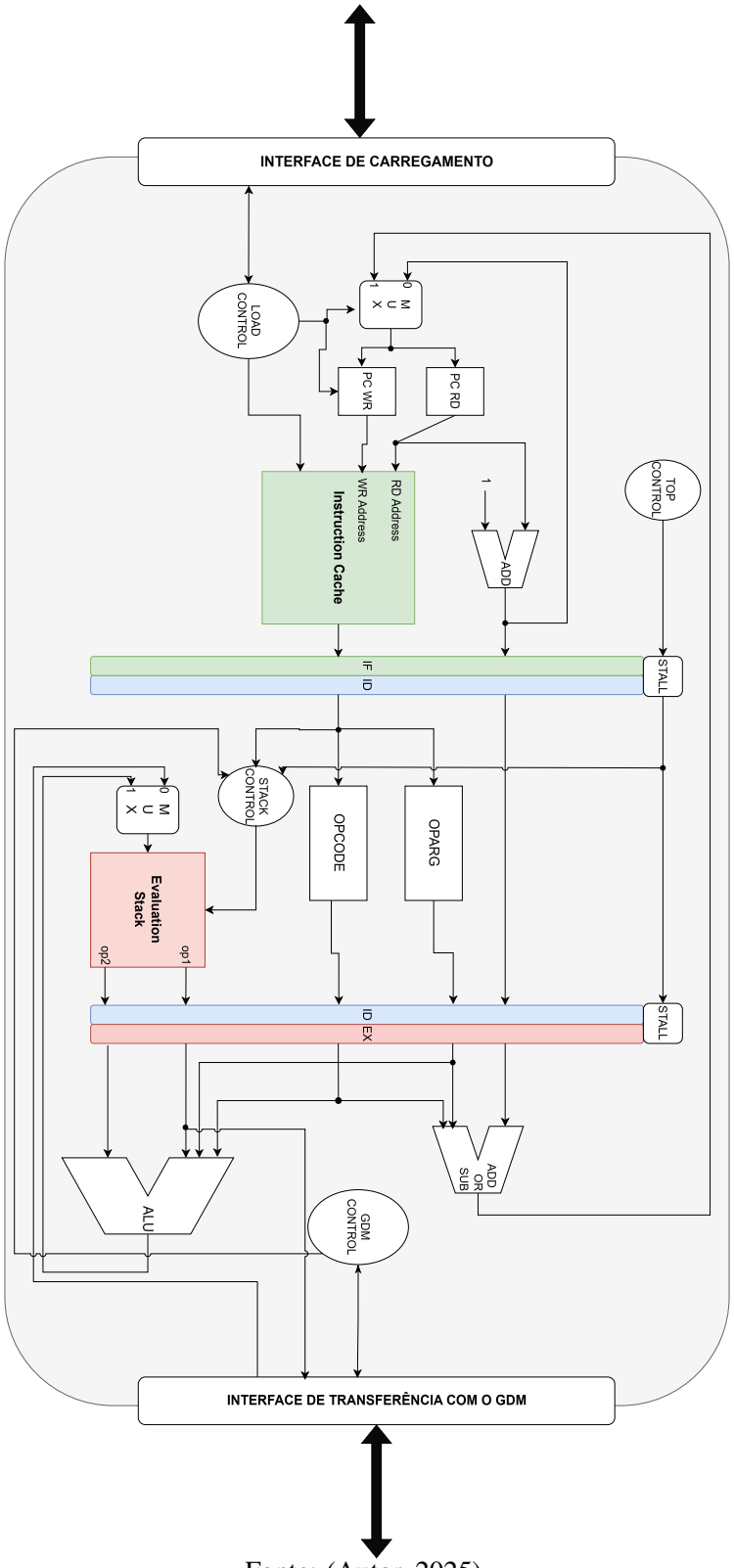
Figura 7 – Forma de onda da interface *handshake*




Fonte: (Autor, 2025)

## 4.3 Arquitetura com acesso interno à instruções e externo a dados

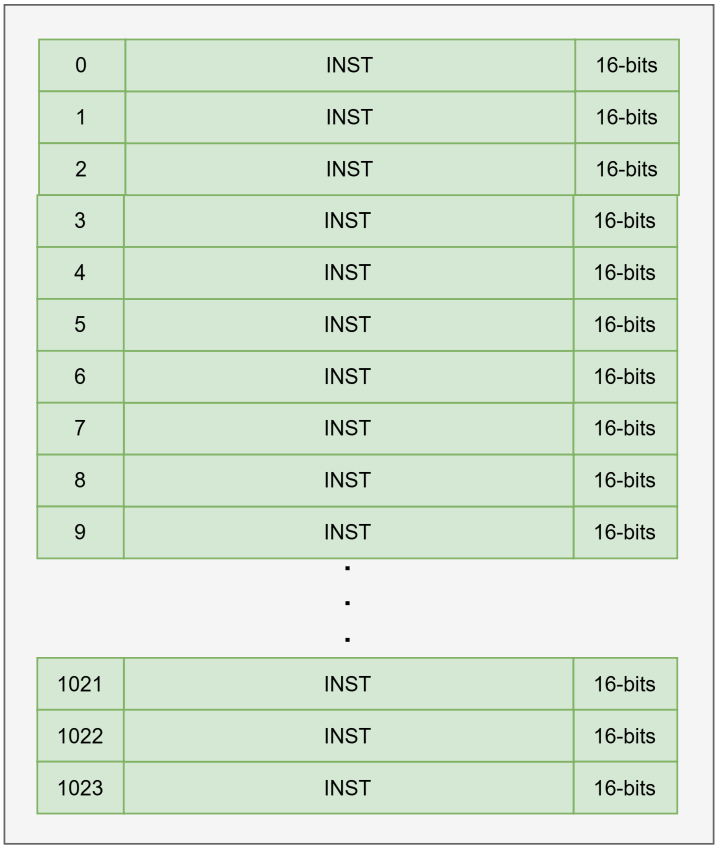
Figura 8 – Datapath arquitetura com acesso externo a dados




Fonte: (Autor, 2025)

Nesta variante, as instruções são executadas a partir de uma *cache* de instruções, enquanto os dados de execução são acessados sob demanda por meio do canal de comunicação com o GDM. A *cache* é linear, com mapeamento direto de uma instrução de 16 bits por posição. O preenchimento ocorre sequencialmente no estágio de carregamento e a leitura segue o mesmo padrão no estágio de execução; apenas instruções de desvio alteram o índice de leitura, após o que o fluxo volta ao avanço sequencial. A estrutura de armazenamento da *cache* de instruções está ilustrada na figura 9.

Figura 9 – Formato da cache de instruções

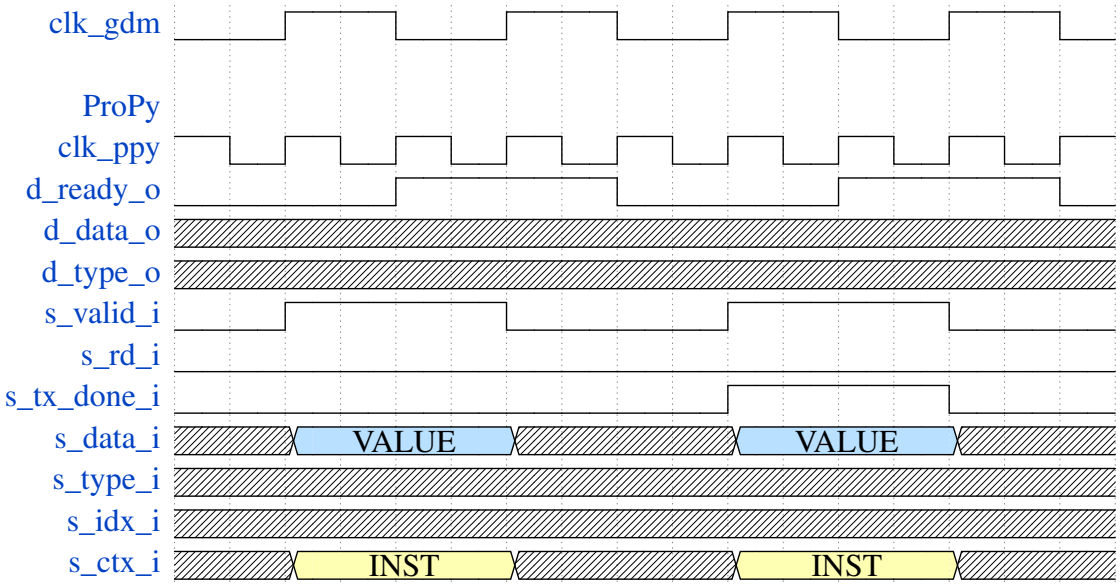



Fonte: (Autor, 2025)

O carregamento de instruções ocorre apenas quando o processador está ocioso, isto é, quando `proc_busy_o = 1'b0`. Nessa condição, a máquina de estados de recepção, assume o controle e recebe transferências pelo *handshake* de quatro fases conforme a requisição do GDM. Cada instrução aceita é escrita sequencialmente na memória de instruções por um contador de programa dedicado à escrita, incrementado a cada palavra recebida, até a detecção explícita da última transação via `tx_done`. Concluído o carregamento, o processador transita para o modo de execução e apenas retorna ao estado ocioso após e executar a última instrução do *frame*.

Conforme a figura 10, o GDM atua como mestre e o ProPy como escravo. Para cada palavra, o GDM coloca a instrução em `s_data` e ativa `s_valid`; o ProPy captura o dado e ativa `d_ready`; em seguida, o GDM desativa `s_valid` e, por fim, o ProPy desativa `d_ready`, concluindo a transação. Esse ciclo se repete até o término do *frame*. Na última transferência, o GDM ativa `tx_done` concomitantemente à última palavra em `s_data`; após completar o protocolo dessa última palavra, o processador reconhece o fim do carregamento e inicia a execução do *frame*.

Figura 10 – Forma de onda de carregamento para a arquitetura com acesso à dados externos

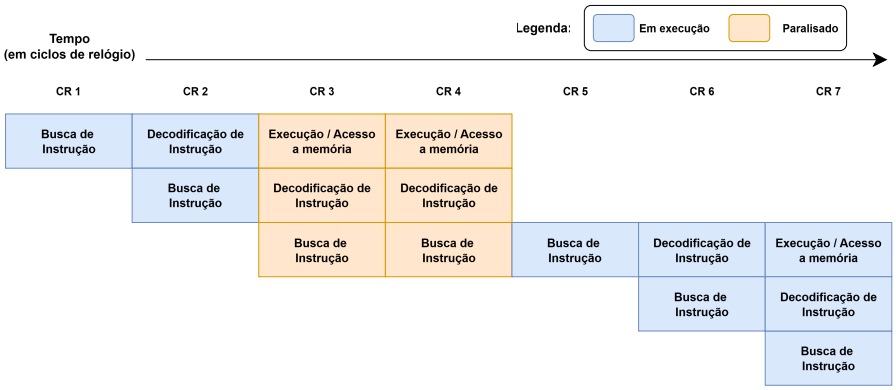



Fonte: (Autor, 2025)

Durante a execução, o ProPy opera um *pipeline* de três estágios — busca (IF), decodificação (ID) e execução/acesso à memória (EX/MA). Em IF, o contador de programa de leitura fornece o endereço, e a palavra de instrução é lida da *cache* de instruções.

No estágio ID, a instrução é decodificada em *opcode* e *oparg*, classificando-a como computacional, de memória ou de controle. Os operandos são obtidos do topo da pilha de avaliação ou diretamente do *oparg* conforme a semântica da PVM. O rótulo de tipo que acompanha os valores é apenas propagado com o dado, sem interpretação local nesta variante. Em desvios, o contador de programa de leitura é desviado baseado no *oparg* da instrução de desvio decodificada e os registradores de IF/ID são limpos, evitando a propagação de instruções inválidas, mediante a paralização do pipeline, como na figura 11.

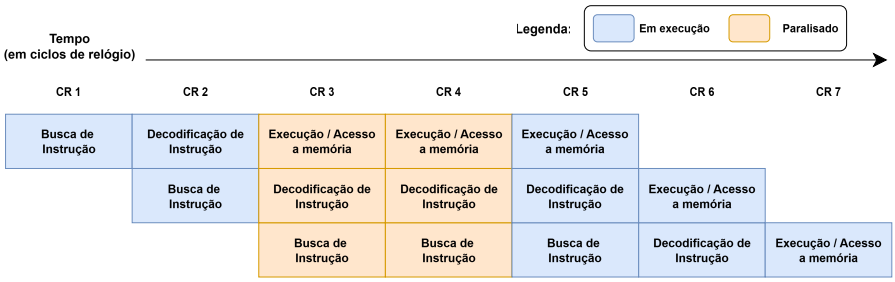
Figura 11 – Paralisação do pipeline em instruções de pulos




Fonte: (Autor, 2025)

No estágio EX/MA, as instruções computacionais são resolvidas em um ciclo pela ALU, com suporte exclusivo a operações inteiras. O resultado é produzido no mesmo ciclo e atualizado na pilha de avaliação. As instruções de memória acessam dados externamente e provocam paralisação do *pipeline*, como na figura 12: a detecção ocorre em ID, o processador é preparado para entrar em paralisação no próximo ciclo em EX/MA e a máquina de comunicação executa o *handshake* de quatro fases entre domínios de relógio distintos.

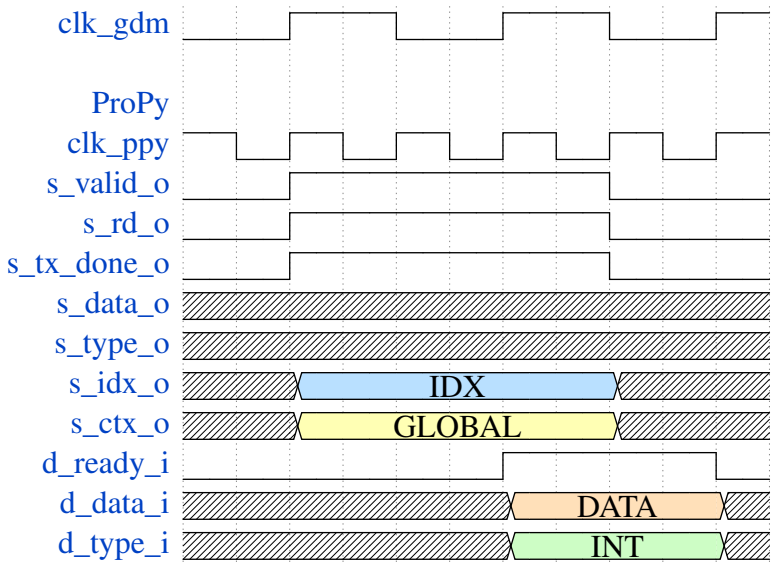
Figura 12 – Paralisação do pipeline na arquitetura com acesso à dados externos em requisição de dados




Fonte: (Autor, 2025)

Em leitura, o processador, atuando como mestre, emite o *payload* com {contexto, índice}, aguarda o reconhecimento, captura o par {valor, tipo} retornado e realiza o *push* na pilha, conforme figura 13.

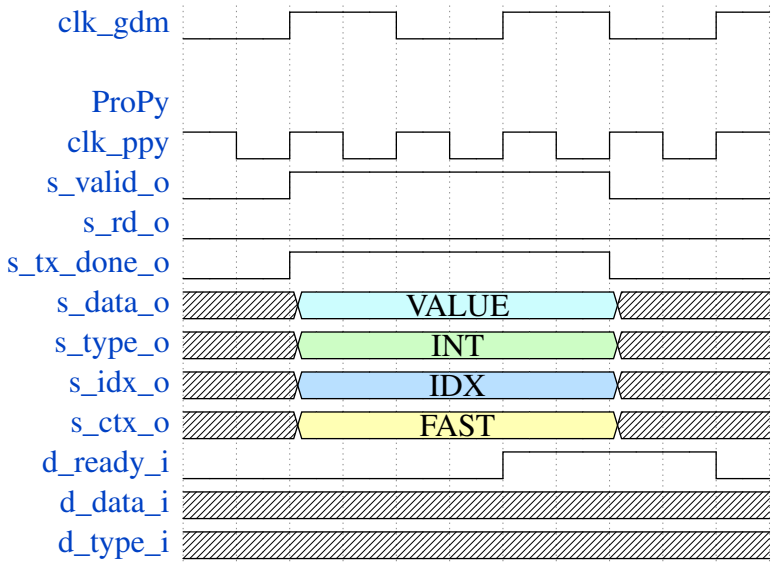
Figura 13 – Forma de onda na requisição de leitura de dados para a arquitetura com acesso à dados externos




Fonte: (Autor, 2025)

Em escrita, o processador, atuando como mestre, emite o topo da pilha com {contexto, índice, tipo, valor}, conforme figura 14, sendo o contexto e índice derivados do *opcode* e *oparg* e remove o elemento após o reconhecimento. Ao término da transação, a paralisação é liberada e IF/ID retomam o fluxo.

Figura 14 – Forma de onda na requisição de escrita de dados para a arquitetura com acesso à dados externos

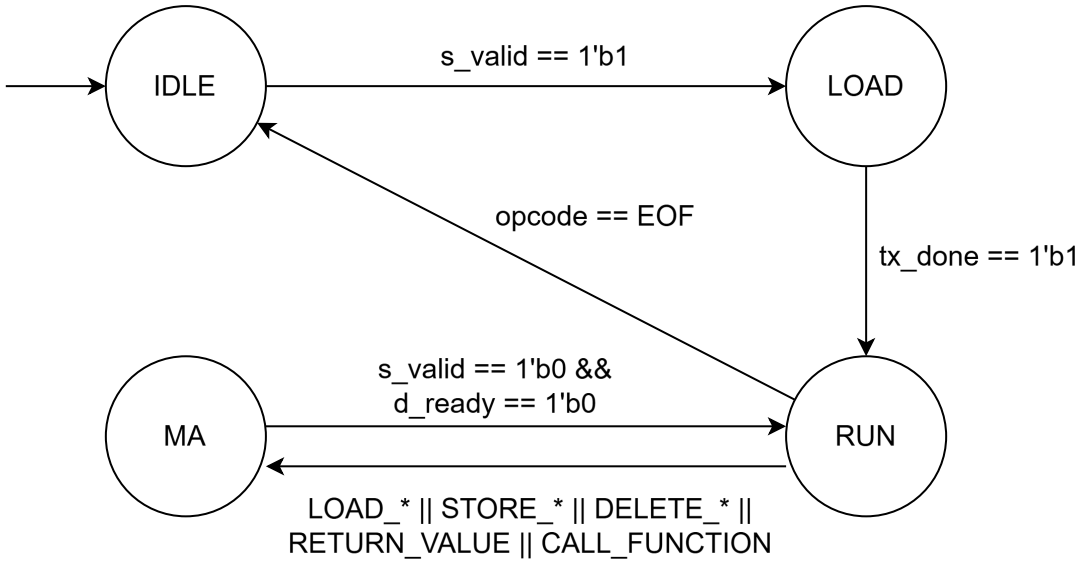



Fonte: (Autor, 2025)

O controle global segue três regras: (i) paralisação ocorre apenas em operações de memória e em desvios; instruções puramente computacionais permanecem em ciclo

único; (ii) a correta sequência das instruções é preservada pelo protocolo de comunicação e pela limpeza de registradores em mudanças de fluxo; (iii) a execução do *frame* termina com a instrução `EOF`, que coloca o processador em modo ocioso até o próximo carregamento de instruções. Para garantir a conclusão correta da última instrução do *frame*, adotou-se a inclusão de uma instrução extra ao final do fluxo. Em *frames* cujo término se dá por `RETURN_VALUE`, além do envio do topo da pilha ao GDM, é necessário efetuar o *pop* correspondente. Sem uma instrução extra, esse descarte poderia exigir lógica excepcional (por exemplo, introduzir um ciclo extra condicionado na decodificação de `RETURN_VALUE`). Com `EOF`, uma operação nula, o pipeline dispõe do ciclo adicional para realizar o *pop* e, na decodificação subsequente, encontrar `EOF` e encerrar o *frame* de forma limpa. Para evitar acréscimos de controle no ProPy, definiu-se que a responsabilidade de anexar `EOF` ao fim do *frame* é do GDM, que a transmite juntamente com as demais instruções.

Figura 15 – FSM de execução para a arquitetura com acesso à dados externos



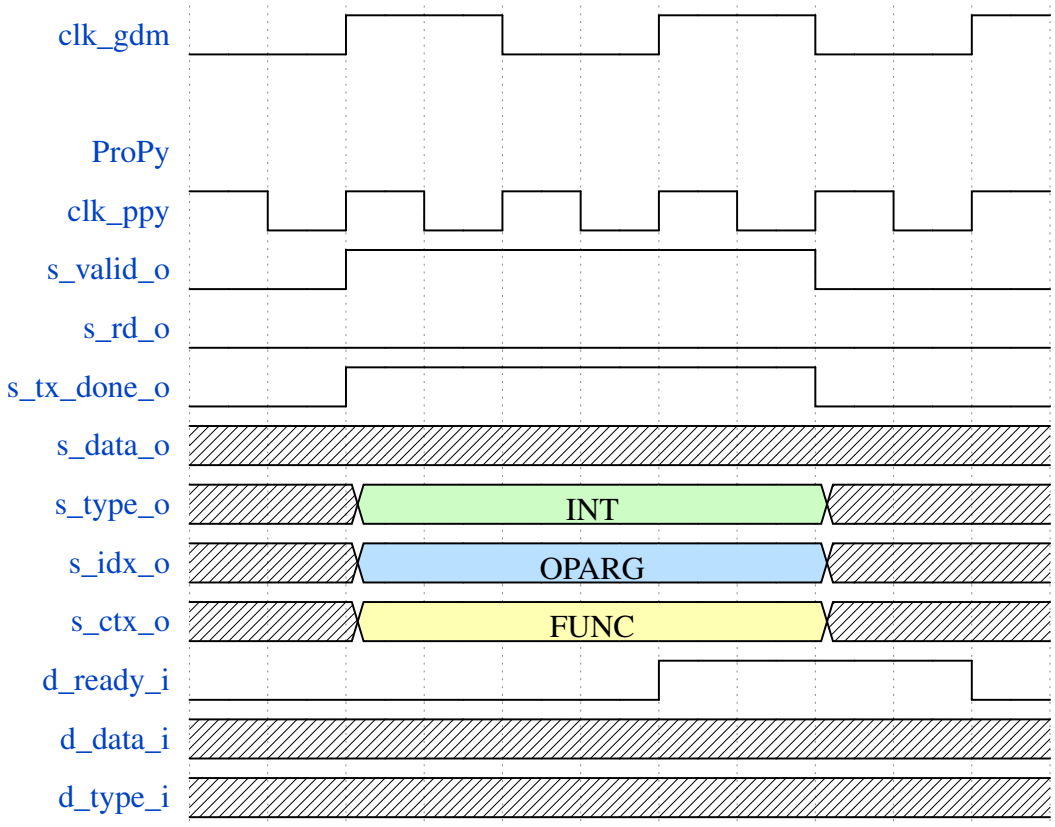

Fonte: (Autor, 2025)

A máquina de estados da Figura 15 resume o controle: (a) **IDLE** — o processador permanece ocioso após *reset* ou ao concluir um *frame*; ao detectar `s_valid` proveniente do GDM, transita para **LOAD**; (b) **LOAD** — a máquina de recepção assume a interface e grava instruções sequenciais na memória até `tx_done`; então transita para **RUN**; (c) **RUN** — o fluxo segue como descrito acima até que uma instrução de acesso à memória (`LOAD_`, `STORE_`, `DELETE_`, `CALL_FUNCTION`, `RETURN_VALUE`) seja decodificada, quando ocorre a transição para **MA**, com paralisação do *pipeline*; (d) **MA** — a máquina

de requisição executa a transação com o GDM; concluída a comunicação, retorna a **RUN**; (e) ao esgotar as instruções do *frame*, ocorre a transição para **IDLE**, identificado pelo sinal `proc_busy_o = 1'b0`, e o núcleo aguarda novo carregamento.

0.38 Quando a execução demanda a chamada de uma função (`CALL_FUNCTION`), o processador, atuando como mestre, solicita o carregamento do novo *frame* realizando uma requisição de escrita ao GDM, figura 16, informando como índice (*idx*) o `oparg` da instrução decodificada, o contexto (*ctx*) como `FUNC` e o tipo (*type)* como inteiro. O GDM inicia a transferência do novo *frame* apenas quando `proc_busy_o = 1'b0`, assegurando que o núcleo esteja em **IDLE** antes de realizar o carregamento.

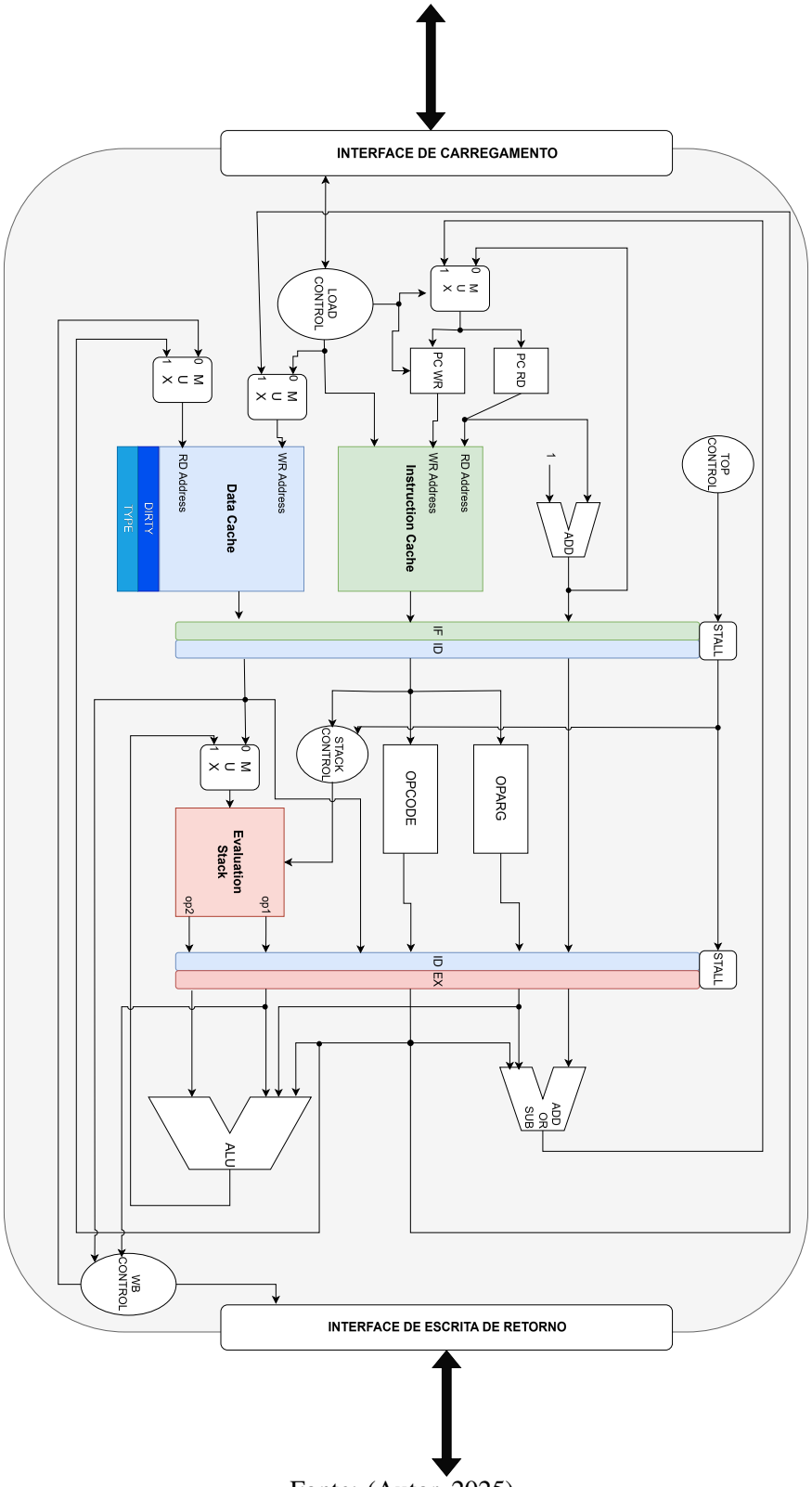
Figura 16 – Forma de onda na requisição de uma função




Fonte: (Autor, 2025)

## 4.4 Arquitetura com cache de dados *on-chip*

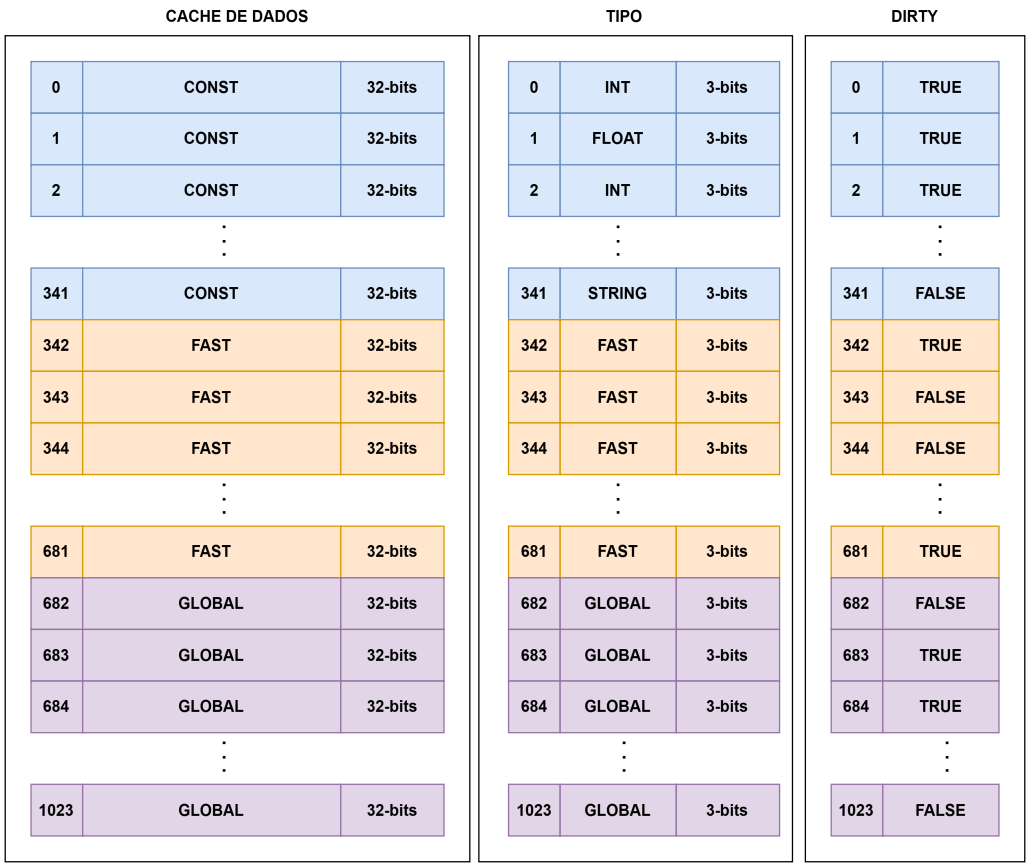
Figura 17 – Datapath arquitetura com cache de dados




Fonte: (Autor, 2025)

Nesta variante, os dados do *frame* ativo são carregados, assim como as instruções, para dentro da arquitetura, criando uma região de memória *scratchpad*, usada para armazenar dados modificados em tempo de execução, limitando a realização de transações externas com o GDM. A comunicação com o GDM restringe-se à duas situações recorrentes durante o processamento: primeiro, o **carregamento** do conjunto de instruções e dados; ao final, a **escrita em bloco** dos valores modificados (*write-back*). Assim como na seção anterior, o carregamento só ocorre quando o núcleo está ocioso (`proc_busy_o = 1'b0`); a máquina de recepção assume a interface e, por *handshake* de quatro fases, recebe palavras sucessivas até a detecção explícita da última transação via `tx_done`. A diferença é o destino do conteúdo: além da *cache* local de instruções, há uma cache local de dados, segmentada por contextos (`CONST`, `FAST` e `GLOBAL`), endereçados por mapeamento linear baseado em *offsets*, como $BASE_{CTX} + idx$. Em paralelo, metadados auxiliares mantêm, por posição, o *tipo* do valor e um *dirty bit*, informando se o dado ao qual referencia foi modificado ou excluído. A estrutura de armazenamento da *cache* de dados pode ser vista na figura 18.

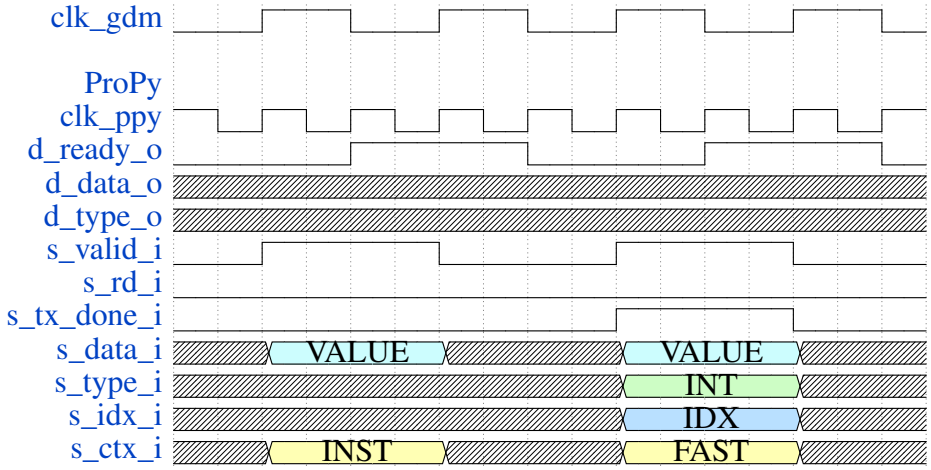
Figura 18 – Formato da cache de dados




Fonte: (Autor, 2025)

No carregamento inicial, as linhas são marcadas limpas, e, durante a execução, `STORE_` e `DELETE_` atualizam o tipo e marcam `dirty=1'b1`. A forma de onda do carregamento, na qual o processador atua como escravo, é mostrada na Figura 19, idêntica em protocolo ao caso sem *cache* (Figura 10), mas direcionando cada *payload* à *cache* correspondente.

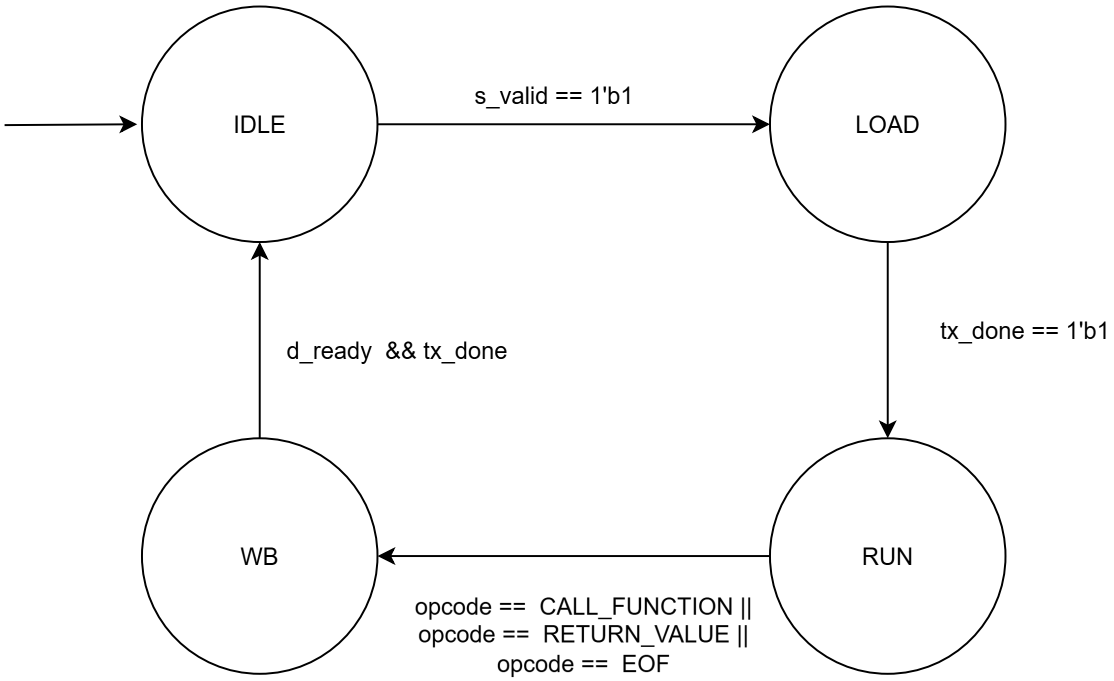
Figura 19 – Forma de onda de carregamento para a arquitetura com cache de dados




Fonte: (Autor, 2025)

Concluído o carregamento, o ProPy transita para o modo de execução e opera o mesmo *pipeline* de três estágios — busca (IF), decodificação (ID) e execução/acesso à memória (EX/MA) — descrito anteriormente. Em IF, o *program counter* de leitura endereça a *cache* de instruções; em desvios, aplica-se o ajuste por `oparg` e faz-se a limpeza dos registradores IF/ID, evitando a propagação de instruções inválidas, como ilustrado na Figura 11. No estágio ID, a instrução é decodificada em *opcode* e *oparg*, com obtenção dos operandos no topo da pilha conforme a semântica da PVM. Em EX/MA, as instruções computacionais são resolvidas em ciclo único pela ALU inteira; já as instruções de acesso a dados (`LOAD_`/`STORE_`/`DELETE_`) acessam diretamente o banco local, sem interação com o GDM. Com isso, eliminam-se paralisações por E/S no intra-*frame*; *stalls* se mantêm apenas por desvios no *program counter* de leitura ou quando o núcleo entra na fase de *write-back*. A FSM do topo durante a execução é apresentada na Figura 20.

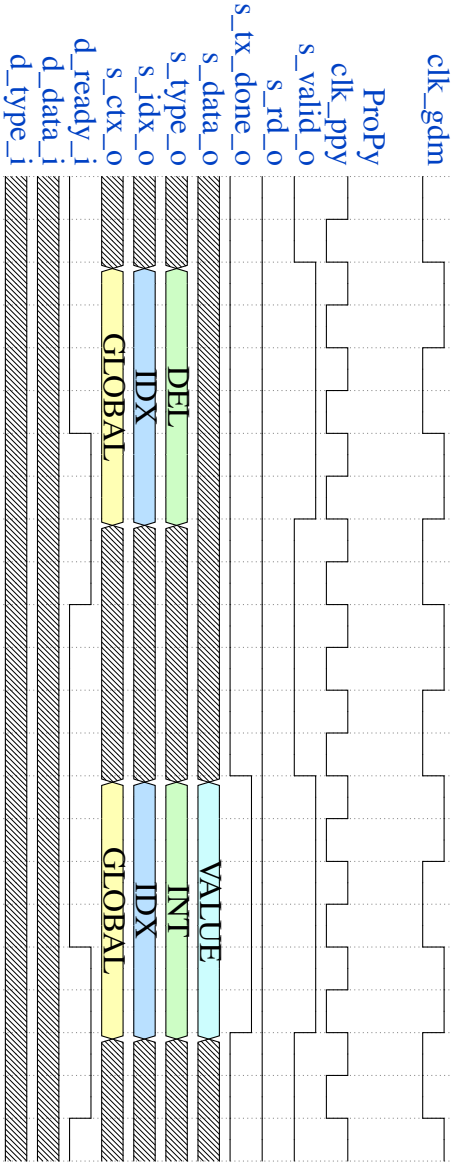
Figura 20 – FSM de execução para a arquitetura com cache de dados




Fonte: (Autor, 2025)

A máquina de estados de topo gerencia o ciclo **carregar–executar–salvar**. Após *reset*, o núcleo entra em **IDLE** (`proc_busy_o = 1'b0`) com a interface externa desabilitada. Ao detectar `s_valid` do GDM, transfere o controle para a máquina de recepção e transita a **LOAD**, onde instruções e dados (`INST`, `CONST`, `FAST`, `GLOBAL`) são recebidos por *handshake* de quatro fases e escritos nos bancos locais; o término é indicado por `tx_done`. Em seguida, a FSM entra em **RUN**, quando o pipeline consome exclusivamente as memórias *on-chip*; acessos a dados não requerem o GDM e eventuais desvios apenas promovem *flush*/realinhamento do `program counter` de leitura, sem sair de **RUN**. Ao esgotar o *frame* ou quando ocorrem as instruções `EOF`, `RETURN_VALUE` ou `CALL_FUNCTION`, a FSM transita para **WB**: o vetor `dirty` é varrido e, para cada linha suja em `GLOBAL`, emite-se ao GDM o pacote {*valor*, *tipo*, *contexto*, *índice*} mediante o mesmo protocolo de *handshake*, figura 21; a conclusão é sinalizada pelo envio de `CLOSE` (`tx_done = 1`). Finalizado o *write-back*, o núcleo retorna a **IDLE**, entra em modo ocioso `proc_busy_o = 1'b0` e fica apto a um novo **LOAD**. Em chamadas de função, o carregamento do novo *frame* ocorre apenas após esse retorno a **IDLE**, ocioso, garantindo a consistência entre o estado local e o estado global. Durante **LOAD** e **WB**, os estágios IF/ID/EX permanecem paralisados; somente as máquinas de recepção e de escrita em bloco operam.

Figura 21 – Forma de onda de *write-back* em bloco para a arquitetura com cache de dados




Fonte: (Autor, 2025)

# 5 RESULTADOS

Esta seção apresenta a validação funcional do processador, o caso de teste utilizado nas duas variantes arquiteturais (acesso externo de dados sob demanda e com cache de dados *on-chip*), o ambiente de medição e os resultados quantitativos de síntese e de execução em simulação.

## 5.1 Metodologia de validação

Todos os blocos básicos e protocolo de *handshake* em quatro fases, foram validados individualmente por meio de *testbenches* autocontidos, com estímulos controlados e respostas previamente conhecidas, permitindo uma verificação um-para-um. Essa etapa assegurou a correção funcional local antes da integração no topo, reduzindo o risco de falhas nas lógicas fundamentais.

Concluída a validação dos módulos, foram desenvolvidos dois topos, foco deste trabalho: (i) sem cache de dados, com acesso externo sob demanda; e (ii) com cache de dados, no modelo carregar–executar–salvar. Os *testbenches* de ambas as arquiteturas empregam os mesmos conjuntos de dados, mas respeitam as particularidades de cada modelo de comunicação. Para a versão com acesso externo sob demanda, o *testbench* carrega as instruções do *frame* no processador, mantém os dados externamente e os manipula conforme requisitado; ao término da execução, confronta os valores obtidos com as referências esperadas. Já na arquitetura com cache de dados, todo o *frame* (instruções e dados) é previamente carregado no processador e, após escrita de retorno do bloco de dados pelo processador, os resultados reportados são validados contra os valores esperados.

## 5.2 Caso de teste: Bubble Sort (N=10) em *Frame* único

O algoritmo selecionado foi o **Bubble Sort**, algoritmo de ordenação simples. A versão utilizada realiza varreduras decrescentes (N-1 ... 1) e, em cada passada, compara pares adjacentes (k, k+1) realizando a troca entre eles quando a > b. A ISA suportada pelo ProPy possui ausência de suporte direto a instruções de manipulação de vetores, sendo assim, foi empregada a ideia de mapear um vetor em endereços contíguos da região global de

memória.

A implementação de referência apresentada na figura 22 segue fielmente esse comportamento e pela tabela 5 fica evidente os passos e número de comparações executadas em cada passada $(9, 8, \ldots, 1)$.

Figura 22 – Bubble Sort de referência em Python para o *Frame* único

```
def bubble10(a0, a1, a2, a3, a4, a5, a6, a7, a8, a9):
    # Passada 1 (até índice 9)
    if a0 > a1: a0, a1 = a1, a0
    if a1 > a2: a1, a2 = a2, a1
    if a2 > a3: a2, a3 = a3, a2
    if a3 > a4: a3, a4 = a4, a3
    if a4 > a5: a4, a5 = a5, a4
    if a5 > a6: a5, a6 = a6, a5
    if a6 > a7: a6, a7 = a7, a6
    if a7 > a8: a7, a8 = a8, a7
    if a8 > a9: a8, a9 = a9, a8

    # Passada 2 (até índice 8)
    if a0 > a1: a0, a1 = a1, a0
    if a1 > a2: a1, a2 = a2, a1
    if a2 > a3: a2, a3 = a3, a2
    if a3 > a4: a3, a4 = a4, a3
    if a4 > a5: a4, a5 = a5, a4
    if a5 > a6: a5, a6 = a6, a5
    if a6 > a7: a6, a7 = a7, a6
    if a7 > a8: a7, a8 = a8, a7

    ...

    # Passada 8 (até índice 2)
    if a0 > a1: a0, a1 = a1, a0
    if a1 > a2: a1, a2 = a2, a1

    # Passada 9 (até índice 1)
    if a0 > a1: a0, a1 = a1, a0

    return a0, a1, a2, a3, a4, a5, a6, a7, a8, a9
```

Fonte: (Autor, 2025)

| **Passada** | (0,1) | (1,2) | (2,3) | (3,4) | (4,5) | (5,6) | (6,7) | (7,8) | (8,9) |
|---|---|---|---|---|---|---|---|---|---|
| 1 | • | • | • | • | • | • | • | • | • |
| 2 | • | • | • | • | • | • | • | • | |
| 3 | • | • | • | • | • | • | • | | |
| 4 | • | • | • | • | • | • | | | |
| 5 | • | • | • | • | • | | | | |
| 6 | • | • | • | • | | | | | |
| 7 | • | • | • | | | | | | |
| 8 | • | • | | | | | | | |
| 9 | • | | | | | | | | |

Tabela 5 – Matriz de comparações do *Bubble Sort* para $N{=}10$. A marca • indica comparações executadas em cada passada.

O *bytecode* gerado pelo algoritmo de referência foi inspecionado com o módulo `dis` do Python e reescrito no subconjunto suportado pelo ProPy. A sequência para trocas gerada

oficialmente pela PVM (CPython) 3.11 é:

```
2 LOAD_FAST 0 (a0)
4 LOAD_FAST 1 (a1)
6 COMPARE_OP 4 (>)
12 POP_JUMP_FORWARD_IF_FALSE 4 (to 22)
14 LOAD_FAST 1 (a1)
16 LOAD_FAST 0 (a0)
18 STORE_FAST 1 (a1)
20 STORE_FAST 0 (a0)
```

A tradução do par $(k, k+1)$ do *compare-and-swap* difere pelo tratamento dos contextos de dados. Na PVM, privilegia-se o contexto `FAST` (variáveis locais) para encurtar o caminho crítico, evitando acessos a `GLOBAL`. No ProPy, a arquitetura com cache de dados mapeia `FAST` e `GLOBAL` como segmentações do mesmo espaço endereçável, de modo que acessar um ou outro não alonga o caminho crítico. Além disso, pelo protocolo de comunicação com o GDM, apenas modificações efetivas no escopo global são escritas de volta, enquanto variáveis locais não têm efeitos inter-funcionais. Assim, a tradução carrega $a$ e $b$ de `GLOBAL`, deposita-os temporariamente em `FAST` para a comparação e só atualiza `GLOBAL` quando há troca; já a sequência oficial da PVM permanece integralmente em `FAST`.

1. **Carregamento dos operandos globais** (empilham TOS/NTOS):
   (a) `LOAD_GLOBAL k` ($push\ G[k]$)
   (b) `LOAD_GLOBAL k+1` ($push\ G[k+1]$)
2. **Armazenamento temporário em `FAST`** (preserva os valores para uso posterior):
   (a) `STORE_FAST 3` ($pop \rightarrow$ `FAST[3]` $\leftarrow G[k+1]$)
   (b) `STORE_FAST 2` ($pop \rightarrow$ `FAST[2]` $\leftarrow G[k]$)
3. **Comparação** (*recarrega* os temporários e avalia a condição):
   (a) `LOAD_FAST 2` ($push$ `FAST[2]` $= a$)
   (b) `LOAD_FAST 3` ($push$ `FAST[3]` $= b$)
   (c) `COMPARE_OP >` (*consome* $a, b$ e *produz* booleano)
   (d) `POP_JUMP_FORWARD_IF_FALSE off` ($pop$ condição; salta se $a \leq b$)

4. **Troca condicional** (executada apenas se $a > b$):

   (a) `LOAD_FAST 3` (*push b*)

   (b) `STORE_GLOBAL k` ($G[k] \leftarrow b$)

   (c) `LOAD_FAST 2` (*push a*)

   (d) `STORE_GLOBAL k+1` ($G[k+1] \leftarrow a$)

A partir dessa sequência elementar de *compare–and–swap* para um par $(k, k+1)$, constrói-se uma **passada** encadeando o mesmo padrão para todos os pares adjacentes do prefixo $[0, \ldots, end]$, com $k = 0, 1, \ldots, end - 1$. Em cada passada, o maior elemento presente no prefixo é sucessivamente deslocado para a direita, até ocupar a posição *end*. Esse é o comportamento esperado do *bubble sort*: após a *i*-ésima passada, o sufixo final de tamanho *i* encontra-se ordenado e contém os *i* maiores elementos do arranjo. Assim, o algoritmo procede com $end = N-1, N-2, \ldots, 1$, "*borbulhando*" o máximo local para a extremidade a cada iteração, até que toda a lista esteja ordenada.

### 5.3 Caso de teste: Bubble Sort (N=10) com subrotinas de troca

O segundo caso de teste também utiliza o **Bubble Sort** (N=10), mas delega as trocas entre valores de pares adjacentes a uma subrotina dedicada. Sempre que uma troca é detectada, um novo *frame* é carregado apenas para executar a permuta do par e retornar o resultado. Em seguida, o *frame* chamador retoma a execução na instrução imediatamente posterior a `CALL_FUNCTION`.

O *bytecode* gerado oficialmente pela PVM (CPython) 3.11 para a implementação de referência, figura 23, é:

- *swap*

```
2 LOAD_FAST 1 (y)
4 LOAD_FAST 0 (x)
6 BUILD_TUPLE 2
8 RETURN_VALUE
```

- *bubble10*

```
2 LOAD_FAST 0 (a0)
4 LOAD_FAST 1 (a1)
```

```
6 COMPARE_OP 4 (>)
10 POP_JUMP_FORWARD_IF_FALSE 15 (to 42)
12 LOAD_GLOBAL 1 (NULL + swap)
22 LOAD_FAST 0 (a0)
24 LOAD_FAST 1 (a1)
26 CALL 2
34 UNPACK_SEQUENCE 2
38 STORE_FAST 0 (a0)
40 STORE_FAST 1 (a1)
```

Figura 23 – Bubble Sort de referência em Python com subrotinas de troca

```python
def swap(x, y):
    return y, x

def bubble10(a0, a1, a2, a3, a4, a5, a6, a7, a8, a9):
    # Passada 1 (até índice 9)
    if a0 > a1: a0, a1 = swap(a0, a1)
    if a1 > a2: a1, a2 = swap(a1, a2)
    if a2 > a3: a2, a3 = swap(a2, a3)
    if a3 > a4: a3, a4 = swap(a3, a4)
    if a4 > a5: a4, a5 = swap(a4, a5)
    if a5 > a6: a5, a6 = swap(a5, a6)
    if a6 > a7: a6, a7 = swap(a6, a7)
    if a7 > a8: a7, a8 = swap(a7, a8)
    if a8 > a9: a8, a9 = swap(a8, a9)

    ...

    # Passada 9 (até índice 1)
    if a0 > a1: a0, a1 = swap(a0, a1)

    return a0, a1, a2, a3, a4, a5, a6, a7, a8, a9
```

Fonte: (Autor, 2025)

Os *bytecodes* gerados para esta implementação do ***Bubble Sort*** incluem instruções não tratadas diretamente pelo ProPy, e o modelo de comunicação com o GDM, bem como a organização de memória da arquitetura, não privilegiam uma correspondência 1:1 com o comportamento em software. Assim, para reproduzir o comportamento esperado, torna-se necessária a manipulação desses *bytecodes*. Primeiro, na arquitetura com *cache* de dados, os retornos do processador ocorrem apenas para dados do contexto global. Segundo,

instruções que operam sobre tipos de dados estruturados, como tuplas (`BUILD_TUPLE`, `UNPACK_SEQUENCE`), entre outras, não são suportadas. Além disso, o ProPy somente suporta o carregamento de um único *frame* por vez, sendo necessário o descarregamento e o carregamento completo quando ocorre uma troca de *frame* para execução.

Dessa forma, a tradução para o ProPy inicia-se tal como na versão sem subrotinas, repetindo as etapas: (1) **carregamento dos operandos globais**, (2) **armazenamento temporário em `FAST`** e (3) **comparação**. Quando identificada a necessidade de permuta, o par é movido para a pilha de avaliação e, em seguida, invoca-se a subrotina `swap`; esta é executada e, após o `RETURN_VALUE`, o controle retorna à função chamadora na instrução imediatamente posterior a `CALL_FUNCTION`, conforme o fluxo abaixo:

1. **Carregamento dos operandos globais** (empilham TOS/NTOS):
   (a) `LOAD_GLOBAL k` ($push\ G[k]$)
   (b) `LOAD_GLOBAL k+1` ($push\ G[k+1]$)
2. **Armazenamento temporário em `FAST`** (preserva os valores para uso posterior):
   (a) `STORE_FAST 3` ($pop \rightarrow$ `FAST[3]` $\leftarrow G[k+1]$)
   (b) `STORE_FAST 2` ($pop \rightarrow$ `FAST[2]` $\leftarrow G[k]$)
3. **Comparação** (*recarrega* os temporários e avalia a condição):
   (a) `LOAD_FAST 2` ($push$ `FAST[2]` $= a$)
   (b) `LOAD_FAST 3` ($push$ `FAST[3]` $= b$)
   (c) `COMPARE_OP >` ($consome\ a, b$ e $produz$ booleano)
   (d) `POP_JUMP_FORWARD_IF_FALSE off` ($pop$ condição; salta se $a \leq b$)
4. **Preparação e chamada de função** (executada apenas se $a > b$):
   (a) `LOAD_FAST 3` ($push\ b$)
   (b) `LOAD_FAST 2` ($push\ a$)
   (c) `CALL_FUNCTION` ($swap$)
5. **Troca e retorno para função chamadora**:
   (a) `STORE_GLOBAL k+1` ($G[k+1] \leftarrow a$)
   (b) `STORE_GLOBAL k` ($G[k] \leftarrow b$)
   (c) `RETURN_VALUE 0`

## 5.4 Ambiente de simulação

Os *testbenches* utilizam dois domínios de relógio: um para o processador em 200 $MHz$ (frequência-alvo obtida em síntese lógica, apresentado na próxima seção) e um para o GDM a 133 $MHz$, mais lento, de modo a exercitar a travessia de domínios. A comunicação entre o DUT e GDM é realizada por duas instâncias da interface de *handshake* de quatro fases, em canais independentes: no canal **REQ** o ProPy atua como *fonte* (emissor) e o GDM como *destino* (consumidor); no canal **RSP** os papéis se invertem. O testbench disponibiliza *tasks* genéricas de emissão e captura que impõem o protocolo *handshake* nível-a-nível $VALID \uparrow \rightarrow READY \uparrow \rightarrow VALID \downarrow \rightarrow READY \downarrow$, garantindo *return-to-zero* entre transferências e captura única por nível.

**Fases temporais e contagem de ciclos** - Para comparação entre as duas arquiteturas, sem cache e com cache de dados, os tempos de execução para cada etapa do sistema, baseados nos ciclos de relógio do processador, são contabilizados por fase de operação:

- `cycles_load`: do primeiro $VALID = 1'b1$ do *frame* até `tx_done` $= 1'b1$ e $VALID = 1'b0$ (carregamento completo). No caso com subrotinas, o contador é acrescido a cada nova fase de carregamento, assegurando que todo o custo de *load* seja computado.

- `cycles_run`: do primeiro `proc_busy_o` $= 1'b1$ pós-*load* até a emissão de `EOF` ou entrada no estágio de escrita de retorno no caso da arquitetura com *cache* de dados *on-chip*. No caso com subrotinas, o contador é acrescido a cada nova fase de execução, assegurando que todo o custo de *run* seja computado.

- `cycles_wb`: do início do estado de escrita de retorno até `tx_done` $= 1'b1$ concluindo todas as transferências. No caso com subrotinas, o contador é acrescido a cada nova fase de escrita de retorno, assegurando que todo o custo de *wb* seja computado.

A partir desses contadores, define-se o total:

$$\mathit{CiclosTotais} = \texttt{cycles\_load} + \texttt{cycles\_run} + \texttt{cycles\_wb}.$$

E dessa forma, conseguimos obter as razões:

$$CPI_{efetivo} = \frac{CiclosTotais}{\texttt{\#inst}} \qquad e \qquad IPC = \frac{1}{CPI_{efetivo}},$$

O CPI efetivo quantifica o número médio de ciclos por instrução ao longo de uma execução completa, incorporando paralisações no pipeline e latências de comunicação. Já o IPC mede a vazão média do processador em instruções por ciclo e é o inverso do CPI. Essas métricas permitem realizarmos uma comparação de desempenho direta entre as duas arquiteturas.

### 5.4.1 Métricas para o caso de teste

Para a aplicação de referência, *Bubble Sort* ($N$=10), foi gerado um *frame* com 542 instruções estáticas no caso de teste com um único *frame* e, 686 instruções para o caso de teste com chamadas de subrtorina, sob a ISA suportada pelo ProPy. O conjunto de instruções no caso de teste com chamadas de subrotina é menor uma vez que as instruções para a troca são reutilizadas, A seguir, são apresentados os resultados por implementação (com e sem cache de dados), para a execução dos casos de teste e, na sequência, os resultados do *benchmark* em software para a aplicação de referência, utilizando a implementação oficial do Python 3.11, avaliando o desempenho em um processador de propósito geral.

O ambiente empregado para a versão de referência em software utiliza o processador AMD Ryzen 7 2700x (sem *overclock*) e 16 GB de RAM a 2400 MHz, executando o sistema operacional Ubuntu 24.04.3 LTS no subsistema WSL para Windows. As medições foram obtidas com a ferramenta ***perf***, disponível nas ferramentas genéricas do sistema operacional, configurada para: (i) fixar a execução da aplicação Python a um único núcleo lógico (`taskset`), mitigando variações por migração de *threads*; e (ii) coletar contadores apenas em espaço de usuário (`cycles:u`, `instructions:u`), isolando o custo da aplicação sem interferência do *kernel* do sistema operacional. As métricas reportadas correspondem à média de cinco execuções sob a mesma configuração. Ressalta-se que, ao medir a execução em software, quantifica-se o custo de operação da PVM, responsável por interpretar os *bytecodes* e traduzi-los em instruções nativas em tempo de execução; esse caminho interpretativo contrasta com o ProPy, que executa nativamente um subconjunto desses *bytecodes*.

A Tabela 6 apresenta a decomposição do tempo para tratamento do *frame* único na variante com acesso externo a dados, evidenciando os custos de carregamento exclusivo das instruções e os custos de execução (RUN) devido a acessos de memória sob demanda, que introduzem paralisações no pipeline. Não há fase de escrita de retorno (WB) nesta configuração, uma vez que os dados já são manipulados na memória externa durante a fase de execução. Registra-se, ainda, a distinção entre o conjunto de instruções estáticas carregadas e o subconjunto efetivamente executado para o vetor de entrada considerado, reflexo do fluxo de controle do algoritmo, com caminhos não tomados durante a execução.

| Fase | Ciclos | % do total | Tempo @ 200 MHz | Observação |
|---|---|---|---|---|
| Carregamento (LOAD) | 8138 | 57,25% | 40,690 $\mu$s | Instruções do *frame*. |
| Execução (RUN) | 6077 | 42,75% | 30,385 $\mu$s | Acesso externo sob demanda. |
| Escrita de retorno (WB) | — | — | — | Não aplicável nesta variante. |
| **Total** | **14215** | **100%** | **71,075 $\mu$s** | |
| **Instruções carregadas (estáticas do *frame*):** | | | | **542** |
| **Instruções executadas (para {42, 17, 93, 0, 5, 77, 77, 12, 9, 1}):** | | | | **474** |
| **CPI efetivo:** | | | | **29,99** |
| **IPC:** | | | | **0,0333** |

Tabela 6 – *Frame* único: Fases temporais e contagem de ciclos — arquitetura sem *cache* de dados.

A Tabela 7 apresenta a decomposição temporal para tratamento do *frame* único da variante com *cache* de dados, na qual o carregamento inicial (LOAD) inclui instruções e dados do *frame*. A execução (RUN) ocorre de forma local, com acesso a dados *on-chip*, reduzindo paralisações no pipeline. A fase de escrita de retorno (WB) limita-se aos dados modificados durante a fase de execução.

| Fase | Ciclos | % do total | Tempo @ 200 MHz | Observação |
|---|---|---|---|---|
| Carregamento (`LOAD`) | 8304 | 92,41% | 41,520 $\mu$s | Instruções + dados do *frame*. |
| Execução (`RUN`) | 461 | 5,13% | 2,305 $\mu$s | Acesso a dados *on-chip*. |
| Escrita de retorno (`WB`) | 221 | 2,46% | 1,105 $\mu$s | Varredura de linhas *dirty*. |
| **Total** | **8986** | **100%** | **44,930 $\mu$s** | |
| **Instruções carregadas (estáticas do *frame*):** | | | | **542** |
| **Instruções executadas (para {42, 17, 93, 0, 5, 77, 77, 12, 9, 1}):** | | | | **474** |
| **CPI efetivo:** | | | | **18,96** |
| **IPC:** | | | | **0,0527** |

Tabela 7 – *Frame* único: Fases temporais e contagem de ciclos — arquitetura com *cache* de dados.

**Benchmark do algoritmo em software (CPython 3.11) sem subrotina** Conforme o procedimento descrito (contadores em espaço de usuário, espaço de execução, mediana/repetição), na tabela 8 consolidou-se a execução do *Bubble Sort* ($N$=10) com os valores extraídos pelo *benchmark* `perf`.

| Métrica | Valor | Observação |
|---|---|---|
| Ciclos (`cycles:u`) | 57193144 | Espaço de usuário |
| Instruções (`instructions:u`) | 72005891 | Espaço de usuário |
| Tempo total | 22,38 ms | Estimada pelo `perf` |
| Frequência média | 2555,55 MHz | Estimada pelo `perf` |
| $CPI_{efetivo}$ | 0,794 | $CPI = cycles/instructions$ |
| $IPC$ | 1,259 | $IPC = 1/CPI$ |

Tabela 8 – Métricas do *benchmark* em software (CPython 3.11) para o *Bubble Sort* ($N$=10) sem subrotinas.

Para o caso de teste com chamadas de subrotinas, a Tabela 9 apresenta a decomposição temporal da variante com acesso externo a dados, explicitando os custos de `LOAD` de instruções, contabilizando todas as trocas de *frame* e os custos de `RUN` com acessos à memória externa sob demanda, que introduzem paralisações no pipeline.

| Fase | Ciclos | % do total | Tempo @ 200 MHz | Observação |
|---|---|---|---|---|
| Carregamento (`LOAD`) | 136260 | 76,56% | 681,300 $\mu$s | Instruções. |
| Execução (`RUN`) | 41723 | 23,44% | 208,615 $\mu$s | Acesso externo sob demanda. |
| Escrita de retorno (`WB`) | — | — | — | Não aplicável nesta variante. |
| **Total** | **177983** | **100%** | **889,915 $\mu$s** | |
| **Instruções carregadas (estáticas do *frame*):** | | | | **686** |
| **Instruções executadas (para {42, 17, 93, 0, 5, 77, 77, 12, 9, 1}):** | | | | **632** |
| **CPI efetivo:** | | | | **281,62** |
| **IPC:** | | | | **0,0036** |

Tabela 9 – Aplicação com subrotinas: Fases temporais e contagem de ciclos — arquitetura sem *cache* de dados.

A Tabela 10 apresenta a decomposição temporal para tratamento do caso de teste com chamadas de subrotina na variante com *cache* de dados. Assim como na variante com acesso a dados externos, os custos de `LOAD` de instruções contabilizam todas as trocas de *frame*. Os custos de `RUN` não apresentam paralisação do pipeline, sendo o custo de escrita de retorno ao GDM contabilizado no estágio de `WB`. A execução (`RUN`) ocorre de forma local, com acesso a dados *on-chip*, reduzindo paralisações no pipeline. A fase de escrita de retorno (`WB`) limita-se aos dados modificados durante a fase de execução.

| Fase | Ciclos | % do total | Tempo @ 200 MHz | Observação |
|---|---|---|---|---|
| Carregamento (`LOAD`) | 140460 | 97,30% | 702,300 $\mu$s | Instruções + dados do *frame*. |
| Execução (`RUN`) | 758 | 0,53% | 3,790 $\mu$s | Acesso a dados *on-chip*. |
| Escrita de retorno (`WB`) | 3134 | 2,17% | 15,670 $\mu$s | Varredura de linhas *dirty*. |
| **Total** | **144352** | **100%** | **721,760 $\mu$s** | |
| **Instruções carregadas (estáticas do *frame*):** | | | | **686** |
| **Instruções executadas (para {42, 17, 93, 0, 5, 77, 77, 12, 9, 1}):** | | | | **632** |
| **CPI efetivo:** | | | | **228,40** |
| **IPC:** | | | | **0,0044** |

Tabela 10 – Aplicação com subrotinas: Fases temporais e contagem de ciclos — arquitetura com *cache* de dados.

**Benchmark do algoritmo em software (CPython 3.11) com subrotina** Assim como realizado para o algoritmo sem subrotina, foi seguido o procedimento descrito (contadores em espaço de usuário, espaço de execução, mediana/repetição), consolidando os valores extraídos na execução do *Bubble Sort* ($N$=10) pelo *benchmark* `perf` na tabela 11.

| Métrica | Valor | Observação |
|---|---|---|
| Ciclos (`cycles:u`) | 58108478 | Espaço de usuário |
| Instruções (`instructions:u`) | 72925986 | Espaço de usuário |
| Tempo total | 22,14 ms | Estimada pelo `perf` |
| Frequência média | 2623,49 MHz | Estimada pelo `perf` |
| $CPI_{efetivo}$ | 0,783 | $CPI = cycles/instructions$ |
| $IPC$ | 1,277 | $IPC = 1/CPI$ |

Tabela 11 – Métricas do *benchmark* em software (CPython 3.11) para o *Bubble Sort* ($N$=10) com subrotinas.

### 5.4.2 Análise comparativa

A variação no *CPI* nas duas implementações do ProPy evidencia o efeito do paradigma *carregar–executar–salvar*, viabilizado pela *cache* de dados *on-chip*. Quando a execução depende de acessos externos, a etapa intra-*frame* sofre paralisações recorrentes em cada `LOAD`/`STORE`; com a *cache* de dados, os acessos tornam-se locais ao estágio EX, e o custo de comunicação é deslocado para as fronteiras do *frame*. Nessa configuração, para o *frame* único, as transações com o GDM passam a concentrar $\approx 94{,}87\%$ do tempo total, distribuídas entre o pré-carregamento do *frame* ($\approx 92{,}41\%$) e o *write-back* apenas das posições *dirty* ($\approx 2{,}46\%$), enquanto a execução propriamente dita permanece curta e local ($\approx 5{,}13\%$), com leituras/escritas concluídas em um único ciclo.

Em comparação direta com a implementação sem *cache* de dados, o total de ciclos reduz-se em $\approx 36{,}8\%$ (14215 $\rightarrow$ 8986), o que se traduz em aceleração de $\approx 1{,}58\times$. O *CPI* global cai de 29,99 para 18,96 e, reciprocamente, o *IPC* aumenta, refletindo a remoção do gargalo de memória durante a execução e a reestruturação do custo para operações de borda do *frame*.

Avaliando a execução de referência em CPython 3.11, foram consumidos, em média, $\sim$22,38 *ms* (medição *user-space* com `perf`, delimitada a um único núcleo), emitindo 72005891 instruções nativas para realizar o *Bubble Sort* de $N$=10. Em contraste, o *frame*

equivalente sob a ISA do ProPy exige apenas 474 instruções. Essa diferença de ordem de grandeza

$$\frac{72005891}{474} \approx 1{,}51 \times 10^5 \approx 151911{,}2 \; vezes$$

decorre do overhead interpretativo da PVM, inexistente no ProPy, cujo conjunto de instruções mapeia diretamente os *bytecodes* relevantes e opera com dados *on-chip* na variante com *cache*.

Apesar de o CPython apresentar $CPI \approx 0{,}794$ e $IPC \approx 1{,}26$ (valores intrínsecos à microarquitetura do processador hospedeiro e, portanto, *não* comparáveis diretamente entre ISAs), o tempo de solução é significativamente maior. Com *cache* de dados, o ProPy conclui o caso de teste em 44,93 $\mu s$; sem *cache* de dados, em 71,08 $\mu s$, ambos a 200 $MHz$. Os *speedups* relativos são:

$$ProPy_{cache} = \frac{22{,}38\ ms}{44{,}93\ \mu s} \approx 498\times, \qquad ProPy_{semcache} = \frac{22{,}38\ ms}{71{,}08\ \mu s} \approx 314\times.$$

Avaliando um cenário de prototipação, de forma a estimar o impacto da latência externa no acesso a DDR SRAM pelo GDM, considerou-se uma penalidade de 40 ciclos por transferência no canal de *handshake* entre ProPy e GDM para valores inteiros. Na arquitetura *sem cache de dados*, a penalidade totaliza 75080 ciclos adicionais, e na arquitetura *com cache de dados*, a penalidade agrega 24 000 ciclos. Os valores de CPI e IPC, tempo de execução e *speedup* são atualizados conforme a tabela 12.

| **Arquitetura** | **Penalidade (ciclos)** | **$CPI_{efetivo}$** | **IPC** | **T [ms]** | **Speedup vs. SW** |
|---|---|---|---|---|---|
| Sem *cache* de dados | 75 080 | 188,39 | 0,00531 | 0,446475 | $\approx 50\times$ |
| Com *cache* de dados | 24 000 | 69,59 | 0,01436 | 0,16493 | $\approx 135\times$ |

Tabela 12 – Normalização das métricas de desempenho do ProPy a 200 MHz para a execução do *frame* único (referência em software: 22,38 ms).

No caso de teste com subrotina, o carregamento torna-se ainda mais predominante, consumindo $\approx 76{,}56\%$ e $\approx 97{,}30\%$ do tempo total nas arquiteturas sem e com *cache* de dados, respectivamente. Na variante com acesso externo a dados, a execução encarece por estar atrelada a paralisações para comunicação com o GDM; já na variante com *cache* de dados, a execução é ínfima (não chega a 1%) e a escrita de retorno também exerce baixa influência ($\approx 2{,}17\%$).

Em comparação direta com a implementação sem *cache* de dados, o total de ciclos reduz-se em $\approx 18{,}89\%$ (177 983 $\rightarrow$ 144 352), resultando em aceleração de $\approx 1{,}23\times$. O

*CPI* global praticamente se mantém, caindo de 281,620 para 228,940; reciprocamente, o *IPC* aumenta, refletindo a remoção do gargalo de memória durante a execução e a realocação de custo para operações de borda do *frame*. Fica evidente que o acesso à memória é o limitante de desempenho: em *frames* com milhares de instruções, o carregamento tende a dominar o tempo total em ambas as variantes.

Frente à referência em software, a disparidade de instruções dinâmicas persiste — 632 no ProPy com subrotina contra 72 925 986 no software. Os tempos de solução são ∼22,14 *ms* no CPython, 721,76 $\mu s$ na arquitetura com *cache* de dados e 889,91 $\mu s$ na arquitetura sem *cache*, levando aos *speedups*:

$$ProPy_{cache} = \frac{22{,}14\ ms}{721{,}76\ \mu s} \approx 30\times, \qquad ProPy_{sem\,cache} = \frac{22{,}14\ ms}{889{,}91\ \mu s} \approx 25\times .$$

Assim como no caso sem subrotinas, também se quantificou o impacto da latência externa no acesso à DDR/SRAM pelo GDM, considerando penalidade de 40 ciclos por transferência: na arquitetura *com cache de dados*, a penalidade agrega 404 520 ciclos; na arquitetura *sem cache de dados*, 382 120 ciclos. Os valores de CPI e IPC, tempo de execução e *speedup* são atualizados conforme a tabela 13.

| **Arquitetura** | **Penalidade (ciclos)** | **$CPI_{efetivo}$** | **IPC** | **T [ms]** | **Speedup vs. SW** |
|---|---|---|---|---|---|
| Sem *cache* de dados | 382 120 | 886,24 | 0,00113 | 2,800515 | $\approx 8\times$ |
| Com *cache* de dados | 404 520 | 868,47 | 0,0115 | 2,74436 | $\approx 8\times$ |

Tabela 13 – Normalização das métricas de desempenho do ProPy a 200 MHz para a execução com chamadas de subrotina (referência em software: 22,14 ms).

## 5.5 Síntese lógica

A síntese lógica foi realizada com a ferramenta **Cadence Genus Synthesis Solution** (versão `23.14-s090_1`) e a tecnologia de células adotada foi a biblioteca de 28 nm da **STMicroelectronics**. Todo o projeto foi descrito em **SystemVerilog** (RTL), contemplando as duas variantes de topo (*sem cache* e *com cache* de dados). As mesmas restrições temporais foram aplicadas a ambas as variantes, com **clock nominal de 200 MHz**, uma vez que o caminho crítico em ambas as arquiteturas está localizado na divisão inteira do módulo ALU, e assegura comparabilidade direta dos relatórios de *área*, *frequência máxima atingível ($F_{max}$)* e *potência* apresentados nas subseções seguintes.

A Tabela 14 consolida, por topo, a frequência máxima, potência de operação e a área total

após síntese lógica.

Tabela 14 – Consolidação dos resultados de síntese para o ProPy

| Arquitetura | Estágios de Pipeline | Células Lógicas Totais | Células Lógicas Caches | Frequência de Operação | Potência de Operação |
|---|---|---|---|---|---|
| ProPy com cache de instruções | 3 | 36.582 | 26.212 (71.65%) | 200 MHz | 17,575 mW |
| ProPy com cache de dados e instruções | 3 | 100.148 | 79.378 (79.26%) | 200 MHz | 44,430 mW |

### 5.5.1 Área ocupada em ASIC

A área ocupada em síntese reflete os parâmetros definidos para os blocos de memória parametrizáveis da arquitetura, assim como ao tamanho de dados definidos na arquitetura. Para o caso de teste do *Bubble Sort* ($N$=10), o *frame* gerado possui 542 instruções. Assim, a cache de instruções foi dimensionado para 10 bits de endereçamento ($2^{10}$), resultando em 1024 palavras/endereços e, 16 bits de largura, seguindo o padrão da linguagem Python. Na variante com *cache* de dados, o mesmo dimensionamento, 10 bits de endereço, foi adotado para a *cache* de dados, porém com a largura de dados de 32 bits, tendo em vista que esse é o tamanho definido para os dados da arquitetura. Os metados *sideband* da *cache* (tipo, 3 bits, *dirty*, 1 bit) acompanham o dimensionamento de 10 bits de endereços da cache de dados. A pilha de avaliação foi configurada com profundidade de 8 elementos e dados de 32 bits, disponibilizando espaço suficiente para tratar variáveis em tempo de execução para o *Bubble Sort* exemplo, e ficar em conformidade com o tamanho de dados utilizados na arquitetura.

As Tabelas 15 e 16 apresentam a distribuição de recursos em síntese, respectivamente, para a arquitetura sem e com *cache* de dados, destacando a participação percentual de cada instância na área total. Adota-se a seguinte nomenclatura: `u_inst_mem` denota a *cache* de instruções; `u_alu`, a ALU; `u_data_stack`, a pilha de avaliação; `u_hds_req`, o controle de *handshake* no qual o ProPy atua como mestre; `u_hds_rsp`, o controle de *handshake* no qual o ProPy atua como escravo; e, por fim, `u_data_mem`, a *cache* de dados. Essa organização evidencia quais blocos dominam em área e interconexão, facilitando a leitura comparativa entre as duas variantes.

Tabela 15 – Distribuição de recursos por bloco — arquitetura sem cache de dados

| **Bloco** | **Células** | **Área Total ($\mu$m$^2$)** |
|---|---|---|
| Memória de instruções (`u_inst_mem`) | 26,212 | 87,995,753 |
| | ≈ 71,65% | ≈ 82,67% |
| ALU (`u_alu`) | 9,056 | 14,436,077 |
| | ≈ 24,76% | ≈ 13,56% |
| Pilha de dados (`u_data_stack`) | 778 | 2,222,694 |
| | ≈ 2,13% | ≈ 2,09% |
| Handshake externo (`u_hds_req + rsp`) | 118 | 493,294 |
| | ≈ 0,32% | ≈ 0,46% |
| Demais blocos | 418 | 1,293,715 |
| | ≈ 1,14% | ≈ 1,22% |
| **Total do design** | **36,582** | **106,441,533** |
| | **100,00%** | **100,00%** |

Tabela 16 – Distribuição de recursos por bloco — arquitetura com cache de dados

| **Bloco** | **Células** | **Área Total ($\mu$m$^2$)** |
|---|---|---|
| Cache/memória de dados (`u_data_mem`) | 53,662 | 175,515,860 |
| | ≈ 53,58% | ≈ 57,04% |
| Memória de instruções (`u_inst_mem`) | 25,716 | 87,420,336 |
| | ≈ 25,68% | ≈ 28,41% |
| ALU (`u_alu`) | 9,314 | 14,935,785 |
| | ≈ 9,30% | ≈ 4,85% |
| Pilha de dados (`u_data_stack`) | 882 | 2,140,148 |
| | ≈ 0,88% | ≈ 0,70% |
| Handshake externo (`u_hds_req + rsp`) | 119 | 498,500 |
| | ≈ 0,12% | ≈ 0,16% |
| Demais blocos | 10,455 | 27,187,275 |
| | ≈ 10,44% | ≈ 8,84% |
| **Total do design** | **100,148** | **307,697,904** |
| | **100,00%** | **100,00%** |

O restante da área provém da lógica combinacional e sequencial: *front-end* do *pipeline* (IF/ID), ALU, controle da pilha, máquinas de estados (carregamento, EX/MA e `WB`), mapeador de endereços por contexto e interfaces de *handshake* com sincronizadores de domínio. A variante com *cache* de dados adiciona a FSM de *write-back*, comparadores/contadores para varredura de linhas *dirty* e registradores de metadados; todavia, esses acréscimos são modestos quando comparados ao impacto das caches de instruções e dados. Em complemento, os valores agregados para ambas as variantes encontram-se nas Tabelas 17 e 18. Nessas tabelas, a coluna *Cell-Count* indica a contagem de células instanciadas; *Cell-Area* reporta a área ocupada pelas células; *Net-Area* corresponde à área estimada da fiação entre células; e *Total-Area* reflete o orçamento de área efetivo do design, somando *Cell-Area* e *Net-Area*.

Tabela 17 – Resumo de área de síntese — arquitetura sem cache de dados

| **Instance** | **Cell-Count** | **Cell-Area ($\mu$m$^2$)** | **Net-Area ($\mu$m$^2$)** | **Total-Area ($\mu$m$^2$)** |
|---|---|---|---|---|
| ppy_nocache_top | 36582 | 84190.310 | 22251.223 | 106441.533 |
| u_alu | 9056 | 9747.120 | 4688.957 | 14436.077 |
| u_data_stack | 778 | 1785.898 | 436.796 | 2222.694 |
| u_hds_req | 92 | 364.915 | 16.377 | 381.292 |
| u_hds_rsp | 26 | 107.549 | 4.454 | 112.002 |
| u_inst_mem | 26212 | 71440.637 | 16555.116 | 87995.753 |

Tabela 18 – Resumo de área de síntese — arquitetura com cache de dados

| **Instance** | **Cell-Count** | **Cell-Area ($\mu$m$^2$)** | **Net-Area ($\mu$m$^2$)** | **Total-Area ($\mu$m$^2$)** |
|---|---|---|---|---|
| ppy_wcache_top | 100148 | 246337.834 | 61360.071 | 307697.904 |
| u_alu | 9314 | 10080.701 | 4855.084 | 14935.785 |
| u_data_mem | 53662 | 142354.138 | 33161.722 | 175515.860 |
| u_data_stack | 882 | 1692.384 | 447.764 | 2140.148 |
| u_hds_req | 60 | 241.046 | 10.486 | 251.532 |
| u_hds_rsp | 59 | 236.640 | 10.328 | 246.968 |
| u_inst_mem | 25716 | 71025.946 | 16394.390 | 87420.336 |

A distribuição de recursos da arquitetura ***sem cache de dados*** evidencia a dominância da cache de instruções: são 26212 células ($\approx$ 71,65% do total) responsáveis por 87995,753 de área ($\approx$ 82,67%), além de concentrarem $\approx$ 84,86% da *cell-area* e $\approx$ 74,40% da

*net-area*; em contraste, a ALU reúne 9056 células ($\approx$ 24,76%) para 14436,077 de área ($\approx$ 13,56%), contribuindo com $\approx$ 11,58% da *cell-area* e $\approx$ 21,07% da *net-area*. Os demais blocos têm participação modesta: a pilha de dados soma 778 células ($\approx$ 2,13%) e 2222,694 de área ($\approx$ 2,09%), o *handshake* externo (`u_hds_req`+`u_hds_rsp`) agrega 118 células ($\approx$ 0,32%) e 493,294 de área ($\approx$ 0,46%), enquanto os blocos residuais completam 418 células ($\approx$ 1,14%) e 1293,715 de área ($\approx$ 1,22%). Já na arquitetura ***com cache de dados***, a inclusão da hierarquia de dados desloca a massa dominante para a **cache de dados**: 53662 células ($\approx$ 53,58%) e 175515,860 de área ($\approx$ 57,04%); a memória de instruções passa a 25716 células ($\approx$ 25,68%) e 87420,336 de área ($\approx$ 28,41%), enquanto a ALU permanece em patamar semelhante de células (9314; $\approx$ 9,30%), mas com participação relativa de área reduzida (14935,785; $\approx$ 4,85%) devido à diluição frente às caches. A pilha de dados representa 882 células ($\approx$ 0,88%) e 2140,148 de área ($\approx$ 0,70%); o *handshake* externo totaliza 119 células ($\approx$ 0,12%) e 498,500 de área ($\approx$ 0,16%); os demais blocos elevam-se para 10455 células ($\approx$ 10,44%) e 27187,275 de área ($\approx$ 8,84%). Em termos absolutos, a variante com cache aumenta o número de células de 36582 para 100148 ($\sim 2{,}74\times$) e a área total de 106441,533 para 307697,904 ($\sim 2{,}89\times$).

### 5.5.2 Frequência máxima de operação

A frequência máxima ($F_{max}$) foi determinada por Análise Estática de Tempo (STA) após síntese, a partir do critério

$$T_{clk} \;\geq\; D_{crtico} + T_{setup} \quad \Rightarrow \quad F_{max} \;=\; \frac{1}{T_{clk,min}}.$$

Inicialmente uma restrição de clock de 500 $MHz$ foi utilizada; em seguida, procedeu-se ao ajuste iterativo do período até que o *worst negative slack* (WNS) se tornasse não negativo ($WNS \geq 0$). Em ambas as variantes arquiteturais o caminho crítico está localizado no estágio *EX*, na ALU, devido à implementação de divisão inteira em um único ciclo, implementada com o operando padrão do System Verilog (/), que impõe o maior atraso de dados do design. Em testes adicionais, removendo-se o suporte à divisão (mantidas as demais condições de síntese), o caminho crítico foi deslocado e foi possível atingir $F_{max} \approx 500\ MHz\ \ (T_{clk} = 2{,}0\ ns)$.

As figuras 24 e 25 mostram os *slacks* finais alcançados para ambas as variantes do ProPy,

tendo ambas a mesma velocidade máxima final

$$F_{max} = 200\ MHz\ \ (T_{clk} = 5{,}0\ ns)\,.$$

A introdução da cache de dados não altera $F_{max}$, pois o caminho crítico permanece na ALU. Estratégias como (i) pipeline do divisor ou (ii) execução multi-ciclo para divisão tendem a reduzir $D_{crtico}$ do estágio *EX*, viabilizando frequências superiores sob as mesmas condições de síntese.

Figura 24 – *Slack* para a arquitetura sem cache de dados

```
@genus:root: 12> report_timing
============================================================
  Generated by:           Genus(TM) Synthesis Solution 23.14-s090_1
  Generated on:           Oct 08 2025  12:33:52 am
  Module:                 ppy_nocache_top
  Operating conditions:   _nominal_
  Interconnect mode:      global
  Area mode:              physical library
============================================================


Path 1: MET (0 ps) Setup Check with Pin u_data_stack/memory_reg_7_1/CP->D
        Group: clk_ppy_i
   Startpoint: (R) u_data_stack/depth_q_reg_2/CP
        Clock: (R) clk_ppy_i
     Endpoint: (F) u_data_stack/memory_reg_7_1/D
        Clock: (R) clk_ppy_i

                  Capture       Launch
     Clock Edge:+    5000            0
     Src Latency:+      0            0
     Net Latency:+      0 (I)        0 (I)
         Arrival:=    5000            0

           Setup:-      81
   Required Time:=    4919
    Launch Clock:-       0
       Data Path:-    4919
           Slack:=       0
```

Fonte: (Autor, 2025)

Figura 25 – *Slack* para a arquitetura com cache de dados

```
@genus:root: 11> report_timing
============================================================
  Generated by:           Genus(TM) Synthesis Solution 23.14-s090_1
  Generated on:           Oct 06 2025  03:32:33 am
  Module:                 ppy_wcache_top
  Operating conditions:   _nominal_
  Interconnect mode:      global
  Area mode:              physical library
============================================================


Path 1: MET (0 ps) Setup Check with Pin u_data_stack/memory_reg_7_1/CP->D
        Group: clk_ppy_i
   Startpoint: (R) u_data_stack/depth_q_reg_1/CP
        Clock: (R) clk_ppy_i
     Endpoint: (F) u_data_stack/memory_reg_7_1/D
        Clock: (R) clk_ppy_i

                  Capture       Launch
     Clock Edge:+    5000            0
     Src Latency:+      0            0
     Net Latency:+      0 (I)        0 (I)
         Arrival:=    5000            0

           Setup:-      81
   Required Time:=    4919
    Launch Clock:-       0
       Data Path:-    4919
           Slack:=       0
```

Fonte: (Autor, 2025)

### 5.5.3 Potência de operação

A arquitetura ProPy sem cache de dados possui um consumo total de **17,575 mW**, enquanto a arquitetura com cache de dados possui um consumo total de **44,430 mW**.
0.38 A arquitetura com cache de dados apresenta aumento de potência de aproximadamente 2,53× em relação à variante sem cache, conforme figuras 26 e 27. Esse incremento está concentrado na componente *internal* de células sequenciais, coerente com a maior quantidade de estados e controle associados ao protocolo *Carregar–Executar–Salvar* e ao estágio adicional de *write-back*. Em termos relativos, a parcela *switching* diminui (de 22,15% para 12,05%) porque o crescimento da *internal* é mais pronunciado. O *leakage* também aumenta em termos absolutos, porém mantém participação pequena no total.

Figura 26 – Potência de operação para a arquitetura sem cache de dados

```
@genus:root: 14> report_power -unit mW
Info    : Joules engine is used. [RPT-16]
        : Joules engine is being used for the command report_power.
Instance: /ppy_nocache_top
Power Unit: mW
PDB Frames: /stim#0/frame#0
  ------------------------------------------------------------------------
  Category         Leakage     Internal    Switching        Total    Row%
  ------------------------------------------------------------------------
    memory     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
  register     4.06089e-01  1.11432e+01  3.85504e-02  1.15878e+01  65.93%
     latch     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
     logic     1.65184e-01  1.96865e+00  3.85351e+00  5.98735e+00  34.07%
      bbox     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
     clock     0.00000e+00  0.00000e+00  1.45800e-04  1.45800e-04   0.00%
       pad     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
        pm     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
  ------------------------------------------------------------------------
  Subtotal     5.71273e-01  1.31118e+01  3.89221e+00  1.75753e+01 100.00%
Percentage           3.25%       74.60%       22.15%      100.00% 100.00%
  ------------------------------------------------------------------------
```

Fonte: (Autor, 2025)

Figura 27 – Potência de operação para a arquitetura com cache de dados

```
@genus:root: 13> report_power -unit mW
Info    : Joules engine is used. [RPT-16]
        : Joules engine is being used for the command report_power.
Instance: /ppy_wcache_top
Power Unit: mW
PDB Frames: /stim#0/frame#0
  ------------------------------------------------------------------------
    Category         Leakage     Internal    Switching        Total    Row%
  ------------------------------------------------------------------------
      memory     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
    register     1.30753e+00  3.53301e+01  4.05576e-02  3.66782e+01  82.55%
       latch     2.62071e-04  2.53262e-03  3.34719e-03  6.14189e-03   0.01%
       logic     3.83736e-01  2.05300e+00  5.30926e+00  7.74600e+00  17.43%
        bbox     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
       clock     0.00000e+00  0.00000e+00  1.45800e-04  1.45800e-04   0.00%
         pad     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
          pm     0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00   0.00%
  ------------------------------------------------------------------------
    Subtotal     1.69152e+00  3.73856e+01  5.35332e+00  4.44304e+01  99.99%
  Percentage           3.81%       84.14%       12.05%      100.00% 100.00%
  ------------------------------------------------------------------------
```

Fonte: (Autor, 2025)

Para 200 $MHz$, a energia por ciclo é $E_{ciclo} \approx 87{,}9\ pJ/ciclo$ (`ppy_nocache_top`) e $E_{ciclo} \approx 222{,}2\ pJ/ciclo$ (`ppy_wcache_top`). A análise de *energia por tarefa* ($E_{frame} \approx E_{ciclo} \times$ `ciclos_totais`) não depende apenas da potência instantânea: como a variante com cache elimina *stalls* por acesso sob demanda ao GDM, o número de ciclos para concluir o *frame* tende a reduzir. Dessa forma, mesmo com maior $E_{ciclo}$, $E_{frame}$ pode ser menor na arquitetura com cache se a redução de ciclos for suficiente.

# 6 CONSIDERAÇÕES FINAIS

Ao final deste trabalho, é possível avaliar que os objetivos estabelecidos foram alcançados. Há, agora, um processador pipeline Python em *SystemVerilog*, parametrizável, apto a executar aplicações mapeadas para o subconjunto de *bytecodes* suportado, e que se comunica com o GDM, emulado em simulação, por meio do protocolo apresentado.

A arquitetura evolui e consolida a prova de conceito de (BITENCOURT, 2019), transformando-a em uma base utilizável. A validação funcional por meio do caso de teste para o algoritmo *Bubble Sort*, demonstra a capacidade da arquitetura em executar aplicações reais, embora necessária a manipulação dos bytecodes originais para os suportados pela ISA do ProPy no momento.

A caracterização de Potência, Performance e Área (PPA), obtida por síntese, evidencia que o caminho crítico que limita a frequência da arquitetura é imposto pela operação de divisão inteira implementada em um único ciclo na ALU. Na variante com *cache* de dados, o aumento de área e de potência é esperado e decorre da incorporação de memória *on-chip* e da lógica de controle associada ao esquema *Carregar–Executar–Salvar*, além do estágio de *write-back*. Em contrapartida, o pré-carregamento do contexto e a execução local do *frame* eliminam *stalls* por acesso externo ao GDM, reduzindo o número de ciclos para conclusão e, portanto, a energia por tarefa, ainda que a potência instantânea seja superior.

Permanecem como limitações imediatas: (i) cobertura parcial da ISA da PVM; (ii) ausência de suporte nativo a vetores e estruturas compostas recorrentes em *frames* reais; (iii) semântica incompleta para operações polimórficas dependentes de tipo; e (iv) divisor não pipelineado, atualmente responsável pelo caminho crítico. Para endereçar esses pontos, propõe-se, como trabalhos futuros, (a) extensão do subconjunto de *bytecodes* com suporte a vetores, objetos e métodos; (b) tratamento de diferentes tipos de dados pela ALU; (c) adoção de *pipelining* ou execução multi-ciclo para a operação de divisão; e (d) prototipação em FPGA integrando o processador ao GDM de (BENFICA, 2024), visando testes em bancada e demonstrações didáticas.

Em horizonte mais amplo, uma vez que a cobertura completa da PVM for alcançada, a arquitetura poderá atuar como alternativa em *hardware* para execução de aplicações Python, atuando como um co-processador/acelerador em diferentes sistemas.

# REFERÊNCIAS

BENFICA, T. R. Desenvolvimento de um gerenciador de contexto de memória para processador python. **UNIPAMPA**, 2024.

BITENCOURT, T. P. Implementação de um processador, em vhdl, para executar algoritmos escritos em python. **UNIPAMPA**, v. 1, n. 8, p. 159, 2019.

BROUGH, D. B.; WHEELER, D.; KALIDINDI, S. R. Materials knowledge systems in python—a data science framework for accelerated development of hierarchical materials. **Integrating materials and manufacturing innovation**, Springer, v. 6, p. 36–53, 2017.

CANNON, B. **Python Enhancement Proposal 339: Python Compiler**. 2005. <https://peps.python.org/pep-0339/>. Acessado em: 19 de junho de 2025.

CASS, S. The top programming languages 2024. **IEEE Spectrum**, aug 2024. Acessado em 23 de junho de 2025.

CUTTING, V.; STEPHEN, N. Comparative review of java and python. **International Journal of Research and Development in Applied Science and Engineering (IJRDASE)**, v. 21, n. 1, 2021.

DOGARU, R.; DOGARU, I. Optimization of gpu and cpu acceleration for neural networks layers implemented in python. In: IEEE. **2017 5th International Symposium on Electrical and Electronics Engineering (ISEEE)**. [S.l.], 2017. p. 1–6.

DYER, R.; CHAUHAN, J. An exploratory study on the predominant programming paradigms in python code. In: **Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2022. (ESEC/FSE 2022), p. 684–695. ISBN 9781450394130. Disponível em: https://doi.org/10.1145/3540250.3549158.

FERREIRA, R.; MAGALHÃES, S. V. G. de; NACIF, J. A. Métricas e números: Desmistificando a progra-mação de alto desempenho em gpu. **Minicursos do XXX Simpósio em Sistemas Computacionais de Alto Desempenho, WSCAD**, p. 5–34, 2019.

GREVERA, G.; UDUPA, J.; ODHNER, D. An order of magnitude faster isosurface rendering in software on a pc than using dedicated, general purpose rendering hardware. **IEEE Transactions on Visualization and Computer Graphics**, v. 6, n. 4, p. 335–345, 2000.

IKE-NWOSU, O. **Inside the Python Virtual Machine**. [S.l.]: Lean Publishing, 2015.

KWAME EZEKIEL MENSAH MARTEY, A. G. C. A. E. Qualitative assessment of compiled, interpreted and hybrid programming languages. **Communications on Applied Electronics**, Foundation of Computer Science (FCS), NY, USA, New York, USA, v. 7, n. 7, p. 8–13, Oct 2017. ISSN 2394-4714. Disponível em: https://www.caeaccess.org/archives/volume7/number7/764-2017652685/.

LAPLANTE, P. A. et al. **Dictionary of computer science, engineering and technology**. [S.l.]: CRC Press, 2017.

LINDHOLM, T. et al. **The Java virtual machine specification**. [S.l.]: Addison-wesley, 2013.

MAIER, O. et al. Pyqmri: an accelerated python based quantitative mri toolbox. **Journal of Open Source Software**, v. 5, n. 56, p. 2727, 2020.

MCGHAN, H.; O'CONNOR, M. Picojava: a direct execution engine for java bytecode. **Computer**, v. 31, n. 10, p. 22–30, 1998.

MILLMAN, K. J.; AIVAZIS, M. Python for scientists and engineers. **Computing in Science Engineering**, v. 13, n. 2, p. 9–12, 2011.

MITTAL, S.; VETTER, J. S. A survey of methods for analyzing and improving gpu energy efficiency. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 47, n. 2, p. 1–23, 2014.

MULLER, G. et al. Harissa: a flexible and efficient java environment mixing bytecode and compiled code. In: **Proceedings of the 3rd Conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 3**. USA: USENIX Association, 1997. (COOTS'97), p. 1.

NEWHALL, T.; MILLER, B. Performance measurement of interpreted programs. **Lecture Notes in Computer Science**, 06 1998.

O'CONNOR, J.; TREMBLAY, M. picojava-i: the java virtual machine in hardware. **IEEE Micro**, v. 17, n. 2, p. 45–53, 1997.

PEREIRA, D.; AYCOCK, J. **Instruction set architecture of Mamba, a new virtual machine for Python**. [S.l.], 2002.

PETTERS, T. **Python Enhancement Proposal 20: The Zen of Python**. 2004. <https://peps.python.org/pep-0020/>. Acessado em: 19 de junho de 2025.

POWER, R.; RUBINSTEYN, A. How fast can we make interpreted python? **CoRR**, abs/1306.6047, 2013. Disponível em: http://arxiv.org/abs/1306.6047.

PUFFITSCH, W.; SCHOEBERL, M. picojava-ii in an fpga. In: **Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems**. New York, NY, USA: Association for Computing Machinery, 2007. (JTRES '07), p. 213–221. ISBN 9781595938138. Disponível em: https://doi.org/10.1145/1288940.1288972.

RAYHAN, A. **The Rise of Python: A Survey of Recent Research**. 2023.

ROSSUM, G. V.; DRAKE, F. L. **Python 3 Reference Manual**. Scotts Valley, CA: CreateSpace, 2009. ISBN 1441412697.

SAABITH, A. S.; VINOTHRAJ, T.; FAREEZ, M. Popular python libraries and their application domains. **International Journal of Advance Engineering and Research Development**, v. 7, n. 11, 2020.

SCHOEBERL, M. A java processor architecture for embedded real-time systems. **Journal of Systems Architecture**, v. 54, n. 1, p. 265–286, 2008. ISSN 1383-7621. Disponível em: https://www.sciencedirect.com/science/article/pii/S1383762107000963.

SCHOENHOLZ, S. S.; CUBUK, E. D. Jax md: End-to-end differentiable, hardware accelerated, molecular dynamics in pure python. 2019.

SKALICKY, S. et al. Hot spicy: Improving productivity with python and hls for fpgas. In: **2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)**. [S.l.: s.n.], 2018. p. 85–92.

Stack Overflow. **Stack Overflow Developer Survey 2024**. 2024. <https://survey.stackoverflow.co/2024/>. Acessado em 23 de junho de 2025.

SURBIRYALA, J.; RONG, C. Cloud computing: History and overview. In: **2019 IEEE Cloud Summit**. [S.l.: s.n.], 2019. p. 1–7.

The IronPython Team. **IronPython: An open-source implementation of Python for .NET**. 2025. <https://ironpython.net>. Acessado em 23 de junho de 2025.

The Jython Project. **Jython: Python for the Java Platform**. 2025. <https://www.jython.org/>. Acessado em 23 de junho de 2025.

The PyPy Team. **PyPy**. 2025. <https://www.pypy.org/>. Acessado em 23 de junho de 2025.

TIOBE Software. **TIOBE Index**. 2025. <https://www.tiobe.com/tiobe-index/>. Acessado em 23 de junho de 2025.

TSAI, C.-J. et al. JAIP-MP: A Four-Core Java Application Processor for Embedded Systems. In: **IFIP Advances in Information and Communication Technology**. Daejeon, South Korea: [s.n.], 2015. (VLSI-SoC: Design for Reliability, Security, and Low Power, AICT-483), p. 170–192. Disponível em: https://inria.hal.science/hal-01578615.

TSAI, C.-J. et al. Jaip-mp: A four-core java application processor for embedded systems. In: SHIN, Y. et al. (Ed.). **VLSI-SoC: Design for Reliability, Security, and Low Power**. Cham: Springer International Publishing, 2016. p. 170–192. ISBN 978-3-319-46097-0.

WATTERS, A.; ROSSUM, G. V.; AHLSTROM, J. **Internet Programming with Python**. M&T Books, 1996. ISBN 9781558514843. Disponível em: https://books.google.com.br/books?id=WaYkkgEACAAJ.

YIYU, T. et al. A java processor with hardware-support object-oriented instructions. **Microprocessors and Microsystems**, v. 30, n. 8, p. 469–479, 2006. ISSN 0141-9331. Disponível em: https://www.sciencedirect.com/science/article/pii/S0141933105000967.