UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO
PRÓ-REITORIA DE GRADUAÇÃO
DEPARTAMENTO DE ENGENHARIAS E TECNOLOGIA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

WEDRYSON LUCAS XAVIER OLIVEIRA SOUZA

EXTENSÃO FUNCIONAL EM PROCESSADOR RISC-V PARA ACELERAÇÃO DE PRODUTO ESCALAR EM FPGA

PAU DOS FERROS - RN
2025

WEDRYSON LUCAS XAVIER OLIVEIRA SOUZA

EXTENSÃO FUNCIONAL EM PROCESSADOR RISC-V PARA ACELERAÇÃO DE PRODUTO ESCALAR EM FPGA

Monografia apresentada a Universidade Federal Rural do Semi-Árido como requisito para obtenção do título de Bacharel em Engenharia de Computação.

Orientador: Pedro Thiago Valerio de Souza, Prof. Dr.

Coorientador: Francisco de Assis Brito Filho, Prof. Dr.

PAU DOS FERROS - RN
2025

S719e Souza, Wedryson Lucas Xavier Oliveira.
Extensão funcional em processador RISC-V para aceleração de produto escalar em FPGA / Wedryson Lucas Xavier Oliveira Souza. - 2025.
59 f. : il.

Orientador: Pedro Thiago Valerio de Souza.
Coorientador: Francisco de Assis Brito Filho.
Monografia (graduação) - Universidade Federal Rural do Semi-árido, Curso de Engenharia de Computação, 2025.

1. RISC-V. 2. FPGA. 3. Aceleração de hardware. 4. Instruções customizadas. 5. LiteX. I. Souza, Pedro Thiago Valerio de, orient. II. Brito Filho, Francisco de Assis, co-orient. III. Título.

Ficha catalográfica elaborada por sistema gerador automático em conformidade com AACR2 e os dados fornecidos pelo) autor(a).
Biblioteca Campus Pau dos Ferros
Bibliotecária: Erlanda Maria Lopes da Silva
CRB: 15/966

WEDRYSON LUCAS XAVIER OLIVEIRA SOUZA

EXTENSÃO FUNCIONAL EM PROCESSADOR RISC-V PARA ACELERAÇÃO DE PRODUTO ESCALAR EM FPGA

Monografia apresentada a Universidade Federal Rural do Semi-Árido como requisito para obtenção do título de Bacharel em Engenharia de Computação.

Defendida em: 16 de dezembro de 2025

BANCA EXAMINADORA

_____
Pedro Thiago Valério de Souza, Prof. Dr. (UFERSA)
Presidente

_____
Franciso de Assis Brito Filho, Prof. Dr. (UFERSA)
Membro Examinador

_____
Petrúcio Ricardo Tavares de Medeiros, Prof. Dr. (UFERSA)
Membro Examinador

_____
Kennedy Reurison Lopes, Prof. Dr. (UFERSA)
Membro Examinador

AGRADECIMENTOS

Agradeço primeiramente a Deus, pela luz, sabedoria e coragem para seguir até aqui.

Manifesto minha profunda gratidão à minha mãe, pelo apoio incondicional e por sempre incentivar a realização dos meus sonhos e metas. Estendo meus agradecimentos à minha irmã, ao meu irmão, ao meu pai e a toda a minha família, pelo amor, suporte e compreensão em todos os momentos. Sou igualmente grato à minha namorada e amiga, Vivian Araújo, pelo apoio constante, pelo estímulo diário e pela paciência demonstrada especialmente nesta etapa final.

Ao meu orientador e amigo, Prof. Dr. Pedro Thiago Valerio de Souza, expresso minha sincera gratidão por todo o conhecimento compartilhado, pelas palavras de incentivo, pela confiança depositada em meu potencial e pela disponibilidade exemplar durante todas as fases do projeto.

Ao meu coorientador, Prof. Dr. Francisco de Assis Brito Filho, agradeço pelo suporte atento, por cada orientação precisa e por todo o auxílio prestado ao longo deste trabalho.

Agradeço também aos professores que marcaram minha formação acadêmica, tanto na UFERSA, quanto no EmbarcaTech, ao Prof. Me. Diego Vinicius Cirilo do Nascimento por todas as dúvidas esclarecidas e aos colegas que estiveram presentes na jornada, compartilhando desafios, aprendizados e momentos de descontração.

A todos que contribuíram direta ou indiretamente para a realização deste TCC, deixo registrados os meus mais sinceros e profundos agradecimentos, bem como à Banca Examinadora, pelo interesse e disponibilidade.

Se eu vi mais longe, foi por estar sobre ombros de gigantes.

Isaac Newton

RESUMO

Este trabalho apresenta o desenvolvimento e a avaliação de uma extensão funcional destinada à aceleração do produto escalar em um processador RISC-V implementado em *Field-Programmable Gate Array (FPGA)*, motivado pela necessidade de aumentar o desempenho computacional, foram projetadas três microarquiteturas: máquina de estados finitos sequencial, máquina de estados finitos com *pipeline* e *pipeline* registrado, com o objetivo de analisar o impacto de diferentes estratégias de paralelismo na redução de latência e aumento do *throughput*. A solução foi integrada a um *SoC* construído com o *framework* LiteX, utilizando registradores de controle e status como interface entre o processador PicoRV32 e o acelerador dedicado. A metodologia cobriu todo o fluxo de desenvolvimento em hardware digital, incluindo descrição RTL em SystemVerilog, simulação, síntese, *place-and-route* e validação em hardware real na FPGA ECP5 LFE5U-45. Os resultados experimentais evidenciaram melhorias substanciais no desempenho, com destaque para a arquitetura *pipeline* registrado, que atingiu latência de 4 ciclos e *throughput* de 120 Milhões de operações por segundo, superando significativamente a versão FSM básica. A integração hardware–software mostrou-se eficiente, e a validação cruzada confirmou a corretude funcional das três microarquiteturas. O trabalho apresenta um fluxo completo, reprodutível e apoiado integralmente em ferramentas *open-source* para o desenvolvimento de extensões funcionais na arquitetura RISC-V, demonstrando o potencial da abordagem para acelerar operações vetoriais em sistemas embarcados.

Palavras-Chave: RISC-V; FPGA; aceleração de hardware; instruções customizadas; produto escalar; LiteX.

ABSTRACT

This work presents the development and evaluation of a functional extension aimed at accelerating the scalar product in a RISC-V processor implemented on an Field-Programmable Gate Array (FPGA). Motivated by the need to increase computational performance, three microarchitectures were designed: Sequential Finite State Machine, Finite State Machine with pipeline, and pipeline, with the objective of analyzing the impact of different parallelism strategies on reducing latency and increasing throughput. The solution was integrated into a SoC built with the LiteX framework, using control and status registers as an interface between the PicoRV32 processor and the dedicated accelerator. The methodology covered the entire digital hardware development flow, including RTL description in SystemVerilog, simulation, synthesis, place-and-route, and validation on real hardware in the ECP5 LFE5U-45 FPGA. Experimental results showed substantial performance improvements, particularly for the registered pipeline architecture, which achieved a latency of 4 cycles and a throughput of 120 million operations per second, significantly surpassing the basic FSM version. Hardware-software integration proved efficient, and cross-validation confirmed the functional correctness of the three microarchitectures. This work presents a complete, reproducible workflow, fully supported by open-source tools, for developing functional extensions to the RISC-V architecture, demonstrating the potential of this approach to accelerate vector operations in embedded systems.

Keywords: RISC-V; FPGA; hardware acceleration; custom instructions; scalar product; LiteX.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

RISC-V	*Reduced Instruction Set Computer – Version Five* (Arquitetura aberta baseada em conjunto reduzido de instruções)
FPGA	*Field-Programmable Gate Array* (Arranjo de portas programável em campo)
ASICs	*Application-Specific Integrated Circuit* (Circuito integrado específico para aplicação)
GPUs	*Graphics Processing Unit* (Unidade de processamento gráfico)
NRE	*Non-Recurring Engineering* (Custo de engenharia não recorrente — custo único para desenvolvimento)
LUT's	*Look-Up Table* (Tabela de consulta usada para implementar lógica combinacional em FPGAs)
ISA	*Instruction Set Architecture* (Arquitetura de conjunto de instruções)
FSM	*Finite State Machine* (Máquina de estados finitos)
CSR	*Control and Status Register* (Registradores de Controle e Status)
SoC	*System-on-Chip* (Sistema em Chip)
FFs	*Flip-Flops* (Elementos de memória)
DSP	*Digital Signal Processor* (Contexto da FPGA: bloco aritmético dedicado (multiplicadores/ACC))
CLB	*Configurable Logic Block* (Bloco lógico configurável da FPGA)
OPC	*Operations Per Cycle* (Operações por ciclo)
MOPS	*Million Operations Per Second* (Milhões de operações por segundo)
BRAM	*Block RAM* (Memória RAM em bloco dentro da FPGA)
RTL	*Register Transfer Level* (Transferência entre Registradores)
TB	*Testbench* (Ambiente de teste para simulação de hardware)
HDL	*Hardware Description Language* (Linguagem de descrição de hardware, ex: Verilog, VHDL, SystemVerilog)
DSL	*Domain-Specific Language* (Linguagem específica de domínio, projetada para expressar soluções em um domínio particular)
DMA	*Direct Memory Access* (Acesso direto à memória, mecanismo que permite a transferência de dados entre periféricos e a memória principal sem intervenção da CPU)

SUMÁRIO

1 INTRODUÇÃO

1.1 Contextualização

A crescente demanda por desempenho computacional em aplicações como simulações científicas, visão computacional, aprendizado de máquina e sistemas embarcados tem impulsionado a busca por arquiteturas capazes de oferecer alta eficiência aliada à flexibilidade de personalização; essas aplicações envolvem operações matemáticas intensivas, como aritmética vetorial, que exigem paralelismo, baixa latência e otimizações específicas de hardware para atender requisitos de tempo real e consumo energético reduzido (Sze *et al.*, 2017).

Nesse contexto, a arquitetura RISC-V tem se destacado como uma alternativa promissora por possuir um conjunto de instruções aberto, modular e amplamente adotado em pesquisas e sistemas customizados. Por ser uma ISA livre de licenças e restrições proprietárias, o RISC-V permite que desenvolvedores criem processadores personalizados, adicionando instruções específicas para suas aplicações sem dependência de arquiteturas fechadas. Essa liberdade é especialmente vantajosa em sistemas embarcados, onde instruções customizadas podem proporcionar ganhos expressivos de desempenho, eficiência energética e redução da complexidade do software (Hennessy; Patterson, 2017).

A combinação dessa capacidade de customização com o potencial dos dispositivos FPGAs *(Field-Programmable Gate Arrays)*, que oferecem reconfigurabilidade e paralelismo em nível de hardware, abre oportunidades para o desenvolvimento de soluções otimizadas voltadas à aceleração de operações fundamentais, como as vetoriais e matriciais. Tais soluções não apenas melhoram o desempenho de aplicações atuais, mas também impulsionam pesquisas em áreas emergentes, como aprendizado profundo embarcado e otimização de aceleradores para predição de modelos em FPGA (Nechi *et al.*, 2023).

Diante desse cenário, este trabalho investiga e implementa um acelerador vetorial integrado a um processador RISC-V em FPGA, com foco em acelerar operações fundamentais da álgebra linear, especialmente o produto escalar entre vetores. Foram desenvolvidas e avaliadas três arquiteturas: FSM, FSM com *pipeline* e *pipeline* registrado, possibilitando uma análise comparativa em termos de latência, *throughput*, utilização de recursos lógicos (LUTs) e eficiência global. A comunicação entre o processador e o acelerador foi realizada por meio de registradores CSR *(Control and Status Registers)*, evidenciando o potencial das instruções customizadas como estratégia de extensão funcional e aceleração de hardware em sistemas baseados em RISC-V.

1.2 Justificativas e relevância

O produto escalar constitui uma operação elementar, porém computacionalmente intensiva, presente em algoritmos críticos de diversas áreas, tais como classificadores lineares, redes neurais, filtros adaptativos, compressão de dados e sistemas de navegação inercial. Quando implementado em software sobre arquiteturas de propósito geral, seu desempenho é frequentemente limitado pela execução sequencial de múltiplas multiplicações e somas em laços apertados, especialmente em vetores de alta dimensão, configurando-se como gargalo em aplicações embarcadas e de tempo real (Li *et al.*, 2018).

Soluções baseadas em hardware dedicado, como ASICs, oferecem ganhos expressivos de desempenho, mas impõem custos elevados de NRE *(Non-Recurring Engineering)* e rigidez arquitetural, inviabilizando sua adoção em contextos de pesquisa, prototipagem ou produção em pequena escala. (Hennessy; Patterson, 2017). Por sua vez, GPUs, embora eficientes em paralelismo massivo, apresentam consumo energético elevado e latência significativa de transferência de dados, tornando-as inadequadas para sistemas de baixa potência.

Nesse cenário, a arquitetura RISC-V, com sua ISA aberta e extensível, combinada à reconfigurabilidade de FPGAs, emerge como alternativa viável e promissora. A inserção de extensões funcionais por meio de instruções customizadas ou *Custom CSRs*, permitem especializar o processador para o produto escalar sem comprometer a compatibilidade com o ecossistema de software existente (Harris; Harris, 2022).

Embora iniciativas como a extensão vetorial padrão (RISC-V) abordem aceleração de operações em vetores, poucos trabalhos acadêmicos reportam implementações práticas, validadas em hardware real (FPGA), de aceleradores dedicados ao produto escalar com avaliação quantitativa de latência, *throughput*, consumo de recursos (LUTs e FFs) e ganho de desempenho em relação à execução puramente em software, especialmente quando integrados via mecanismos como *Custom CSR*, que conciliam flexibilidade e baixa complexidade de integração.

Assim, justifica-se o presente trabalho pela proposição, implementação e avaliação de uma extensão funcional concreta para aceleração do produto escalar em um *softcore* RISC-V integrado a FPGA, contribuindo com:

- Contribuição prática: Desenvolvimento de uma solução replicável, de baixo custo e baseada em tecnologias abertas para aceleração de operações críticas em sistemas embarcados;

- Contribuição experimental: Análise quantitativa rigorosa de desempenho, área e

eficiência em comparação com a execução puramente em software;

- Contribuição metodológica: Estruturação de um estudo de caso aplicável como referência para o desenvolvimento de futuras extensões funcionais em RISC-V, reforçando o potencial do co-projeto processador-acelerador em arquiteturas abertas.

1.3 Objetivos

1.3.1 Objetivo geral

Projetar, implementar e avaliar uma extensão funcional de aceleração vetorial para operações de produto escalar em um *SoC* RISC-V embarcado em FPGA, empregando registros CSR como interface de comunicação entre hardware e software, a fim de investigar ganhos de desempenho e eficiência arquitetural.

1.3.2 Objetivos específicos

- Realizar revisão bibliográfica sobre RISC-V, extensões funcionais e aceleração de álgebra linear em FPGA;
- Projetar e implementar um acelerador de produto escalar em SystemVerilog;
- Integrar o acelerador a um *softcore* RISC-V (picorv32) via *framework* LiteX;
- Desenvolver firmware em C com validação cruzada (hardware x software);
- Implementar três microarquiteturas alternativas para o acelerador (FSM, FSM+*pipeline*, *pipeline* registrado);
- Avaliar desempenho, uso de recursos e eficiência entre as versões;
- Documentar todo o fluxo com ferramentas *open-source* (Yosys, nextpnr, Trellis).

1.4 Organização e estrutura

Este trabalho está estruturado em seis capítulos, organizados de forma a apresentar o desenvolvimento do projeto de maneira lógica e coerente.

O Capítulo 1 introduz o tema, contextualizando a justificativa, a relevância e os objetivos do estudo, além de apresentar a estrutura geral do trabalho.

O Capítulo 2 aborda o referencial teórico, reunindo os conceitos fundamentais sobre arquiteturas RISC-V, extensões customizadas, FPGA e o *framework* LiteX, bem como uma revisão de trabalhos relacionados.

O Capítulo 3 descreve a metodologia empregada, detalhando a arquitetura geral do sistema, o fluxo de desenvolvimento adotado e as ferramentas utilizadas durante o projeto.

O Capítulo 4 apresenta a implementação prática da aplicação, incluindo o desenvolvimento dos módulos em SystemVerilog, a integração com o firmware via CSR e os procedimentos de simulação e síntese em FPGA.

O Capítulo 5 discute os resultados obtidos e sua análise, contemplando métricas de latência, *throughput*, utilização de recursos lógicos e comparativos de desempenho entre as arquiteturas implementadas.

Por fim, o Capítulo 6 reúne as conclusões do trabalho, destacando as contribuições alcançadas e propondo possíveis direções para aprimoramentos e estudos futuros.

2 REFERENCIAL TEÓRICO

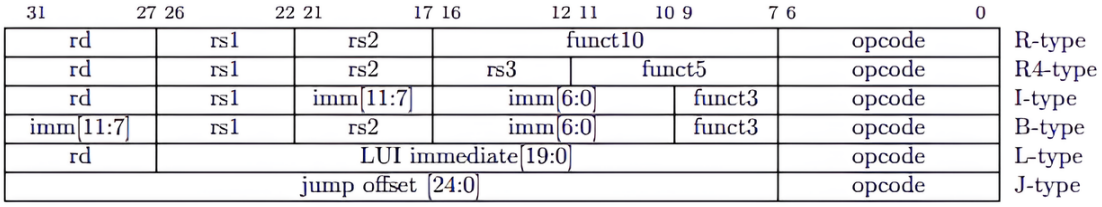
2.1 Fundamentação teórica

2.1.1 Arquitetura RISC-V

A arquitetura RISC-V é um conjunto de instruções (ISA), projetada com o objetivo principal de ser aberta e disponibilizada de forma gratuita, baseada em princípios de simplicidade, modularidade e escalabilidade, desenvolvida inicialmente na Universidade da Califórnia, Berkeley, em 2010, ela tem como objetivo oferecer uma alternativa padronizada e livre de *royalties* para o desenvolvimento de processadores em contextos acadêmicos e industriais. Diferentemente de arquiteturas proprietárias, como x86 ou ARM, o RISC-V permite que pesquisadores e desenvolvedores modifiquem e implementem seus próprios núcleos com liberdade total, o que torna ideal para aplicações embarcadas, computação de alto desempenho e pesquisas em microarquitetura (Waterman *et al.*, 2011).

Na Figura 1, são apresentados os principais formatos de instrução utilizados na abordagem do RISC-V descrita no relatório (Waterman *et al.*, 2011). Cada instrução possui 32 bits e é organizada de acordo com o tipo de operação a ser executada. O formato *R-type* é empregado em instruções aritméticas e lógicas entre registradores, contendo campos para dois operandos de origem (rs1, rs2), um registrador destino (rd) e bits de função que definem a operação. O *R4-type*, estendido, inclui um terceiro registrador fonte (rs3) para operações especializadas. O formato *I-type* incorpora um valor imediato de 12 bits, sendo usado em instruções de carga, operações aritméticas imediatas e acessos a registradores. Já o *B-type* organiza o imediato de forma particionada, sendo utilizado em desvios condicionais. O formato *L-type (LUI)* carrega diretamente um valor imediato de 20 bits para os bits mais significativos de um registrador, enquanto o *J-type* define o deslocamento de salto usado em instruções de controle de fluxo. Esses formatos evidenciam a abordagem modular e consistente do RISC-V, permitindo simplicidade na decodificação e flexibilidade na extensão do conjunto de instruções.

Figura 1 – Formatos de instruções básicas no RISC-V



Fonte: Waterman *et al.* (2011).

2.1.2 Extensões customizadas ISA

Instruções customizadas são novas operações adicionadas ao conjunto de instruções básico de uma arquitetura, com o objetivo de acelerar tarefas específicas, elas são implementadas diretamente em hardware, as arquiteturas de conjunto de instruções (ISAs) geralmente fornecem apenas um conjunto básico de operações, suficiente para atender a aplicações de propósito geral, no entanto, áreas modernas como inteligência artificial, processamento digital de sinais (DSP), criptografia e computação vetorial exigem operações mais complexas e de alto desempenho, para atender a essas demandas, tornou-se comum o uso de extensões customizadas, que permitem incorporar novas instruções diretamente no hardware, resultando em menor latência e consumo energético (Cui; Li; Wei, 2023).

No contexto do RISC-V, essa flexibilidade é um dos seus principais diferenciais, como destacado na pesquisa de (Cui; Li; Wei, 2023), a arquitetura foi projetada com uma abordagem modular: um núcleo obrigatório mínimo, complementado por diversas extensões padrão opcionais, espaço reservado para instruções completamente customizadas, sem quebrar a compatibilidade com o ecossistema. Esse design aberto tem impulsionado uma vasta quantidade de pesquisas acadêmicas e industriais que propõem instruções especializadas para domínios específicos, como produto escalar *(dot-product)*, convoluções, multiplicações matriciais, transformadas numéricas e aceleração de algoritmos criptográficos.

Esses trabalhos demonstram que extensões customizadas são uma estratégia poderosa para o desenvolvimento de processadores orientados a domínios específicos *(domain-specific processors)*. Nesse sentido, a proposta deste trabalho, a criação de uma instrução dedicada ao produto escalar, está plenamente alinhada com as tendências atuais de otimização arquitetural e segue o mesmo racional adotado por diversas extensões bem-sucedidas analisadas na literatura recente (Cui; Li; Wei, 2023).

2.1.3 FPGA

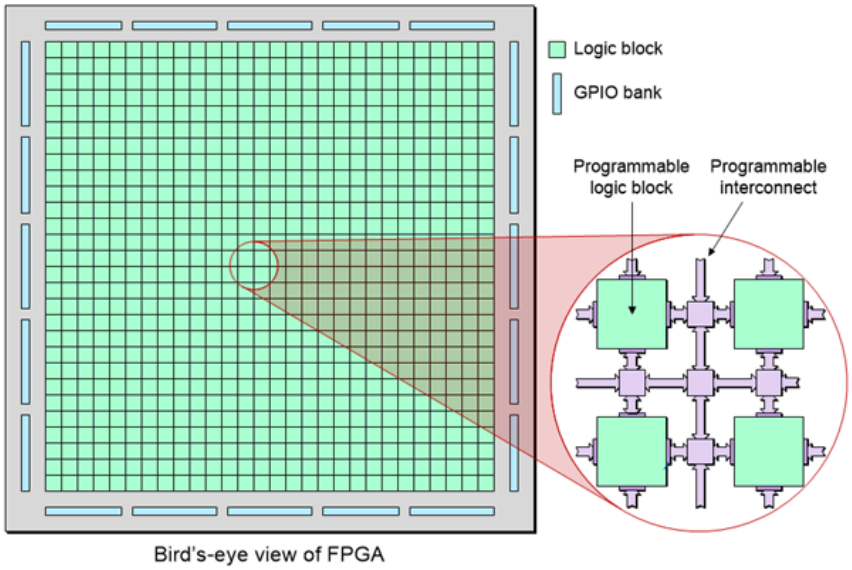
(Field-Programmable Gate Arrays) são circuitos integrados programáveis que permitem implementar lógica digital mesmo após a fabricação, perfeito para experimentar, prototipar e testar novas ideias de hardware sem precisar criar um chip do zero, como apontam (Harris; Harris, 2022), com as FPGAs é possível transformar código escrito em Verilog ou VHDL diretamente em circuitos funcionais, acelerando muito o ciclo de desenvolvimento: você escreve, simula, implementa e testa blocos lógicos ou até arquiteturas completas em poucas horas.

Sua estrutura interna é composta por blocos lógicos programáveis e uma malha de interconexões configuráveis, o que possibilita executar múltiplas operações em paralelo, essa capacidade de processamento paralelo torna as FPGAs especialmente adequadas para aplicações que demandam alto desempenho e baixa latência, como computação de borda, inteligência artificial e sistemas embarcados avançados (Semiconductor, 2024).

Na Figura 2, é apresentada uma visão geral de uma FPGA, nela, é possível observar uma grande matriz composta por diversos blocos lógicos *(logic blocks)*, que são os elementos responsáveis por implementar as funções digitais dentro do dispositivo, esses blocos estão interligados por uma malha de interconexões programáveis *(programmable interconnect)*, permitindo que qualquer bloco possa ser conectado a outro conforme a necessidade do projeto.

Ao redor da matriz, também estão localizados os bancos de GPIO, que fazem a interface entre a FPGA e os sinais externos, na ampliação à direita da figura destaca um conjunto de blocos lógicos e suas interconexões, mostrando como cada bloco pode se comunicar com os blocos vizinhos por meio dessa rede de roteamento configurável.

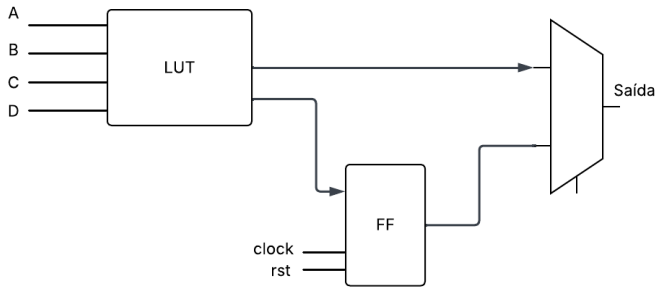
Figura 2 – Representação de uma FPGA



Fonte: Semiconductor (2024).

Na Figura 3, é ilustrada a estrutura interna de um bloco lógico configurável (CLB) de uma FPGA. A figura mostra uma *Look-Up Table (LUT)* de quatro entradas (A, B, C e D), responsável por implementar a lógica combinacional desejada. O resultado produzido pela *LUT* pode seguir diretamente para a saída, ou pode ser armazenado previamente em um flip-flop, que é acionado pelo sinal de *clock* e possui entrada de reset. Por fim, um multiplexador seleciona se a saída do bloco será o valor combinacional da *LUT* ou o valor registrado no flip-flop, permitindo que o mesmo bloco opere tanto como lógica combinacional quanto como lógica sequencial.

Figura 3 – *Configurable Logic Blocks*



Fonte: Autoria própria (2025).

2.1.4 Álgebra linear e operações vetoriais

A álgebra linear é um ramo fundamental da matemática que trata de vetores, matrizes e operações entre eles, sendo amplamente utilizada em diversas áreas da computação. No contexto de arquitetura de computadores, suas aplicações estão presentes em tarefas que envolvem o processamento paralelo de dados, como aprendizado de máquina, visão computacional, gráficos digitais e simulações científicas. As operações vetoriais e matriciais permitem representar e manipular grandes volumes de dados de forma estruturada e eficiente, servindo como base para algoritmos de alto desempenho. Entre essas operações, destaca-se o produto escalar, uma operação elementar que consiste na soma dos produtos dos elementos correspondentes de dois vetores. Essa operação é amplamente utilizada em modelos matemáticos e computacionais, como em classificadores lineares, redes neurais e transformadas de sinais, desempenhando papel essencial em diversos sistemas de processamento digital e arquiteturas vetoriais (Hennessy; Patterson, 2017; Anton; Rorres, 2012).

2.2 Trabalhos relacionados

Diversos trabalhos recentes abordam o desenvolvimento e otimização de arquiteturas voltadas ao aumento de desempenho em aplicações computacionais intensivas, seja pela criação de novos núcleos de processadores, seja pela implementação de aceleradores específicos em hardware reconfigurável.

O processo de desenvolvimento de uma CPU RISC-V

Em (Silva, 2021), é apresentado o processo completo de desenvolvimento de uma CPU baseada na arquitetura RISC-V, incluindo as etapas de descrição em Verilog, simulação, verificação e síntese em FPGA. O foco do trabalho recai sobre a metodologia de verificação do processador, demonstrando como o uso de linguagens de descrição de hardware e testes automatizados pode garantir o comportamento correto do circuito. A relevância desse estudo para o presente trabalho está na base metodológica de projeto e verificação de módulos RISC-V, servindo como referência para o desenvolvimento da unidade de execução customizada implementada neste TCC.

FPGA-based deep learning onference accelerators: where are we standing?

Já em (Nechi *et al.*, 2023), os autores realizam uma análise abrangente sobre o estado da arte em aceleradores baseados em FPGA aplicados à inferência de redes neurais profundas. O artigo compila mais de 120 implementações acadêmicas, discutindo critérios de

desempenho, consumo de energia e técnicas de paralelismo em arquiteturas reconfiguráveis. A principal contribuição está na sistematização de estratégias de otimização e avaliação de desempenho em sistemas FPGA, evidenciando o potencial dessas plataformas para acelerar operações matemáticas intensivas, como multiplicações e produtos escalares, diretamente relacionados ao escopo deste trabalho.

A GPU-Outperforming FPGA accelerator architecture for binary convolutional neural networks

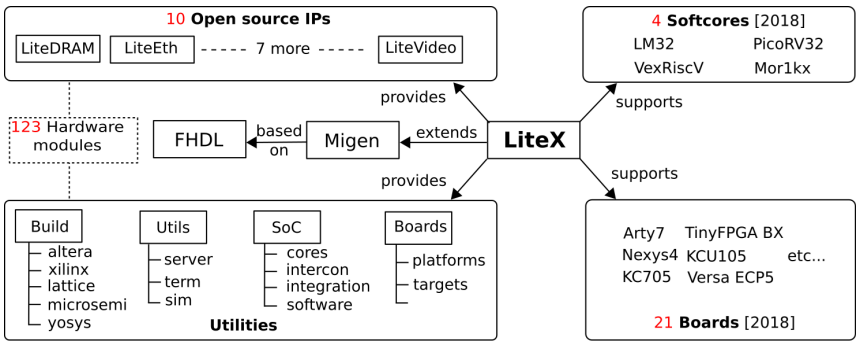
Em (Li *et al.*, 2018), é proposto uma arquitetura de acelerador em FPGA para redes neurais convolucionais binárias *(BCNNs)*, capaz de superar o desempenho de GPUs em determinadas condições de execução. O projeto explora fortemente o paralelismo espacial e o *pipeline* profundo, atingindo ganhos expressivos de *throughput* e eficiência energética. Essa abordagem demonstra como a adaptação da arquitetura para operações de baixa precisão e a exploração do paralelismo intrínseco da FPGA podem resultar em melhorias significativas de desempenho, princípios também aplicados na instrução customizada de produto escalar proposta neste trabalho.

LiteX: an open-source SoC builder and library based on migen python DSL

Em (Kermarrec *et al.*, 2020), é apresentado o *framework* LiteX, uma biblioteca e construtor de *SoCs open-source* desenvolvida pela Enjoy-Digital. O LiteX é baseado em uma *Domain-Specific Language (DSL)* interna escrita em Python (Migen), permitindo a criação de sistemas em FPGA com suporte a diversos núcleos de CPU, incluindo RISC-V. O artigo demonstra dois estudos de caso em FPGA, destacando a flexibilidade e a independência de IPs proprietários.

A Figura 4, apresenta uma visão geral da arquitetura do LiteX, mostrando como o *framework* se organiza em torno do Migen e da FHDL para gerar lógica RTL. No topo, são listados os principais IPs *open-source* disponíveis, enquanto à direita aparecem os *softcores* suportados. Na base, estão os utilitários responsáveis pela construção do *SoC*, integração de periféricos e suporte às diversas plataformas FPGA. O diagrama evidencia também a quantidade de módulos de hardware e placas compatíveis, ilustrando o caráter modular e expansível do LiteX.

Figura 4 – Uma visão geral do *framework* LiteX



Fonte: Kermarrec *et al.* (2020).

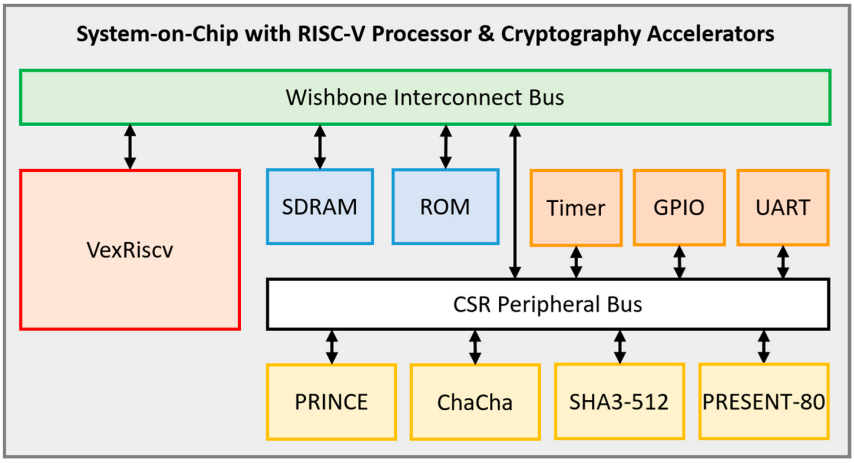
A relevância para este trabalho está na demonstração de que é possível projetar e integrar processadores RISC-V em FPGA com alto grau de customização, ainda que sem foco em instruções específicas para operações de álgebra linear.

Design of an SoC based on 32-Bit RISC-V processor with low-latency lightweight cryptographic cores in FPGA

Em (Ma *et al.*, 2023), os autores descrevem o desenvolvimento de um *SoC* baseado em processador RISC-V integrado a núcleos criptográficos leves e de baixa latência, implementado em FPGA. O trabalho enfatiza o potencial da arquitetura RISC-V para integração de aceleradores dedicados, validando experimentalmente ganhos de desempenho e eficiência energética.

Na Figura 5, é apresentado o diagrama geral da arquitetura do *SoC* desenvolvido, composto por um núcleo RISC-V (VexRiscv) 32 bits e um conjunto de aceleradores criptográficos *lightweight* (PRINCE, PRESENT-80, ChaCha e SHA3-512). Os módulos são integrados por meio do barramento de registradores CSR, que fornece a interface de controle entre o processador e os aceleradores, a figura ilustra a modularidade da arquitetura RISC-V e sua aptidão para a incorporação de unidades funcionais especializadas em FPGA.

Figura 5 – Arquitetura *SoC* com acelerador de funções criptográficas



Fonte: Ma *et al.* (2023).

Essa abordagem reforça a ideia de que extensões arquiteturais específicas podem aumentar o desempenho em domínios de aplicação distintos, conceito explorado neste TCC para o desenvolvimento do produto escalar

Improving fault tolerance for FPGA SoCs through post-radiation design analysis

Por fim, (Wilson *et al.*, 2024) apresenta uma metodologia de tolerância a falhas em sistemas baseados em FPGA, utilizando um *SoC* RISC-V implementado com o *framework* LiteX como plataforma de referência, o estudo demonstra a flexibilidade do LiteX para o desenvolvimento de sistemas complexos e a facilidade de integração com módulos externos, embora o foco seja confiabilidade frente a eventos de radiação, em vez de desempenho.

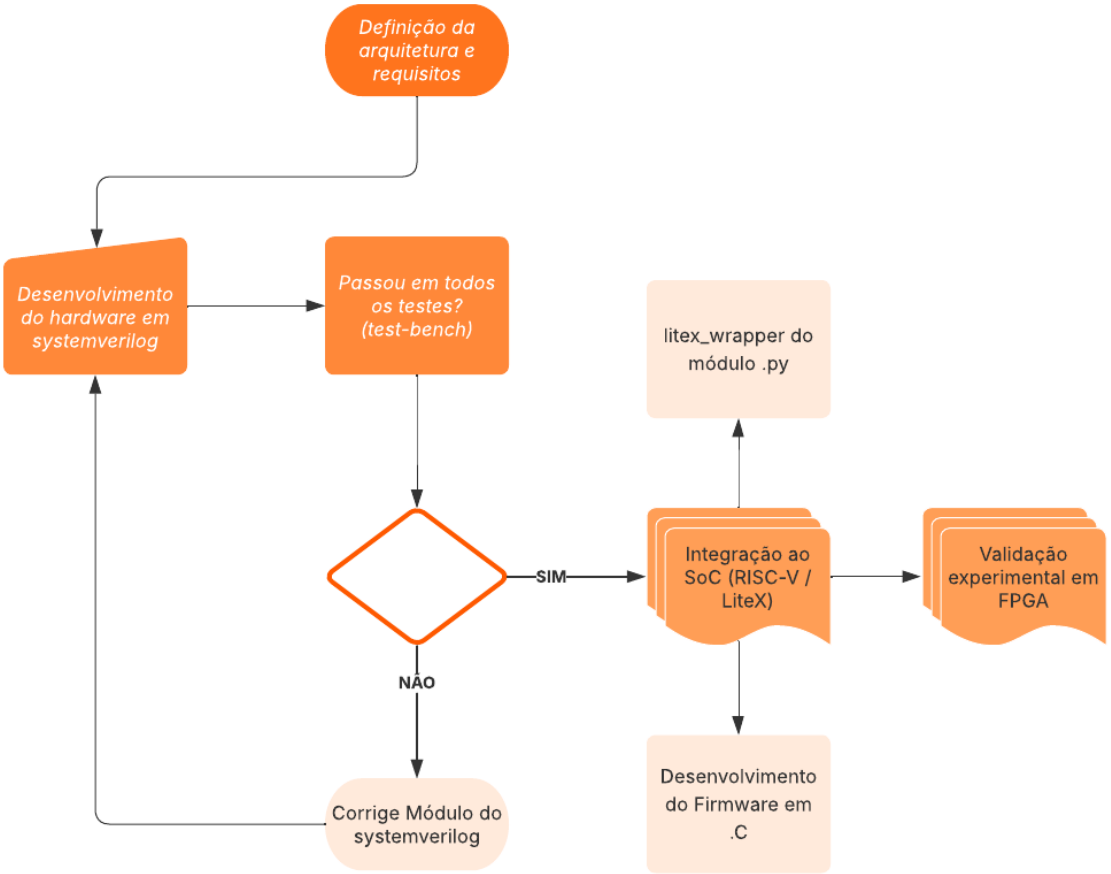
Assim, o presente trabalho se diferencia ao utilizar a mesma base tecnológica (RISC-V em FPGA) para explorar a aceleração de operações matemáticas através de instruções customizadas.

Dessa forma, observa-se que as pesquisas atuais convergem para o uso de FPGAs como plataformas flexíveis e eficientes para aceleração de operações computacionais, incluindo aplicações em aprendizado de máquina e processamento vetorial. No entanto, poucos trabalhos exploram a extensão direta do conjunto de instruções RISC-V com foco em operações aritméticas vetoriais específicas, como o produto escalar. Este trabalho busca contribuir nesse contexto, implementando e avaliando uma instrução customizada em FPGA que visa demonstrar o potencial de aceleração obtido com esse tipo de extensão arquitetural.

3 METODOLOGIA

A metodologia adotada neste trabalho foi organizada em etapas sequenciais e iterativas, abrangendo desde o projeto da unidade aceleradora até sua integração ao processador RISC-V e posterior avaliação em FPGA. A Figura 6 apresenta uma visão geral do fluxo metodológico seguido.

Figura 6 – Fluxograma do projeto



Fonte: Autoria própria (2025).

3.1 Desenvolvimento

Para colocar em prática a metodologia ilustrada na Figura 6, o trabalho de desenvolvimento foi dividido em cinco etapas principais, detalhadas nas subseções a seguir. Inicialmente, realizou-se a especificação funcional do acelerador, definindo-se precisamente seu comportamento. Na sequência, efetuou-se a escolha do *softcore*. Posteriormente,

procedeu-se à implementação em hardware, com a exploração de três microarquiteturas distintas, todas desenvolvidas em SystemVerilog. A etapa seguinte consistiu na integração ao *SoC* LiteX, na qual o módulo foi conectado ao processador RISC-V por meio da interface CSR (*Control and Status Register*). Por fim, o desenvolvimento do firmware em C permitiu o controle do acelerador, a execução de testes de validação e a comparação de seu desempenho com a versão puramente em software. O funcionamento do sistema completo foi verificado experimentalmente em FPGA. O código-fonte completo do projeto está disponível publicamente no repositório do GitHub.

3.1.1 Especificação da extensão funcional

A primeira etapa do desenvolvimento consistiu na definição precisa dos requisitos funcionais do acelerador de produto escalar, essa especificação descreve o comportamento esperado do módulo, de forma independente de sua implementação interna, garantindo uniformidade funcional entre as três microarquiteturas desenvolvidas.

Os requisitos funcionais e interfaces são os seguintes:

- Operação: Cálculo do produto escalar entre dois vetores unidimensionais de oito elementos, conforme:

$$R = \sum_{i=0}^{7} A[i] \cdot B[i] \tag{1}$$

  em que A e B são vetores de inteiros com sinal de 32 bits, representados em complemento de dois, e o resultado R é um inteiro com sinal de 64 bits, também em complemento de dois. Toda a operação é executada em hardware

- Interface de comunicação com a CPU: Comunicação realizada exclusivamente por meio de registradores CSR (*Control and Status Registers*) mapeados em memória, conforme a infraestrutura do LiteX.

- Protocolo de controle: A operação é iniciada por um pulso no sinal (iniciar) e finalizada quando o sinal (concluído) é ativado, indicando que o resultado já está disponível para leitura.

Essas especificações asseguram a integração adequada entre o acelerador, a CPU PicoRV32 e a infraestrutura de barramento do LiteX, permitindo que o módulo opere como um periférico mapeado em memória. Com isso, o software pode acioná-lo, acompanhar seu estado e ler os resultados de forma direta e transparente, sem a necessidade de protocolos adicionais.

3.1.2 Escolha do softcore RISC-V (PicoRV32)

O processador PicoRV32 foi selecionado como núcleo RISC-V deste trabalho por apresentar uma microarquitetura simples, modular e amplamente consolidada em ambientes acadêmicos e de pesquisa, características que o tornam especialmente adequado para estudos envolvendo extensões funcionais e aceleração por hardware.

Por se tratar de um *softcore open-source*, com código enxuto e bem documentado, o PicoRV32 facilita a compreensão do fluxo interno de execução e a integração de módulos customizados, permitindo a inserção de aceleradores especializados sem a necessidade de modificações complexas na estrutura do processador. Além disso, sua compatibilidade nativa com o *framework* LiteX e o uso de registradores CSR mapeados em memória simplificam a comunicação entre software e hardware, atendendo diretamente aos objetivos deste trabalho.

A escolha também se justifica pelo baixo consumo de recursos lógicos, o que viabiliza sua implementação em FPGAs de médio porte, como a ECP5 LFE5U-45, sem comprometer a flexibilidade necessária para a experimentação e avaliação de diferentes microarquiteturas de aceleração.

3.1.3 Projeto hardware em systemverilog

O acelerador de produto escalar foi desenvolvido integralmente em SystemVerilog, contemplando três microarquiteturas distintas, o objetivo dessa variação estrutural foi avaliar os impactos de diferentes graus de paralelismo e profundidade de *pipeline* sobre a latência, o *throughput*, a utilização de recursos lógicos e a complexidade de controle. Cada microarquitetura implementa o mesmo comportamento funcional, diferenciando-se apenas na organização do *datapath* e da lógica de controle.

As três versões desenvolvidas são apresentadas a seguir:

1. FSM *(Finite state machine)*: Arquitetura totalmente sequencial, baseada em uma máquina de estados finitos que percorre iterativamente as etapas de multiplicação e acumulação, em cada ciclo de *clock* apenas uma operação parcial do produto escalar é executada, resultando em baixa paralelização e na menor utilização de área entre as três versões. O *datapath* é compacto, composto por um único multiplicador e um acumulador reutilizado ao longo das iterações, enquanto a lógica de controle é responsável pelo avanço entre estados, pelo controle dos multiplexadores e pela sinalização de conclusão. Trata-se da implementação mais simples, com temporização fácil de fechar, porém apresenta a maior latência.

2. FSM com *pipeline*: Arquitetura híbrida que preserva a lógica de controle sequencial da FSM, mas introduz estágios de *pipeline* nas operações mais críticas, especialmente multiplicação e soma acumulada, essa estrutura reduz a latência por operação ao permitir a sobreposição parcial das etapas internas, ao mesmo tempo em que mantém um controle relativamente simples baseado em uma FSM compacta. O *datapath* possui maior paralelismo que a versão sequencial pura, com registradores adicionais para os estágios intermediários, resultando em complexidade moderada e uso de área maior, porém ainda inferior ao *pipeline* registrado.

3. *Pipeline* registrado: Arquitetura integralmente orientada a *pipeline*, segmentada em múltiplos estágios sequenciais com lógica combinacional balanceada entre eles. Essa organização reduz a latência total ao distribuir o cálculo em etapas sucessivas, permitindo que o resultado seja produzido após apenas alguns ciclos de *clock*. Diferentemente de *pipelines* contínuos, esta implementação processa um único conjunto de dados por ativação, uma vez que os operandos são amostrados apenas no pulso de início. A lógica de controle permanece simples, baseada na propagação de sinais de validade entre estágios e na geração do pulso de conclusão, enquanto o *datapath* fragmentado exige maior quantidade de registradores e maior rigor no fechamento de temporização. Trata-se da versão mais rápida entre as três, embora também seja a mais complexa em termos estruturais.

O processo de desenvolvimento seguiu o fluxo tradicional de projeto digital, iniciando pela descrição RTL em SystemVerilog e pela elaboração do respectivo *testbench*, utilizado nas simulações funcionais realizadas com o Icarus Verilog para exercitar cada microarquitetura e verificar o comportamento esperado dos sinais internos e das interfaces. A análise das formas de onda no GTKWave complementou essa etapa, permitindo ciclos iterativos de refinamento estrutural, correções de lógica e validação temporal.

Esse fluxo permitiu que cada microarquitetura evoluísse de uma descrição funcional inicial para uma implementação RTL estável e verificável. Ao término das iterações, todas as versões apresentaram comportamento consistente e temporização adequada, concluindo de forma sólida a etapa de projeto em hardware.

3.1.4 Integração ao *SoC* LiteX

A integração do acelerador de produto escalar ao *System-on-Chip* foi realizada utilizando o *framework* LiteX, que oferece uma infraestrutura completa e altamente modular para construção de *SoCs* baseados em diferentes *softcores*, incluindo o RISC-V. A estrutura

interna do LiteX utiliza a *DSL* Migen, que permite descrever hardware de forma declarativa em Python e automatiza a geração de interconexões, registradores e lógica de controle. No presente trabalho, essa *DSL* foi empregada para encapsular o acelerador RTL em um periférico compatível com o barramento Wishbone/CSR do LiteX.

O processo de integração foi organizado nas seguintes etapas:

- Criação do módulo customizado: Desenvolvimento de um *wrapper* em Python utilizando Migen, responsável por adaptar a interface genérica do módulo RTL ao padrão de registradores CSR do LiteX. Esse *wrapper* mapeia sinais internos (operandos, início, conclusão e resultados) para registradores acessíveis pela CPU.

- Mapeamento de endereços: Alocação de uma faixa contínua de registradores CSR, abrangendo os sinais de controle (iniciar), os operandos (a0–a7 e b0–b7), os registradores de monitoramento de desempenho (latência, número de operações e validade da medição) e os registradores de saída (resultado e concluído), assegurando compatibilidade direta com o barramento.

- Conexão ao barramento: Instanciação do *wrapper* como periférico CSR do *SoC*, permitindo que a CPU PicoRV32 acesse o acelerador por meio de instruções padrão de carga e armazenamento. Nesse processo, o LiteX gera automaticamente toda a lógica de decodificação, multiplexação e interconexão no barramento Wishbone, de modo que nenhum código adicional de barramento precisa ser escrito manualmente.

- Definição e construção do *SoC*: Implementação de um script de configuração em Python utilizando o *framework* LiteX, responsável por definir a arquitetura completa do *System on Chip (SoC)* na FPGA. Esse arquivo realiza a instanciação do módulo customizado como periférico CSR, promove sua integração ao barramento Wishbone e configura os domínios de *clock*, controladores de memória, interfaces de comunicação e demais periféricos do sistema, estabelecendo a organização lógica e funcional do *SoC*.

- Síntese e implementação: A etapa de implementação física do *SoC* foi realizada por meio da *toolchain OSS CAD Suite*, utilizando o Yosys para a síntese lógica do projeto, o nextpnr-ecp5 para o posicionamento e roteamento dos recursos na arquitetura ECP5 e o Project Trellis para a geração do *bitstream* final destinado aos dispositivos ECP5, foi utilizada a FPGA Lattice ECP5 LFE5U-45, presente na placa Colorlight i9.

Após a integração, o acelerador passou a operar como um periférico mapeado em memória, acessível por meio de instruções de carga e armazenamento do conjunto RISC-V.

A CPU PicoRV32 controla a execução escrevendo nos registradores CSR para configurar os operandos e acionar o início do cálculo, enquanto acompanha o sinal concluído e obtém o resultado final por meio de leituras CSR, estabelecendo uma comunicação eficiente entre software e hardware acelerador.

3.1.5 Desenvolvimento do firmware em C

O firmware responsável pelo gerenciamento do acelerador de produto escalar foi desenvolvido em linguagem C e executado diretamente pelo processador PicoRV32 integrado ao *SoC*. Seu objetivo principal é coordenar a transferência de dados entre a CPU e o módulo acelerador, bem como validar o funcionamento correto do hardware.

Inicialmente, os vetores de entrada são carregados na memória principal do sistema, permitindo que a CPU acesse seus elementos de forma eficiente. Em seguida, o firmware realiza a escrita dos operandos nos registradores CSR disponibilizados pelo *wrapper* do acelerador, sendo que cada registrador corresponde a um elemento dos vetores A e B utilizados no cálculo. Essa abordagem foi adotada por apresentar custo de comunicação determinístico e baixa complexidade de hardware, evitando a necessidade de acesso direto do acelerador à memória principal, o que exigiria mecanismos adicionais de arbitragem ou DMA.

O carregamento dos operandos envolve um número fixo de acessos aos registradores, totalizando 16 escritas em CSRs para os vetores de entrada, além de acessos adicionais para controle, leitura de status e obtenção do resultado final. Dessa forma, o *overhead* de comunicação é constante, uma vez que o número de acessos à memória e aos registradores CSR é determinado exclusivamente pelo tamanho fixo dos vetores de entrada e pelo protocolo de comunicação definido entre a CPU e o acelerador, não variando em função dos dados processados nem da microarquitetura interna.

Consequentemente, quaisquer diferenças de desempenho observadas entre as microarquiteturas avaliadas decorrem exclusivamente da execução em hardware, e não do custo de comunicação entre a CPU e o acelerador.

Após a configuração dos operandos, a CPU aciona a operação de produto escalar por meio da escrita no registrador de controle, que envia o sinal de início ao módulo RTL. Durante o processamento, o firmware realiza a leitura do registrador de status do acelerador, monitorando o sinal de conclusão para identificar o término da operação.

Ao final do cálculo, o resultado é lido a partir do registrador CSR de saída, permitindo à CPU recuperar o valor do produto escalar computado em hardware.

Com o objetivo de garantir a corretude funcional e facilitar a análise experimental,

o firmware também executa uma implementação equivalente do cálculo em software. O resultado produzido pelo acelerador é, então, comparado com o valor obtido pela implementação de referência, assegurando a validação funcional do módulo e possibilitando a detecção de eventuais discrepâncias durante o processo de desenvolvimento.

A compilação do firmware foi realizada utilizando a *toolchain* RISC-V integrada ao fluxo de geração do LiteX. Após a compilação, o binário resultante é automaticamente incorporado ao *SoC* como parte da memória ROM, garantindo que o firmware seja carregado e executado imediatamente após a configuração da FPGA.

3.2 Material e método

3.2.1 Ferramentas utilizadas

O desenvolvimento do acelerador e do *SoC* envolveu um conjunto de ferramentas complementares que abrangem desde a descrição de hardware em alto nível até a síntese, simulação e programação da FPGA, cada ferramenta desempenha um papel específico dentro do fluxo de projeto, garantindo reprodutibilidade, automação e alinhamento com metodologias *open-source*. A Tabela 1 apresenta uma visão consolidada das principais ferramentas empregadas e de suas respectivas funções.

Tabela 1 – Ferramentas utilizadas no desenvolvimento do projeto

Ferramenta / Plataforma	**Descrição**
LiteX	*Framework* em Python para construção do *SoC*, integração entre módulos e geração do hardware mapeado em memória.
Migen	Biblioteca Python para descrição de hardware RTL de forma modular e parametrizável.
Yosys	Ferramenta de síntese lógica responsável por gerar a *netlist* compatível com FPGAs.
nextpnr	Ferramenta de *place and route* que mapeia a *netlist* na FPGA alvo.
Project Trellis	Conjunto de utilitários para gerar o *bitstream* final para dispositivos ECP5.
openFPGALoader	Ferramenta *open-source* utilizada para programar a FPGA com o *bitstream* gerado.
Icarus Verilog	Simulador RTL utilizado para validação funcional do acelerador antes da síntese.
GTKWave	Visualizador de formas de onda utilizado para analisar sinais internos durante simulação.
RISC-V GNU Toolchain	*Toolchain* utilizada para compilar o firmware que interage com o acelerador.
Placa FPGA ECP5	Plataforma de execução física do *SoC*, utilizada para testes e coleta de resultados.

Fonte: Autoria própria (2025).

A combinação dessas ferramentas permitiu a realização de todo o fluxo de desenvolvimento do projeto, desde a descrição e simulação do hardware em SystemVerilog, passando pela síntese e implementação física na FPGA, até a geração e execução do firmware em RISC-V. O uso do *OSS CAD Suite* integrando diversas ferramentas *open-source* simplificou a programação, simulação e análise do *SoC*, enquanto o LiteX e a *toolchain* RISC-V possibilitaram a integração do acelerador de produto escalar com o processador e a validação funcional completa do sistema embarcado.

3.2.2 Avaliação experimental

A avaliação experimental das três microarquiteturas desenvolvidas para o acelerador de produto escalar foi conduzida a partir de um conjunto de métricas de desempenho e custo, de forma a analisar tanto o comportamento temporal quanto o impacto em área das implementações. As métricas selecionadas refletem parâmetros clássicos de avaliação em arquiteturas digitais, permitindo caracterizar a eficiência estrutural, o paralelismo explorado e o custo computacional associado a cada alternativa de projeto.

- Latência: Tempo necessário para que o acelerador produza o primeiro resultado, expresso em ciclos de *clock*;
- *Throughput*: Taxa efetiva de processamento de operações por segundo (MOPS). O *throughput* real é obtido a partir do produto entre as operações realizadas por ciclo (OPC) e a frequência de *clock*.

$$\textit{Throughput} = \left(\frac{\text{Operações}}{\text{Ciclos}}\right) \times f_{\text{clock}} = \text{MOPS} \tag{2}$$

- Eficiência: Métrica definida como a razão entre o *throughput* real e a frequência de *clock*, resultando na quantidade média de operações concluídas por ciclo (OPC). Essa medida quantifica o aproveitamento da capacidade teórica da arquitetura, sendo aplicável tanto a implementações sequenciais quanto a versões com diferentes níveis de paralelismo ou *pipeline*.

$$\text{Eficiência} = \frac{\textit{Throughput}}{f_{\text{clock}}} = \text{OPC} \tag{3}$$

- Utilização de recursos: Quantidade de *LUTs*, flip-flops, DSPs e blocos de memória BRAM consumidos após a síntese e *place-and-route* na FPGA. Essa métrica quantifica o custo em área de cada microarquitetura e permite avaliar a escalabilidade e a viabilidade de integração em sistemas maiores.
- Tempo total de execução: Duração completa da operação considerando a latência medida e a frequência de operação. Essa métrica reflete o tempo absoluto necessário para conclusão de uma instância do cálculo de produto escalar.
- Tempo por operação: Atraso médio associado à execução de cada operação elementar dentro do acelerador, permitindo análise comparativa entre arquiteturas sequenciais e com *pipeline*.

Todas as versões foram sintetizadas, implementadas e medidas de forma independente, garantindo equivalência metodológica e permitindo comparações diretas e imparciais entre as diferentes alternativas microarquiteturais

3.2.3 Validação

A validação da solução proposta foi realizada de forma sistemática, seguindo três etapas complementares que abrangem desde a verificação lógica até a avaliação experimental em hardware real:

1. Simulação em nível RTL: Nesta etapa, os módulos desenvolvidos em SystemVerilog foram submetidos a simulações funcionais utilizando o Icarus Verilog, empregando conjuntos variados de vetores de teste. O objetivo foi verificar o comportamento lógico interno, identificar possíveis condições incorretas e garantir a conformidade com a especificação antes da síntese.

2. Co-verificação hardware–software: Após a integração do acelerador ao *SoC* LiteX, foram executados testes combinando o firmware em C compilado com *(RISC-V GNU Toolchain)* e o módulo de hardware, os resultados produzidos pelo acelerador foram comparados com aqueles obtidos por uma implementação equivalente em software no processador PicoRV32, assegurando consistência e correção funcional na interação HW/SW.

3. Testes em FPGA: Na etapa final, o sistema completo foi implementado na placa FPGA ECP5 para validação física. Foram realizadas medições reais de latência, análise de *throughput* e coleta de dados referentes à utilização de recursos (LUTs, FFs, DSPs e BRAMs), por meio do fluxo de síntese Yosys/nextpnr. Esses testes permitiram confirmar o desempenho prático do acelerador e avaliar sua eficiência sob condições reais de operação.

A execução integrada dessas etapas garantiu que o acelerador atendesse aos requisitos funcionais e de desempenho estabelecidos, validando a solução tanto em ambiente de simulação quanto em hardware real.

4 IMPLEMENTAÇÃO DA APLICAÇÃO

Este capítulo apresenta o detalhamento prático da implementação do sistema desenvolvido, diferentemente da seção de metodologia, esta parte descreve, de forma detalhada, os componentes de software e hardware construídos, a organização do repositório, os pseudocódigos mais relevantes (*wrapper* em Migen, módulo em SystemVerilog e firmware em C), além do fluxo completo de compilação, síntese, programação da FPGA e execução dos testes, o objetivo é fornecer informações suficientes para permitir a reprodução experimental e esclarecer o entendimento da integração entre hardware e software adotado no projeto.

Organização do projeto

O repositório (Xavier, 2025) foi organizado de forma padronizada para as três microarquiteturas implementadas (versão1_FSM), (versão2_FSM_*pipeline*) e (versão3_*Pipeline* _registrado). Cada versão possui exatamente a mesma organização interna, diferenciando-se apenas pelo conteúdo específico de cada microarquitetura.

A estrutura geral é ilustrada abaixo utilizando a versão1_FSM como exemplo representativo:

- Versão1_FSM/
  - rtl/: Contém o código-fonte em SystemVerilog correspondente à microarquitetura implementada.
  - tb/: Abriga o *testbench* funcional utilizado para validação comportamental e verificação da lógica;
  - firmware/: Reúne o firmware em C empregado na integração hardware–software, incluindo os programas utilizados para acionar o acelerador, coletar resultados e realizar as medições de desempenho;
  - litex/: Diretório destinado aos *wrappers* em Migen, aos arquivos de descrição do periférico e aos scripts de geração e construção do *SoC* no ambiente LiteX.

As pastas versão2_FSM_*pipeline* e versão3_*pipeline*_registrado seguem a mesma estrutura apresentada acima, diferenciando-se apenas pelo conteúdo dos diretórios *rtl/*, *tb/*, *litex/* e *firmware/*, que refletem suas respectivas microarquiteturas com FSM + *pipeline* e *pipeline* registrado, essa padronização garante consistência no fluxo de desenvolvimento,

facilita a reprodutibilidade experimental e permite comparar as versões sob condições idênticas de organização e ferramentas.

Toolchain utilizada

A implementação do projeto foi realizada utilizando integralmente ferramentas de código aberto, disponibilizadas por meio do *OSS CAD Suite*, esse conjunto integra os principais utilitários necessários ao fluxo completo de desenvolvimento de projetos HDL–FPGA, proporcionando compatibilidade e automação em todas as etapas, desde a descrição do hardware até a programação do dispositivo físico.

Para a simulação do código HDL, foram utilizados o Verilator e o Icarus Verilog, permitindo a validação funcional dos módulos antes da síntese. A análise e visualização dos sinais internos durante a simulação foram conduzidas com o GTKWave, ferramenta que oferece representação detalhada das formas de onda e facilita a identificação de comportamentos incorretos. A síntese do código RTL foi realizada com o Yosys, que gera a *netlist* compatível com as FPGAs da família Lattice ECP5, enquanto o posicionamento e roteamento físico do design foram efetuados por meio das ferramentas nextpnr e Project Trellis. A gravação do *bitstream* final na FPGA foi efetuada utilizando o openFPGALoader, garantindo a correta programação do dispositivo.

Além das ferramentas de síntese e programação, o projeto faz uso do *framework* LiteX para a construção do sistema em chip *(SoC)*, integrando o acelerador de produto escalar ao processador PicoRV32. O firmware do processador foi compilado utilizando a *riscv-gnu-toolchain*, que fornece o compilador, linker e demais utilitários necessários ao desenvolvimento *bare-metal* para a arquitetura RISC-V. Essa combinação de ferramentas assegura um fluxo de desenvolvimento completo, reprodutível e alinhado às práticas *open-source*.

Instalação do LiteX

O LiteX é distribuído como um conjunto de módulos desenvolvidos em linguagem Python e sua instalação deve ser realizada, preferencialmente, dentro do ambiente Python fornecido pela *OSS CAD Suite*, de modo a assegurar compatibilidade com as ferramentas de síntese e demais componentes do fluxo de projeto. O procedimento adotado consiste na criação de um diretório específico para o *framework*, seguida do download do script oficial de instalação e de sua execução utilizando a configuração completa. Adicionalmente, realizam-se a instalação de dependências complementares, como os pacotes *meson* e *ninja* através do *pip3* do ambiente *OSS CAD Suite*, e a configuração da *toolchain* RISC-V, garantindo também que o compilador *riscv32-unknown-elf-gcc* esteja devidamente

disponível no PATH do sistema operacional. Após a conclusão desses passos, o LiteX encontra-se totalmente integrado ao ambiente da *OSS CAD Suite*, permitindo a construção automatizada do *System on Chip (SoC)*.

Geração do *SoC*

A geração do *System on Chip (SoC)* é realizada por meio do *framework* LiteX, que permite a definição programática da arquitetura do sistema utilizando scripts em Python. Nesse estágio, são descritos e instanciados os componentes fundamentais do *SoC*, incluindo o processador, barramento Wishbone, periféricos customizados implementados como registros CSR, controladores de memória, interfaces de comunicação e domínios de *clock*. O LiteX automatiza ainda o fluxo de síntese, posicionamento e roteamento, integrando ferramentas externas disponibilizadas pela *OSS CAD Suite*, o que viabiliza a transformação do projeto lógico em uma descrição física pronta para geração do *bitstream* da FPGA.

Gravação da FPGA

A etapa de gravação da FPGA corresponde ao processo de programação do dispositivo com o arquivo de configuração gerado após a síntese e o *place & route*, esse arquivo, denominado *bitstream*, contém a representação final do hardware configurado e é transferido para a FPGA por meio da ferramenta específica incluída no *OSS CAD Suite*, esse procedimento assegura que a arquitetura definida no *SoC* seja corretamente implementada no hardware físico, permitindo sua execução em ambiente real na placa Colorlight i9.

Execução do firmware

Após a programação na FPGA, é realizada a execução do firmware responsável pelo controle e operação do sistema embarcado, o firmware é desenvolvido de modo a interagir diretamente com os periféricos mapeados no barramento Wishbone, acessando os registradores CSR e coordenando o funcionamento do acelerador implementado. Essa etapa é fundamental para validar o correto funcionamento do *SoC*, bem como para permitir a comunicação entre software e hardware dentro do ambiente embarcado.

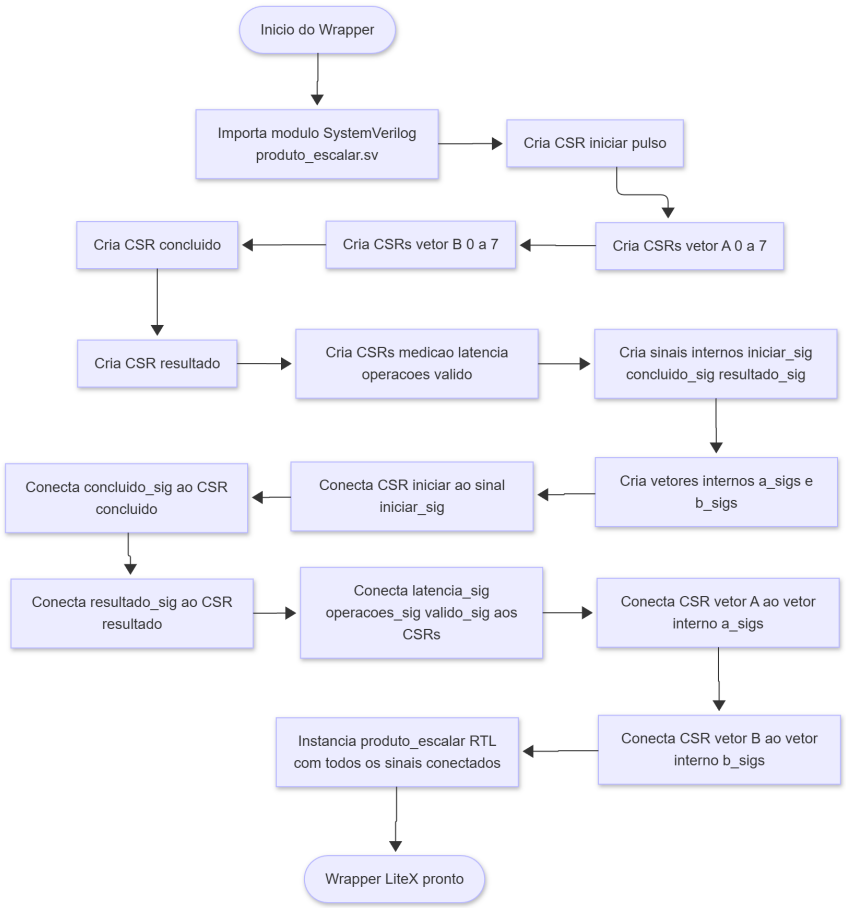
Códigos-fonte

A seguir, apresentam-se as representações visuais dos principais módulos desenvolvidos neste trabalho: *wrapper*, firmware e RTL. As figuras ilustram de forma clara e estruturada o funcionamento interno de cada componente, evidenciando o fluxo de controle, as principais etapas de processamento e as interações entre os blocos do sistema. A inclusão dessas imagens complementa a análise técnica, permitindo uma compreensão mais intuitiva da

lógica implementada e facilitando a correlação com o comportamento descrito no texto.

A Figura 7 mostra a sequência de operações do *wrapper* Python desenvolvido para gerenciar a execução do acelerador.

Figura 7 – Fluxograma do *wrapper* Python



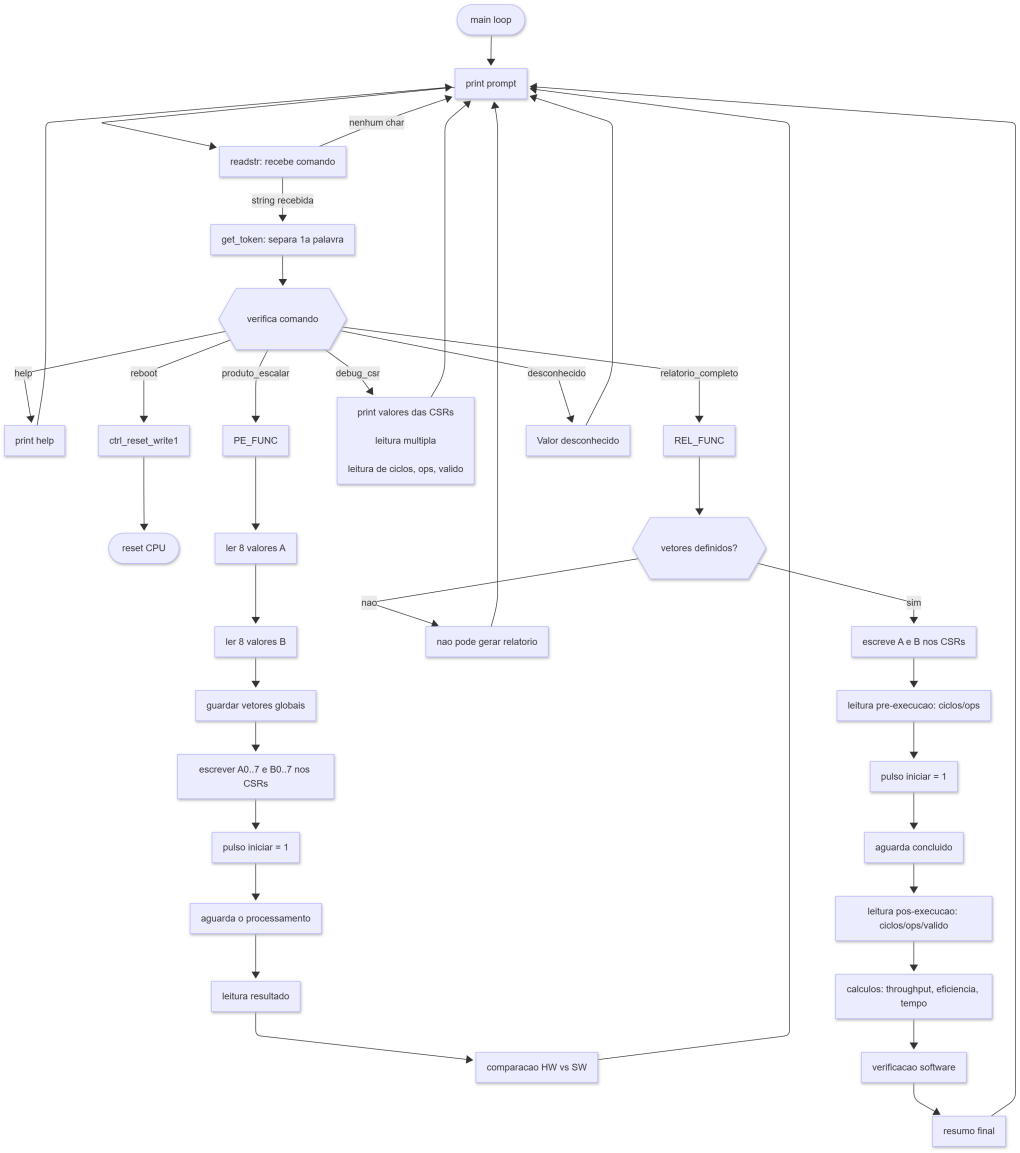
Fonte: Autoria própria (2025).

Como podemos ver a Figura 7 detalha as etapas principais do *wrapper*, incluindo a configuração dos CSRs para inicialização do acelerador, a geração do pulso de início e o envio dos vetores de entrada ao hardware, o monitoramento do status de conclusão e a leitura do resultado final. Ela evidencia como o *wrapper* automatiza a comunicação com o acelerador, permitindo medições consistentes de latência e simplificando a integração do hardware ao fluxo de testes do projeto.

A Figura 8 apresenta o diagrama do firmware responsável por gerenciar a interação entre o usuário e o acelerador de produto escalar, o esquema evidencia as principais etapas do firmware: exibição do *prompt*, leitura de comandos via console serial, identificação da operação solicitada e execução das rotinas correspondentes. Entre as funcionalidades

estão o cálculo do produto escalar, a geração de relatórios completos, a depuração dos CSRs, o reinício do sistema e o tratamento de comandos inválidos. Além disso, mostra o fluxo interno de cada operação, incluindo leitura dos vetores de entrada, envio ao hardware, emissão do pulso de início, espera pela conclusão e verificação dos resultados, permitindo compreender de forma clara a lógica sequencial do firmware e sua função como camada de controle entre o usuário e o acelerador.

Figura 8 – Representação do firmware em C

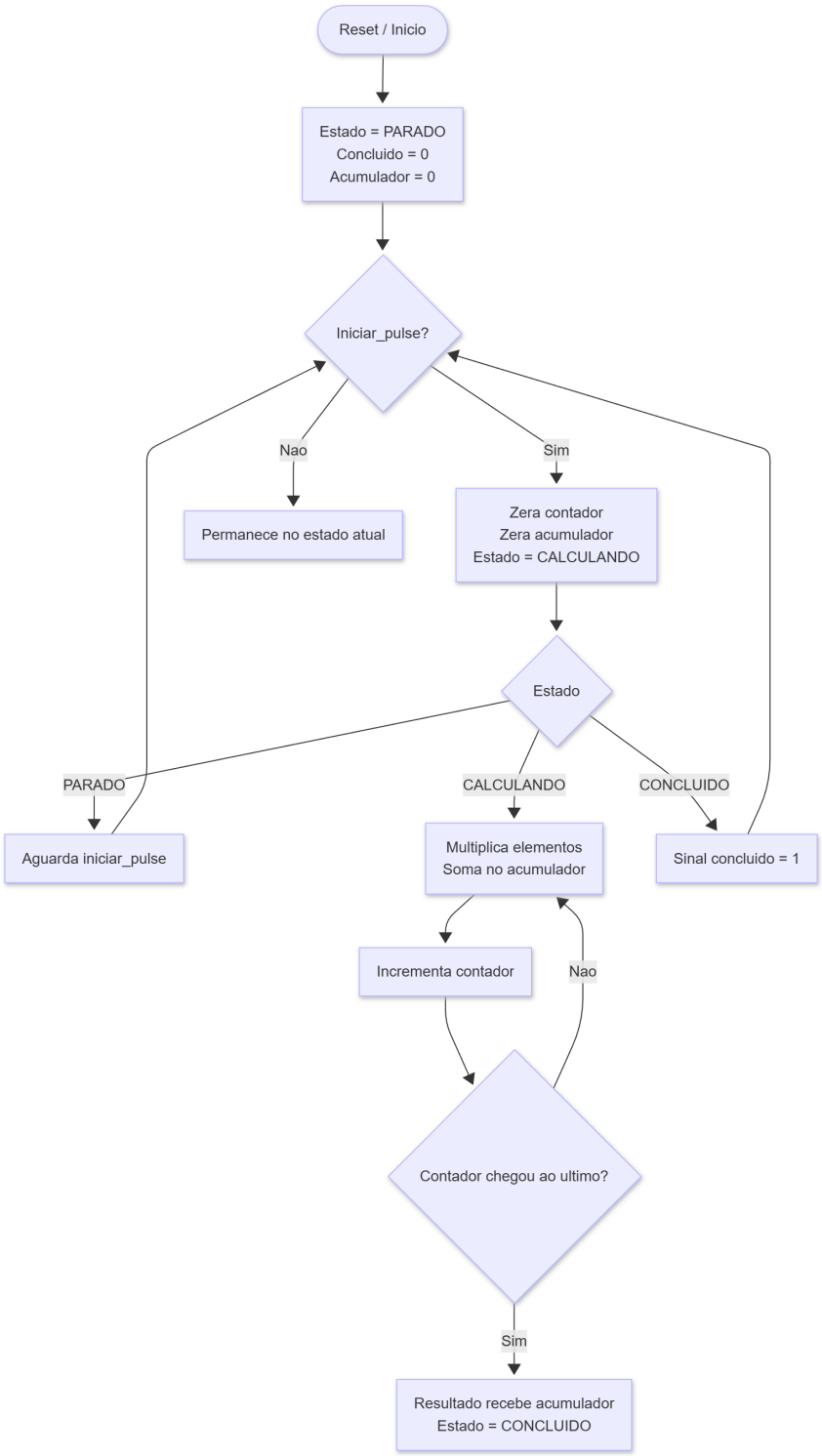


Fonte: Autoria própria (2025).

A Figura 9, descreve de maneira estruturada a lógica operacional do módulo de produto escalar da versão 1, contemplando a máquina de estados finitos, os mecanismos de controle

do contador e do acumulador, bem como o processo de geração do sinal de conclusão.

Figura 9 – Fluxograma da versão 1

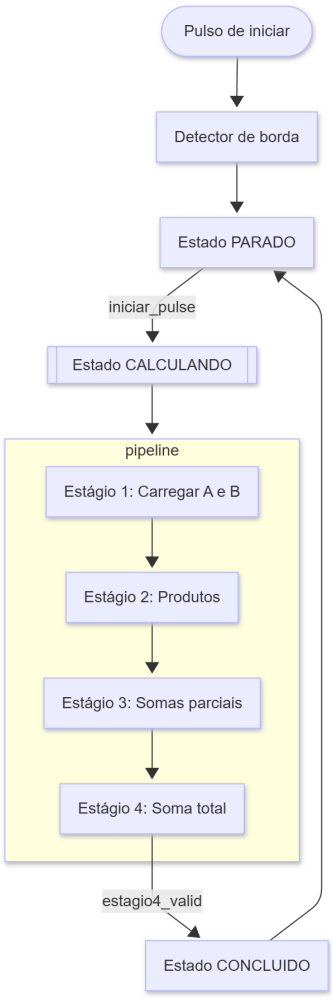


Fonte: Autoria própria (2025).

O diagrama apresenta os três estados principais da FSM: PARADO, CALCULANDO e CONCLUIDO. No estado (PARADO), o módulo aguarda o sinal iniciar_pulse. Ao receber este pulso, o estado passa para (CALCULANDO), onde ocorre a multiplicação dos elementos dos vetores de entrada e a soma acumulativa no registrador acumulador, incrementando o contador a cada operação. Quando o contador atinge o último elemento, a FSM transita para (CONCLUIDO), armazenando o resultado final e acionando o sinal concluído. O esquema também evidencia a possibilidade de reinício do cálculo caso um novo pulso de (iniciar_pulse) seja detectado, reiniciando o acumulador e o contador.

A Figura 10 apresenta o fluxograma da Versão 2 do módulo de produto escalar, ilustrando a estrutura geral do *pipeline* e sua interação com a máquina de estados finitos.

Figura 10 – Fluxograma da versão 2



Fonte: Autoria própria (2025).

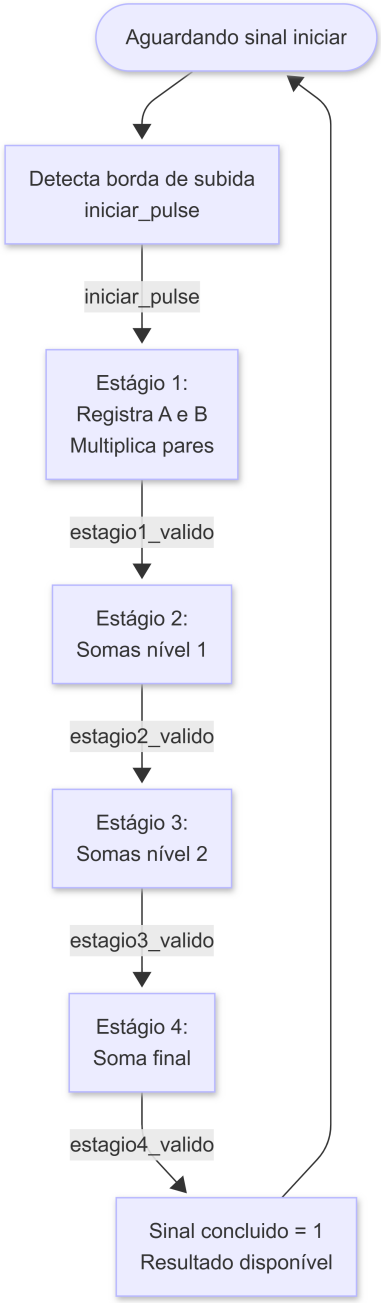
A Figura 10 evidencia o funcionamento do *pipeline* em quatro estágios, que operam

de forma sequencial e sincronizada pelo *clock*, no estágio 1 ocorre o carregamento dos operandos nos registradores internos; no estágio 2 são realizadas as multiplicações elemento a elemento; no estágio 3 são computadas as somas parciais e, finalmente, no estágio 4 é obtida a soma total que compõe o produto escalar.

A máquina de estados finitos controla a ativação desse fluxo. No estado (PARADO) o módulo aguarda o pulso de início. Ao recebê-lo, transita para (CALCULANDO), permitindo que os estágios do *pipeline* avancem a cada ciclo. Quando o Estágio 4 produz um resultado válido, o sistema passa para (CONCLUIDO), disponibilizando o valor final e aguardando um novo pulso para reiniciar o processo. Essa versão apresenta maior organização estrutural e latência determinística, característica resultante do uso do *pipeline*.

A Figura 11 apresenta o fluxograma da Versão 3 do módulo de produto escalar, implementado como um *pipeline* registrado. Nessa arquitetura, cada estágio possui registradores próprios que armazenam dados intermediários, e o controle do fluxo é realizado pelos sinais de validade estagioX_valido que se propagam de um estágio para o seguinte.

Figura 11 – Fluxograma da Versão 3



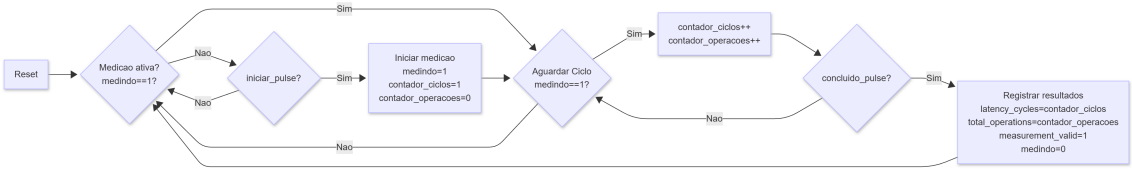
Fonte: Autoria própria (2025).

O *pipeline* registrado é composto por quatro estágios sequenciais: no Estágio 1 ocorre o registro dos operandos e a execução das oito multiplicações iniciais; no Estágio 2, a primeira redução das somas parciais; no Estágio 3, a segunda redução a dois acumulados intermediários; e no Estágio 4, a soma final gera o resultado completo e aciona o sinal concluído.

Como o *pipeline* é sequencial, cada estágio processa os dados apenas após o estágio anterior ter produzido resultados válidos. Essa organização garante previsibilidade e simplifica o controle interno, tornando a Versão 3 eficiente e adequada para aplicações que exigem determinismo e estrutura clara de processamento.

A Figura 12 apresenta o fluxograma da máquina de medição de desempenho implementada nos três módulos de produto escalar, este subsistema especializado é responsável por capturar métricas quantitativas de performance durante a execução das operações de cálculo.

Figura 12 – Fluxograma da máquina de medição de desempenho



Fonte: Autoria própria (2025).

Conforme ilustrado no diagrama, a máquina opera através de um fluxo sequencial que inicia com o reset do sistema, posicionando-o em estado ocioso. A transição para o estado ativo ocorre mediante a detecção do pulso de início (iniciar_pulse), que dispara a inicialização dos contadores: o (contador_ciclos) é configurado para 1, representando o ciclo inicial de latência, de forma análoga, o registrador (contador_operacoes) é inicializado com 0, uma vez que as operações efetivas são contabilizadas dinamicamente durante a execução do acelerador.

Durante a fase ativa, a máquina mantém um *loop* contínuo de incremento do contador de ciclos, monitorando simultaneamente o sinal de conclusão do cálculo principal. A detecção da borda de subida do sinal (concluido_pulse) marca o término da operação, desencadeando o registro final das métricas. Neste momento, o valor acumulado de contador_ciclos é transferido para *(latency_cycles)*, capturando assim a latência total da operação em ciclos de *clock*, enquanto o número de operações é registrado em (total_operations).

O sinal *(measurement_valid)* é ativado para indicar que as métricas estão disponíveis e consistentes, permitindo que sistemas externos possam ler os valores com confiabilidade. Após o registro, a máquina retorna ao estado ocioso, mantendo os valores das métricas até que uma nova medição seja iniciada, assegurando a persistência dos dados até a próxima operação.

Esta implementação unificada nos três módulos permite uma comparação direta e objetiva do desempenho entre as diferentes arquiteturas, fornecendo dados empíricos sobre

a eficiência de cada abordagem em termos de latência e *throughput*, essenciais para análise de *trade-offs* em projetos de sistemas digitais.

A implementação apresentada neste capítulo descreveu de forma completa o processo de construção do sistema, desde a organização do repositório até a execução do firmware no *SoC* gerado pelo LiteX, foram detalhadas as três microarquiteturas do acelerador de produto escalar: versão 1 (FSM), versão 2 (FSM com *pipeline*) e versão 3 (*pipeline* registrado), bem como os módulos auxiliares, incluindo *wrappers*, firmware e a máquina de medição de desempenho, cada componente foi projetado seguindo um fluxo padronizado, facilitando a comparação direta entre as arquiteturas e garantindo a reprodutibilidade dos experimentos.

O uso integral de ferramentas *open-source*, disponibilizadas pela *OSS CAD Suite*, permitiu um fluxo consistente e transparente de simulação, síntese, *place & route* e gravação da FPGA, consolidando um ambiente de desenvolvimento acessível, automatizado e alinhado às boas práticas de engenharia de sistemas digitais, a integração entre hardware e software viabilizada pelo LiteX e pela *toolchain* RISC-V, demonstrou-se eficaz para a construção e validação do *SoC*, possibilitando testes reais e coleta confiável de métricas de desempenho.

Com isso, o capítulo estabelece a base técnica necessária para compreender a implementação prática do projeto, preparando o terreno para a análise comparativa e avaliação quantitativa das diferentes arquiteturas apresentadas no capítulo seguinte.

5 RESULTADOS

Este capítulo apresenta a análise experimental das três microarquiteturas do acelerador de produto escalar implementadas: a FSM, a FSM com *pipeline* e o *pipeline* registrado. A implementação em software executada no processador PicoRV32 foi utilizada como referência para validação funcional, assegurando a correção dos resultados do cálculo do produto escalar obtidos pelo acelerador em hardware.

O objetivo desta análise consiste em avaliar quantitativamente o desempenho, o custo em área e a eficiência de cada microarquitetura, validando a viabilidade da extensão funcional proposta para a aceleração de operações vetoriais em arquiteturas RISC-V.

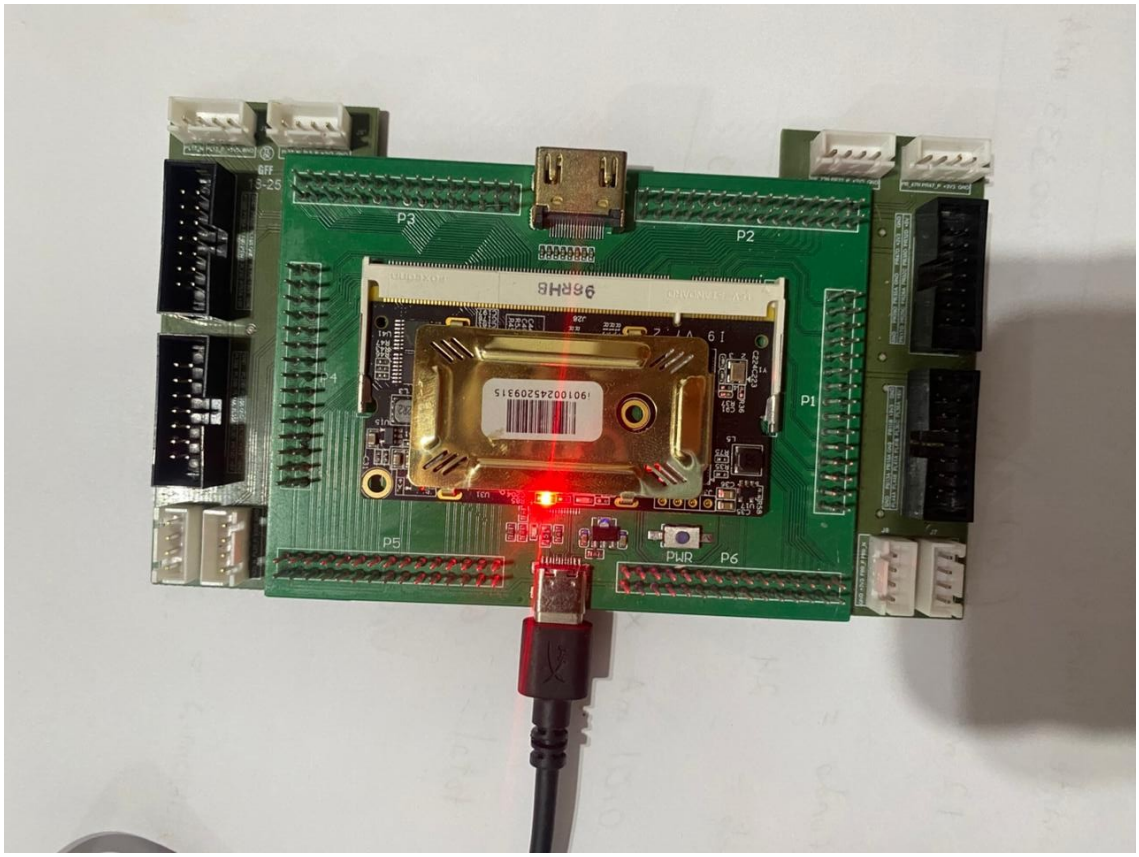
As avaliações realizadas contemplam métricas amplamente utilizadas na literatura de arquitetura de computadores, incluindo latência, *throughput*, eficiência (OPC – Operações por Ciclo) e utilização de recursos da FPGA, como *LUTs*, FFs, DSPs e BRAMs. A organização dos resultados busca evidenciar, de forma clara e sistemática, os efeitos das diferentes estratégias microarquiteturais adotadas, permitindo uma avaliação comparativa fundamentada e coerente com os objetivos estabelecidos neste trabalho.

5.1 Ambiente experimental

Os experimentos foram realizados na placa FPGA Colorlight i9, equipada com o dispositivo Lattice ECP5 LFE5U-45, utilizando o *SoC* gerado pelo *framework* LiteX com núcleo PicoRV32. Todas as versões do acelerador foram sintetizadas e implementadas sob as mesmas condições de *clock*, com frequência fixa de 60 MHz, garantindo comparabilidade direta entre as arquiteturas. A placa utilizada nos testes é mostrada na Figura 13.

A FPGA, dispositivo fabricado em tecnologia CMOS de 40 nm, caracterizado por oferecer um equilíbrio entre desempenho computacional, flexibilidade arquitetural e baixo consumo de energia, dispõe de aproximadamente 44 mil *LUTs*, organizadas em unidades funcionais programáveis (PFUs), além de 108 blocos de memória embarcada *sysMEM* de 18 kbits, totalizando cerca de 1,9 Mbits de memória interna, e aproximadamente 351 kbits de memória distribuída. Para aplicações de processamento intensivo, a FPGA integra 72 multiplicadores dedicados de 18 × 18 bits, possibilitando a implementação eficiente de operações aritméticas, tais como multiplicação, acumulação e operações em ponto fixo. O subsistema de *clock* do dispositivo inclui quatro PLLs e quatro DLLs, permitindo a geração e o gerenciamento preciso de múltiplos domínios de *clock*. (Semiconductor, 2025)

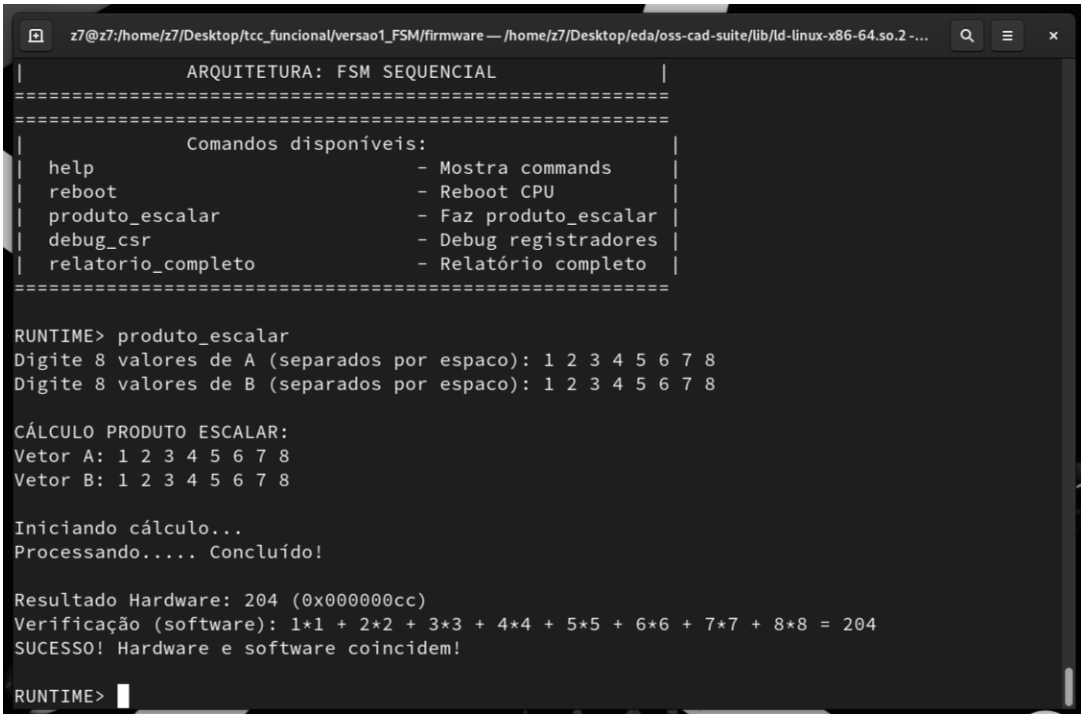
Figura 13 – Placa Lattice ECP5 LFE5U-45



Fonte: Arquivo pessoal (2025).

A Figura 14 apresenta a saída do terminal correspondente à execução do produto escalar utilizando a microarquitetura FSM sequencial. O print inclui a inicialização do *SoC* com o núcleo PicoRV32, o envio dos operandos pelos registradores CSR e o resultado final calculado pelo acelerador. Essa execução representa o comportamento típico da versão sequencial, servindo como referência para comparação com as demais microarquiteturas.

Figura 14 – Produto Escalar no *SoC* (v1)



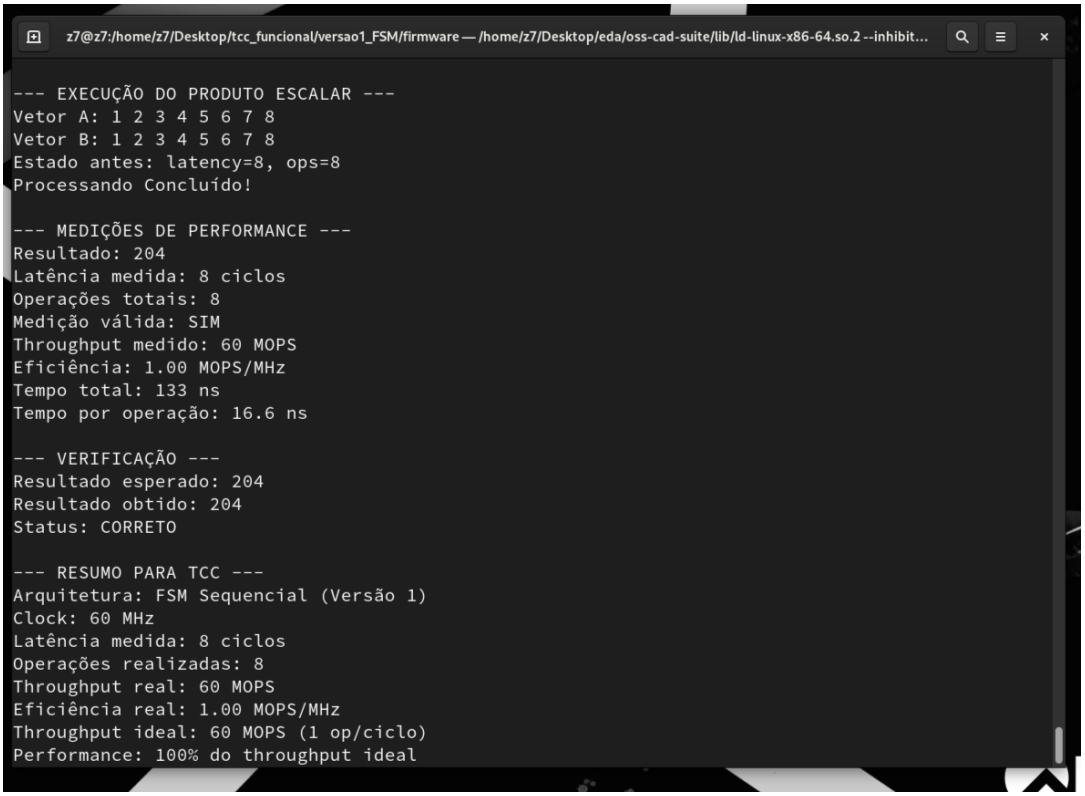
Fonte: Autoria própria (2025).

Os testes consistiram na execução repetida do cálculo do produto escalar entre vetores de oito elementos, com valores inteiros de 32 bits gerados de forma controlada pelo firmware. Para cada versão do acelerador, foram coletados os seguintes parâmetros:

- Latência em ciclos de *clock*;
- Tempo total de execução (ns)
- *Throughput* em MOPS;
- Operações por ciclo (OPC);
- Utilização de *LUTs*, FFs, DSPs e BRAMs;
- Comparação com o resultado em software.

A Figura 15 apresenta o relatório completo gerado durante a execução dos testes para a versão 1 do acelerador, o print reúne em um único registro, as medições capturadas pelo firmware, incluindo o valor final do produto escalar e os indicadores de desempenho extraídos pelo sistema de monitoramento, servindo como referência para as análises apresentadas nas próximas seções.

Figura 15 – Relatório no *SoC* (v1)



Fonte: Autoria própria (2025).

5.2 Resultados de latência e tempo

A Tabela 2 apresenta a latência medida em ciclos de *clock*, o tempo total de execução e o tempo médio por operação para cada microarquitetura analisada.

Tabela 2 – Latência e tempo de execução por microarquitetura

Microarquitetura	Latência (ciclos)	Tempo total (ns)	Tempo por operação (ns)
FSM	8	133	16,6
FSM com *Pipeline*	5	83	10,4
Pipeline Registrado	4	66	8,3

Fonte: Autoria própria (2025).

A arquitetura FSM apresentou a maior latência e o maior tempo total de execução, consequência direta de sua natureza sequencial, na qual cada multiplicação e acumulação é realizada em ciclos distintos. Já a versão FSM com *pipeline* demonstrou redução significativa tanto na latência quanto no tempo, evidenciando o impacto positivo da sobreposição parcial das operações internas.

A arquitetura *pipeline* registrado obteve os melhores resultados dentre todas as versões, alcançando a menor latência (4 ciclos) e o menor tempo total de execução (66 ns), representando uma redução aproximada de 50% em relação à FSM convencional. Observa-se igualmente a diminuição progressiva do tempo por operação, que passou de 16,6 ns na FSM para 8,3 ns no *Pipeline* Registrado, confirmando a maior eficiência temporal proporcionada pela segmentação completa do *datapath*.

5.3 *Throughput* e eficiência

O *throughput* foi calculado conforme a Equação 2, sendo expresso em MOPS, enquanto a eficiência foi determinada de acordo com a Equação 3, expressa em OPC. A Tabela 3 apresenta os valores de *throughput* e eficiência obtidos para cada microarquitetura implementada.

Tabela 3 – *Throughput* e eficiência por microarquitetura

Microarquitetura	*Throughput* (MOPS)	Eficiência (OPC)
FSM	60	1,00
FSM com *pipeline*	96	1,60
Pipeline registrado	120	2,00

Fonte: Autoria própria (2025).

A arquitetura FSM apresentou o menor *throughput* entre as microarquiteturas avaliadas, resultado da execução estritamente sequencial das operações e da inexistência de paralelismo interno. A adoção de técnicas de *pipeline* proporcionou ganhos progressivos de desempenho, com a FSM com *pipeline* atingindo 96 MOPS, enquanto a arquitetura *pipeline* registrado obteve o maior *throughput*, alcançando 120 MOPS, evidenciando a superioridade da segmentação completa do *datapath*.

A eficiência, expressa em operações por ciclo (OPC), reflete o nível de aproveitamento do ciclo de *clock* por cada arquitetura. Nesse contexto, a FSM apresentou o menor índice de eficiência, enquanto a FSM com *pipeline* demonstrou melhoria intermediária. A arquitetura *pipeline* registrado apresentou o maior aproveitamento do paralelismo temporal entre as três versões avaliadas resultando em maior produtividade por ciclo de processamento e maior regularidade no fluxo de operações.

5.4 Utilização de recursos

A Tabela 4 apresenta a utilização de recursos lógicos na FPGA para cada versão do acelerador.

Tabela 4 – Utilização de recursos por microarquitetura

Recurso	Capacidade	Versão 1	Versão 2	Versão 3
LUTs	43.848	7.512 (17,13%)	8.281 (18,88%)	8.270 (18,86%)
Flip-Flops	43.848	3.431 (7,82%)	4.653 (10,61%)	4.198 (9,57%)
DSPs	72	4 (5,55%)	32 (44,44%)	32 (44,44%)
BRAM	108	41 (37,96%)	41 (37,96%)	41 (37,96%)

Fonte: Autoria própria (2025).

Observa-se que a Versão 1 (FSM) apresentou o menor consumo de *LUTs* e flip-flops, com ocupação de 17,13% e 7,82%, respectivamente, refletindo sua organização estrutural mais simples e a ausência de paralelismo significativo. Tal característica resulta em menor complexidade de controle e menor necessidade de registradores intermediários, contribuindo para a reduzida área ocupada na FPGA.

Em contrapartida, as Versões 2 e 3, baseadas em arquiteturas *pipeline*, apresentam aumento perceptível na utilização de *LUTs* e registradores, decorrente da inserção de estágios intermediários e da necessidade de sincronização entre os blocos funcionais. A Versão 2 alcança 18,88% de uso de *LUTs* e 10,61% de flip-flops, enquanto a Versão 3 apresenta valores muito próximos, com 18,86% e 9,57%, respectivamente. Esse incremento é esperado em arquiteturas *pipeline*, pois o aumento do paralelismo e da profundidade do *pipeline* implica maior consumo de recursos lógicos.

No que se refere aos DSPs, verifica-se uma diferença mais acentuada entre as arquiteturas: enquanto a Versão 1 utiliza apenas 4 unidades 5,55%, as Versões 2 e 3 empregam 32 DSPs 44,44%, evidenciando que as arquiteturas *pipeline* exploram de forma mais intensiva as unidades aritméticas dedicadas para maximizar o paralelismo em nível de dados. Esse comportamento indica que o ganho de desempenho nessas versões está diretamente associado à maior capacidade de processamento simultâneo.

Quanto ao uso de BRAM, observa-se que todas as três arquiteturas apresentam a mesma ocupação, 41 blocos, correspondentes a 37,96%, do total disponível. Esse comportamento uniforme ocorre porque as três implementações compartilham a mesma organização de memória interna, sem variações no tamanho dos *buffers* ou na quantidade de dados armazenados. Além disso, o valor permanece dentro de uma faixa confortável de utilização do dispositivo, indicando que ainda existe margem significativa para futuras expansões, aumento do tamanho dos vetores ou inserção de novos módulos sem comprometer os recursos de memória da FPGA.

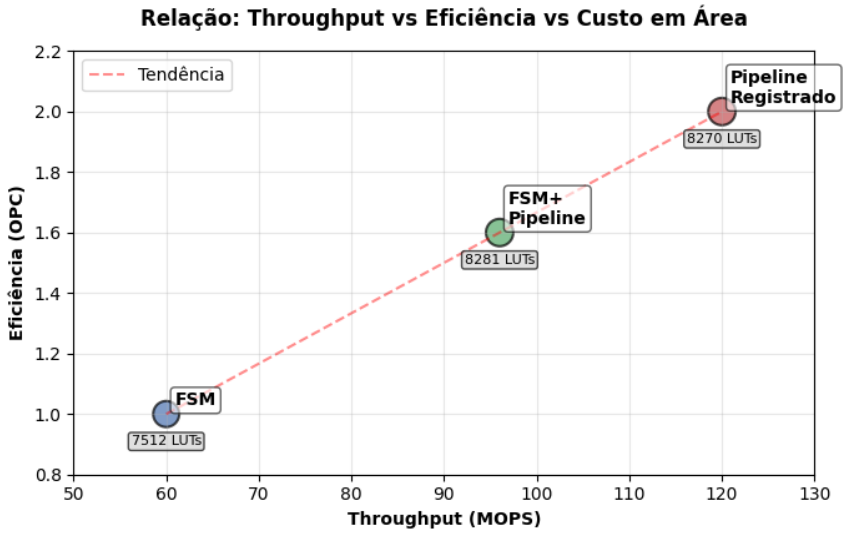
De modo geral, os resultados evidenciam um *trade-off* clássico entre área e desempenho: arquiteturas mais simples priorizam a economia de recursos, enquanto versões *pipeline*

sacrificam área em prol de maior *throughput* e eficiência computacional. Dessa forma, a escolha da microarquitetura deve considerar não apenas o desempenho bruto, mas também as restrições de área e consumo do sistema alvo, de acordo com os requisitos da aplicação.

5.5 Análise comparativa

A Figura 16 apresenta uma comparação geral entre as três microarquiteturas desenvolvidas, considerando simultaneamente *throughput*, eficiência e custo em área. A arquitetura FSM representa a solução mais simples, exibindo o menor *throughput* e eficiência, mas também o menor consumo de recursos lógicos. Já a versão FSM+*Pipeline* introduz paralelismo parcial, elevando o desempenho de forma significativa e resultando em um aumento moderado na utilização de *LUTs* devido aos registradores e estágios intermediários adicionados.

Figura 16 – Comparação geral entre as microarquiteturas



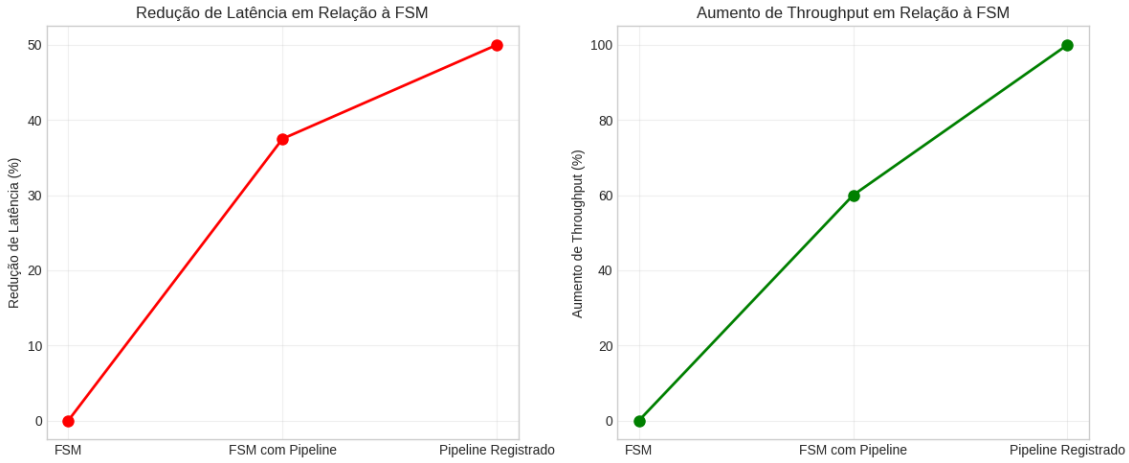
Fonte: Autoria própria (2025).

Como podemos ver na Figura 16, a arquitetura *pipeline* registrado atinge o melhor desempenho entre as três versões, apresentando os maiores valores de *throughput* e eficiência, enquanto mantém um consumo de área semelhante ao da versão intermediária. De modo geral, observa-se uma tendência clara de que o aprofundamento do *pipeline* leva a ganhos consistentes de desempenho, sem provocar um aumento expressivo no custo em área, reforçando a adequação do uso de paralelismo para otimização do acelerador.

A Figura 17 apresenta os resultados referentes à redução percentual de latência e ao aumento percentual de *throughput* das arquiteturas avaliadas em relação ao modelo FSM,

permitindo visualizar de forma direta os ganhos proporcionados pelas abordagens com *pipeline*.

Figura 17 – Redução de latência e aumento de *throughput* em relação à arquitetura FSM



Fonte: Autoria própria (2025).

Observa-se que a adoção do *pipeline* nas versões FSM+*Pipeline* e *Pipeline* Registrado resulta em reduções significativas de latência quando comparadas à arquitetura FSM. A versão intermediária já apresenta uma diminuição próxima de 37,5%, enquanto a versão *Pipeline* Registrado alcança aproximadamente 50%, evidenciando que o aprofundamento do *pipeline* contribui para diminuir o tempo total de execução ao permitir maior sobreposição de operações entre estágios.

De forma complementar, a Figura também evidencia o aumento percentual de *throughput* obtido com a aplicação progressiva de paralelismo. A arquitetura FSM+*Pipeline* apresenta cerca de 60% de ganho, enquanto a arquitetura *pipeline* registrado atinge aproximadamente 100%, dobrando o desempenho da solução de referência, esses resultados demonstram que o uso de *pipelines* mais estruturados não apenas reduz a latência, mas amplia substancialmente a taxa de processamento, consolidando a abordagem *pipeline* como a mais eficiente entre as microarquiteturas avaliadas.

Esses resultados confirmam que a adoção de *pipeline* e paralelismo em nível de hardware influencia diretamente a eficiência do sistema, validando a hipótese central deste trabalho.

6 CONCLUSÕES

A implementação e a análise do acelerador de produto escalar desenvolvidos neste trabalho mostraram que é realmente possível melhorar o desempenho de operações vetoriais em sistemas RISC-V usando extensões funcionais e uma FPGA como plataforma de teste. As três microarquiteturas criadas: FSM sequencial, FSM com *pipeline* e *pipeline* registrado, permitiram observar de forma clara como o paralelismo influencia o comportamento do sistema. Entre elas, a versão com *pipeline* registrado se destacou por entregar a menor latência e o maior *throughput*, mantendo um uso de recursos ainda razoável.

Durante o desenvolvimento, foi seguido todo o fluxo tradicional de projeto digital: descrição em RTL, síntese, *place-and-route* e, finalmente, testes em hardware real. Tudo isso foi feito utilizando ferramentas *open-source*, o que também reforça a ideia de que soluções acessíveis podem ser usadas para projetos de extensão funcional. Tanto as simulações quanto os testes na FPGA ECP5 LFE5U-45 confirmaram o funcionamento correto do acelerador, e os resultados batem com o cálculo em software no PicoRV32. A comunicação via registradores CSR mostrou-se simples e funcionou bem dentro do ambiente LiteX.

Entre as contribuições do trabalho estão: a implementação das três microarquiteturas, a comparação experimental entre elas e a validação de uma extensão funcional prática para operações vetoriais em RISC-V. Além disso, o projeto deixa um fluxo de desenvolvimento completo e reproduzível que pode servir de base para trabalhos futuros.

Em resumo, os objetivos propostos foram atingidos. O acelerador funcionou como esperado, trouxe ganhos reais de desempenho e se mostra como uma alternativa viável para aplicações embarcadas que precisam de mais velocidade sem aumentar demais os custos.

Como trabalhos futuros, o projeto pode ser ampliado em diversas direções, incluindo o aumento do tamanho dos vetores suportados, a adição de suporte a ponto flutuante e a extensão das instruções customizadas para operações mais complexas, como produtos matriciais e convoluções, aproximando o acelerador de aplicações comuns em visão computacional e aprendizado de máquina. Também é relevante investigar o comportamento do hardware em termos de consumo energético e impacto térmico, fortalecendo sua aplicabilidade em sistemas embarcados de alto desempenho. Além disso, uma linha de pesquisa promissora consiste em adaptar o acelerador para tarefas de processamento digital aplicadas à leitura de qubits, como filtragem FIR/IIR, integração e operações de redução vetorial, possibilitando avaliar sua viabilidade como componente auxiliar em controladores quânticos modernos, reduzindo a latência do caminho clássico e contribuindo

para experimentos que demandam detecção rápida e *feedback* em tempo quase real.

REFERÊNCIAS

ANTON, H.; RORRES, C. **Álgebra Linear com Aplicações**. 9. ed. Porto Alegre: Bookman, 2012. Tradução técnica de Claus Ivo Doering. Obra original: Elementary Linear Algebra: Applications Version, 10th ed. ISBN 9780470432051.

CUI, E.; LI, T.; WEI, Q. Risc-v instruction set architecture extensions: A survey. **IEEE Access**, v. 11, p. 24696–24711, 2023.

HARRIS, S. L.; HARRIS, D. **Digital Design and Computer Architecture: RISC-V Edition**. Cambridge, MA: Elsevier, 2022. ISBN 978-0-12-820064-3.

HENNESSY, J. L.; PATTERSON, D. A. Computer architecture: A quantitative approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2017. Sixth Edition.

KERMARREC, F. *et al.* **LiteX: an open-source SoC builder and library based on Migen Python DSL**. 2020. Disponível em: https://arxiv.org/abs/2005.02506.

LI, Y. *et al.* A gpu-outperforming fpga accelerator architecture for binary convolutional neural networks. **J. Emerg. Technol. Comput. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 14, n. 2, jul. 2018. ISSN 1550-4832. Disponível em: https://doi.org/10.1145/3154839.

MA, C. *et al.* Design of an soc based on 32-bit risc-v processor with low-latency lightweight cryptographic cores in fpga. **Future Internet**, MDPI, v. 15, n. 5, p. 186, 2023. Disponível em: https://www.mdpi.com/1999-5903/15/5/186.

NECHI, A. *et al.* Fpga-based deep learning inference accelerators: Where are we standing? **ACM Transactions on Reconfigurable Technology and Systems**, ACM, New York, NY, USA, v. 16, n. 4, p. 1–32, 2023. Disponível em: https://doi.org/10.1145/3613963.

SEMICONDUCTOR, L. **What is an FPGA?** 2024. Disponível em: https://www.latticesemi.com/en/What-is-an-FPGA. Acesso em: 13 nov. 2025.

SEMICONDUCTOR, L. **Lattice FPGA Datasheet**. 2025. <https://www.latticesemi.com/view_document?document_id=50461>.

SILVA, E. U. P. d. **O Processo de Desenvolvimento de uma CPU RISC-V**. Dissertação (Monografia (Bacharelado em Ciência da Computação)) — Universidade de São Paulo (USP), São Paulo, 2021. Instituto de Matemática e Estatística. Disponível em: <https://linux.ime.usp.br/~vitors/mac0499/>. Acesso em: 11 nov. 2025.

SZE, V. *et al.* Efficient processing of deep neural networks: A tutorial and survey. **Proceedings of the IEEE**, IEEE, Piscataway, NJ, USA, v. 105, n. 12, p. 2295–2329, 2017.

WATERMAN, A. *et al.* **The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA**. [S.l.], 2011. Disponível em: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.html.

WILSON, A. E. *et al.* Improving fault tolerance for fpga socs through post-radiation design analysis. **ACM Trans. Reconfigurable Technol. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 17, n. 3, set. 2024. ISSN 1936-7406. Disponível em: https://doi.org/10.1145/3674841.

XAVIER, L. **SoC LiteX - Produto Escalar**. 2025. <https://github.com/lucasxavier9/Soc-LiteX-ProdutoEscalar>. Acesso em: 27 nov. 2025.

APÊNDICE A

```
// rtl/produto_escalar_v1_com_medicao.sv
module produto_escalar(
  input  logic         clk_i,
  input  logic         rst_i,
  input  logic         iniciar,
  input  logic [31:0]  a0, a1, a2, a3, a4, a5, a6, a7,
  input  logic [31:0]  b0, b1, b2, b3, b4, b5, b6, b7,
  output logic [63:0]  resultado,
  output logic         concluido,

// Sinais de medicao de performance
  output logic [31:0] ciclos_latencia,
  output logic [31:0] total_operacoes,
  output logic        medicao_valida
);

  // FSM: Estados
  typedef enum logic [1:0] {
      ESTADO_PARADO     = 2'b00,
      ESTADO_CALCULANDO = 2'b01,
      ESTADO_CONCLUIDO  = 2'b10
  } estado_t;

  estado_t estado, estado_proximo;
  logic [2:0] contador, contador_proximo;
  logic signed [63:0] acumulador, acumulador_proximo;
  logic [63:0] resultado_proximo;

  // Arrays para calculo
  logic signed [31:0] a_signed [0:7];
  logic signed [31:0] b_signed [0:7];

  // Detector de borda de subida
  logic iniciar_prev;
  always_ff @(posedge clk_i) begin
    if (rst_i) iniciar_prev <= 1'b0;
    else iniciar_prev <= iniciar;
  end
  wire iniciar_pulse = iniciar && !iniciar_prev;

  // Desempacotar entradas para arrays
  always_comb begin
    a_signed[0] = a0; a_signed[1] = a1; a_signed[2] = a2; a_signed[3] =
        a3;
    a_signed[4] = a4; a_signed[5] = a5; a_signed[6] = a6; a_signed[7] =
        a7;

    b_signed[0] = b0; b_signed[1] = b1; b_signed[2] = b2;   b_signed[3]
```

```
        = b3;
47    b_signed[4] = b4; b_signed[5] = b5; b_signed[6] = b6; b_signed[7] =
          b7;
48  end
49
50  // FSM combinacional + calculo
51  always_comb begin
52    estado_proximo      = estado;
53    contador_proximo    = contador;
54    acumulador_proximo  = acumulador;
55    resultado_proximo   = resultado;
56
57    case (estado)
58      ESTADO_PARADO: begin
59        if (iniciar_pulse) begin
60          estado_proximo     = ESTADO_CALCULANDO;
61          contador_proximo   = 3'd0;
62          acumulador_proximo = 64'sd0;
63        end
64      end
65
66      ESTADO_CALCULANDO: begin
67        // Calculo do proximo acumulador
68        acumulador_proximo = acumulador + (a_signed[contador] *
              b_signed[contador]);
69        contador_proximo   = contador + 3'd1;
70
71        if (contador == 3'd7) begin
72          estado_proximo    = ESTADO_CONCLUIDO;
73          resultado_proximo = acumulador_proximo;
74        end
75      end
76
77      ESTADO_CONCLUIDO: begin
78        // Mantem resultado estavel ate proximo iniciar
79        if (iniciar_pulse) begin
80          estado_proximo     = ESTADO_CALCULANDO;
81          contador_proximo   = 3'd0;
82          acumulador_proximo = 64'sd0;
83        end
84      end
85
86      default: begin
87        estado_proximo = ESTADO_PARADO;
88      end
89    endcase
90  end
91
92  // FSM sequencial
93  always_ff @(posedge clk_i) begin
94    if (rst_i) begin
95      estado      <= ESTADO_PARADO;
96      contador    <= 3'd0;
```

```
97        acumulador  <= 64'sd0;
98        resultado   <= 64'sd0;
99        concluido   <= 1'b0;
100     end else begin
101       estado      <= estado_proximo;
102       contador    <= contador_proximo;
103       acumulador  <= acumulador_proximo;
104       resultado   <= resultado_proximo;
105
106       // Sinal de concluido
107       if (estado_proximo == ESTADO_CONCLUIDO) begin
108         concluido <= 1'b1;
109       end else if (iniciar_pulse) begin
110         concluido <= 1'b0;
111       end
112     end
113   end
114
115 endmodule
```

Listing 1 – Módulo *produto_escalar* versão 1 - FSM