Universidade Federal de São Carlos
Departamento de Computação

Trabalho de Conclusão de Curso

Engenharia de Computação

Análise de Desempenho de Pipeline em Processadores RISC-V implementados em FPGA

Estudante: **Thiago Martins**

Orientador: **Prof. Dr. Ricardo Menotti**

São Carlos, 6 de março de 2025

Resumo

No cenário atual de desenvolvimento de hardware, existem diversos tipos de processadores no mercado, cada um com configurações específicas que atendem a diferentes necessidades de desempenho, consumo de energia e complexidade. A arquitetura RISC-V surge como uma solução modular e flexível, permitindo a adaptação do número de estágios de pipeline e a inclusão de otimizações conforme a aplicação desejada. Este trabalho analisa o impacto de diferentes configurações de pipeline e otimizações, como *bypass* e *early branch*, no desempenho, consumo de energia e uso de recursos lógicos em processadores baseados no núcleo VexRiscv, uma implementação altamente configurável da arquitetura RISC-V. Foram avaliados processadores com dois, três e quatro estágios de pipeline, considerando operações com e sem multiplicação em hardware. Os resultados mostram que processadores com mais estágios oferecem maior desempenho quando estão com otimizações ativas, mas com um aumento significativo no consumo de energia e uso de recursos. Processadores de três estágios apresentam um equilíbrio ideal entre desempenho e consumo, enquanto os de dois estágios destacam-se pela simplicidade e baixo consumo de energia, mas com desempenho reduzido. A ativação do *bypass* e do *early branch* demonstrou melhorar significativamente o desempenho, especialmente em operações complexas como multiplicação, mas com um custo adicional em termos de consumo de energia e recursos. A análise também revelou que a inclusão da multiplicação hardware eleva o consumo de energia e recursos, mas as otimizações de pipeline compensam esse aumento ao melhorar a eficiência do processador. Este estudo destaca a importância de equilibrar desempenho e eficiência energética em sistemas embarcados e de alta performance, destacando a versatilidade da arquitetura RISC-V e do núcleo VexRiscv como ferramentas poderosas para o desenvolvimento de hardware personalizado.

Abstract

In the current hardware development scenario, there are several types of processors on the market, each with specific configurations that meet different performance, power consumption and complexity needs. The RISC-V architecture emerges as a modular and flexible solution, allowing the adaptation of the number of pipeline stages and the inclusion of optimizations according to the desired application. This work analyzes the impact of different pipeline configurations and optimizations, such as bypass and early branch, on performance, power consumption and logical resource use in VexRiscv-based processors, a highly configurable implementation of the RISC-V architecture. Processors with two, three and four stages of pipeline were evaluated, considering operations with and without multiplication. The results show that processors with more stages, offer higher performance when they are active optimizations, but with a significant increase in energy consumption and resource use. Three-stage processors have an ideal balance between performance and power consumption, while two-stage processors stand out for simplicity and lower power consumption but with reduced performance. The activation of bypass and early branch has been shown to significantly improve performance, especially in complex operations such as multiplication, but with an additional cost in terms of energy consumption and resources. The analysis also revealed that the inclusion of multiplication increases energy and resource consumption, but pipeline optimizations offset this increase by improving processor efficiency. This study reinforces the importance of understanding these trade-offs to optimize the performance and efficiency of embedded systems and high performance, highlighting the versatility of RISC-architectureV and the VexRiscv core as powerful tools for custom hardware development.

Sumário

Lista de Figuras . . . . **ii**

1 INTRODUÇÃO **1**
1.1 Objetivos **2**
1.1.1 Objetivo Geral 2
1.1.2 Objetivo Específico 2

2 EMBASAMENTO TEÓRICO **3**
2.1 Field-Programmable Gate Array **3**
2.2 Linguagem de Descrição de Hardware **3**
2.3 SpinalHDL **4**
2.4 Arquitetura Geral de Processadores **5**
2.4.1 Arquitetura do conjunto de instruções 5
2.4.2 Estruturas e estágios de Pipeline 6
2.4.3 Bypass 7
2.4.4 Early Branch 8
2.5 RISC-V **9**
2.6 VexRISCV **9**
2.6.1 Microcontrolador Murax 10
2.7 Dhrystone **11**

3 DESENVOLVIMENTO **13**
3.1 Resultados e Análises **14**
3.1.1 Frequência máxima 17
3.1.2 Elementos lógicos 17
3.1.3 Registradores 18
3.1.4 Métricas Dhrystone 19
3.1.5 Consumo de Energia 20
3.1.6 Análise comparativa 21

4 CONCLUSÃO **25**

A APÊNDICE **27**
A.1 Murax(I) síntese IDE Gowin **27**
A.1.1 Saídas 28

REFERÊNCIAS **31**

Lista de Figuras

Figura 1 – Work Flow de desenvolvimento SpinalHDL 5
Figura 2 – Os Cinco Estágios de Pipeline RISC-V Baseados na Arquitetura Harvard . 7
Figura 3 – Arquitetura do SOC Murax 11
Figura 4 – Estrutura Geral de Configuração de Estágios 14
Figura 5 – Estrutura Geral de Configuração de *Bypass* e *Early Branch* 14
Figura 6 – Comparativo microcontroladores 22
Figura 7 – Comparativo microcontroladores 23
Figura 8 – Resultados da simulação do Dhrystone no Icarus Verilog. 29
Figura 9 – Resultados da execução do Dhrystone na placa DE10 Standard. 30

1 Introdução

No cenário atual de desenvolvimento de hardware, a arquitetura RISC-V emerge como uma solução modular e flexível, permitindo a adaptação do número de estágios de pipeline e a inclusão de otimizações conforme a aplicação desejada. Este trabalho analisa o impacto de diferentes configurações de pipeline e otimizações, como *bypass* e *early branch*, no desempenho, consumo de energia e uso de recursos lógicos em processadores baseados no núcleo VexRiscv, uma implementação altamente configurável da arquitetura RISC-V.

A escolha da arquitetura VexRiscv (SpinalHDL 2024) para esta pesquisa deve-se à sua flexibilidade, modularidade e eficiência, características que a tornam ideal para projetos de hardware personalizados e de alto desempenho. O VexRiscv permite a adaptação do pipeline, a inclusão de extensões de instruções e a integração de periféricos específicos, atendendo a uma ampla gama de aplicações, desde sistemas embarcados de baixo consumo até processadores de alta performance. A linguagem SpinalHDL (SpinalHDL 2024), utilizada para descrever o VexRiscv, facilita a prototipagem rápida e a verificação formal, reduzindo o tempo de desenvolvimento e aumentando a confiabilidade do design. Além disso, a comunidade ativa no GitHub contribui para a resolução de problemas e para a implementação de novas funcionalidades, oferecendo diversos módulos RISC-V que podem ser combinados para atender a diferentes necessidades. Essa combinação de modularidade, eficiência e suporte a ferramentas modernas faz do VexRiscv uma solução versátil e poderosa, justificando sua escolha para estudos e implementações em projetos acadêmicos e industriais.

O tema foi escolhido devido ao crescente interesse na arquitetura RISC-V, que oferece um desempenho notável, especialmente em termos de flexibilidade e personalização. Dada a variedade de implementações disponíveis em plataformas como o GitHub, é essencial compreender os requisitos para escolher a configuração adequada. Isso inclui o dimensionamento do processador ideal, considerando consumo de energia, frequência máxima de operação e a área ocupada na FPGA após a síntese.

Com este estudo, pretende-se aprofundar o conhecimento na área de arquitetura de processadores e contribuir com informações práticas que auxiliem outros desenvolvedores a otimizar custos e recursos em suas aplicações.

1.1 Objetivos

1.1.1 Objetivo Geral

Este trabalho tem como objetivo geral analisar o impacto de diferentes configurações de pipeline e otimizações no desempenho, consumo de energia e uso de recursos lógicos em processadores baseados no núcleo VexRiscv, uma implementação altamente configurável da arquitetura RISC-V. Além disso, busca-se aprofundar o conhecimento na área de arquitetura de processadores e fornecer informações práticas que auxiliem outros desenvolvedores na otimização de custos e recursos em suas aplicações.

1.1.2 Objetivo Específico

Visando alcançar o objetivo geral, algumas metas específicas devem ser atendidas:

- Avaliar o impacto das otimizações, como *bypass* e *early branch*, no desempenho de microcontroladores.
- Analisar o consumo de energia e o uso de recursos lógicos em diferentes configurações de pipeline do VexRiscv.
- Explorar a flexibilidade e modularidade da arquitetura VexRiscv para adaptar o pipeline e incluir extensões conforme a necessidade.
- Estudar a influência da escolha da configuração do processador em relação a consumo de energia, frequência máxima de operação e área ocupada na FPGA após a síntese.
- Contribuir com informações práticas que auxiliem desenvolvedores na escolha da melhor configuração para suas aplicações.

2 Embasamento Teórico

Esta seção apresenta os conceitos e tecnologias fundamentais que sustentam a análise realizada neste trabalho. Serão abordados os princípios teóricos essenciais para compreender o desenvolvimento e a avaliação de processadores, incluindo técnicas de otimização e métricas de desempenho.

2.1 Field-Programmable Gate Array

Um Arranjo de Portas Programáveis em Campo (FPGAs[1]), é um tipo de circuito integrado altamente flexível, projetado para que suas funções lógicas possam ser configuradas pelo usuário após a fabricação (Farooq, Marrakchi e Mehrez 2012). Diferentemente dos processadores convencionais, que executam instruções de software, os FPGAs são compostos por blocos lógicos reconfiguráveis que realizam operações lógicas. Essa reconfigurabilidade permite que um FPGA seja adaptado para atender a requisitos específicos de aplicação, como processamento de dados em alta velocidade, controle em tempo real ou prototipagem de circuitos digitais complexos.

Os FPGAs são compostos por elementos lógicos programáveis, interconexões configuráveis e blocos de memória, permitindo a realização de operações básicas (como AND, OR e XOR) e a combinação dessas operações para funções mais complexas. Uma das unidades principais desses elementos lógicos são as ALUTs [2], que são tabelas de consulta adaptáveis usadas para implementar funções lógicas combinacionais. As ALUTs permitem a configuração de várias funções lógicas em uma única célula lógica, aumentando a flexibilidade e eficiência do FPGA (Intel Corporation 2024).

As interconexões conectam esses elementos de diversas formas, formando circuitos digitais personalizados (Bobda e Hartenstein 2007). Alguns FPGAs também incluem blocos de memória embutidos, facilitando o armazenamento e o gerenciamento de dados durante o processamento (Compton, Hauck e Compton 2000). Essa arquitetura torna o FPGA versátil, capaz de implementar desde tarefas simples de controle até algoritmos intensivos em computação.

2.2 Linguagem de Descrição de Hardware

As Linguagens de Descrição de Hardware (HDLs[3]), são linguagens especializadas desenvolvidas para modelar o comportamento e a estrutura de circuitos digitais. Diferentemente das linguagens de programação convencionais, que servem para escrever softwares executados em

[1] Do inglês: *Field-Programmable Gate Array*
[2] Do inglês: *Adaptive Look-Up Tables*
[3] Do inglês: *Hardware Description Languages*

processadores, as HDLs são utilizadas para projetar circuitos integrados como processadores, controladores e interfaces de comunicação. Elas permitem que engenheiros e desenvolvedores especifiquem a lógica do hardware em um nível abstrato, que pode posteriormente ser sintetizado e implementado em dispositivos físicos, como FPGAs ou Circuitos Integrados de Aplicação Específica (ASICs[4]) (Thomas e Moorby 2008).

As linguagens de descrição de hardware mais utilizadas na indústria são o Verilog e o VHDL, amplamente adotadas por sua capacidade de representar circuitos complexos de forma clara e eficiente (Golson e Clark 2016). Essas linguagens permitem a modelagem de módulos de hardware, incluindo operações lógicas, aritméticas, temporização e estruturas de controle, além de possibilitarem a simulação do circuito antes de sua implementação física (Thomas e Moorby 2008). Essa etapa é essencial para o processo de verificação, permitindo que os desenvolvedores testem e validem o design do hardware, reduzindo erros e garantindo a consistência do sistema final.

Além das linguagens tradicionais, surgiram recentemente novas linguagens, como o SpinalHDL, que introduzem um nível de abstração mais alto e facilitam o desenvolvimento de projetos de hardware complexos. Ao permitir o uso de estruturas de controle avançadas e modularidade, essas linguagens modernas ajudam a simplificar a criação de circuitos mais elaborados, favorecendo a produtividade e a manutenção do design ao longo do tempo (Allam et al. 2024).

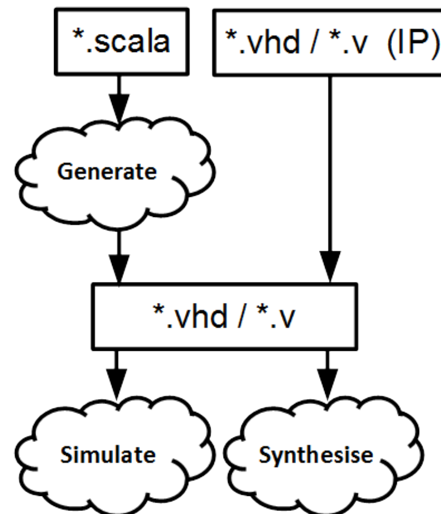
2.3 SpinalHDL

O SpinalHDL é uma linguagem de descrição de hardware moderna, projetada para facilitar o desenvolvimento de circuitos digitais. Baseada em Scala, uma linguagem de programação funcional e orientada a objetos, o SpinalHDL oferece uma abordagem mais abstrata para o design de hardware em comparação com HDLs tradicionais, como Verilog e VHDL. Essa abstração permite que os desenvolvedores expressem arquiteturas de maneira mais intuitiva, reutilizando componentes e estruturas de forma modular e eficiente, o que resulta em um processo de design mais ágil e menos propenso a erros (SpinalHDL 2024).

Uma das características mais notáveis do SpinalHDL é sua capacidade de gerar código em Verilog ou VHDL automaticamente a partir de descrições de alto nível, conforme apresentado na Figura 1. Isso facilita a integração com ferramentas tradicionais de síntese e simulação, permitindo que os desenvolvedores aproveitem as vantagens da programação em um nível superior sem perder compatibilidade com as ferramentas e fluxos de trabalho convencionais de desenvolvimento de hardware. Além disso, o SpinalHDL possibilita o uso de estruturas de controle, como loops e condicionais, de forma muito mais fluida e intuitiva, o que simplifica a criação de circuitos complexos, como processadores e sistemas embarcados (SpinalHDL 2024).

[4] Do inglês: *Application-Specific Integrated Circuits*

Figura 1 – Work Flow de desenvolvimento SpinalHDL



Fonte:(SpinalHDL 2024).

O SpinalHDL é amplamente utilizado na comunidade de desenvolvedores que trabalham com arquiteturas RISC-V, especialmente em projetos que buscam escalabilidade, como o processador VexRiscv. Por oferecer uma curva de aprendizado suave para aqueles familiarizados com linguagens de programação, o SpinalHDL tem atraído uma base crescente de usuários entre profissionais e pesquisadores de hardware (Ahmadi-Pour, Herdt e Drechsler 2022).

2.4 Arquitetura Geral de Processadores

2.4.1 Arquitetura do conjunto de instruções

A Arquitetura do Conjunto de Instruções (ISA [5]), é o componente fundamental de um processador que define o conjunto de operações que ele pode executar, as instruções que ele pode entender e como essas instruções são formatadas (Flynn 1995). Em outras palavras, a ISA especifica a interface entre o software e o hardware de um sistema computacional, fornecendo as regras e o formato para os comandos que o processador irá executar. Essa arquitetura determina o tipo de dados manipulados pelo processador, os tipos de operações que ele pode realizar (como soma, subtração, manipulação de memória), o conjunto de registradores disponíveis e os modos de endereçamento para acessar a memória.

Existem diferentes tipos de ISA, sendo as mais comuns as arquiteturas *CISC* [6] e *RISC* [7]. No modelo *CISC*, o processador possui um conjunto de instruções mais complexo, capaz de realizar operações mais sofisticadas em uma única instrução, o que pode reduzir o número de instruções necessárias para completar uma tarefa (Masood 2011). Já na arquitetura *RISC*, o

[5] Do inglês: *Instruction Set Architecture*
[6] Do inglês: *Complex Instruction Set Computer*
[7] Do inglês: *Reduced Instruction Set Computer*

conjunto de instruções é simplificado, com o objetivo de realizar operações mais rápidas e eficientes, aproveitando melhor o paralelismo e o desempenho do hardware.

A compreensão da ISA é crucial para o desenvolvimento de sistemas embarcados, processadores customizados e a programação de baixo nível, pois ela estabelece as bases para a comunicação entre o software e o hardware (Flynn 1995) . Em sistemas modernos, a escolha e implementação de uma ISA adequada é fundamental para otimizar o desempenho e a eficiência energética.

2.4.2 Estruturas e estágios de Pipeline

Um pipeline em arquitetura de processadores é uma técnica de design que permite a execução simultânea de múltiplas instruções em diferentes estágios em um processador. Isso é feito dividindo a execução de uma instrução em várias etapas, de modo que, enquanto uma instrução está sendo executada em um estágio, outra pode ser processada no estágio seguinte, aumentando assim a eficiência e o desempenho do processador. O pipeline melhora a taxa de transferência [8] e a utilização do processador, permitindo que várias instruções sejam processadas em paralelo.

No contexto do RISC-V, o pipeline típico é dividido em cinco estágios principais:

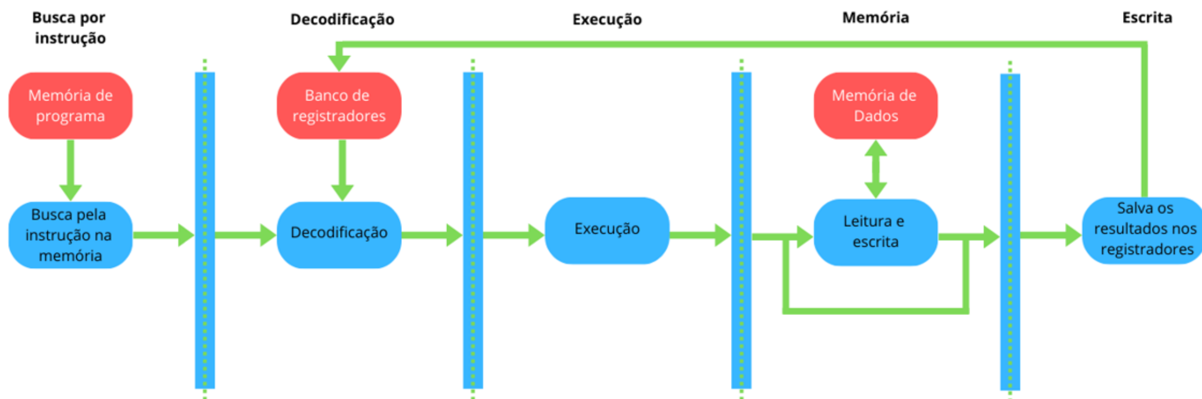
- *Busca*: A instrução é buscada na memória de programa.
- *Decodificação*: A instrução é decodificada, e os operandos armazenados no registrador de instruções são recuperados.
- *Execução*: A operação especificada pela instrução é executada, que pode incluir operações aritméticas ou o cálculo de endereços.
- *Memória*: Caso a instrução envolva acesso à memória, como uma operação de leitura ou escrita, isso ocorrerá neste estágio.
- *Reescrita*: O resultado da execução é armazenado no registrador de destino.

Esses cinco estágios permitem que o processador execute múltiplas instruções simultaneamente, otimizando o fluxo de dados e reduzindo o tempo total de execução. A Figura 2 ilustra a organização do pipeline em um processador compatível com a arquitetura RISC-V.

Embora um pipeline tradicional de cinco estágios seja comum em muitos processadores RISC-V, não é necessário que todos os processadores sigam esse modelo de cinco estágios completos. Um processador pode operar com dois ou mais estágios de pipeline, conforme seu design e os requisitos específicos da aplicação.

[8] Do inglês: *throughput*

Figura 2 – Os Cinco Estágios de Pipeline RISC-V Baseados na Arquitetura Harvard



Fonte: Adaptado de (Verma e Stan 2022).

A principal razão para isso é a simplicidade e otimização do projeto. Em sistemas de baixo custo, como microcontroladores ou em sistemas embarcados que não exigem alta taxa de processamento, um pipeline mais simples pode ser suficiente. Nesse caso, o processador pode ter apenas os estágios essenciais, como o *busca* e *execução*, sem a necessidade de estágios adicionais como *decodificação*, *memória* ou *reescrita*. Isso pode reduzir a complexidade do hardware e melhorar o consumo de energia.

Além disso, algumas arquiteturas ou aplicações podem não necessitar de todos os estágios para alcançar o desempenho desejado. Processadores com menos estágios podem ser adequados para tarefas específicas que não envolvem operações de leitura e escrita de memória frequentemente ou operações aritméticas complexas, o que permite a simplificação do pipeline sem sacrificar o desempenho geral. Portanto, a escolha do número de estágios depende das necessidades do sistema e da carga de trabalho esperada.

2.4.3 Bypass

O *bypass* é um recurso amplamente estudado e utilizado em diversos processadores, pode ser conhecido como *Forwarding* ou *Data Forwarding*. Este recurso é disponibilizado na estrutura VexRiscv de forma simplificada e permite que os dados produzidos em um estágio do pipeline sejam utilizados imediatamente em um estado subsequente, sem a necessidade de esperar que esses dados sejam escritos de volta nos registradores. Isso é especialmente útil para se evitar *stalls* (paradas) no pipeline, que ocorrem quando uma instrução depende do resultado da instrução anterior que ainda não foi concluída. Com menos ciclos perdidos devido a esperas, o *throughput* do processador tende a aumentar.

Seu funcionamento se baseia na detecção de dependências de dados, durante a fase de decodificação, na qual o processador verifica se a instrução atual depende do resultado de uma instrução anterior que ainda não foi escrita no banco de registradores. Se uma dependência

for detectada, no encaminhamento dos dados, o processador utiliza a saída da unidade de execução da instrução anterior como entrada para a unidade de execução da instrução atual, evitando assim a necessidade de esperar pela fase de escrita ao registrador de destino.

O uso do *bypass* proporciona uma significativa melhoria no desempenho do processador, pois reduz a latência associada às esperas por dados. Ao eliminar a necessidade de aguardar a escrita nos registradores, o processador consegue manter um fluxo contínuo de execução, o que é especialmente vantajoso em sistemas de alta performance ou em aplicações que exigem processamento rápido. Além disso, o *bypass* contribui para a eficiência energética, já que diminui a quantidade de ciclos de clock desperdiçados, resultando em um melhor aproveitamento dos recursos do sistema.

2.4.4 Early Branch

Conhecido comumente como *Branch Prediction*, o *Early Branch* é uma técnica universal amplamente utilizada para otimizar o desempenho do pipeline ao lidar com instruções de desvio (*branch*). Em processadores convencionais, quando uma instrução de desvio é encontrada, o pipeline precisa esperar até que a condição do desvio seja avaliada na fase de execução para determinar qual instrução buscar a seguir. Esse tempo de espera cria "bolhas"no pipeline, reduzindo a eficiência. O *Early Branch* resolve esse problema ao permitir que o processador comece a buscar e decodificar instruções subsequentes antes mesmo que a decisão sobre o desvio seja finalizada, reduzindo os ciclos de espera e melhorando o *throughput* (Patterson e Hennessy 2016).

O funcionamento do *Early Branch* baseia-se em mecanismos de predição de desvio, que tentam antecipar se o desvio será tomado ou não. Essa predição ocorre antes que a condição do desvio seja avaliada na fase de execução. Se a predição estiver correta, as instruções subsequentes já estarão em processo de busca e decodificação, evitando interrupções no fluxo do pipeline. Caso a predição esteja errada, o processador precisa descartar as instruções incorretas que foram buscadas, o que gera um pequeno custo adicional em ciclos de clock. No entanto, quando a predição é acertada com frequência, o ganho de desempenho é significativo.

A principal vantagem do *Early Branch* é a redução da latência no pipeline, especialmente em arquiteturas RISC-V, onde as instruções são frequentemente encadeadas e dependências podem causar paradas (*stalls*). Ao manter o pipeline cheio e minimizar os ciclos ociosos, o *Early Branch* aumenta o desempenho geral do processador. No entanto, sua implementação requer circuitos adicionais para gerenciar a predição e o fluxo de instruções, o que pode aumentar a complexidade do design e o consumo de energia. Além disso, predições incorretas podem levar à execução de instruções desnecessárias, impactando temporariamente a eficiência. Apesar desses desafios, o *Early Branch* é uma técnica valiosa para melhorar o desempenho em processadores modernos, especialmente em aplicações que exigem alta eficiência e baixa latência.

2.5 RISC-V

RISC-V é uma arquitetura de conjunto de instruções aberta e livremente licenciada, projetada com base nos princípios do design RISC. Desenvolvido inicialmente pela Universidade da Califórnia, em Berkeley, o RISC-V foi criado com o objetivo de proporcionar uma arquitetura de instruções modular, que atendesse às necessidades modernas da computação em diversos contextos, desde sistemas embarcados até supercomputadores. Ao contrário de outras ISAs amplamente utilizadas, como x86 e ARM, o RISC-V permite que qualquer pessoa o implemente ou modifique, sem custos de licenciamento, o que o torna uma escolha atraente tanto para a academia quanto para a indústria (Patterson e Waterman 2017).

A modularidade do RISC-V permite a personalização do conjunto de instruções, ajustando-o para as demandas específicas de cada aplicação. Isso significa que o RISC-V pode ser implementado em diferentes configurações, desde microcontroladores de baixa potência até sistemas de alto desempenho (Chisnall 2024). Essa flexibilidade, combinada com uma comunidade de desenvolvedores global e crescente, tornou o RISC-V uma alternativa promissora no mercado de processadores, especialmente em tempos em que as limitações de licenciamento e os altos custos das arquiteturas proprietárias representam desafios significativos para a inovação (Patterson e Waterman 2017).

O RISC-V também se destaca pela sua arquitetura orientada à eficiência e otimização de recursos, que possibilita tanto a implementação de processadores econômicos em termos de consumo de energia quanto soluções de alto desempenho. A simplicidade do conjunto de instruções facilita o desenvolvimento de implementações eficientes, permitindo o uso otimizado de recursos em dispositivos com restrições de área e consumo (Bora e Paily 2021). Por essas razões, o RISC-V está rapidamente se consolidando como uma tecnologia relevante para pesquisa e desenvolvimento, oferecendo uma plataforma sólida para inovações em arquitetura de computadores e sistemas embarcados (Chisnall 2024).

2.6 VexRISCV

O VexRiscv é um núcleo de processador baseado na arquitetura RISC-V, projetado para ser altamente configurável e eficiente, especialmente em ambientes de sistemas embarcados e FPGAs. Desenvolvido em SpinalHDL, o VexRiscv tira proveito da flexibilidade e da abstração fornecidas por essa linguagem para oferecer um design modular e escalável, que permite aos desenvolvedores ajustar o processador de acordo com requisitos específicos de desempenho, área e consumo de energia. O uso do SpinalHDL facilita a personalização e a reutilização de componentes internos, o que resulta em um processo de design mais ágil e adaptável às necessidades dos projetos, desde sistemas de baixo custo até aplicações mais complexas que exigem uma arquitetura de processamento robusta (Papon 2020).

No VexRiscv, a implementação do conjunto de instruções RV32I do RISC-V é acom-

panhada de opções de expansão que podem incluir unidades de multiplicação, divisão e até mesmo suporte a instruções de ponto flutuante, dependendo da configuração. Essa flexibilidade é fundamental para aplicações que variam de simples controladores a sistemas de alta performance, possibilitando um equilíbrio preciso entre complexidade e eficiência. A modularidade do SpinalHDL permite que o núcleo do VexRiscv seja facilmente modificado ou adaptado, seja para adicionar novos periféricos ou para configurar cache, interconexões e unidades de processamento específicas de acordo com a aplicação alvo (Efinix 2024).

A utilização do SpinalHDL no desenvolvimento do VexRiscv também torna o processo de verificação e validação mais eficiente. Como o SpinalHDL permite a geração de código Verilog ou VHDL a partir de descrições de alto nível, o núcleo VexRiscv pode ser testado e sintetizado em diferentes ferramentas e FPGAs, como Quartus e Vivado, de forma simplificada (Papon 2020). Além disso, a estrutura de controle e as abstrações disponíveis no SpinalHDL facilitam a implementação de funcionalidades complexas de maneira organizada, o que reduz a quantidade de código e minimiza potenciais erros de projeto (SpinalHDL 2024).

A comunidade de desenvolvedores do VexRiscv tem utilizado esse núcleo como base para projetos educacionais, protótipos e até mesmo aplicações comerciais, dada sua adaptabilidade e o suporte oferecido pela documentação e pela própria comunidade (Hacker News 2024). A combinação do VexRiscv com o SpinalHDL demonstra como uma linguagem de descrição de hardware moderna pode transformar o processo de design, tornando a criação de processadores uma tarefa mais acessível e prática (Efinix 2024).

2.6.1 Microcontrolador Murax

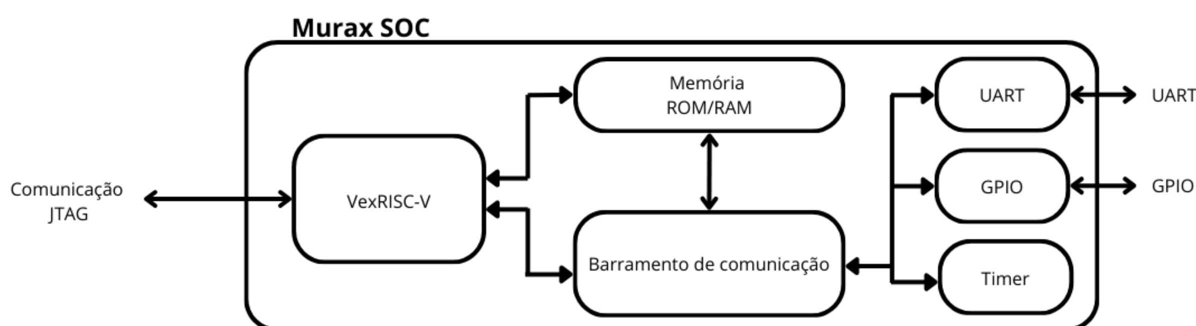
O Murax é um exemplo representativo da arquitetura VexRiscv, desenvolvido com o propósito de demonstrar a flexibilidade e a personalização possíveis em implementações baseadas na arquitetura RISC-V. Esse microcontrolador foi projetado para ser simples e funcional, proporcionando uma introdução acessível aos conceitos de design de microcontroladores e tornando-se uma ferramenta didática para quem deseja compreender o seu funcionamento e o desenvolvimento. Ideal para aplicações que não demandam alto desempenho e de baixa complexidade, o Murax permite explorar as possibilidades da arquitetura RISC-V em FPGAs de pequeno e médio porte, sendo facilmente adaptável e expansível.

Um dos principais atrativos do Murax é sua compatibilidade com ferramentas populares de desenvolvimento de hardware, como o Quartus e o Vivado, amplamente utilizadas no processo de síntese de circuitos. Além disso, ele é compatível com o Yosys, uma ferramenta de código aberto que facilita as simulações e análises, tornando o fluxo de teste e validação mais acessível a pesquisadores e desenvolvedores. O núcleo do Murax é o processador VexRiscv, uma CPU de 32 bits implementada em SpinalHDL, o que possibilita um alto nível de personalização e eficiência. Essa arquitetura permite o desenvolvimento de variantes personalizadas do núcleo, equilibrando desempenho e utilização de recursos, tornando o Murax uma plataforma atrativa

para sistemas embarcados que requerem um design compacto e eficiente.

Popular entre a comunidade de desenvolvedores, o Murax é frequentemente utilizado como ponto de partida para pesquisa e desenvolvimento em FPGAs e processadores RISC-V. Sua documentação disponível no GitHub e o suporte em fóruns especializados tornam essa plataforma acessível tanto para iniciantes quanto para profissionais experientes. O Murax suporta o conjunto de instruções RV32I, um subconjunto básico do ISA RISC-V, que permite a execução de tarefas comuns de microcontroladores, como manipulação de dados e controle de dispositivos externos. Este conjunto de instruções reduzido mantém o design do Murax simples e otimizado para a síntese em FPGAs de baixo custo, tornando-o uma solução prática e eficiente para aplicações educacionais e de pesquisa.

Figura 3 – Arquitetura do SOC Murax



Fonte: Adaptado de (SpinalHDL 2024).

O Murax foi projetado como um sistema completo e auto-suficiente, que inclui um conjunto de periféricos básicos. Esse conjunto inclui controladores e interfaces comuns para sistemas embarcados, como GPIOs [9], controladores de UART [10] e uma timer para controle de tempo e interrupções de sistema, como pode ser visto na Figura 3, o que permite que o Murax seja aplicado em projetos de controle e monitoramento. Seu design econômico utiliza uma quantidade reduzida de recursos de LUTs e FFs [11], o que o torna adequado para FPGAs de menor porte. Além disso, o Murax conta com um controlador de memória que permite o uso de uma RAM interna ou externa, dependendo das necessidades de cada aplicação.

2.7 Dhrystone

A forma encontrada para testar a eficacia dessas técnicas e como isso afeta o pipeline foi por meio do Dhrystone, um *benchmark* sintético desenvolvido em 1984 por Reinhold P. Weicker, com o objetivo de medir o desempenho de sistemas computacionais, especialmente em relação à programação de inteiros(York 2002). Este programa se tornou uma referência

[9] Do inglês: *General Purpose Input/Output*
[10] Do inglês: *Universal Asynchronous Receiver-Transmitter*
[11] Do inglês: *Flip-Flops*

para avaliar a performance de SoCs e compiladores, sendo amplamente utilizado na indústria para realizar comparações de desempenho. A escolha do nome "Dhrystone"é um trocadilho com o benchmark Whetstone, que foca na performance de ponto flutuante, destacando assim a especialização do Dhrystone em operações inteiras (Weicker 1984).

O funcionamento do Dhrystone se baseia na execução de um conjunto de operações que imitam o uso típico de programas em linguagens como FORTRAN, PL/1 e Pascal. A métrica gerada pelo benchmark é expressa em "Dhrystones por segundo", que representa o número de iterações do código principal executadas em um segundo. Essa abordagem permite que os resultados sejam facilmente interpretados e comparados entre diferentes sistemas, embora a simplicidade do programa também levante questões sobre sua representatividade em cenários práticos mais complexos (Weicker 1984).

Apesar de sua popularidade inicial, o Dhrystone tem limitações significativas. Ele não inclui operações de ponto flutuante e pode ser suscetível a otimizações de compiladores que distorcem os resultados reais. Por exemplo, como as strings utilizadas no benchmark têm tamanhos constantes e são alinhadas em limites naturais, os compiladores podem otimizar a cópia dessas strings de maneiras que não ocorrem em programas reais, levando a uma superestimação do desempenho do sistema (York 2002).

3 Desenvolvimento

O microcontrolador Murax será o foco da nossa análise, pois ele permite a configuração de dois a quatro estágios de pipeline sem alterações em sua estrutura externa, como os periféricos. Dentre os aspectos analisados estão as alterações no processador, como a ativação ou desativação do *bypass*, do *early branch*, a variação no número de estágios e a adição de instruções de multiplicação e divisão. No entanto, não é possível gerar um microcontrolador Murax com apenas dois estágios sem a presença do *early branch*, pois, nesse caso, a resolução antecipada de desvios é essencial para evitar *stalls* prolongados no pipeline. Sem essa otimização, a falta de um estágio intermediário para tratar mudanças no fluxo de execução resultaria em penalidades significativas no desempenho.

Para avaliar o impacto dessas configurações, utilizaremos o benchmark Dhrystone com 200 iterações, tendo como métrica principal os Dhrystones por segundo, medindo a eficiência do microcontrolador em termos de operações executadas.

O microcontrolador Murax utiliza uma arquitetura Harvard, o que significa que ele possui caminhos separados para instruções e dados, permitindo uma maior eficiência no acesso à memória. Nesse microcontrolador, os dois estágios principais, busca (*fetch*) e decodificação (*decode*), são combinados em um único estágio. Essa fusão ocorre devido à simplicidade da arquitetura do Murax, o que reduz a complexidade do design e facilita a personalização. Com isso, a busca da instrução e a decodificação ocorrem de forma sequencial e compartilhada, sem a necessidade de um estágio dedicado para cada uma dessas operações.

Quando o processador é configurado para operar com apenas dois estágios, as funções de execução, memória e escrita no registrador ocorrem de forma assíncrona, ou seja, elas não são sequenciadas diretamente no pipeline. Ao invés disso, essas operações são realizadas conforme os dados estão disponíveis, sem a restrição de um ciclo de clock específico para cada operação. Esse comportamento assíncrono permite que as instruções sejam processadas de maneira mais flexível, o que pode ser vantajoso em sistemas simples que não exigem altos requisitos de desempenho.

Nos casos em que o processador tem estágios adicionais, como os de memória (*Memory*) e escrita no registrador (*WriteBack*), eles podem ser configurados para otimizar o desempenho de acordo com as necessidades da aplicação, conforme ilustrado no exemplo da Figura 4. Assim, o Murax permite uma customização eficiente, balanceando entre simplicidade, desempenho e recursos de acordo com o contexto do projeto.

Figura 4 – Estrutura Geral de Configuração de Estágios

```
1 val cpu = new VexRiscv(
2     config = VexRiscvConfig(
3         withMemoryStage = true,
4         withWriteBackStage = true,
5         plugins = cpuPlugins += new DebugPlugin(
6             debugClockDomain, hardwareBreakpointCount)
7     )
8 )
```

Fonte: Elaborado pelo autor.[11]

De maneira similar, o código apresentado na Figura 5 ilustra a implementação do *bypass* e do *early branch* na geração do microcontrolador. Essas otimizações são ajustadas conforme os requisitos de desempenho e eficiência energética, proporcionando um controle refinado sobre o funcionamento do pipeline e permitindo um balanceamento entre velocidade de execução e utilização de recursos.

Figura 5 – Estrutura Geral de Configuração de *Bypass* e *Early Branch*

```
1    new HazardSimplePlugin(
2      bypassExecute = bypass,
3      bypassMemory = bypass,
4      bypassWriteBack = bypass,
5      bypassWriteBackBuffer = bypass,
6      pessimisticUseSrc = false,
7      pessimisticWriteRegFile = false,
8      pessimisticAddressMatch = false
9    ),
10   new BranchPlugin(
11     earlyBranch = earlyBranch,
12     catchAddressMisaligned = false
13   )
```

Fonte: Elaborado pelo autor.[11]

Essas configurações permitem uma análise detalhada do impacto de cada otimização e do número de estágios no desempenho e no consumo energético do microcontrolador. Ao variar essas configurações, é possível identificar o equilíbrio ideal entre desempenho e eficiência, adaptando o microcontrolador às necessidades específicas de cada aplicação.

3.1 Resultados e Análises

Esta seção apresenta os resultados dos experimentos realizados para avaliar o impacto do número de estágios do pipeline e das otimizações de *bypass* e *early branch* no desempenho, consumo de energia e uso de recursos lógicos. Os resultados são organizados em diferentes métricas e comparações para identificar as configurações mais eficientes.

[11] Disponível em: ⟨https://github.com/thiago0003/TCC_RISC_V/blob/main/src/main/scala/vexriscv/demo/Murax.scala⟩.

A Tabela 1 abaixo resume os principais dados coletados, incluindo a frequência máxima de operação, a utilização de elementos lógicos, o número de registradores, o desempenho medido em Dhrystones por segundo e o consumo total de energia. As configurações analisadas variam conforme o número de estágios de pipeline e a ativação ou desativação de otimizações como o *bypass* e o *early branch*.

Para a geração dos dados, foi utilizada a placa Intel DE-10 Standard, com o auxílio da ferramenta Quartus II para a síntese e análise da utilização de recursos. O consumo de energia foi estimado a partir de um testbench desenvolvido com a ferramenta Icarus Verilog, que gerou um arquivo VCD contendo a atividade dos sinais durante a execução do programa Dhrystone. Esse arquivo foi então processado pelo Quartus Power, permitindo a análise detalhada do consumo energético do microcontrolador em diferentes configurações.

Os nomes dos microcontroladores seguem uma convenção específica para indicar suas características. A base do processador é representada por RV32I ou RV32IM, onde RV32I se refere a um processador RISC-V de 32 bits com instruções de inteiros, e RV32IM indica a inclusão de multiplicação em hardware. Em seguida, as siglas B e E são utilizadas para indicar a ativação das otimizações de *bypass* e *early branch*, respectivamente. Por exemplo, um microcontrolador identificado como RV32I3BE corresponde a um núcleo RISC-V de 32 bits com três estágios de pipeline, instruções de inteiros e as otimizações de *bypass* e *early branch* ativadas. Essa nomenclatura facilita a identificação das características de cada configuração.

Tabela 1 – Registradores - Murax(I) (MHz)

Microcontrolador	Frequência Máxima (MHz)	Elementos lógicos	Registradores	Dhrystones	Consumo Total (mW)
RV32I2E	80.95	1130	911	6.65	252.77
RV32I2BE	76.75	1241	911	7.89	253.99
RV32I3	84.28	1136	959	5.94	253.64
RV32I3E	81.95	1129	928	6.43	252.50
RV32I3B	74.90	1225	992	7.74	253.54
RV32I3BE	73.30	1211	961	8.40	253.14
RV32I4	70.40	1112	1040	5.46	252.92
RV32I4E	78.75	1120	1009	5.88	253.18
RV32I4B	80.73	1315	1075	7.59	253.97
RV32I4BE	81.06	1289	1044	8.21	253.65
RV32IM4	75.65	1495	1394	6.42	253.64
RV32IM4E	79.71	1501	1363	7.08	256.23
RV32IM4B	75.06	1740	1429	9.09	257.75
RV32IM4BE	76.70	1705	1398	9.97	258.13

Fonte: Elaborado pelo autor.

3.1.1 Frequência máxima

A frequência máxima de operação de um microcontrolador é um fator crítico que influencia diretamente o desempenho e a eficiência energética. Pipelines mais longos tendem a apresentar maior impacto com otimizações, já que há mais estágios para gerenciar dependências de dados.

Em pipelines mais curtos, como os de dois estágios, o impacto do *early branch* é menos significativo, pois a penalidade de ramificação já é naturalmente menor. Isso ocorre porque, em pipelines curtos, o tempo necessário para resolver uma instrução de desvio é reduzido, minimizando os ciclos de espera. No entanto, mesmo nesses casos, a ativação do *early branch* pode trazer benefícios ao evitar paradas (*stalls*) no pipeline, especialmente em cenários onde há uma alta frequência de instruções de desvio. A simplicidade do pipeline de dois estágios permite que o microcontrolador opere em frequências mais altas, mas com desempenho limitado em operações complexas.

Por outro lado, em pipelines mais longos, como os de quatro estágios, a capacidade de resolver ramificações antecipadamente é crucial para evitar desperdícios de ciclos e manter a eficiência. O *early branch* permite que o microcontrolador comece a buscar e decodificar instruções subsequentes antes mesmo que a decisão sobre o desvio seja finalizada, reduzindo os ciclos de espera e melhorando o *throughput*. Isso é especialmente importante em pipelines longos, onde a latência entre a busca e a execução de uma instrução de desvio pode ser significativa. A Tabela 1 mostra que, em processadores de quatro estágios, a ativação do *early branch* resulta em um aumento na frequência máxima de clock, permitindo que o microcontrolador opere em velocidades mais altas sem comprometer a eficácia do processamento.

Desativar o *bypass* para operações de multiplicação em hardware resultou em um aumento na frequência máxima, conforme mostrado na Tabela 1. Embora o *bypass* reduza a necessidade de espera entre instruções dependentes, ele adiciona circuitos extras no pipeline, aumentando a complexidade do caminho crítico e limitando a frequência máxima. Essa redução na frequência é causada pela lógica adicional necessária para encaminhar os dados entre os estágios do pipeline, o que pode aumentar a latência e limitar o desempenho. Ao desativar o *bypass*, o pipeline se simplifica, permitindo que as operações de multiplicação sejam processadas diretamente, sem depender da lógica de encaminhamento. Como consequência, o microcontrolador pode operar com uma frequência mais alta, resultando em um melhor aproveitamento dos recursos e maior eficiência na execução.

3.1.2 Elementos lógicos

Para os microcontroladores sem multiplicação em hardware, quando o *bypass* e o *early branch* estão desativados, a utilização de ALUTs é menor, como pode ser observado na Tabela 1. Isso ocorre porque a ausência dessas funcionalidades reduz a complexidade do circuito, eliminando a lógica adicional necessária para implementar o *bypass* e o *early*

branch. No entanto, ao ativar o *bypass*, a utilização de ALUTs aumenta significativamente. Isso acontece porque o *bypass* requer lógica extra para encaminhar dados entre os estágios do pipeline, o que adiciona circuitos adicionais e aumenta a carga de trabalho sobre as ALUTs.

O *early branch* também contribui para um aumento na utilização de ALUTs, embora em menor proporção. Sua ativação introduz lógica adicional para a previsão e resolução antecipada de desvios, o que resulta em uma complexidade maior do design. Quando combinado com o *bypass*, a utilização de ALUTs é ainda maior, refletindo a complexidade acumulada da implementação dessas funcionalidades.

Além disso, o número de estágios do pipeline também influencia a utilização de ALUTs. Configurações com quatro estágios apresentam uma utilização mais elevado devido aos controles adicionais necessários para a separação das fases de execução. A redução para três estágios tende a diminuir ligeiramente a utilização, pois há menos lógica de controle envolvida. No entanto, para um pipeline de apenas dois estágios, a ativação do *early branch* se torna essencial, o que impede uma redução significativa no uso de ALUTs, já que a lógica de controle para desvios se torna crucial para o funcionamento eficiente do pipeline.

A inclusão da multiplicação no *design* resulta em um aumento significativo na utilização de ALUTs, pois a multiplicação é uma operação complexa que demanda mais recursos lógicos. Esse impacto é ainda maior quando o *bypass* está ativado, já que a lógica adicional necessária para o *bypass* eleva ainda mais a utilizaçõa de ALUTs. Em comparação com configurações sem multiplicação, fica evidente que a complexidade da operação e a ativação do *bypass* contribuem para um uso mais intensivo de recursos.

Entre os modelos testados, a configuração RV32I4BE apresentou o maior utilização de ALUTs, enquanto RV32I2E teve o menor. Isso confirma que pipelines mais complexos exigem mais recursos lógicos devido ao maior número de estágios intermediários e lógica de controle.

3.1.3 Registradores

O número de registradores utilizados em um microcontrolador é diretamente influenciado pela configuração do pipeline, especialmente pelo número de estágios. Em processadores com quatro estágios, a necessidade de registradores é maior, pois cada estágio adicional requer registradores para armazenar dados intermediários e garantir a transferência correta e sincronizada dos dados ao longo do pipeline. Isso ocorre porque, em pipelines mais longos, os dados precisam ser mantidos em registradores por mais tempo, à medida que passam por cada estágio de processamento. Como resultado, microcontroladores com mais estágios exigem um número maior de registradores em comparação com microcontroladores de pipelines mais curtos.

A ativação de funcionalidades como *bypass* e *early branch* também impacta o design, adicionando lógica adicional que pode aumentar ligeiramente o número de registradores necessários Tabela 1. No entanto, esse aumento é pequeno quando comparado ao impacto

causado pelo número de estágios. Em microcontroladores de dois estágios, a lógica é mais simplificada e direta, resultando em um uso mais eficiente dos registradores. Já em pipelines com mais estágios, a necessidade de mais registradores é essencial para garantir a passagem correta dos dados entre os estágios, o que pode reduzir a eficiência em termos de uso de recursos.

A inclusão da multiplicação no microcontrolador aumenta a utilização de registradores, já que essa operação complexa exige mais recursos de armazenamento temporário. Esse aumento é observado em todas as configurações analisadas, refletindo a necessidade de mais registradores para lidar com a complexidade da multiplicação Tabela 1.

Quando o *bypass* está desativado, a utilização de registradores é menor, pois a lógica adicional necessária para encaminhar dados entre os estágios do pipeline não está presente. Por outro lado, a ativação do *bypass* eleva a utilização de registradores, devido à complexidade introduzida pela lógica de *bypass*.

O *early branch* reduz a necessidade de armazenar certas instruções intermediárias, pois antecipa a decisão de desvios, resultando em um uso ligeiramente menor de registradores. Mesmo quando combinado com o *bypass*, o *early branch* ajuda a mitigar o impacto na utilização de registradores, embora o uso de ambos ainda resulte em uma maior utilização comparado às configurações sem *bypass*.

3.1.4 Métricas Dhrystone

Para avaliar o desempenho dos microcontroladores, utilizamos o benchmark Dhrystone, que mede a eficiência na execução de operações inteiras. O resultado, expresso em Dhrystones por segundo, indica a capacidade de processamento da CPU e permite comparar diferentes arquiteturas e pipelines.

O *bypass* melhora significativamente o desempenho ao reduzir o tempo de espera entre instruções dependentes. Sem essa técnica, uma instrução precisa aguardar a escrita e posterior leitura do resultado no banco de registradores, adicionando ciclos extras ao pipeline. Com o *bypass*, os resultados são encaminhados diretamente do estágio de execução para a instrução seguinte, eliminando a espera e aumentando a taxa de execução. Os resultados mostram que a ativação do *bypass* aumentou o número de *Dhrystones* por segundo em todas as configurações testadas, com ganhos mais expressivos nos microcontroladores com três e quatro estágios de pipeline.

O *early branch* minimiza o impacto de desvios condicionais antecipando a decisão sobre a próxima instrução. Sem essa otimização, o microcontrolador precisa aguardar a avaliação da condição antes de continuar, causando ciclos ociosos (stalls). Com o *early branch*, essa decisão ocorre mais cedo, permitindo a execução sem atrasos desnecessários. Isso melhora a eficiência em programas com alta frequência de desvios, como os do benchmark *Dhrystone*, que contém diversas operações de controle de fluxo.

A análise dos resultados revelou que a combinação de *bypass* e *early branch* proporcionou o maior ganho de desempenho, otimizando tanto a execução sequencial de operações quanto a gestão de desvios condicionais. Os microcontroladores de três estágios apresentaram o melhor equilíbrio entre consumo de energia e desempenho, enquanto aqueles com quatro estágios atingiram o maior número absoluto de Dhrystones por segundo, embora com um aumento na utilização de recursos lógicos.

3.1.5 Consumo de Energia

Os FPGAs, em geral, apresentam um consumo de potência significativamente maior do que os ASICs implementados com a mesma tecnologia. A potência estática, que é o consumo necessário para manter o dispositivo ligado, pode variar conforme a temperatura de operação. Já a potência dinâmica, que é o consumo requerido para realizar mudanças de estado no dispositivo, varia diretamente com a frequência de operação e também é influenciada pela temperatura. Na Tabela 1, é apresentado o consumo de potência dinâmico estimado em miliwatts utilizando a ferramenta Intel Quartus Power, considerando uma temperatura constante.

A análise dos dados revela que o consumo de energia varia de acordo com o número de estágios do pipeline e as otimizações de *bypass* e *early branch*. Sem essas otimizações, o consumo de energia é razoavelmente consistente. A ativação do *early branch* reduz levemente o consumo, sendo que os microcontroladores de três estágios apresentam o menor consumo. Por outro lado, ativar o *bypass* aumenta ligeiramente o consumo, devido à lógica adicional necessária para um fluxo de dados mais eficiente. A combinação de *bypass* e *early branch* resulta em um consumo que reflete a sobrecarga energética causada pela lógica extra dessas otimizações.

Para microcontroladores de quatro estágios, o consumo é ligeiramente maior quando as otimizações estão ativas. Em três estágios, o consumo varia, com algumas configurações apresentando menor consumo, especialmente com o *early branch* ativado. Já para dois estágios, os dados limitados mostram que a simplicidade do pipeline pode resultar em maior eficiência energética em certas configurações. Em geral, desativar o *bypass* e o *early branch* tende a consumir menos energia, enquanto ativá-los adiciona uma sobrecarga energética, embora proporcione melhorias no desempenho. Portanto, ao decidir quais otimizações implementar, é crucial considerar o equilíbrio entre desempenho e consumo de energia, adaptando as configurações às necessidades específicas da aplicação.

A análise dos dados da Tabela 1 revela que o consumo de energia aumenta conforme as otimizações de *bypass* e *early branch* são ativadas. Sem essas otimizações, o consumo de energia é menor, representando a base de comparação. A ativação do *early branch* eleva o consumo devido à complexidade adicional na predição de ramificações, enquanto o *bypass* aumenta o consumo, pois facilita o fluxo de dados no pipeline. A combinação de ambas

as otimizações resulta no maior consumo de energia, devido à maior quantidade de lógica necessária para gerenciar essas funcionalidades.

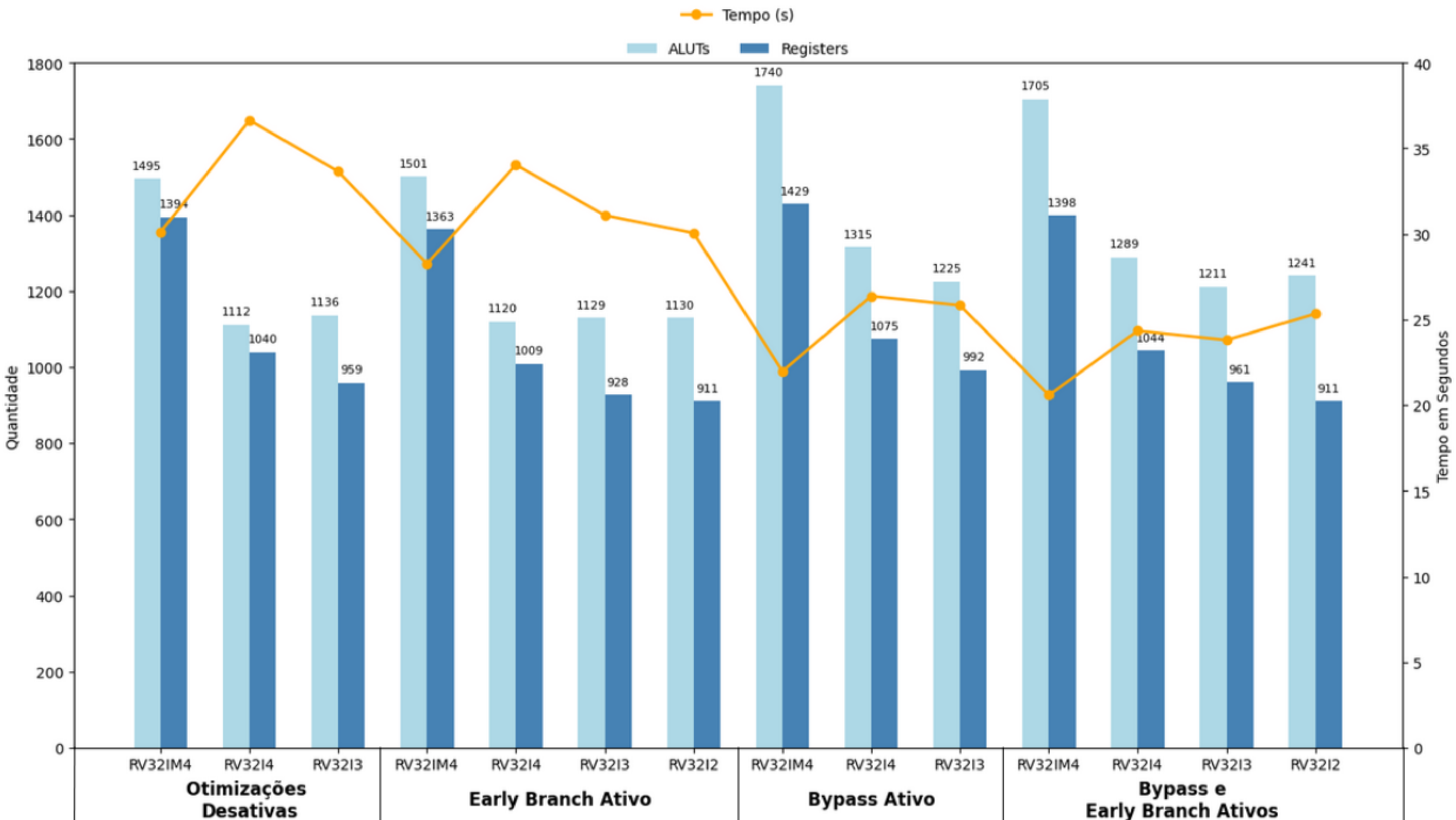
A inclusão da multiplicação, uma operação complexa, contribui para um aumento geral no consumo de energia em todas as configurações. Embora as otimizações de *bypass* e *early branch* melhorem o desempenho, elas também elevam o consumo de energia, como observado nos dados. Ativar essas otimizações pode ser essencial para alcançar o desempenho desejado em operações complexas, mas é importante considerar o trade-off entre eficiência energética e performance, especialmente em aplicações que exigem um equilíbrio entre esses fatores.

3.1.6 Análise comparativa

A análise do gráfico da Figura 6, que inclui dados de microcontroladores com e sem multiplicação em hardware, permite uma comparação detalhada entre a utilização de recursos e o desempenho em diferentes configurações de pipeline e otimizações, como o *bypass* e o *early branch*

Microcontroladores com mais estágios tendem a oferecer menor tempo de execução, mas exigem mais elementos lógicos (ALUTs e registradores), aumentando o custo de hardware. Por outro lado, arquiteturas com menos estágios são mais compactas e eficientes em termos de recursos, mas podem sofrer penalizações de desempenho devido a *stalls* e maior tempo de execução. Os resultados obtidos mostram que microcontroladores com dois estágios são mais simples e consomem menos recursos, tornando-se adequados para aplicações com restrição de área ou consumo de energia, onde um tempo de execução ligeiramente maior pode ser aceitável. Microcontroladores com três estágios apresentam um equilíbrio interessante, conseguindo reduzir significativamente o tempo de execução sem um aumento expressivo na utilização de hardware. Arquiteturas de quatro estágios atingem o melhor desempenho, porém com um custo elevado em termos de utilização de recursos, o que pode ser inviável para sistemas embarcados ou FPGAs com capacidade limitada.

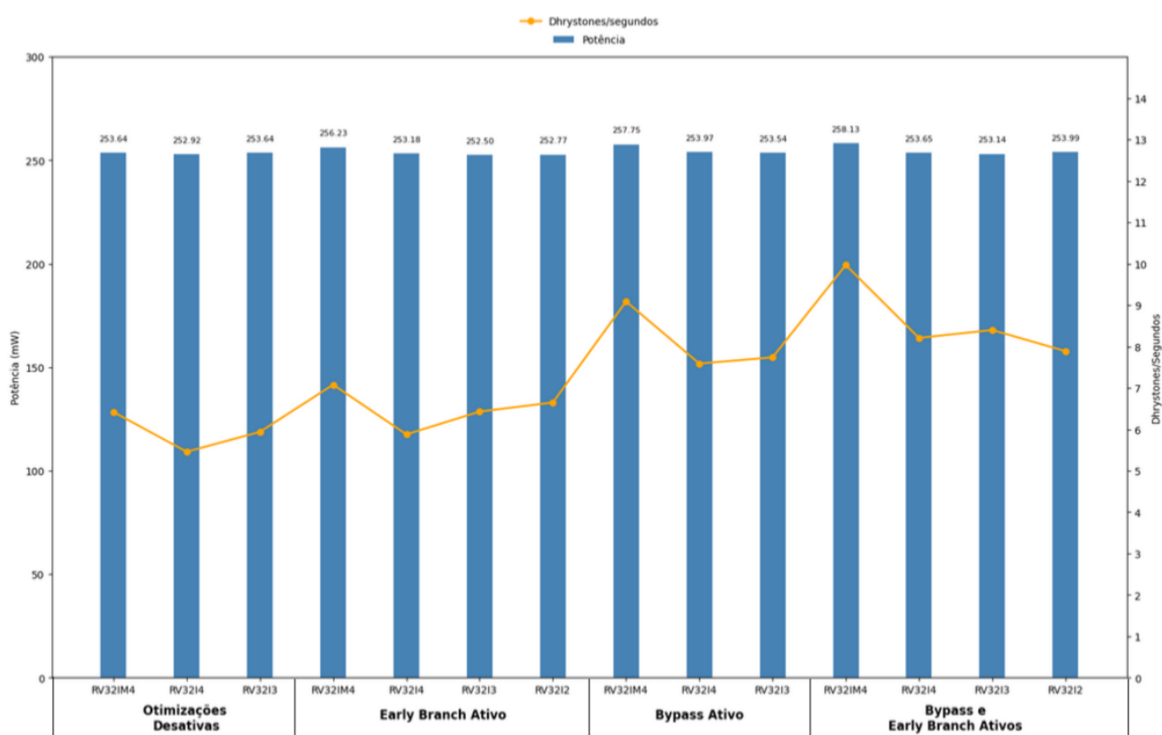
Figura 6 – Comparativo microcontroladores



Fonte: Elaborado pelo autor.

Microcontroladores sem multiplicação em hardware apresentam um consumo de energia que varia conforme o número de estágios e as otimizações aplicadas. Microcontroladores com quatro estágios consomem mais energia, especialmente com o *bypass* ativado, devido à maior complexidade do pipeline. Já microcontroladores com três estágios mostram um consumo intermediário, enquanto os de dois estágios consomem menos energia, mas com limitações de desempenho. Por outro lado, microcontroladores com multiplicação em hardware exigem um aumento significativo no consumo de energia, especialmente em configurações de quatro estágios, devido à complexidade da operação. A ativação do *bypass* e do *early branch* aumenta ainda mais o consumo de energia, mas também melhora o desempenho, reduzindo o tempo de execução.

Figura 7 – Comparativo microcontroladores



Fonte: Elaborado pelo autor.

Em termos arquiteturais, os níveis de pipelining são importantes para a redução do consumo de potência por operação. Ao adotar uma arquitetura mais eficiente em termos de *throughput*, é possível diminuir a frequência de operação sem comprometer o desempenho requerido em termos de tempo de execução. O *pipelining* permite que várias instruções sejam processadas simultaneamente em diferentes estágios, aumentando a eficiência do microcontrolador e reduzindo a necessidade de operar em frequências mais altas, que consomem mais energia. Além disso, a divisão das tarefas em estágios menores e mais especializados contribui para uma melhor gestão do consumo de energia, já que cada estágio pode ser otimizado individualmente para operar de forma mais eficiente.

A ativação do *bypass* e do *early branch* aumenta o desempenho em todas as configurações, mas também eleva o consumo de energia. Isso é particularmente evidente em

microcontroladores com multiplicação em hardware, onde a combinação dessas otimizações resulta em um desempenho significativamente melhor, mas com um custo maior em termos de consumo de energia. Em microcontroladores sem multiplicação em hardware, as otimizações também melhoram o desempenho, mas com um aumento menos pronunciado no consumo de energia. Portanto, a escolha entre ativar ou desativar essas otimizações deve considerar o equilíbrio entre desempenho e eficiência energética, adaptando-se às necessidades específicas da aplicação. Em cenários que exigem alta performance, como processamento de operações complexas, a ativação dessas otimizações é justificada, enquanto em aplicações que priorizam baixo consumo de energia, a simplificação do pipeline pode ser mais vantajosa.

A escolha da configuração ideal depende das necessidades específicas da aplicação. Para sistemas embarcados com restrição de hardware, um pipeline mais curto pode ser a melhor alternativa, mesmo que o tempo de execução seja maior. Já em aplicações que demandam maior *throughput*, um pipeline mais profundo, aliado a otimizações como *bypass* e *early branch*, pode justificar o maior utilização de recursos. Portanto, a definição do número de estágios deve considerar o compromisso entre velocidade e eficiência de hardware, sempre alinhado às restrições do sistema.

O código-fonte e a documentação deste trabalho estão disponíveis no repositório do GitHub[1], onde é possível acessar todas as implementações e configurações utilizadas para compô-lo.

[1] ⟨https://github.com/thiago0003/TCC_RISC_V⟩

4 Conclusão

Este trabalho analisou detalhadamente o desempenho, consumo de recursos e o consumo de energia dos processadores Murax em diferentes configurações, explorando as características e os trade-offs de processadores com diferentes números de estágios de pipeline. Através do benchmark Dhrystone, foram avaliadas as vantagens e desvantagens de cada configuração em termos de desempenho, consumo de energia e complexidade de design. Processadores com dois estágios destacam-se pela simplicidade de implementação e menor consumo de energia, sendo ideais para aplicações que priorizam eficiência energética e baixa complexidade. No entanto, seu desempenho é limitado, especialmente em operações complexas, como multiplicações.

Processadores com três estágios oferecem um equilíbrio entre desempenho e consumo de energia, sendo uma solução intermediária que atende a uma ampla gama de aplicações. Eles proporcionam uma melhoria significativa no desempenho em comparação com os de dois estágios, mas ainda enfrentam limitações em relação à frequência máxima de clock e ao consumo de energia, devido à lógica de controle adicional. Por outro lado, processadores com quatro estágios apresentam o maior desempenho, graças à maior divisão de tarefas, o que os torna adequados para aplicações que exigem alta performance, como operações complexas de multiplicação e divisão. No entanto, essa vantagem vem acompanhada de um aumento na complexidade do design e no consumo de recursos lógicos e energia.

A análise de desempenho revelou que a ativação do *bypass* e do *early branch* melhora significativamente os resultados em termos de Dhrystones por segundo. Para processadores de inteiros sem multiplicação, a combinação dessas otimizações resultou em um aumento notável no desempenho, especialmente em processadores com três e quatro estágios. A inclusão de multiplicação manteve essa tendência, com os valores de Dhrystones por segundo alcançando o máximo quando ambas as otimizações estavam ativadas. A análise dos ALUTs mostrou que a ativação do *bypass* e do *early branch* aumenta o consumo de recursos lógicos. No entanto, essa sobrecarga é um compromisso aceitável para alcançar melhor performance, já que desativar essas otimizações resulta em menor consumo de ALUTs e registradores, mas também em desempenho reduzido.

A análise do consumo de energia destacou que, enquanto as otimizações de pipeline melhoram o desempenho, elas também aumentam o consumo de energia. A inclusão de multiplicação introduziu uma carga adicional de energia, mas, ainda assim, as otimizações de *bypass* e *early branch* demonstraram ser eficientes ao melhorar o desempenho de forma significativa. Em síntese, a escolha do número de estágios e a ativação de otimizações devem ser guiadas pelas necessidades específicas da aplicação, considerando o equilíbrio entre desempenho, consumo de energia e complexidade do design.

A escolha entre implementar a multiplicação em hardware ou software envolve um

trade-off significativo entre desempenho, consumo de energia e uso de recursos. A multiplicação em hardware, embora aumente o consumo de energia e o uso de recursos lógicos (ALUTs e registradores), oferece um ganho substancial em desempenho, especialmente em operações complexas e repetitivas. Por outro lado, a multiplicação em software reduz o consumo de energia e a complexidade do design, sendo uma opção viável para aplicações que priorizam eficiência energética e simplicidade, mas com um custo em termos de desempenho, já que operações de multiplicação em software são intrinsecamente mais lentas. Portanto, a decisão entre hardware e software deve considerar as necessidades específicas da aplicação, como a frequência de operações de multiplicação e a disponibilidade de recursos, para equilibrar desempenho e eficiência.

A escolha do número de estágios de pipeline deve ser guiada pelas necessidades específicas da aplicação. Para sistemas que priorizam eficiência energética e simplicidade, processadores de dois estágios são a melhor opção, embora apresentem desempenho reduzido. Para aplicações que exigem um equilíbrio entre desempenho e consumo, processadores de três estágios são recomendados, enquanto cenários que demandam alta performance se beneficiam de processadores de quatro estágios, apesar do maior custo em termos de complexidade e consumo de energia. A comparação entre os números de estágios revelou que processadores com três e quatro estágios tendem a ter melhor desempenho quando as otimizações de *bypass* e *early branch* estão ativadas, enquanto processadores com dois estágios podem apresentar menor consumo de energia, mas com desempenho limitado.

Os resultados obtidos sugerem que há um equilíbrio delicado entre desempenho e eficiência de recursos. Desativar o *bypass* e o *early branch* pode ser benéfico para designs que priorizam o consumo de energia e a simplicidade, enquanto ativá-los é essencial para alcançar a máxima performance, especialmente em operações complexas como multiplicação. Processadores com mais estágios de pipeline podem executar tarefas menores e mais específicas em cada estágio, aumentando a eficiência por operação. No entanto, isso requer mais circuitos e lógica para gerenciar o pipeline, aumentando o consumo de energia total. Portanto, ao considerar diferentes aplicações, é crucial ajustar as configurações de pipeline para otimizar tanto o desempenho quanto o consumo de energia, garantindo que as escolhas de design atendam aos requisitos específicos de cada cenário.

Este estudo reforça a importância de compreender os trade-offs envolvidos no design de processadores para otimizar o desempenho e a eficiência de sistemas embarcados e de alta performance, servindo como um guia para futuras implementações que buscam um equilíbrio adequado entre eficiência energética e desempenho.

A Apêndice

A.1 Murax(I) síntese IDE Gowin

Os dados gerados pea IDE Gowin, apresentados abaixo, não levam em consideração o arquivo de simulação VCD gerado pelo Icarus Verilog, dessa forma os dados são apenas do processador estático não sendo passível de definir o consumo real de potência durante a execução de um programa. Os dados de potência leva em conta apenas o consumo do processador ligado e o consumo dos periféricos UART, JTag.

Devido as limitações de otimizações da Gowin pode explicar valores de clock inferiores em comparação com a ferramenta da Intel (Quartus II) Tabela 2.

Tabela 2 – Clock máximo Murax(I) (MHz)

Bypass	Early Branch	Quatro estágios	Três estágios	Dois estágios
False	False	55.99	56.97	N/A
False	True	45.84	46.54	45.84
True	False	49.92	55.43	N/A
True	True	44.71	45.13	49.75

Fonte: Elaborado pelo autor.

A Tabela 3 apresenta o consumo de elementos lógicos (ALUTs) para o processador Murax em diferentes configurações de pipeline e otimizações (*bypass* e *early branch*).

Tabela 3 – Elementos lógicos Murax(I)

Bypass	Early Branch	Quatro estágios	Três estágios	Dois estágios
False	False	1497	1504	N/A
False	True	1486	1457	1466
True	False	1778	1809	N/A
True	True	1772	1736	1611

Fonte: Elaborado pelo autor.

A Tabela 4 apresenta o consumo de registradores para o processador Murax em diferentes configurações de pipeline e otimizações (*bypass* e *early branch*).

Tabela 4 – Registradores Murax(I)

Bypass	Early Branch	Quatro estágios	Três estágios	Dois estágios
False	False	967	886	N/A
False	True	936	855	838
True	False	1002	919	N/A
True	True	971	888	838

Fonte: Elaborado pelo autor.

A Tabela 5 apresenta o consumo de potência para o processador Murax em diferentes configurações de pipeline e otimizações (*bypass* e *early branch*).

Tabela 5 – Potência Murax(I) (mW)

Bypass	Early Branch	Quatro estágios	Três estágios	Dois estágios
False	False	30.388	30.377	N/A
False	True	30.385	30.373	30.369
True	False	30.429	30.424	N/A
True	True	30.426	30.413	30.413

Fonte: Elaborado pelo autor.

A.1.1 Saídas

A.1.1.1 Simulação

Os dados apresentados na Figura 8 correspondem à saída da simulação em formato VCD, gerada pelo Icarus Verilog. Esses resultados demonstram que o processador executou o programa Dhrystone corretamente, conforme esperado. A simulação valida o funcionamento do processador em um ambiente controlado, garantindo que a lógica implementada está operando conforme o projetado.

A.1.1.2 Execução

Já a Figura 9 exibe a saída capturada diretamente da placa DE10 Standard da Intel, por meio dos pinos UART. Ao comparar os resultados obtidos na simulação com os observados na execução real, verifica-se que o processador executou o programa Dhrystone de maneira correta e consistente. Essa concordância entre a simulação e a execução prática confirma a precisão do projeto e a correta implementação do processador.

```
martins@pop-os:~/Downloads/VexRiscv$ iverilog -o murax top.v Murax.v tb_murax.v
martins@pop-os:~/Downloads/VexRiscv$ vvp murax
VCD info: dumpfile dump_murax_4_stage.vcd opened for output.

Dhrystone Benchmark, Version 2.1 (Language: C)

Program compiled without 'register' attribute

Please give the number of runs through the benchmark:
Execution starts, 200 runs through Dhrystone
Execution ends

Final values of the variables used in the benchmark:

Int_Glob:            5
        should be:   5
Bool_Glob:           1
        should be:   1
Ch_1_Glob:           A
        should be:   A
Ch_2_Glob:           B
        should be:   B
Arr_1_Glob[8]:       7
        should be:   7
Arr_2_Glob[8][7]:    210
        should be:   Number_Of_Runs + 10
Ptr_Glob->
  Ptr_Comp:          -2147458812
        should be:   (implementation-dependent)
  Discr:             0
        should be:   0
  Enum_Comp:         2
        should be:   2
  Int_Comp:          17
        should be:   17
  Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Next_Ptr_Glob->
  Ptr_Comp:          -2147458812
        should be:   (implementation-dependent), same as above
  Discr:             0
        should be:   0
  Enum_Comp:         1
        should be:   1
  Int_Comp:          18
        should be:   18
  Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Int_1_Loc:           5
        should be:   5
Int_2_Loc:           13
        should be:   13
Int_3_Loc:           7
        should be:   7
Enum_Loc:            1
        should be:   1
Str_1_Loc:           DHRYSTONE PROGRAM, 1'ST STRING
        should be:   DHRYSTONE PROGRAM, 1'ST STRING
Str_2_Loc:           DHRYSTONE PROGRAM, 2'ND STRING
        should be:   DHRYSTONE PROGRAM, 2'ND STRING
```

Figura 8 – Resultados da simulação do Dhrystone no Icarus Verilog.

Fonte: Elaborado pelo autor.

```
martins@vlab:~$ cat /dev/ttyUSB0

Dhrystone Benchmark, Version 2.1 (Language: C)

Program compiled without 'register' attribute

Please give the number of runs through the benchmark:
Execution starts, 200 runs through Dhrystone
Execution ends

Final values of the variables used in the benchmark:

Int_Glob:            5
        should be:   5
Bool_Glob:           1
        should be:   1
Ch_1_Glob:           A
        should be:   A
Ch_2_Glob:           B
        should be:   B
Arr_1_Glob[8]:       7
        should be:   7
Arr_2_Glob[8][7]:    210
        should be:   Number_Of_Runs + 10
Ptr_Glob->
  Ptr_Comp:          -2147458812
        should be:   (implementation-dependent)
  Discr:             0
        should be:   0
  Enum_Comp:         2
        should be:   2
  Int_Comp:          17
        should be:   17
  Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Next_Ptr_Glob->
  Ptr_Comp:          -2147458812
        should be:   (implementation-dependent), same as above
  Discr:             0
        should be:   0
  Enum_Comp:         1
        should be:   1
  Int_Comp:          18
        should be:   18
  Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Int_1_Loc:           5
        should be:   5
Int_2_Loc:           13
        should be:   13
Int_3_Loc:           7
        should be:   7
Enum_Loc:            1
        should be:   1
Str_1_Loc:           DHRYSTONE PROGRAM, 1'ST STRING
        should be:   DHRYSTONE PROGRAM, 1'ST STRING
Str_2_Loc:           DHRYSTONE PROGRAM, 2'ND STRING
        should be:   DHRYSTONE PROGRAM, 2'ND STRING
```

Figura 9 – Resultados da execução do Dhrystone na placa DE10 Standard.

Fonte: Elaborado pelo autor.

Referências

Ahmadi-Pour, Herdt e Drechsler 2022 AHMADI-POUR, S.; HERDT, V.; DRECHSLER, R. The microrv32 framework: An accessible and configurable open source risc-v cross-level platform for education and research. *Journal of Systems Architecture*, v. 133, p. 102757, 2022. ISSN 1383-7621. Disponível em: ⟨https://www.sciencedirect.com/science/article/pii/S1383762122002429⟩.

Allam et al. 2024 ALLAM, O. E. et al. Explore open source hardware solutions for iot applications. In: *2024 International Conference on Circuit, Systems and Communication (ICCSC)*. [S.l.: s.n.], 2024. p. 1–6.

Bobda e Hartenstein 2007 BOBDA, C.; HARTENSTEIN, R. *Introduction to reconfigurable computing: architectures, algorithms, and applications*. [S.l.]: Springer, 2007. v. 1.

Bora e Paily 2021 BORA, S.; PAILY, R. A high-performance core micro-architecture based on risc-v isa for low power applications. *IEEE Transactions on Circuits and Systems II: Express Briefs*, v. 68, n. 6, p. 2132–2136, 2021.

Chisnall 2024 CHISNALL, D. How to design an isa: The popularity of risc-v has led many to try designing instruction sets. *Queue*, Association for Computing Machinery, New York, NY, USA, v. 21, n. 6, p. 27–46, jan. 2024. ISSN 1542-7730. Disponível em: ⟨https://doi.org/10.1145/3639445⟩.

Compton, Hauck e Compton 2000 COMPTON, K.; HAUCK, S.; COMPTON, K. An introduction to reconfigurable computing. *IEEE Computer*, Citeseer, v. 9, 2000.

Efinix 2024 Efinix. *RISC-V Products*. 2024. Acessado em: 10 nov. 2024. Disponível em: ⟨https://www.efinixinc.com/products-riscv.html⟩.

Farooq, Marrakchi e Mehrez 2012 FAROOQ, U.; MARRAKCHI, Z.; MEHREZ, H. Fpga architectures: An overview. In: ______. *Tree-based Heterogeneous FPGA Architectures: Application Specific Exploration and Optimization*. New York, NY: Springer New York, 2012. p. 7–48. ISBN 978-1-4614-3594-5. Disponível em: ⟨https://doi.org/10.1007/978-1-4614-3594-5_2⟩.

Flynn 1995 FLYNN, M. J. Computer architecture pipelined and parallel processor design" by jones and bartlett publishers. *Inc copyrights*, 1995.

Golson e Clark 2016 GOLSON, S.; CLARK, L. Language wars in the 21st century: verilog versus vhdl–revisited. *Synopsys Users Group (SNUG)*, 2016.

Hacker News 2024 Hacker News. *Discussion on RISC-V*. 2024. Acessado em: 10 nov. 2024. Disponível em: ⟨https://news.ycombinator.com/item?id=23263752⟩.

Intel Corporation 2024 Intel Corporation. *ALUT Definition - Quartus Help*. 2024. Acessado em: 10 fev. 2025. Disponível em: ⟨https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/reference/glossary/def_alut.htm⟩.

Masood 2011 MASOOD, F. Risc and cisc. *arXiv preprint arXiv:1101.5364*, 2011.

Papon 2020 PAPON, C. *VexRiscv*. [S.l.]: VexRiscv, 2020.

Patterson e Waterman 2017 PATTERSON, D.; WATERMAN, A. *The RISC-V Reader: an open architecture Atlas*. [S.l.]: Strawberry Canyon, 2017.

Patterson e Hennessy 2016 PATTERSON, D. A.; HENNESSY, J. L. *Computer organization and design ARM edition: the hardware software interface*. [S.l.]: Morgan kaufmann, 2016.

SpinalHDL 2024 SpinalHDL. *Issue #128: Discussion on VexRiscv*. 2024. Acessado em: 10 nov. 2024. Disponível em: ⟨https://github.com/SpinalHDL/VexRiscv/issues/128⟩.

SpinalHDL 2024 SpinalHDL. *SpinalHDL Documentation*. 2024. Acessado em: 10 nov. 2024. Disponível em: ⟨https://spinalhdl.github.io/SpinalDoc-RTD/master/index.html⟩.

SpinalHDL 2024 SpinalHDL. *SpinalHDL Introduction*. 2024. Acessado em: 10 nov. 2024. Disponível em: ⟨https://spinalhdl.github.io/SpinalDoc-RTD/master/SpinalHDL/Introduction/SpinalHDL.html⟩.

SpinalHDL 2024 SPINALHDL. *VexRiscv*. 2024. ⟨https://github.com/SpinalHDL/VexRiscv⟩. Accessed: 2024-08-31.

Thomas e Moorby 2008 THOMAS, D.; MOORBY, P. *The Verilog® hardware description language*. [S.l.]: Springer Science & Business Media, 2008.

Verma e Stan 2022 VERMA, V.; STAN, M. R. Ai-pim—extending the risc-v processor with processing-in-memory functional units for ai inference at the edge of iot. *Frontiers in Electronics*, Frontiers Media SA, v. 3, p. 898273, 2022.

Weicker 1984 WEICKER, R. P. Dhrystone: a synthetic systems programming benchmark. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 27, n. 10, p. 1013–1030, out. 1984. ISSN 0001-0782. Disponível em: ⟨https://doi.org/10.1145/358274.358283⟩.

York 2002 YORK, R. Benchmarking in context: Dhrystone. *ARM, March*, Citeseer, 2002.