Desenvolvimento de RISC-V para Uso Aeroespacial

Giancarlo Vanoni Ruggiero (giancarlovr@al.insper.edu.br)
Luciano Felix Dias (lucianofd@al.insper.edu.br)
Tiago Vitorino Seixas (tiagovs1@al.insper.edu.br)

Trabalho de Conclusão de Curso

Relatório
Versão Final
do Projeto Final de Engenharia

São Paulo-SP
Março 2024

Giancarlo Vanoni Ruggiero
Luciano Felix Dias
Tiago Vitorino Seixas

Desenvolvimento de RISC-V para Uso Aeroespacial

Relatório Versão Final do Projeto Final de Engenharia

Relatório apresentado ao curso de Engenharia, como requisito para o Trabalho de Conclusão de Curso.

Professor Orientador: Prof. Rafael Corsi Ferrão

Mentor na Empresa: Saulo Finco

Coordenador TCC/PFE: Prof. Dr. Luciano Pereira Soares

São Paulo-SP
Março 2024

Giancarlo Vanoni Ruggiero
Luciano Felix Dias
Tiago Vitorino Seixas

RISC-V Development for Aerospatial Application

Capstone Design Report Report Final Version

Undergraduate Capstone Design Project Report submitted to the Engineering program (Computer/Mechanical/Mechatronic) in partial fulfillment of the requirements for the Bachelor's Degree in Engineering.

Advisor: Rafael Corsi Ferrão

Company Mentor: Saulo Finco

General Coordinator TCC/PFE: Dr. Luciano Pereira Soares

São Paulo - SP
Março 2024

Giancarlo Vanoni Ruggiero
Luciano Felix Dias
Tiago Vitorino Seixas

Relatório Versão Intermediária do Projeto Final de Engenharia

Projeto Final de Engenharia apresentado ao programa Graduação em Engenharia (Computação/Mecânica/Mecatrônica) como requisito parcial para a obtenção do título de Bacharel em Engenharia.

Orientador: Rafael Corsi Ferrão

Banca Examinadora

Rafael Corsi Ferrão
Insper

Paulo Carlos Ferreira dos Santos
Insper

Rodrigo de Marca França
IMT

Sumário

Resumo

O objetivo central deste projeto é o desenvolvimento de uma Propriedade Intelectual (Intellectual Property - IP) de um processador baseado na arquitetura RISC-V para aplicações aeroespaciais. O processador deve ter sua confiabilidade garantida para uso em ambientes aeroespaciais por meio de testes dos elementos individuais desenvolvidos em VHDL, assim como testes da CPU como um todo. Utilizou-se da metodologia bottom-up para guiar o desenvolvimento do projeto, onde o foco principal era inicialmente na implementação de componentes, e conforme o projeto progredia, havia uma transição do foco para a integração dos componentes. O projeto foi feito com o auxílio de um mural Kanban no GitHub e a criação de tarefas com alocação de responsáveis e revisores. Quando é feita alguma atualização na base de código, é realizada uma sequência automatizada de testes de validação na nuvem utilizando GitHub Actions. Além disso, há uma documentação e referência do projeto como material de apoio em uma página publicada on-line.

Palavras-chave: RISC-V; Aplicação Aeroespacial; VHDL; FPGA; Python; Cocotb; Yosys; CI/CD; Bottom-Up; Mural Kanban.

Abstract

The central objective of this project is the development of an Intellectual Property (IP) for a processor based on the RISC-V architecture for aerospace applications. The processor must ensure its reliability for use in aerospace environments through tests of the individual elements developed in VHDL, as well as tests of the CPU as a whole. The project followed a bottom-up methodology, initially focusing on component implementation, and gradually transitioning to component integration as the project progressed. A Kanban board on GitHub was used to guide the project, with tasks assigned to responsible individuals and reviewers. Whenever there is an update to the codebase, an automated validation test sequence is performed in the cloud using GitHub Actions. Additionally, there is documentation and project references available as supporting material on an online page.

Keywords: RISC-V; Aerospace Application; VHDL; FPGA; Python; Cocotb; Yosys; CI/CD; Bottom-Up; Kanban.

1. Introdução

O objetivo deste projeto final de engenharia (PFE) é desenvolver um processador baseado na arquitetura aberta de conjunto de instruções RISC-V, proposto pelo Centro de Tecnologia da Informação Renato Archer (CTI) para uso em aplicações aeroespaciais. A princípio, este projeto surgiu da necessidade de tornar o Brasil independente no setor de tecnologia aeroespacial e desenvolver processadores para os satélites do programa espacial nacional dentro do país. Uma vez que o cliente não possui uma arquitetura de processador própria, como ARM (*Advanced RISC Machine*) da ARM Holdings e x86 da Intel, há dependência de Propriedade Intelectual (Intellectual Property - IP) de empresas que vendem processadores, o que pode levar à falta destes componentes no Brasil dependendo da oferta disponível do mercado ou caso ocorram conflitos de interesse. Como solução para o problema apresentado, foi proposto o projeto pelo CTI como tema de PFE de longo prazo para os alunos de Engenharia da Computação, sendo o RISC-V a escolha do cliente.

Foi acordado entre os alunos do PFE e os membros da CTI que, até a entrega intermediária, deve-se ter como resultado a base do que se deseja desenvolver: um processador RISC-V funcional, desenvolvido seguindo boas práticas de desenvolvimento, como desenvolvimento hierárquico modular para facilitar a atualização do projeto para suportar uma arquitetura com tolerância a falhas ao processador. Também deve ser criado um ambiente de testes de unidade e de integração em um ambiente de desenvolvimento conteinerizado, utilizando Docker, para que a infraestrutura seja replicável.

O projeto não demonstra ter questões no âmbito legal, e pode ter uso prático futuramente de acordo com as necessidades do CTI, porém como a proposta do CTI é desenvolver o projeto a longo prazo em parceria com o Insper, sendo este PFE a primeira etapa para a criação da IP desejada, o andamento do projeto fica dependente do interesse de alunos do PFE do Insper de semestres posteriores continuarem o desenvolvimento.

1.1 Escopo do projeto

Após conversas com o cliente, o escopo do projeto foi definido como sendo o desenvolvimento de um processador funcional com testes unitários e de integração implementados, sendo que a empresa seria responsável pela fabricação da CPU.

1. O que faz parte do projeto
    - Desenvolvimento de um processador baseado na arquitetura RISC-V;
    - Infraestrutura de testes unitários e de integração para o processador e seus componentes;
    - Criação de um manual com o guia de uso e referência para o projeto e o processador.
2. O que não faz parte do projeto
    - Fabricação da CPU

Vale ressaltar que foi acordado junto ao cliente que a arquitetura do processador a ser desenvolvida seria a do Conjunto de Inteiros para 32 *bits* (RV32I), que possui um total de 40 instruções, das quais 3 foram definidas como estando fora de escopo, uma vez que são necessárias para processadores superescalares, que não é o objetivo deste projeto. Também foi proposto como desafio implementar a extensão de multiplicação para expandir a funcionalidade do processador.

Foi desenvolvido um processador capaz de executar as 37 instruções de forma individual e integrada, assim como acordado com o cliente, com todos os componentes individuais tendo pelo menos um caso de teste implementado, exceto pelas etapas do *pipeline*, que foram desenvolvidas usando registros, o que impede que sejam diretamente testadas usando a metodologia de testes deste projeto, porém apesar desse percalço, as etapas foram validadas indiretamente nos testes das instruções, que também servem para validar o processador como um todo, e a integração de seus elementos.

Essa integração fica mais clara ao observar a Figura 1, que ilustra a visão geral (*top level*) do processador desenvolvido até o momento, com os barramentos representando os registradores entre as etapas do *pipeline*.

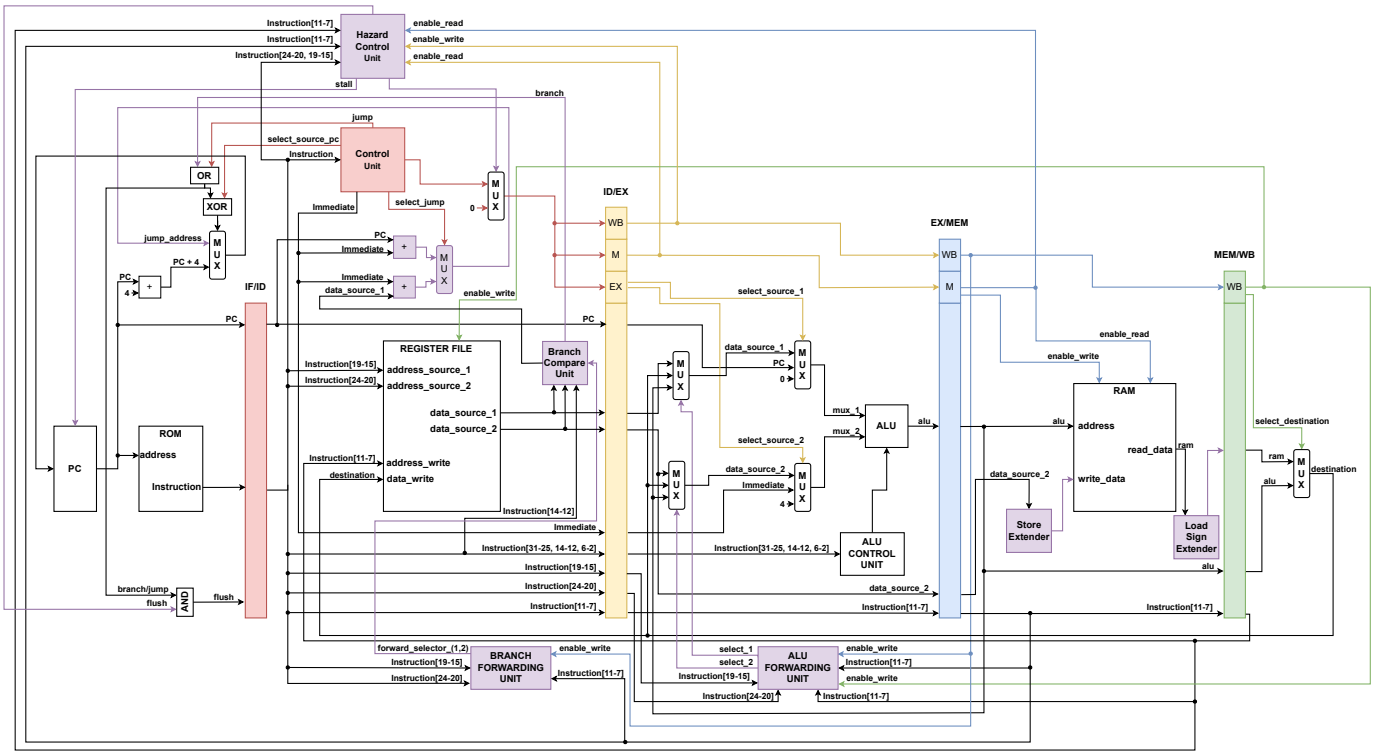


Figura 1 - Diagrama do processador implementado, ilustração baseada em um simulador *web* de RISC-V (MARIOTTI; GIORGI, 2022)

1.2 Recursos

Para o desenvolvimento do processador, foi necessário o uso de um conjunto de ferramentas:

- **Visual Studio Code**, um editor de texto muito utilizado por programadores para desenvolver código, com propósito de facilitar o processo de desenvolvimento e teste do processador e seus componentes (MICROSOFT, 2015);
- **GHDL**, uma ferramenta open-source de sintetização para o projeto em VHDL e simulação dos componentes (GINGOLD, 2023);

- **Yosys**, é uma ferramenta open-source de sintetização e elaboração do projeto em VHDL (WOLF, 2020);
- **Netlistsvg**, é uma ferramenta open-source de elaboração de diagramas vetoriais dos componentes para o Yosys (TURLEY, 2016);
- **Cocotb**, um framework open-source de verificação de design de chips em python, com propósito de testar os componentes desenvolvidos (COCOTB, 2024);
- **Pytest**, um framework open-source de testes em python, com propósito de testar os componentes desenvolvidos (KREKEL; TEAM, 2015);
- **GitHub**, um ambiente de desenvolvimento colaborativo com controle de versão, com propósito de gestão do projeto e automação da execução de testes unitários e de integração por meio de uma infraestrutura em nuvem (GITHUB, 2024);
- **GitHub Pages**, uma infraestrutura em nuvem, com propósito de publicar o guia de uso e documentação do projeto on-line (GITHUB PAGES, 2024).
- **Quartus**, um software de design de dispositivos lógicos da Intel (INTEL, 2024).

1.3 Mapeamento dos *stakeholders*

O mapeamento dos *stakeholders* pode ser visto na Tabela *Stakeholders*:

Stakeholder	Posição	Papel no Projeto	Expectativas
Saulo Finco	Tecnologista Sênior	Orientador da empresa	Desenvolvimento de um RISC-V para uso aeroespacial com infraestrutura CI/CD de testes para a criação de uma propriedade intelectual baseada nele
Líderes do CTI/Ministério da Ciência, Tecnologia e Inovação	Empresa parceira	Empresa parceira	Desenvolvimento em uma propriedade intelectual baseada no RISC-V para obter soberania no setor aeroespacial
Rafael Corsi	Professor	Professor orientador	Desenvolvimento de um RISC-V com infraestrutura CI/CD de testes, utilizando os conceitos aprendidos durante o curso pelos alunos
RISC-V International	Organização Internacional	Entidade interessada	Desenvolvimento da arquitetura RISC-V, maior contribuição do Brasil para a RISC-V International

Tabela *Stakeholders* - Tabela de mapeamento dos *stakeholders*.

1.4 Questões Éticas e Profissionais

Um passo na busca pela soberania do Brasil no setor de tecnologia aeroespacial está na posse de conhecimento prático do desenvolvimento de componentes de eletrônica avançada utilizados em programas espaciais. A dependência de empresas que possuem as propriedades intelectuais desses componentes, o que faz o uso dessas tecnologias ocorrer por meio de licenças, pode comprometer o desenvolvimento desses programas no Brasil por oferta do mercado ou conflitos de interesse.

Por isso, o desenvolvimento do RISC-V, uma arquitetura aberta gerida por uma entidade internacional da qual o Brasil faz parte desde o final de fevereiro de 2024 (MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E INOVAÇÃO, 2024), é de interesse da nação brasileira e da RISC-V International, assim como dos países presentes nesta organização.

1.5 Normas Técnicas

Utilizou-se das normas definidas pela entidade que é responsável pela especificação da arquitetura RISC-V (2022, RISC-V International), onde se define que o processador deve ser capaz de interpretar um conjunto de instruções, que está especificado no Anexo A, e quaisquer futuras iterações do projeto devem seguir essas normas também.

No que se refere às normas de desenvolvimento da Gaisler (GAISLER, 2024), o projeto não foi desenvolvido inteiramente com uso de records pois a infraestrutura de testes não possibilita teste de componentes que usam records.

1.6 Revisão do Estado da Arte

O RISC-V (Reduced Instruction Set Computing - V) é uma arquitetura de conjunto de instruções aberta, desenvolvida pela Universidade da Califórnia, Berkeley, que tem como principal característica a modularidade, possuindo o core e permitindo acrescentar extensões de acordo com necessidade. Diversas empresas,e universidades pelo mundo tem investido na arquitetura para aplicações distintas (RISC-V FOUNDATION, 2023).

Alguns projetos que tem sido desenvolvidos pelas organizações são:

- Xuantie-910: criado pelo Alibaba, possui 16 núcleos feitos com RV64GCV, e apresenta extensões personalizadas para operações aritméticas, manipulação de bits, *load* e _store, e operações de cache. Cada core possui um pipeline de 12 estágios, alcançando um clock máximo de 2.5 GHz. Implementações do Xuantie FPGA, foram instaladas nos *data centers* da Alibaba para acelerar determinadas aplicações, como blockchain (CHEN et al., 2020).

- PULP: a ETH Zurich e a University of Bologna, tornaram abertos os códigos da plataforma PULP (Parallel Ultra Low Power). Essa plataforma contém diversas IPs, indo desde um núcleo com 2 estágios de 32 bits, com o RISC-V RV32EC para um com 6 estágios, com cache e com a nomenclatura RV64GC. A plataforma também possui IPs para comunicação e pode ser integrada em uma FPGA (ROSSI et al., 2015).

O desenvolvimento de um processador com essa arquitetura para o ramo aeroespacial possui alguns desafios, pois é preciso ser resistente a radiação, temperaturas extremas, vibrações mecânicas e ao vácuo do espaço. Um exemplo de processador utilizado para o ramo aeroespacial é o LEON, que possui tolerância a falhas e os requisitos necessários, que é baseado na arquitetura SPARC, e utilizado pela Agência Espacial Européia (ESA) (MASCIO et al., 2019).

A Gaisler, empresa que desenvolve produtos LEON, desenvolveu o NOEL-V, um processador RISC-V para uso aeroespacial. Ele possui possui tolerância a falhas, sendo de acordo com a empresa, ideal para aplicações aeroespaciais. O NOEL-V também faz uso da modularidade da arquitetura RISC-V, tendo várias versões, cada uma com extensões diferentes, que pode ser visto na Tabela NOEL-V, permitindo atender necessidades específicas de cada aplicação (GAISLER, 2022).

Configuration	Target	Architecture	Pipeline	RISC-V extensions	MMU	PMP	Privilege modes	Example SW
HP	High-performance processing	32 or 64 bits	Dual issue	IMAFDB*CH	Yes	Yes	Supervisor, User and Machine + Virtualization	Hypervisor, Linux, VxWorks
GP	General purpose processing	32 or 64 bits	Dual or single issue	IMAFDB*CH	Yes	Yes	Supervisor, User and Machine + Virtualization	Hypervisor, Linux, VxWorks
GP-lite	General purpose processing Area optimized	32 or 64 bits	Dual or single issue	IMAFDB*C	Yes	No	Supervisor, User and Machine	Linux, VxWorks
MC	Controller applications	32 or 64 bits	Single issue	IMAFDB*C	No	Yes	User and Machine	RTEMS
MC-lite	Controller applications Area Optimized	32 or 64 bits	Single issue	IMA	No	No	User and Machine	RTEMS

Tabela *NOEL-V* - Tabela das versões do NOEL-V.

Além da Gaisler, a empresa SiFive, também desenvolve IPs cores para a linha de FPGAs da Microchip, utilizando o seu ecossistema chamado Mi-V. Essa linha de FPGAs inclui placas tolerantes a radiação (MICROCHIP, 2022).

O uso de uma arquitetura aberta permite o reuso e a acumulação do conhecimento de design, que acelera o desenvolvimento de novos produtos. Além disso, ao utilzar uma arquitetura aberta, facilita o desenvolvimento de software, pois é possível manter seu ecossistema em diferentes projetos (FURANO et al., 2022).

Atualmente para o ecossistema de software do RISC-V, existem diversas ferramentas de desenvolvimento, como os compiladores GCC (GCC, 2024) e LLVM (LLVM, 2024), para linguagens como C e C++. Além disso, a ferramenta de *debug* GDB (GDB, 2024) também é compatível com a arquitetura.

Algo que contribui para o ecossistema de software são os sistemas operacionais disponíveis. Atualmente, O FreeRTOS (FREERTOS, 2020), Zephyr (ZEPHYR, 2015), e algumas distribuições do Linux, como o Fedora (FEDORA, 2023), Debian (DEBIAN, 2021) e Ubuntu (UBUNTU, 2022), possuem suporte para a arquitetura RISC-V.

2. Metodologia

Para a estruturação do processador, optou-se por uma metodologia de desenvolvimento hierárquico modularizado. A hierarquia do desenvolvimento se refere à separação do processador em diferentes níveis de integração, e a modularização à divisão dos elementos que compõem o processador em arquivos separados, ambos métodos que foram usados em conjunto para deixar o projeto mais organizado e facilitar futuras iterações do projeto, como implementação de tolerância a falhas por meio de tripla votação por exemplo, que é a redundância de componentes para o caso de um falhar, onde os outros são usados para definir o valor correto a ser considerado.

Os níveis de integração do projeto foram:

- O nível mais baixo corresponde ao nível dos componentes genéricos, como está ilustrado na Figura 2;
- Acima dos genéricos, há o nível dos componentes específicos da arquitetura RV32I, ilustrados na Figura 3;
- Em seguida, o nível dos módulos, ilustrado na Figura 4. Vale mencionar que o Módulo Unidade de Controle é o único que implementa a lógica diretamente, sem fazer uso de componentes, já que esse módulo atua como ponte entre duas etapas do *pipeline*, mas ainda faz parte de uma etapa;
- Há, então, o nível da CPU, onde estão os elementos que, integrados, compõem o processador, sendo esses as etapas do *pipeline* e os elementos responsáveis por resolver os hazards, como se vê na Figura 5. Vale notar que os registradores das etapas são simbólicos, pois não existe um componente registrador específico para uso no *pipeline*. Ao invés disso, as próprias etapas do mesmo tem função de registrador.
- Também faz parte do nível da CPU a implementação dos componentes que resolvem os *hazards* do processador, como está ilustrado na Figura 6. Se faz necessário mencionar, que devido ao fato de o cliente querer usar memórias externas ao processador, o projeto foi realizado de forma que as memórias desenvolvidas pelo grupo para propósito de teste estão fora da CPU, e por isso estão representadas apenas para propósito de facilitar o entendimento da mesma, caso a situação não fosse essa, o grupo teria implementado as memórias nas etapas ID e MEM, o que é mais fácil de visualizar na Figura 1 (ver Seção 1.1).
- Por fim, tem-se o nível de visão geral do projeto, com a implementação da CPU, cujo *top level* também faz parte do nível da CPU, e das memórias usadas para teste, como está ilustrado na Figura 7.

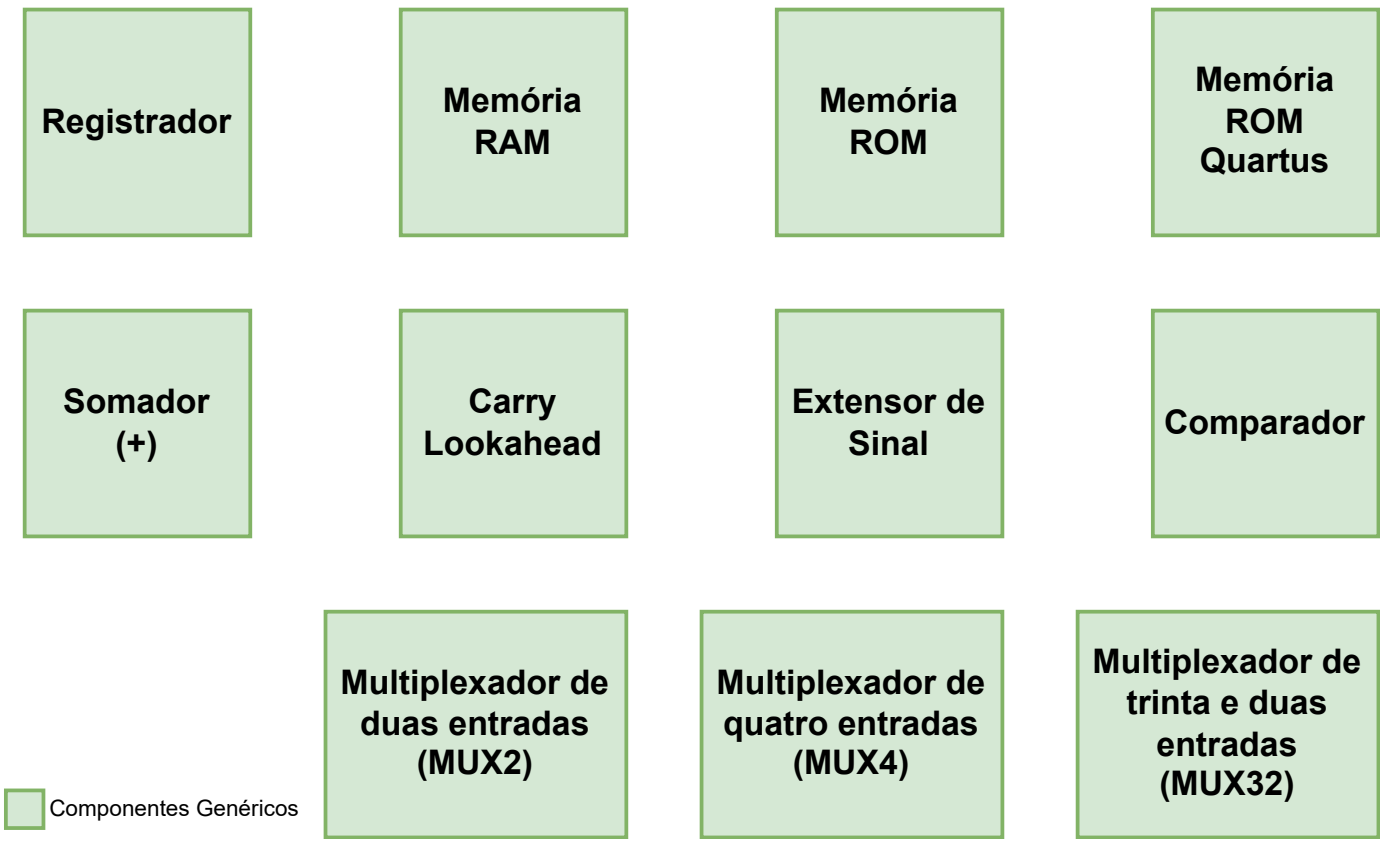


Figura 2 - Diagrama dos componentes genéricos implementados.

Componentes RV32I

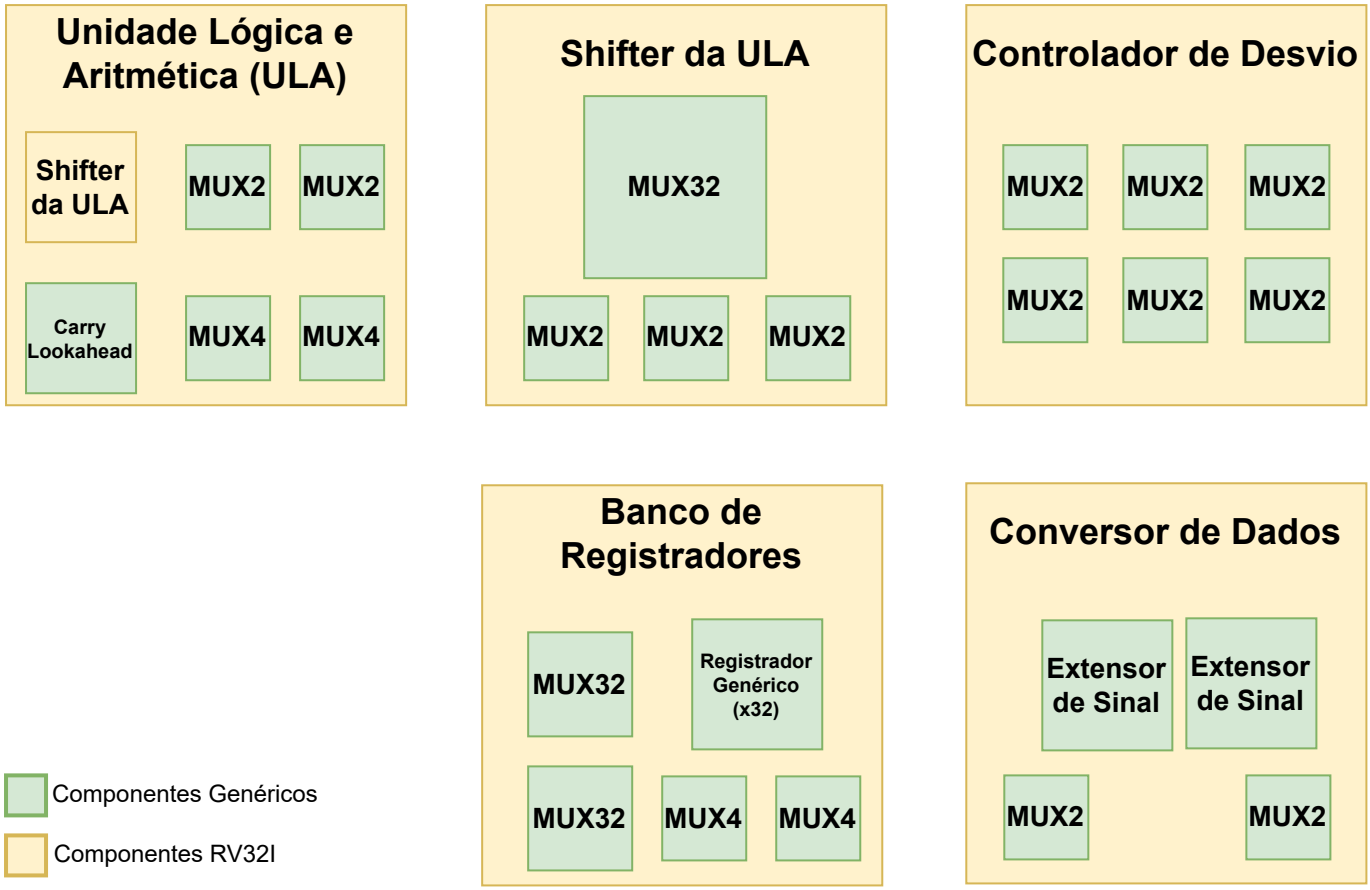


Figura 3 - Diagrama dos componentes RV32I implementados.

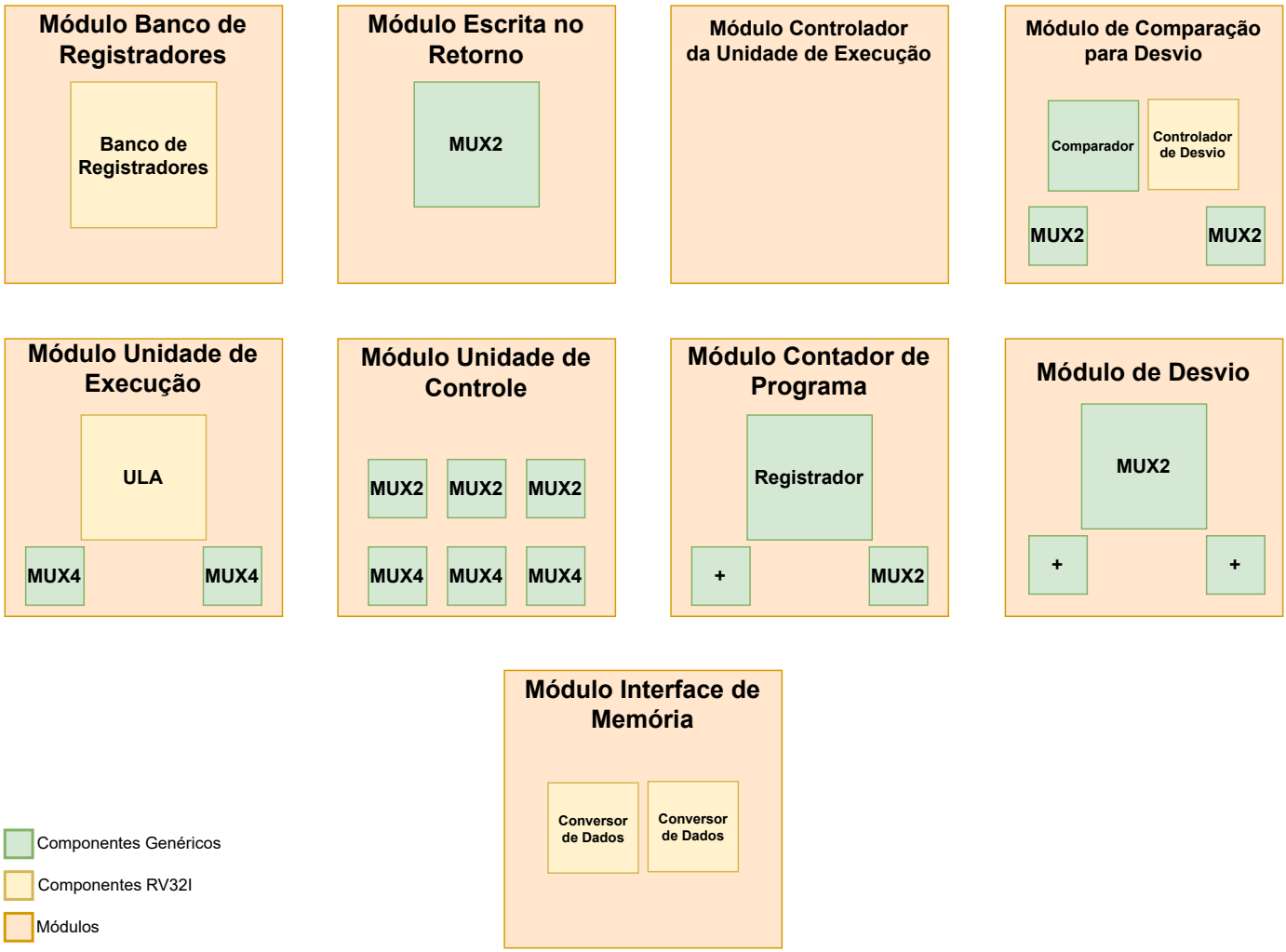


Figura 4 - Diagrama dos módulos implementados.

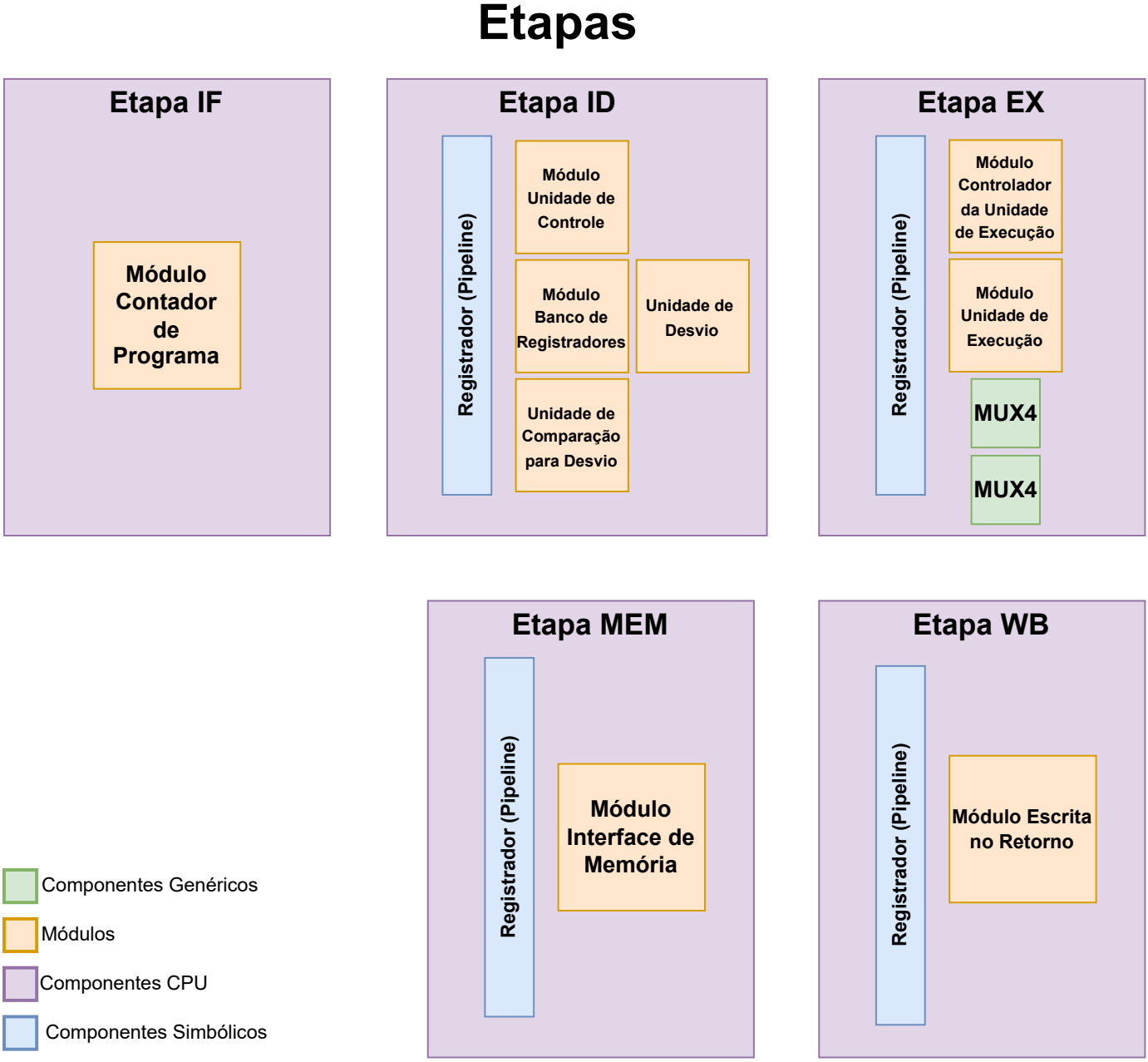


Figura 5 - Diagrama das etapas implementadas.

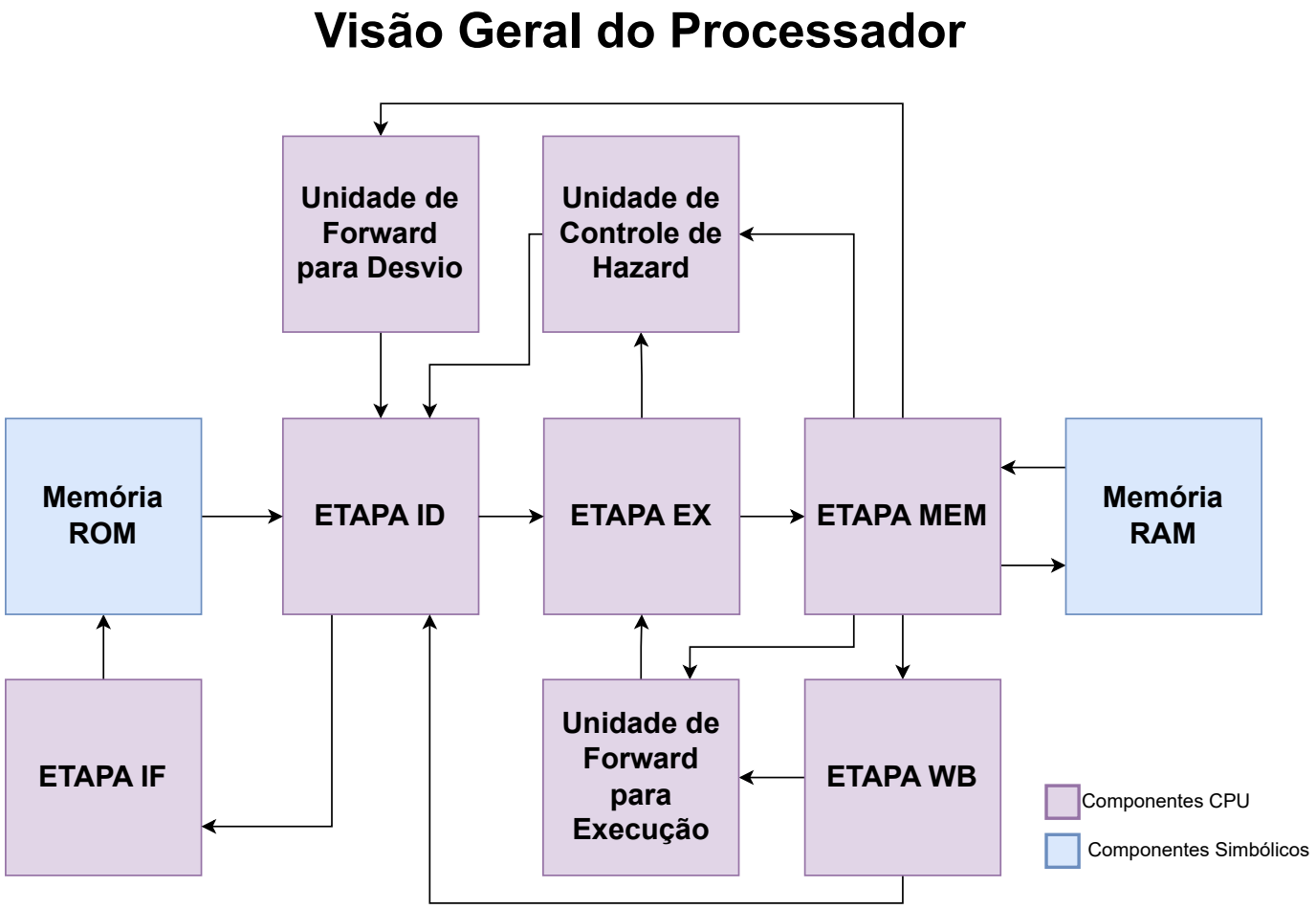


Figura 6 - Diagrama de Visão Geral da CPU.

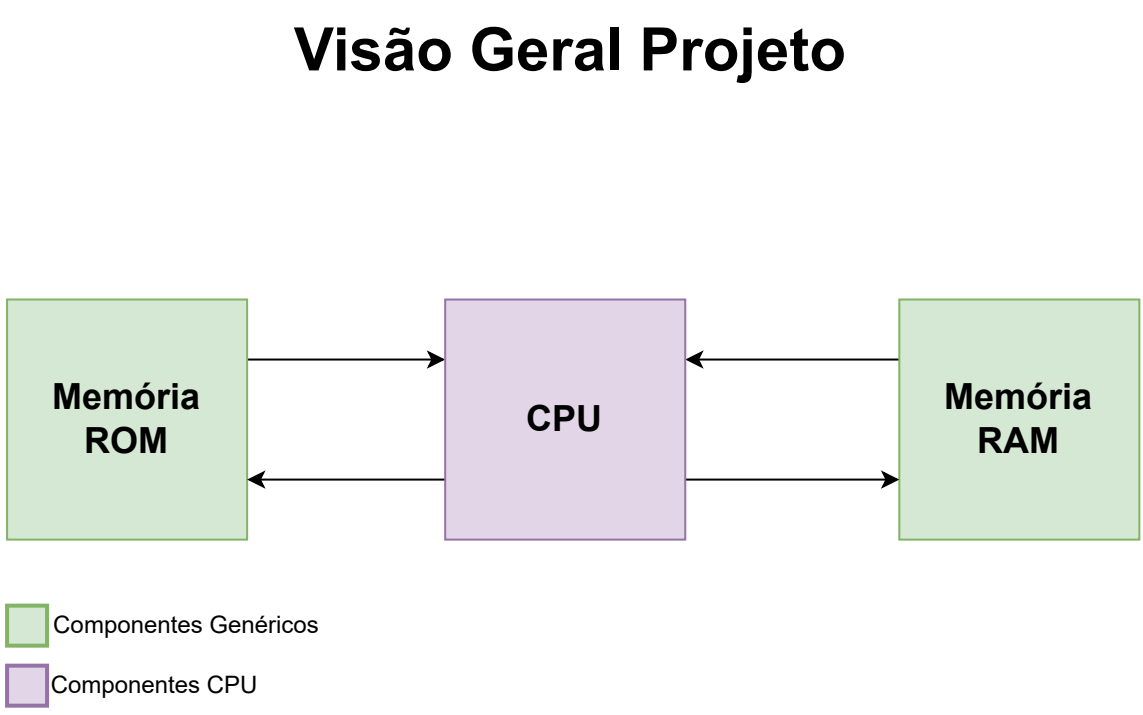


Figura 7 - Diagrama de Visão Geral do Projeto.

Já no que se refere à modularização, a mesma foi realizada seguindo a padronização explicitada no Apêndice A.

Para realizar a validação dos componentes foram utilizadas diferentes estratégias, utilizando as ferramentas: *Cocotb*, *Pytest*, *GHDL*, *Quartus*, *WaveDrom*, *Yosys*, *NetlistSVG*,*Docker* e *GitHub Actions*. Os testes podem ser realizados tanto no ambiente de desenvolvimento, feito no *Docker*, quanto em um ambiente na nuvem, desenvolvido no *GitHub Actions*.

Para os testes do *Quartus*, utilizado para o desenvolvimento na FPGA, foi criado através da linha de comando do próprio as seguintes etapas de compilação: *Synthesis*, *Elaboration*, *Fitter* e *Timing Analysis*. Essas etapas podem ser executadas tanto localmente, através de botões no *VSCode*, quanto no *GitHub Actions*, que executa automaticamente após um *push* no repositório do projeto na pasta */src*.

Com exceção do *Quartus*, os outros testes foram feitos utilizando a biblioteca *Pytest*, sendo separados os que são realizados utilizando o *Yosis* e o *Cocotb* em conjunto com o *GHDL*. O *Yosys* é utilizado para a síntese e para a geração do RTL que em conjunto com o *NetlistSVG* é possível visualizar o circuito gerado.

Enquanto o *GHDL* e o *Cocotb* são utilizados para a simulação dos componentes, realizando os testes de entrada e saída, e verificando se o comportamento do componente está correto, resultando em *PASS* ou *FAIL*. Também foi incluído o *WaveDrom* para visualizar o comportamento dos sinais em cada ciclo de clock.

O fluxo de como os testes são realizados pode ser visto na Figura 8, em que é possível observar em que momento cada teste é realizado e cada ferramenta é utilizada.

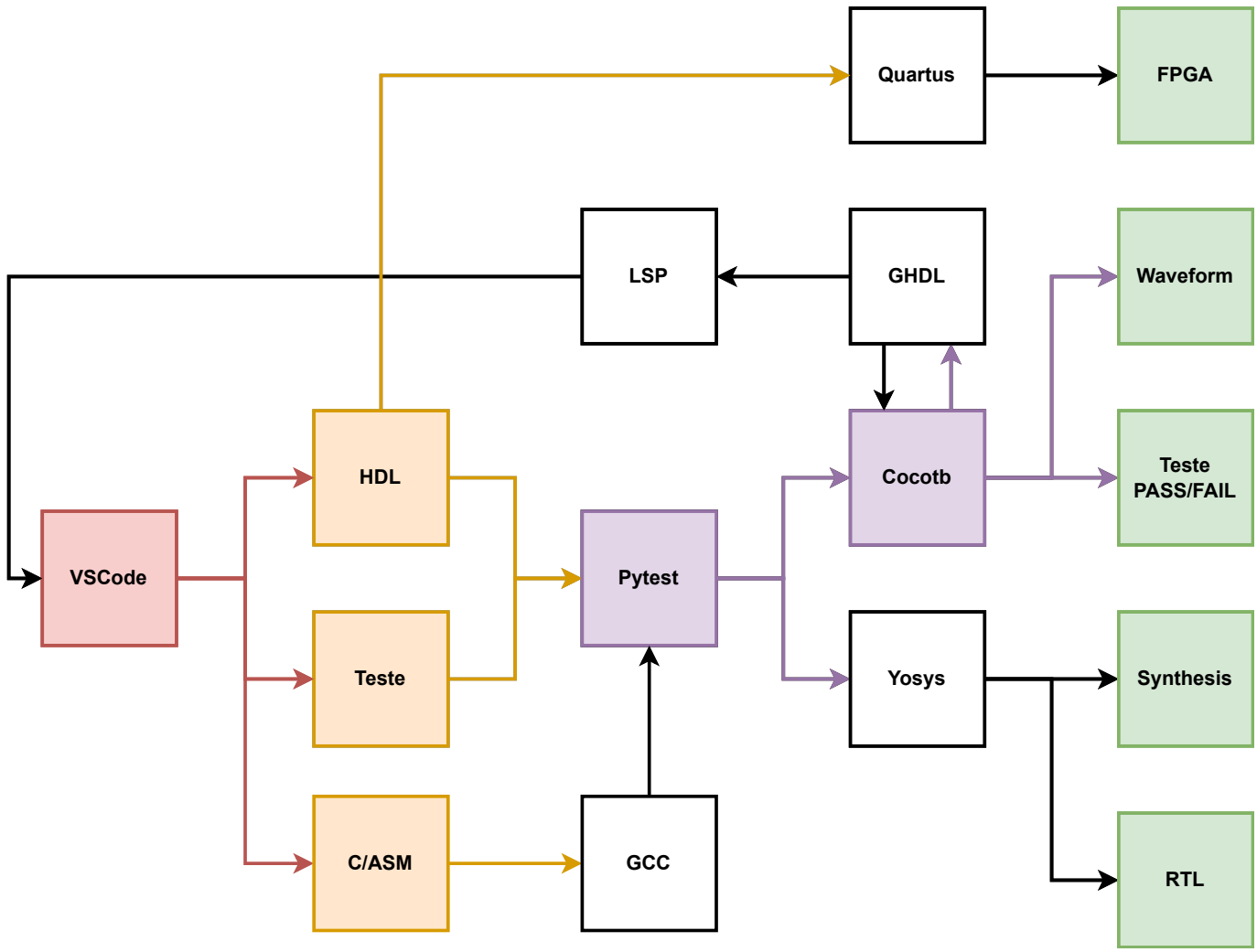


Figura 8 - Diagrama do fluxo de testes.

Para separar os testes executados pelo *Pytest*, foram utilizados markers(PYTEST, 2015), separando-os em testes de síntese, testes unitários e testes de stress. Os testes de síntese são realizados utilizando o *Yosys*, enquanto os testes unitários e de stress são realizados utilizando o *GHDL* e o *Cocotb*.

Nos testes unitários, ou casos de testes, são inseridos valores definidos manualmente na entrada do componente e posteriormente verificado se a saída corresponde ao esperado. Um exemplo de teste unitário para o componente Somador é mostrado no Código SOMADOR:

```python
...

@GENERIC_ADDER.testcase
async def tb_GENERIC_ADDER_case_1(dut: GENERIC_ADDER, trace: utils.Trace):
    dut.source_1.value = BinaryValue("00000000000000000000000000000000")
    dut.source_2.value = BinaryValue("00000000000000000000000000000000")

    await trace.cycle()
    yield trace.check(dut.destination, "00000000000000000000000000000000")

    dut.source_1.value = BinaryValue("00000000000000000000000000000000")
    dut.source_2.value = BinaryValue("00000000000000000000000000000001")

    await trace.cycle()
    yield trace.check(dut.destination, "00000000000000000000000000000001")

...
```

Código SOMADOR - Caso de teste do componente Somador

Ao analisar testes unitários, é possível verificar se os componentes estão funcionando corretamente para um conjunto limitado de entradas. Entretanto, ao olhar para o espaço de testes coberto, ele é limitado. Uma ilustração pode ser vista na Figura 9, em que todo o espaço de testes é representado por um quadrado com fundo branco, os testes cobertos por um quadrado com fundo cinza

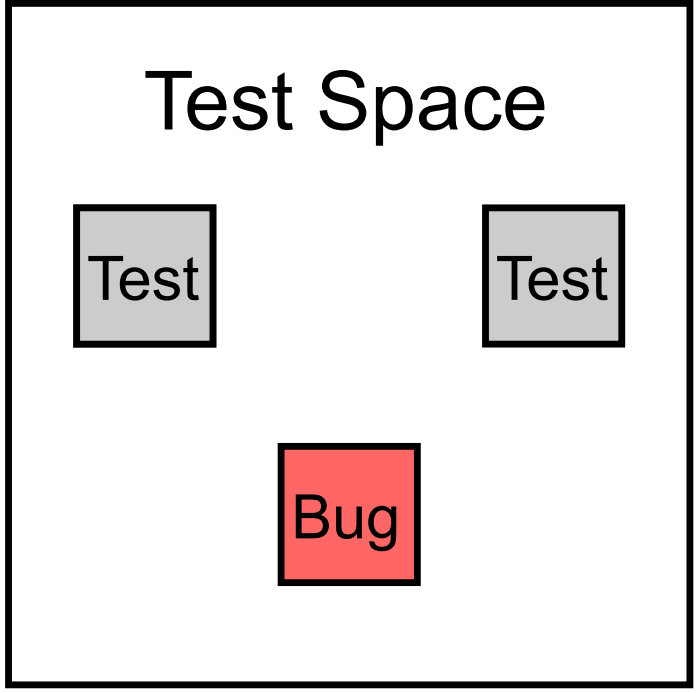


Figura 9 - Espaço de testes coberto por testes unitários.

Como foi solicitado e discutido com cliente, para aumentar a cobertura de testes, uma maneira é o uso de testes de stress, ou testes exaustivos. Eles consistem em testar o componente com todas as possibilidades, ou um número muito grande, de entradas possíveis. Como exemplificado na Figura 10, o espaço de testes coberto, representado por um quadrado com fundo cinza, é completo.

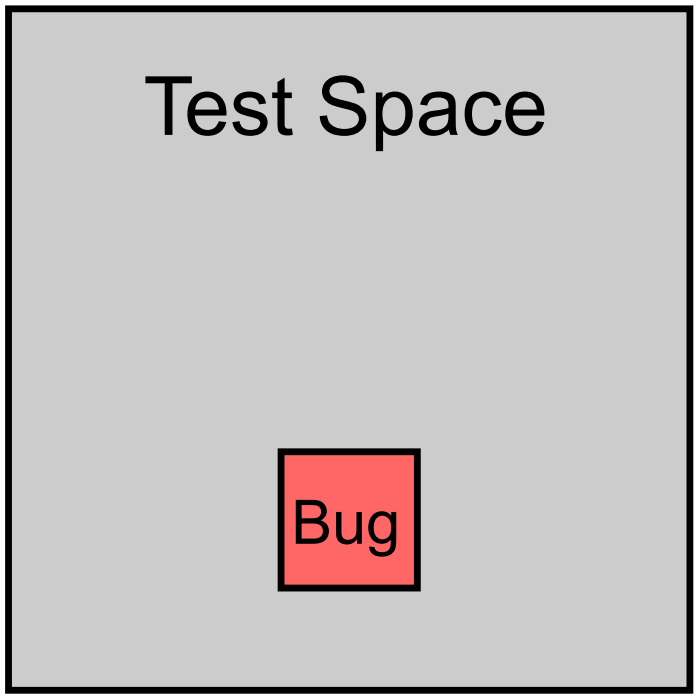


Figura 10 - Espaço de testes coberto por testes exaustivos.

Caso fosse realizar testes exaustivos para encontrar todas as possíveis entradas seria algo inviável para a largura de dados do projeto, que é de 32 *bits*, pois seria necessário multiplicar todas as possíveis entradas. Por exemplo, para um *MUX 2X1*, que possui 2 entradas de 32 *bits* e uma entrada de 1 *bit*, seriam necessários $2^{32} * 2^{32} * 2 = 2^{65}$ casos de testes. Caso cada teste leve em torno de 1 ms, levariam cerca de 1169884 séculos, o que torna inviável.

Para conseguir aumentar a chance de encontrar possíveis erros nos componentes, foi utilizada a estratégia Constraint Random Verification (CRV) (CHIPVERIFY, 2015). Ela consiste em criar um número grande de testes aleatorizados, com algumas restrições, para conseguir cobrir uma quantidade maior de cenários. Como os testes são aleatórios, quanto mais testes forem realizados, maior a chance de encontrar um erro. A Figura 11 ilustra o espaço de testes coberto pelo CRV, sendo representado por uma nuvem cinza, que se desloca pelo espaço de testes, encontrando possíveis erros.

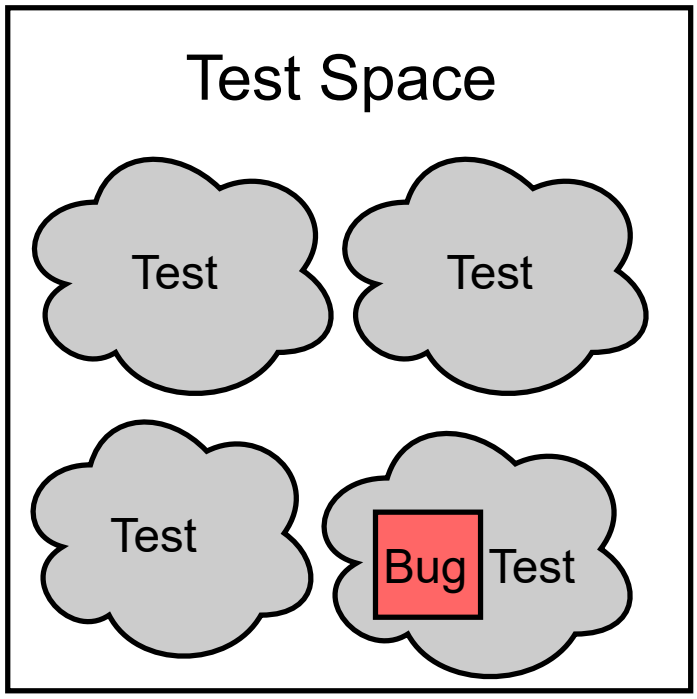


Figura 11 - Espaço de testes coberto pelo CRV.

A implementação do CRV foi feita utilizando um loop que executa uma certa quantidade de vezes, em conjunto com a biblioteca random, nativa do Python, que gera aleatoriamente os valores das entradas, em alguns casos para determinadas entradas, em outros para todas elas. Enquanto para a validação, o comportamento esperado do componente é simulado no *Python* para verificar se a saída corresponde. Um exemplo da implementação para o componente *ADDER* é mostrado no Código *Stress*.

```python
import random

...

@GENERIC_ADDER.testcase
async def tb_GENERIC_ADDER_stress(dut: "GENERIC_ADDER", trace: utils.Trace):
    for _ in range(1_000_000):
        source_1 = random.getrandbits(32)
        source_2 = random.getrandbits(32)

        dut.source_1.value = BinaryValue('{0:0{1}b}'.format(source_1, 32))
        dut.source_2.value = BinaryValue('{0:0{1}b}'.format(source_2, 32))

```

```
        await trace.cycle()

        message = f"source_1: {'{0:0{1}b}'.format(source_1, 32)}, source_2:
{'{0:0{1}b}'.format(source_2, 32)}"

        yield trace.check(dut.destination, '{0:0{1}b}'.format(source_1+source_2, 32)[-32:],
message)

...
```

Código *Stress* - Caso de teste de *stress* do componente Somador

Outra estratégia utilizada foi para validar a lógica do componente utilizando uma largura de dados menor. Por meio dessa estratégia é possível conseguir realizar um teste exaustivo que consegue verificar todas as entradas possíveis. Novamente, utilizando o *MUX 2x1* como exemplo, ao reduzir a largura de dados de 32 *bits* para 5 *bits*, é possível atingir $2^5 * 2^5 * 2 = 2^{11} = 2048$ possibilidades, um número viável de testes. A implementação foi feita criando loops aninhados para cada entrada e depois comparando a saída com a simulação do modelo em Python. Um exemplo para o componente *ADDER* é mostrado no Código *Stress Bits*.

python

```
...

@GENERIC_ADDER.testcase
async def tb_GENERIC_ADDER_stress_5_bits(dut: "GENERIC_ADDER", trace:
utils.Trace):
    bits = 5
    for i in range(2**bits):
        for j in range(2**bits):
            source_1 = '{0:0{1}b}'.format(i, bits)
            source_2 = '{0:0{1}b}'.format(j, bits)

            dut.source_1.value = BinaryValue(source_1)
            dut.source_2.value = BinaryValue(source_2)

            message = f"source_1: {source_1}, source_2: {source_2}"

            await trace.cycle()
            yield trace.check(dut.destination, '{0:0{1}b}'.format(i+j, bits)[-bits:], message)

...
```

Código *Stress Bits* - Caso de teste de *stress* de 5 *bits* do componente Somador

Assim, as duas estratégias de testes, CRV e redução da largura de dados, se complementam pois ao executar ambas é possível observar se o comportamento do componente muda conforme a largura de dados se altera.

2.1 Coleta de dados sobre o projeto

Os dados utilizados para realização deste projeto vieram de material, físico e digital, levantado pelos integrantes do projeto enquanto pesquisavam sobre o assunto, de conteúdo ensinado no curso de Engenharia de Computação, em específico na matéria de Design de Computadores lecionada por Paulo Carlos Ferreira dos Santos, de material que o orientador forneceu para ajudar na realização do projeto, de informações fornecidas pelo cliente, e de material que Rodrigo de Marca França forneceu.

No que se refere ao material levantado pelos integrantes, foi essencial o uso da documentação sobre a arquitetura de conjunto de instruções RISC-V (RISC-V INTERNATIONAL, 2019) fornecido pela RISC-V International, que é uma entidade sem fins lucrativos sediada na suíça, à qual o Brasil se juntou no final de fevereiro de 2024 (ver Seção 1.5) e que busca manter a neutralidade do desenvolvimento dessa arquitetura. Além disso, também foi encontrado um simulador *web* (MARIOTTI; GIORGI, 2022) de um processador RISC-V com conjunto base de instruções para inteiros com extensão para multiplicação que funciona para uma arquitetura de 32 *bits*, que é precisamente o foco para este projeto, e auxiliou para que o fosse possível visualizar como que a arquitetura seria implementada, e como o processador interage com diferentes instruções.

Já o conteúdo lecionado na matéria Design de Computadores, se refere ao desenvolvimento de um processador de arquitetura MIPS em VHDL usando o Quartus, que serviu como base de conhecimento para o desenvolvimento do projeto, já que foi experiência prática de desenvolvimento de um processador em VHDL, foco deste projeto, com uso da ferramenta Quartus, que foi necessária para validação deste projeto. Vale complementar que além do conhecimento adquirido nessa matéria, o curso de Engenharia da Computação também proporcionou conhecimento prático do uso de ferramentas como Docker, GitHub, GitHub Actions, Python e VSCode, o que resultou no estruturamento da metodologia de desenvolvimento técnico deste projeto.

Por sua vez, se tratando do material fornecido pelo orientador, o mais importante foi um repositório com componentes desenvolvidos em VHDL, onde os mesmos eram sintetizados usando GHDL, seus testes eram realizados usando Cocotb e Pytest, sendo que essas foram as ferramentas utilizadas no projeto para realizar os testes unitários e de integração de sistemas durante o projeto. Além disso, ele também forneceu livros que ajudaram no estudo da arquitetura e das suas instruções.

Já no que se refere às informações fornecidas pelos clientes, está a necessidade de modularização mencionada anteriormente na metodologia, assim como *feedback* sobre as decisões tomadas ao longo do processo de desenvolvimento.

Por fim, o material fornecido por Rodrigo de Marca França foi o material de referência da agência espacial europeia, a ESA (European Space Agency).

2.2 Análise dos Dados Coletados

Na documentação sobre a arquitetura fornecida pelos materiais utilizados estão detalhadas as instruções que compõem o conjunto base de instruções para inteiros de 32 *bits*, sua estrutura e como elas funcionam, e o mesmo se aplica às instruções que compõem a extensão de multiplicação do processador (olhar Anexo A para mais informações sobre os conjuntos de instruções).

As instruções do conjunto base de inteiros são 40 no total, porém 3 destas instruções não foram implementadas (ECALL, EBREAK e FENCE) por elas serem de uso de processadores superescalares, que não faz parte do escopo. O conjunto de instruções da extensão de multiplicação serão implementadas se for factível a depender do tempo que sobrar após o término do desenvolvimento do processador funcional com as instruções do conjunto base implementadas, como acordado com o cliente.

Baseado nos conhecimentos adquiridos no curso de Engenharia de Computação, decidiu-se usar o GitHub Actions para automatizar o processo de testes CI/CD solicitados pelo cliente, e o Docker para padronizar o sistema operacional de desenvolvimento. Com o feedback positivo do cliente, essa metodologia de desenvolvimento se tornou parte essencial da entrega do projeto.

3. Resultados

O ambiente de teste foi desenvolvido para ser realizado tanto no GitHub Actions quanto no ambiente de desenvolvimento criado com o Docker e utilizado no VSCode. Enquanto no ambiente de desenvolvimento todas as ferramentas necessárias estão instaladas no container Docker, no GitHub Actions foi possível separar em dois ambientes diferentes: um para o *Quartus* e outro para os testes de síntese, unitários e stress.

No ambiente do *Quartus* foi utilizado o mesmo container Docker do ambiente de desenvolvimento, enquanto para os outros testes foi feita a instalação somente das ferramentas necessárias, sem o *Quartus* que é a ferramenta que ocupa mais espaço.

No Github Actions foi desenvolvido um ambiente de testes para a verificação de cada módulo VHDL. Através dele é possível realizar os testes de sínteses, unitários, stress e a compilação do *Quartus* com as etapas de *synthesis*, *elaboration*, *fitter* e *timing analysis*. Sendo possível ver os *logs* e as etapas, na página do GitHub Actions, como mostra a Figura 12.

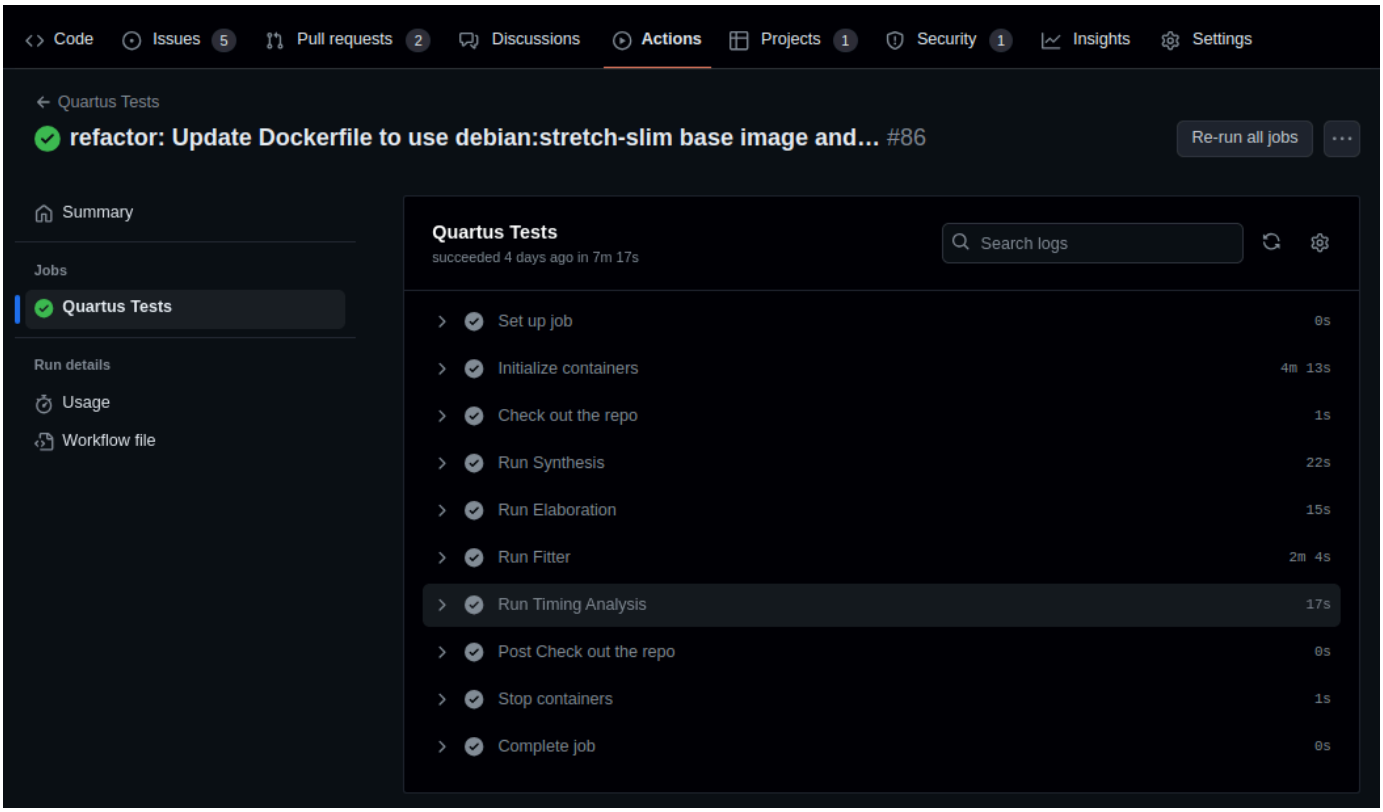


Figura 12 - Logs Quartus.

Os testes unitários e os testes de stress são realizados separadamente, mas em conjunto com o teste de síntese, sendo que o teste de síntese é realizado antes deles. Os testes unitários são realizados após cada *push* no repositório, na pasta */src*, sendo possível ver os *logs* e as etapas, na página do GitHub Actions, como mostra a Figura 13.

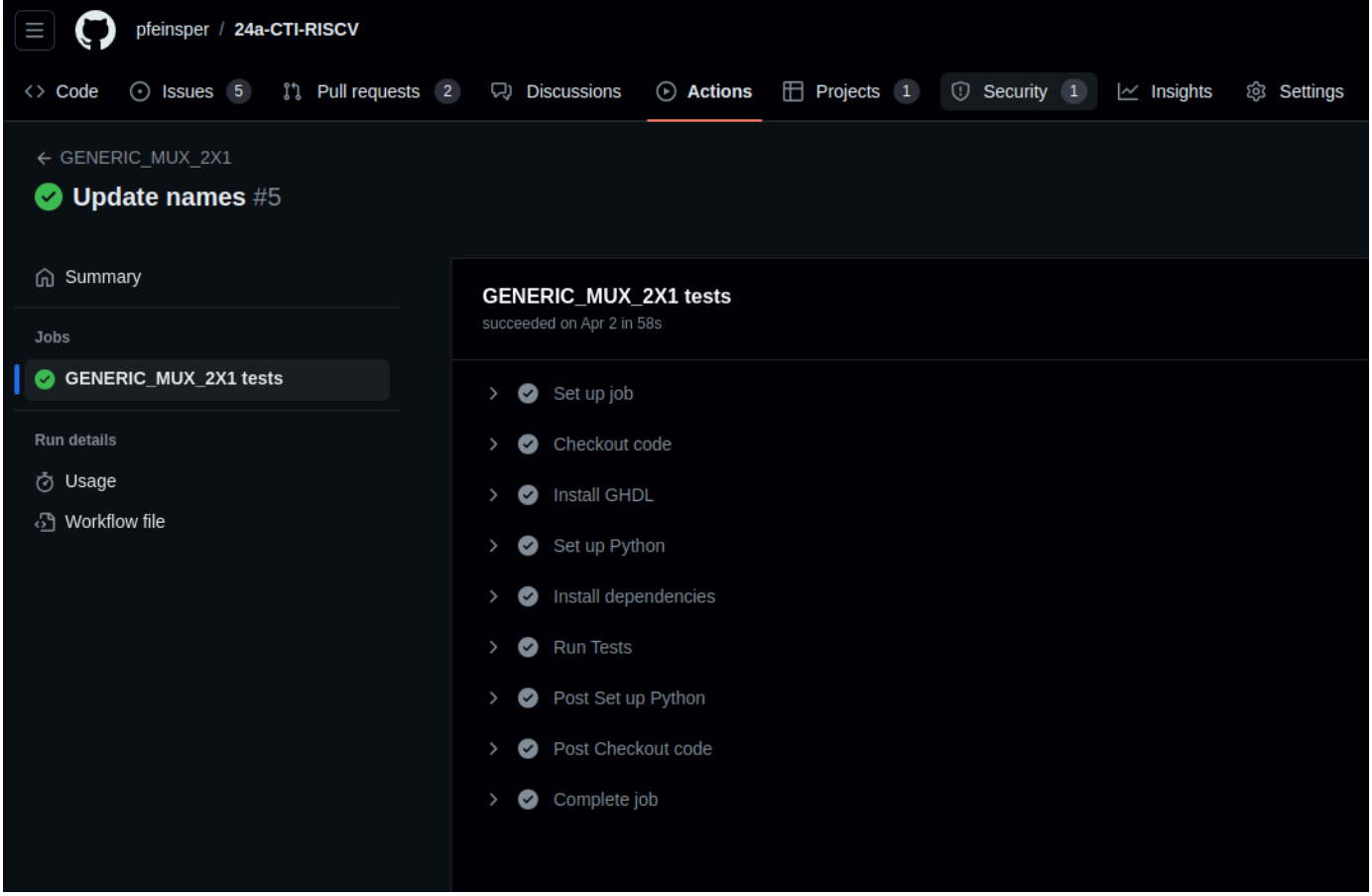


Figura 13 - Casos de testes no Github Actions.

Enquanto os testes de stress são realizados somente ao clicar no botão *Run workflow* na página do GitHub Actions, como mostra a Figura 14

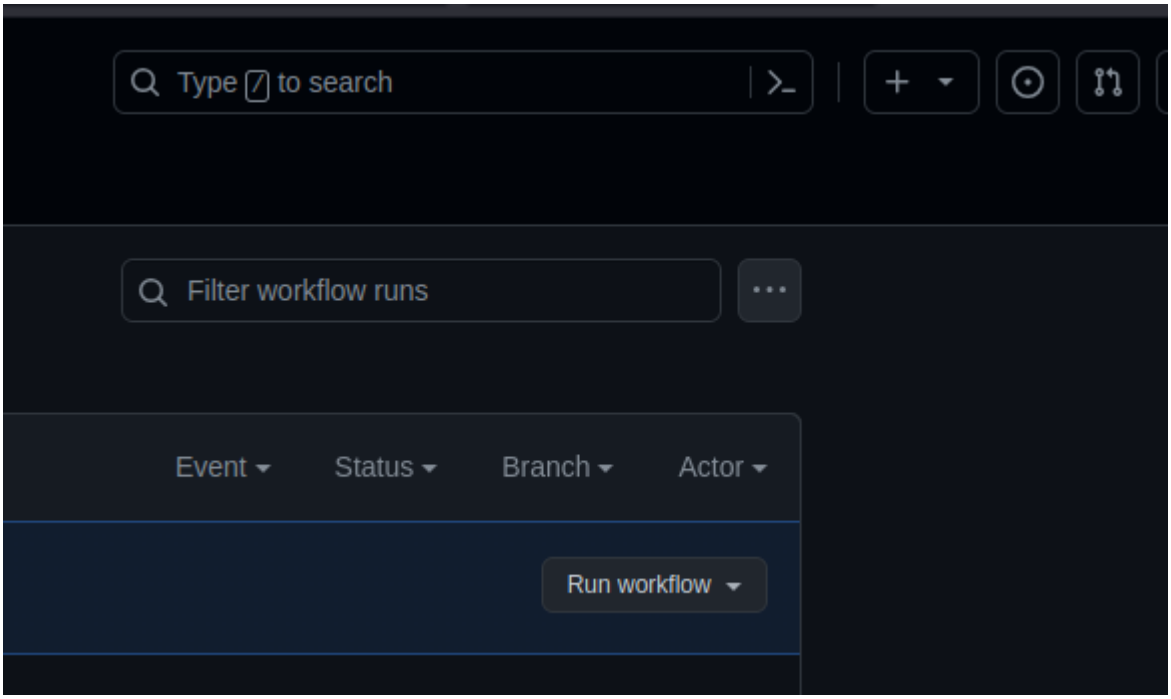


Figura 14 - Botão para a execução dos testes de stress.

No ambiente de desenvolvimento é possível executar os testes de síntese, unitários e stress, através do menu lateral de teste do VSCode, como mostra a Figura 15.

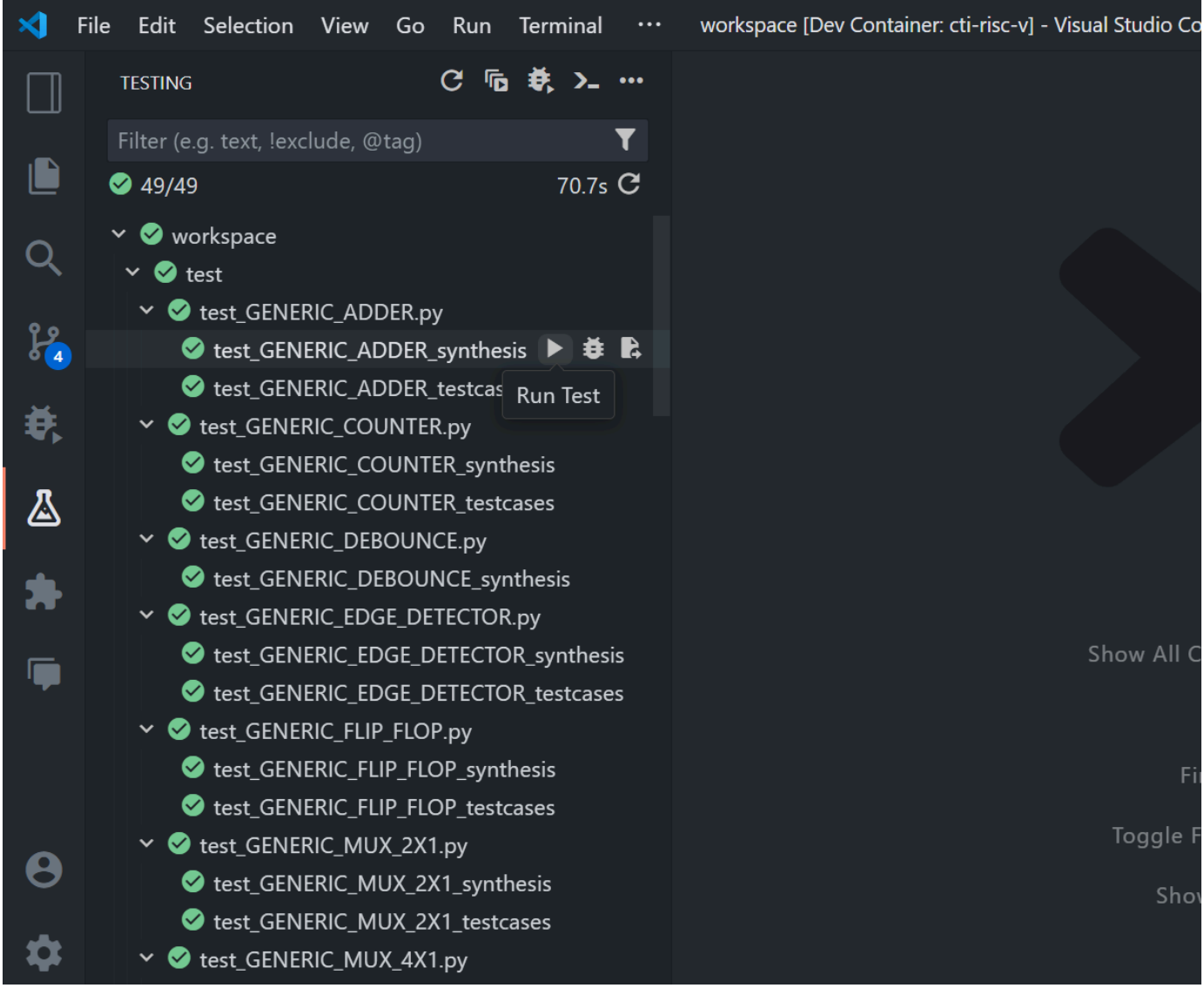


Figura 15 - Menu de testes do VSCode.

No ambiente de desenvolvimento, também é possível executar os testes do *Quartus* através de botões localizados na parte inferior da tela, bem como é possível abrir a interface gráfica do *Quartus* , como mostra a Figura 16.

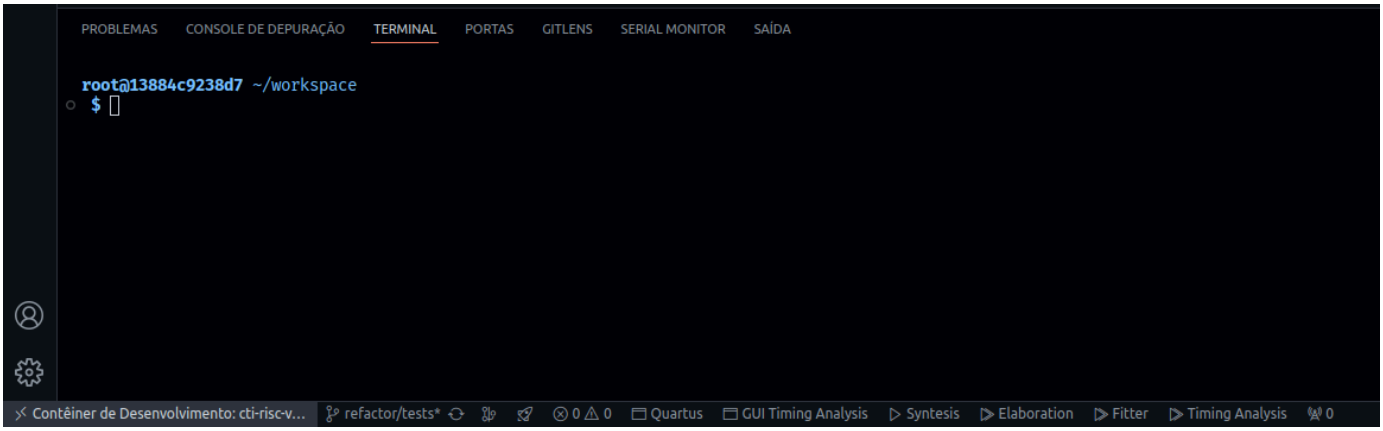
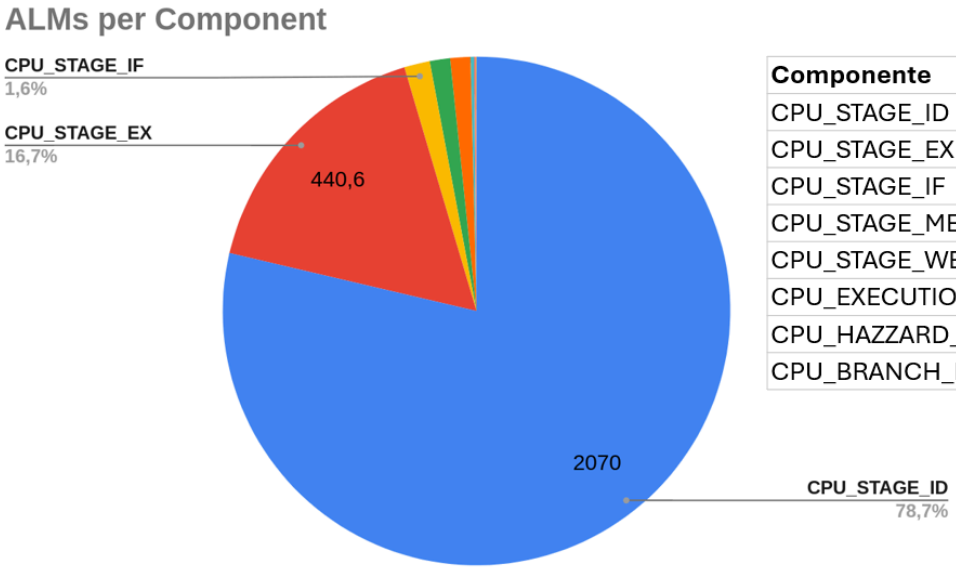


Figura 16 - Menu de testes do VSCode.

Através do Quartus foi possível obter a frequência máxima do processador que ficou em 67.77 MHz, com uma área de 2985 ALMs, ocupando 16% da FPGA utilizada no projeto, que é a Cyclone V 5CEBA4F23C7N. Para efeito de comparação o projeto *VexRiscv* (SPINALHDL, 2024), um projeto de RISC-V desenvolvido para FPGA, que foi executado na Cyclone V e foi feito utilizando *SpinalHDL*, a sua

versão com a menor frequência máxima possui 124 MHz, enquanto a com maior 194 MHz. Enquanto para a área, a versão com menos ALMs possui 352 ALMs e a versão com mais possui 1764 ALMs.

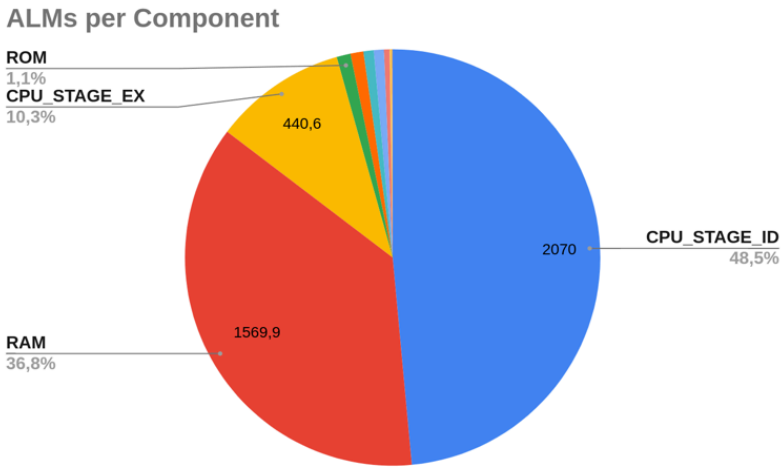
Ao analisar o consumo de ALMs por etapa do *pipeline* foi encontrado o grágico da Figura 17, em que a etapa que mais consome ALMs é a etapa ID, pois é onde está localizado o banco de registradores.



Componente	ALMs
CPU_STAGE_ID	2070
CPU_STAGE_EX	440,6
CPU_STAGE_IF	42,9
CPU_STAGE_MEM	34,1
CPU_STAGE_WB	33,8
CPU_EXECUTION_FORWARDING_UNIT	5,7
CPU_HAZZARD_CONTROL_UNIT	3.7
CPU_BRANCH_FORWARDING_UNIT	3,2

Figura 17 - Consumo de ALMs por etapa do pipeline.

Ao incluir as memórias RAM e ROM, a etapa ID continua sendo a que mais consome ALMs, porém a memória RAM passa a ser a segunda que mais consome ALMs, como mostra a Figura 18.



Componente	ALMs
CPU_STAGE_ID	2070
RAM	1569,9
CPU_STAGE_EX	440,6
ROM	46
CPU_STAGE_IF	42,9
CPU_STAGE_MEM	34,1
CPU_STAGE_WB	33,8
LOW_FREQ	18,5
CPU_EXECUTION_FORWARDING_UNIT	5,7
CPU_HAZZARD_CONTROL_UNIT	3.7
CPU_BRANCH_FORWARDING_UNIT	3,2
UPDATE_LED	1,1

Figura 18 - Consumo de ALMs por etapa do pipeline com memórias RAM e ROM.

Os testes do *Quartus* são realizados somente a partir do *Top Level* do projeto, sendo que os testes de síntese e unitários a partir de todos os módulos. Já os testes de stress foram criados para os seguintes módulos:

- *Generic Adder*
- *Generic Mux 2x1*
- *Generic Mux 4x1*
- *Generic Register*

- *RV32I ALU*

Para que o processador possa ser programado em uma linguagem de alto nível, foi utilizado o *gcc* para compilar o código em C, e em conjunto com o *Python* foi possível transformar o código em binário e gerar o arquivo *.mif* que é utilizado para programar a memória ROM.

O teste na FPGA feito para demonstrar o funcionamento do processador, foi adicionado um registrador conectado aos LEDs da placa, que é atualizado quando ocorre a escrita em um determinado endereço de memória. A implementação é demonstrada na Figura 19, em que *update_led* corresponde ao registrador que atualiza os LEDs da placa.

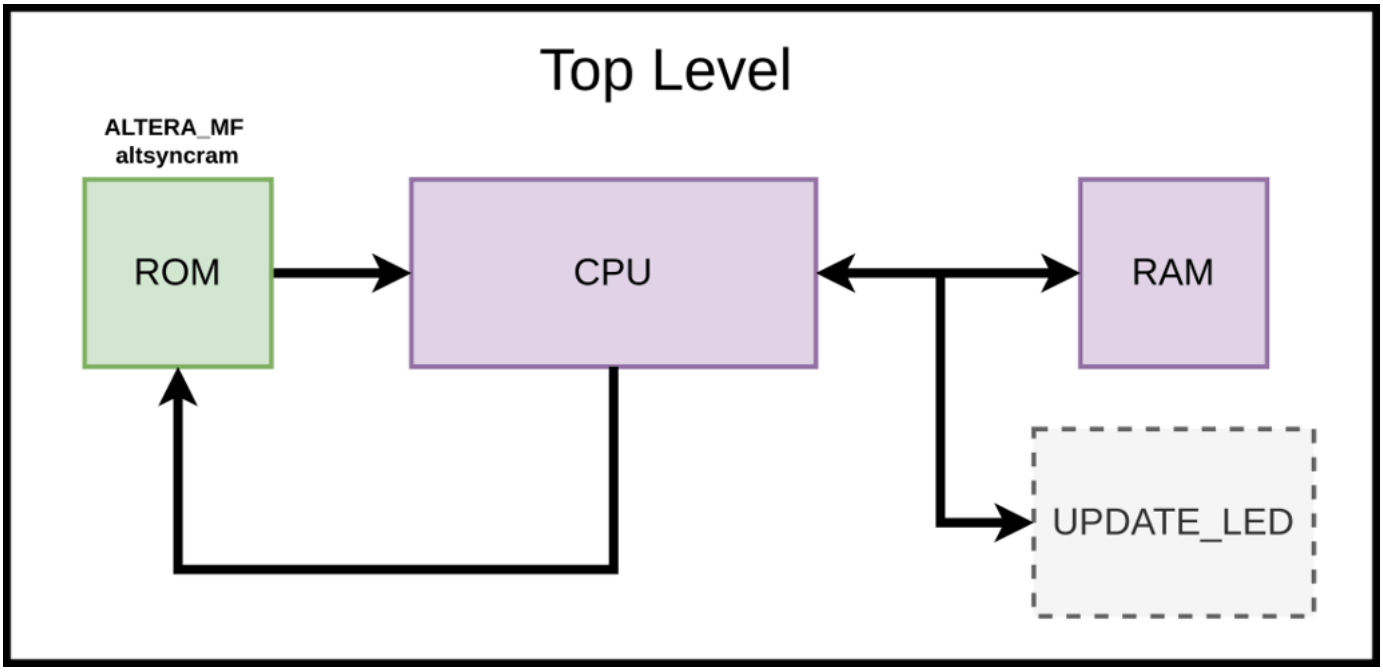


Figura 19 - Diagrama de Blocos da FPGA

Para a demonstração na FPGA foi feita a implementação do seguinte código em C:

```c

#include "base.h"

void main() {
    while (1) {
        digitalWrite(LEDR, 1);
        ASM("NOP");
        ASM("NOP");
        ASM("NOP");
        digitalWrite(LEDR, 0);
    }
}
```

E a biblioteca *base.h* foi implementada da seguinte forma:

```c

#ifndef __BASE_H__
#define __BASE_H__

#define LEDR                    128
#define sleep(cycles)            for (long i = 0; i < (cycles / 2); i++) asm("");
#define digitalWrite(pin, value)    (*((volatile int *)pin) = value)


void main() __attribute__((noreturn));

#endif
```

No código em C da demonstração foi utilizado a função *digitalWrite* que é responsável por escrever em um endereço de memória da RAM que está conectado aos LEDs da placa. Nesse caso foi utilizado o endereço 128. Foram acrescentados três instruções *NOP*, pois ao executar a instrução que retorna ao início do loop, não acontece instantaneamente, afinal tem a execução da instrução em si, o *flush* do pipeline e a busca da próxima instrução.

Ao testar na FPGA, foi possível identificar alguns problemas. Foi necessário acrescentar um registrador na etapa ID, pois para a FPGA é utilizada uma memória com *clock*, que fazia com que ao realizar o salto, não fosse para o endereço de memória correto. Também foi necessário utilizar um *clock* reduzido de 1 Hz para que fosse possível visualizar o funcionamento do processador, pois com o *clock* de 50 MHz, o processador executava as instruções muito rapidamente, não sendo possível visualizar o funcionamento.

Na Figura 20, é possível observar a FPGA, em que a palavra *RISC-V* é exibida no painel de sete segmentos, o *LED 09*, que fica mais a esquerda, é utilizado para indicar o *clock* do processador, e o *LED 00* é conectado ao registrador que atualiza ao escrever no endereço 128 da memória RAM.

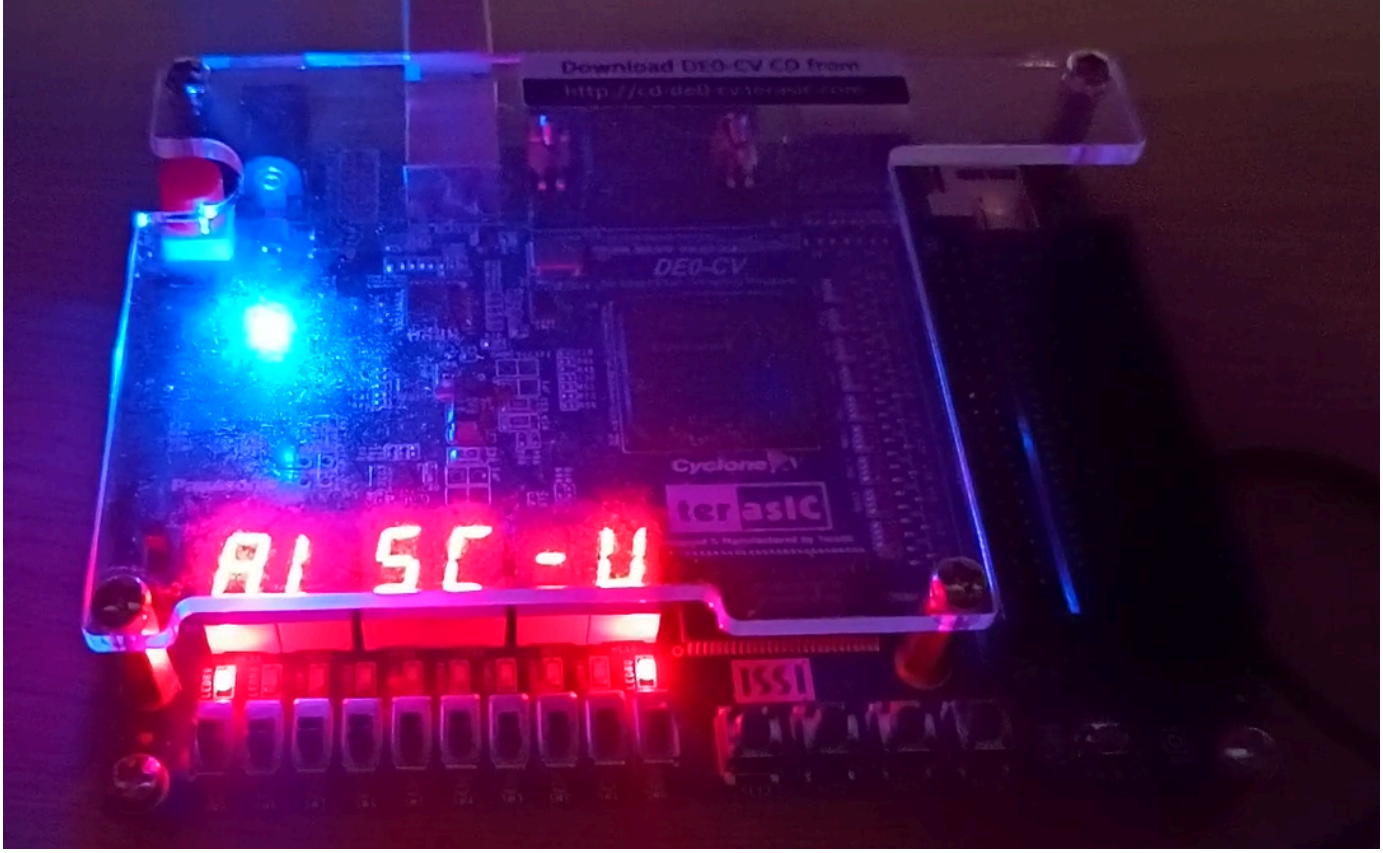


Figura 20 - FPGA

Também foi realizada a documentação do projeto em um website cujo link está nas referências (DIAS, L. F.; RUGGIERO, G. V.; SEIXAS, T. V., 2024) e que está ilustrado na Figura 21, hospedado no GitHub Pages utilizando de VitePress, uma ferramenta que gera sites estáticos por meio de arquivos Markdown.

Figura 21 - Página inicial do site de documentação (DIAS, L. F.; RUGGIERO, G. V.; SEIXAS, T. V., 2024)

Os recursos desenvolvidos permitem que testes sejam realizados ao longo do processo de desenvolvimento, garantindo que os componentes funcionem de maneira adequada e trazem uma facilidade para o desenvolvedor, que não precisa instalar todas as ferramentas manualmente, nem executar

comandos no terminal. Foi integrado o processador com pipeline (Figura 1, Escopo do Projeto), com a implementação dos componentes genéricos (ver Seção 3.3), dos componentes RV32I (ver Seção 3.2), módulos (ver Seção 3.3) e etapas (ver Seção 3.4) sendo realizada, assim como a validação dos componentes e módulos por meio de testes automatizados, e a validação da integração por meio de testes das instruções do Conjunto Base dos Inteiros para arquiteturas de 32 *bits*.

3.1 Nível dos Componentes Genéricos

O nível que contém os elementos do processador que se encontram na base da hierarquia (ver Seção 2). Eles estão classificados como componentes genéricos, que são componentes que poderiam ser aplicados em outros projetos sem modificação devido ao fato de serem genéricos.

3.1.1 *Carry Lookahead* (Genérico)

Componente que contém a lógica necessária para realizar soma *bit* a *bit* entre dois vetores de dados. Esse componente está ilustrado na Figura 22

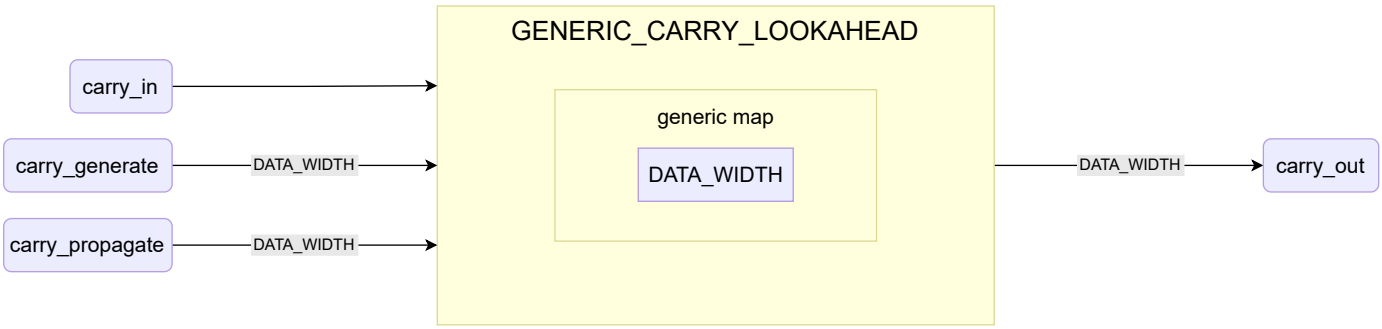


Figura 22 - Topologia do Componente Genérico *Carry Lookahead*

3.1.2 Somador (Genérico)

Componente que recebe dois vetores de entrada de múltiplos *bits*, quantidade de *bits* ajustável de acordo com a necessidade, e soma esses dois valores usando do componente genérico *carry lookahead*, devolvendo na saída o resultado. Esse componente está ilustrado na Figura 23.

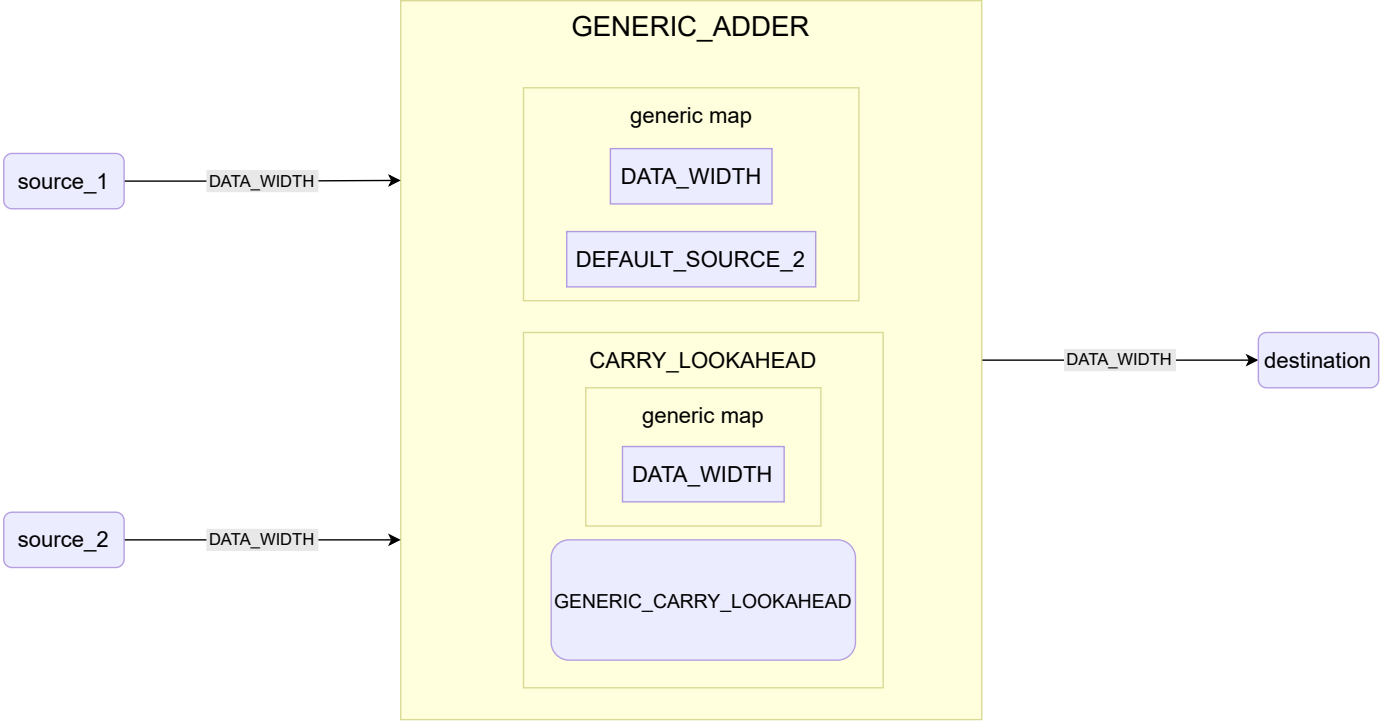


Figura 23 - Topologia do Componente Genérico Somador

3.1.3 Multiplexador de duas entradas (Genérico)

Componente que recebe dois vetores de entrada de múltiplos *bits*, quantidade de *bits* ajustável de acordo com a necessidade, e recebe um seletor de 1 *bit*, e dependendo do valor do seletor, um dos valores de entrada passa a ser valor de saída. Esse componente está ilustrado na Figura 24.

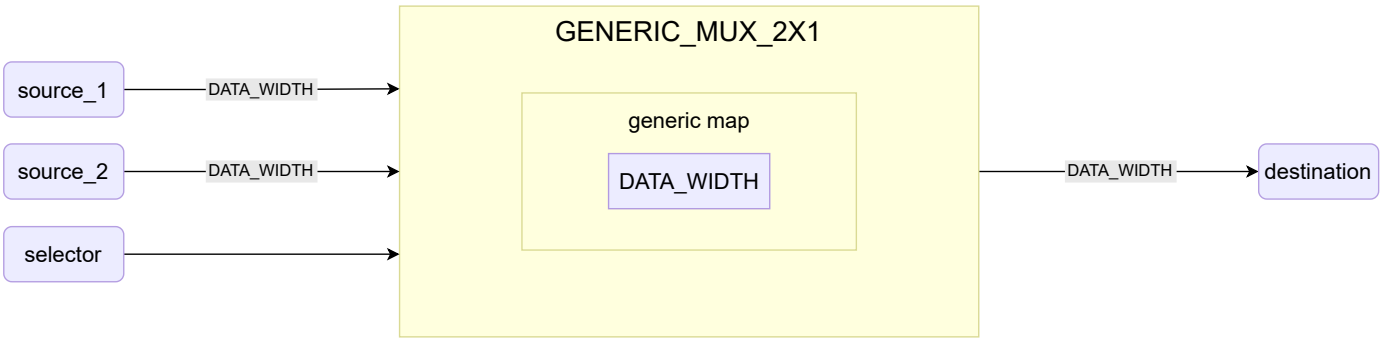


Figura 24 - Topologia do Componente Genérico Multiplexador de duas entradas

3.1.4 Multiplexador de quatro entradas (Genérico)

Componente que recebe quatro vetores de entrada de múltiplos *bits*, quantidade de *bits* ajustável de acordo com a necessidade, e recebe um seletor de 2 *bits*, e dependendo do valor do seletor, um dos valores de entrada passa a ser valor de saída. Esse componente está ilustrado na Figura 25.

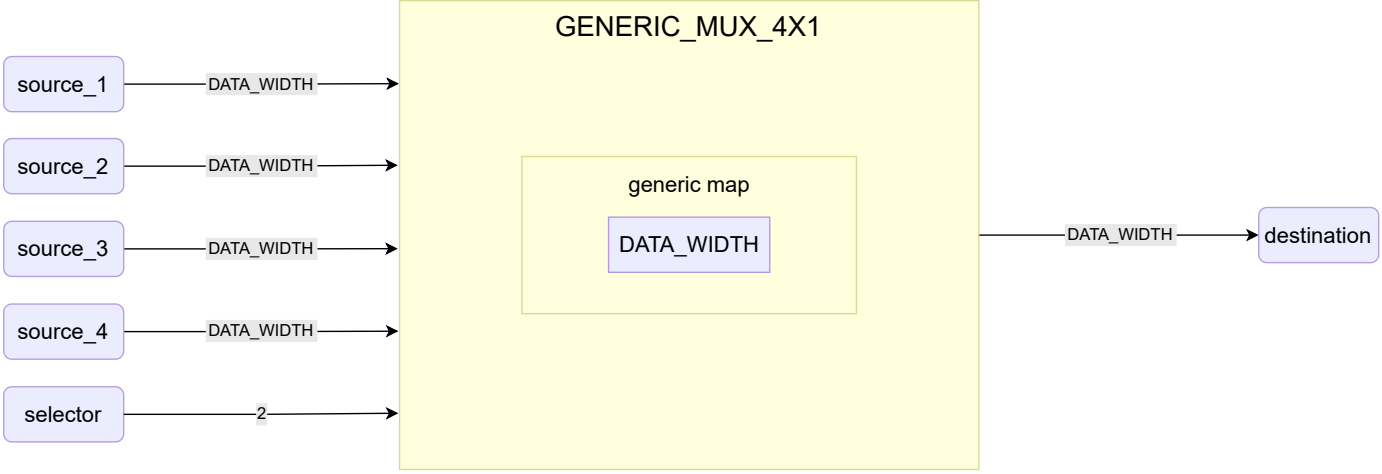


Figura 25 - Topologia do Componente Genérico Multiplexador de quatro entradas

3.1.5 Multiplexador de trinta e duas entradas (Genérico)

Componente que recebe trinta e dois vetores de entrada de múltiplos *bits*, quantidade de *bits* ajustável de acordo com a necessidade, e recebe um seletor de 5 *bits*, e dependendo do valor do seletor, um dos valores de entrada passa a ser valor de saída. Esse componente está ilustrado na Figura 26.

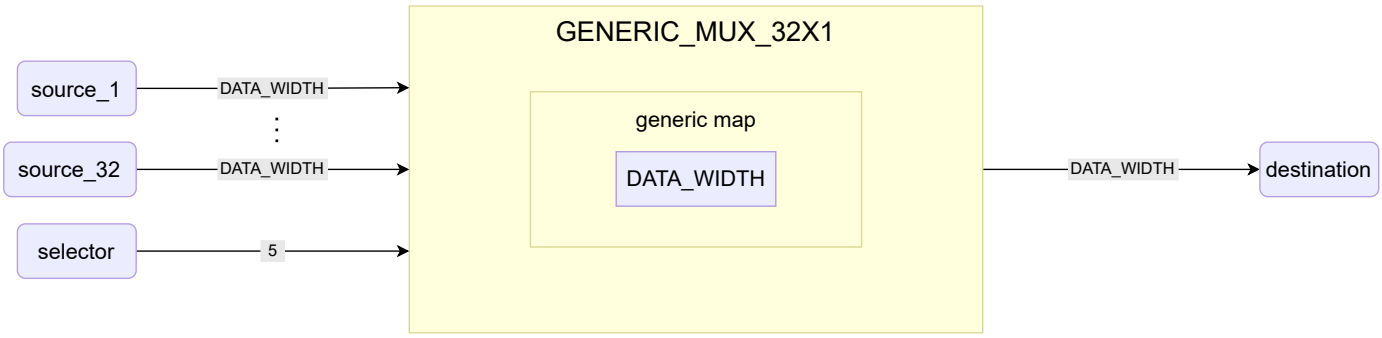


Figura 26 - Topologia do Componente Genérico Multiplexador de trinta e duas entradas

3.1.6 Memória ROM (Genérico)

Componente onde ficam armazenadas as instruções do programa a ser rodado no processador, recebe um vetor de endereço de múltiplos *bits*, e devolve na saída a instrução de múltiplos *bits* armazenada nesse endereço. Esse componente está ilustrado na Figura 27.

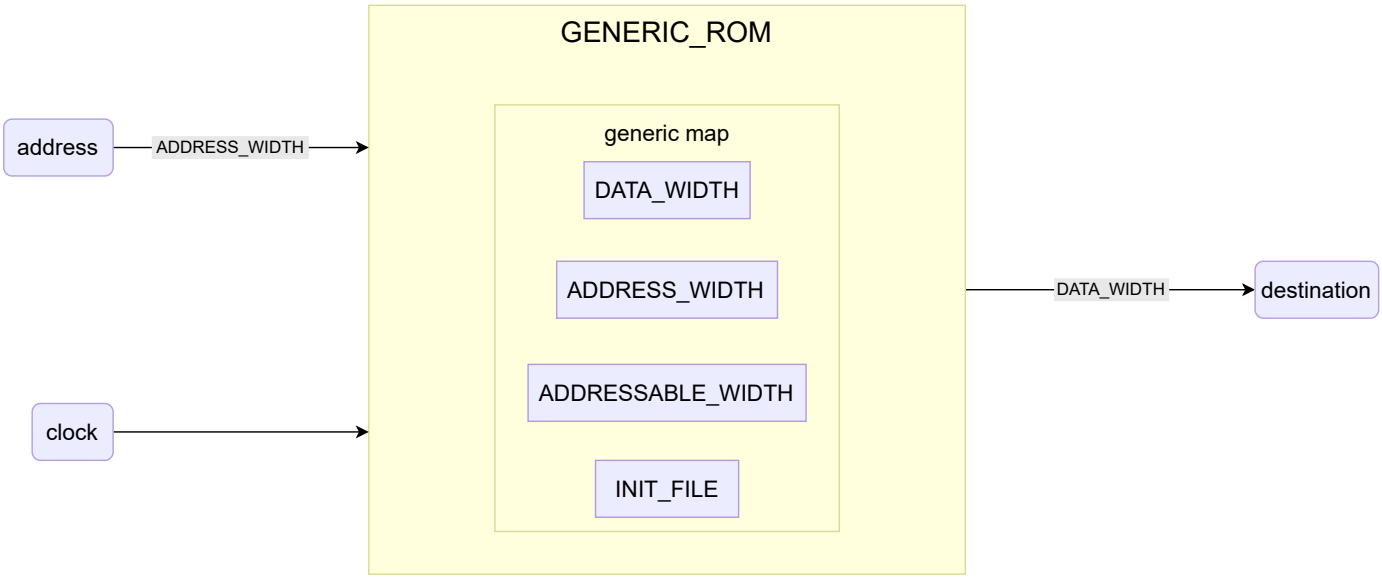


Figura 27 - Topologia do Componente Genérico Memória ROM

A memória ROM foi desenvolvida apenas para propósitos de validação da integração do processador, pois os integrantes da CTI mencionaram que usariam uma memória externa.

3.1.7 Memória ROM Quartus (Genérico)

Memória ROM desenvolvida para testes na FPGA, topologia da Figura 27 também se aplica.

3.1.8 Memória RAM (Genérico)

Componente que funciona com clock e depende de um sinal de ativação, armazena valores de múltiplos *bits* recebidos na entrada, que é escrito na posição da memória apontada pelo endereço de múltiplos *bits* recebido, dependendo do sinal de escrita, e devolve na saída o valor armazenado no mesmo endereço, dependendo do sinal de leitura. Esse componente está ilustrado na Figura 28.

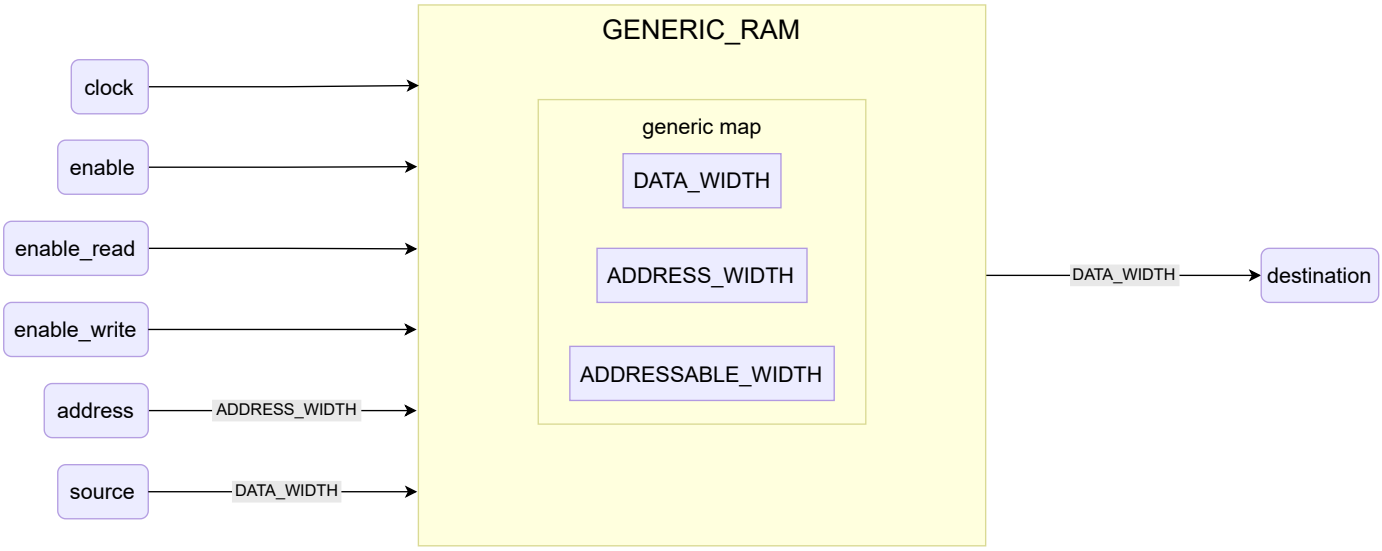


Figura 28 - Topologia do Componente Genérico Memória RAM

3.1.9 Registrador (Genérico)

Componente que funciona com clock e armazena um vetor de múltiplos *bits*, que é recebido no vetor de entrada, de acordo com um sinal de ativação do registrador, devolvendo esse valor na saída, e caso o sinal de limpar esteja ativado, devolve um vetor de múltiplos zeros. Esse componente está ilustrado na Figura 29.

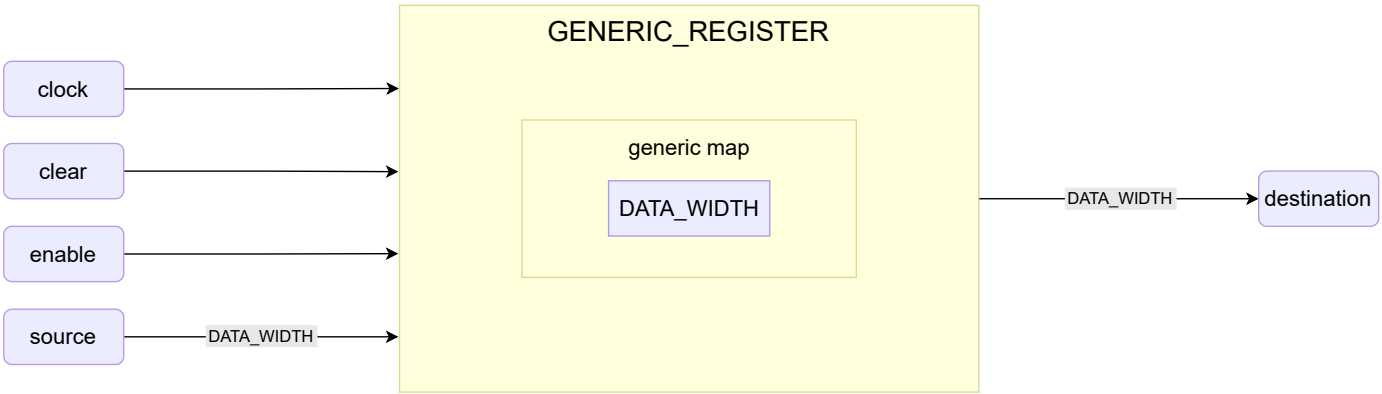


Figura 29 - Topologia do Componente Genérico Registrador

3.1.10 Comparador (Genérico)

Componente que recebe dois vetores de 32 *bit* e devolve três flags, uma que indica se ambos os vetores são iguais, outra que indica se o vetor de entrada primário é menor do que o secundário, e a última indica se o vetor primário é maior ou igual ao secundário. Esse componente está ilustrado na Figura 30.

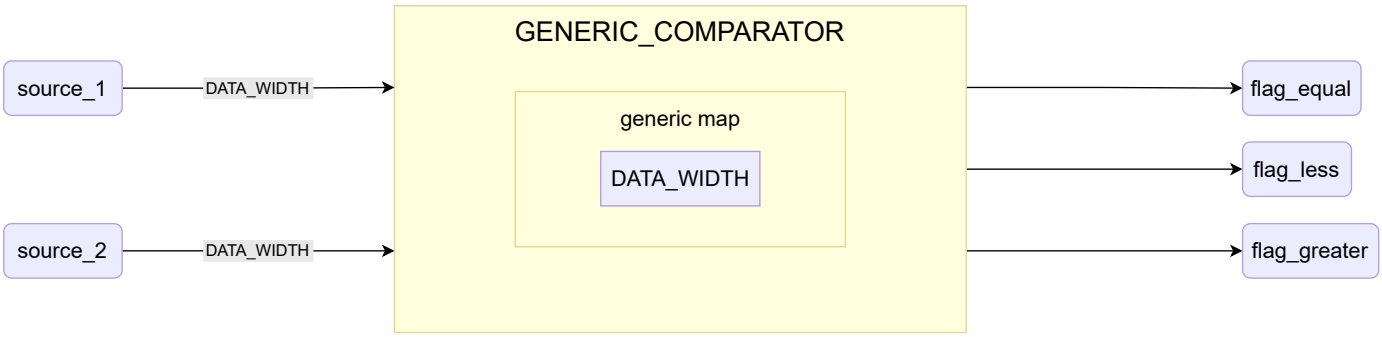


Figura 30 - Topologia do Componente Genérico Comparador

3.1.11 Extensor de Sinal (Genérico)

Componente que extende o sinal de um vetor de dados de 32 *bits*, ilustrado na Figura 31.

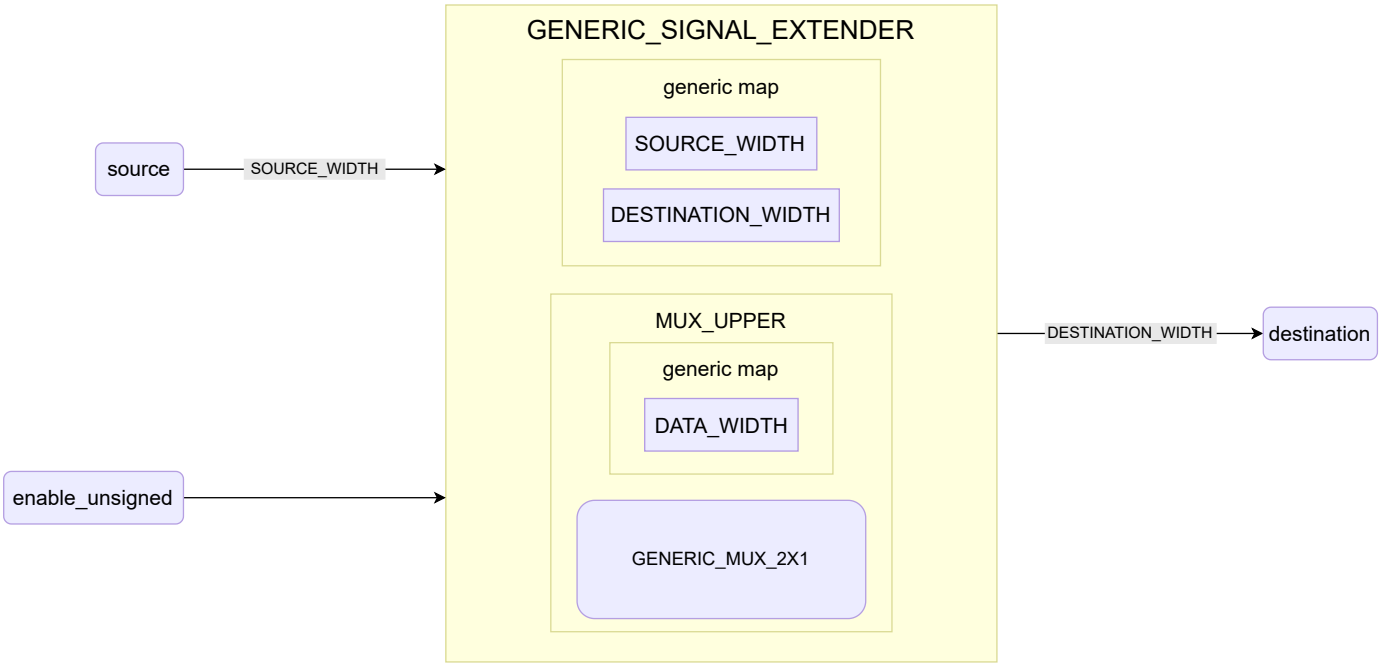


Figura 31 - Topologia do Componente Genérico Comparador

3.2 Nível dos Componentes RV32I

O segundo nível mais baixo do processador, contém componentes que implementam uma lógica para atender às especificações de um processador de arquitetura RV32I.

3.2.1 Deslocador da ULA (RV32I)

Componente que recebe uma instrução de 32 *bits* e desloca os *bits* para a direita ou para a esquerda, preenchendo os espaços vazios com 0, ou no caso dos deslocamentos aritméticos, com extensão de sinal. Esse componente está ilustrado na Figura 32.

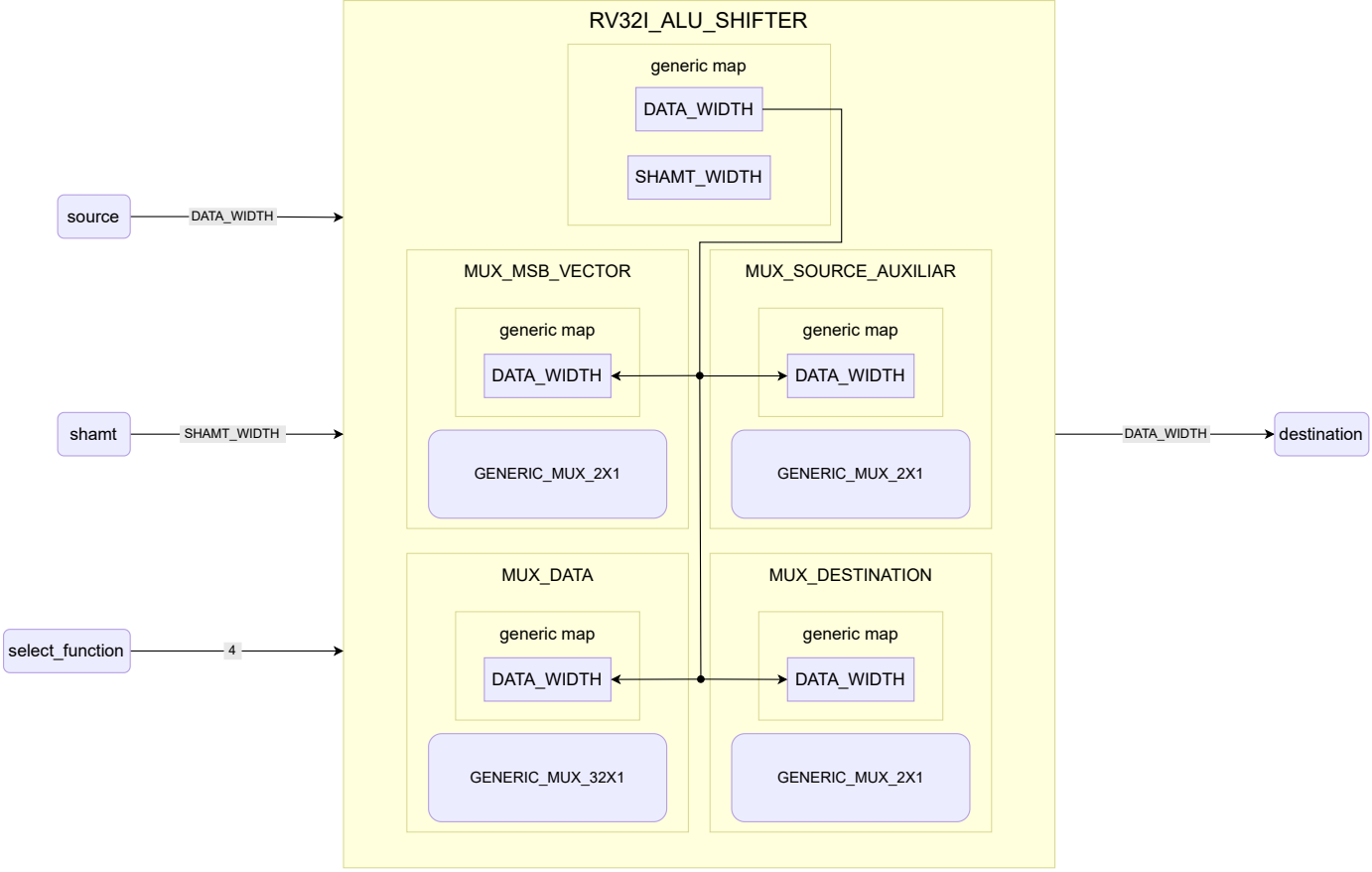


Figura 32 - Topologia do Componente RV32I Deslocador da ULA

3.2.2 Unidade Lógica e Aritmética (RV32I)

Componente que recebe dois vetores de entrada de 32 *bits*, e realiza operações entre elas, devolvendo o resultado na saída. Para realizar as operações, o componente recebe um seletor de 4 *bits* para decidir qual operação será realizada, duas entradas usadas nas operações, e devolve o resultado na saída, além de liberar um sinal que indica se houve ou não overflow. Esse componente está ilustrado na Figura 33.

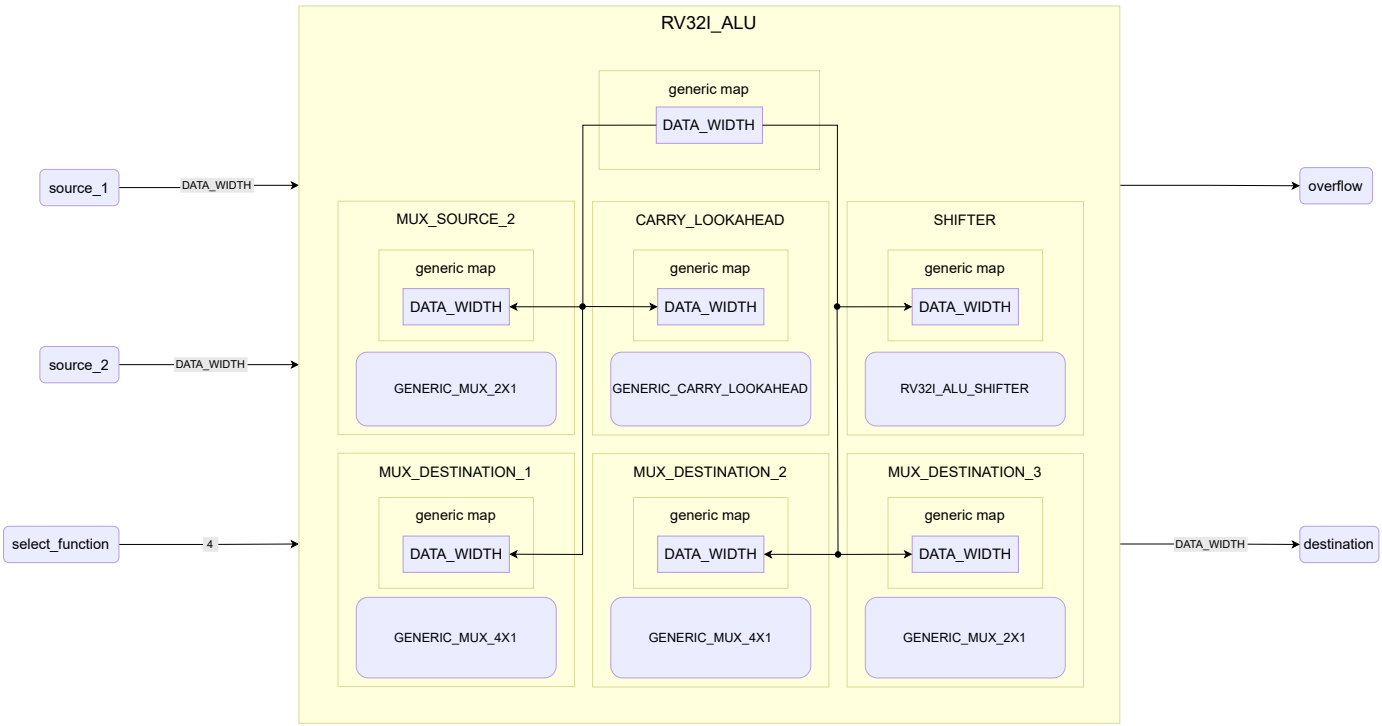


Figura 33 - Topologia do Componente RV32I Unidade Lógica e Aritmética

3.2.3 Banco de Registradores (RV32I)

Componente que funciona com clock e sinal de ativação, recebe três endereços de 5 *bits*, que apontam para 3 registradores do banco, e um vetor de 32 *bits* a ser escrito em um dos endereços, e devolve dois vetores de 32 *bits* que estavam guardados nos registradores apontados pelos outros dois endereços. Esse componente está ilustrado na Figura 34.

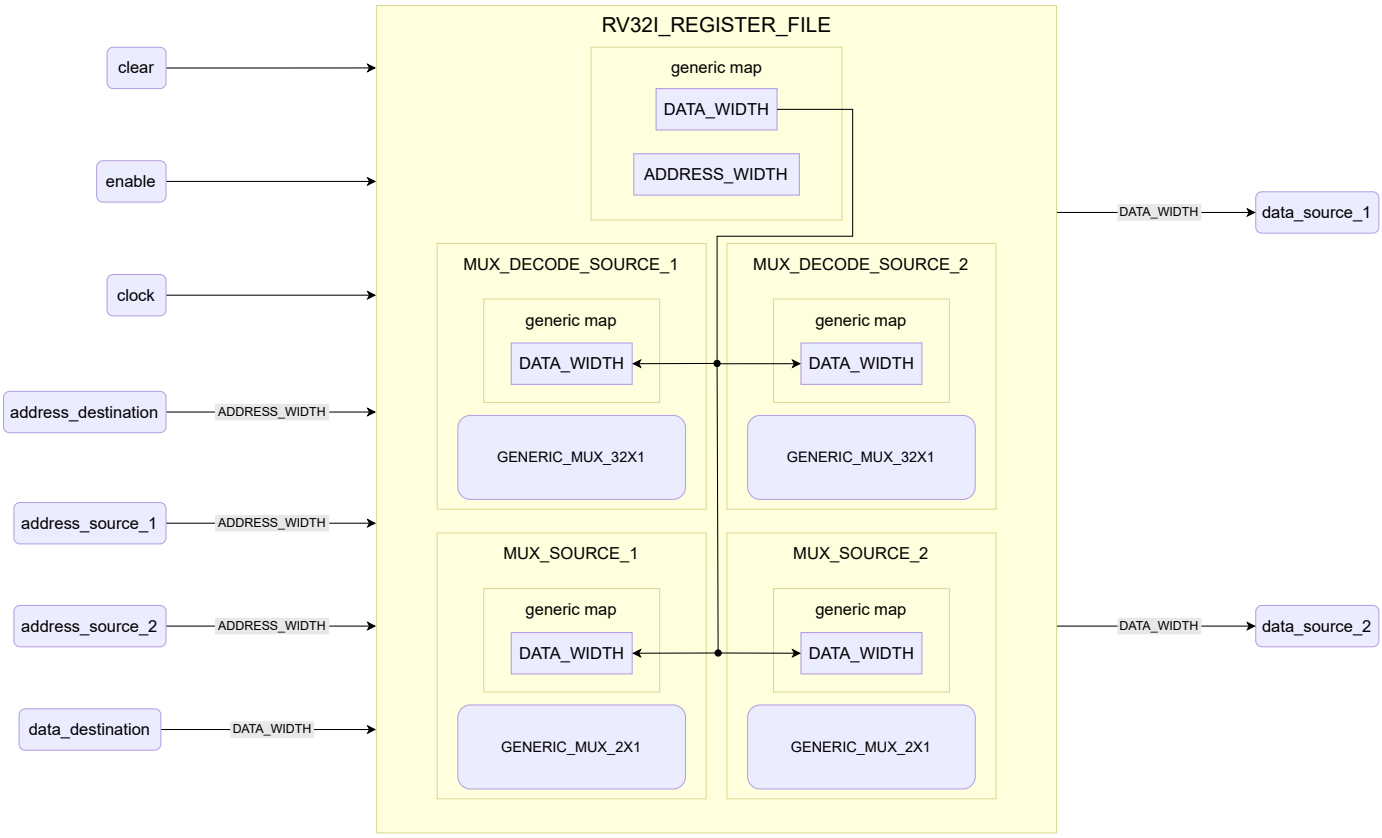


Figura 34 - Topologia do Componente RV32I Banco de Registradores

3.2.4 Controlador de Desvio (RV32I)

Componente que recebe duas flags de sinal, que indicam se cada um dos vetores de dados que foram comparados no comparador têm sinal, e três flags que indicam se o vetor de dados 1 é maior, igual ou menor ao vetor de dados 2. O componente também recebe a função da instrução, para saber se o resultado da comparação gera condição de desvio, e isso é indicado pelo sinal de saída. Esse componente está ilustrado na Figura 35.

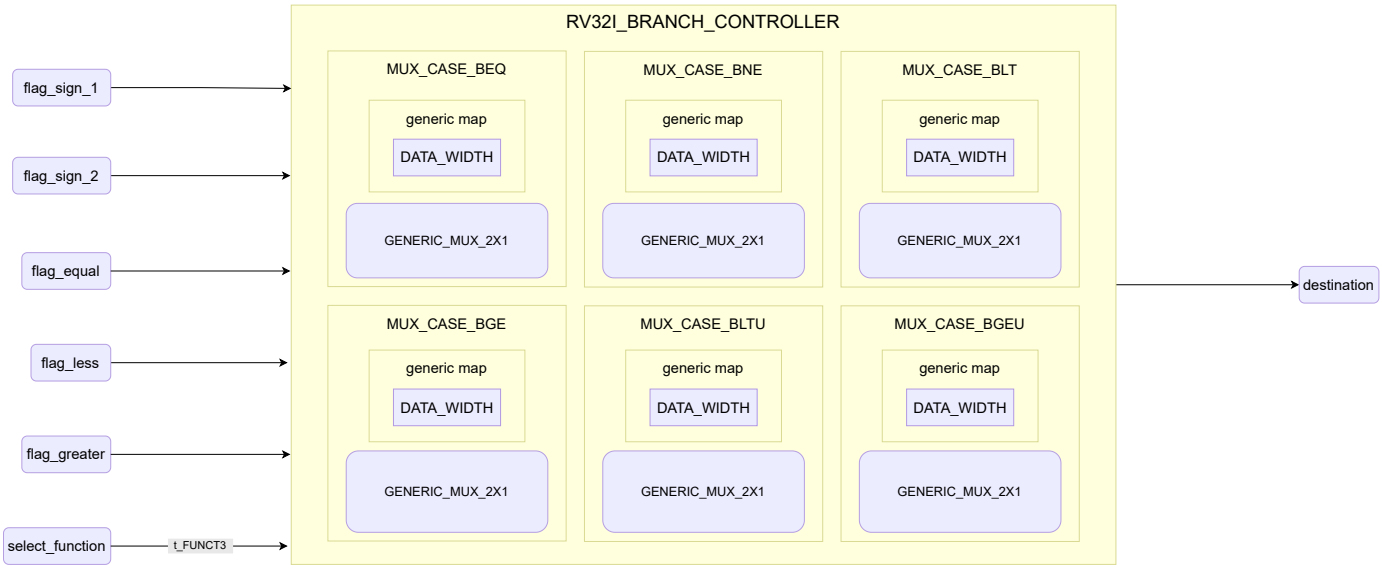


Figura 35 - Topologia do Componente RV32I Controlador de Desvio

3.2.5 Conversor de Tipo (RV32I)

Componente que recebe o tipo de dado deve ser trabalhado, se é *byte* (8 *bits*), *halfword* (16 *bits*) ou *word* (32 *bits*), e se os *bits* mais significativos devem ser 0 ou extensão de sinal, trata o valor recebido na entrada e devolve na saída. Esse componente está ilustrado na Figura 36.

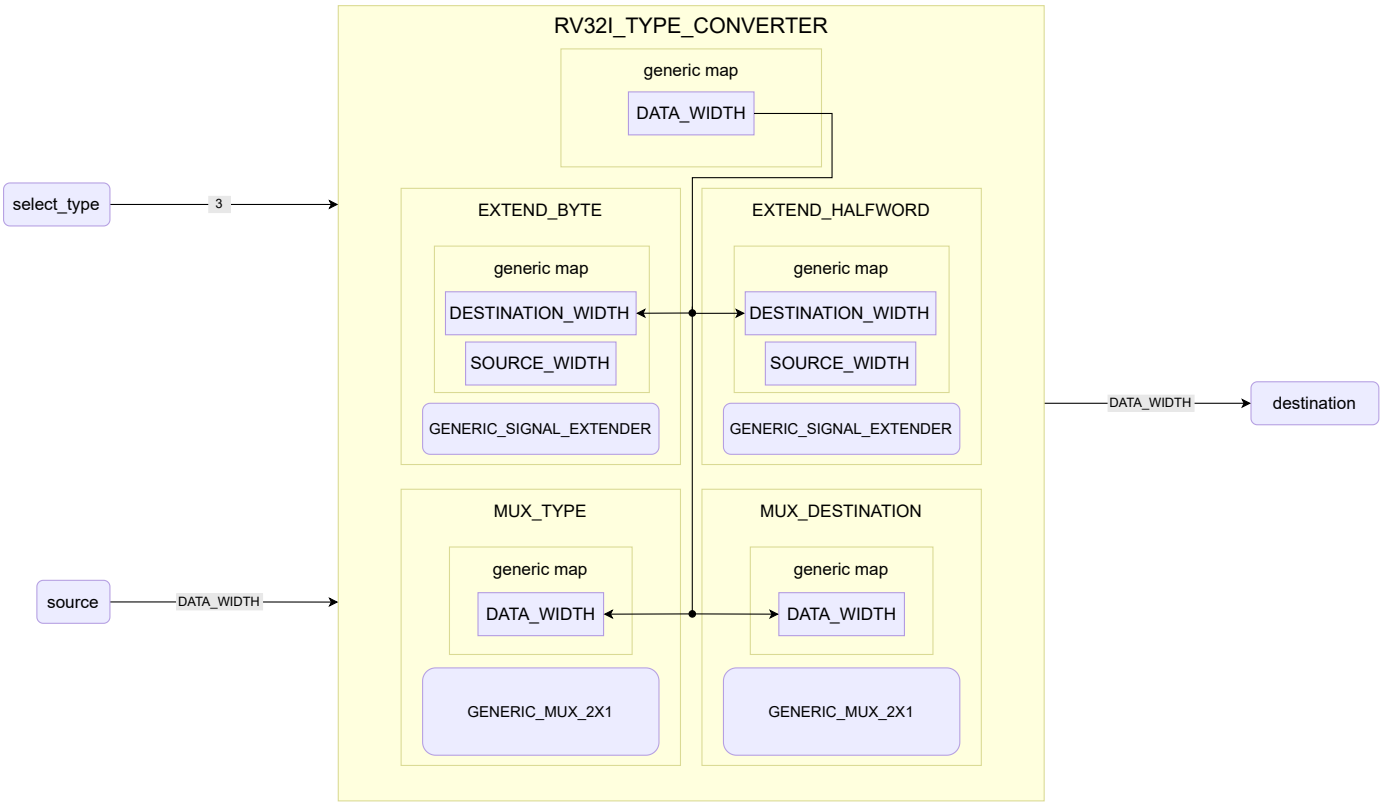


Figura 36 - Topologia do Componente RV32I Conversor de Tipo

3.3 Nível dos Módulos Desenvolvidos

O nível dos módulos é o nível intermediário entre os componentes e as etapas da *pipeline*. Nesses módulos estão contidas as integrações de componentes genéricos e RV32I que são implementados no processador.

3.3.1 Módulo Controlador da Unidade de Execução

Módulo que implementa a Unidade de Controle da ULA. Esse módulo está ilustrado na Figura 37.

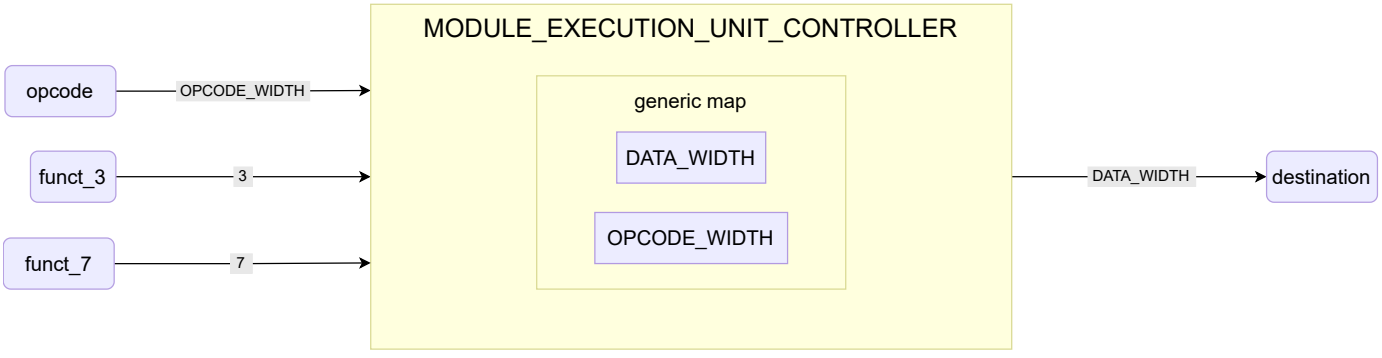


Figura 37 - Topologia do Módulo Controlador da Unidade de Execução

3.3.2 Módulo Unidade de Execução

Módulo que implementa a ULA. Esse módulo está ilustrado na Figura 38.

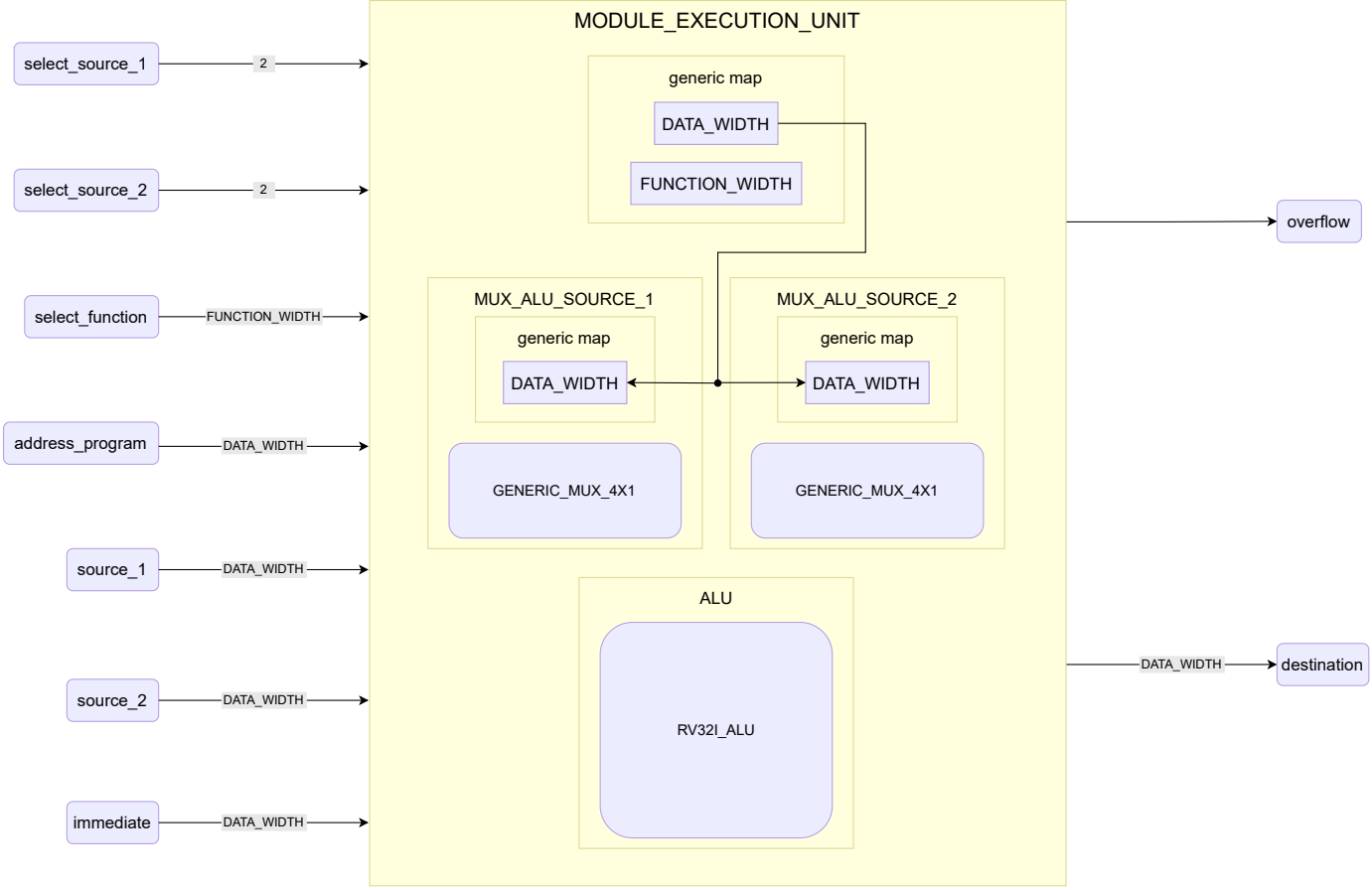


Figura 38 - Topologia do Módulo Unidade de Execução

3.3.3 Módulo Unidade de Controle

Módulo que recebe a instrução de 32 *bits* vinda da memória e devolve o imediato e os sinais de controle. Este módulo acabou por ser o único módulo que ao invés de implementar um componente com lógica próprio, implementa a lógica por si só, já que é o módulo responsável por mapear a lógica do ISA do RISC-V em pontos de controle e imediato. Esse módulo está ilustrado na Figura 39.

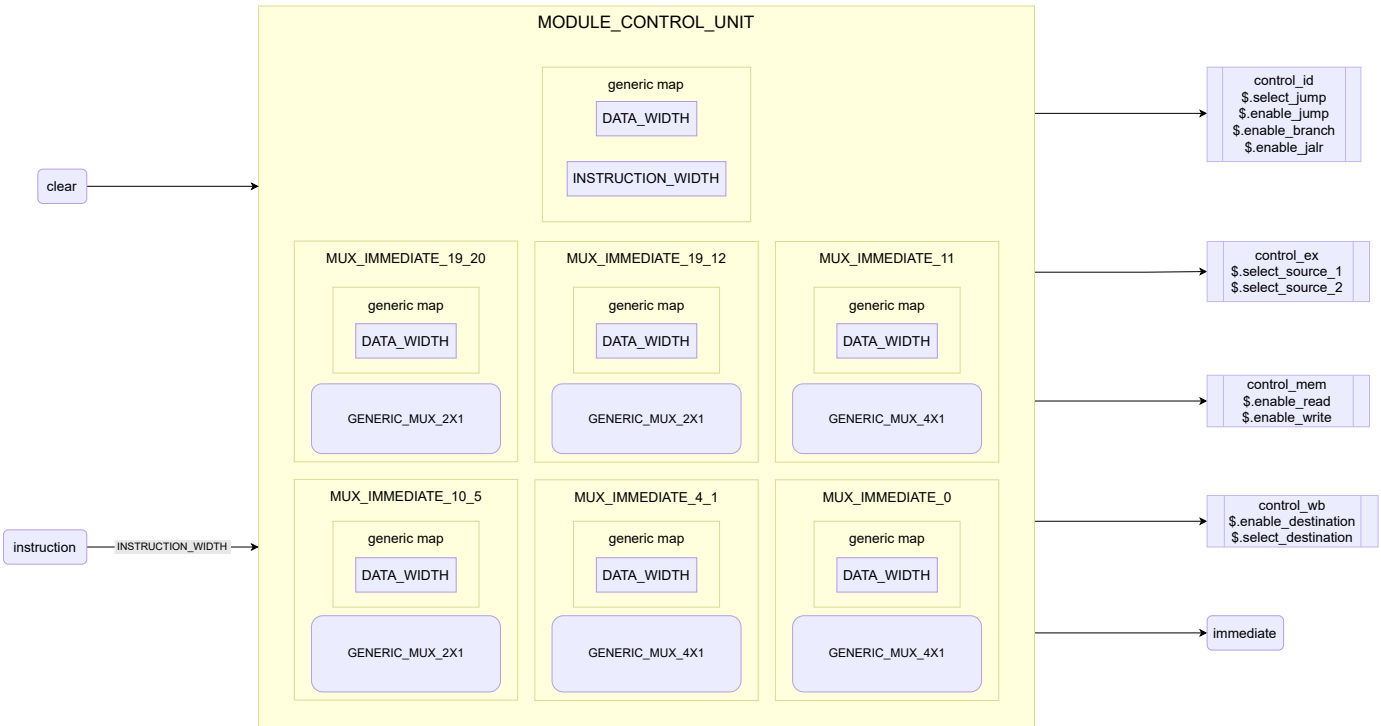


Figura 39 - Topologia do Módulo Unidade de Controle

3.3.4 Módulo Contador de Programa

Módulo que implementa o Contador de Programa, que é um Registrador Genérico. Esse módulo está ilustrado na Figura 40.

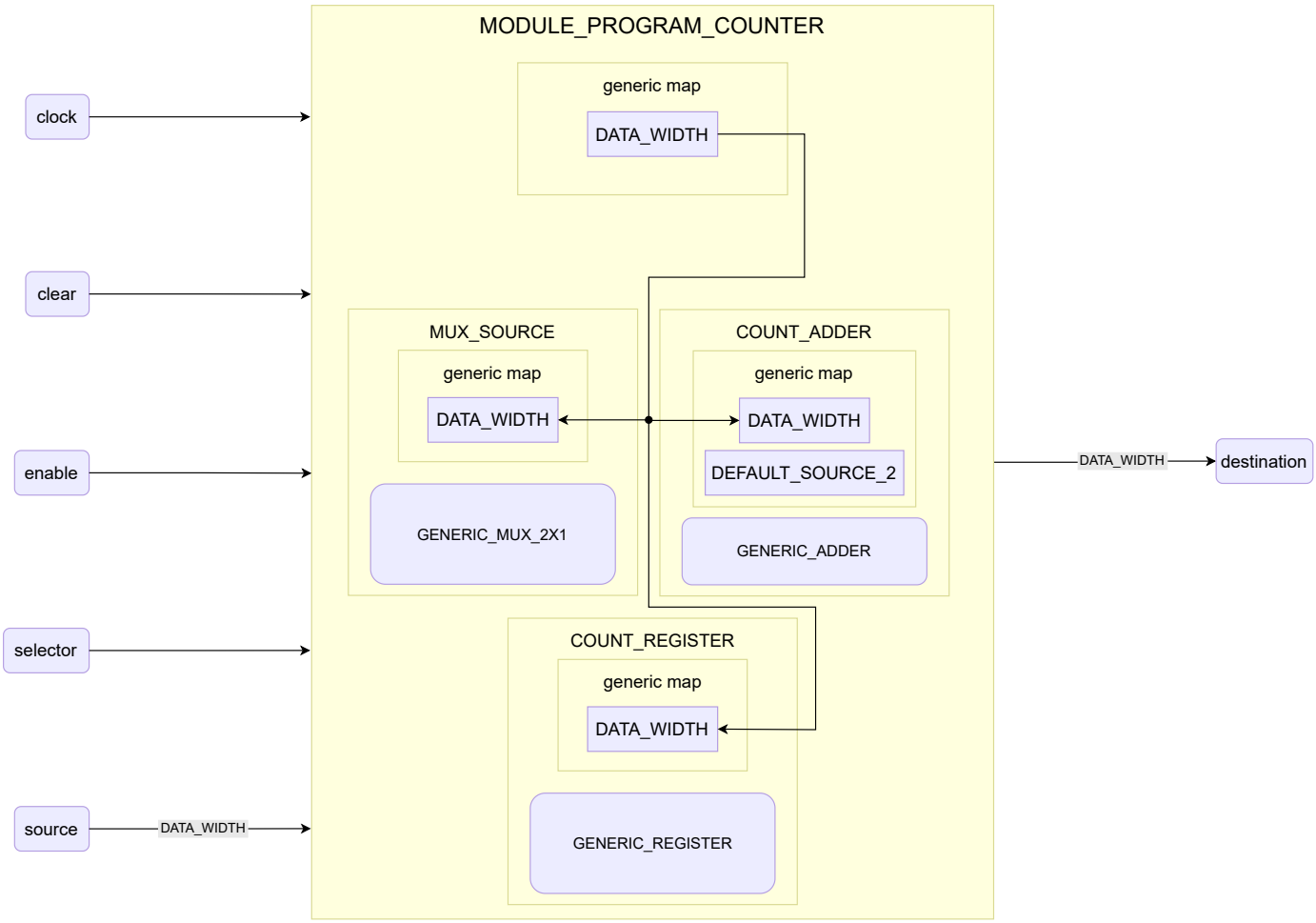


Figura 40 - Topologia do Módulo Contador de Programa

3.3.5 Módulo Banco de Registradores

Módulo que implementa o Banco de Registradores. Esse módulo está ilustrado na Figura 41.

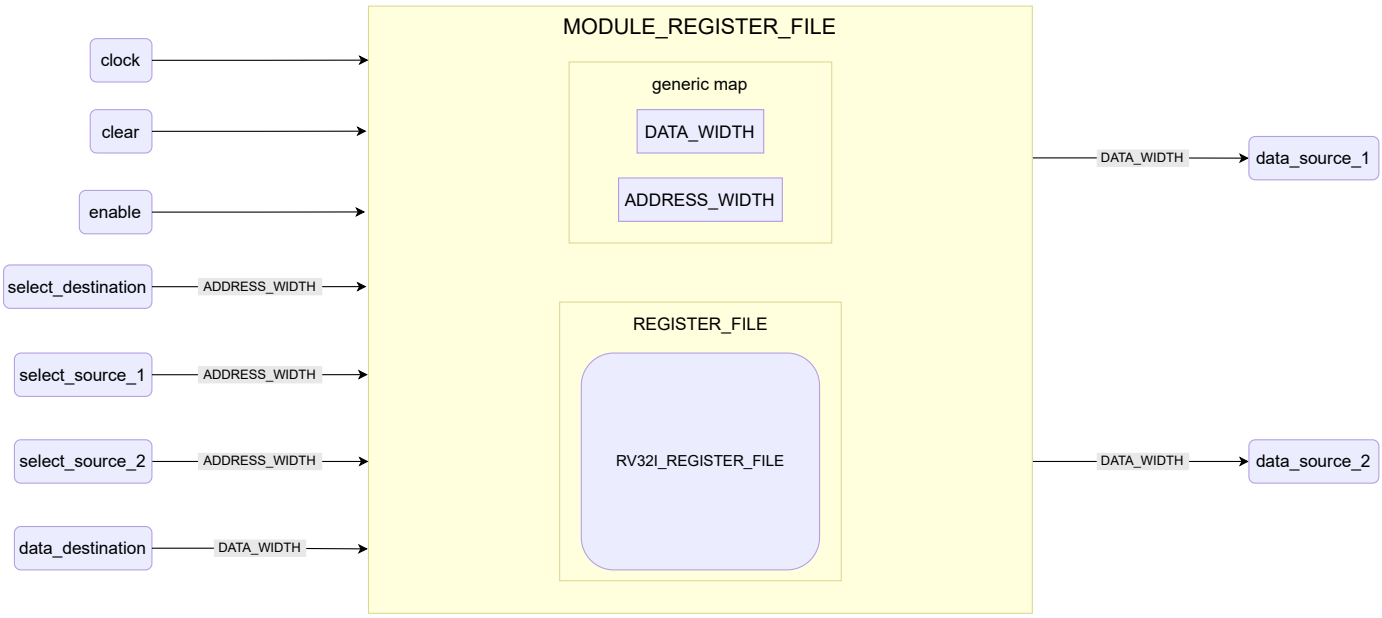


Figura 41 - Topologia do Módulo Banco de Registradores

3.3.6 Módulo Escrita no Retorno

Módulo que implementa o multiplexador de escrita no banco de registradores. Esse módulo está ilustrado na Figura 42.

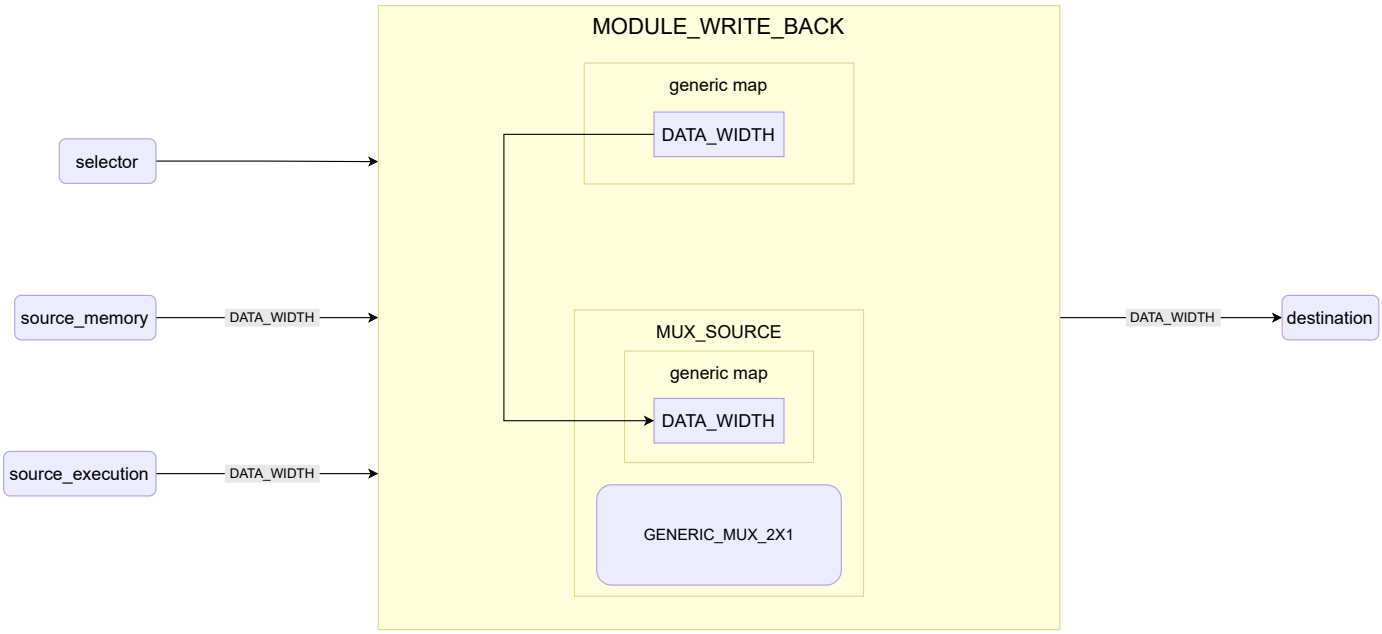


Figura 42 - Topologia do Módulo Escrita no Retorno

3.3.7 Módulo Comparação para Desvio

Módulo que implementa o comparador. Esse módulo está ilustrado na Figura 43.

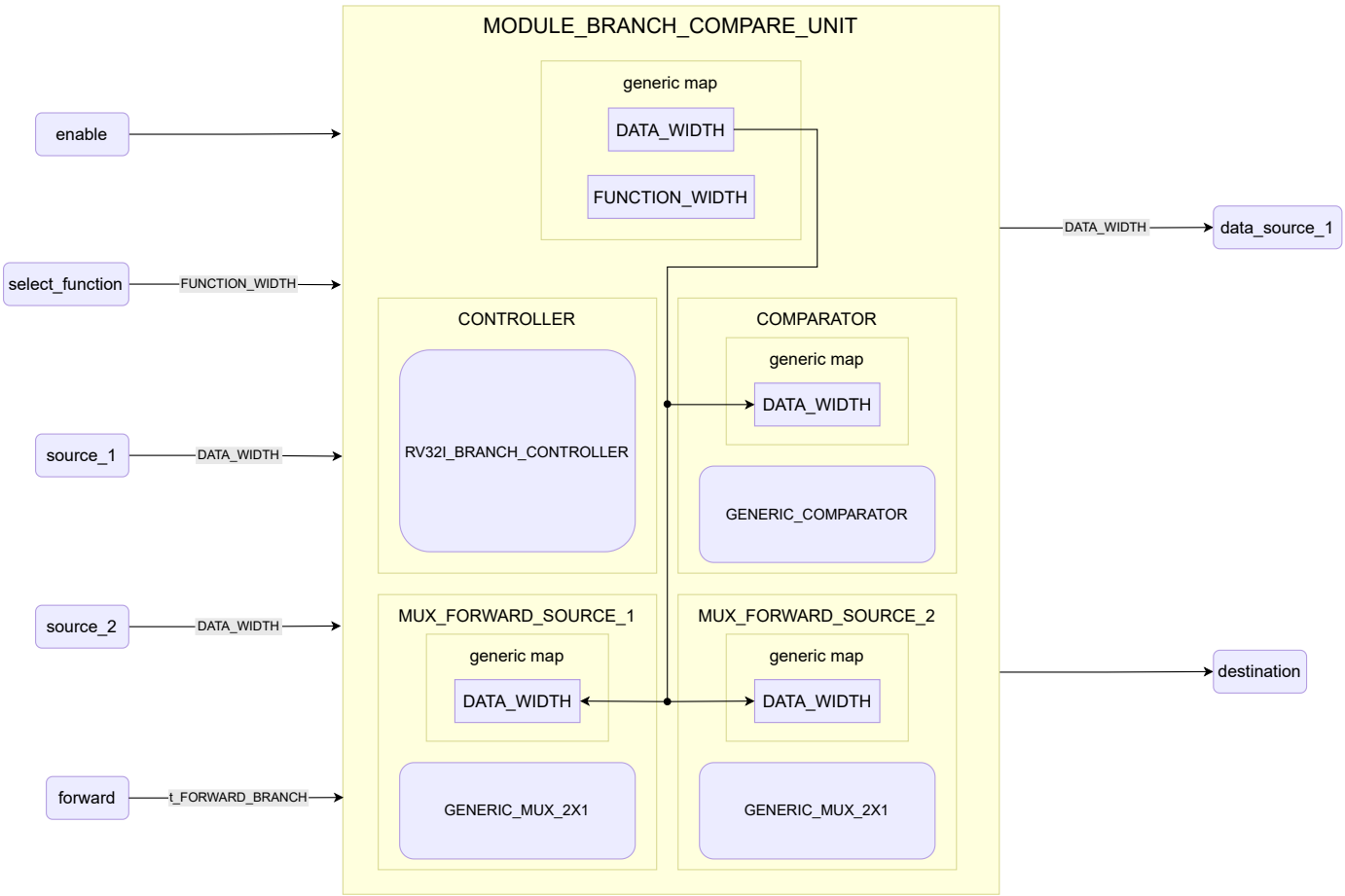


Figura 43 - Topologia do Módulo Comparação para Desvio

3.3.8 Módulo Desvio

Módulo que implementa a lógica que determina o endereço de destino de uma instrução de desvio. Esse módulo está ilustrado na Figura 44.

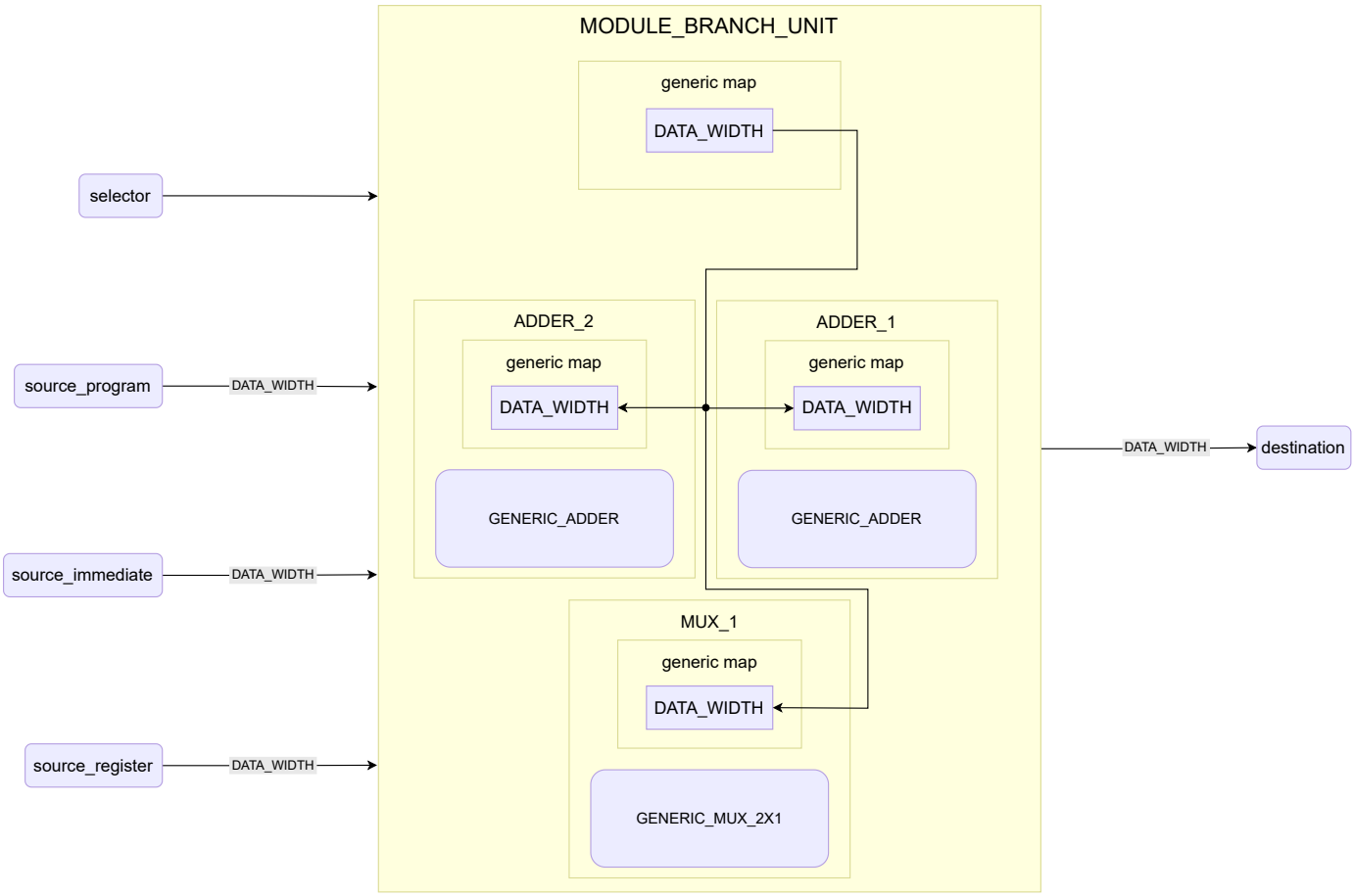


Figura 44 - Topologia do Módulo Desvio

3.3.9 Módulo Interface de Memória

Módulo que implementa a lógica de processamento de dados para interação com a memória RAM, ilustrado na Figura 45.

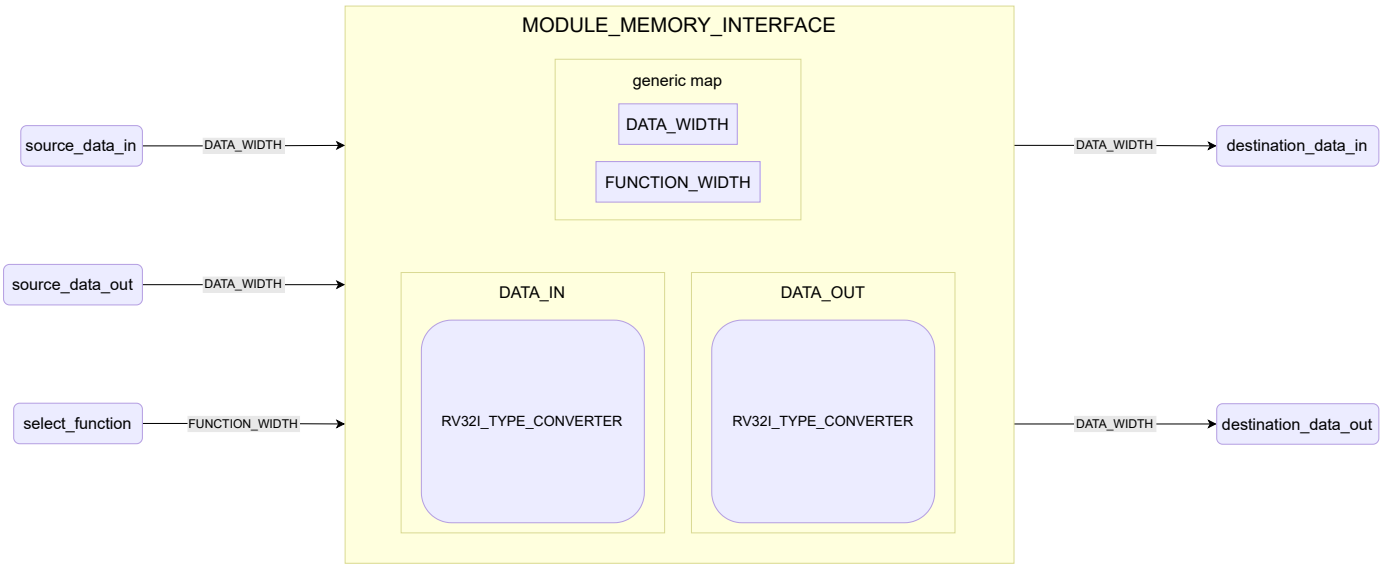


Figura 45 - Topologia do Módulo Desvio

3.4 Nível dos Componentes da CPU Desenvolvidos

O nível mais alto da hierarquia (ver Seção 2), onde estão implementadas as etapas do *pipeline* do processador, o *top level* do mesmo, e os elementos que implementam solução de *hazards* inerentes ao processador, tanto *hazards* de controle quanto de dados.

3.4.1 Etapa IF

A Etapa de Busca de Instrução é responsável por definir a próxima instrução a ser executada pelo processador. Nesta etapa, foi implementado somente o módulo Contador de Programa, uma vez que o cliente planeja usar uma memória externa no lugar da memória ROM, que não está implementada dentro da CPU. Esta é a única etapa que não tem função de registrador. Essa etapa está ilustrada na Figura 46.

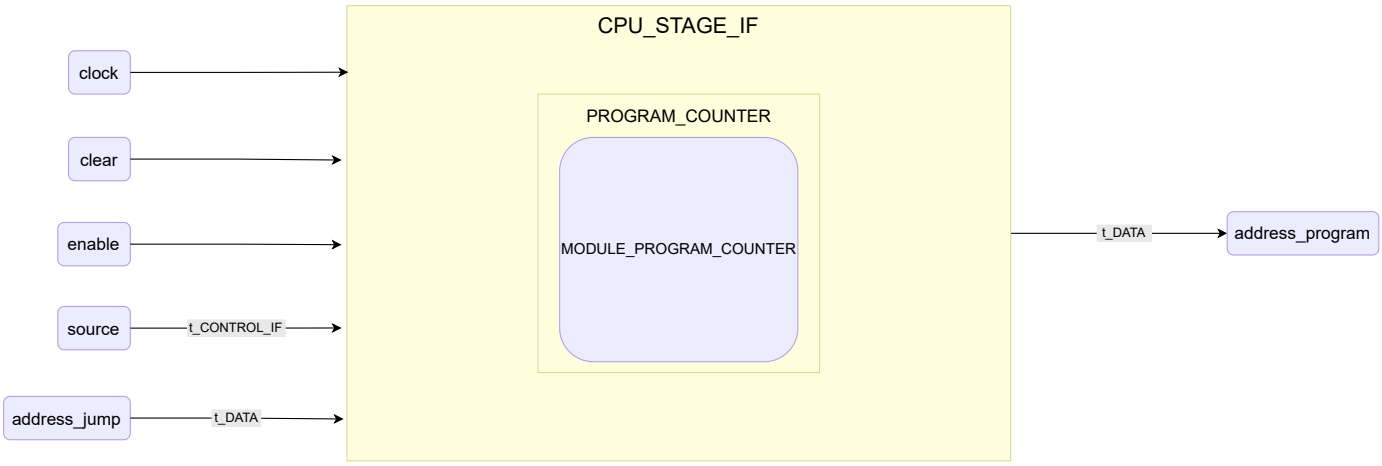


Figura 46 - Topologia da Etapa IF

3.4.2 Etapa ID

A Etapa de Decodificação de Instrução é responsável por interpretar o que será executado e definir os sinais de controle. Nesta etapa, foram implementados os módulos Unidade de Controle, Desvio, Banco de Registradores e Comparação para Desvio. Essa etapa está ilustrada na Figura 47.

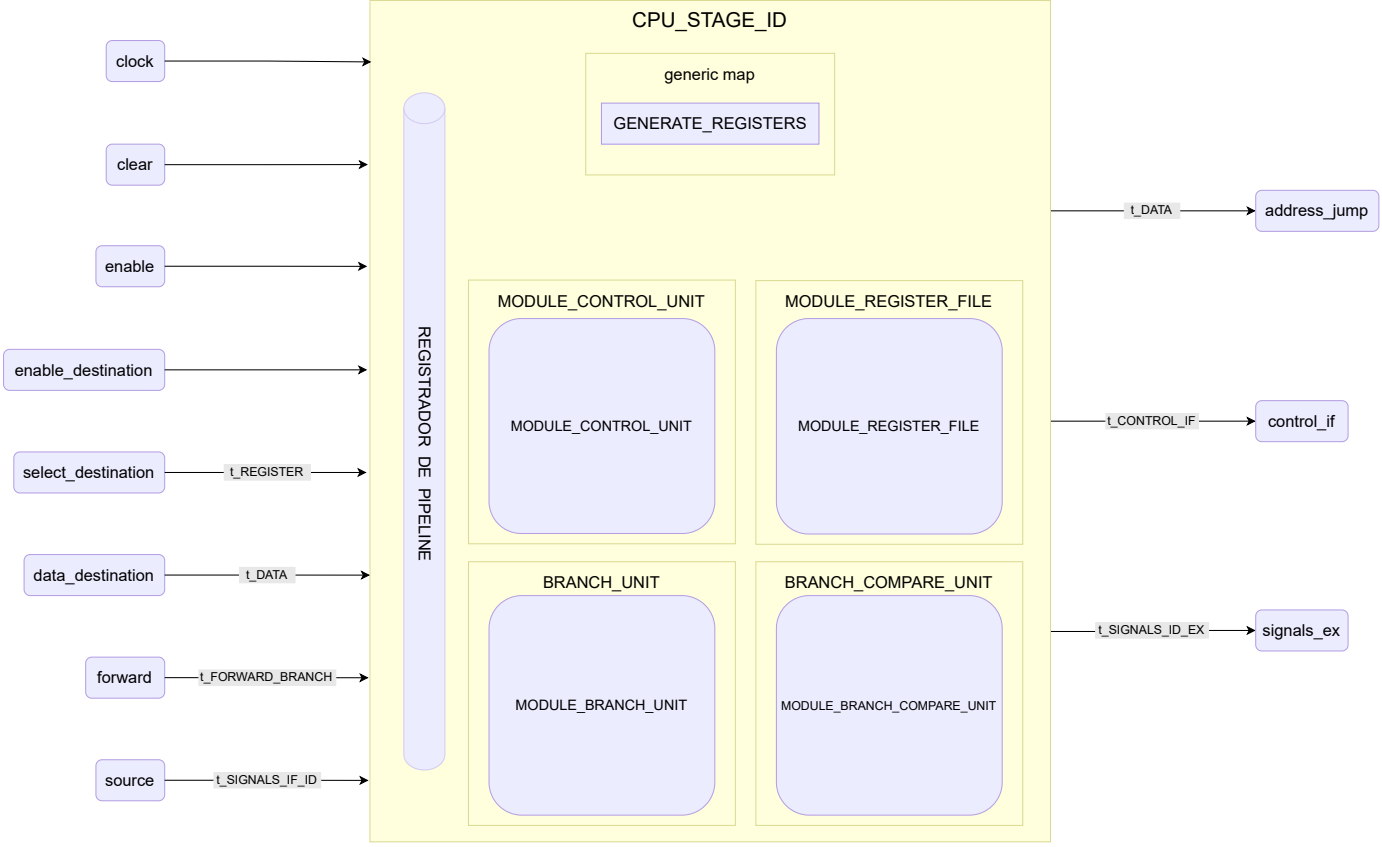


Figura 47 - Topologia da Etapa ID

3.4.3 Etapa EX

A Etapa de Execução é onde se realizam as operações do processador. Nesta etapa, foram implementados os módulos Controlador da Unidade de Execução, Unidade de Execução e Unidade Encaminhamento de Dados para Execução. Essa etapa está ilustrada na Figura 48.

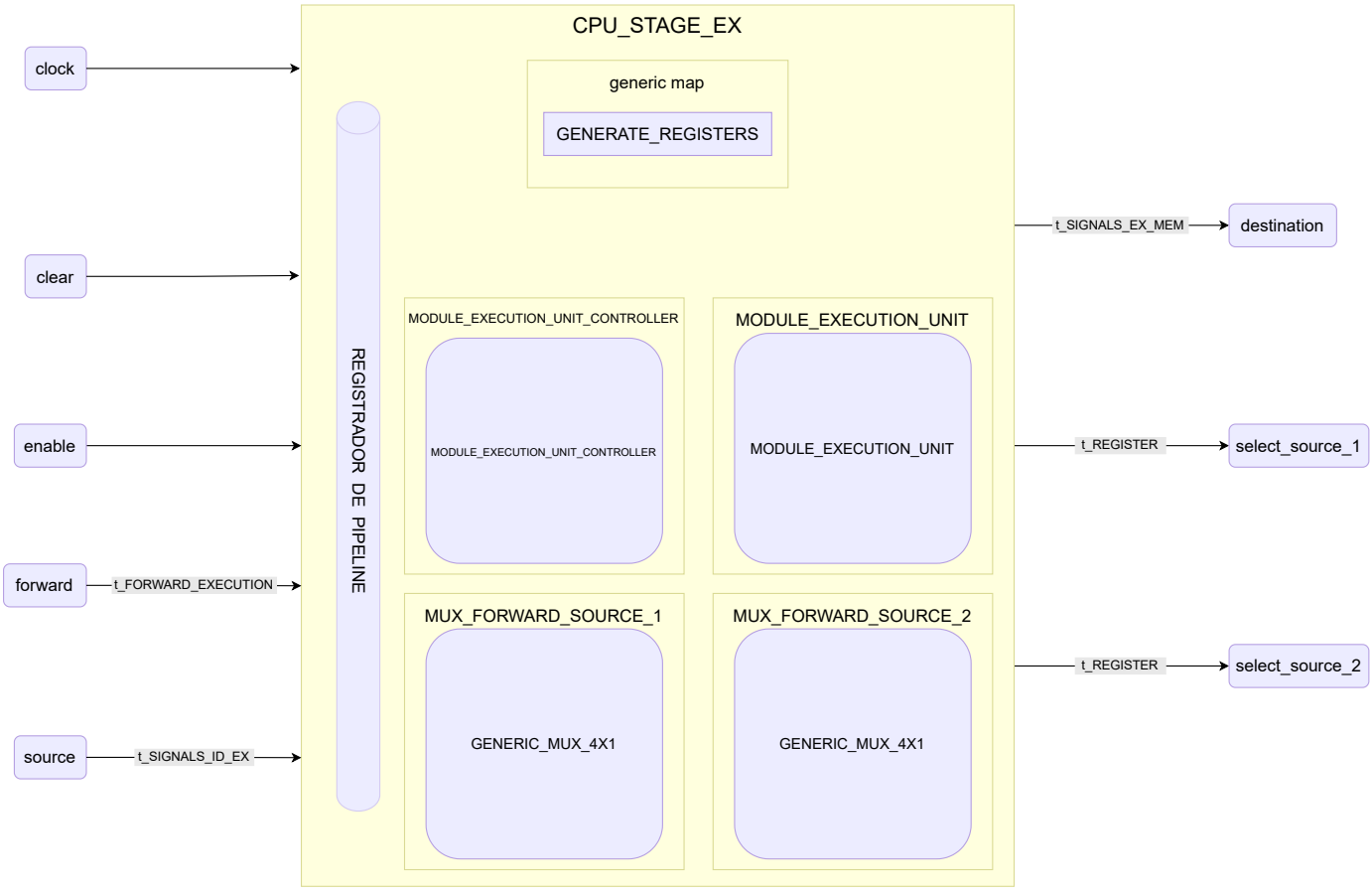


Figura 48 - Topologia da Etapa EX

3.4.4 Etapa MEM

A Etapa de Acesso à Memória é responsável por guardar valores ou usar valores nos endereços da memória RAM. No entanto, como a memória RAM será externa, de acordo com a vontade do cliente, a mesma só tem os componentes Extensor de *bits* para STORE e Extensor de Sinal para LOAD implementados. Essa etapa está ilustrada na Figura 49.

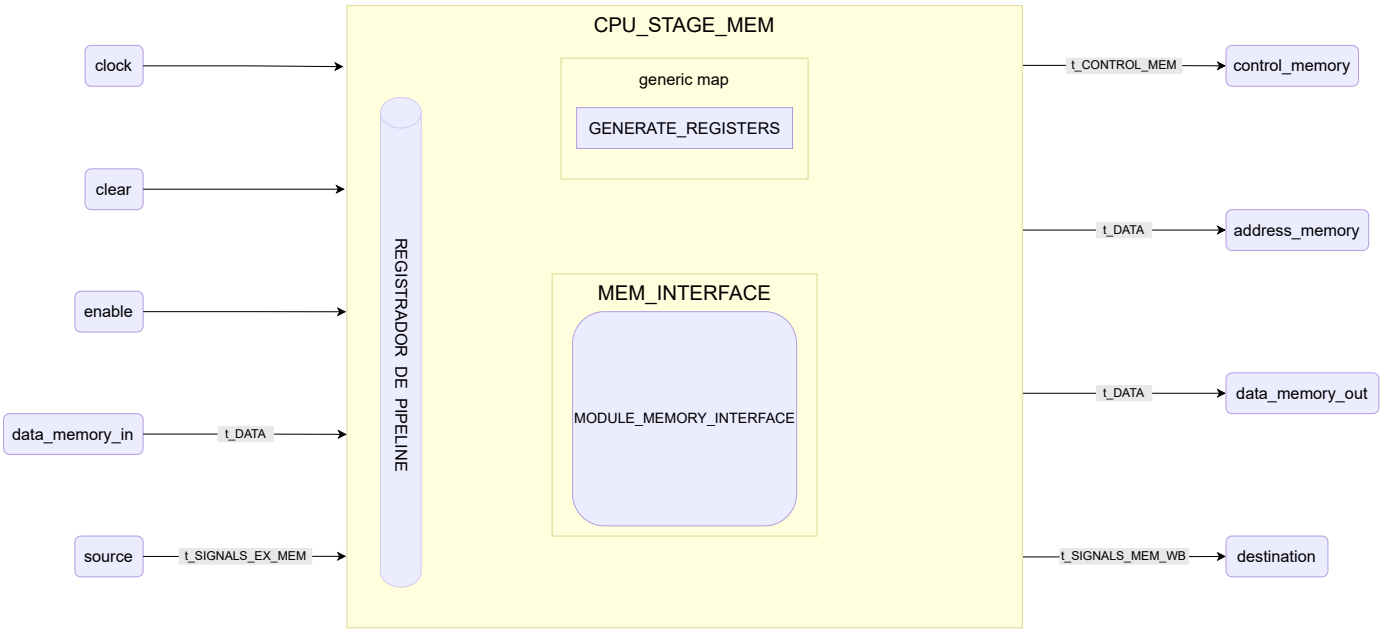


Figura 49 - Topologia da Etapa MEM

3.4.5 Etapa WB

A Etapa de Escrita no Retorno é onde o resultado da operação do processador ou o valor acessado na memória RAM sai para ser escrito no Banco de Registradores. É nesta etapa que se implementa o módulo Escrita no Retorno. Essa etapa está ilustrada na Figura 50.

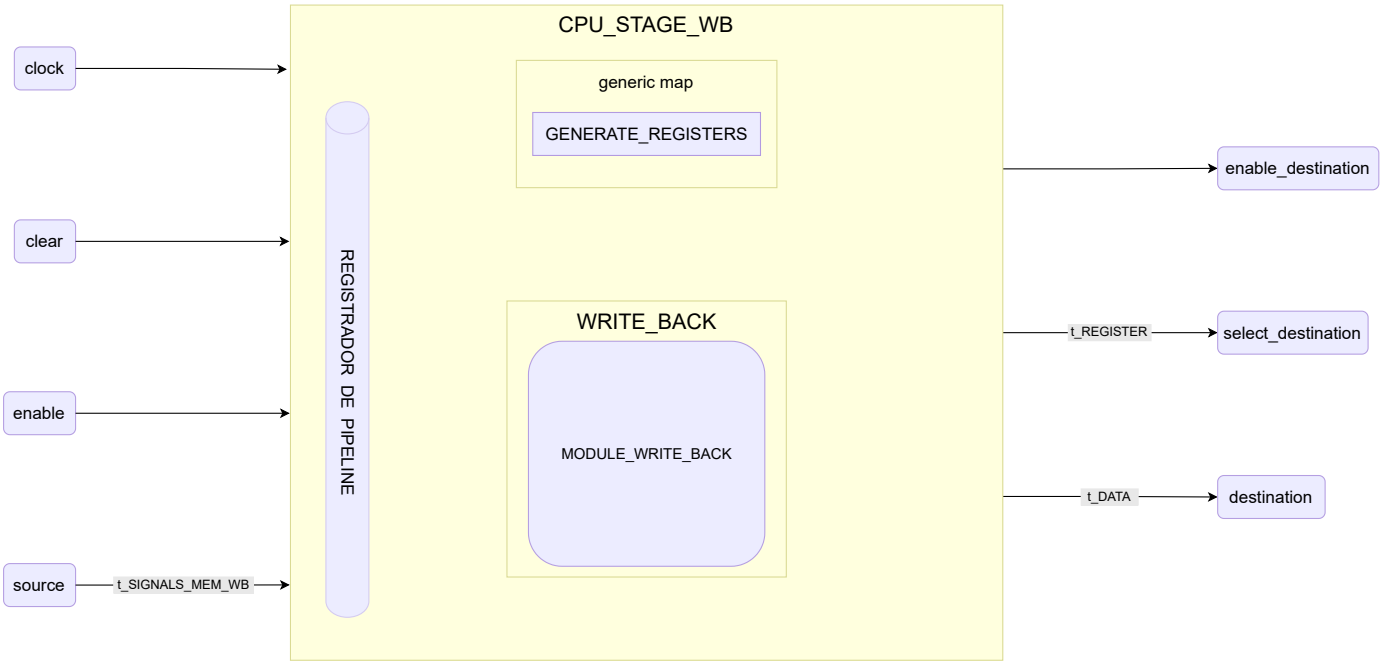


Figura 50 - Topologia da Etapa WB

3.4.6 Unidade de Encaminhamento de Dados para Execução (CPU)

Componente que encaminha dados das etapas MEM e WB (ver Seção 2) para a etapa EX (ver Seção 2), para resolver um *hazard* de dado. Esse componente está ilustrado na Figura 51.

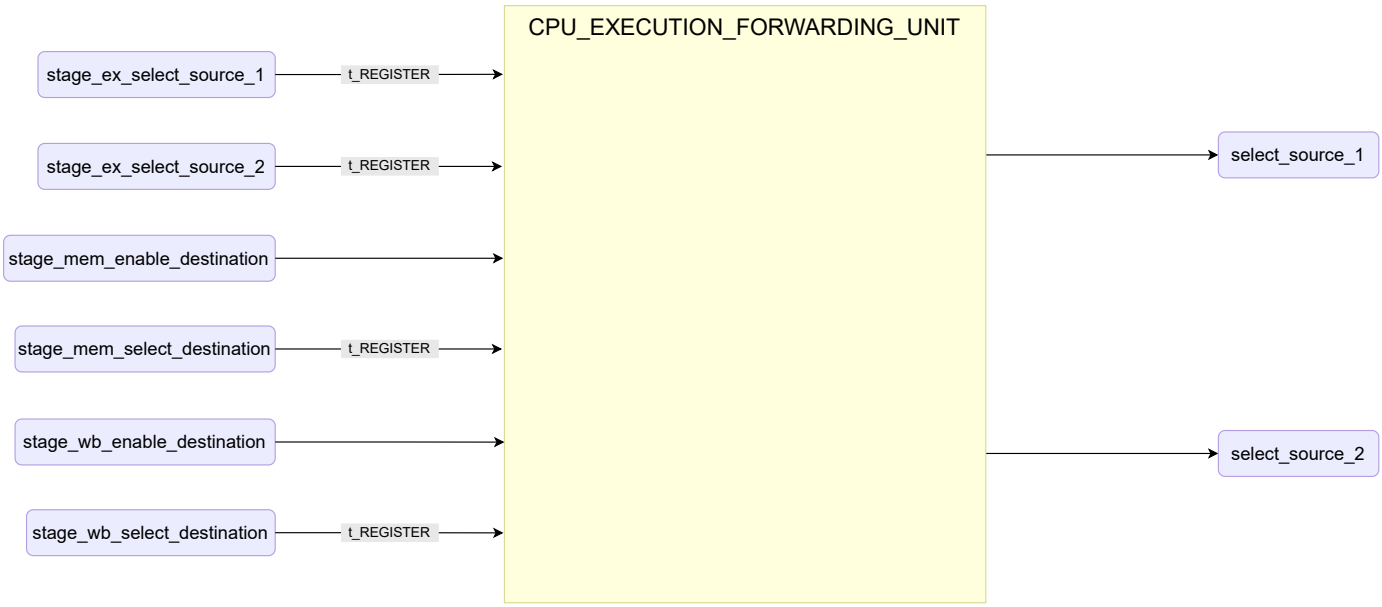


Figura 51 - Topologia do Componente de CPU Unidade de Encaminhamento de Dados para Execução

3.4.7 Unidade de Encaminhamento de Dados para Desvio (CPU)

Componente que encaminha dados da etapa MEM (ver Seção 2) para a etapa ID (ver Seção 2), para resolver um *hazard* de dado. Esse componente está ilustrado na Figura 52.

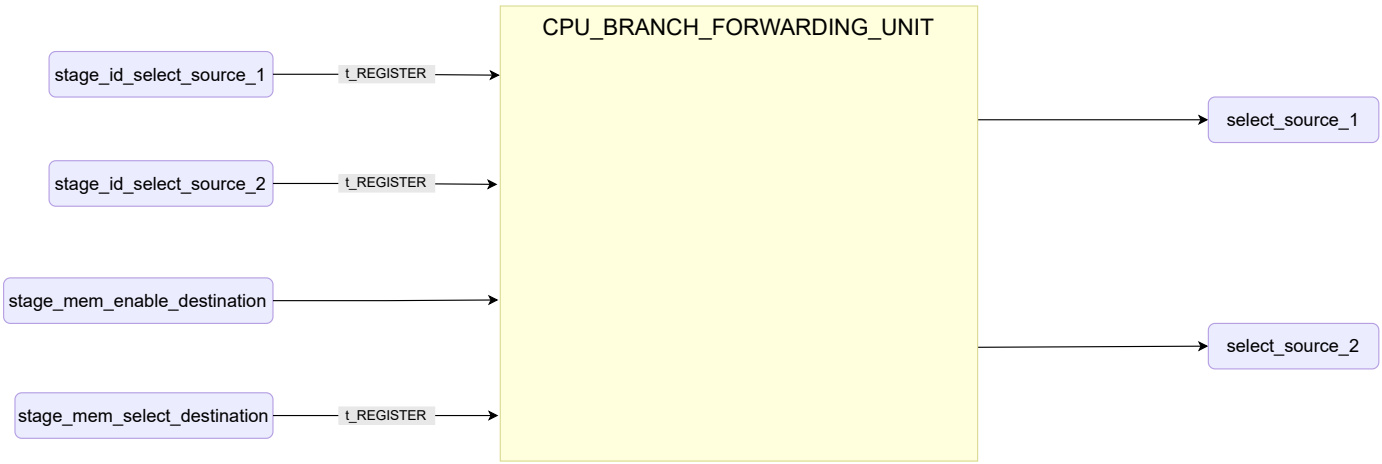


Figura 52 - Topologia do Componente de CPU Unidade de Encaminhamento de Dados para Desvio

3.4.8 Unidade de Controle de Hazard (CPU)

Componente que responsável por solucionar *hazards* de controle que existem no processador. Esse componente está ilustrado na Figura 53.

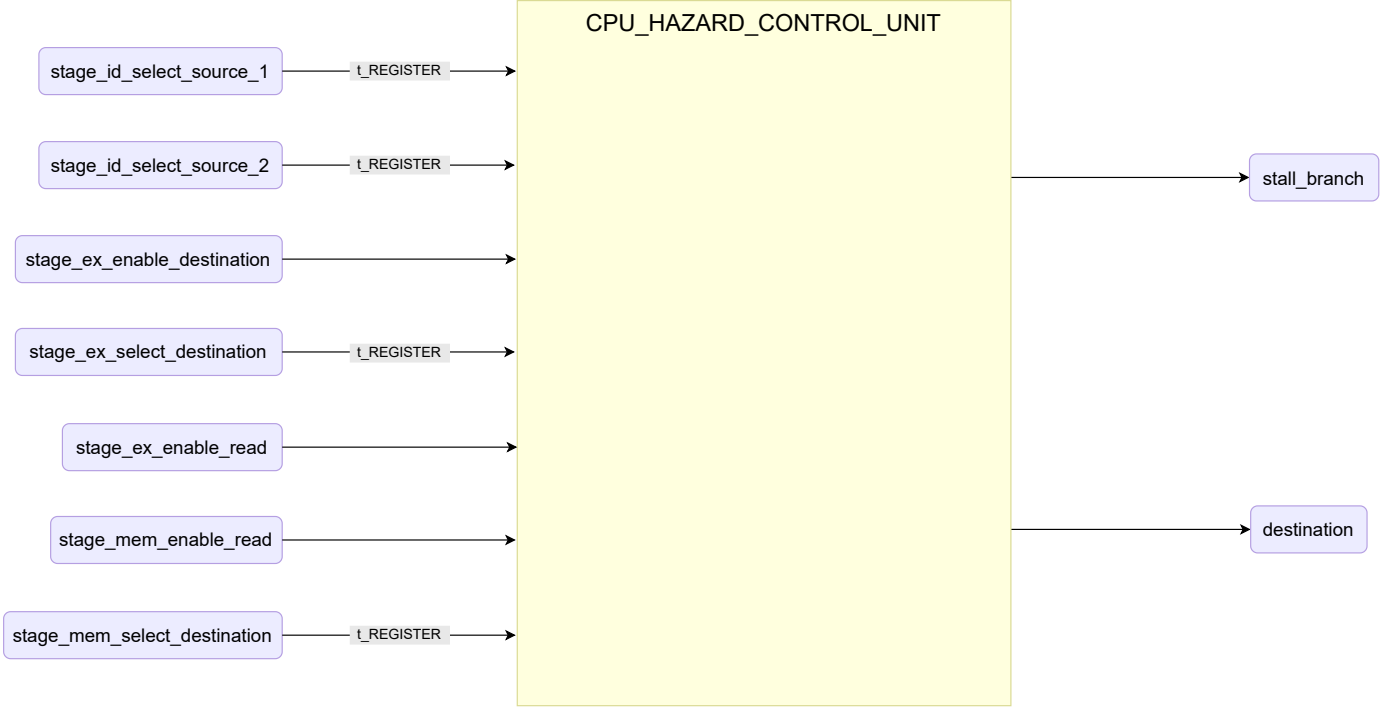


Figura 53 - Topologia do Componente de CPU Unidade de Controle de Hazard

3.4.9 *Top Level* (CPU)

Componente que integra os outros componentes da etapa CPU, ilustrado na Figura 54.

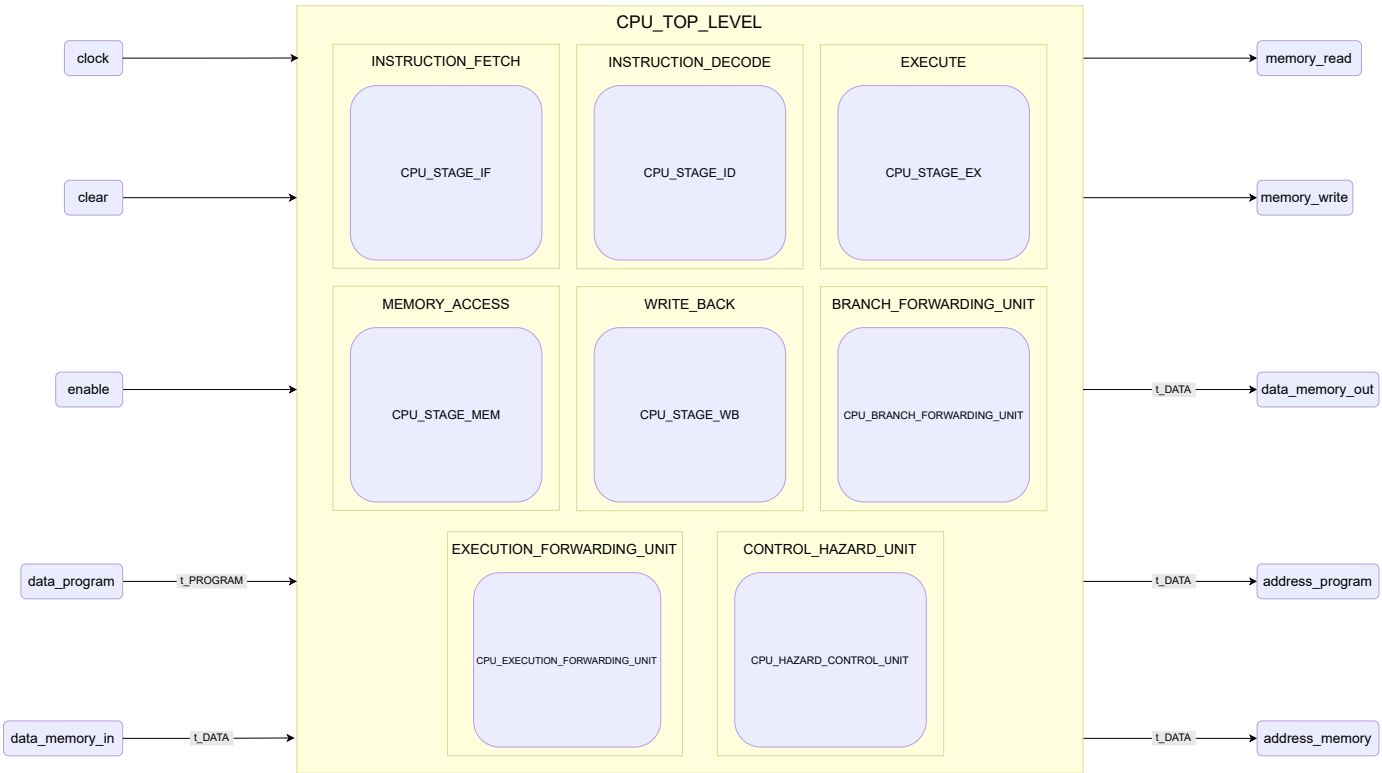


Figura 54 - Topologia do Componente de CPU *Top Level*

3.5 Nível de Visão Geral do Projeto

O nível composto apenas pelo componente *top level* que integra a CPU com as memórias, como ilustrado na Figura 55.

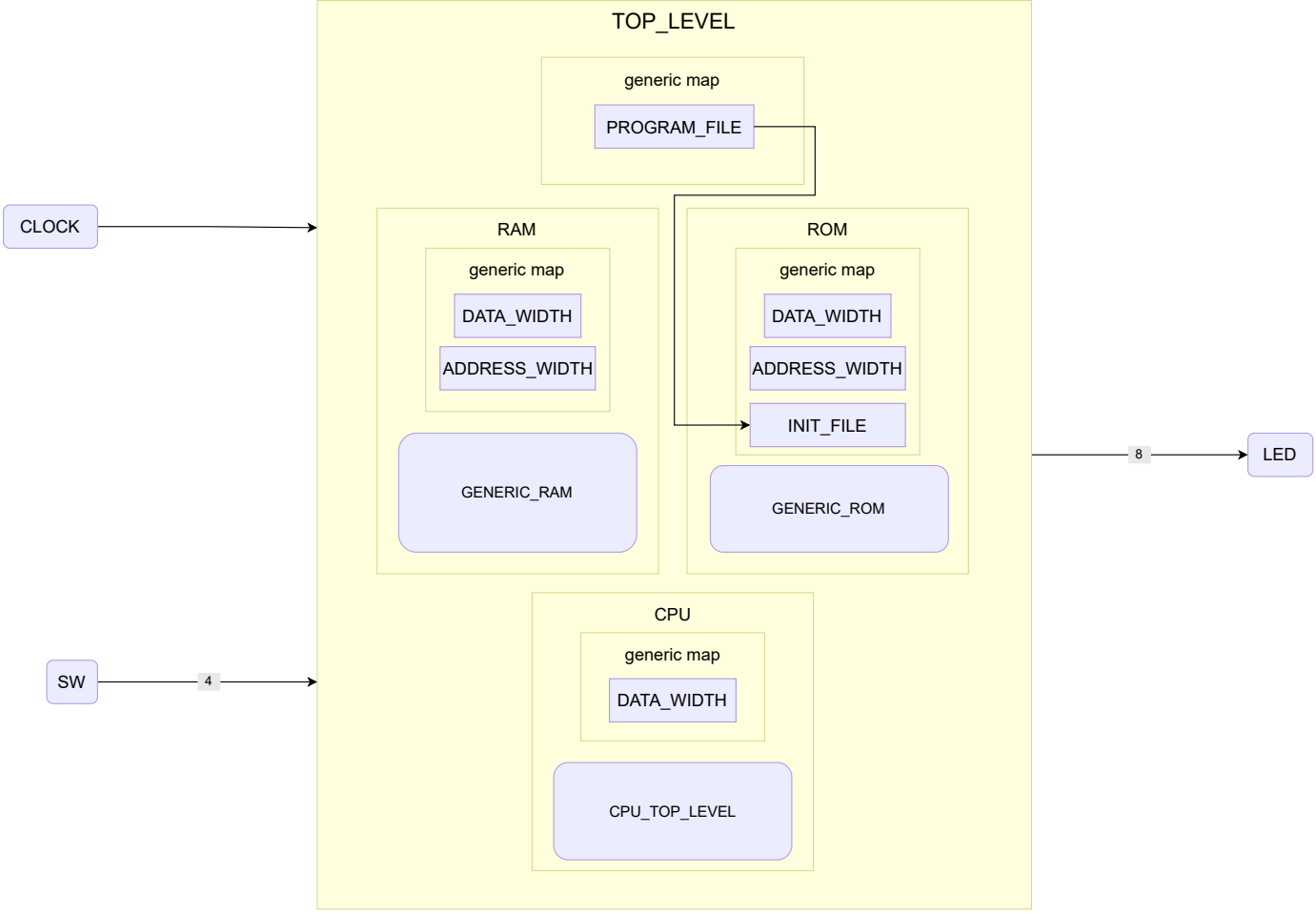


Figura 55 - Topologia do Componente *Top Level* do Projeto

4. Conclusões e trabalhos futuros

Até o momento, se desenvolveu um processador funcional com *pipeline*, com testes de integração e unidade implementados, e se documentou o projeto com um site que serve de guia sobre o mesmo. O processador tem os seus *hazards* resolvidos, o que permite que sejam testadas instruções como um conjunto.

Esse projeto não só agrega valor para o cliente, como também ajudou os autores a aprender mais sobre processadores, VHDL e setor aeroespacial, além de proporcionar aos mesmos uma oportunidade de ter contato com membros do CTI e com o Centro Nacional de Pesquisa em Energia e Materiais (CNPEM).

Além do que havia sido proposto pelo CTI, percebeu-se ao longo do projeto que o trabalho realizado pelo grupo contribuiu para que o cliente conhecesse outras ferramentas de desenvolvimento, mais associadas ao desenvolvimento de software.

Para os próximos passos do projeto, planeja-se implementar a extensão de multiplicação e divisão no processador, além de que se comece o desenvolvimento mais voltado a parte de software do mesmo, assim como a adição de periféricos.

Algo que se pensou também seria o desenvolvimento de um *offshoot* do processador que se adequa às especificações da gaisler, modificando os diversos pinos de entrada e saída dos componentes para registros (*records*).

Referências

ARM LTD. **About Arm, Company Information, Benefits, History.** Disponível em: https://www.arm.com/company. Acesso em: 16 mar. 2024.

CHEN, C. et al. **Xuantie-910: A Commercial Multi-Core 12-Stage Pipeline Out-of-Order 64-bit High Performance RISC-V Processor with Vector Extension : Industrial Product.** 1 maio 2020.

COCOTB. **Cocotb 1.8.1 documentation.** Disponível em: https://docs.cocotb.org/en/stable/. Acesso em: 27 fev. 2024.

EDINGTON, A. A.; DOMINGUES, B. S.; VALE, L. L.; SANTOS, R. D. **Conformance Tester for Tags EPC-GEN2 UHF RFID.** Insper, 2021. Disponível em: https://pfeinsper.github.io/21b-indago-rfid-conformance-tester/. Acesso em: 27 fev. 2024.

DEBIAN. **RISC-V - Debian Wiki.** Disponível em: https://wiki.debian.org/RISC-V. Acesso em: 15 maio. 2024.

DIAS, L. F.; RUGGIERO, G. V.; SEIXAS, T. V. **RISC-V para uso Aeroespacial - Documentação do Projeto.** Insper, 2024. Disponível em: https://insper-riscv.github.io/docs/. Acesso em: 17 mar. 2024.

FEDORA. **Architectures/RISC-V - Fedora Project Wiki.** Disponível em: https://fedoraproject.org/wiki/Architectures/RISC-V. Acesso em: 15 maio. 2024.

FREERTOS. **FreeRTOS for RISC-V RV32 and RV64.** Disponível em: https://www.freertos.org/Using-FreeRTOS-on-RISC-V.html. Acesso em: 15 maio. 2024.

FURANO, G. et al. **A European Roadmap to Leverage RISC-V in Space Applications.** 2022 IEEE Aerospace Conference (AERO), 5 mar. 2022.

GAISLER. **GRLIB IP Library.** Disponível em: https://www.gaisler.com/index.php/downloads/grlib. Acesso em: 17 mar. 2024.

GAISLER. **NOEL-V (RISC-V).** Disponível em: https://www.gaisler.com/index.php/products/processors/noel-v. Acesso em: 15 maio. 2024.

GCC. **RISC-V Options (Using the GNU Compiler Collection (GCC)).** Disponível em: https://gcc.gnu.org/onlinedocs/gcc/RISC-V-Options.html. Acesso em: 15 maio. 2024.

GDB. **GDB** . Disponível em: https://sourceware.org/gdb/news/. Acesso em: 15 maio. 2024. GINGOLD, T. **GHDL.** GNU General Public License, 2023. Disponível em: https://ghdl.github.io/ghdl/. Acesso em: 28 abr. 2024.

GITHUB. **GitHub.** Github, Inc. Disponível em: https://github.com/. Acesso em: 28 abr. 2024.

GITHUB PAGES. **GitHub.** Github, Inc. Disponível em: https://pages.github.com/. Acesso em: 28 abr. 2024.

GUL, S.; AFTAB, N.; ARFA. **A Comparison between RISC and CISC Microprocessor Architectures.** International Journal of Science Engineering and Advance Technology, IJSEAT, v. 4, n. 5, 2016.

HENNESSY, J. L.; PATTERSON, D. A. (2020) **Computer Organization and Design: The Hardware/Software Interface (RISC-V Edition).** p. 314.

INSPER. **Insper/Z01.1-cocotb-telemetry-playgroud.** Disponível em: https://github.com/Insper/Z01.1-cocotb-telemetry-playgroud. Acesso em: 17 mar. 2024.

INTEL. **Intel® Quartus® Prime Lite Edition Design Software Version 23.1 for Linux.** Intel Corporation. Disponível em: https://www.intel.com/content/www/us/en/software-kit/795187/intel-quartus-prime-lite-edition-design-software-version-23-1-for-linux.html. Acesso em: 07 mai. 2024.

KREKEL, H. TEAM, PYTEST-DEV. **Pytest.** MIT License, 2015. Disponível em: https://docs.pytest.org/. Acesso em: 28 abr. 2024.

LLVM. **User Guide for RISC-V Target.** Disponível em: https://llvm.org/docs/RISCVUsage.html. Acesso em: 15 maio. 2024.

MICROSOFT. **Visual Studio Code.** MIT License, 2015. Disponível em: https://code.visualstudio.com/. Acesso em: 28 abr. 2024.

MARIOTTI, G.; GIORGI, R. **WebRISC-V: A 32/64-bit RISC-V pipeline simulation tool.** SoftwareX, v. 18, p. 101105–101105, 1 jun. 2022.

MASCIO, D. et al. **The Case for RISCV in Space.** (S. Saponara, D. Gloria, Eds.)Applications in Electronics Pervading Industry, Environment and Society. Anais...Cham: Springer International Publishing, 2019.

MICROCHIP.**Mi-V Ecosystem | Microchip Technology.** Disponível em: https://www.microchip.com/en-us/products/fpgas-and-plds/fpga-and-soc-design-tools/mi-v. Acesso em: 15 maio. 2024.

MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E INOVAÇÃO. **Brasil se torna membro da Aliança RISC-V International.** Disponível em: https://www.gov.br/mcti/pt-br/acompanhe-o-mcti/noticias/2024/03/brasil-se-torna-membro-da-alianca-risc-v-international. Acesso em: 17 mar. 2024.

PATTERSON, D. A.; WATERMAN, A. **The RISC-V reader: An open architecture atlas.** Strawberry Canyon LLC, 2017.

PYTEST. **Working with custom markers — pytest documentation.** Disponível em: https://docs.pytest.org/en/latest/example/markers.html. Acesso em: 15 maio. 2024.

RISC-V FOUNDATION. **Members – RISC-V International.** Disponível em: https://riscv.org/members/. Acesso em: 15 maio. 2024.

RISC-V INTERNATIONAL. **The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213.** Editors Andrew Waterman and Krste Asanovic, 2019. Disponível em: https://riscv.org/technical/specifications/. Acesso em: 27 fev. 2024.

ROSSI, D. et al. PULP: A parallel ultra low power platform for next generation IoT applications. 1 ago. 2015.

SPINALHDL. **SpinalHDL/VexRiscv: A FPGA friendly 32 bit RISC-V CPU implementation.** Disponível em: https://github.com/SpinalHDL/VexRiscv?tab=readme-ov-file. Acesso em: 15 maio. 2024.

TURLEY, N. **netlistsvg.** MIT License, 2016. Disponível em: https://github.com/nturley/netlistsvg. Acesso em: 28 abr. 2024.

UBUNTU. Download Ubuntu for RISC-V Platforms | Ubuntu. Disponível em: https://ubuntu.com/download/risc-v. Acesso em: 15 maio. 2024.

WOLF, C. **Yosys Open SYnthesis Suite.** ISC License, 2020. Disponível em: https://yosyshq.net/yosys/. Acesso em: 28 abr. 2024.

WRIGHT, A. **What Is RISC, What Is RISC-V, and How Do They Differ?** Disponível em: https://www.makeuseof.com/what-is-risc-what-is-risc-v-how-do-they-differ/. Acesso em: 27 fev. 2024.

ZEPHYR.**Zephyr support status on RISC-V processors — Zephyr Project Documentation.** Disponível em: https://docs.zephyrproject.org/latest/hardware/arch/risc-v.html. Acesso em: 15 maio. 2024.

Anexo A

Conjunto de Instruções

Sintaxe

As instruções são vetores binários de 32 *bits*, podendo ser classificadas de duas formas: segundo sua sintaxe, ou de acordo com sua função. No que se refere à sintaxe, as instruções podem ser do tipo R, I, S, B, U ou J, como demonstrado na Tabela Sintaxe.

TIPO	31	30-25	24-21	20	19-15	14-12	11-8	7	6-0
R	funct7		rs2		rs1	funct3	rd		opcode
I	imm[31:11]	imm[10:0]			rs1	funct3	rd		opcode
S	imm[31:11]	imm[10:5]	rs2		rs1	funct3	imm[4:0]		opcode
B	imm[31:12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode
U	imm[31:12]						rd		opcode
J	imm[31:20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode

Table: **Tabela Sintaxe** - Tabela de Sintaxe dos tipos de instrução.

Instruções do tipo R são usadas para realizar operações entre registradores.

Instruções do tipo I são utilizadas para realizar operações em registradores com uso de valores imediatos.

Instruções do tipo S armazenam valores na memória.

Instruções do tipo B realizam desvios no programa dependendo do resultado da comparação de valores em dois registradores.

Instruções do tipo U são empregadas em operações que usam os 20 *bits* mais significativos da instrução como imediato, com os *bits* remanescentes sendo 0.

Apenas a instrução JAL (*Jump and Link* - Salto e Conexão) é do tipo J.

Sendo, para cada segmento da instrução:

- `opcode` : Codifica o tipo de instrução ou uma instrução específica;
- `funct3` : Codifica a operacionalização da instrução;
- `funct7` : Codifica uma variação da operacionalização;
- `rs1` : Endereça registrador de recurso primário;
- `rs2` : Endereça registrador de recurso secundário;
- `rd` : Endereça registrador de destinação;

- `imm`: Vetor do imediato.

Observação: Em alguns caso, é possível ver que um_bit_ da instrução representa um intervalo de *bits*, como imm[31:20] por exemplo. Esses são casos onde os *bits* mais significativos do imediato são o_bit_ mais significativo da instrução extendido.

Opcode

Opcodes são segmentos de 7 *bits* do vetor de instrução. Cada tipo de instrução possui um opcode ou uma instrução possui um opcode exclusivo. Para alguns tipos de instrução, são codificados com mais de um opcode, estando estes exemplificados na Tabela Opcode.

None	valor
OP	`0110011`
OP-IMM	`0010011`
STORE	`0100011`
LOAD	`0000011`
BRANCH	`1100011`
Outros	`XXXXX11`

Tabela Opcode - Tabela com exemplos de opcodes comuns.

Imediato

Os imediatos são vetores binários de 32 *bits*. Cada tipo de instrução com imediato possui uma sintaxe de imediato demonstrada na Tabela Imediato.

Tipo	31	30 - 20	19 - 12	11	10 - 5	4 - 1	0
I	inst[31]				inst[30:20]		
S	inst[31]				inst[30:25]	inst[11:7]	
B	inst[31]			inst[7]	inst[30:25]	inst[11:8]	0
U	inst[31:12]			0...0			
J	inst[31]		inst[19:12]	inst[20]	inst[30:21]		0

Tabela Imediato - Tabela com a sintaxe dos imediatos de acordo com seu tipo de instrução.

Sendo, para cada segmento, `inst` o vetor da instrução.

Por sua vez, a classificação das instruções segundo sua funcionalidade divide-as em grupos independentemente de sua estrutura. Esses grupos incluem:

- As instruções de construção, que criam valores em registradores;
- As instruções de deslocamento, que realizam operações de deslocamento de *bits* nos valores armazenados nos registradores;
- As instruções aritméticas, que efetuam operações matemáticas;
- As instruções lógicas, que são responsáveis por operações lógicas;
- As instruções de desvio, que alteram o fluxo de execução do programa com base em condições;
- As instruções de salto, que permitem saltos para outras partes do programa;
- As instruções de carregar, que carregam valores da memória para os registradores;
- As instruções de armazenar, que guardam valores dos registradores na memória.

Nas instruções que se seguem, RV32I Base significa que elas pertencem ao conjunto base de instruções para inteiros de 32 *bits*, e “M” *Standard Extension* significa que elas pertencem à extensão de Multiplicação:

Carrega Constante

`LUI`

Load Upper Immediate (Carregar Superior Imediato).

Carrega registradores com valores constantes de 32 *bits*. `LUI` guarda o valor imediato dos 20 *bits* mais significativos da instrução nos 20 *bits* mais significativos do registrador de destino `rd`, preenchendo os 12 *bits* menos significativos com zero.

Sintaxe

A instrução LUI é do tipo U, tendo um opcode próprio, como ilustrado na Tabela LUI.

Tipo	31-12	11-7	6-0
U	imm[31:12]	rd	`0110111`

Tabela LUI - sintaxe da instrução LUI.

Formato

`lui rd, imm`

Implementação

`x[rd] = imm[31:12] << 12`

`AUIPC`

Add Upper Immediate (Adiciona Superior Imediato).

Desloca o valor do imediato da instrução, que consiste nos 20 *bits* mais significativos, 12 *bits* à esquerda, preenchendo os 12 *bits* menos significativos com zero, e o adiciona ao PC. O resultado é então escrito no registrador de destino `rd`.

Sintaxe

A instrução AUIPC é do tipo U, tendo um opcode próprio, como ilustrado na Tabela AUIPC.

Tipo	31-12	11-7	6-0
U	imm[31:12]	rd	`0010111`

Tabela AUIPC - Tabela com sintaxe da instrução `AUIPC`.

Formato

```
auipc rd, imm
```

Implementação

```
x[rd] = pc + sext(imm[31:12] << 12)
```

Lógica Aritmética

ADD

Add (Adição).

Soma o valor armazenado no registrador `rs1` com o valor armazenado no registrador `rs2` e armazena o resultado no registrador de destino `rd`. Em caso de overflow, ele é ignorado.

Sintaxe

A instrução ADD é do tipo R, tendo uma funct3 e uma funct7 próprias, como ilustrado na Tabela ADD.

Tipo	31-25	24-20	19-15	14-12	11-7	6-0
R	`0000000`	rs2	rs1	`000`	rd	OP

Tabela ADD - Tabela com sintaxe da instrução `ADD`.

Formato

`add rd, rs1, rs2`

Implementação

`x[rd] = x[rs1] + x[rs2]`

ADDI

Add Immediate (Adição Imediata).

Soma o valor armazenado no registrador `rs1` com o sinal estendido do imediato e armazena o resultado no registrador de destino `rd`. Em caso de overflow, ele é ignorado.

Sintaxe

A instrução ADDI é do tipo I, tendo uma funct3 própria, como ilustrado na Tabela ADDI.

Tipo	31-20	19-15	14-12	11-7	6-0
I	imm[11:0]	rs1	`000`	rd	OP-IMM

Tabela ADDI - Tabela com sintaxe da instrução `ADDI`.

Formato

`addi rd, rs1, immediate`

Implementação

`x[rd] = x[rs1] + sext(immediate)`

SUB

Subtract (Subtração).

Subtrai o valor armazenado no registrador `rs2` do valor armazenado no registrador `rs1` e armazena o resultado no registrador de destino `rd`. Em caso de overflow, ele é ignorado.

Sintaxe

A instrução SUB é do tipo R, tendo uma funct3 e uma funct7 próprias, como ilustrado na Tabela SUB.

Tipo	31-25	24-20	19-15	14-12	11-7	6-0
R	`0100000`	rs2	rs1	`000`	rd	OP

Tabela SUB - Tabela com sintaxe da instrução `SUB`.

Formato

`sub rd, rs1, rs2`

Implementação

`x[rd] = x[rs1] - x[rs2]`

MUL

Multiply (Multiplicação).

Multiplica o valor armazenado no registrador `rs1` pelo valor armazenado no registrador `rs2` e armazena o resultado no registrador de destino `rd`. Em caso de overflow, ele é ignorado.

Sintaxe

A instrução MUL é do tipo R, tendo uma funct3 e uma funct7 próprias, como ilustrado na Tabela MUL.

Tipo	31-25	24-20	19-15	14-12	11-7	6-0
R	0000001	rs2	rs1	000	rd	OP

Tabela MUL - Tabela com sintaxe da instrução `MUL`.

Formato

`mul rd, rs1, rs2`

Implementação

`x[rd] = x[rs1] × x[rs2]`

MULH

Multiply High (Multiplicação Superior).

Multiplica o valor armazenado no registrador `rs1` pelo valor armazenado no registrador `rs2` considerando que são números de complemento de dois e armazena a metade superior do produto no registrador de destino `rd`.

Sintaxe

A instrução MULH é do tipo R, tendo uma funct3 e uma funct7 próprias, como ilustrado na Tabela MULH.

Tipo	31-25	24-20	19-15	14-12	11-7	6-0
R	0000001	rs2	rs1	001	rd	OP

Tabela MULH - Tabela com sintaxe da instrução `MULH`.

Formato

`mulh rd, rs1, rs2`

Implementação

`x[rd] = (x[rs1] × x[rs2]) >> XLEN`

MULHSU

Multiply High Signed and Unsigned (Multiplicação Superior com Sinal e Sem Sinal).

Multiplica o valor armazenado no registrador `rs1` pelo valor armazenado no registrador `rs2`, considerando que o valor em rs1 é de complemento de dois e que o valor em rs2 é um número sem sinal, armazenando a metade superior do produto no registrador de destino `rd`.

Sintaxe

A instrução MULHSU é do tipo R, tendo uma funct3 e uma funct7 próprias, como ilustrado na Tabela MULHSU.

Tipo	31-25	24-20	19-15	14-12	11-7	6-0
R	0000001	rs2	rs1	010	rd	OP

Tabela MULHSU - Tabela com sintaxe da instrução `MULHSU`.

Formato

`mulhsu rd, rs1, rs2`

Implementação

`x[rd] = (x[rs1] * x[rs2]) >> XLEN`

MULHU

Multiply High Unsigned (Multiplicação Superior Sem Sinal).

Multiplica o valor armazenado no registrador `rs1` pelo valor armazenado no registrador `rs2`, considerando que ambos são números sem sinal, e armazena a metade superior do produto no registrador de destino `rd`.

Sintaxe

A instrução MULHU é do tipo R, tendo uma funct3 e uma funct7 próprias, como ilustrado na Tabela MULHU.

Tipo	31-25	24-20	19-15	14-12	11-7	6-0
R	0000001	rs2	rs1	011	rd	OP

Tabela MULHU - Tabela com sintaxe da instrução `MULHU`.

Formato

`mulhu rd, rs1, rs2`

Implementação

`x[rd] = (x[rs1] × x[rs2]) >> XLEN`

DIV

Divide (Divisão).

Divide o valor armazenado no registrador `rs1` pelo valor armazenado no registrador `rs2`, considerando que ambos são números de complemento de dois, arredondando para zero, e armazena o quociente no registrador de destino `rd`.

Sintaxe

A instrução DIV é do tipo R, tendo uma funct3 e uma funct7 próprias, como ilustrado na Tabela DIV.

Tipo	31-25	24-20	19-15	14-12	11-7	6-0
R	0000001	rs2	rs1	100	rd	OP

Tabela DIV - Tabela com sintaxe da instrução `DIV`.

Formato

`div rd, rs1, rs2`

Implementação

`x[rd] = x[rs1] ÷ x[rs2]`

DIVU

Divide Unsigned (Divisão Sem Sinal).

Divide o valor armazenado no registrador `rs1` pelo valor armazenado no registrador `rs2`, considerando que são números sem sinal, arredondando para zero, e armazena o quociente no registrador de destino `rd`.

Sintaxe

A instrução DIVU é do tipo R, tendo uma funct3 e uma funct7 próprias, como ilustrado na Tabela DIVU.

Tipo	31-25	24-20	19-15	14-12	11-7	6-0
R	0000001	rs2	rs1	101	rd	OP

Tabela DIVU - Tabela com sintaxe da instrução `DIVU`.

Formato

`div rd, rs1, rs2`

Implementação

`x[rd] = x[rs1] ÷ x[rs2]`

REM

Remainder (Resto).

Divide o valor armazenado no registrador `rs1` pelo valor armazenado no registrador `rs2`, considerando que são números de complemento de dois, arredondando para zero, e armazena o resto no registrador de destino `rd`.

Sintaxe

A instrução REM é do tipo R, tendo uma funct3 e uma funct7 próprias, como ilustrado na Tabela REM.

Tipo	31-25	24-20	19-15	14-12	11-7	6-0
R	0000001	rs2	rs1	110	rd	OP

Tabela REM - Tabela com sintaxe da instrução `REM`.

Formato

`rem rd, rs1, rs2`

Implementação

`x[rd] = x[rs1] % x[rs2]`

REMU

Remainder Unsigned (Resto Sem Sinal).

Divide o valor armazenado no registrador `rs1` pelo valor armazenado no registrador `rs2`, considerando que são números sem sinal, arredondando para zero, e armazena o resto no registrador de destino `rd`.

Sintaxe

A instrução REMU é do tipo R, tendo uma funct3 e uma funct7 próprias, como ilustrado na Tabela REMU.

Tipo	31-25	24-20	19-15	14-12	11-7	6-0
R	0000001	rs2	rs1	111	rd	OP

Tabela REMU - Tabela com sintaxe da instrução `REMU`.

Formato

`rem rd, rs1, rs2`

Implementação

`x[rd] = x[rs1] % x[rs2]`

Lógicas Booleana

`XOR`

Exclusive OR (OU Exclusivo).

Realiza a operação lógica XOR,*bit* a_bit_, entre os valores armazenados nos registradores `rs1` e `rs2` e armazena o resultado no registrador de destino `rd`.

Sintaxe

A instrução XOR é do tipo R, tendo uma funct3 e uma funct7 próprias, como ilustrado na Tabela XOR.

Tipo	31-25	24-20	19-15	14-12	11-7	6-0
R	`0000000`	rs2	rs1	`100`	rd	OP

Tabela XOR - Tabela com sintaxe da instrução `XOR`.

Formato

```
xor rd, rs1, rs2
```

Implementação

```
x[rd] = x[rs1] ^ x[rs2]
```

`XORI`

Exclusive OR Immediate (OU Exclusivo Imediato).

Realiza a operação lógica XOR,*bit* a_bit_, entre o valor armazenado no registrador `rs1` e o imediato com sinal estendido e armazena o resultado no registrador de destino `rd`.

Sintaxe

A instrução XORI é do tipo I, tendo uma funct3 própria, como ilustrado na Tabela XORI.

Tipo	31-20	19-15	14-12	11-7	6-0
I	imm[11:0]	rs1	`100`	rd	OP-IMM

Tabela XORI - Tabela com sintaxe da instrução `XORI`.

Formato

```
xori rd, rs1, immediate
```

Implementação

```
x[rd] = x[rs1] ^ sext(immediate)
```

OR

OR (OU).

Realiza a operação lógica OR,*bit* a_bit_, entre os valores armazenados nos registradores `rs1` e `rs2` e armazena o resultado no registrador de destino `rd`.

Sintaxe

A instrução OR é do tipo R, tendo uma funct3 e uma funct7 próprias, como ilustrado na Tabela OR.

Tipo	31-25	24-20	19-15	14-12	11-7	6-0
R	`0000000`	rs2	rs1	`110`	rd	OP

Tabela OR - Tabela com sintaxe da instrução `OR`.

Formato

```
or rd, rs1, rs2
```

Implementação

```
x[rd] = x[rs1] | x[rs2]
```

ORI

OR Immediate (OU Imediato).

Realiza a operação lógica OR,*bit* a_bit_, entre o valor armazenado no registrador `rs1` e o imediato com sinal estendido e armazena o resultado no registrador de destino `rd`.

Sintaxe

A instrução ORI é do tipo I, tendo uma funct3 própria, como ilustrado na Tabela ORI.

Tipo	31-20	19-15	14-12	11-7	6-0
I	imm[11:0]	rs1	`110`	rd	OP-IMM

Tabela ORI - Tabela com sintaxe da instrução `ORI`.

Formato

`ori rd, rs1, immediate`

Implementação

`x[rd] = x[rs1] | sext(immediate)`

AND

AND (E).

Realiza a operação lógica AND,*bit* a_bit_, entre os valores armazenados nos registradores `rs1` e `rs2` e armazena o resultado no registrador de destino `rd`.

Sintaxe

A instrução AND é do tipo R, tendo uma funct3 e uma funct7 próprias, como ilustrado na Tabela AND.

Tipo	31-25	24-20	19-15	14-12	11-7	6-0
R	`0000000`	rs2	rs1	`111`	rd	OP

Tabela AND - Tabela com sintaxe da instrução `AND`.

Formato

`and rd, rs1, rs2`

Implementação

`x[rd] = x[rs1] & x[rs2]`

ANDI

AND Immediate (E Imediato).

Realiza a operação lógica OR,*bit* a_bit_, entre o valor armazenado no registrador `rs1` e o imediato com sinal estendido e armazena o resultado no registrador de destino `rd`.

Sintaxe

A instrução ANDI é do tipo I, tendo uma funct3 própria, como ilustrado na Tabela ANDI.

Tipo	31-20	19-15	14-12	11-7	6-0
I	imm[11:0]	rs1	`111`	rd	OP-IMM

Tabela ANDI - Tabela com sintaxe da instrução `ANDI`.

Formato

```
andi rd, rs1, immediate
```

Implementação

```
x[rd] = x[rs1] & sext(immediate)
```

Operação de Deslocamento

`SLL`

Shift Left Logical (Deslocamento à Esquerda Lógico).

Desloca o valor armazenado no registrador `rs1` à esquerda pelo número de posições indicado pelos 5 *bits* menos significativos do valor armazenado no registrador `rs2`. Os *bits* remanescentes de `rs2` são ignorados. Os *bits* vazios de `rs1` são preenchidos com zeros. O resultado é escrito no registrador de destino `rd`.

Sintaxe

A instrução SLL é do tipo R, tendo uma funct3 e uma funct7 próprias, como ilustrado na Tabela SLL.

Tipo	31-25	24-20	19-15	14-12	11-7	6-0
R	`0000000`	rs2	rs1	`001`	rd	OP

Tabela SLL - Tabela com sintaxe da instrução `SLL`.

Formato

`sll rd, rs1, rs2`

Implementação

`x[rd] = x[rs1] << x[rs2]`

`SLLI`

Shift Left Logical Immediate (Deslocamento à Esquerda Lógico Imediato).

Desloca o valor armazenado no registrador `rs1` à esquerda pelo número de posições indicado pelo `shamt`. Os *bits* vazios de `rs1` são preenchidos com zeros. O resultado é escrito no registrador de destino `rd`. Caso se decida atualizar o processador para uma arquitetura de 64 bits, esta instrução terá sua sintaxe alterada (o shamt e o funct7 passam a ter 6 bits cada).

Sintaxe

A instrução SLLI é do tipo I, tendo uma funct3 e uma funct7 próprias, como ilustrado na Tabela SLLI.

Tipo	31-25	24-20	19-15	14-12	11-7	6-0
I	0000000	shamt	rs1	001	rd	OP-IMM

Tabela SLLI - Tabela com sintaxe da instrução `SLLI`.

Formato

```
slli rd, rs1, shamt
```

Implementação

```
x[rd] = x[rs1] << shamt
```

SRL

Shift Right Logical (Deslocamento à Direita Lógico).

Desloca o valor armazenado no registrador `rs1` à direita pelo número de posições indicado pelos 5 *bits* menos significativos do valor armazenado no registrador `rs2`. Os *bits* remanescentes de `rs2` são ignorados. Os *bits* vazios de `rs1` são preenchidos com zeros. O resultado é escrito no registrador de destino `rd`.

Sintaxe

A instrução SRL é do tipo R, tendo uma funct3 e uma funct7 próprias, como ilustrado na Tabela SRL.

Tipo	31-25	24-20	19-15	14-12	11-7	6-0
R	0000000	rs2	rs1	101	rd	OP

Tabela SRL - Tabela com sintaxe da instrução `SRL`.

Formato

```
srl rd, rs1, rs2
```

Implementação

```
x[rd] = x[rs1] >> x[rs2]
```

SRLI

Shift Right Logical Immediate (Deslocamento à Direita Lógico Imediato).

Desloca o valor armazenado no registrador `rs1` à direita pelo número de posições indicado pelo `shamt`. Os *bits* vazios de `rs1` são preenchidos com zeros. O resultado é escrito no registrador de destino `rd`. Caso se decida atualizar o processador para uma arquitetura de 64 bits, esta instrução terá sua sintaxe alterada (o shamt e o funct7 passam a ter 6 bits cada).

Sintaxe

A instrução SRLI é do tipo I, tendo uma funct3 e uma funct7 próprias, como ilustrado na Tabela SRLI.

Tipo	31-25	24-20	19-15	14-12	11-7	6-0
I	0000000	shamt	rs1	101	rd	OP-IMM

Tabela SRLI - Tabela com sintaxe da instrução `SRLI`.

Formato

`srli rd, rs1, shamt`

Implementação

`x[rd] = x[rs1] >>shamt`

SRA

Shift Right Arithmetic (Deslocamento à Direita Aritmético).

Desloca o valor armazenado no registrador `rs1` à direita pelo número de posições indicado pelos 5 *bits* menos significativos do valor armazenado no registrador `rs2`. Os *bits* remanescentes de `rs2` são ignorados. Os *bits* vazios de `rs1` são preenchidos com cópias do_bit_ mais significativo de `rs1`. O resultado é escrito no registrador de destino `rd`.

Sintaxe

A instrução SRA é do tipo R, tendo uma funct3 e uma funct7 próprias, como ilustrado na Tabela SRA.

Tipo	31-25	24-20	19-15	14-12	11-7	6-0
R	0100000	rs2	rs1	101	rd	OP

Tabela SRA - Tabela com sintaxe da instrução `SRA`.

Formato

`sra rd, rs1, rs2`

Implementação

`x[rd] = x[rs1] >> x[rs2]`

SRAI

Shift Right Arithmetic Immediate (Deslocamento à Direita Aritmético Imediato).

Desloca o valor armazenado no registrador `rs1` à direita pelo número de posições indicado pelo `shamt`. Os *bits* vazios de `rs1` são preenchidos com cópias do_bit_ mais significativo de `rs1`. O resultado é escrito no registrador de destino `rd`. Caso se decida atualizar o processador para uma arquitetura de 64 bits, esta instrução terá sua sintaxe alterada (o shamt e o funct7 passam a ter 6 bits cada).

Sintaxe

A instrução SRLI é do tipo I, tendo uma funct3 e uma funct7 próprias, como ilustrado na Tabela SRLI.

Tipo	31-25	24-20	19-15	14-12	11-7	6-0
I	0100000	shamt	rs1	101	rd	OP-IMM

Tabela SRAI - Tabela com sintaxe da instrução `SRAI`.

Formato

`srai rd, rs1, shamt`

Implementação

`x[rd] = x[rs1] >> shamt`

Comparação

SLT

Set if Less Than (Definir se Menor que).

Verifica se o valor armazenado no registrador `rs1` é menor que o valor armazenado no registrador `rs2`, considerandoque são complemento de dois, em caso positivo, armazena 1 no registrador de destino `rd`, caso contrário, armazena 0.

Sintaxe

A instrução SLT é do tipo R, tendo uma funct3 e uma funct7 próprias, como ilustrado na Tabela SLT.

Tipo	31-25	24-20	19-15	14-12	11-7	6-0
R	0000000	rs2	rs1	010	rd	OP

Tabela SLT - Tabela com sintaxe da instrução `SLT`.

Formato

```
slt rd, rs1, rs2
```

Implementação

```
x[rd] = x[rs1] < x[rs2]
```

SLTI

Set if Less Than Immediate (Definir se Menor que Imediato).

Verifica se o valor armazenado no registrador `rs1` é menor que o imediato com extensão de sinal, considerando que são complemento de dois, em caso positivo, armazena 1 no registrador de destino `rd`, caso contrário, armazena 0.

Sintaxe

A instrução SLTI é do tipo I, tendo uma funct3 própria, como ilustrado na Tabela SLTI.

Tipo	31-20	19-15	14-12	11-7	6-0
I	imm[11:0]	rs1	010	rd	OP-IMM

Tabela SLTI - Tabela com sintaxe da instrução `SLTI`.

Formato

```
slti rd, rs1, immediate
```

Implementação

```
x[rd] = x[rs1] < sext(immediate)
```

SLTIU

Set if Less Than Immediate Unisgned(Definir se Menor que Imediato Sem Sinal).

Verifica se o valor armazenado no registrador `rs1` é menor que o imediato com extensão de sinal, considerando que são sem sinal, em caso positivo, armazena 1 no registrador de destino `rd`, caso contrário, armazena 0.

Sintaxe

A instrução SLTIU é do tipo I, tendo uma funct3 própria, como ilustrado na Tabela SLTIU.

Tipo	31-20	19-15	14-12	11-7	6-0
I	imm[11:0]	rs1	011	rd	OP-IMM

Tabela SLTIU - Tabela com sintaxe da instrução `SLTIU`.

Formato

```
slti rd, rs1, immediate
```

Implementação

```
x[rd] = x[rs1] < sext(immediate)
```

SLTU

Set if Less Than Unsigned(Definir se Menor que Sem Sinal).

Verifica se o valor armazenado no registrador `rs1` é menor que o valor armazenado no registrador `rs2`, considerando que são valores sem sinal, em caso positivo, armazena 1 no registrador de destino `rd`, caso contrário, armazena 0.

Sintaxe

A instrução SLTU é do tipo R, tendo uma funct3 e uma funct7 próprias, como ilustrado na Tabela SLTU.

Tipo	31-25	24-20	19-15	14-12	11-7	6-0
R	`0000000`	rs2	rs1	`011`	rd	OP

Tabela SLTU - Tabela com sintaxe da instrução `SLTU`.

Formato

`sltu rd, rs1, rs2`

Implementação

`x[rd] = x[rs1] < x[rs2]`

Desvio Incondicional

`JAL`

Jump and Link (Salto e Link).

Escreve o endereço da próxima instrução (PC+4) no registrador de destino `rd` e modifica o PC para o valor atual somado ao offset com extensão de sinal. Se `rd` for omitido, o valor de retorno é armazenado em `x1`.

Sintaxe

A instrução JAL é do tipo J, tendo um opcode próprio, como ilustrado na Tabela JAL.

Tipo	31 - 12	11 - 7	6 - 0
J	offset[20,10:1,11,19:12]	rd	`1101111`

Tabela JAL - Tabela com sintaxe da instrução `JAL`.

Formato

`jal rd, offset`

Implementação

`x[rd] = pc+4; pc += sext(offset)`

`JALR`

Jump and Link Register (Salto e Link por Registrador).

Realiza um cópia do PC para `rs1 + sext(offset)`, mascara_bit_ menos significativo do endereço resultante e armazena o endereço anterior de PC+4 no registrador de destino `rd`. Se `rd` for omitido, o valor é armazenado em `x1`.

Sintaxe

A instrução JALR é do tipo I, tendo uma funct3 e um opcode próprios, como ilustrado na Tabela JALR.

Tipo	31-20	19-15	14-12	11-7	6-0
I	offset[11:0]	rs1	`000`	rd	`1100111`

Tabela JALR - Tabela com sintaxe da instrução `JALR` .

Formato

`jalr rd, offset(rs1)`

Implementação

`t =pc+4; pc=(x[rs1]+sext(offset))&∼1; x[rd]=t`

Desvio Condicional

`BEQ`

Branch if Equal (Desvio se Igual)

Verifica se o valor armazenado no registrador `rs1` é igual ao valor armazenado no registrador `rs2`, em caso positivo, modifica o PC para o valor atual somado ao offset com extensão de sinal.

Sintaxe

A instrução BEQ é do tipo B, tendo uma funct3 própria, como ilustrado na Tabela BEQ.

Tipo	31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
B	offset[12,10:5]	rs2	rs1	`000`	offset[4:1,11]	BRANCH

Tabela BEQ - Tabela com sintaxe da instrução `BEQ`.

Formato

```
beq rs1, rs2, offset
```

Implementação

```
if (rs1 == rs2) pc += sext(offset)
```

`BNE`

Branch if Not Equal (Desvio se Não Igual)

Verifica se o valor armazenado no registrador `rs1` é diferente do valor armazenado no registrador `rs2`, em caso positivo, modifica o PC para o valor atual somado ao offset com extensão de sinal.

Sintaxe

A instrução BNE é do tipo B, tendo uma funct3 própria, como ilustrado na Tabela BNE.

Tipo	31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
B	offset[12,10:5]	rs2	rs1	001	offset[4:1,11]	BRANCH

Tabela BNE - Tabela com sintaxe da instrução `BNE`.

Formato

`bnq rs1, rs2, offset`

Implementação

`if (rs1 != rs2) pc += sext(offset)`

BLT

Branch if Less Than (Desvio se Menor que)

Verifica se o valor armazenado no registrador `rs1` é menor que o valor armazenado no registrador `rs2`, considerando que são números em complemento de dois, em caso positivo, modifica o PC para o valor atual somado ao offset com extensão de sinal.

Sintaxe

A instrução BLT é do tipo B, tendo uma funct3 própria, como ilustrado na Tabela BLT.

Tipo	31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
B	offset[12,10:5]	rs2	rs1	100	offset[4:1,11]	BRANCH

Tabela BLT - Tabela com sintaxe da instrução `BLT`.

Formato

`blt rs1, rs2, offset`

Implementação

`if (rs1 < rs2) pc += sext(offset)`

BGE

Branch if Greater Than or Equal(Desvio se Maior ou Igual que)

Verifica se o valor armazenado no registrador `rs1` é maior ou igual ao valor armazenado no registrador `rs2`, considerando que são números em complemento de dois, em caso positivo, modifica o PC para o valor atual somado ao offset com extensão de sinal.

Sintaxe

A instrução BGE é do tipo B, tendo uma funct3 própria, como ilustrado na Tabela BGE.

Tipo	31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
B	offset[12,10:5]	rs2	rs1	`101`	offset[4:1,11]	BRANCH

Tabela BGE - Tabela com sintaxe da instrução `BGE`.

Formato

```
bge rs1, rs2, offset
```

Implementação

```
if (rs1 >= rs2) pc += sext(offset)
```

BLTU

Branch if Less Than Unsigned(Desvio se Menor que Sem Sinal)

Verifica se o valor armazenado no registrador `rs1` é menor ao valor armazenado no registrador `rs2`, considerando que são números sem sinal, em caso positivo, modifica o PC para o valor atual somado ao offset com extensão de sinal.

Sintaxe

A instrução BLTU é do tipo B, tendo uma funct3 própria, como ilustrado na Tabela BLTU.

Tipo	31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
B	offset[12,10:5]	rs2	rs1	`110`	offset[4:1,11]	BRANCH

Tabela BLTU - Tabela com sintaxe da instrução `BLTU`.

Formato

`bltu rs1, rs2, offset`

Implementação

`if (rs1 < rs2) pc += sext(offset)`

BGEU

Branch if Greater or Equal Than Unsigned(Desvio se Maior ou Igual que Sem Sinal)

Verifica se o valor armazenado no registrador `rs1` é maior ou igual ao valor armazenado no registrador `rs2`, considerando que são números sem sinal, em caso positivo, modifica o PC para o valor atual somado ao offset com extensão de sinal.

Sintaxe

A instrução BGEU é do tipo B, tendo uma funct3 própria, como ilustrado na Tabela BGEU.

Tipo	31 - 25	24 - 20	19 - 15	14 - 12	11 - 7	6 - 0
B	offset[12,10:5]	rs2	rs1	111	offset[4:1,11]	BRANCH

Tabela BGEU - Tabela com sintaxe da instrução `BGEU`.

Formato

`bgeu rs1, rs2, offset`

Implementação

`if (rs1 >= rs2) pc += sext(offset)`

Busca na Memória

`LB`

Load Byte (Carrega Byte).

Carrega um byte da memória no endereço `rs1 + sext(offset)` e armazena o valor no registrador de destino `rd`, com extensão de sinal.

Sintaxe

A instrução LB é do tipo I, tendo uma funct3 própria, como ilustrado na Tabela LB.

Tipo	31-20	19-15	14-12	11-7	6-0
I	offset[11:0]	rs1	`000`	rd	LOAD

Tabela LB - Tabela com sintaxe da instrução `LB`.

Formato

`lb rd, offset(rs1)`

Implementação

`x[rd] = sext(M[x[rs1] + sext(offset)][7:0])`

`LH`

Load Halfword (Carrega Halfword).

Carrega dois bytes da memória no endereço `rs1 + sext(offset)` e armazena o valor no registrador de destino `rd`, com extensão de sinal.

Sintaxe

A instrução LH é do tipo I, tendo uma funct3 própria, como ilustrado na Tabela LH.

Tipo	31-20	19-15	14-12	11-7	6-0
I	offset[11:0]	rs1	001	rd	LOAD

Tabela LH - Tabela com sintaxe da instrução `LH`.

Formato

`lh rd, offset(rs1)`

Implementação

`x[rd] = sext(M[x[rs1] + sext(offset)][15:0])`

LBU

Load Byte Unsigned(Carrega Byte Sem Sinal).

Carrega um byte da memória no endereço `rs1 + sext(offset)` e armazena o valor no registrador de destino `rd`, com extensão de zero.

Sintaxe

A instrução LBU é do tipo I, tendo uma funct3 própria, como ilustrado na Tabela LBU.

Tipo	31-20	19-15	14-12	11-7	6-0
I	offset[11:0]	rs1	100	rd	LOAD

Tabela LBU - Tabela com sintaxe da instrução `LBU`.

Formato

`lbu rd, offset(rs1)`

Implementação

`x[rd] = M[x[rs1] + sext(offset)][7:0]`

LHU

Load Halfword Unsigned(Carrega Halfword Sem Sinal).

Carrega dois bytes da memória no endereço `rs1 + sext(offset)` e armazena o valor no registrador de destino `rd`, com extensão de zero.

Sintaxe

A instrução LHU é do tipo I, tendo uma funct3 própria, como ilustrado na Tabela LHU.

Tipo	31-20	19-15	14-12	11-7	6-0
I	offset[11:0]	rs1	`101`	rd	LOAD

Tabela LHU - Tabela com sintaxe da instrução `LHU`.

Formato

`lhu rd, offset(rs1)`

Implementação

`x[rd] = M[x[rs1] + sext(offset)][15:0]`

LW

Load Word (Carrega Word).

Carrega quatro bytes da memória no endereço `rs1 + sext(offset)` e armazena o valor no registrador de destino `rd`.

Sintaxe

A instrução LW é do tipo I, tendo uma funct3 própria, como ilustrado na Tabela LW.

Tipo	31-20	19-15	14-12	11-7	6-0
I	offset[11:0]	rs1	`010`	rd	LOAD

Tabela LW - Tabela com sintaxe da instrução `LW`.

Formato

`lw rd, offset(rs1)`

Implementação

`x[rd] = sext(M[x[rs1] + sext(offset)][31:0])`

Escrita na Memória

SB

Store Byte (Armazena Byte).

Armazena o byte menos significativo do valor armazenado no registrador `rs2` na memória no endereço `rs1 + sext(offset)`.

Sintaxe

A instrução SB é do tipo S, tendo uma funct3 própria, como ilustrado na Tabela SB.

Tipo	31-25	24-20	19-15	14-12	11-7	6-0
S	offset[11:5]	rs2	rs1	`000`	offset[4:0]	STORE

Tabela SB - Tabela com sintaxe da instrução `SB`.

Formato

```
sb rs2, offset(rs1)
```

Implementação

```
M[x[rs1] + sext(offset)] = x[rs2][7:0]
```

SH

Store Halfword (Armazena Halfword).

Armazena os dois bytes menos significativo do valor armazenado no registrador `rs2` na memória no endereço `rs1 + sext(offset)`.

Sintaxe

A instrução SH é do tipo S, tendo uma funct3 própria, como ilustrado na Tabela SH.

Tipo	31-25	24-20	19-15	14-12	11-7	6-0
S	offset[11:5]	rs2	rs1	001	offset[4:0]	STORE

Tabela SH - Tabela com sintaxe da instrução `SH`.

Formato

`sh rs2, offset(rs1)`

Implementação

`M[x[rs1] + sext(offset)] = x[rs2][15:0]`

SW

Store Word (Armazena Word).

Armazena os quatro bytes menos significativo do valor armazenado no registrador `rs2` na memória no endereço `rs1 + sext(offset)`.

Sintaxe

A instrução SW é do tipo S, tendo uma funct3 própria, como ilustrado na Tabela SW.

Tipo	31-25	24-20	19-15	14-12	11-7	6-0
S	offset[11:5]	rs2	rs1	010	offset[4:0]	STORE

Tabela SW - Tabela com sintaxe da instrução `SW`.

Formato

`sw rs2, offset(rs1)`

Implementação

`M[x[rs1] + sext(offset)] = x[rs2][31:0]`

Apêndice A
Pacote WORK.CPU

Para a topologia, um conjunto de registros foi criado para definir o fluxo de dados em alto nível. Isso possibilita simplificar a implementação de pipelining e manter o código limpo. A partir dos seguintes registros é possível declarar todos os pontos de controle e de dados de todo o fluxo de execução da arquitetura. Além disso, também são especificados valores que caracterizem o comportamento ocioso da arquitetura.

Controles do estágio Busca Instrução

Este é o registro dos pontos de controle do estágio Busca Instrução. A partir do qual é possível controlar os seguintes aspectos, respecitvamente:

- `enable_stall` : caso ativado, trava a contagem do programa;
- `select_source` : caso ativado, seleciona o endereço de desvio como origem do contador de programa.

```vhdl
type t_CONTROL_IF is record
    enable_stall  : std_logic;
    select_source : std_logic;
end record;

constant NULL_CONTROL_IF : t_CONTROL_IF := (
    enable_stall  => '0',
    select_source => '0'
);
```

Controles do estágio Decodifica Instrução

Este é o registro dos pontos de controle do estágio Decodifica Instrução. A partir do qual é possível controlar os seguintes aspectos, respecitvamente:

- `enable_branch` : caso ativado, indica que a instrução é do tipo B;
- `enable_jalr` : caso ativado, indica que a instrução é a JALR;
- `enable_jump` : caso ativado, indica que a instrução é do tipo J;
- `select_jump` : caso ativado, seleciona o endereço de desvio `register + immediate` ; caso contrário, `PC + immediate` .

vhdl

```
type t_CONTROL_ID is record
    enable_branch : std_logic;
    enable_jalr   : std_logic;
    enable_jump   : std_logic;
    select_jump   : std_logic;
end record;


constant NULL_CONTROL_ID : t_CONTROL_ID := (
    enable_branch   => '0',
    enable_jalr     => '0',
    enable_jump     => '0',
    select_jump     => '0'
);
```

Controles do estágio Executa

Este é o registro dos pontos de controle do estágio Executa. A partir do qual é possível controlar os seguintes aspectos, respecitvamente:

- `select_source_1`: Seleciona a fonte da entrada `source_1` entre, respecticamente, registrador, forwarding da saída do estágio Acessa a Memória, forwarding do saída do estágio Escrita de Retorno, e vetor nulo `00...00`;
- `select_source_2`: Seleciona a fonte da entrada `source_2` entre, respecticamente, registrador, forwarding da saída do estágio Acessa a Memória, forwarding do saída do estágio Escrita de Retorno, e vetor nulo `00...00`.

vhdl

```
type t_CONTROL_EX is record
    select_source_1  : std_logic_vector(1 downto 0);
    select_source_2  : std_logic_vector(1 downto 0);
end record;

constant NULL_CONTROL_EX : t_CONTROL_EX := (
    select_source_1  => (others => '0'),
    select_source_2  => (others => '0')
);
```

Controles do estágio Acessa a Memória

Este é o registro dos pontos de controle do estágio Acessa a Memória. A partir do qual é possível controlar os seguintes aspectos, respecitvamente:

- `enable_read` : caso ativado, indica que o núcleo está realizando uma leitura da memória;
- `enable_write` : caso ativado, indica que o núcleo está realizando uma escrina na memória.

```vhdl
type t_CONTROL_MEM is record
    enable_read  : std_logic;
    enable_write : std_logic;
end record;

constant NULL_CONTROL_MEM : t_CONTROL_MEM := (
    enable_read  => '0',
    enable_write => '0'
);
```

Controles do estágio Escrite de Retorno

Este é o registro dos pontos de controle do estágio Escrite de Retorno. A partir do qual é possível controlar os seguintes aspectos, respecitvamente:

- `enable_destination` : caso ativado, habilita a escrita no arquivo de registradores;
- `select_destination` : caso ativado, seleciona o retorno para a entrada de dados da memória; caso contrário, do resultado da unidade lógica e aritmética.

```vhdl
type t_CONTROL_WB is record
    enable_destination : std_logic;
    select_destination : std_logic;
end record;

constant NULL_CONTROL_WB : t_CONTROL_WB := (
    enable_destination => '0',
    select_destination => '0'
);
```

Sinais do registrador de pipeline IF/ID

Este é o registro dos sinais do estágio Decodifica instrução, que recebe os sinais do Busca Instrução e os registra no pipeline. A partir do seguinte registro é possível controlar os seguintes aspectos, respecitvamente:

- `address_program` : Valor de saída do contador de programa;
- `data_instruction` : Valor de saída da memória de programa.

```vhdl
type t_SIGNALS_IF_ID is record
    address_program  : t_DATA;
    data_instruction : t_DATA;
end record;

constant NULL_SIGNALS_IF_ID : t_SIGNALS_IF_ID := (
    address_program  => (others => '0'),
    data_instruction => WORK.RV32I.NULL_INSTRUCTION
);
```

Sinais do registrador de pipeline ID/EX

Este é o registro dos sinais do estágio Executa, que recebe os sinais do Decodifica instrução e os registra no pipeline. A partir do seguinte registro é possível controlar os seguintes aspectos, respecitvamente:

- `control_ex` : Pontos de controle do estágio Executa;
- `control_mem` : Pontos de controle do estágio Acessa a memória;
- `control_wb` : Pontos de controle do estágio Escrita de Retorno;
- `address_program` : Valor de saída do contador de programa;
- `data_source_1` : Vetor selecionado por `sr1` no arquivo de registradores;
- `data_source_2` : Vetor selecionado por `sr2` no arquivo de registradores;
- `data_immediate` : Vetor do imediato decodificado pela instrução;
- `funct_7` : Vetor da instrução segmentado pela região `funct_7` ;
- `funct_3` : Vetor da instrução segmentado pela região `funct_3` ;
- `opcode` : Vetor da instrução segmentado pela região `opcode` ;
- `select_destination` : Vetor da instrução segmentado pela região `ds` ;
- `select_source_1` : Vetor da instrução segmentado pela região `sr1` ;
- `select_source_2` : Vetor da instrução segmentado pela região `sr2` .

vhdl

```
type t_SIGNALS_ID_EX is record
    control_ex        : t_CONTROL_EX;
    control_mem       : t_CONTROL_MEM;
    control_wb        : t_CONTROL_WB;
    address_program   : t_DATA;
    data_source_1     : t_DATA;
    data_source_2     : t_DATA;
    data_immediate    : t_DATA;
    funct_7           : WORK.RV32I.t_FUNCT7;
    funct_3           : WORK.RV32I.t_FUNCT3;
    opcode            : WORK.RV32I.t_OPCODE;
    select_destination : t_REGISTER;
    select_source_1   : t_REGISTER;
    select_source_2   : t_REGISTER;
end record;

constant NULL_SIGNALS_ID_EX : t_SIGNALS_ID_EX := (
    control_ex        => NULL_CONTROL_EX,
    control_mem       => NULL_CONTROL_MEM,
```

```
    control_wb        => NULL_CONTROL_WB,
    address_program   => (others => '0'),
    data_source_1     => (others => '0'),
    data_source_2     => (others => '0'),
    data_immediate    => (others => '0'),
    funct_7           => WORK.RV32I.FUNCT7_ADD,
    funct_3           => WORK.RV32I.FUNCT3_ADDI,
    opcode            => WORK.RV32I.OPCODE_OP_IMM,
    select_source_1   => (others => '0'),
    select_source_2   => (others => '0'),
    select_destination => (others => '0')
);
```

Sinais do registrador de pipeline EX/MEM

Este é o registro dos sinais do estágio Acesso a Memória, que recebe os sinais do Executa e os registra no pipeline. A partir do seguinte registro é possível controlar os seguintes aspectos, respecitvamente:

- `control_mem` : Pontos de controle do estágio Acessa a memória;
- `control_wb` : Pontos de controle do estágio Escrita de Retorno;
- `data_destination` : Vetor resultante da operação entre `data_source_1` e `data_source_2` na unidade lógica e aritmética pelo operando em `funct_7` , `funct_3` , `opcode` ;
- `data_source_2` : Vetor selecionado por `sr2` no arquivo de registradores;
- `select_destination` : Vetor da instrução segmentado pela região `ds` ;
- `funct_3` : Vetor da instrução segmentado pela região `funct_3` .

vhdl

```
type t_SIGNALS_EX_MEM is record
    control_mem       : t_CONTROL_MEM;
    control_wb        : t_CONTROL_WB;
    data_destination  : t_DATA;
    data_source_2     : t_DATA;
    select_destination : t_REGISTER;
    funct_3           : WORK.RV32I.t_FUNCT3;
end record;

constant NULL_SIGNALS_EX_MEM : t_SIGNALS_EX_MEM := (
    control_mem       => NULL_CONTROL_MEM,
    control_wb        => NULL_CONTROL_WB,
```

```
    data_destination   => (others => '0'),
    data_source_2      => (others => '0'),
    select_destination => (others => '0'),
    funct_3            => WORK.RV32I.FUNCT3_ADDI
);
```

Sinais do registrador de pipeline MEM/WB

Este é o registro dos sinais do estágio Escrita de Retorno, que recebe os sinais do Acesso a Memória e os registra no pipeline. A partir do seguinte registro é possível controlar os seguintes aspectos, respecitvamente:

- `control_wb` : Pontos de controle do estágio Escrita de Retorno;
- `data_memory` : Vetor resgatado na memória pelo endereço definido por `data_destination` ;
- `data_destination` : Vetor resultante da operação entre `data_source_1` e `data_source_2` na unidade lógica e aritmética pelo operando em `funct_7` , `funct_3` , `opcode` ;
- `select_destination` : Vetor da instrução segmentado pela região `ds` .

vhdl

```
type t_SIGNALS_MEM_WB is record
    control_wb        : t_CONTROL_WB;
    data_memory       : t_DATA;
    data_destination  : t_DATA;
    select_destination : t_REGISTER;
end record;

constant NULL_SIGNALS_MEM_WB : t_SIGNALS_MEM_WB := (
    control_wb        => NULL_CONTROL_WB,
    data_memory       => (others => '0'),
    data_destination  => (others => '0'),
    select_destination => (others => '0')
);
```

Sinais de forward de desvio

Este é o registro dos controles e sinais de forwarding do comparador de desvio no estágio Decodifica Instrução. A partir do seguinte registro é possível controlar os seguintes aspectos, respecitvamente:

- `select_source_1`: Seleciona a origem do vetor `data_source_1` na unidade de comparação de desvio, respectivamente, entre o arquivo de registradores, o forward do destino do estágio Executa, o forward do destino do estágio Acessa a Memória e o forward do destino do estágio Escrita de Retorno;
- `select_source_2`: Seleciona a origem do vetor `data_source_2` na unidade de comparação de desvio, respectivamente, entre o arquivo de registradores, o forward do destino do estágio Executa, o forward do destino do estágio Acessa a Memória e o forward do destino do estágio Escrita de Retorno;
- `source_mem`: Vetor de destino do estágio Acessa a Memória;

```vhdl
type t_FORWARD_BRANCH is record
    select_source_1 : std_logic_vector(1 downto 0);
    select_source_2 : std_logic_vector(1 downto 0);
    source_mem      : WORK.RV32I.t_DATA;
end record;
```

Sinais do forward de execução

Este é o registro dos controles e sinais de forwarding da unidade lógica e aritmética no estágio Executa. A partir do seguinte registro é possível controlar os seguintes aspectos, respecitvamente:

- `select_source_1`: Seleciona a origem do vetor `data_source_1` na unidade de comparação de desvio, respectivamente, entre o arquivo de registradores no estágio Executa, o forward do destino do estágio Acessa a Memória e o forward destino do estágio Escrita de Retorno;
- `select_source_2`: Seleciona a origem do vetor `data_source_2` na unidade de comparação de desvio, respectivamente, entre o arquivo de registradores no estágio Executa, o forward do destino do estágio Acessa a Memória e o forward destino do estágio Escrita de Retorno;
- `source_mem`: Vetor de destino do estágio Acessa a Memória;
- `source_wb`: Vetor de destino do estágio Escrita de Retorno.

vhdl

```
type t_FORWARD_EXECUTION is record
    select_source_1 : std_logic_vector(1 downto 0);
    select_source_2 : std_logic_vector(1 downto 0);
    source_mem      : WORK.RV32I.t_DATA;
    source_wb       : WORK.RV32I.t_DATA;
end record;
```