Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Departamento de Engenharia Elétrica e Informática

Sistema de emulação para arquiteturas RISC-V

Dimas Germano Brandão Soares Silva

Campina Grande, PB
Dezembro de 2019

Dimas Germano Brandão Soares Silva

Sistema de emulação para arquiteturas RISC-V

Trabalho de Conclusão de Curso apresentado à Coordenação do Curso de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande, Campus Campina Grande, como parte dos requisitos necessários para a obtenção do grau de Bacharel em Engenharia Elétrica.

Área de Concentração: Microeletrônica Digital

Orientador: Prof. Dr. Gutemberg Gonçalves Dos Santos Júnior

Campina Grande, PB
Dezembro de 2019

Dimas Germano Brandão Soares Silva

Sistema de emulação para arquiteturas RISC-V

Trabalho de Conclusão de Curso apresentado à Coordenação do Curso de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande, Campus Campina Grande, como parte dos requisitos necessários para a obtenção do grau de Bacharel em Engenharia Elétrica.

Aprovado em ___/___ /___

Professor Avaliador
Universidade Federal de Campina Grande
Avaliador

Prof Dr. Gutemberg Gonçalves dos Santos Júnior
Universidade Federal de Campina Grande
Orientador

Agradecimentos

Eu quero agradecer a minha mãe, meu irmão e meu pai, por me apoiarem em todas as minhas loucuras.
Eu quero agradecer a todos da Republica Tcheca, pelos grandes ensinamentos da vida.
Eu quero agradecer a todos os amigos que fiz durante minha vida acadêmica, porque nunca vou esquecer de tudo que aprendi com eles.
Eu quero agradecer a todos do laboratório X-MEN, pelos "cafézinhos"que me ensinaram a chegar a quem sou hoje.
E eu quero agradecer a todos as pessoas do mundo, porque todo gigante um dia precisou, ou vai precisar, de um ombro para ver mais longe.

"Eu não quero conquistar nada, para mim o que tem maior liberdade no mar é o Rei dos Piratas"
- Monkey D. Luffy

Resumo

O desenvolvimento de arquiteturas de unidades centrais de processamento é um trabalho de grande dificuldade, desde sua concepção até a implementação de seu hardware. Se faz necessário o conhecimento de diversas estruturas de sistemas digitais e como eles interagem entre si. Além disso, transicionar entre a fase de concepção e a fase de implementação física acarreta em mudanças imprevistas devido a limitações de funcionamento, que em muitos casos, apenas podem ser identificadas com analises práticas de seu funcionamento.

Visando servir como um ponto de transição e um modulo de referência para o funcionamento de tais arquiteturas, um sistema de emulação foi desenvolvido para oferecer a possibilidade de testes práticos e debug, funcional e lógico, de forma antecipada. As arquiteturas desenvolvidas e estudadas ao longo desse projeto usam como base o conjunto de instruções RISC-V, escolhido por sua característica código aberto.

Palavras chave: Processadores, RISC-V, emulação, sistemas digitais.

Abstract

The development of central processing units is job with great hardships. From the conception phase to the hardware implementation. It requires knowledge of several digital systems structures and how they interact with each other. Besides that, the transition betwen conception phase and implementation phase surface several unpredictible changes due to functional limitations, that on most cases are only identifiable through practical functional analysis.

Aiming to be a transition point and reference module to those architecture , an emulation system was developed to offer, ahead of time, practical tests, functional debug and logic dubug. All the architecture developed and studied throughout this project uses RISCV-V as their instruction set. The choice was made due to the open source characteristic.

Keywords: Processors, RISC-V, emulation, digital systems.

Lista de Figuras

Lista de Abreviaturas e Siglas

ALU Unidade lógica aritmétrica

EX Executor

I/O Entrada e saída

ID Decodificador de Instruções

IF Apanhador de Instruções

INST Memória de instruções

MULT-DIV Multiplicador divisor

RAM Memória de uso geral

UE Unidade de Execução

BUS Barramento

RAM Read and Access Memory

ROM Read Only Memory

Sumário

1 Introdução

Em todo o mundo, milhares de sistemas de computadores são fabricados todos os dias, destinados as mais diversas aplicações como computadores pessoais, máquinas de trabalho, servidores, equipamentos médicos e acessórios portáteis. Esses equipamentos serão embarcados ou acoplados para funcionamento em conjunto de outras unidades, compondo um sistema maior. Minuto após minuto, executando uma função específica e passando completamente despercebido do usuário final.

Para a produção desses equipamentos "invisíveis" é despendido um grande esforço em diversas frentes de desenvolvimento. A concepção da arquitetura, a implementação digital do equipamento, a verificação de suas funcionalidades e a implementação física do mesmo, são as principais fases que um projeto de desenvolvimento de hardware está sujeito a passar. As dificuldades enfrentadas nessas categorias de projetos tendem a se concentrar na fase de transição entre concepção e verificação. Modelar um sistema computacional exige controle de diversos parâmetros e componentes que nem sempre apresentam um funcionamento claro até o momento que são postos em prática. Melhorias no fluxo de transição afetam significativamente as chances de se obter resultados satisfatórios ao final do projeto.

O processo de emulação tenta reproduzir, com níveis diferentes de fidelidade, o funcionamento de um equipamento. Quando trabalhamos com equipamentos embarcados e emulação de computadores buscamos mostrar todas as funcionalidades descritas no equipamento utilizando linguagens de alto nível, ou softwares. A complexidade do emulador vai desde a reprodução de I/O (entradas e saídas) até a descrição de todos os passos e operações internas do sistema. Uma arquitetura típica de computadores, Von Neumann, descreve o sistema composto por uma central de processamento com uma unidade aritmética e uma unidade de controle, um barramento pra transmissão de dados, memórias, uma unidade de armazenamento de dados e I/O. Para obtermos uma estrutura fiel ao objeto de estudo os componentes devem ser emulados de acordo com a arquitetura geral do sistema, porém não necessariamente reproduzindo cem por cento de seu funcionamento. O objetivo final da emulação é apresentar as saídas, referentes a um conjunto de entradas, da mesma forma que o hardware de referência faria. Abstrações da arquitetura podem ser feitas ao longo do desenvolvimento em detrimento de limitações do ambiente de desenvolvimento, melhorias de desempenho, pouca relevância para o resultado, e diversos outros motivos. Para isso, é necessário que o desenvolvedor tenha um conhecimento amplo, não só do local e das ferramentas usadas para emulação, mas também da máquina referência, seu ambiente de funcionamento e seu objetivo como hardware.

Este trabalho divide-se em três partes, na primeira, são estudados os elementos mais

importantes para arquiteturas de computadores e sistemas embarcados. Na segunda parte, é estudado o processo de emulação de uma arquitetura de processamento, que utiliza o conjunto de instruções RISC-V. Ambas as partes se complementam e fundamentam o conhecimento do leitor para a terceira parte, onde são apresentados testes de validação e esclarecimentos quanto ao funcionamento esperado para a arquitetura.

2 Embasamento Teórico

2.1 Sistemas embarcados

Definimos sistemas embarcados como a combinação de hardware e software, construídos e designados para uma função específica. Uma definição vaga e que abrange um número demasiado de soluções. Porém, esses sistemas são encontrados em todo tipo de objeto em nossa vida cotidiana, alguns chegando a ter dúzias desses sistemas em um único equipamento. Máquinas de lavar, tvs digitais, relógios inteligentes, videogames, caixas de som, carros, etc. Seu foco é sempre na realização de uma tarefa específica, e isso não muda, independente das modificações que venham a ser realizadas por software. Um sistema composto por um contador e um medidor de temperatura, pode realizar medições em diferentes momentos, ou contagens para diferentes temperaturas, mas não é possível alterá-lo ao ponto de se obter algo além de uma medição de temperatura ou contagem de tempo. Sua operação e escopo de atuação vai estar sempre ligada a essa funcionalidade principal, para a qual ele foi construído, desde a origem de sua arquitetura até o design final do produto.

2.2 Memória

Todo sistema embarcado contém unidades de memória para guardar e buscar informações digitais. Dois tipos principais são as memórias de programa e memórias de dados. A memória programável é usada para execução de programas, firmwares, intrínsecos ao funcionamento do hardware. A memória de dados é usada para armazenamento temporário de informações, auxiliando em todas as operações de fluxo de dados com resultados ou registros intermediários. O código gravado na memória programável pode ser dividido em duas partes: código de aplicação e código de inicialização, assim como apresentado na figura 2.1.As configurações de segurança, espaço de memória alocado para cada um dos códigos e acesso às variáveis dentro do espaço da memória, dependem dos requisitos do projeto e configurações feitas no compilador. A memória de dados é volátil e é usada para guardar variáveis durante a execução do programa, por sua natureza e finalidade, ela é delatada quando se para de suprir energia para o microcontrolador. Dentre as utilizações da memória de dados pode-se destacar: registradores de propósito geral, memória de entrada e saída e RAM interna.

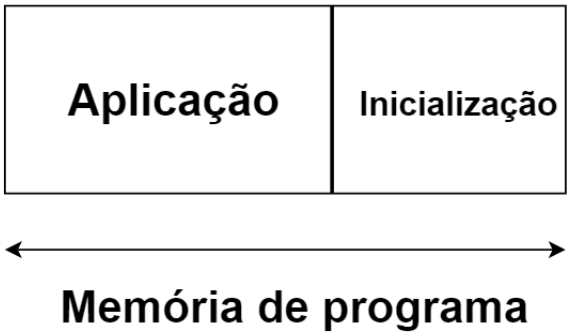

Figura 2.1: Memória de programa. Fonte: Autor.

2.3 Interrupção e Exceção

Interrupções permitem ao software responder de modo temporal a eventos externos e internos do hardware. Coordenando a comunicação com o mundo externo, interrupções podem melhorar em muito a eficiência e o uso de recursos do processador. Para sistemas de tempo real, interrupções exercem um papel mais crucial, uma vez que a resposta aos eventos deve ser tomada em um espaço de tempo limitado, a exemplo de entradas de vídeo, requerendo uma baixa latência e determinismo.

Exceções e interrupções são mecanismos que as CPUs utilizam para parar o fluxo de execução e chamar partes específicas do código quando um evento especial ocorre. A execução de software é interrompida e transferida para uma Rotina de Serviço de Interrupção (ISR). Uma vez terminada a ISR, a execução do software retorna para a próxima instrução que seria executada sem a interrupção, podemos observar esse fluxo na Figura 2.2.

Figura 2.2: Execução da rotina de serviço de interrupção. Fonte: Autor.

Quando o evento de interrupção é sinalizado um novo thread de execução e chamada. O código da ISR será executado dentro desse thread. Diferente um novo processo, que ocuparia seu próprio espaço de memória e recursos do sistema, um thread novo compartilha o mesmo espaço de memória e recursos, porém com o próprio Contador de Programa (PC), pilha e registradores.

2.4 Tempo

Unidades de processamento nem sempre foram tão rápidas como nos padrões atuais, CPUs que operam a poucos MHz precisam se preocupar constantemente com seu tempo de execução. Programas escritos em assembly, para CPUs antigas, levam muito em consideração a quantidade precisa do número de ciclos necessários para execução do código de programa. Devido a extensão média dos códigos e limites de funcionalidade é possível calcular a quantidade de ciclos que o código a ser executado necessitaria. Nos dias atuais, CPUs mais avançadas, com arquiteturas super escalares, grandes pipelines e reordenamento de execução, tornam a tarefa de estimar o número de ciclos quase impossível.

Para máquinas com CPUs lentas os programadores tentam usar todo o poder de processamento disponível no hardware. Programas são construídos levando em consideração como e quando cada equipamento deve ser acessado. Afim de evitar esperas desnecessárias programas usam técnicas como a execução em loop de cálculos até que seus valores sejam requisitados. Em alguns casos equipamentos dispõem de funcionalidades para buscar os valores necessários no momento correto (interrupções e exceções). Em situações que essa funcionalidade não está disponível, é exigido que o acesso ocorra no tempo exato que fora estimado, para sincronia das operações.

Existem diferentes níveis de fidelidade para emular o tempo de uma CPU. Na maioria dos casos não há necessidade de reproduzir algo a mais do que a precisão dos ciclos por instrução, atualizando o tempo para cada instrução lida. No entanto, algumas arquiteturas precisam da reprodução de ciclos intermediários, entre instruções, para a correta reprodução de seus resultados.

A sincronização com a máquina alvo, é um dos fatores principais para emulação de tempo. Computadores modernos já oferecem recursos para saber, com certa precisão, o tempo entre dois eventos. Usando o clock da máquina ou seus registradores internos é possível saber o tempo real que um programa gasta para ser executado. Para garantir a emulação do tempo decorrido devem ocorrer verificações de tempo ciclicamente em uma janela de tempo definida. Um videogame que acelere a execução do jogo, de modo que ele seja visto pelo usuário 10 ou 20 vezes mais rápido do que o esperado vai, inevitavelmente, causar desconforto e a impressão de que ocorreram erros em sua execução.

2.5 Unidade Central de Processamento

Designs modernos de computadores são baseados na arquitetura de John Von Neumann. Esta arquitetura pode ser simplificada em um barramento, conhecido normalmente como bus, uma CPU, memórias e estruturas de I/O, assim como apresentado na Figura 2.3.

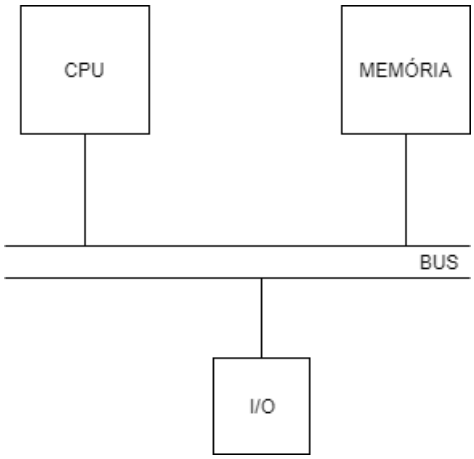

Figura 2.3: Arquitetura Von-Neuman. Fonte: Autor.

Os barramentos são conjuntos de trilhas elétricas que conectam todas as estruturas do sistema. Um caminho comum para transmissão de informação de um componente para outro. Esse caminho é usado para comunicação e pode ter diferentes funcionalidades a depender dos componentes ligados ao barramento. Compartilhamento de dados, endereçamento, alimentação de energia e estruturas de tempo, são os tipos de barramento principais encontrados nos projetos de computadores. Um barramento típico pode ser dividido em três partes: barramento de endereço, contendo a informação de qual equipamento deve ser acessado; barramento de dados, que estabelece uma rota para troca de informações entre a CPU e os periféricos; barramento de controle, que carrega sinais e informações de controle para auxiliar nas decisões de acesso ao longo do sistema.

A memória é a unidade principal de armazenamento. Existem diversos tipos de memórias, sendo as estruturas mais comuns: RAM (Random Access Memory) e ROM (Read-only memory), com suas respectivas variações. Essa unidade guarda o código de programa a ser executado, informações do sistema, resultados dos cálculos e valores que venham ser utilizadas posteriormente.

Os dispositivos de entrada e saída vão servir como ponte entre o mundo externo e o computador. Eles possuem vários propósitos e formas de atuação, desde dispositivos de armazenamento, como CDs e HDs até dispositivos de reprodução de vídeo, como monitores. Basicamente, qualquer dispositivo externo que possa ser controlado, transmitir dados ou se comunicar com a unidade central de processamento.

A CPU, é o centro de controle do computador. Ela interpreta instruções de comando e realiza a maior parte dos cálculos, se tornando parte vital para o funcionamento da arquitetura. Interagindo com as estruturas de memória e de I/O, ele pode utilizar para processamento dados já armazenados no sistema, adquirir dados de unidades externas ou transmitir dados para unidades externas.

A função principal da CPU é executar um conjunto de instruções armazenadas na memória. Uma CPU simples consiste em um conjunto de registradores, uma Unidade Lógica Aritmética (ALU) e uma Unidade de Controle (UC). A depender da arquitetura computacional, o número e designação de cada registrador varia de acordo com as necessidades. Podem ser encontrados registradores de uso geral, com livre acesso de leitura e escrita, e também podem ser encontrados registradores especiais, com atribuições específicas. Por exemplo, registradores que sempre carregam um valor constante ou o próprio Contador de Programa (PC), que guarda o valor da próxima instrução a ser executada.

A ALU, realiza as operações aritméticas, lógicas e de deslocamento requisitadas pelas instruções. A complexidade e número de operações disponíveis dependem da troca entre necessidade de processamento e recursos. Por exemplo, realizar uma operação em um ciclo único pode não ter benefícios suficientes para justificar o aumento do circuito, uma vez que essa mesma operação pudesse ser reproduzida interativamente, gastando mais ciclos, porém reaproveitando flip-flops e transistores já presentes.

Enquanto que a PC é responsável por coordenar as principais atividades da CPU. Seu escopo de atuação vai desde a aquisição das instruções até a execução das mesmas, sempre a depender da complexidade da arquitetura. Em sistemas mais complexos a unidade de controle tende a ser substituída por sub-blocos como "Pré-apanhador de instruções", "Decodificador" e "Executor". A necessidade da quebra em sub-blocos acontece devido ao número de instruções, sua complexidade e organização lógica. Em resumo, a UC atua no direcionamento do fluxo de dados, baseado nas informações trazidas pelas instruções, desde a fase de apanhamento até a fase de registro na memória interna.

2.6 Emulação de hardware

Um interpretador pode ser considerado o método mais direto para emulação de uma CPU. Utilizar o código fonte (bytes) em um ciclo de operações para apanhar instruções, decodificar e executar suas tarefas. Uma CPU simples pode ser modelada com um ciclo de leitura de dados de um endereço da memória, apontado por um registrador especial, PC. Os dados devem conter as informações das instruções que a CPU irá executar. O decodificador então utiliza as informações contidas nos dados para decidir o que realizar com eles. A ação interpretada será executada, atualizando o registrador PC e permitindo que um novo dado

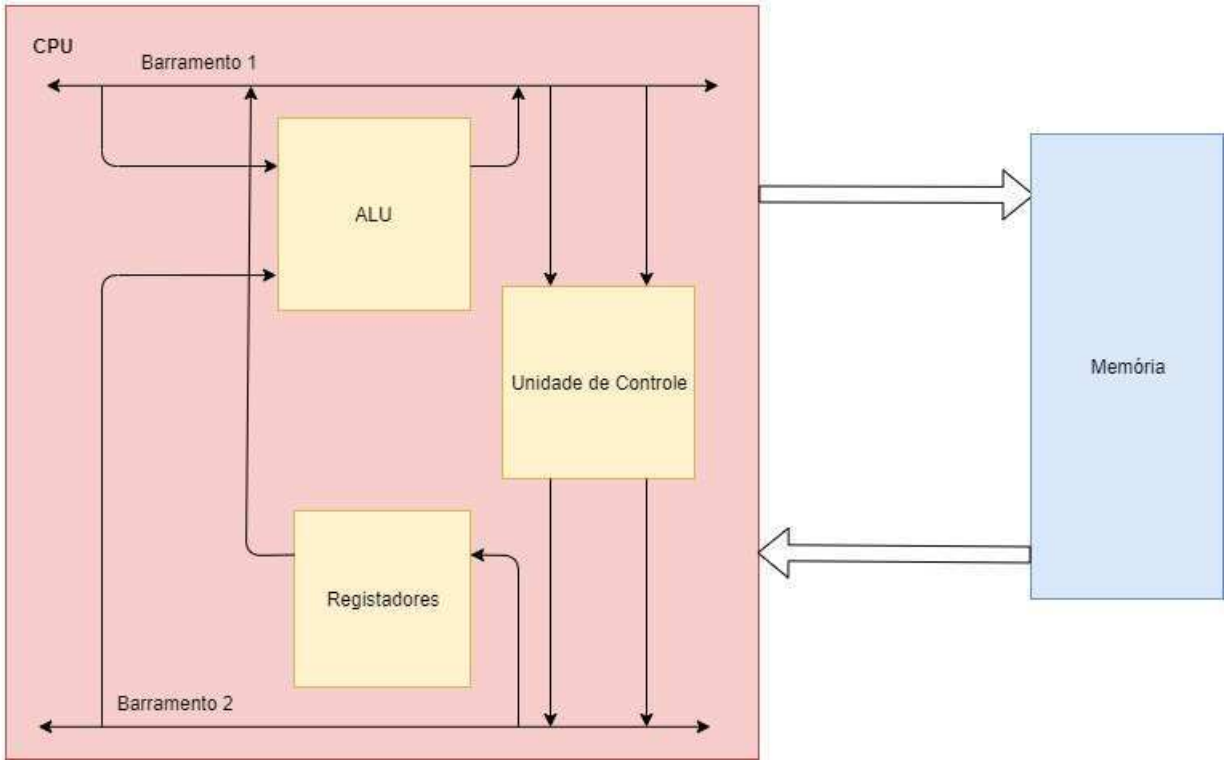

Figura 2.4: CPU com dois barramentos e datapath orientado. Fonte: Autor.

de sua nova posição de memória seja adquirido.

Modelando o funcionamento descrito, a emulação busca reproduzir a funcionalidade de cada um dos estágios. Em termos de software a operação inteira pode ser implementada utilizando uma lista de dados para operação, uma lógica de seleção (geralmente utilizando estruturas como switch/case em C, por exemplo) para cada um dos opcodes. Outras opções seriam, a tradução do código fonte, byte a byte, para código executável pela máquina alvo, de forma que as funcionalidades de uma arquitetura de computador sejam reproduzidas por outra arquitetura. A vantagem dessa opção seria o desempenho em não ter outras camadas de abstração para utilização do hardware disponível, dessa forma utilizando todo o potencial da máquina alvo.

2.7 Conjunto de Instruções (ISA)

Para que o computador consiga entender os comandos dados a ele uma linguagem apropriada deve ser usada. Essa linguagem é um conjunto de instruções, transmitidas ao computador, com um tipo de codificação e um formato pre definido. Uma instrução é um comando, que faz parte de um programa, dado ao computador para realizar uma tarefa especifica. Um conjunto de instruções de arquitetura são conhecidos mais tipicamente como ISA (Instruction

Set Arquitecture). As instruções presentes nas ISAs, são tipicamente classificadas por:

- Instrução de trânsferencia de dados
- Instruções Aritimeticas
- Instruções logicas
- Instruções de Jump e Branch
- Instruções de pilha ou I/O

Instruções de transferência de dados vão trabalhar entre a memória e os registradores internos, movendo dados da memória para os registradores, movendo dados entre os registradores, e pelo caminho oposto, dos registradores para a memória.

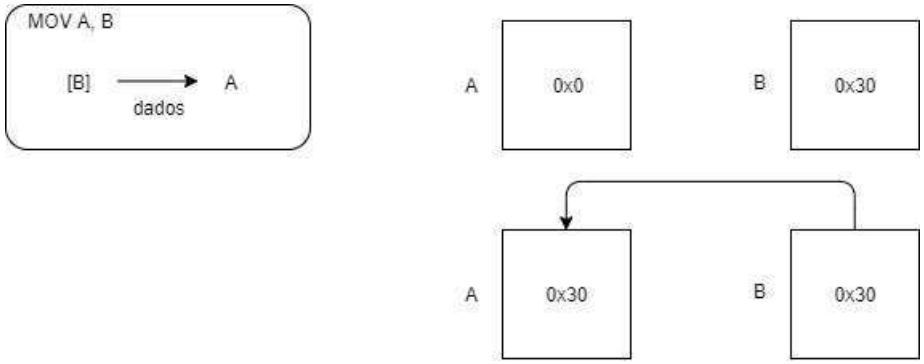

Figura 2.5: Exemplo de Instrução MOV (RISC-V), entre os registradores A e B. Fonte: Autor.

Instruções aritméticas vão definir operações aritméticas(soma, subtração, multiplicação, divisão) utilizando os registradores internos e memória como fonte de dados.

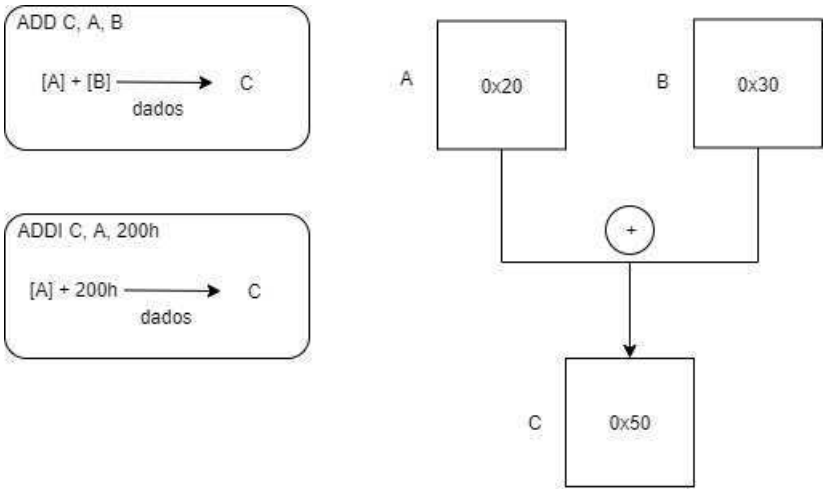

Figura 2.6: Instruções ADD(RISC-V),ADDI (RISC-V) e fluxo de dados para a instrução ADD. Fonte: Autor.

Instruções lógicas vão definir operações lógicas para a manipulação de valores booleanos, novamente operando com valores trazidos de registradores ou da memória, e alocando esses valore em registradores ou posições da memória.

Instruções de Jump e Branch vão realizar o deslocamento da sequência de execução de acordo com condicionais. Essas estruturas permitem o desenvolvimento de lógicas condicionais (if-else) e loopes de execução para a linguagem de máquina.

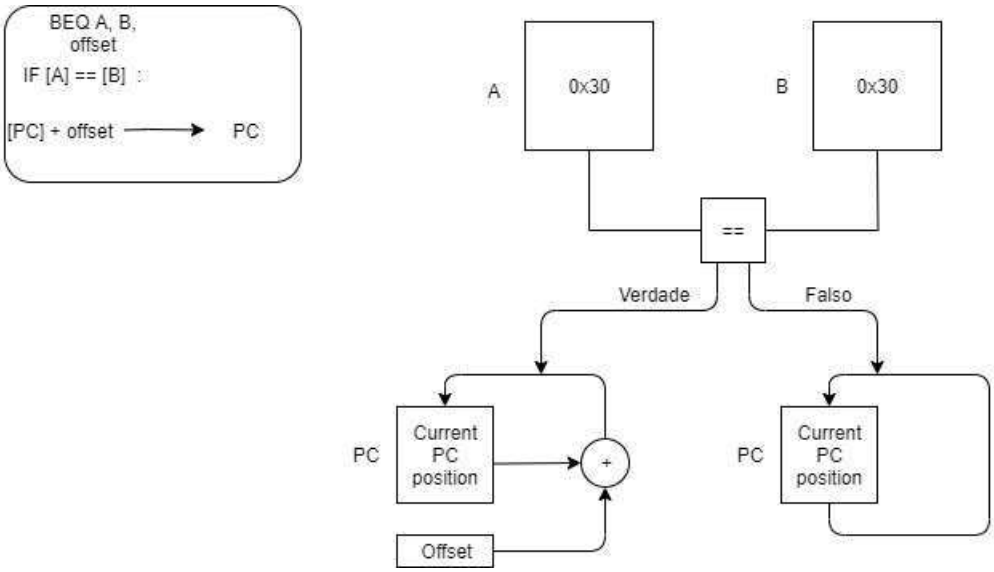

Figura 2.7: Instrução BEQ com deslocamento de PC por offset. Fonte: Autor.

As ISAs mais utilizadas no mercado atualmente são a x86 desenvolvida para processadores Intel/AMD e ARMv8 para processadores ARM. Ambas as ISAs são proprietárias. Ou seja,

todo o desenvolvimento para essas plataformas precisa ser licenciado e sujeito à restrições de modificação.

2.8 Conjunto de Instruções RISC-V

RISC-V é um conjunto de instruções de arquitetura, disponibilizado para livre utilização e com código fonte aberto. Agregado de uma gama de softwares e ferramentas, para o auxilio no desenvolvimento de sua plataforma. Sua característica mais chamativa é ser capaz de juntar a comunidade de pesquisa e desenvolvimento em volta de uma única referência de instruções para especificação de hardware.

RISC-V foi projetado para ser uma ISA modular e extensível. O ponto de partida, ou ISA base, é similar a um processador RISC porém sem instruções de branch e delay, e com suporte a codificação de instruções com tamanho de variável otimizado (técnica de compressão de instruções, reduzindo o tamanho de código de forma a ganhar mais espaço e menor consumo de energia). O seu conjunto base de instruções é limitado em pro de um sólido ponto de partida para as ferramentas periféricas como compiladores, assembler, linker, e sistemas operacionais, de onde podem ser customizados a depender da necessidade do projeto.

Devido ao grande numero de variações que podem ser geradas para a ISA base, uma nomenclatura foi pre-definida para suas extensões. As ISAs base são chamadas de RV32I(onde as palavras tem um comprimento de 32 bits) e RV64I(onde as palavras tem um comprimento de 64 bits). **Extensões padrão** são definidas por serem mais generalistas em sua funcionalidade e nunca entrar em conflito com outra **extensão padrão**.

2.8.1 ISA RV32I

A RV32i foi modelada para oferecer instruções que atendam os requisitos mínimos para o compilador formar um execução e suportar sistemas operacionais modernos. Em seu design também contempla uma redução do hardware necessário em uma implementação mínima. RV32I oferece um total de 47 instruções únicas, entre elas operações com inteiros, imediatos, controle de transferências, load, store, modelos de mória, controle de status para registradores, chamadas de ambiente e pontos de quebra ("breakpoints"). Com uma abrangência mais generalista de suas instruções a RV32I também pode emular todas as demais extensões de ISA, com exceção da extensão A, sendo requerido suporte adicional devido a atomicidade.

Para operações de proposito geral 32 registradores estão disponíveis. O numero de registradores tem um impacto direto em todos os aspectos da máquina(hardware) desenvolvida, de gasto de energia até o tamanho de código compilado. Definidos como **xreg**, todos são de livre acesso por todas as instruções e variam de tamanho a depender do valor de XLEN, que

atribui 32 bits para a RV32I e 64 bits para RV64I. O registrador **x0** é obrigatoriamente uma constante 0 em todos os momentos, independente da operação realizada com ele.

Existem 4 tipos básicos de formato de instruções: **R**, **I** , **S**, **U**. Além desses, uma variação dos formatos S e U são definidos: **B** e **J**. Todas os tipos de instruções apresentadas são fixos em 32-bits e devem estar alinhados em endereços de 4 em 4 bytes na memória. Qualquer acesso de memória desalinhado dos 4 bytes de referência provoca um comando de exceção pela unidade de controle.

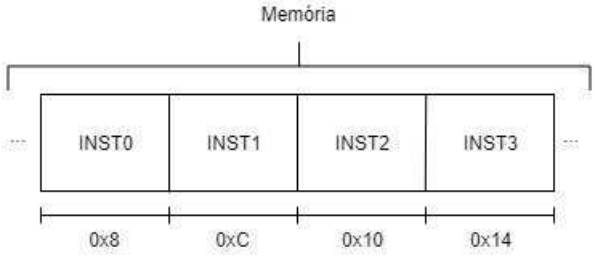

Figura 2.8: Organização das instruções na memória. Fonte: Autor.

Os cinco tipos de instruções podem ser vistos na figura 2.9. Os elementos que compõem as instruções são: **RS2**,**RS1**, **Rd**, **FUNCT7**, **FUNCT3**, **OPCODE** e **IMM**. Cada um deles é um pedaço de informação necessária para realizar a operação da instrução. RS2 e RS1 fazem referência a registros internos de onde a informação vai ser buscada. RD faz referência a um registro interno onde a informação deve ser guardada. FUNTC7 e FUNCT3 permitem a multiplexação de operações a partir de um único opcode, de forma que as instruções sejam agrupadas de acordo com suas funcionalidades. OPCODE define o grupo de instruções a ser executado. IMM é o valor de imediato a ser usado para as operações.

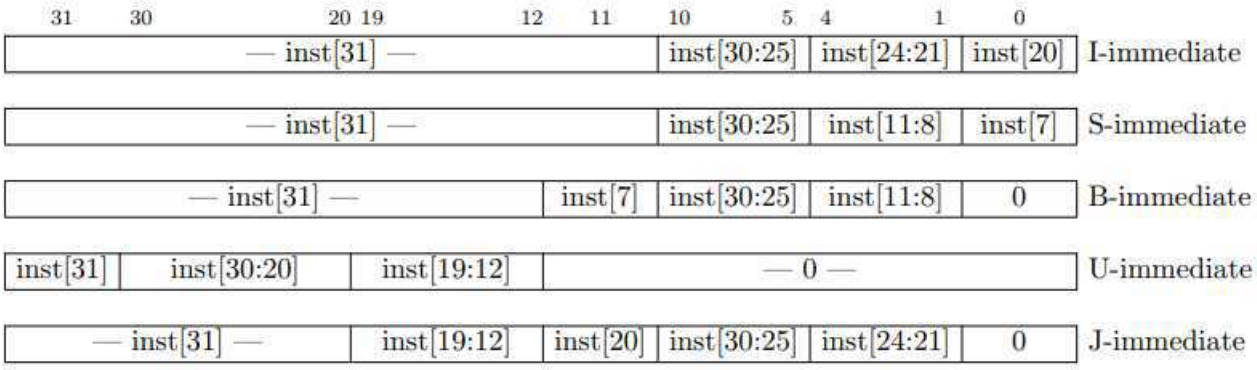

Figura 2.9: Formato das instruções I,S,B,U,J (Volume I: RISC-V Unprivileged ISA V20190608-Base-Ratified 2019). Fonte: Autor.

3 Emulação de uma CPU RISC-V

A arquitetura proposta para emulação de uma unidade central de processamento pode ser vista na figura 3.1. Seu funcionamento gira em torno de um pipeline de 3 estágios: Apanhar instruções (IF), decodificar instruções (ID) e executar instruções (EX). Cada um dos estágios não só tem seu funcionamento descrito como também apresenta conexões e estruturas de blocos internos.

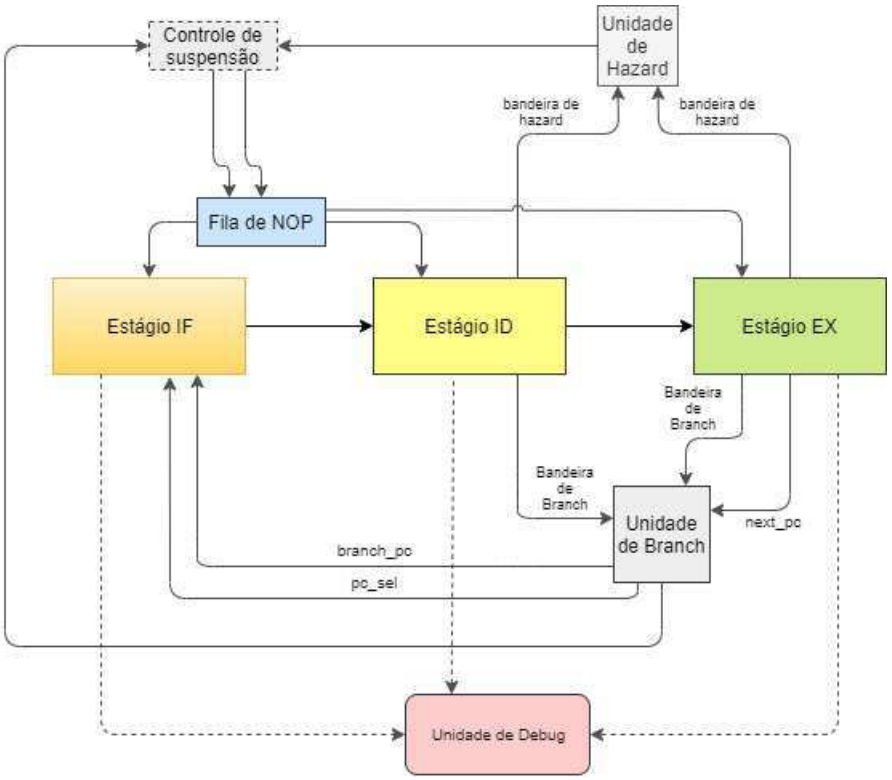

Figura 3.1: Arquitetura de operação. Fonte: Autor.

3.1 Rotina de pipeline

O pipeline de operação é caracterizado por um fluxo de dados que executa a seguinte sequência: Estágio IF, estágio ID e estágio EX. Cada estágio é composto por um mais blocos funcionais, contendo as estruturas de dados e os métodos de execução necessários para seu funcionamento. Além dos blocos principais existem 2 blocos de controle e um de debug que coordenam o funcionamento do pipeline: Unidade de Hazard, unidade de Branch e unidade de debug.

A unidade de hazard recebe informações dos estágios ID e EX para julgar se é necessário a suspensão de atividades do pipeline. A suspensão pode ser feita individual para cada estágio,

de maneira que é possível solucionar problemas de dependência entre instruções ou conflitos de informação durante a execução. Quando necessária a suspensão de atividades, instruções NOP são enviadas para cada estágio alvo, garantindo que o conteúdo executado por ele não influenciará outras operações ou a sequencia de execução das instruções.

A unidade de Branch atua de forma similar a unidade de hazard. As informações do estágio ID e EX são utilizadas para avaliar a validade do branch e se é necessário suspender as atividades ao identifica-lo. Além disso, ela encaminha o valor de BRANCH PC e PC SEL quando uma instrução solicitar mudanças na ordem de execução. O valor de BRANCH PC a ser encaminhado é extraído do resultado da unidade de execução e o valor de PC SEL é definido com base na validade da instrução de branch.

A unidade de debug coleta informações do estado atual de cada estágio e quando instanciada gera um log de execução de todo o sistema.

As operações do pipeline ocorrem em ciclos de execução, dessa forma criando uma referência temporal para suas atividades. Cada estágio trabalha em torno de dois estados de operação: Atualizar componentes e executar componentes. Em um primeiro momento o estado de atualizar componentes é chamado em cada um dos estágios, onde as estruturas de latch de cada um dos blocos são atualizadas de acordo com suas conexões. Em seguida, o estado executar componentes é chamado em cada um dos estágios, onde as operações definidas para eles são de fato executadas e os resultados atualizados nos registradores temporais.

3.2 Apanhador de Instruções

O estágio IF atua como interface de comunicação, entre a memória de instruções e a CPU, e como ordenador das operações a serem executadas. Nesse estágio os métodos de aquisição de instruções, controle do registrador interno PC e reparticionamento de instruções são executados. A figura 3.2, mostra a arquitetura interna dos blocos presentes durante esse estágio.

O registrador interno controlador de programas (**PC**) é a referência de busca e organização para as instruções que serão executadas. Com base no valor atual de **PC** um endereço é enviado para a memória de maneira que ela retorne a instrução a qual ele fez referência. O valor de **PC** também é enviado para uma lógica externa que prepara o próximo valor de **PC**. Instruções como JAL, JALR e BEQ tem o objetivo de alterar o valor de **PC**, avançando a ordem de execução das instruções ou retornando a pontos anteriores. Além disso, em situações de suspensão das operações devido a hazards, branchs ou dependência de instruções necessitam um controle da ordem de operação para que o sistema possa voltar a execução sem perder informações. O bloco **PC Direcionador** utiliza o último valor de PC como referência, preparando a próxima instrução a ser solicitada. O registrador **NEXT PC** guarda o valor

selecionado pelo mux chaveado por **PC SEL** e o encaminha para o registro de **PC** dentro do bloco apanhador de instruções. As opções disponíveis para atualizar o valor de **NEXT PC** são: Branch PC, PC e PC + 4. A opção de Branch PC é utilizada em função da execução de uma operação valida de BRANCH pelo estágio **EX**.

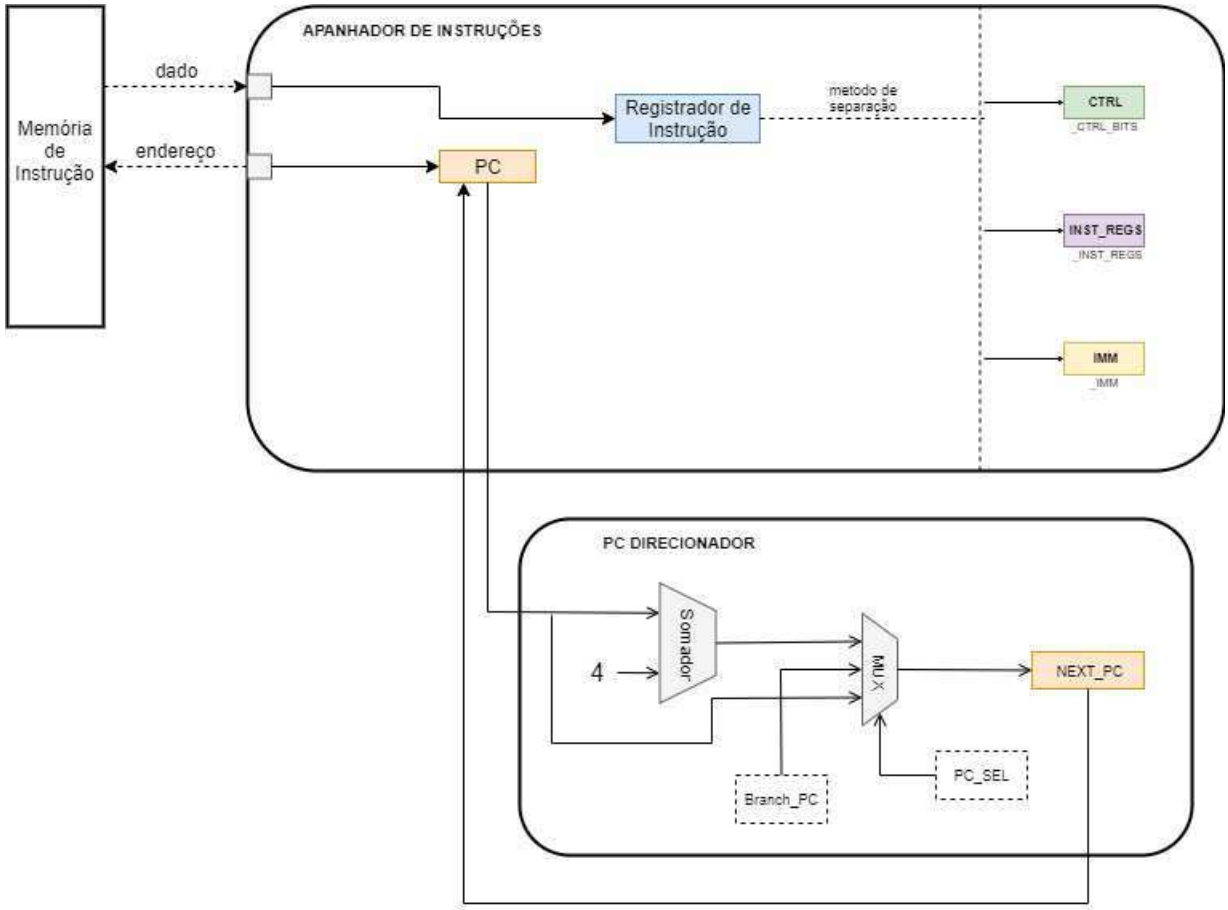

Figura 3.2: Arquitetura de operação do estágio IF, contemplando os blocos apanhador de instruções e PC direcionador com suas respectivas conexões. Fonte: Autor.

Adiante no trabalho será discutido a lógica por trás dessa solicitação e diferentes situações de BRANCH que podem ocorrer. A opção PC é utilizada, em situações de paralisação da execução do estágio IF, para que as informações do estado atual sejam mantidas independente do ciclo de retorno da execução do pipeline. A opção PC + 4 avança o PC, referenciando a próxima instrução na ordem definida pela memória de instrução. Caso nenhuma solicitação externa seja feita, essa opção sempre é executada, avançando PC desde sua referencia inicial até a ultima instrução disponível.

Uma vez que a memória de instruções retorne a instrução solicitada, ela é armazenada no registrador de instrução. Esse registrador mantém sempre a instrução que deve ser executada durante esse ciclo e encaminhada para o pipeline. O método de separação é aplicado ao

conteúdo desse registrador e resulta em 3 estruturas de dados: CTRL BITS, INST REGS e IMM. As três estruturas contém todas as informações codificadas na instrução e cada uma faz referência a um vetor de bits que foi previamente definido com base na documentação "The RISC-V Instruction Set Manual V 2.2".

Cada uma das estruturas é expansível e mutável de modo a se adaptar a diferentes padrões de codificação. Para a abordagem RISCV RV32I elas podem seguir o seguinte padrão de separação: INST REGS define os endereços para RS1, RS2 e RD. CTRL BITS define os valores de funct3, funct7 e OPCODE. IMM define os valores de imediato para cada tipo de instrução disponível IMM I, IMM S, IMM U, IMM B, IMM J.

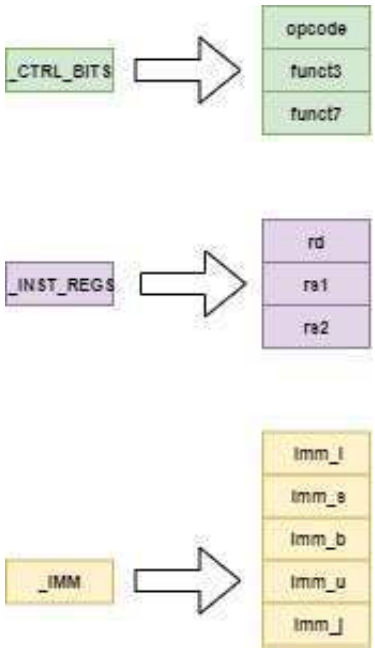

Figura 3.3: Estruturas geradas com o metodo de separação. Fonte: Autor.

3.3 Decodificador de Instruções

O estágio ID realiza a identificação das instruções que estão na fila e busca as informações necessárias para sua execução. Além disso, uma vez identificada a instrução é testada a validade de sua execução. Caso exista dependência de valores que estão para ser gerados ou previsão de um branch a ser executado, devem ser informado ao sistema, por meio de bandeiras, para que ocorra uma suspensão das atividades ou resgate das informações por outros meios.

A informação em CTRL.OPCODE é usada para identificar o grupo a qual a instrução pertence com base na **tabela de opcodes**, definida pelas chaves de execução apresentadas na lista 3.3. Ao longo do código essas chaves definirão as configurações de execução e

os resultados esperados. O grupo da instrução definirá qual imediato e quais registradores devem ser encaminhados para o estágio seguinte. Os grupos de informações não relevantes são marcados como privados e qualquer tentativa de acesso é registrada pelo controle de debug.

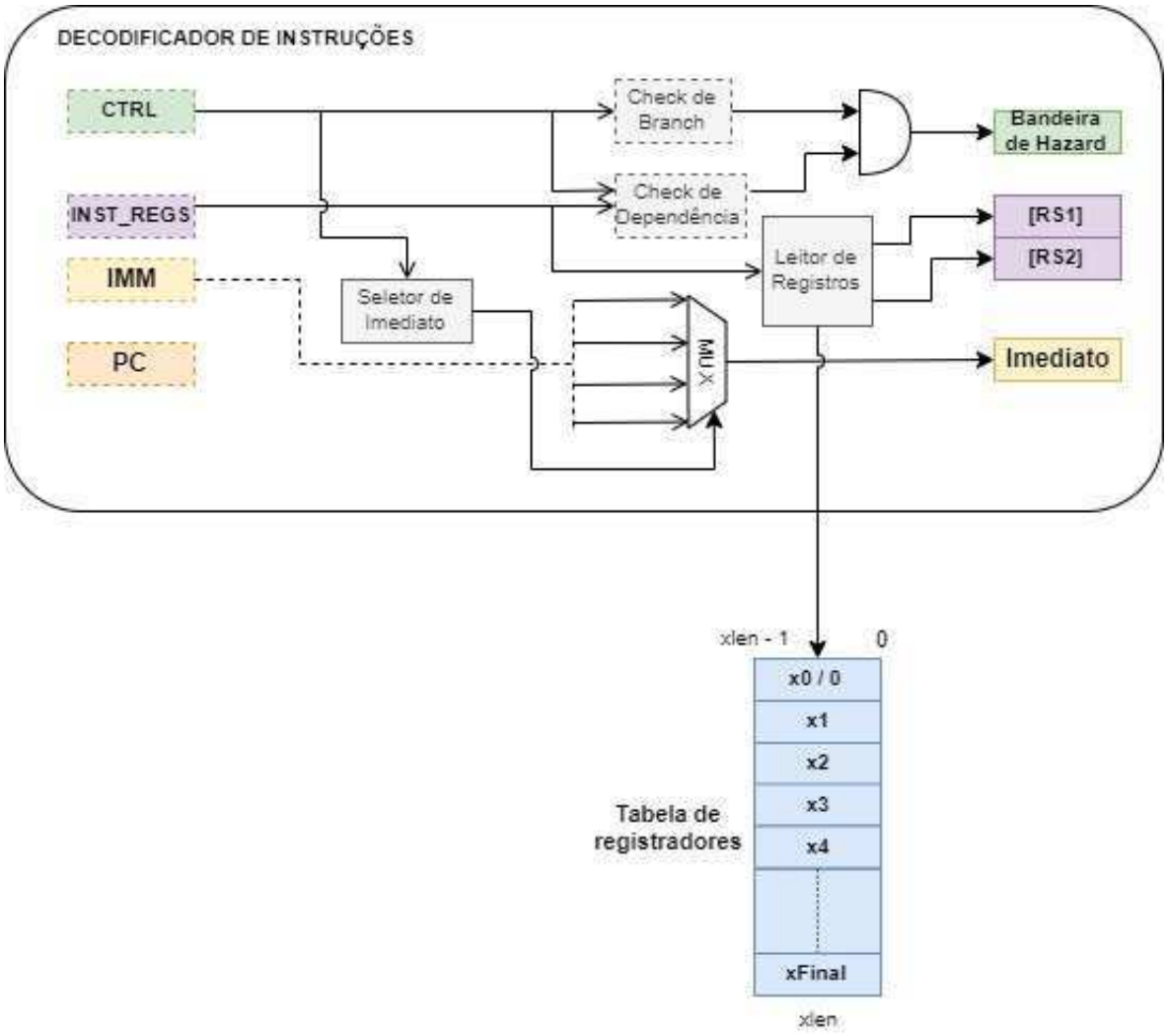

Figura 3.4: Arquitetura de operação do estágio ID, contemplando os blocos decodificador de instruções e a tabela de registradores do sistema. Fonte: Autor.

```
#define SET_RV32I
#define SET_RV64I
#define M_EXTENSION
#define AT_EXTENSION

#define SET_DEBUG
#define SET_OP_TEST
#define SET_UI_TEST
```

```
#define SET_DEP_TEST
#define SET_BASIC_TEST
#define SET_LOADSTORE_TEST
```

Lista 3.3 Macros para instruções RV32I. Fonte: Autor.

Para obter um controle temporal na escrita dos registradores, foi implementada uma estrutura de registradores temporais. Definida como TEMP REG, a estrutura possui duas variáveis para dados, entrada e saída, e um método de atualização. Todo valor a ser registrado é escrito na variável de entrada e para recuperar o valor atual deve-se ler a variável de saída. Ao chamar o método de atualização de registradores o valor atual da variável de entrada é repassado para a variável de saída, convergindo para um valor único no registrador. Um método que verifica a equidade entre as duas variaveis disponíveis é chamado após a atualização, como redundância e garantia do funcionamento. Caso exista violação uma interrupção de operação é chamada.

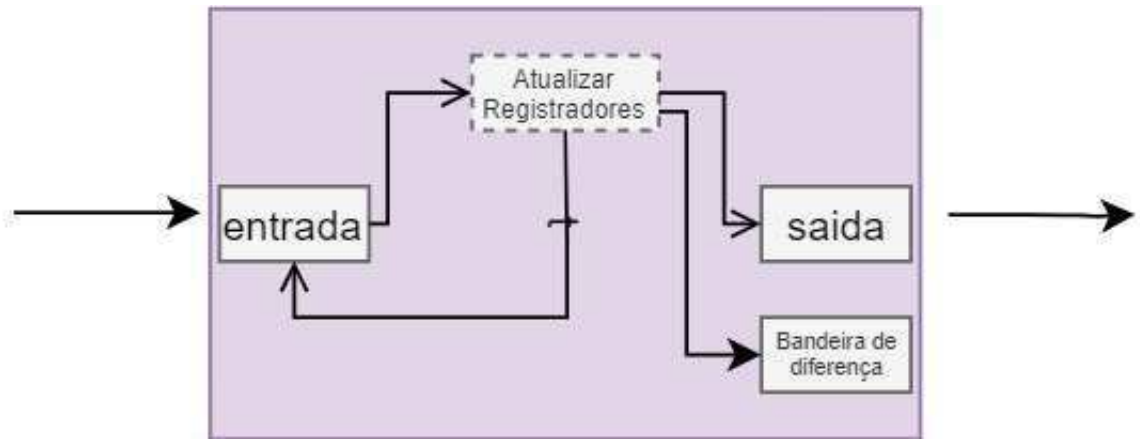

Figura 3.5: Estrutura de registradores temporais. Fonte: Autor.

Com base no grupo da instrução atual, e nos endereços disponíveis em INST_REGS.RS1, INST_REGS.RS2 e INST_REGS.RD, o método **leitor de registros** acessa a **tabela de registradores** e extrai a informação disponível em sua variável de saída. Dependendo dos dados solicitados pela instrução, [RS1] e [RS2] são escritos, com os valores encontrados.

Após a leitura dos registradores, é verificada a dependência entre as instruções presentes no pipeline. Caso a instrução atual no estágio ID dependa de um registrador que está sendo

processado pelo estágio EX uma bandeira de hazard é levantada. A ativação da bandeira não necessariamente confirma a existência de um hazard, ela é encaminhada para o modulo **unidade de hazard**, que deve realizar um handshake com as informações vindas do estágio EX para confirmar a validade do hazard e realizar as medidas necessárias. A verificação de branch utiliza a mesma lógica, no entanto, apenas identifica o grupo de instrução como sujeito a alterar o valor de PC e já realiza a chamada da bandeira.

3.4 Unidade de Execução

O estágio EX realiza as operações da instrução identificada e escreve, quando necessário, na tabela de registradores temporais. Novamente, testes de hazard são realizados para confirmar a previsão feita no estágio anterior **ID**. Os resultados confirmam a execução da instrução ou falha de condicionais, comprovando, ou não, a necessidade de suspensão de atividades no pipeline. A figura 3.6 mostra a arquitetura do estágio mencionado.

Todas as informações necessárias para execução da instrução são agrupadas em uma estrutura de contexto, de modo a flexibilizar suas operações dentro do modulo, e minimizar os esforços de uma mudança de arquitetura.

A estrutura de contexto vai interagir com todos os blocos do estágio se comportando como uma interface de comunicação para o **executor**. Para coordenar atualizações do contexto, em sua inicialização e durante sua execução, o bloco **atualizador de contexto** é utilizado. Ele detém privilégios de escrita em todas as variáveis da estrutura de contexto, e utiliza as entradas do bloco executor para mudar os valores do contexto quando atualizações forem solicitadas. Em contraste, a unidade de execução possui privilégios de escrita limitadas apenas a variável de RESULTADO, e o escritor apenas privilégios de leitura. O **escritor de registros** interage com a tabela de registradores do sistema utilizando os valores presentes no contexto para realizar modificações requisitadas, pelas instruções, nos registradores. A operação desse bloco é independente de outras estruturas, de modo que ele possa ser coordenado por agentes externos de acordo com as necessidades de cada instrução.

A **unidade execução** (UE) é o método principal para as operações do executor. Essa unidade interage puramente com o contexto, utilizando de suas informações atuais para atualizar a variável de **RESULTADO**. Por sua vez, a variável de resultado é uma estrutura composta de todos os possíveis retornos da execução de uma instrução. Devido a sua arquitetura é possível utilizar diferentes implementações da unidade execução de maneira que o sistema se adapte ao nível de abstração buscado com a emulação.

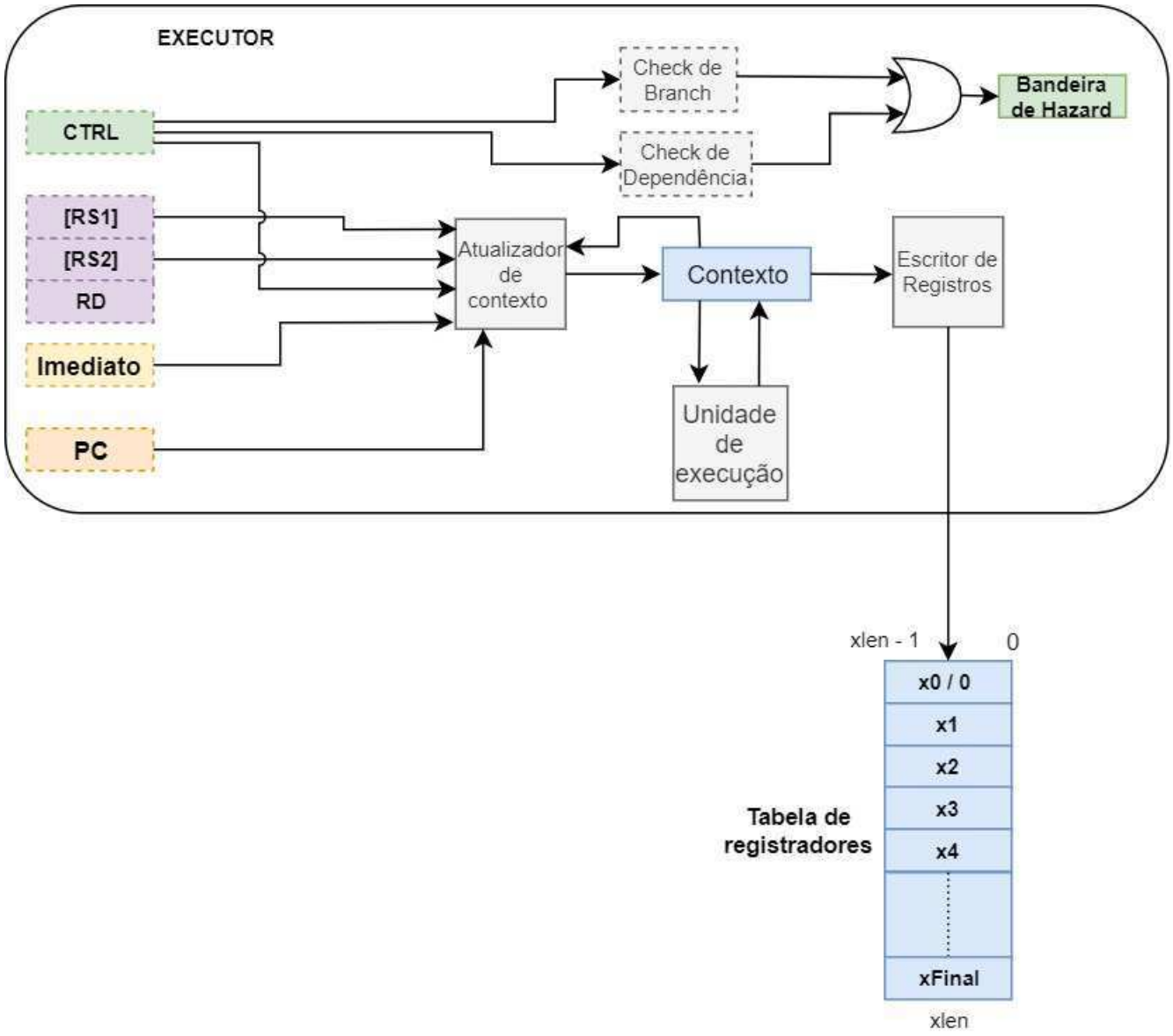

Figura 3.6: Arquitetura de operação do estágio EX, contendo os blocos executor e a tabela de registradores do sistema. Fonte: Autor.

A primeira implementação consiste em abstrair a utilização de uma unidade lógica aritmética (ULA), tomando proveito dos recursos oferecidos por uma linguagem de programação de alto nível, e utilizar métodos individuais para cada instrução existente. Os valores presentes no contexto são organizados de acordo com o tipo de instrução (R,I,S,U) e entregue a um método, escolhido com base na classificação de suas estruturas de controle(OPCODE, FUNCT3, FUNCT7).

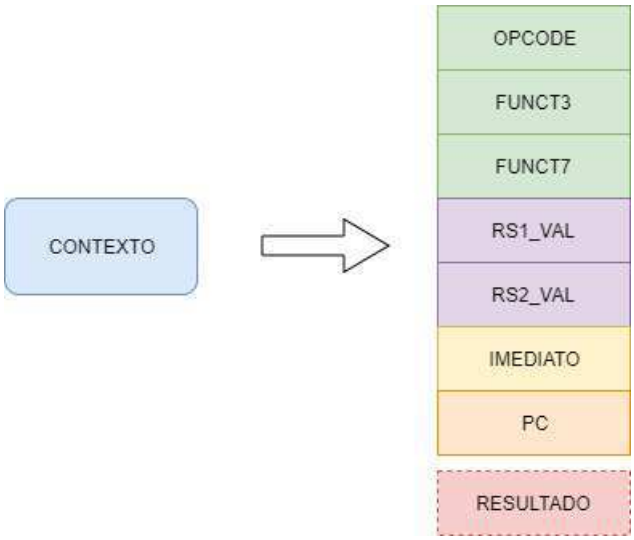

Figura 3.7: Estrutura de contexto.

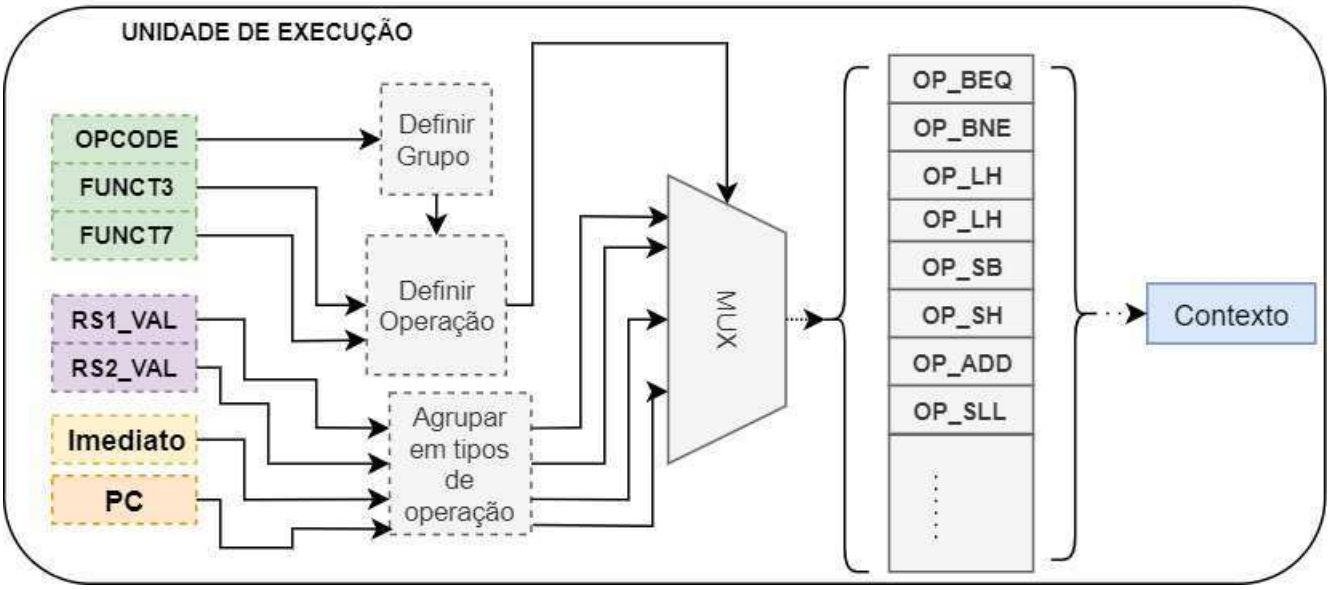

Figura 3.8: Arquitetura da unidade de execução baseada em um mux seletor para descrição de operações. Fonte: Autor.

Cada método tem uma implementação própria de modo a ser objetivo com o resultado final, suas estruturas tendem a ser independentes e adaptadas ao seu objetivo. O código apresentado na lista 3.4, demonstra a implementação do método da instrução SLTIU, para operações entre registradores e imediatos, classificado como tipo I.

```
//Set if less then immediate unsigned
void OP_SLTIU(OP_CONTEXT* ct){
        if(ct->imm == 1){
                if(ct->rs1_val <= ct->imm){
```

```
                        if(ct->rd!=0)
                                ct->result = 1;
                }
                else
                if(ct->rd!=0)
                        ct->result = 0;
        }else
        {
        if( ct->rs1_val < (uint32_t) (int32_t) ct->imm){
        if(ct->rd!=0)
         ct->result = 1;
        }
        else
        if(ct->rd!=0)
         ct->result = 0;
        }

}
```

Lista 3.4: Metodo para execução da instruções classificadas como SLTIU. Fonte: Autor.

A segunda implementação busca uma maneira mais fiel para obter os resultados do estágio EX. Dessa vez, a unidade de execução apresenta dois módulos internos para operações: Unidade lógica aritmética (ALU) e multiplicador divisor(MULT-DIV). O método de controle de operação utiliza as variáveis de controle presentes no contexto, para multiplexação dos valores a serem encaminhados para ALU. Por sua vez, a ALU opera sempre com dois valores de entrada e o resultado da operação é escrito na variável RESULTADO dentro do contexto. Os valores são encaminhados para unidade de MULT-DIV que responde aos estímulos de controle do método de operação. Dessa forma, é possível obter não somente o resultado final mas também a reprodução dos passos necessários para construí-lo.

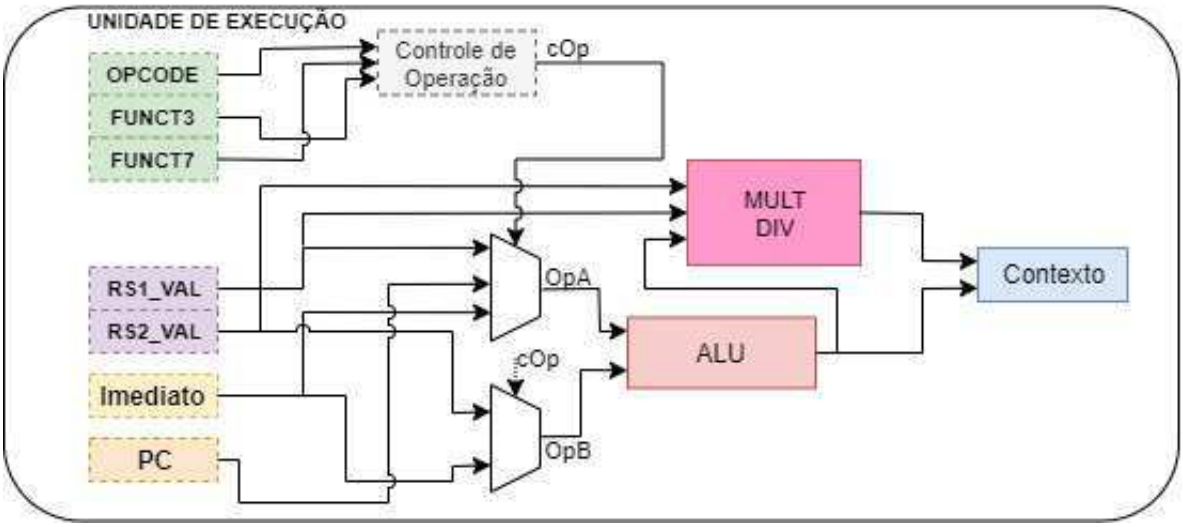

Figura 3.9: Arquitetura da unidade de execução baseada na interação entre os blocos ALU e MULT-DIV. Fonte: Autor.

3.5 Memórias

O sistema possui duas memórias baseadas na mesma estrutura: A memória de instruções(INST) e a memória de uso geral (RAM). A estrutura de memória define um vetor de endereços de tamanho limitado, alocado durante a fase de inicialização, e um conjunto de métodos para inicialização, verificação de resultados e operações de escrita e leitura. Cada operação de escrita ou leitura deve corresponder ao tipo de acesso que está sendo feito à memória, acessos de 8-bits, 16-bits e 32-bits.

A memória de instruções (INST) é preenchida no inicio da simulação utilizando um arquivo externo como referência. Ela contém todas as operações que devem ser executadas pela unidade de processamento e não possui restrições de acesso.

A memória de uso geral (RAM) é inicializada no inicio da simulação e verificado se seu conteúdo se encontra com o valor de reset correto. Qualquer falha no inicialização da memória com os valores corretos deve repercutir de maneira grosseira nos resultados, os métodos internos da estrutura de memória são utilizados para verificação de seu conteúdo.

A seguir são apresentados alguns métodos implementados, para a estrutura de memória na lista 3.5.

```
bool write_mem_u8(uint8_t val, uint32_t addr){

        addr -= ram_start;
        if(addr > RAM_SIZE){
                std::cout << "illegal write"<<std::endl;
```

```
                EXCEPTION();
                return 1;
        } else {
                uint32_t* p = RAM + addr;
                p[0] = val & 0xff;
        }

        return 0;
}

bool read_mem_u32(uint32_t *p_val, uint32_t addr){

        addr -= ram_start;
        if(addr > RAM_SIZE){
                *p_val = 0;
                std::cout << "illegal read"<<std::endl;
                return 1;
        } else {
                uint32_t* p =  RAM + addr;
                *p_val = p[0];
        }

        return 0;
}

void check_memory()
{
        uint32_t tmp;

        std::cout <<"Checking memory:" <<std::endl;
        for(int i = 0; i < RAM_SIZE; i++){
                read_mem_u32(&tmp, i);
                std::cout << "mem[" << i << "]=" <<tmp << std::endl;
                }
}


void FetchData(char *file_name, uint32_t *mem_add, int offset)
{
        //Open Rom file and fetch its data
        FILE *f = fopen(file_name,"rb");
        if(f == NULL) std::cout << "error";

        fseek(f, 0L, SEEK_END);
```

```
        int fsize = ftell(f);
        fseek(f, 0L, SEEK_SET);

        //Set a pointer buffer to our main memory address
        uint32_t *buffer = &mem_add[offset];

        //Variable to fetch strings from file
        unsigned char *foo;
        foo = malloc(0x10000);

        //Use This in case of binary Data
                //fread(buffer, 1 , fsize , f);


        //Create a String stream and throw hex strings on it
        std::stringstream ss;
        while(fgets(foo,fsize, f) ) {
        //std::cout <<"foo:"<< foo << std::endl;
        ss << std::hex << foo;
        };

        //Fetch the Hex values from the Stream to our main memory
        for(int i=0; i < 0x800 ; i++ ) ss >> buffer[i];

        fclose(f);

}
```

Lista 3.5: Exemplos de métodos implementados para as estruturas de memória, respectivamente: Escrita de 8-bits, leitura de 32-bits, verificação de acessibilidade de conteúdo e apanhador de dados para construção da memória. Fonte: Autor.

4 Resultados

Um dos modulos desenvolvidos ao longo do projeto foi o DEBUG. Esse modulo guarda informações de todas as operações executadas e as organiza de modo a gerar um log de atividades. Com essa ferramenta, é possível observar o caminho percorrido pelas instruções ao longo do pipeline e como sua execução influência componentes internos do sistema.

Para melhor avaliar o funcionamento de cada uma das partes projetadas foi desenvolvido um conjunto de testes para o sistema. Um ou mais testes podem ser chamados em cadeia e configurados antes do inicio de cada simulação.

O primeiro conjunto de testes desenvolvido foca na correta execução de cada operação prevista pela ISA RV32I, dividindo-se nos seguintes testes: Operandos U-I, BRANCH, LOADSTORE, OP. O teste **operandos U-I** trabalha com todas as instruções codificadas nos formatos U e I, focando em operações com imediatos e na distinção entre valores signed e unsigned que deve ser feita durante as operações e na apresentação dos resultados.

Todos os testes são construídos dentro do próprio sistema, em sua faze de inicialização, com o auxilio do construtor de instruções RISC-V desenvolvido no inicio do projeto.

O segundo conjunto de testes foca na execução geral do sistema, onde é avaliado se ele consegue exercer um fluxo continuo de operação sem que nenhuma combinação de instruções quebre a operação ou chame uma bandeira de erro interna. Os testes contruídos para esse grupo são: Basic test e DEP test. O basic test passa por instruções de cada um dos tipos (R,S,I,U,J) e realiza operações diversas buscando interagir com todos os registradores e porções variadas da memória. O DEP test faz a chamada de instruções que sequenciadas criam uma dependência no acesso a registradores ou posição de memória.

Todos os testes mencionados tiveram exito em seu funcionamento, retornando os resultados esperados. Em múltiplos casos foi necessário realizar o debug das estruturas e dos casos de teste, afim de corrigir erros tanto na implementação do código quanto na lógica de funcionamento. Para esses casos foi utilizado como referência, a ferramenta de simulação código aberto Rv8(R), simulador RISC-V para arquiteturas x86-64, e o RISC-V Compliance, um grupo de testes validados para arquiteturas RISC-V.

5 Conclusão

Com esse trabalho foi desenvolvido um sistema de emulação para unidades de processamento, utilizando como base a arquitetura apresentada na seção 3. Com as técnicas e estruturas implementadas foi possível obter um sistema de emulação flexível, e de fácil adaptação para diferentes arquiteturas. Ao longo do desenvolvimento foi necessário um estudo geral sobre arquiteturas de sistemas digitais, seu comportamento esperado e suas melhores práticas de desenvolvimento. Reproduzir as informações disponíveis nessa área de conhecimento, em um sistema prático e funcional, se provou uma tarefa árdua e com diversos níveis de dificuldade. Sendo assim, cabe ao desenvolvedor ter uma noção aguçada de quando optar por caminhos mais difíceis e quando não são necessários.

Interpretar o funcionamento de um hardware, ou fazer modificações em uma arquitetura já existente, exige o controle não apenas local das modificações, mas uma análise global do sistema para entender se existem repercussões e onde elas irão ocorrer. Desenvolver esse projeto com a visão de um sistema adaptável a múltiplas arquiteturas proporcionou um grande aprendizado no aspecto de visão de projeto e base estrutural para o desenvolvimento de trabalhos nessa área.

Para trabalhos futuros é esperado o desenvolvimento de uma interface gráfica, que possa apresentar os logs de execução em tempo real e seja integrado com um interpretador de código assembly para criação execução de programas de forma mais prática. Além disso, a criação de mais arcabouços de estruturas e um método de generalização para o pipeline de operação, de modo que se possa adicionar n-estágios de operação sem que grandes mudanças sejam necessárias.

Por fim, trabalhar com RISC-V, mais uma vez, foi uma atividade extremamente satisfatória. O suporte a essa ISA é amplo e conta com um acervo, considerável, de trabalhos já desenvolvidos. Sua filosofia código-aberto, certamente, influencia a comunidade que gira a seu redor e a cada projeto novo demonstra o potencial que temos, quando as ferramentas certas são disponibilizadas a todos, independente de origem, cor e conhecimento.

6 Referências

[1] Huang, Andrew "bunnie", 2017 *The Hardware Hacker*.

[2] Arora, Mohit, 2016 *Introduction To SOC System Architecture*.

[3] Stroustrup, Bjarne, 2014 *A tour of C++*.

[4] *Intel 8080 Assembly Language Programming Manual*

[5] *The RISC-V Instruction Set Manual*. Disponível em: https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

[6] *RISC-V External Debug Support Version 0.13.2*. Disponível em: https://content.riscv.org/wp-content/uploads/2019/03/riscv-debug-release.pdf

[7] *Ibex user manual*. Disponivel em : https://ibex-core.readthedocs.io/en/latest/index.html