UNIVERSIDADE FEDERAL DO RIO GRANDE

Gabriel da Costa Barbosa

Simulador de Caminho de Dados para o Ensino Integrado de Arquitetura e Organização de Computadores

Brasil

2025

UNIVERSIDADE FEDERAL DO RIO GRANDE

Gabriel da Costa Barbosa

Simulador de Caminho de Dados para o Ensino Integrado de Arquitetura e Organização de Computadores

Trabalho acadêmico apresentado ao Curso de Engenharia de Computação da Universidade Federal do Rio Grande, como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação

Orientador: Ewerson L. de S. Carvalho

Universidade Federal do Rio Grande – FURG

Centro de Ciências Computacionais

Curso de Engenharia de Computação

Brasil

2025

Resumo

O presente trabalho propõe o desenvolvimento de um simulador gráfico interativo voltado ao ensino introdutório de Arquitetura e Organização de Computadores, com ênfase na representação visual do *Datapath* e da Unidade de Controle durante a execução das instruções. A proposta surge diante da importância crescente de recursos visuais para o entendimento de conceitos complexos da área, especialmente em disciplinas introdutórias, nas quais os alunos frequentemente têm dificuldade em visualizar o funcionamento interno da CPU. Para justificar a necessidade da ferramenta, foi realizada uma revisão de trabalhos e simuladores existentes, como o Ripes, o MIPS-Datapath e o WebRISC-V, identificando limitações comuns como interfaces pouco intuitivas, ausência de controle passo a passo, falta de visualização dos sinais de controle ou uso de ISAs completas que dificultam o aprendizado. Como alternativa, foi projetada uma arquitetura computacional simplificada, adequada ao contexto educacional, acompanhada de um conjunto reduzido de instruções e uma Linguagem Assembly didática. A ferramenta desenvolvida permite observar o fluxo de informações e os sinais de controle da CPU em tempo real, nos modos Monociclo e Multiciclo. Ela é acessível via web, com código aberto, garantindo seu uso por diferentes instituições e usuários. Durante a etapa inicial do projeto, foram concluídas atividades como a revisão bibliográfica, definição da arquitetura e metodologia, levantamento de requisitos, prototipagem da interface e implementação parcial da ferramenta. Na fase final do projeto, foi dada continuidade com a finalização dos modos de simulação, testes, integração da interface completa. Os resultados obtidos indicam que o simulador atende aos objetivos propostos, permitindo a visualização do fluxo de dados e dos sinais de controle durante a execução das instruções, e os testes realizados demonstraram o funcionamento nos modos Monociclo e Multiciclo.

Palavras-chave: Simulação. *Datapath*. Unidade de Controle. Ensino de Arquitetura.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ADD	Instrução que Soma dois registradores
AND	Instrução que faz E lógico entre dois registradores
C3	Centro de Ciências Computacionais da FURG
CISC	Computador com um Conjunto Complexo de Instruções
CPU	Unidade Central de Processamento
E/S	Entrada e Saída
FSM	Máquina de Estados Finitos
GPR	*General Purpose Register* (Banco de Registradores da CPU)
HLT	Instrução que Encerra a execução do programa
HTML	Linguagem de Marcação de Hipertexto
IR	*Instruction Register* (Registrador de Instrução)
ISA	Arquitetura de Conjunto de Instruções
JMP	Instrução que Salta para um endereço
JN	Instrução que Salta para um endereço se flag N ativada
JZ	Instrução que Salta para um endereço se flag Z ativada
LDA	Instrução que Carrega conteúdo de memória em um registrador
LDAI	Instrução que Carrega um valor imediato em um registrador
LM	Linguagem de Máquina
MAR	*Memory Address Register* (Registrador de Endereço de Memória)
MDR	*Memory Data Register* (Registrador de Dados de Memória)
MIPS	*Microprocessor without Interlocked Pipeline Stages*
NOT	Instrução que faz NÃO lógico entre dois registradores
OR	Instrução que faz OU lógico entre dois registradores

PC	*Program Counter* (Contador de Programa)
PG	Projeto de Graduação
RAM	Memória de Acesso Aleatório
RISC	Computador com um Conjunto Reduzido de Instruções
STA	Instrução que Armazena o conteúdo de um registrador na memória
SUB	Instrução que Subtrai dois registradores
UC	Unidade de Controle
ULA	Unidade Lógica e Aritmética

Sumário

1 Introdução

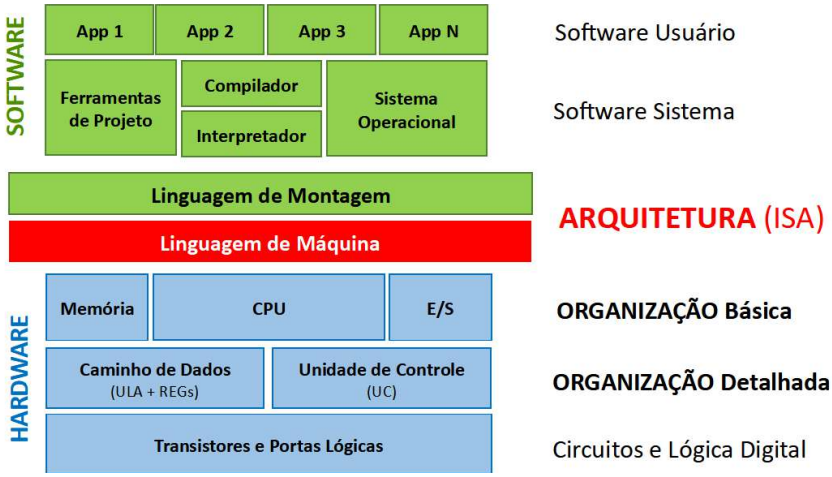
Sistemas Computacionais são máquinas programáveis, compostos por uma parte física denominada Hardware e outra lógica, o Software. São computadores usados para processar dados brutos, gerando informação útil. Eles estão presentes em praticamente todas as áreas da sociedade, sendo fundamentais para o funcionamento de dispositivos e serviços utilizados diariamente. Desde computadores pessoais, servidores e smartphones até dispositivos, como eletrodomésticos, veículos, equipamentos médicos e sistemas industriais, todos contam com algum tipo de Sistema Computacional. Esses sistemas oferecem benefícios significativos, como automação de tarefas, aumento de eficiência, processamento rápido de informações e conectividade. Cabe destacar que, além dos computadores convencionais, os Sistemas Embarcados também são computadores, contudo eles são projetados para executar funções específicas em dispositivos maiores. A Figura 1 ilustra exemplos de dispositivos que incorporam Sistemas Computacionais, presentes em nosso cotidiano.

Figura 1 – Dispositivos que incorporam Sistemas Computacionais

Fonte: Griebler (2010, p. 14)

Para compreender melhor os Sistemas Computacionais, é comum estudá-los a partir de diferentes níveis de abstração. Essa abordagem facilita o entendimento das diversas camadas que compõem um sistema, isolando a complexidade de cada etapa. Nos cursos de Computação, costuma-se dar maior ênfase ao estudo do Software, como Linguagens de Programação, Sistemas Operacionais e Algoritmos. No entanto, é igualmente importante explorar os níveis mais baixos, onde o funcionamento básico da máquina é definido. A Figura 2 apresenta uma visão multinível da Organização dos Sistemas Computacionais, destacando desde os níveis mais abstratos até os mais próximos do Hardware. Nos níveis mais baixos temos a Linguagem de Máquina, que compreende as instruções binárias diretamente executadas pelo processador, e o nível de Lógica Digital, estuda-se a construção de Circuitos Digitais com Portas Lógicas e Transistores.

Figura 2 – Máquina Multinível para um Sistema Computacional



Fonte: Elaborada pelo autor

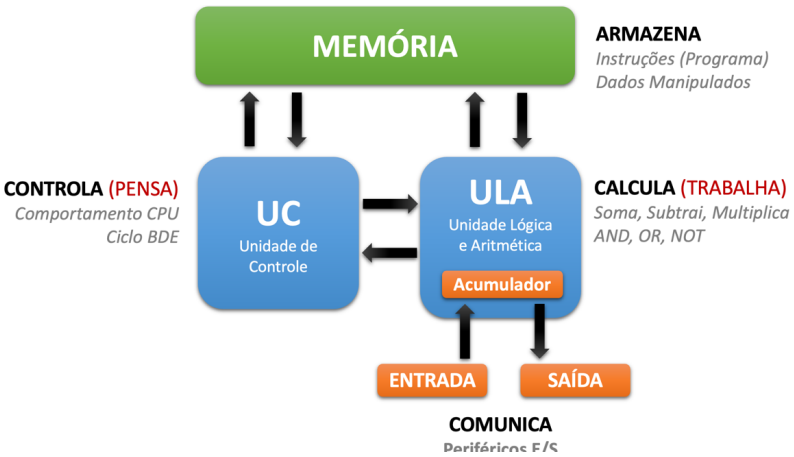
Neste trabalho, o foco está nos níveis de Arquitetura e Organização de Computadores. O estudo dessas áreas é essencial para entender como os programas são interpretados e executados pelo Hardware, e como os componentes físicos de um sistema interagem para cumprir as instruções. Autores como Patterson e Hennessy destacam que compreender a Arquitetura e a Organização de Computadores permite projetar sistemas mais eficientes, escolher melhor os componentes de Hardware e desenvolver Software otimizado e compatível com diferentes plataformas (HENNESSY, 2020). Tanenbaum e Austin reforçam esse ponto ao destacar que Hardware e Software são logicamente equivalentes, indicando a importância de compreender suas interações para garantir compatibilidade e desempenho (TANENBAUM; AUSTIN, 2013). Complementando, Mano e Kime observam que entender o projeto lógico dos componentes de Hardware é fundamental para a implementação eficiente de sistemas (MANO; KIME, 2008).

A Arquitetura de Computadores refere-se à estrutura conceitual de um Sistema Computacional, especialmente à interface entre Software e Hardware. Um dos principais conceitos nessa área é a ISA (*Instruction Set Architecture*), que define o Conjunto de Instruções que o processador é capaz de executar, os Modos de Endereçamento disponíveis, os Registradores e o comportamento esperado das operações. A ISA é fundamental para garantir a compatibilidade entre programas e diferentes processadores. Nesse nível, são estudados tópicos como Linguagem de Máquina e Assembly, permitindo entender como códigos escritos por humanos se traduzem em operações compreensíveis pela máquina.

Por outro lado, a Organização de Computadores trata da implementação física e lógica da Arquitetura definida. Ela descreve como os componentes internos do computador são interconectados e como funcionam para executar as instruções. Um modelo clássico para representar essa estrutura é o modelo de Von Neumann, que propõe uma Organização

composta por cinco elementos principais: a Unidade de Controle (UC), a Unidade Lógica e Aritmética (ULA), a Memória e os dispositivos de Entrada/Saída. A Figura 3 ilustra esse modelo, evidenciando a interação entre os componentes do sistema.

Figura 3 – Modelo de Computador proposto por von Neumann



Fonte: Elaborada pelo autor

Em abordagens mais modernas de projeto de processadores, é comum dividir internamente a CPU em duas grandes partes: o Caminho de Dados e o Controle. O *Datapath* ou Caminho de Dados é responsável por conduzir os dados pelos diferentes blocos funcionais do processador, como a ULA, Registradores e Multiplexadores, permitindo a execução das instruções. Já o Controle determina quais operações o Caminho de Dados deve realizar, com base nas instruções recebidas. Essa separação permite uma implementação mais organizada e modular do processador, facilitando a compreensão e o aprimoramento do sistema. Uma visão detalhada de *Datapath* e Controle encontra-se no Capítulo seguinte.

1.1 Simulação de Processadores

Para compreender de forma mais completa o funcionamento de um processador, é muito útil visualizar a execução de um programa de maneira gráfica, acompanhando o Caminho dos Dados em tempo real. Uma simulação gráfica permite observar o fluxo de informações por cada componente da CPU durante a execução de uma instrução, proporcionando uma aprendizagem mais concreta e intuitiva sobre a organização dos computadores. Além disso, ao observar como cada instrução do programa percorre os blocos funcionais, torna-se mais fácil entender a razão de existência de certos elementos no *Datapath* bem como o papel que cada um desempenha. Essa abordagem não somente favorece o entendimento, mas também tem potencial para despertar maior interesse do aluno ao tornar visível o que normalmente é abstrato.

Entre os Simuladores Não Gráficos mais relevantes para o estudo da Arquitetura de Computadores, destacam-se quatro ferramentas. O Simple 8-bit Assembler Simulator (SILVIUS, 2014) é uma ferramenta educacional que permite a simulação de instruções em uma Arquitetura de 8 bits. Ele é usado para introduzir os conceitos fundamentais de execução de instruções e manipulação de registradores textualmente, sem interfaces gráficas complexas. O Simulador Venus (CS61C Staff, 2019) é uma alternativa online para simulação da Arquitetura MIPS. Ele conta com uma interface simples baseada em texto que permite foco direto no código e nos Registradores. Já o Simulador OneCompiler (OneCompiler Team, 2019) é uma plataforma baseada na web que suporta múltiplas linguagens, incluindo *Assembly*, oferecendo um simulador textual com resposta imediata, ideal para testes rápidos de código. Por fim, o Emulador Assembly x86 (Emulator.IO Team, 2014) permite a simulação de código *Assembly* para a Arquitetura x86 textualmente, auxiliando no aprendizado das instruções e do funcionamento interno da CPU.

Entre os Simuladores Gráficos mais relevantes, destacam-se quatro ferramentas. O Ripes (PEDERSEN, 2018) é um simulador interativo voltado para a Arquitetura RISC-V, permitindo visualizar o Pipeline, os Registradores e a Memória de forma sincronizada com a execução das instruções. O Simulador MIPS 101 (DAVIS, 2018) apresenta uma abordagem gráfica para fim educacional, oferecendo uma visualização do *Datapath* de um processador MIPS simplificado, ideal para iniciantes. O Simulador MIPS-Datapath (MIPS Datapath Dev Team, 2015) oferece uma interface gráfica clara para acompanhar o ciclo de execução das instruções MIPS, focando no controle e na propagação de sinais pelo processador. Já o Simulador WebRISC-V (WebRISC-V Project, 2022) é uma plataforma online com visualização gráfica interativa da execução de instruções RISC-V, destacando o fluxo de dados em tempo real.

1.2 Motivação do Trabalho

Apesar da utilidade dos Simuladores Não Gráficos, eles apresentam limitações importantes no ensino introdutório de Organização de Computadores. O principal problema é a ausência de uma visualização do que ocorre dentro da CPU quando uma instrução é executada. Essa ausência torna difícil compreender o impacto de cada linha de código sobre os Registradores, a Memória e os Sinais de Controle. Além disso, muitos desses simuladores utilizam Conjuntos de Instruções completos, como o MIPS ou o ARM, que embora relevantes academicamente, acabam exigindo do aluno um foco excessivo em detalhes específicos da Arquitetura, desviando a atenção dos conceitos fundamentais que devem ser aprendidos primeiro, como o funcionamento básico do *Datapath* e da UC.

A utilização de Simuladores Gráficos vai além de simplesmente tornar o aprendizado mais atrativo, ela contribui diretamente para a construção de uma compreensão mais profunda e integrada de conceitos. Ao permitir que o estudante veja, em tempo real, como os sinais percorrem a CPU e como as instruções afetam seus componentes internos, a simulação gráfica favorece a criação de conexões entre diferentes níveis de abstração, algo que frequentemente se perde quando esses conteúdos são estudados de forma fragmentada. Essa abordagem integrada é especialmente importante para estudantes em fase de formação, pois reduz a distância entre o código escrito e sua execução.

No entanto, os Simuladores Gráficos também enfrentam limitações. Alguns deles possuem interfaces complexas ou densas demais, o que pode dificultar o uso em contextos educacionais iniciais, especialmente quando o foco deve estar nos conceitos e não na ferramenta. Além disso, muitos adotam ISAs completas, o que pode novamente desviar o foco do que realmente importa nos estágios iniciais do aprendizado. Outra limitação encontrada é a ausência de suporte ao modo passo a passo, que impede o estudante de acompanhar detalhadamente cada fase do ciclo de execução, dificultando a construção de um raciocínio mais analítico. Em alguns casos, o fluxo de informações sequer é visível, o que compromete ainda mais seu valor didático.

Diante dessas limitações, o presente trabalho propõe desenvolver um Simulador Gráfico focado especificamente no ensino introdutório de Arquitetura e Organização de Computadores. A proposta visa criar uma ferramenta que permita visualizar, de maneira clara e acessível, como as instruções percorrem o *Datapath* da CPU, quais Sinais de Controle são ativados e como o estado dos Registradores e da Memória se altera a cada ciclo. O objetivo não é simular uma ISA complexa, mas oferecer uma abstração simplificada o suficiente para os discentes poderem se concentrar na aprendizagem dos conceitos fundamentais. Com isso, espera-se proporcionar um ambiente mais eficaz para a assimilação dos conteúdos estudados.

Complementando essa proposta, o simulador será projetado com foco na facilidade de uso. A interface será planejada para destacar visualmente o Fluxo de Dados e os Sinais de Controle, com uma disposição clara dos elementos da CPU, a fim de facilitar o acompanhamento do funcionamento do sistema, mesmo para quem está tendo o primeiro contato com o tema. A aplicação será disponibilizada via Web, permitindo que discentes do C3 bem como de outras instituições possam utilizá-la como recurso didático. Além disso, o código-fonte do simulador será aberto, permitindo que outros pesquisadores, docentes ou discentes possam contribuir com melhorias, adaptando a ferramenta a novas necessidades ou integrando a ela funcionalidades adicionais.

1.3 Objetivos do Trabalho

Os Objetivos Estratégicos deste trabalho envolvem diversas ações complementares ao desenvolvimento do simulador propriamente dito, fundamentais para a consolidação do projeto como uma contribuição acadêmica significativa. Um dos principais objetivos é estudar e avaliar o uso de simuladores existentes voltados ao ensino de Arquitetura e Organização de Computadores, investigando suas funcionalidades, limitações e adequação ao contexto educacional. Também se busca praticar e aplicar os princípios de Projeto e Desenvolvimento de Software segundo as boas práticas da Engenharia de Software, com ênfase na clareza, manutenibilidade e reutilização de código. Além disso, há a intenção de Projetar e Desenvolver Aplicações para ambiente Web, utilizando tecnologias que garantam, portabilidade e uma interface amigável, facilitando o uso por discentes e docentes.

No que se refere aos Objetivos Específicos do trabalho, pretende-se propor uma Arquitetura Computacional simplificada, com finalidades exclusivamente educacionais, fundamentada em um conjunto reduzido de instruções e operações essenciais para o aprendizado. A partir dessa Arquitetura, será desenvolvido um Simulador Textual, que permita escrever e executar programas utilizando sua Linguagem *Assembly*, com ênfase na clareza didática. Complementarmente, será projetada uma Organização interna para a Arquitetura, incluindo o detalhamento de um *Datapath* e de uma Unidade de Controle que viabilizem a execução das instruções de forma compreensível para os estudantes.

Também será desenvolvido um Simulador Gráfico capaz de representar visualmente o Fluxo de Dados e os Sinais de Controle durante a execução de instruções, permitindo o acompanhamento passo-a-passo do funcionamento da CPU. Todo esse sistema será acompanhado por documentação, que explique a Arquitetura, a Organização, a Linguagem e o uso do simulador. A ferramenta e seu código-fonte serão públicos, garantindo acesso por parte de estudantes e professores. Outro objetivo é realizar uma validação prática da ferramenta com usuários reais, obtendo feedback sobre sua usabilidade no apoio ao ensino de Organização de Computadores, caso o mesmo esteja disponível a tempo.

1.4 Estrutura do Volume

No Capítulo 1 foram apresentados os fundamentos dos Sistemas Computacionais, suas classificações e importância em diferentes contextos. Para isso, foram abordados os níveis de abstração utilizados para o estudo desses sistemas, com foco nos níveis de Arquitetura e Organização de Computadores. Também foram discutidos os conceitos de *Datapath* e Unidade de Controle, além da relevância da Simulação Gráfica para o ensino. Por fim, foi apresentada uma breve análise dos simuladores existentes, seguida da motivação do trabalho proposto, seus objetivos estratégicos e específicos.

Seguindo, o Capítulo 2 apresenta o Referencial Teórico para fundamentar o trabalho desenvolvido. Essa fundamentação está dividida em duas partes. A primeira trata dos conceitos relacionados à Arquitetura e Organização de Computadores, abordando tópicos como formatos de instrução e o funcionamento do *Datapath* e da Unidade de Controle. A segunda parte é dedicada à Revisão do Estado da Arte, incluindo simuladores educacionais existentes, com destaque para suas funcionalidades, limitações e adequação ao ensino.

O Capítulo 3 descreve a Arquitetura desenvolvida especialmente para este trabalho. Inicialmente, são apresentadas as principais decisões que nortearam o projeto da Arquitetura, como a escolha de um conjunto reduzido de instruções e a definição de um número limitado de registradores. Em seguida, são detalhados o conjunto de instruções, os formatos adotados, a Linguagem *Assembly* correspondente e os elementos que compõem o *Datapath* e o Controle.

Já o Capítulo 4 apresenta a Metodologia de trabalho adotada ao longo do desenvolvimento do Projeto de Graduação, bem como o projeto e a implementação da ferramenta simuladora. Nele são descritas as etapas seguidas, as decisões técnicas tomadas, as ferramentas utilizadas e a estrutura geral do sistema, incluindo a interface gráfica, o montador e os modos de simulação Monociclo e Multiciclo.

O Capítulo 5 descreve os Resultados Obtidos com a implementação do simulador. Nele são discutidos os testes realizados para validar o funcionamento das instruções e de alguns programas desenvolvidos justamente para testar todo conjunto de instruções. Também encontra-se neste capítulo um pequeno manual de uso, com orientações sobre o acesso à ferramenta, suas principais funcionalidades e a forma correta de utilizá-la.

Finalizando, o Capítulo 6 apresenta as conclusões do trabalho, discutindo as contribuições alcançadas em relação aos objetivos propostos. Além disso, são analisadas as limitações do projeto, indicando o que foi implementado e o que não pôde ser realizado conforme o escopo definido, bem como são apontadas possibilidades de trabalhos futuros.

2 Trabalhos Relacionados

Neste Capítulo serão abordados dois tópicos principais. Na seção 2.1, será apresentada a Fundamentação Teórica, a qual é essencial para o entendimento dos conceitos relacionados ao trabalho proposto. Essa revisão visa esclarecer os principais termos, definições e tecnologias que servem de base para o desenvolvimento deste projeto. Em seguida, na seção 2.2, será realizada a revisão dos Trabalhos Relacionados. Esta revisão é fundamental para nortear a proposta, uma vez que permite identificar lacunas e oportunidades de melhoria em relação ao que foi desenvolvido em trabalhos anteriores.

2.1 Fundamentação Teórica

A Arquitetura de uma CPU, também conhecida como Arquitetura de Conjunto de Instruções (ISA - *Instruction Set Architecture*), pode ser entendida como a definição formal e abstrata do funcionamento da CPU, descrevendo como ela interage com o Software e com o ambiente externo. A ISA estabelece a interface entre o Hardware e o Software, determinando como os programas devem ser escritos para executar naquele processador.

Ao se especificar uma ISA, alguns pontos importantes precisam ser definidos para garantir uma implementação eficiente e funcional. Primeiramente, é necessário estabelecer o Conjunto de Operações, que determina quais instruções o processador será capaz de executar, incluindo operações aritméticas, lógicas, de controle de fluxo e de movimentação de dados. Também é fundamental definir os Tipos de Dados suportados pela Arquitetura, como inteiros, números em ponto flutuante e caracteres. Já os Modos de Endereçamento, indicam as formas disponíveis para acessar operandos na Memória ou em Registradores. Por fim, deve-se especificar os Formatos de Operandos, ou seja, como os operandos serão fornecidos às instruções, seja por Registradores, posições de Memória ou valores imediatos, e o Formato das Instruções, que corresponde à estrutura binária de cada instrução, incluindo campos para código da operação, operandos e demais parâmetros.

Existem diferentes abordagens na construção de uma ISA, sendo as duas principais conhecidas como CISC (*Complex Instruction Set Computer*) e RISC (*Reduced Instruction Set Computer*). ISAs CISC são caracterizadas por um conjunto de instruções amplo e variado, com instruções complexas capazes de realizar múltiplas operações com apenas um comando. Já as RISC possuem um conjunto de instruções mais enxuto e simplificado, com foco em instruções rápidas e de execução uniforme, geralmente uma instrução por Ciclo de Clock. Processadores CISC são mais comumente empregados para computação convencional, como em PCs e Notebooks. Já os Processadores RISC são mais encontrados em Sistemas Embarcados, como Smartphones, Switches e Consoles de Videogame.

Outro aspecto fundamental a ser considerado é a relação entre a ISA e a Organização da CPU. Enquanto a ISA representa a especificação lógica e abstrata da CPU, a Organização corresponde à forma como essa ISA será implementada fisicamente em Hardware. Em outras palavras, a Organização define aspectos como, os componentes internos, a quantidade de unidades funcionais e como elas se conectam. Um mesmo conjunto de instruções pode ser implementado por diferentes organizações. Um exemplo clássico é a Arquitetura x86, que ao longo das décadas é utilizada tanto pela Intel quanto pela AMD, com diversas variações organizacionais, mas sempre mantendo a mesma ISA. Essa abordagem garante a compatibilidade de Software, permitindo que programas escritos para uma determinada ISA continuem funcionando mesmo quando o Hardware passa por mudanças internas.

2.1.1 Linguagem de Máquina e *Assembly*

As CPUs, não são capazes de entender diretamente linguagens de programação de alto nível como C, Java ou Python. Isso ocorre porque, por se tratar de Hardware que opera sobre sinais elétricos representando valores numéricos simples (como 0s e 1s), a CPU precisa ser programada em uma linguagem também simples e diretamente interpretável pelo seu circuito lógico. Nesse contexto, surge a relação direta entre a Arquitetura da CPU e a sua Linguagem de Máquina (LM), que nada mais é que a forma mais elementar e próxima do Hardware e que se pode expressar a ISA.

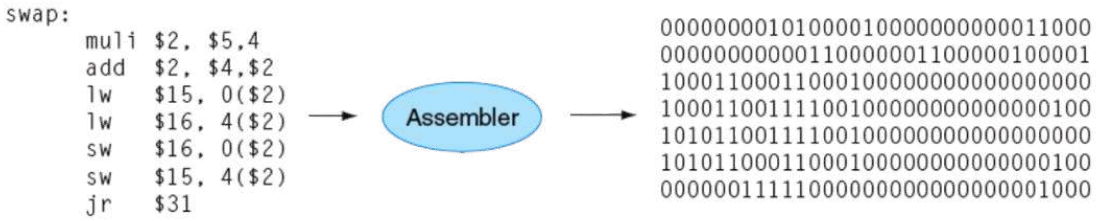
A Linguagem de Máquina descreve o conjunto de instruções que uma CPU consegue interpretar e executar diretamente. Cada instrução em Linguagem de Máquina é representada por uma sequência de bits, com um formato bem definido pela ISA. Essas instruções são organizadas em formatos de instrução, que estabelecem como os bits são distribuídos em campos, sendo cada campo responsável por um tipo de informação. Os principais campos incluem geralmente o *Opcode* (qual operação será realizada), os *Operandos* (quais Registradores ou posições de Memória serão utilizados), e, em alguns casos, *Valores Imediatos*. Cada combinação específica de bits possui um significado próprio, que a CPU interpreta para executar determinada operação.

Devido à dificuldade de programar manipulando sequências de bits, surgiu uma linguagem intermediária entre a Linguagem de Máquina e as Linguagens de Alto Nível. A Linguagem de Montagem (*Assembly*) é uma representação textual legível da Linguagem de Máquina, onde cada instrução é descrita por meio de mnemônicos (abreviações simbólicas que representam as operações). Por exemplo, ao invés de programar usando uma sequência binária `000000 01001 01010 01011 00000 100000`, o programador usa `ADD $t3, $t1, $t2`, tornando o desenvolvimento mais compreensível e menos propenso a erros.

É importante destacar que a Linguagem *Assembly* e a Linguagem de Máquina são equivalentes, já que cada instrução *Assembly* corresponde diretamente a uma única instrução em Linguagem de Máquina. No entanto, os computadores não entendem *Assembly*.

Por isso, é necessário utilizar um programa chamado Montador (*Assembler*), responsável por traduzir o código *Assembly* para o binário da Linguagem de Máquina. Para ilustrar essa equivalência, a Figura 4 apresenta um exemplo de código na Arquitetura MIPS.

Figura 4 – Linguagem *Assembly* montada para Linguagem de Máquina



Fonte: Universidade Federal de Pernambuco (s.d., p. 25)

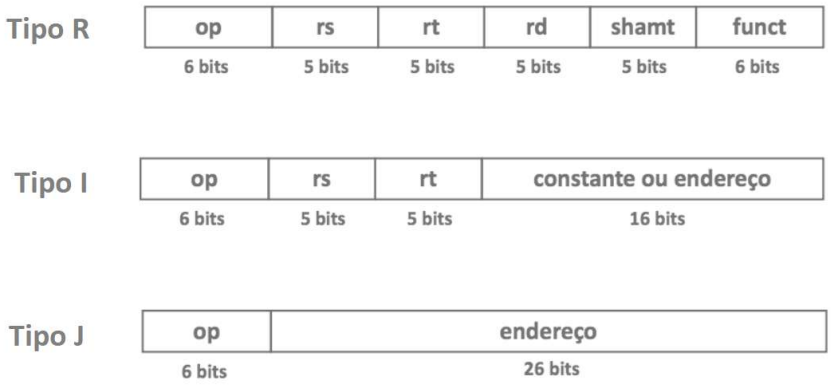
Cabe destacar que as explicações que seguem ao longo deste trabalho serão feitas no nível *Assembly*. Essa escolha foi feita para facilitar a compreensão dos conceitos e operações da Arquitetura, evitando a complexidade da leitura direta de códigos binários.

2.1.2 *Datapath* e Controle Monociclo

O Projeto de uma CPU pode ser dividido em dois componentes fundamentais: o *Datapath* e a Unidade de Controle. Essa separação é essencial na área de Arquitetura e Organização de Computadores, pois permite analisar de forma clara como os dados são processados e como as decisões de controle são tomadas. O *Datapath* compreende os elementos responsáveis pelas operações aritméticas, lógicas e de movimentação de dados, enquanto a Unidade de Controle determina a sequência e o momento em que essas operações ocorrem, segundo as instruções do programa. Essa abordagem, amplamente utilizada para fins educacionais, é adotada por Patterson e Hennessy (HENNESSY, 2020). Aqui, será seguido o mesmo caminho proposto pelos Autores para a explicação do funcionamento interno de uma CPU baseada na Arquitetura MIPS.

O MIPS (*Microprocessor without Interlocked Pipeline Stages*) é uma ISA do tipo RISC, frequentemente adotada no meio acadêmico por sua clareza estrutural e simplicidade no ensino de conceitos. Seus formatos de instrução são divididos em três tipos principais. Conforme a Figura 5, o Formato R é utilizado para instruções aritméticas e lógicas entre registradores (como `ADD`, `SUB`). O Formato I é usado em instruções que envolvem constantes imediatas ou endereços relativos (como `LW`, `SW` e `BEQ`). Já o Formato J é utilizado em instruções de salto incondicional (`J`). Cada formato possui campos bem definidos, como opcode (`op`), registradores de origem (`rs`) e destino (`rt`), e campos de imediato ou endereço (`rd`), dependendo do tipo.

Figura 5 – Formatos de Instrução da Arquitetura MIPS



Fonte: Elaborada pelo autor

Assim como no Livro de Patterson e Hennessy, as explicações a seguir se concentram em um subconjunto reduzido de instruções, suficiente para ilustrar os princípios de funcionamento de um processador MIPS simplificado. O projeto de um *Datapath* envolve identificar as etapas necessárias para executar essas instruções e estabelecer os caminhos que os dados percorrem entre os componentes. Para isso, inicialmente, considera-se a etapa de Busca, comum a todas as instruções, que requer uma Memória de Instruções, um Contador de Programa (`PC`) e um Somador para gerar o próximo endereço.

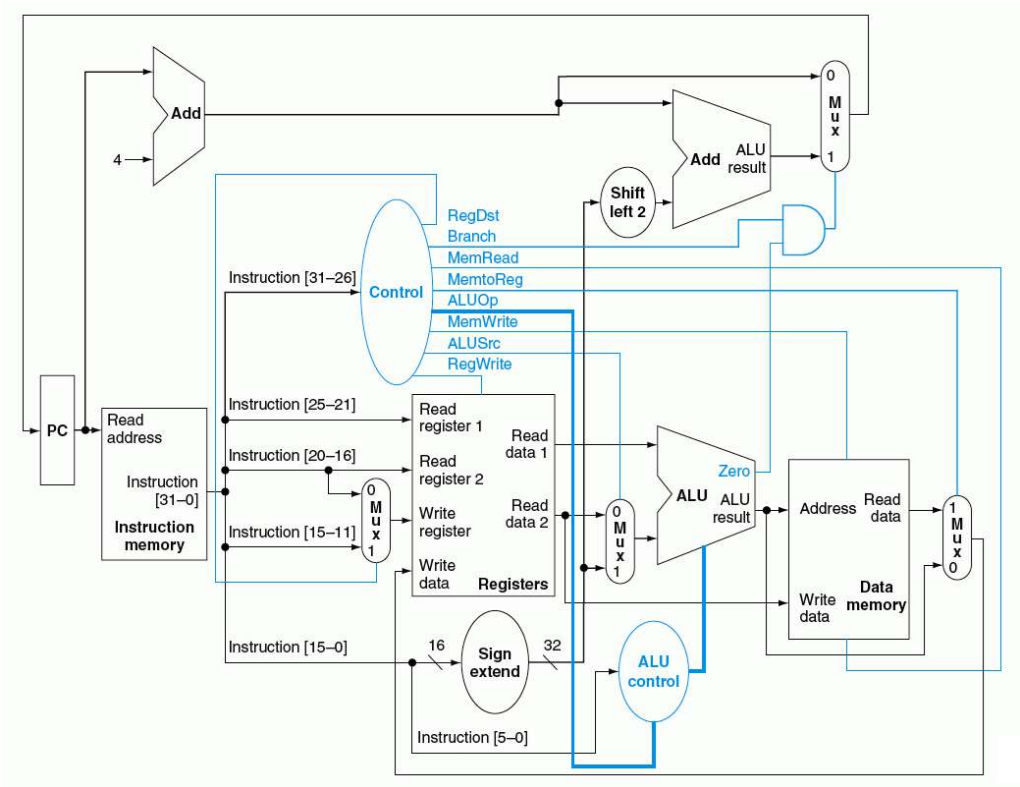
Em seguida, analisa-se cada tipo de instruções, identificando os recursos necessários. Instruções aritméticas utilizam um Banco de Registradores e uma ULA, enquanto as de acesso à memória demandam uma Memória de Dados conectada ao mesmo banco. Já as instruções de desvio atuam diretamente sobre o `PC`, com base em campos da instrução e sinais de controle. A partir dessa análise, são definidas as conexões entre os componentes e inseridos Multiplexadores onde há múltiplas fontes para um mesmo destino. Ao final, todos os caminhos descobertos são reunidos para formar o *Datapath* completo.

A Figura 6 apresenta uma visão geral do *Datapath* proposto para o MIPS Simplificado. Dentre os principais blocos presentes no diagrama, destacam-se a Memória de Instruções, os Registradores, a ULA, a Memória de Dados, além de componentes auxiliares como Multiplexadores, Extensores de Sinal e unidades de Deslocamento de Bits. Cada um desses blocos desempenha uma função específica no processamento das instruções, formando juntos o caminho que os dados percorrem durante a execução.

Na mesma Figura vemos ainda a representação da Unidade de Controle. Ela é responsável por gerar os Sinais de Controle apropriados para cada tipo de instrução, com base nos campos da instrução recebida. Esses sinais determinam ações como seleção de operandos, ativação de escrita em Registradores, controle de leitura/escrita de Memória, e escolha das operações da ULA. Na solução Monociclo, cada instrução é executada

completamente em um único Ciclo de Clock, o que significa que todas as etapas da instrução ocorrem dentro do mesmo intervalo de tempo, desde sua Busca até a Escrita do Resultado. Todos os Sinais de Controle são ativados ao mesmo tempo, permitindo o uso de uma lógica de controle simples baseada em Circuitos Combinacionais. Tal abordagem não mais é usada, contudo é estudada por facilitar o entendimento inicial do funcionamento da CPU.

Figura 6 – *Datapath* e Controle Monociclo para o MIPS Simplificado



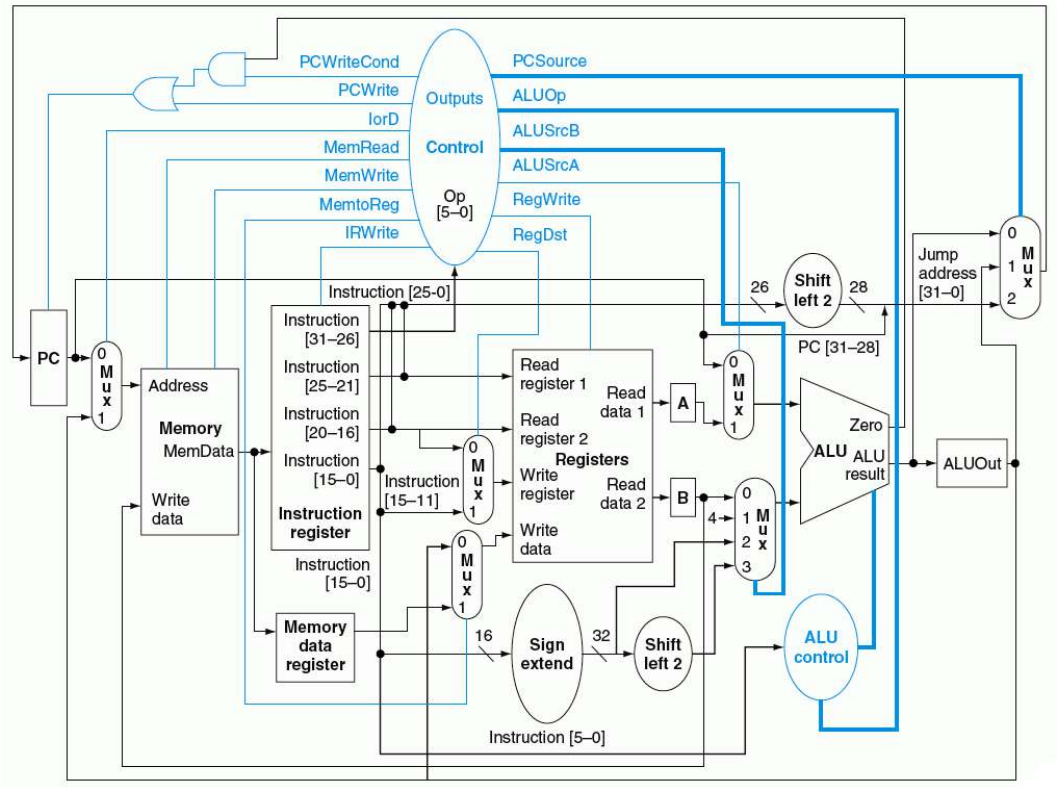
Fonte: David Patterson; John Hennessy (2020, p. 25)

2.1.3 *Datapath* e Controle Multiciclo

O *Datapath* Multiciclo segue uma abordagem diferente. Enquanto no *Datapath* Monociclo todas as instruções eram executadas em um único Ciclo de Clock, no Multiciclo a execução de cada instrução é dividida em múltiplas etapas. Para o projeto da solução Multiciclo, o primeiro passo consiste em dividir cada instrução em etapas, de forma que cada uma gaste exatamente um Ciclo de Clock. Essas etapas devem ter complexidade semelhante e utilizar uma única unidade funcional para garantir equilíbrio no tempo de execução. Em seguida, é necessário inserir estruturas intermediárias de armazenamento, como Registradores, que mantêm os dados gerados em uma etapa para serem usados na etapa seguinte. Por fim, é preciso reprojetar o controle, que passa a ser sequencial, com sinais sendo gerados por etapas conforme uma sequência de estados de operação.

A Figura 7 apresenta o *Datapath* Multiciclo para o MIPS Simplificado. Visualmente, o *Datapath* Multiciclo possui blocos semelhantes aos do modelo monociclo, incluindo a Memória de Instruções, os Registradores, a ULA, a Memória de Dados, Multiplexadores e unidades de extensão de sinal. No entanto, um diferencial importante é a presença de Registradores Intermediários (como `IR`, `A`, `B`, `ALUOut`, entre outros), utilizados para armazenar temporariamente os dados e resultados de uma fase para a outra. Além disso, pode-se notar que as Memórias de Dados e Instrução foram unificadas.

Figura 7 – *Datapath* e Controle Multiciclo para o MIPS Simplificado



Fonte: David Patterson; John Hennessy (2020)

A Unidade de Controle (UC) Multiciclo é implementada de forma distinta da do modelo anterior. Neste caso, o controle é representado por meio de uma Máquina de Estados Finita (FSM, *Finite State Machine*), caracterizando um circuito sequencial. A UC Multiciclo determina, a cada Ciclo de Clock, qual etapa da execução da instrução será realizada, ativando apenas os Sinais de Controle correspondentes à fase atual. Esta abordagem Multiciclo é a atualmente utilizada nos processadores, visto que permite que diferentes instruções sejam concluídas em tempos distintos, já que cada uma usa apenas as etapas necessárias, conforme sua complexidade.

Não cabe aqui discutir qual abordagem é superior. Ambas as estratégias são tradicionalmente ensinadas na disciplina de Arquitetura e Organização de Computadores,

e por isso, são consideradas no desenvolvimento do simulador proposto. As Figura 6 e Figura 7 têm como objetivo ilustrar as diferenças estruturais entre os modelos, mas não cabe aqui explicá-las em detalhes, uma vez que a Arquitetura MIPS não é o foco do trabalho. No próximo Capítulo, *Datapath* e Controle voltarão a ser discutidos, dessa vez baseados na ISA projetada especificamente para este simulador. No entanto, ao observar as figuras apresentadas, já é possível perceber que os caminhos de dados podem se tornar complexos mesmo em versões simplificadas de processadores.

2.2 Trabalhos Relacionados

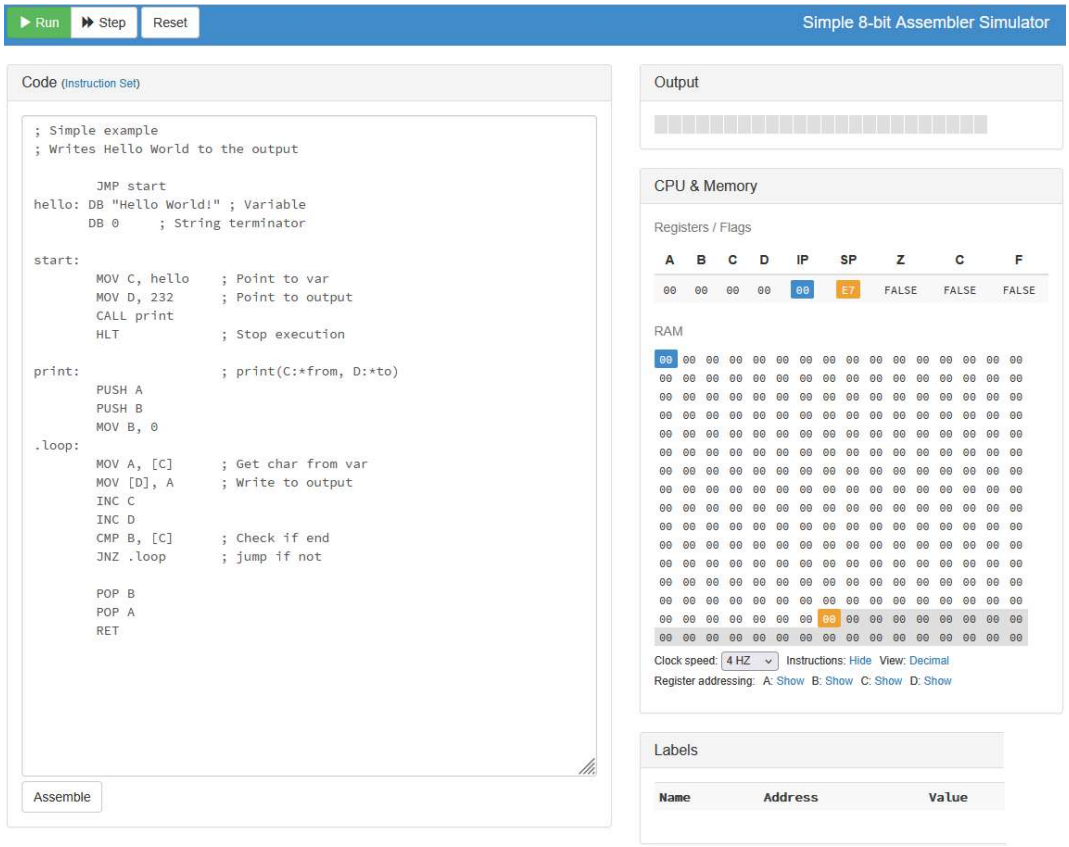
O estudo de Arquitetura e Organização de Computadores é fundamental para a formação de profissionais de Computação, pois permite compreender como o Hardware executa as instruções de Software, desde o nível mais baixo. Conforme mencionado na Introdução, estudar a Arquitetura permite conhecer o que a CPU sabe fazer, enquanto estudar a Organização permite compreender como a CPU faz. Estudar esses dois conceitos separadamente pode dificultar a compreensão, já que a execução de uma instrução envolve aspectos de ambas as áreas. Por isso, é importante estudá-las de forma integrada, conectando teoria e prática para um entendimento mais completo.

Dentro desse contexto, os Simuladores de Arquitetura e Organização se tornam ferramentas essenciais para facilitar o aprendizado. Eles permitem visualizar o funcionamento interno de uma CPU e observar, de forma controlada, como as instruções são processadas passo a passo. Esta Seção está organizada em duas partes. Primeiramente, serão apresentados alguns simuladores que não possuem visualização do *Datapath*. Em seguida, serão apresentados simuladores que permitem visualização.

2.2.1 Simuladores Sem *Datapath*

O Simple 8-bit Assembler Simulator foi desenvolvido em 2014 por Tony Silvius (SILVIUS, 2014). Ele tem como principal objetivo permitir a execução de programas escritos em uma arquitetura educacional de 8 bits. Este simulador trabalha diretamente com código em *Assembly*, oferecendo ao usuário a possibilidade de carregar programas e observar os resultados da execução. A entrada principal do simulador é o próprio código *Assembly*, que pode ser inserido manualmente. Durante a execução, o usuário pode acompanhar o conteúdo dos Registradores, do Contador de Programa e da Memória, ainda que sem uma visualização gráfica do *Datapath*. A Figura 8 apresenta a interface principal do simulador, onde temos a área de edição onde o código *Assembly* pode ser digitado. Na parte superior encontram-se os botões de controle da simulação. Na direita, são exibidos os valores atuais de cada Registrador e *flags*, e na parte inferior pode-se visualizar o conteúdo da Memória RAM durante a execução.

Figura 8 – Interface do Simple 8-BIT Assembler Simulator



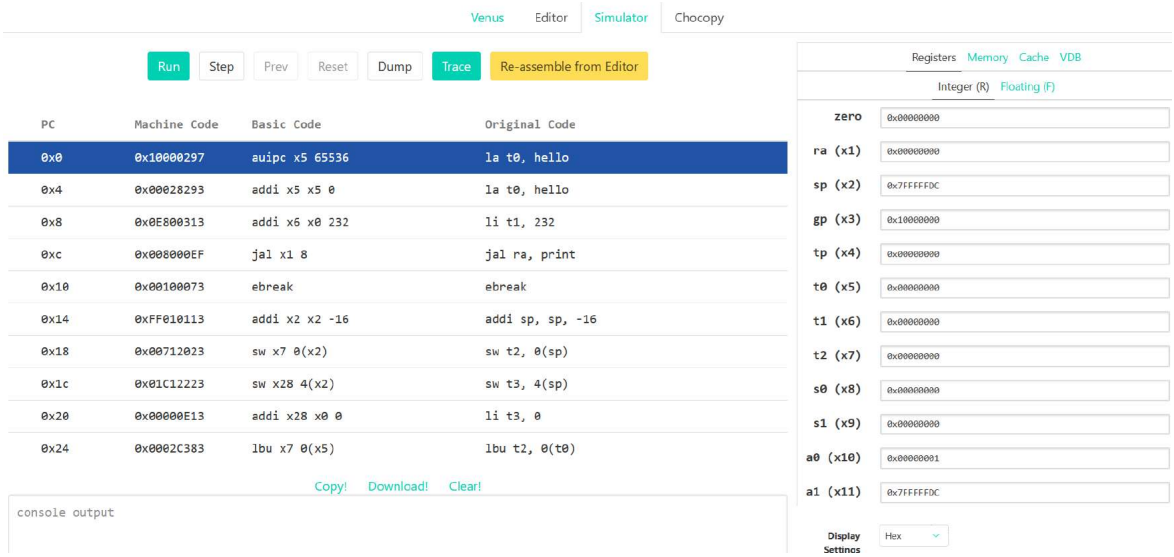
Fonte: Tony Silvius (SILVIUS, 2014)

O Simple 8-bit Assembler Simulator apresenta pontos positivos como a sua simplicidade e foco em conceitos fundamentais, como instruções, ciclos e operações com registradores. Além disso, possui acesso gratuito via Web. Contudo, sua limitação está na abrangência. Trata-se de uma arquitetura minimalista, que não representa diretamente uma arquitetura real como MIPS ou x86, restringindo seu uso em cursos mais avançados. Ainda assim, é uma excelente opção para estudantes em seu primeiro contato com o funcionamento de uma CPU.

O Simulador Venus foi desenvolvido em Berkeley (CS61C Staff, 2019). Ele tem como principal objetivo permitir a execução de programas escritos em *Assembly* RISC-V. Este simulador funciona via Web, oferecendo ao usuário a possibilidade de editar, compilar e executar programas com acompanhamento em tempo real do estado dos registradores. A entrada principal é o código *Assembly*, com execução passo a passo e monitoramento textual dos Registradores. A Figura 9 apresenta a interface principal do Venus. Na parte superior, encontra-se o menu para a área de edição de código. Logo abaixo, o comando para compilar e executar o código. À direita, vemos os Registradores, assim como a Memória. Na parte inferior, há também um espaço onde se pode ver o log de saída e mensagens

de erro. O Venus apresenta como pontos positivos sua praticidade e acesso gratuito via Web, sem necessidade de instalação. Além disso, sua interface responsiva e intuitiva torna o aprendizado mais acessível. Seu ponto negativo é a ausência de suporte a operações gráficas ou visuais, exigindo que o aluno entenda o funcionamento da CPU a partir de resultados textuais.

Figura 9 – Interface do Simulador Venus

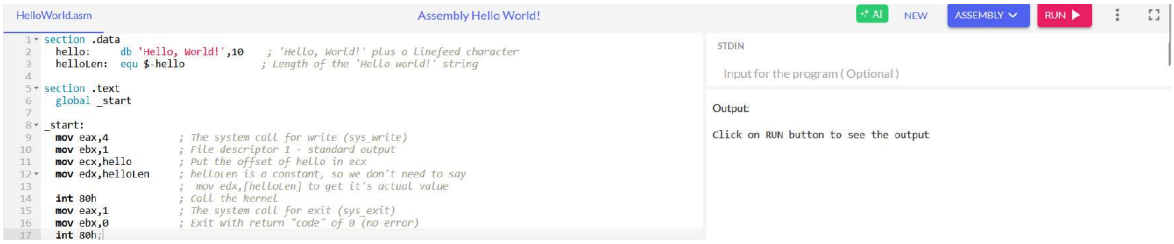


Fonte: Berkeley (CS61C Staff, 2019)

O Simulador OneCompiler (OneCompiler Team, 2019) é uma plataforma Web criada para simular e testar código em diversas linguagens, incluindo *Assembly*. Lançado com atualizações recentes em 2024, permite a execução rápida de trechos de código em um ambiente textual simples. O foco é na praticidade e na resposta imediata, sendo ideal para testes rápidos ou para exercícios introdutórios. A Figura 10 mostra a interface principal do OneCompiler. A parte esquerda é a área de código, onde o usuário digita o programa. A parte superior direita contém os botões de compilação e execução. Logo abaixo, está o terminal de saída. O principal ponto positivo do OneCompiler é o suporte a várias linguagens de programação. Contudo, ele é limitado para uso pedagógico, pois não oferece recursos específicos para acompanhar estados internos da CPU ou Registradores. Assim, é indicado para testes simples e não para simulações completas de arquitetura.

O Emulador Assembly x86 foi desenvolvido pela equipe Emulator.IO (Emulator.IO Team, 2014). Trata-se de uma ferramenta Web para simulação de código *Assembly* da Arquitetura x86. Este simulador permite acompanhar o estado dos Registradores, *flags* e Memória RAM. A ferramenta é utilizada para compreender como instruções x86 interagem com os recursos internos da CPU. A Figura 11 apresenta a interface do Emulador x86, incluindo a área central que permite a edição de código. A Seção abaixo mostra os

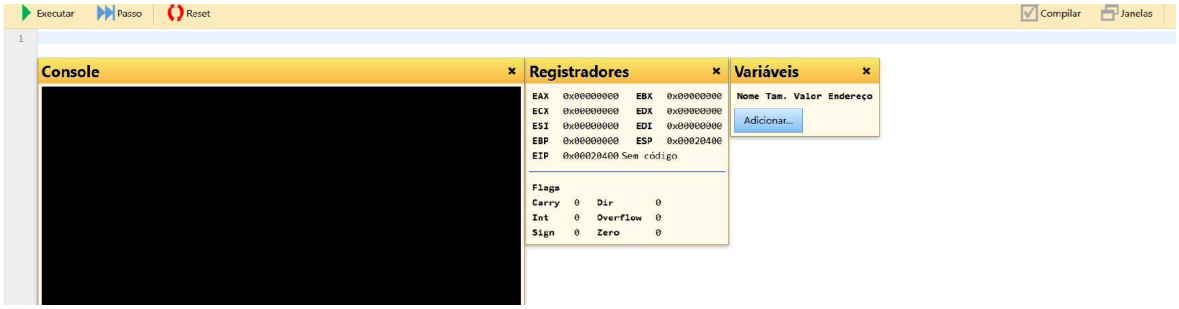
Figura 10 – Interface do Simulador OneCompiler



Fonte: OneCompiler (OneCompiler Team, 2019)

Registradores, a Memória RAM e o console com o resultado da execução. Existem ainda botões para controle de execução na parte superior. O Emulador x86 apresenta vantagens como o suporte a instruções reais da arquitetura Intel, interface limpa e simulação precisa dos registradores. Sua desvantagem é que, por ser textual e sem abstrações visuais, pode tornar a curva de aprendizado mais íngreme.

Figura 11 – Interface do Emulador Assembly x86

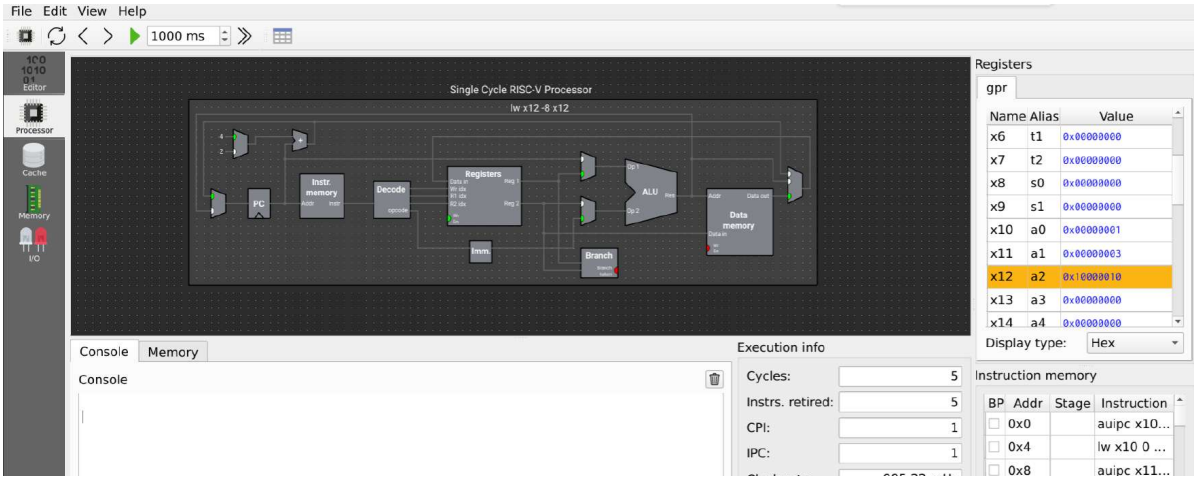


Fonte: Emulator.IO (Emulator.IO Team, 2014)

2.2.2 Simuladores Com *Datapath*

O Ripes foi desenvolvido por Anders D. Pedersen (PEDERSEN, 2018). Trata-se de uma ferramenta projetada para auxiliar no ensino de Arquitetura de Computadores, permitindo a visualização gráfica do Pipeline e do *Datapath* durante a execução das instruções. Ele é baseado na Arquitetura RISC-V, e seu objetivo principal é permitir que os discentes acompanhem a propagação de dados e sinais internamente no processador em cada Ciclo de Clock. A Figura 12 apresenta a interface principal do Ripes. Na esquerda, é possível visualizar o menu do editor de código. Na parte superior, as opções que permitem executar o programa, com avanço por ciclo ou instrução. No centro, o simulador exibe graficamente o Pipeline da CPU. Na direita, podem ser visualizados os valores atuais dos Registradores, Memória e `PC`, sincronizados com a execução.

Figura 12 – Interface do Simulador Ripes



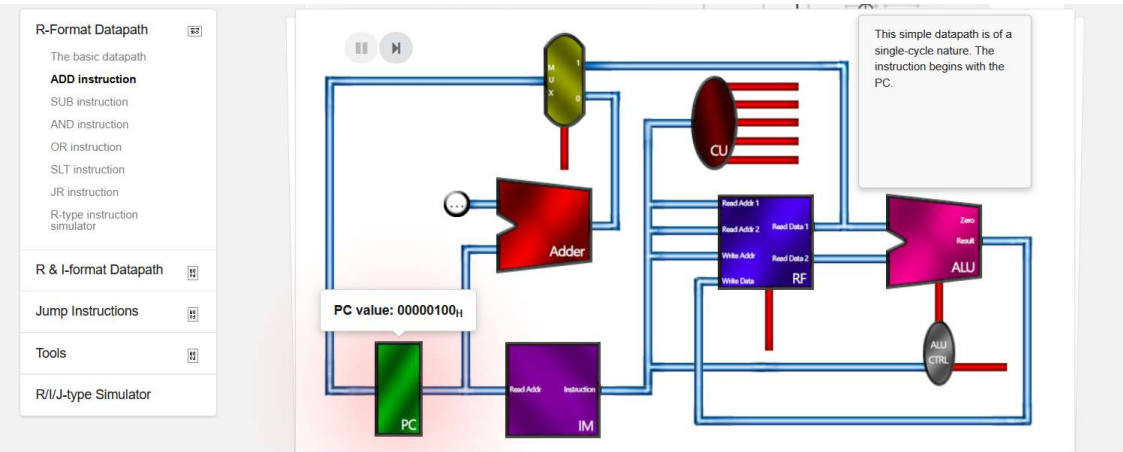
Fonte: Anders D. Pedersen (PEDERSEN, 2018)

Entre os pontos positivos do Ripes, destaca-se sua capacidade de oferecer uma visão integrada da execução das instruções em processadores baseados na ISA RISC-V, permitindo que o usuário acompanhe o andamento das instruções passo a passo. A ferramenta atualiza em tempo real o conteúdo dos Registradores e da Memória, facilitando a compreensão de como os dados são manipulados durante a execução. Outro ponto relevante é a possibilidade de escolher entre diferentes Arquiteturas RISC-V, o que permite comparar os efeitos das variações estruturais no comportamento e no desempenho do processador. O simulador oferece ainda um visualizador do *Datapath* que ajuda a ilustrar como as instruções transitam entre os componentes da CPU.

O Simulador MIPS 101 foi desenvolvido por Ben Davis (DAVIS, 2018). Trata-se de uma ferramenta projetada para auxiliar o ensino, permitindo a visualização gráfica do *Datapath* durante a execução das instruções. Ele é baseado na Arquitetura MIPS, e seu objetivo principal é permitir que os discentes acompanhem o percurso dos dados internamente no processador. A Figura 13 apresenta a interface principal do MIPS 101. Na parte esquerda, é possível visualizar o menu de seleção do código *Assembly*. A parte superior permite executar o programa passo a passo, avançando informações pelos componentes. No centro, o simulador exibe graficamente o *Datapath*, destacando os componentes ativados. Já na parte superior direita, podem ser visualizadas as explicações sobre o avanço correspondente.

Entre os pontos positivos do MIPS 101, destacam-se a sua interface, a possibilidade de acompanhar a execução em nível de componentes internos da CPU e a visualização gráfica do *Datapath*, que contribui significativamente para o aprendizado. Por outro lado, algumas limitações podem ser observadas, como a necessidade de selecionar a instrução a ser executada individualmente. Ou seja, este simulador não suporta códigos completos.

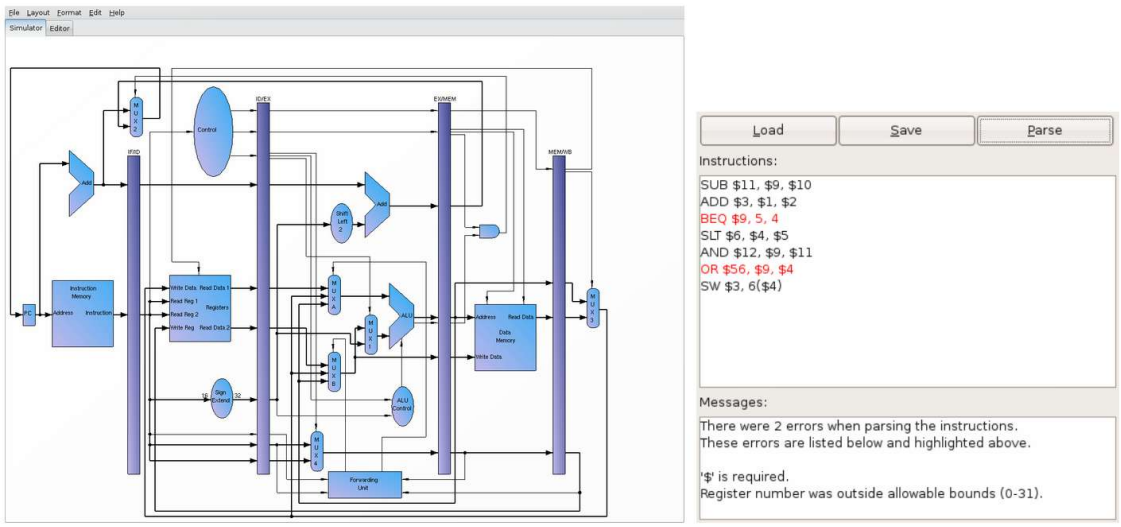
Figura 13 – Interface do Simulador MIPS 101



Fonte: Ben Davis (DAVIS, 2018)

O Simulador MIPS-Datapath (MIPS Datapath Dev Team, 2015) é uma ferramenta que oferece uma interface gráfica clara para acompanhar o ciclo de execução das instruções MIPS, focando no controle e na propagação de sinais pelo processador. Ele é voltado ao ensino da Arquitetura MIPS, destacando os estágios do ciclo de instrução e os sinais de controle envolvidos em cada etapa. A Figura 14 apresenta a interface principal do MIPS-Datapath. A área superior esquerda exibe o menu para a seleção do código MIPS a ser executado. Na parte central, o *Datapath* é mostrado com sua Unidade de Controle, demonstrando o *Datapath* completo e seus blocos funcionais, e durante a simulação, o caminho muda de cor por onde os dados deveriam passar entre os componentes.

Figura 14 – Interface do Simulador MIPS-DATAPATH

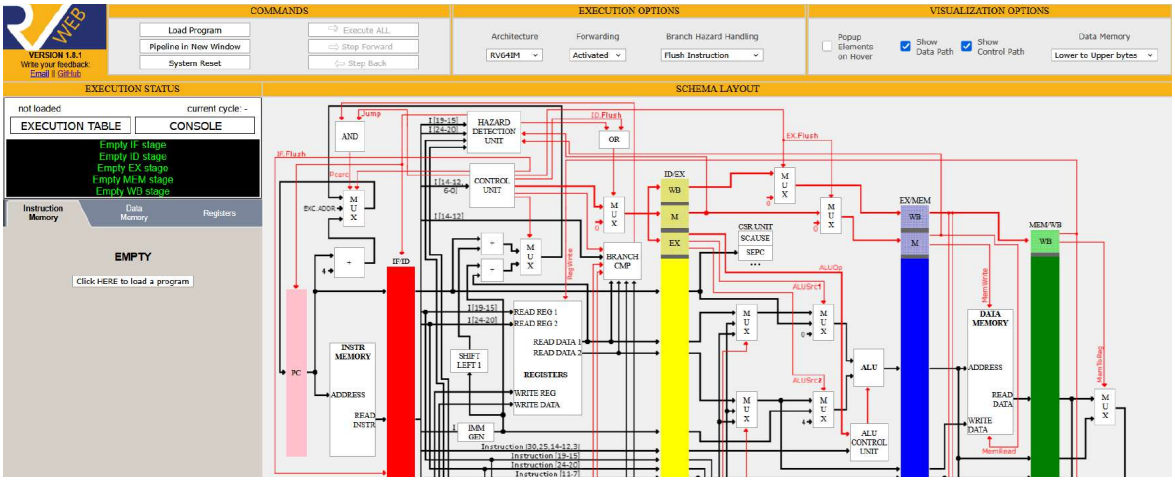


Fonte: (MIPS Datapath Dev Team, 2015)

Entre os pontos positivos do simulador MIPS Datapath, destaca-se a clareza visual, que facilita significativamente o entendimento do funcionamento interno do processador, especialmente no que se refere ao controle e à execução das instruções. A representação gráfica dos Sinais de Controle permite que o usuário acompanhe de forma intuitiva como cada etapa da instrução percorre o *Datapath*. No entanto, o simulador apresenta algumas limitações, sendo a principal delas o suporte restrito a um conjunto fixo de instruções básicas predefinidas, ou seja, ele não suporta a ISA completa do MIPS.

O Simulador WebRISC-V (WebRISC-V Project, 2022) é um simulador Web que permite visualizar graficamente a execução de instruções da Arquitetura RISC-V, evidenciando o fluxo de dados conforme ele ocorre. Criado a partir de um esforço colaborativo, a ferramenta busca apoiar o ensino de Arquitetura de Computadores por meio de uma interface visual. A Figura 15 apresenta a interface do WebRISC-V. Na parte esquerda, é possível acessar o menu do editor ou carregar um código RISC-V. Na parte superior, há os botões de controle para execução. Na parte central, o *Datapath* interativo mostra a passagem de dados entre os blocos, sendo necessário interagir com os blocos para visualizar os dados, enquanto na esquerda são exibidos os estados dos Registradores e da Memória.

Figura 15 – Interface do Simulador WebRISC-V



Fonte: (WebRISC-V Project, 2022)

Entre os aspectos positivos do Simulador WebRISC-V, destaca-se a possibilidade de acesso direto por Web, sem a necessidade de instalação de um programa, facilitando o uso em diferentes dispositivos e ambientes de estudo. Além disso, a visualização gráfica em tempo real das operações realizadas pelo processador contribui para uma melhor compreensão do fluxo de dados e da execução das instruções. Por outro lado, a interface do simulador pode ser considerada complexa, exigindo um tempo maior de familiarização por parte dos usuários, o que pode representar dificuldade adicional no início da aprendizagem.

2.2.3 Considerações Simuladores

Esta Subseção apresenta uma comparação entre os simuladores discutidos. Para facilitar a análise, foi elaborada a Tabela 1 que resume as principais características analisadas. A coluna ISA Alvo indica qual Arquitetura o simulador é capaz de interpretar e executar. A coluna entrada refere-se ao tipo de conteúdo que pode ser fornecido ao simulador, como códigos completos, instruções isoladas ou formatos limitados. Módulos visuais descreve os elementos da interface que são apresentados de forma gráfica ao usuário, como estruturas internas e representações de Memória ou Registradores. A presença de *Datapath* mostra se há uma visualização gráfica dos componentes internos da CPU e das conexões entre eles. Fluxo indica se o simulador permite acompanhar em tempo real a movimentação dos dados e Sinais de Controle entre os blocos. A coluna Ciclo de Clock representa a capacidade de simular a execução das instruções dividida em ciclos, permitindo observar o avanço interno passo a passo. Já Etapa BDE informa se o simulador separa e exibe claramente as três fases principais do processamento de instruções: Busca, Decodificação e Execução. Plataforma identifica o ambiente em que o simulador pode ser utilizado, seja via Web ou por meio de instalação local.

Ao analisar os simuladores sem *Datapath* visível, podemos observar alguns pontos em comum, como o foco principal na execução de programas em *Assembly* e o acompanhamento do conteúdo de Registradores e Memória. Em geral, esses simuladores apresentam interfaces mais simples, com menos recursos gráficos, o que pode facilitar o uso. Contudo, eles diferem quanto ao nível de detalhamento do fluxo de dados e na possibilidade ou não de execução passo a passo. Os simuladores mais detalhados são o Simple 8-bit Assembler Simulator e o Venus, pois ambos destacam o fluxo das instruções e permitem a execução por Ciclo de Clock, mesmo sem mostrar o *Datapath* graficamente. Já os menos detalhados são o Onecompiler e o Emulador Assembly x86, que se limitam a mostrar apenas o editor de código e, em alguns casos, os Registradores e a Memória, sem oferecer visão clara do funcionamento interno nem execução por etapas. Nenhum desses simuladores apresenta as etapas do ciclo de execução de forma explícita, reduzindo a possibilidade de análise detalhada do funcionamento do processador.

Tabela 1 – Comparativo entre os Simuladores Estudados

	RIPES	MIPS 101	MIPS-DPath	WebRISC-V	Simple 8-bit	Venus	Onecompiler	Emul. ASM x86	Proposta
Ano	2018	2018	2015	2022	2014	2019	2019	2014	2025
ISA Alvo	RISC-V	MIPS	MIPS	RISC-V	-	RISC-V	x86	x86	ISA Própria
Tipo Entrada	Programa Assembly	Instrução Indvidual	Programas pré-definidos	Programa Assembly	Programa Assembly	Programa Assembly	Programa Assembly	Programa Assembly	Programa Assembly
Módulos Visuais	Assembly Mem. REGs	L.Máquina REGs	Assembly Memória	L.Máquina Mem. REGs	Assembly Mem. REGs	Assembly Mem. REGs	Assembly E/S	Assembly Mem. REGs	Assembly Mem. REGs
Datapath	Sim	Sim	Sim	Sim	Não	Não	Não	Não	Sim
Ciclo Clock	Sim	Sim	Sim	Sim	Sim	Sim	Não	Não	Sim
Etapa BDE	Sim	Não	Não	Sim	Sim	Sim	Não	Sim	Sim
Plataforma	WEB	WEB	Win, Linux	WEB	WEB	WEB	WEB	WEB	WEB
OpenCode	Sim	Não	Sim	Sim	Sim	Não	Não	Não	Sim

Já entre os simuladores com visualização de *Datapath*, os pontos em comum incluem a capacidade de representar graficamente os blocos internos da CPU e o fluxo de dados durante a execução. Essa característica torna esse tipo de ferramenta especialmente útil para disciplinas que exigem a compreensão da organização interna da CPU. Por outro lado, as diferenças ficam evidentes na qualidade e nível de detalhamento da visualização, no suporte a diferentes formatos de instrução e no grau de interatividade com o usuário. Os simuladores RIPES e o WebRISC-V se destacam como os mais completos, pois além de apresentarem *Datapath* visível, também permitem visualizar o fluxo de controle, suportam execução em Ciclos de Clock e oferecem diversos módulos interativos, como editores de código, visualização de Registradores, Memória e Linguagem de Máquina. Ambos também são acessíveis via Web e possuem código aberto, facilitando seu uso e customização.

Por outro lado, o MIPS-Datapath, embora também tenha visualização do *Datapath* e do fluxo, é mais limitado, já que não permite execução por Ciclos de Clock, não exibe as etapas de Busca, Decodificação e Execução separadamente e depende de códigos pré-definidos como entrada, restringindo a flexibilidade do usuário. Ele também não está disponível como ferramenta Web, sendo necessário instalá-lo localmente. Já o simulador MIPS 101 ocupa uma posição intermediária, com uma interface simples, execução por etapas e visualização clara do *Datapath*, mas com foco em instruções isoladas, o que pode limitar o entendimento de fluxos mais complexos. Seu código não é aberto, o que restringe customizações ou integrações com outras ferramentas.

Com base na análise dos simuladores existentes, foi possível identificar algumas lacunas e limitações que justificam a elaboração do Trabalho Proposto, apresentado na última coluna da Tabela 1. O trabalho busca unir as principais vantagens encontradas nas ferramentas analisadas, como a visualização detalhada do *Datapath*, a execução passo a passo, e a interface amigável, mas também pretende superar algumas das limitações observadas, como a ausência de recursos de análise mais avançada e uma interface mais limpa. O objetivo é oferecer uma solução que permita aos discentes entenderem, de forma integrada, tanto a Arquitetura quanto a Organização de uma CPU. Nos próximos Capítulos, será apresentado o Trabalho Proposto.

3 Arquitetura Alvo

Compreender a Arquitetura Alvo é fundamental para acompanhar o desenvolvimento do trabalho, pois ela serve como a base para todas as fases do projeto. Este Capítulo descreve a concepção da Arquitetura, que foi criada do zero, desde a definição da ISA até a Organização interna da CPU. Este caminho foi tomado porque as ISAs usadas nos simuladores estudados não atendem aos requisitos de simplicidade e adequação didática necessários. Na seção 3.1, será apresentada a Arquitetura do Conjunto de Instruções, abordando o processo de definição das instruções e suas principais características. Na seção 3.2, será detalhada a Organização da CPU, mostrando como foi estruturado o Caminho de Dados necessário para suportar a execução das instruções propostas. Por fim, na seção 3.3, será apresentado o Formato das Instruções, explicando como as instruções foram codificadas, visando simplicidade na decodificação e implementação da CPU.

3.1 Arquitetura do Conjunto de Instruções

Esta Seção visa apresentar a definição da ISA desenvolvida para este trabalho. O foco está na criação de um conjunto de instruções simples, porém funcional, que permita a implementação de programas básicos para testes e validação do processador. Serão descritas as categorias de instruções, suas funcionalidades e os critérios usados definir ISA, tornando a mesma adequada ao propósito didático da proposta.

3.1.1 Características Gerais

Antes do início do desenvolvimento deste trabalho, algumas decisões arquiteturais foram tomadas visando garantir que a Arquitetura fosse adequada ao seu propósito principal: o ensino de conceitos básicos de Arquitetura de Computadores e Caminho de Dados. Por esse motivo, ficou definido que a Arquitetura deveria ser simples e de fácil compreensão, permitindo que os discentes pudessem acompanhar todas as etapas de execução das instruções sem dificuldades.

Além disso, foi decidido que o desempenho da CPU não seria uma preocupação central. Em vez de buscar otimizações que reduzissem o número de ciclos ou o custo de Hardware, optou-se por soluções que facilitassem o entendimento. Por exemplo, a Arquitetura admite a existência de módulos redundantes, como unidades dedicadas somente ao incremento do Contador de Programa (`PC`), mesmo que, em soluções comerciais, tais funções sejam integradas ou otimizadas.

Outra decisão importante foi a de não seguir estritamente os paradigmas clássicos de Arquitetura de conjunto de instruções, como CISC ou RISC. O objetivo não é comparar estilos arquiteturais, mas sim proporcionar uma plataforma simples e controlada para o ensino de conceitos como Ciclo de Execução, decodificação de instruções, acesso à Memória e funcionamento do Caminho de Dados.

Quanto às características específicas da CPU proposta, optou-se por utilizar um número reduzido de Registradores, suficiente para suportar os programas de teste e os conceitos abordados em aula. Essa escolha visa diminuir a complexidade do Banco de Registradores e simplificar a lógica da Unidade de Controle. O tamanho das Memórias também é reduzido, já que a nossa Arquitetura não visa a execução de programas de grande porte. Pequenas quantidades de Memória para instruções e dados são suficientes para nosso propósito didático.

Da mesma forma, o conjunto de instruções foi limitado a um número pequeno de operações, escolhidas de forma a cobrir os principais tipos de instruções (como operações aritméticas, lógicas, de controle e de acesso à memória), mas sem excessos que pudessem tornar o Caminho de Dados desnecessariamente complexo. Os formatos das instruções também foram mantidos em quantidade mínima. Adotar poucos formatos facilita a etapa de decodificação e simplifica o Hardware necessário para executar cada instrução.

Por fim, definiu-se que todas as operações de entrada e saída seriam realizadas exclusivamente por meio da Memória. Embora essa decisão não represente a forma como a maioria das Arquiteturas comerciais trata E/S (Entrada e Saída), ela simplifica o projeto, evitando a necessidade de criar Barramentos ou módulos adicionais de comunicação com Periféricos, mantendo o foco na aprendizagem do funcionamento básico do processador.

3.1.2 Conjunto de Instruções

O conjunto de instruções definido para a Arquitetura Alvo foi projetado para a implementação de algoritmos básicos e a demonstração clara dos principais conceitos de execução em um processador. Para isso, as instruções foram organizadas em cinco grupos principais, são eles: Movimentação de Dados, Operações Aritméticas. Operações Lógicas, Controle de Fluxo Condicional e Controle de Fluxo Incondicional. A Tabela 2 apresenta um resumo das instruções que compõem a ISA proposta.

As Instruções de Movimentação de Dados são responsáveis por transferir informações entre Registradores e posições de Memória. Três instruções básicas foram definidas para essa finalidade. A instrução `LDA` permite carregar o conteúdo de uma posição de Memória diretamente em um Registrador, enquanto `LDAI` insere um valor imediato no Registrador, sem acessar a Memória. Já a instrução STA realiza a operação inversa, armazenando o conteúdo de um Registrador em uma posição de Memória específica.

As Instruções Aritméticas realizam operações matemáticas básicas entre os valores armazenados nos Registradores. Para manter a ISA simples, foram incluídas apenas duas operações: `ADD`, que realiza a soma de dois operandos, e `SUB`, que executa a subtração entre eles. Ambas armazenam o resultado em um terceiro Registrador. Essas instruções afetam *flags* internas que indicam características do resultado, como se ele foi Zero, Negativo ou se ocorreu um Estouro (*Overflow*).

As Instruções Lógicas têm como objetivo realizar operações bit a bit entre os Registradores. Três operações foram incluídas: `AND` que aplica a operação lógica E entre dois operandos; `OR` que aplica a operação lógica OU; e `NOT` que executa a negação bit a bit sobre um único operando. Essas instruções são úteis não apenas para manipulação de dados, mas também para a construção de condições e controle de decisões no programa.

As Instruções de Controle de Fluxo Condicional permitem que o programa altere seu caminho de execução com base em determinadas situações. Foram implementadas duas instruções voltadas para esse propósito: `JZ` que realiza um salto se a *flag* de Zero estiver ativada (indicando que o resultado anterior foi zero); e `JN` que desvia o fluxo se a *flag* de Negativo estiver ativa. Essas instruções são fundamentais para a construção de estruturas de decisão, como condicionais e laços controlados por resultado.

Por fim, as Instruções de Controle de Fluxo Incondicional permitem modificar o caminho de execução do programa independentemente de qualquer condição. A instrução `JMP` executa um salto direto para uma posição de Memória especificada, sendo amplamente utilizada para *loops* e saltos fixos. Já a instrução `HLT` encerra completamente a execução do programa. Ambas são essenciais para o controle geral do programa e a delimitação de sua estrutura lógica. A Tabela 2 abaixo resume Instruções que compõem a ISA Alvo.

Tabela 2 – Resumo das Instruções da ISA Proposta

`LDA`	Carrega o conteúdo de uma posição de memória em um registrador
`LDAI`	Carrega um valor imediato diretamente em um registrador
`STA`	Armazena o conteúdo de um registrador em uma posição de memória
`ADD`	Soma dois valores de registradores e armazena o resultado em um terceiro
`SUB`	Subtrai dois valores de registradores e armazena o resultado em um terceiro
`AND`	Aplica a operação lógica E entre dois registradores
`OR`	Aplica a operação lógica OU entre dois registradores
`NOT`	Aplica a negação bit a bit em um registrador
`JZ`	Salta para um endereço se a flag de zero estiver ativada
`JN`	Salta para um endereço se a flag de negativo estiver ativada
`JMP`	Executa salto incondicional para um endereço de memória
`HLT`	Encerra a execução do programa

A escolha por manter um número reduzido de instruções em cada tipo foi intencional, com o objetivo de preservar a simplicidade da ISA e facilitar o processo de aprendizagem. Embora haja apenas duas operações aritméticas, elas já são suficientes para a implementação de algoritmos diversos, já que a soma e a subtração servem como base para outras operações matemáticas. O mesmo critério foi aplicado às instruções lógicas, limitadas a três operações fundamentais que, em conjunto com as aritméticas, permitem a construção de qualquer lógica de controle necessária.

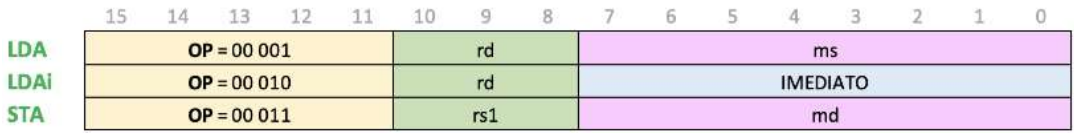
Quanto aos saltos condicionais, a decisão de baseá-los exclusivamente no estado das *flags* foi uma escolha consciente para manter o controle de fluxo simples e direto. Por isso, instruções comuns em outras ISAs, como `BEQ` (*Branch if Equal*) e `BNE` (*Branch if Not Equal*) não foram incluídas. No entanto, essa decisão não prejudica a programação, já que qualquer tipo de salto condicional pode ser implementado por meio da combinação adequada de instruções aritméticas seguidas do teste de *flags* de condição.

3.1.3 Formatos de Instruções

Nesta Seção, serão apresentados os formatos definidos para cada um dos tipos de instruções da ISA proposta. Todas as instruções possuem o mesmo Tamanho de Palavra, 16 bits, uma decisão de projeto adotada para facilitar o processo de decodificação e simplificar seu circuito. As instruções serão apresentadas agrupadas por tipo.

Como pode-se observar na Figura Figura 16, as Instruções de Movimentação de Dados seguem um formato padronizado de 16 bits, composto por três campos principais: o campo Opcode `OP` (com 5 bits) identifica a operação a ser realizada; o campo do registrador (com 3 bits) representa um endereço de registrador e o campo de endereço de memória ou valor imediato (com 8 bits). A instrução `LDA` utiliza o formato `00001 rd ms`, onde o conteúdo de uma posição de Memória é carregado para um Registrador. A instrução `LDAI`, no formato `00010 rd imediato`, permite carregar um valor imediato no Registrador. Já a instrução `STA`, no formato `00011 rs1 md`, armazena o conteúdo de um Registrador em uma posição específica da Memória. O uso de 8 bits para o endereço ou imediato garante flexibilidade tanto para o endereçamento completo da Memória quanto para o carregamento de valores.

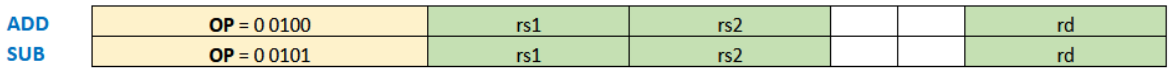
Figura 16 – Formato para Instruções de Movimentação de Dados



Fonte: Elaborada pelo autor

As Instruções Aritméticas também seguem uma estrutura de 16 bits, organizada de forma a permitir operações entre Registradores. Como pode ser visto nesta Figura 17, o formato utilizado é composto por um campo Opcode de 5 bits, dois registradores fonte (`rs1` e `rs2`, com 3 bits cada) e um registrador de destino (`rd`, também com 3 bits). Esse formato é aplicado a instruções como `ADD` (`01000 rs1 rs2 rd`) e `SUB` (`01001 rs1 rs2 rd`), permitindo a realização de operações matemáticas entre dois operandos e o armazenamento do resultado em um terceiro registrador.

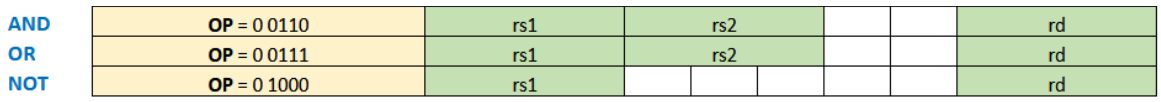
Figura 17 – Formato para Instruções Aritméticas



Fonte: Elaborada pelo autor

As Instruções Lógicas compartilham o mesmo formato das aritméticas. Como pode ser visto na Figura 18, instruções como `AND` e `OR` utilizam dois registradores fonte e um de destino, com os respectivos Opcodes `01010` e `01011`. Já a instrução `NOT`, por ser unária, utiliza o formato `01100 rs1 rd`, onde o valor de um registrador é invertido bit a bit e o resultado armazenado em outro registrador.

Figura 18 – Formato para Instruções Lógicas

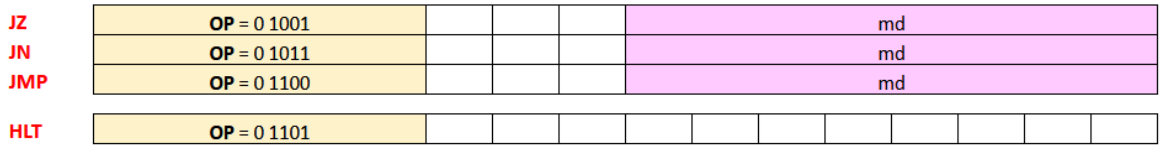


Fonte: Elaborada pelo autor

As Instruções de Saltos Condicionais são responsáveis por alterar o fluxo de execução com base no estado das *flags* definidas por operações anteriores. Conforme a Figura 19, essas instruções são compostas por um Opcode de 5 bits, seguido por um campo de 8 bits, representando o endereço de destino de memória (`md`) do salto. Instruções como `JZ` (`10000 md`) e `JN` (`10010 md`) verificam, respectivamente, as *flags* de Zero e Negativo, realizando o desvio apenas quando estiverem ativas. O uso de 8 bits para o endereço garante cobertura de todo espaço de Memória, permitindo saltos para qualquer ponto do programa.

As Instruções de Saltos Incondicionais utilizam o mesmo formato das condicionais, com um Opcode seguido por um campo de endereço de 8 bits, conforme a Figura 19. A instrução `JMP`, codificada como `10011 md`, realiza o salto para um endereço especificado sem verificar nenhuma condição. A instrução `HLT`, por sua vez, representa a finalização da execução do programa. Ela é composta apenas pelo Opcode `10100`.

Figura 19 – Formato para Instruções de Controle de Fluxo



Fonte: Elaborada pelo autor

A escolha dos 5 bits para o campo de Opcode garante a possibilidade de representar até 32 diferentes instruções ($2^5 = 32$), o que é mais do que o número de instruções atualmente implementadas. Essa decisão foi tomada de forma intencional para permitir futuras expansões da ISA, caso seja necessário incluir novas instruções.

O número de bits dos campos de registradores foi definido com base no tamanho do Banco de Registradores usado. Como a ISA disponibiliza 8 registradores, são necessários 3 bits para endereçar cada registrador ($2^3 = 8$). Já o tamanho dos campos de endereçamento de Memória foi estabelecido de acordo com a quantidade de posições disponíveis, que é de 256 palavras, justificando o uso de 8 bits ($2^8 = 256$).

Por fim, é importante destacar que não foram incluídas instruções de ponto flutuante, uma vez que o objetivo deste trabalho não é a implementação de uma ULA completa ou o tratamento de operações matemáticas complexas. O foco permanece na apresentação didática de como projetar uma CPU, priorizando o entendimento dos conceitos de Arquitetura, Organização e execução de instruções.

3.1.4 Linguagem de Montagem

Nesta Seção serão apresentadas as instruções da ISA considerando sua representação em *Assembly*, ou Linguagem de Montagem. A definição da linguagem visa permitir que programadores possam escrever programas de forma legível e compreensível, utilizando mnemônicos em vez de códigos binários ou hexadecimais. A Figura 20 apresenta todas as instruções disponíveis na Arquitetura Alvo, agora no formato *Assembly*. Nela é possível visualizar os mnemônicos utilizados para cada operação, bem como a estrutura das instruções na linguagem de montagem.

A sintaxe geral da instrução `LDA` é `LDA M, RD`, onde `M` representa o endereço de memória a ser lido e `RD` é o registrador de destino que receberá o dado. Para valores imediatos, utiliza-se a instrução `LDAI N, RD`, onde `N` é um valor constante, um Número a ser carregado no registrador `RD`. Já para armazenar valores na memória, utiliza-se a instrução `STA RS, M`, onde `RS` é o registrador de origem e `M` é a posição destino de memória.

Figura 20 – Formato das Instruções em Linguagem de Montagem

Fonte: Elaborada pelo autor

As instruções Aritméticas e Lógicas utilizam uma estrutura baseada em três operandos para as operações binárias e dois operandos para operações unárias. A sintaxe geral é `OP RS1, RS2, RD`, onde `OP` pode ser `ADD`, `SUB`, `AND` ou `OR`; `RS1` e `RS2` são os registradores que fornecem os operandos de entrada, e `RD` é o registrador onde o resultado será armazenado. Para a operação `NOT`, utiliza-se a forma `NOT RS, RD`, onde o conteúdo de `RS` é invertido bit a bit e armazenado em `RD`.

A instrução `JZ M` realiza um salto para o endereço `M` caso a *flag* de Zero esteja ativada. De forma semelhante, a instrução `JN M` salta se a *flag* de Negativo estiver ativa. Já a instrução `JMP M` sempre desvia a execução para o endereço `M`. A instrução `HLT` representa a finalização do programa. Ela não requer operandos e é utilizada para delimitar o fim de um programa ou rotina.

Para facilitar a programação e a leitura do código, é permitido o uso de *Labels* (Rótulos), que funcionam como identificadores simbólicos associados a endereços de instrução. Como pode-se observar no exemplo da Figura 21, `INICIO` e `FIM` são *Labels* que marcam posições específicas no código. O programador pode referenciar esses pontos de forma simbólica, melhorando a legibilidade e facilita a manutenção do código. O uso de *Labels* elimina a necessidade de calcular manualmente os endereços dos saltos.

Figura 21 – Uso de *Labels* em Linguagem de Montagem

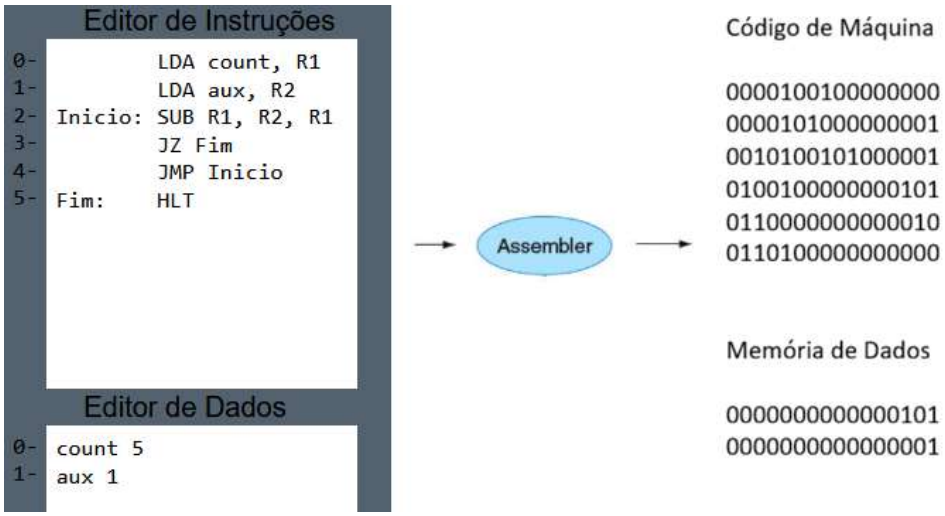
```
INICIO:
    LDA 10, R1
    ADD R1, R2, R1
    JZ FIM
    JMP INICIO
FIM:
    STA R1, 20
```

Fonte: Elaborada pelo autor

Os valores utilizados nas instruções são aceitos apenas em formato Decimal. É possível declarar tanto valores Positivos quanto Negativos, os quais serão armazenados internamente em Complemento de 2. Além disso, *Labels* podem ser utilizados para nomear variáveis, facilitando sua referência ao longo do código.

O processo de montagem do programa será realizado por um Montador, responsável por converter o código *Assembly* para a Linguagem de Máquina correspondente. Esse processo inclui a tradução dos mnemônicos para seus respectivos códigos binários, a substituição dos *Labels* por endereços reais, e a produção final do código binário que será executado pela CPU. Como pode ser visto nesta Figura 22, o Montador desempenha um papel essencial para transformar o código simbólico em instruções operacionais. Cabe destacar que este trabalho contempla a implementação do montador para a ISA Alvo.

Figura 22 – Conversão de *Assembly* para LM na ISA Alvo



Fonte: Elaborada pelo autor

3.2 Caminho de Dados Monociclo

O objetivo desta Seção é apresentar o Caminho de Dados Monociclo, a primeira Organização de CPU implementada neste trabalho. Nesta abordagem, cada instrução é executada em um único Ciclo de Clock, o que implica que todas as operações necessárias devem ser realizadas simultaneamente dentro deste ciclo. Serão discutidos os detalhes do projeto deste Caminho de Dados, desde os componentes utilizados até as conexões internas. Além disso, o funcionamento de cada etapa de execução será apresentado.

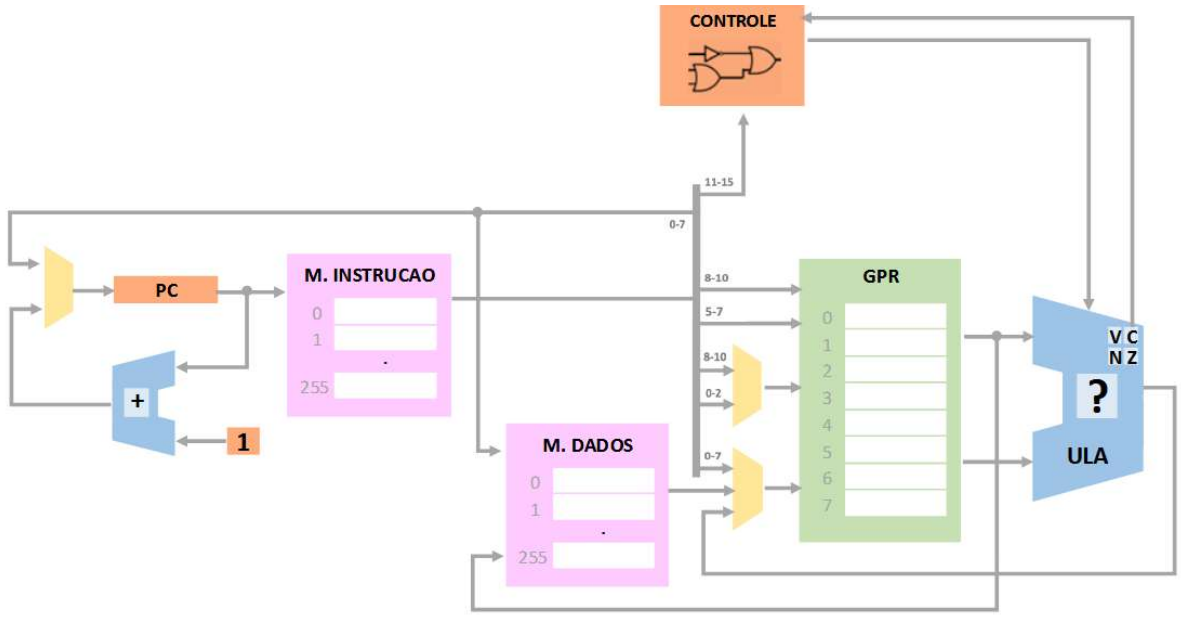
3.2.1 Componentes e Conexões

A Figura 23 apresenta o Caminho de Dados Monociclo Completo projetado para a execução das instruções da ISA desenvolvida para o trabalho. A partir da análise de cada um dos formatos de instrução definidos anteriormente, foi possível identificar os blocos necessários para implementar a execução de todas as operações. Cada componente tem um papel específico dentro do ciclo de execução, conforme discutido a seguir.

No processamento de qualquer instrução, tudo se inicia pelo Contador de Programa (`PC`). Este registrador de controle é responsável por apontar para o endereço de memória da próxima instrução a ser executada. Sua entrada pode ser o valor de incremento normal (para a próxima instrução) ou um novo endereço (no caso de saltos). Para a CPU, ele é um ponteiro que regra o andamento da execução do programa.

A Memória de Instruções é responsável por armazenar todas as instruções do programa que será executado. Sua entrada principal é o endereço vindo do `PC`, e sua saída é a instrução lida, que será utilizada pela Unidade de Controle e demais componentes. Além dos sinais de controle que não aparecem, a Memória de Instruções conta com uma Entrada de Endereços de 8 Bits, e sua Saída de Instrução de 16 bits.

Figura 23 – Caminho de Dados Monociclo para a ISA Alvo



Fonte: Elaborada pelo autor

A Memória de Dados armazena as informações utilizadas e manipuladas durante a execução dos programas. Ela é acessada principalmente pelas instruções de *LOAD* e *STORE*, recebendo como entrada o endereço de memória e os dados (no caso de escrita), e gerando como saída os dados lidos (no caso de leitura). A Memória de Dados tem uma

Entrada de Endereço de 8 Bits acima, uma Entrada de Dados de 16 Bit abaixo e uma Saída de Dados de 16 Bits, sem considerar os controles de Escrita e Leitura.

O Banco de Registradores (GPR) armazena os dados temporários usados nas operações. Ele possui entradas para os endereços dos registradores fonte e destino, bem como para o dado a ser escrito, e saídas que fornecem os operandos para a ULA ou outros blocos, dependendo da instrução em execução. O Banco de Registradores tem duas Entradas de Endereço de Escrita de 3 Bits logo acima, e no outro lado as duas Saídas de Dados de 16 Bits. Além disso, conta com uma porta de escrita composta por uma terceira Entrada de Endereço e uma Entrada de Dados, de 3 e 16 bits respectivamente.

A ULA realiza as operações aritméticas e lógicas necessárias. Recebe como entrada dois operandos vindos do Banco de Registradores e gera como saída o resultado da operação, além de sinalizar as condições que afetam as *Flags*. O Bloco de *Flags* armazena os sinais de condição resultantes das operações da ULA, como a *Flag* de Zero (`Z`), utilizado nas decisões de salto condicional. A ULA possui duas Entradas de Dados de 16 Bits, e uma Saída de 16 Bits. Também conta com uma Entrada de Sinal de Controle responsável por selecionar a operação executada, além de uma Saída de *Flag* utilizada para sinalizar condições específicas do resultado gerado.

A Unidade de Controle interpreta os campos da instrução e gera os Sinais de Controle necessários para coordenar o funcionamento de todos os demais componentes, como sinais de leitura e escrita, seleção de multiplexadores, operação da ULA, entre outros. A Unidade de Controle contém sete Saídas de Sinais de Controle que coordenam o funcionamento dos demais blocos do processador, bem como uma Entrada de `Flag` proveniente da ULA, permitindo ajustar o comportamento da execução conforme o estado dos resultados aritméticos e lógicos.

As conexões entre os componentes foram definidas a partir da análise dos operandos e campos de cada formato de instrução. Além disso, obviamente foram consideradas as interfaces dos Blocos utilizados. Cada conexão foi planejada para garantir que todos os dados necessários possam ser encaminhados ao longo do Caminho de Dados de forma sincronizada durante o único Ciclo de Clock de cada instrução.

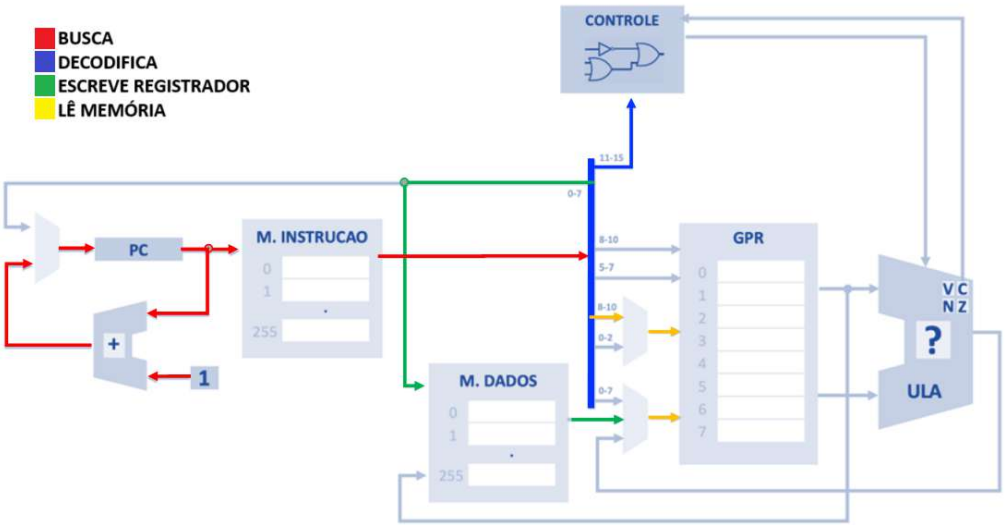
Por fim, alguns Multiplexadores são utilizados em pontos do caminho de dados. A utilização de multiplexadores foi necessária em pontos onde havia mais de uma possível fonte para um sinal de entrada de um bloco. Por exemplo, a seleção entre o valor `PC` vindo do registrador ou da memória de instrução é feita por um multiplexador. A explicação sobre o papel de cada multiplexador será tratada na próxima seção.

3.2.2 Fluxos de Execução

Aqui será apresentado o Fluxo de Execução para cada um dos Tipos de Instruções definidos na ISA. Cada fluxo será descrito com base no Caminho de Dados Monociclo, destacando quais blocos são ativados em cada caso. É importante destacar que, embora as figuras mostrem as etapas separadas (como Busca, Decodificação, Execução, Memória e Escrita de Resultado), no Caminho de Dados Monociclo todas essas ações acontecem simultaneamente em um único Ciclo de Clock. A separação por etapas serve somente como recurso para facilitar o entendimento.

A Figura 24 apresenta o fluxo de execução da instrução `LDA`, que carrega um valor da Memória de Dados para um registrador. Inicialmente, ocorre a Busca da instrução na Memória de Instruções com base no valor do `PC`. Em seguida, a Decodificação é realizada pela Unidade de Controle, que interpreta o Opcode e gera os sinais adequados. O campo de endereço da instrução é encaminhado para a Memória de Dados, que retorna o valor armazenado na posição especificada. Esse valor é então encaminhado para o Banco de Registradores, onde é escrito no registrador de destino.

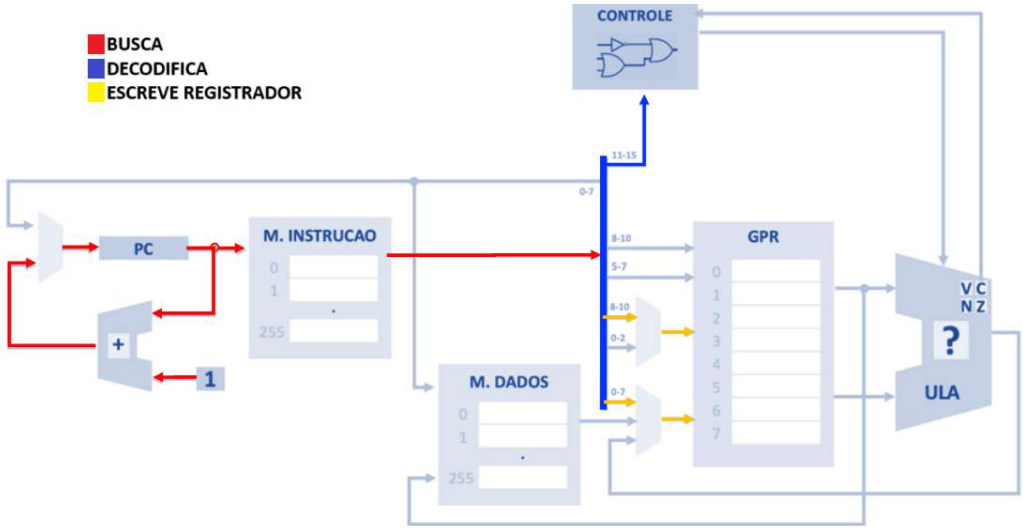
Figura 24 – Fluxo de Execução de uma Instrução `LDA`



Fonte: Elaborada pelo autor

A Figura 25 apresenta o fluxo da instrução `LDAI`, que carrega um valor imediato diretamente em um registrador, sem acessar a Memória de Dados. Como nas demais instruções, ocorre primeiro a Busca e a Decodificação. No entanto, neste caso, o dado a ser carregado está embutido na própria instrução, ou seja, vem diretamente do campo imediato. Esse valor é enviado ao Banco de Registradores e armazenado no registrador de destino. Comparando as Figuras 24 e 25 pode-se perceber que a fonte do dado a ser escrito nos Registradores é diferente no `MUX` inferior.

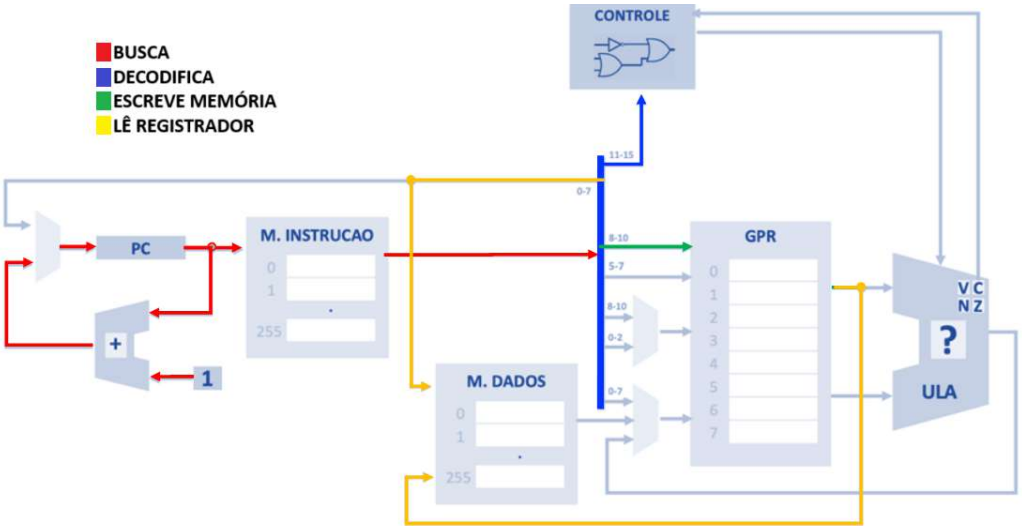
Figura 25 – Fluxo de Execução de uma Instrução `LDAi`



Fonte: Elaborada pelo autor

A Figura 26 mostra o fluxo da instrução `STA`, que armazena o valor de um registrador na Memória de Dados. Após a Busca e a Decodificação, o conteúdo do registrador de origem é lido do Banco de Registradores e enviado à Memória de Dados. A escrita ocorre na posição de memória especificada pelo campo de endereço da instrução. O que distingue esse fluxo é a escrita na memória, ao contrário de `LDA` e `LDAI` que escrevem em registradores.

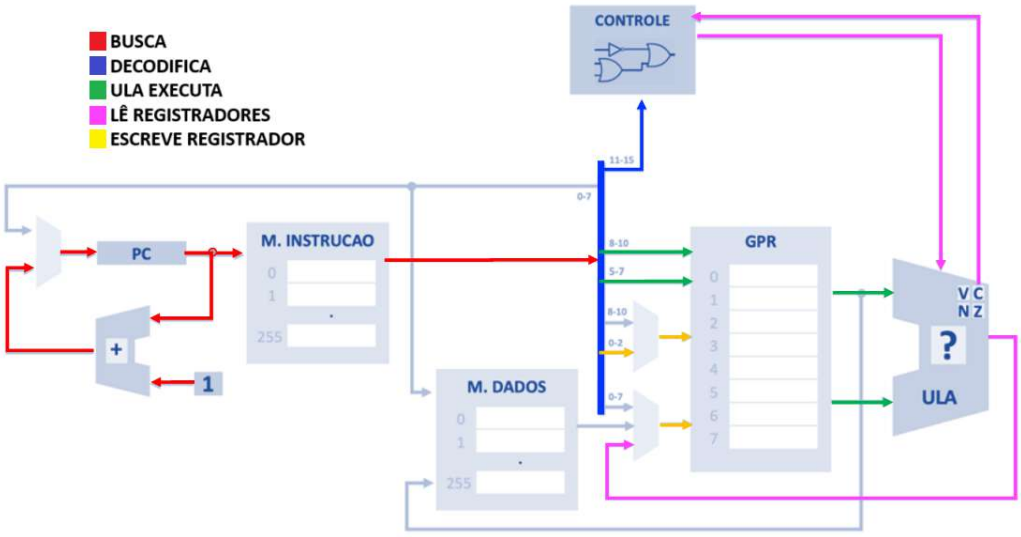
Figura 26 – Fluxo de Execução de uma Instrução `STA`



Fonte: Elaborada pelo autor

A Figura 27 mostra o fluxo de execução das instruções aritméticas e lógicas. Após a Busca e Decodificação, dois registradores são lidos como operandos (exceto `NOT`, que usa apenas um). A ULA realiza a operação indicada pela instrução, e o resultado é armazenado no registrador de destino. O que diferencia esse fluxo é a ausência total de acesso à Memória de Dados, concentrando-se exclusivamente no Banco de Registradores e na ULA.

Figura 27 – Fluxo de Execução de uma Instrução que usa ULA

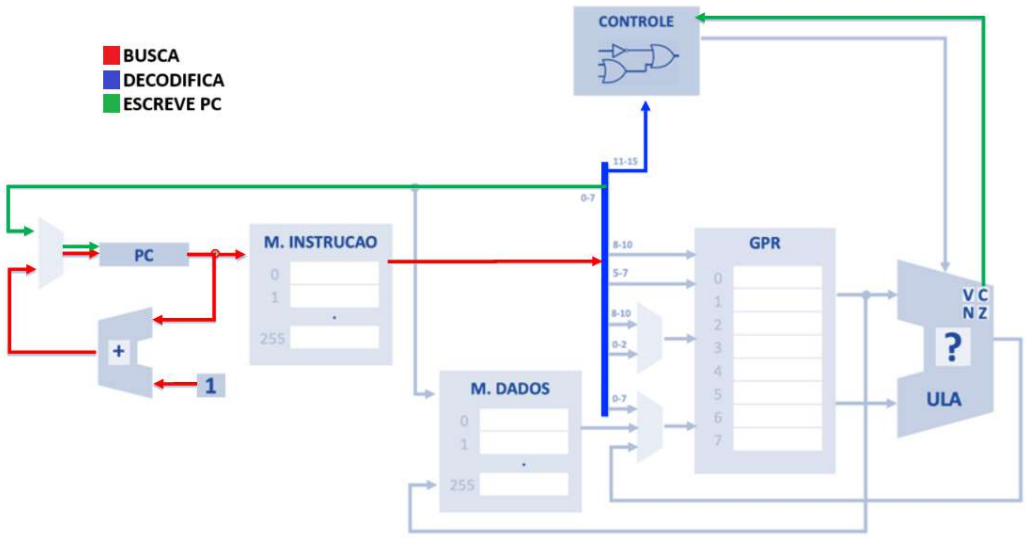


Fonte: Elaborada pelo autor

A Figura 28 representa o fluxo das instruções de salto condicional e incondicional, como `JZ`, `JN` e `JMP`. Após a Busca e Decodificação, a Unidade de Controle verifica a `Flag` correspondente (`Zero` ou `Negativo`) armazenada no Bloco de *Flags* no caso condicional, atualizado pela última operação da ULA. Se a condição for verdadeira, o endereço da instrução é carregado no `PC`, redirecionando o fluxo do programa. Caso contrário, o `PC` é apenas incrementado normalmente. A distinção aqui é que não há escrita em registradores ou acesso à memória de dados a única ação concreta é a atualização condicional do `PC`, com base em uma verificação lógica.

No caso incondicional depois da Busca e Decodificação, o endereço de destino especificado na instrução é diretamente carregado no `PC`. Não há qualquer verificação lógica envolvida, e o salto ocorre sempre. Isso faz com que `JMP` seja o fluxo mais direto entre as instruções de controle.

Figura 28 – Fluxo de Execução de uma Instrução de SALTO



Fonte: Elaborada pelo autor

3.2.3 Sinais de Controle

Nesta Subseção, serão descritos os Sinais de Controle necessários para o funcionamento do Caminho de Dados Monociclo. O Controle Monociclo é baseado em lógica combinacional, que a partir do Opcode de cada instrução gera todos os sinais necessários para ativar somente os blocos e caminhos relacionados à execução daquela instrução. É importante destacar que, nesta etapa do projeto, não será detalhado o circuito interno da Unidade de Controle. O foco aqui é entender quais sinais precisam ser gerados e qual o efeito de cada um deles durante a execução das instruções. A Figura 29 apresenta os Sinais de Controle para o Caminho de Dados Monociclo, onde:

- `wrReg`: Controla a escrita no Banco de Registradores. É ativado nas instruções que precisam armazenar um valor no registrador de destino, como `LDA`, `LDAI`, `ADD`, `SUB`, `AND`, `OR` e `NOT`. Permanece desativado nas instruções `STA`, `JZ` e `JMP`.

- `rdMem`: Controla a leitura da Memória de Dados. É ativado apenas na instrução `LDA`, pois ela exige que um valor seja carregado da memória para um registrador. Nas demais instruções, este sinal permanece desativado.

- `wrMem`: Controla a escrita na Memória de Dados. É ativado apenas na instrução `STA`, que armazena o conteúdo de um registrador em uma posição da memória.

- `muxPC`: Define a fonte de atualização do `PC`. Quando configurado como `0`, o novo valor do `PC` é extraído dos `bits 00-07` da instrução (para saltos). Quando configurado

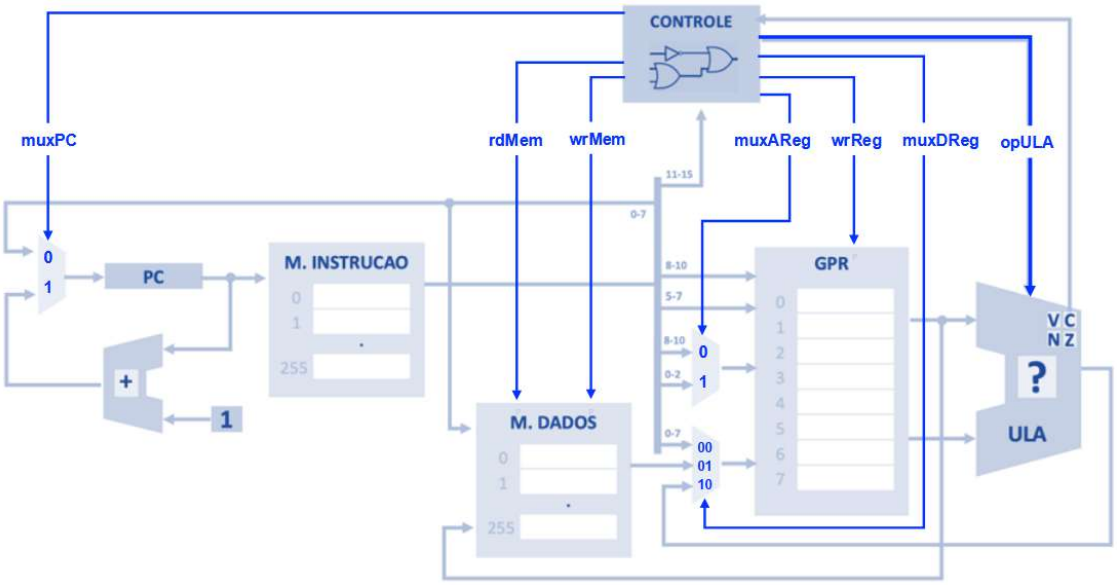
como `1`, o `PC` é atualizado com o valor fornecido pela saída do somador (`PC + 1`), que representa o avanço sequencial do programa.

- `muxAReg`: Seleciona qual campo da instrução será usado para indicar o registrador de destino. Quando configurado como `0`, os `bits 08-10` são usados (em `LDA` e `LDAI`). Quando em `1`, os `bits 00-02` são usados (em instruções aritméticas e lógicas).

- `muxDReg`: Determina a origem do dado a ser escrito no registrador de destino. Os valores possíveis são: `00` para valor imediato presente nos `bits 00-07` da instrução (`LDAI`); `01` para valor lido da Memória de Dados (`LDA`); e `10` para resultado da ULA (operações aritméticas e lógicas).

- `opULA`: Define qual operação será realizada pela ULA. Os possíveis valores são: `000` para `ADD`; `001` para `SUB`; `100` para `AND`; `110` para `OR`; e `111` para `NOT`.

Figura 29 – Sinais de Controle da Solução Monociclo



Fonte: Elaborada pelo autor

Com esses sinais, a Unidade de Controle pode configurar dinamicamente o comportamento do *Datapath* a cada ciclo, ativando apenas os componentes necessários conforme a instrução lida. A Tabela 3 apresenta a configuração de cada Sinal de Controle para cada tipo de instrução da ISA. Essa tabela é fundamental, pois servirá de base para a implementação dos Circuitos Combinacionais que irão gerar automaticamente os sinais, conforme a instrução lida. A lógica interna desta Unidade já foi desenvolvida, mas, como mencionado anteriormente, a descrição dos circuitos está fora do escopo deste trabalho.

Tabela 3 – Configuração de Sinais para o Controle Monociclo

Instrução	opcode	wrReg	rdMem	wrMem	MuxAReg	MuxAReg1	MuxAReg0	opULA2	opULA1	opULA0	muxPC
LDA	0 0001	1	1	0	0	0	1	0	0	0	1
LDAi	0 0010	1	0	0	0	0	0	0	0	0	1
STA	0 0011	0	0	1	1	1	0	0	0	0	1
ADD	0 0100	1	0	0	1	1	0	0	0	0	1
SUB	0 0101	1	0	0	1	1	0	0	0	1	1
AND	0 0110	1	0	0	1	1	0	1	0	0	1
OR	0 0111	1	0	0	1	1	0	1	1	0	1
NOT	0 1000	1	0	0	1	1	0	1	1	1	1
JZ	0 1001	0	0	0	1	1	0	0	0	0	NOT(Z)
JN	0 1011	0	0	0	1	1	0	0	0	0	NOT(N)
JMP	0 1100	0	0	0	1	1	0	0	0	0	0

3.3 Caminho de Dados Multiciclo

O objetivo desta Seção é apresentar o Caminho de Dados Multiciclo, a segunda Organização de CPU implementada neste trabalho. A apresentação desta Seção será feita com base na comparação entre os dois Caminhos de Dados. O foco é destacar o que foi alterado. Ou seja, serão descritas as modificações físicas e conceituais necessárias para transformar o Caminho de Dados Monociclo em um Multiciclo.

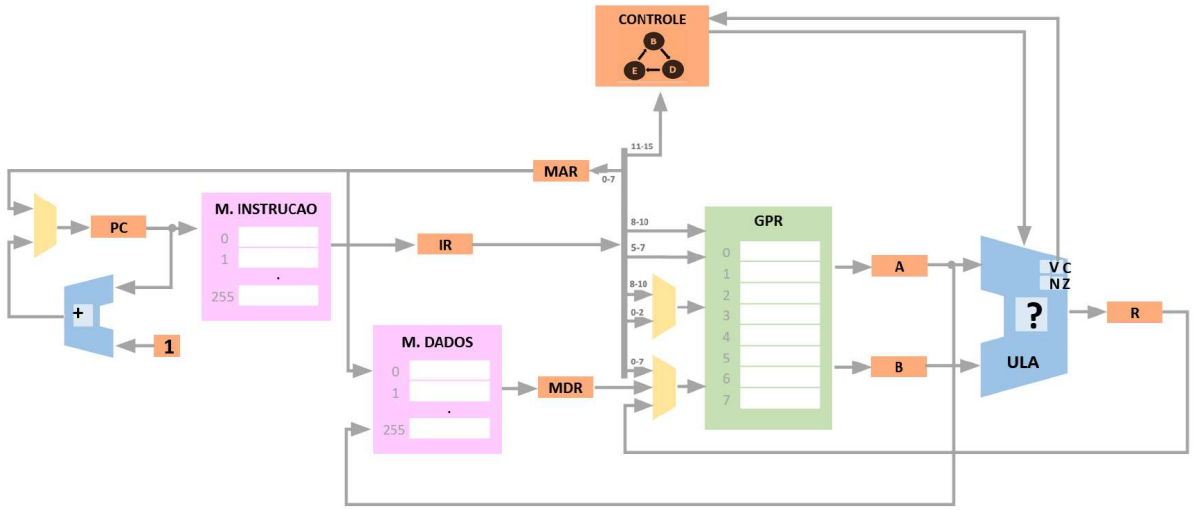
Antes de detalhar as modificações físicas, é importante relembrar as principais diferenças conceituais entre o Caminho de Dados Monociclo e o Multiciclo. No Monociclo todas as etapas da execução de uma instrução acontecem em um único Ciclo de Clock de duração fixa e longa, no Multiciclo a execução é dividida em várias etapas menores. Essa mudança permite que instruções simples sejam executadas rapidamente (em poucos ciclos), enquanto instruções mais complexas usam mais ciclos, sem obrigar todas as instruções a esperar o tempo máximo, como acontecia no Monociclo. Isso possibilita o uso de um período de Clock menor, aumentando a eficiência geral da execução.

3.3.1 Componentes e Conexões

A Figura 30 apresenta o Caminho de Dados Multiciclo Completo. Para melhor compreensão das mudanças, recomenda-se que o leitor compare esta figura com a Figura 23 do Caminho de Dados Monociclo. As alterações em relação ao Monociclo incluem: adição de Registradores Intermediários; e Ajustes nas conexões internas. Para permitir a divisão em etapas, foram adicionados Registradores de Uso Específico no Caminho de Dados.

A Tabela 4 apresenta um resumo dos Registradores. O registrador `IR` (*Instruction Register*) é responsável por armazenar a instrução lida da memória durante a etapa de busca, garantindo que ela permaneça disponível ao longo das demais etapas de execução.

Figura 30 – Caminho de Dados Multiciclo para a ISA Alvo



Fonte: Elaborada pelo autor

O registrador `A` armazena o conteúdo do primeiro registrador fonte lido do Banco de Registradores, servindo como uma das entradas da ULA. De forma semelhante, o registrador `B` guarda o conteúdo do segundo registrador fonte, podendo também ser utilizado como entrada da ULA ou como dado a ser escrito na memória, dependendo da instrução. Após a execução da operação na ULA, o resultado é armazenado no registrador `R`, de onde poderá ser transferido para Memória ou Banco de Registradores.

O registrador `MDR` (*Memory Data Register*) armazena o dado lido da memória, servindo como intermediário para a escrita no Banco de Registradores, especialmente em instruções de carga como `LDA`. Além desses, foi adicionado o registrador `MAR` (*Memory Address Register*), cuja função é armazenar o endereço de memória a ser acessado, seja para usado para leitura ou para escrita. Os registradores tornam possível a separação clara entre as etapas do ciclo de instrução no ambiente Multiciclo, permitindo que informações sejam preservadas entre ciclos.

Tabela 4 – Registradores Intermediários de Caminho de Dados Multiciclo

Registrador	Função	Etapas de Uso
`IR`	Guardar a instrução	Da Busca até o fim da execução
`A`	Guardar o Operando Fonte 1	Da Leitura de Registradores até a Execução na ULA
`B`	Guardar o Operando Fonte 2	Da Leitura de Registradores até a Execução ou Memória
`R`	Guardar o Resultado da ULA	Da Execução até a Escrita ou Acesso à Memória
`MDR`	Guardar o Dado vindo da Memória	Da Leitura da Memória até a Escrita
`MAR`	Guardar o Endereço de Memória	Da Decodificação até o Acesso à Memória

No Caminho de Dados Monociclo, existem duas memórias separadas (Memória de Instruções e Memória de Dados, blocos rosa) para que seja possível acessar ambas dentro do mesmo ciclo. Isso é obrigatório porque, em um ciclo único, era preciso buscar a instrução e acessar os dados ao mesmo tempo. No Multiciclo, como as etapas são separadas, a mesma Memória pode ser usada tanto para instruções quanto para dados, acessados em ciclos diferentes. Embora em um primeiro momento tenha sido projetado um Caminho com Memórias Unificadas, foi decidido manter as memórias separadas para facilitar a comparação entre as abordagens por parte dos estudantes.

A Tabela 5 apresenta os controles dos Multiplexadores (blocos amarelos). Cada um desses multiplexadores será devidamente controlado por um Sinal de Controle, são eles:

Tabela 5 – Multiplexadores do Caminho de Dados Multiciclo

Multiplexador	Função	Opções de Seleção
`muxPC`	Seleciona fonte de atualização do PC	`PC+1` ou `Endereço` de Salto
`muxAReg`	Seleciona o campo da instrução com registrador alvo	`Bits 08-10` (`LDA`, `LDAI`) ou `Bits 00-02`
`muxDReg`	Seleciona a origem do dado a se escrito nos Registradores	`Imediato` (`LDAI`) ou `Endereço` (`LDA`) ou `Saída ULA`

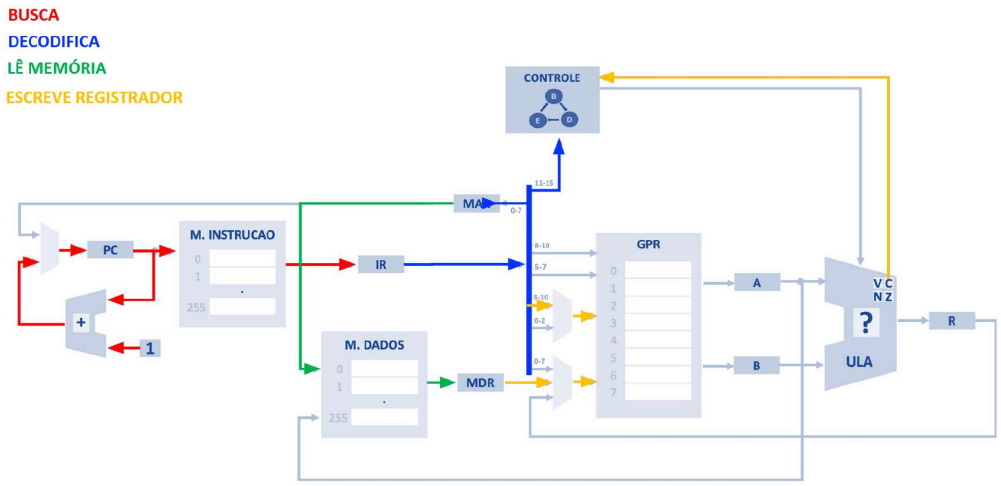
3.3.2 Fluxo de Execução

Nesta Seção, será apresentado detalhadamente o Fluxo de Execução de algumas instruções, agora considerando o Caminho de Dados Multiciclo. É importante destacar que, diferentemente do controle Monociclo, aqui cada etapa da execução ocorre em um Ciclo de Clock distinto.

A Figura 31 apresenta o fluxo de execução da instrução `LDA` no modo Multiciclo. As etapas realizadas são exatamente as mesmas mostradas na Figura 24: Busca, Decodificação, Acesso à Memória e Escrita no Banco de Registradores. A diferença fundamental é que, no Multiciclo, cada etapa armazena temporariamente suas informações em registradores internos, garantindo que os dados permaneçam disponíveis até que a próxima etapa seja executada. Após a Busca, a instrução é carregada em `IR`. Durante o acesso à Memória de Dados, o valor lido é armazenado em `MDR` antes de ser transferido ao Banco de Registradores. Cada caminho percorrido no circuito é destacado em cores distintas, evidenciando o uso desses registradores intermediários.

A Figura 32 apresenta o fluxo de execução das instruções aritméticas e lógicas no processador Multiciclo e deve ser comparada à Figura 27, que mostra o mesmo tipo de instrução no modelo monociclo. No Multiciclo, a instrução é inicialmente buscada e armazenada no registrador `IR`, que conserva seus dados durante as etapas seguintes. Com

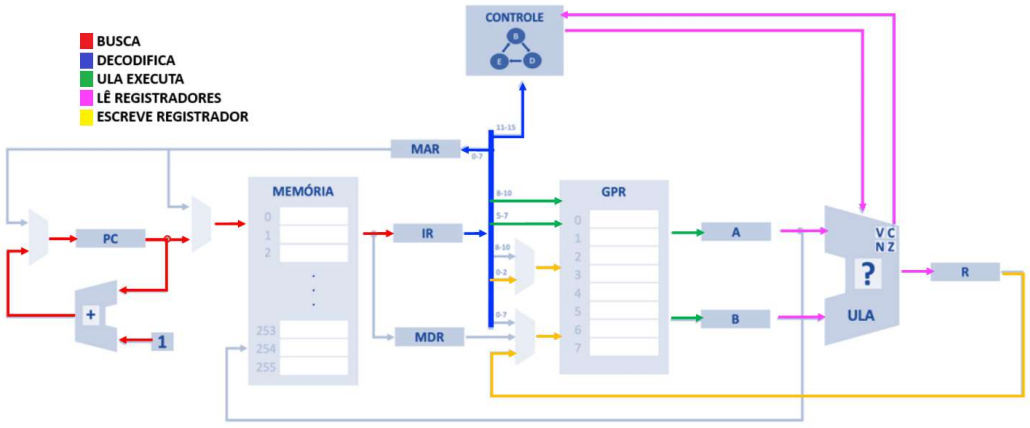
Figura 31 – Etapas de Execução de uma Instrução `LDA`



Fonte: Elaborada pelo autor

base nos dados do `IR`, o Banco de Registradores é acessado e os valores dos registradores são lidos. Esses valores são então transferidos para os registradores internos `A` e `B`, armazenando seus dados. Em seguida, a ULA utiliza `A` e `B` para realizar a operação especificada pelo código `opULA`. O resultado produzido é armazenado no registrador interno `R`, que preserva esse valor até o momento da escrita. Por fim, o conteúdo de `R` é enviado de volta ao Banco de Registradores e gravado no registrador de destino conforme a seleção indicada por `muxDReg`. Esse fluxo evidencia a principal diferença em relação ao Monociclo, uma vez que o uso de `IR`, `A`, `B` e `R` possibilita dividir a execução em várias etapas sucessivas em vez de realizar toda a operação em um único ciclo.

Figura 32 – Etapas de Execução de uma Instrução que usa a ULA



Fonte: Elaborada pelo autor

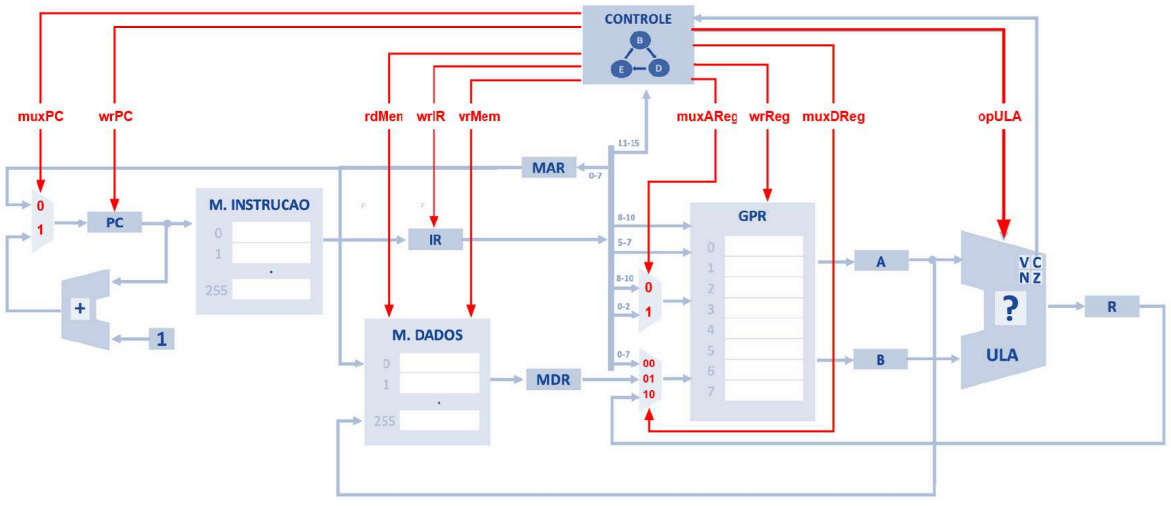
No modelo Multiciclo, a execução é organizada em etapas sucessivas e cada fase produz informações que precisam ser mantidas temporariamente para que a operação prossiga corretamente nos ciclos seguintes. Observa-se que essas etapas correspondem às mesmas fases gerais presentes no Monociclo e seguem a mesma ordem lógica de processamento. A diferença essencial é que, no Multiciclo, existe a necessidade de preservar resultados intermediários entre uma etapa e outra para garantir a continuidade da execução ao longo do tempo. Como esse comportamento se repete de forma equivalente em todos os demais casos, não é necessário apresentar individualmente os outros fluxos.

3.3.3 Sinais de Controle

O Controle Multiciclo funciona por meio de um módulo específico, que, a partir do valor do Opcode, gera os Sinais de Controle necessários para ativar apenas as partes do Caminho de Dados que são requeridas na etapa atual da instrução. Embora o funcionamento interno desse módulo de controle não seja detalhado aqui, é importante entender que ele é responsável por garantir que a execução ocorra de forma ordenada e eficiente, respeitando as dependências e sequenciamento das etapas. Este controle será implementado via Circuito Sequencial, que pode ser representado através da Máquina de Estados Finitos (Figura 34).

Na Figura 33, estão ilustradas todas as conexões dos Sinais de Controle com os diversos componentes do caminho de dados. É possível observar que alguns sinais permanecem os mesmos em relação à organização anterior, enquanto outros foram adicionados. Cabe destacar que até mesmo os sinais mantidos, agora devem apresentar um comportamento diferente, orientado a etapas.

Figura 33 – Sinais de Controle da Solução Multiciclo



Fonte: Elaborada pelo autor

O sinal `wrPC` é novo. Ele é responsável por controlar a escrita no registrador `PC`. Esse sinal é ativado em dois momentos distintos: durante a etapa de busca, para armazenar o valor `PC+1`, e nas etapas finais de instruções de salto, onde o novo endereço de desvio deve ser carregado. Sua atuação seletiva garante que o `PC` seja atualizado apenas no instante apropriado. Na FSM apresentada na Figura 34, observa-se que o `wrPC` é ativado especificamente na etapa de Busca da Instrução, quando o valor `PC+1` é armazenado, e também nas etapas de Execução das instruções de desvio, quando o endereço de salto é carregado pelo caminho de dados.

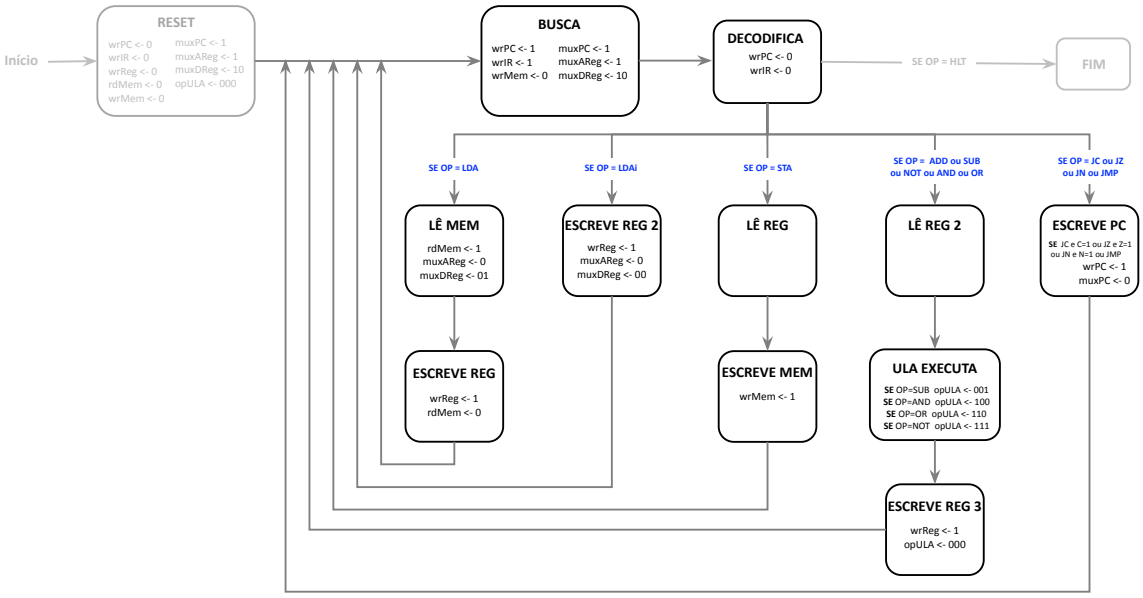
O sinal `wrIR` também é novo. Ele controla a escrita no registrador `IR`, sendo ativado exclusivamente durante a etapa de busca da instrução. Ele permite que o conteúdo lido da memória seja armazenado e mantido durante as demais etapas do ciclo de execução. Como se vê na FSM, esse sinal aparece ativo apenas no primeiro estado, destacando sua função específica e pontual no início do processamento.

O sinal `muxPC` define a fonte do novo valor do `PC`, sendo crucial nas instruções de salto. Ele seleciona entre a saída do somador (`PC+1`) e o campo de endereço da instrução (em saltos). Nas instruções sequenciais, o `muxPC` seleciona a saída do somador durante a etapa de Busca da Instrução, garantindo o avanço natural do `PC`. Já nas instruções de desvio, ele passa a selecionar o campo de endereço durante as etapas de Execução em que o salto é resolvido. Essa seleção pode ser observada na FSM justamente nos estados em que o novo endereço é definido e o fluxo de controle é atualizado.

O sinal `muxAReg` controla qual campo da instrução será usado para identificar o registrador de destino. Ele escolhe entre os `bits 08 a 10`, utilizados quando a etapa de Decodificação identifica instruções de carregamento como `LDA` e `LDAI`, e os `bits 00 a 02`, utilizados quando a mesma etapa reconhece instruções aritméticas ou lógicas. Essa distinção mantém a compatibilidade com diferentes padrões de codificação durante a Decodificação. Na FSM, observa-se que a configuração desse sinal ocorre especificamente nos estados destinados à Decodificação, momento em que o tipo da instrução é determinado e o registrador de destino é selecionado.

O sinal `muxDReg` define a origem do dado que será escrito no Banco de Registradores. Ele seleciona entre três fontes distintas: o valor imediato utilizado quando a etapa de Execução trata uma instrução `LDAI`, o dado vindo da Memória de Dados quando a etapa de Execução corresponde a uma instrução `LDA` e o resultado produzido pela ULA quando a operação aritmética ou lógica é finalizada. Cada uma dessas fontes só se torna válida em etapas específicas da Execução. Na FSM, observa-se que o `muxDReg` é configurado justamente nos estados em que cada um desses valores é disponibilizado, imediatamente antes da etapa em que ocorre a escrita no registrador.

Figura 34 – FSM base do Controle Multiciclo para a ISA alvo



Fonte: Elaborada pelo autor

Neste Capítulo foi apresentada toda a Arquitetura Alvo que serviu como base para a especificação do projeto de uma CPU. Essa ISA se caracteriza por sua simplicidade, utilizando instruções de 16 bits, um conjunto reduzido de instruções e poucos formatos de codificação, facilitando a compreensão e implementação dos conceitos. Foram discutidas duas versões do Caminho de Dados, o modelo Monociclo e o Multiciclo, que serão implementadas no Simulador proposto. Também foram analisados os fluxos de execução para cada tipo de instrução da ISA, permitindo observar que, para cada instrução, diferentes partes do Caminho de Dados são ativadas. Essa análise evidencia como os componentes internos do processador interagem durante a execução, com destaque para os Registradores Intermediários, os Multiplexadores e os Sinais de Controle. Agora que temos uma visão clara, podemos avançar para o próximo Capítulo, onde será apresentada a Metodologia do Trabalho e a implementação do Simulador Proposto.

4 Simulador de Caminho de Dados

Esse Capítulo introduz o funcionamento geral do Simulador proposto, desde a interação do usuário com a interface até a forma como a execução é apresentada visualmente. Para isso, é descrito o papel do Montador na análise e tradução do código *Assembly*, e também a etapa de simulação textual, responsável por gerar as informações que alimentam a interface gráfica. Por fim, é apresentada a distinção entre os modos Monociclo e Multiciclo, destacando como cada um organiza a execução das instruções no tempo, bem como isso se reflete na atualização da interface durante a simulação.

4.1 Visão Geral

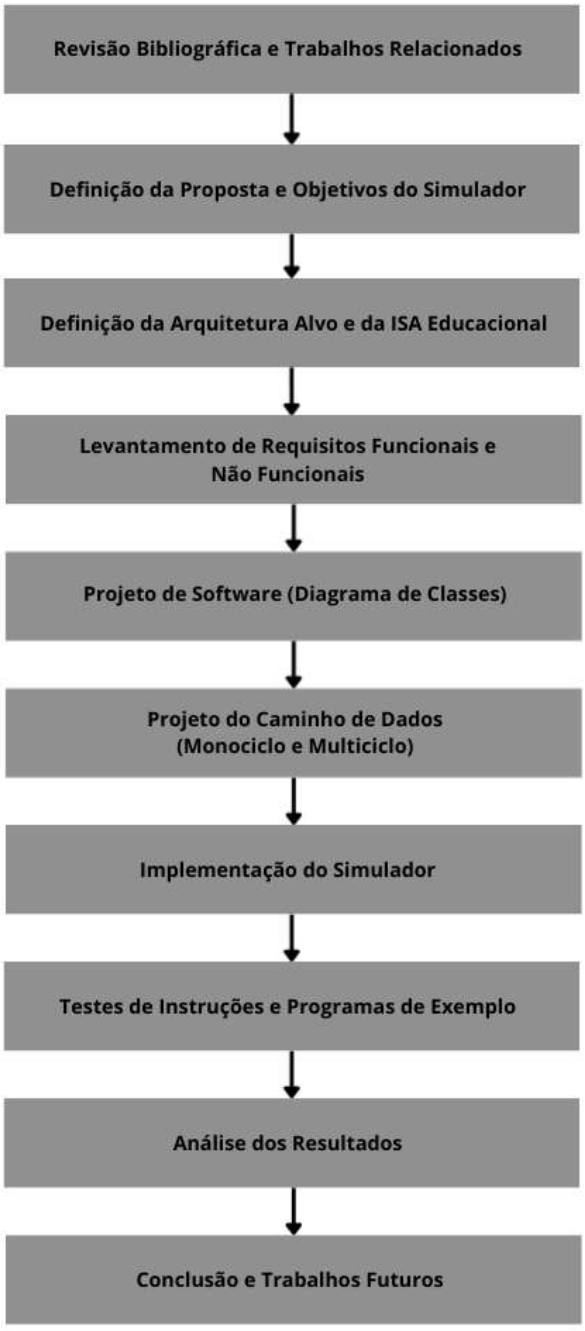
O objetivo central deste trabalho é o desenvolvimento de um simulador educacional voltado para disciplinas de Organização e Arquitetura de Computadores, promovendo um estudo integrado entre os conceitos de Arquitetura e Execução de Instruções no nível de Organização da CPU, com visualização do Caminho de Dados. O simulador busca facilitar o aprendizado por meio de uma abordagem visual e interativa, permitindo a simulação passo-a-passo do funcionamento interno da CPU desenvolvida para a Arquitetura Alvo.

A Figura 35 apresenta de forma visual a sequência das etapas seguidas no desenvolvimento do simulador. Como etapa inicial do trabalho, foi realizada uma revisão bibliográfica sobre conceitos de Arquitetura e Organização de Computadores, simuladores educacionais existentes e abordagens de ensino aplicadas à área. Os resultados dessa análise foram apresentados e discutidos no Capítulo 2, servindo como base teórica para as decisões de projeto adotadas ao longo do desenvolvimento.

A partir dessa Fundamentação, foi definida uma Arquitetura Computacional própria, desenvolvida especificamente para fins didáticos, com um conjunto reduzido de instruções e uma estrutura simplificada, adequada aos objetivos educacionais traçados. Em seguida, foram definidos os Requisitos Funcionais e Não Funcionais da ferramenta, destacando-se a necessidade de simplicidade, clareza didática e acessibilidade por Web em dispositivos desktop e móveis. A descrição dos requisitos encontra-se na seção 4.2.

A Metodologia adotada baseou-se em um processo de desenvolvimento incremental, com validações parciais ao longo das etapas, permitindo maior flexibilidade e adaptação conforme o avanço do projeto. Após a definição da arquitetura e dos requisitos, foram escolhidas as tecnologias a serem utilizadas, com ênfase em ferramentas Web, visando facilitar o acesso e a portabilidade da aplicação.

Figura 35 – Metodologia do Trabalho



Fonte: Elaborada pelo autor

Na sequência, foi realizado o projeto do Caminho de Dados, contemplando tanto a versão Monociclo quanto a Multiciclo, possibilitando diferentes abordagens de simulação do funcionamento da CPU. Paralelamente, foi desenvolvido o projeto do simulador, incluindo a definição da estrutura do Software e o esboço da Interface Gráfica.

Com o projeto definido, iniciou-se a fase de implementação. Primeiramente, foi criado um Montador (*Assembler*), responsável por converter as instruções em linguagem

simbólica para sua representação binária, permitindo a execução correta das simulações. Em seguida, foi implementado o Simulador Monociclo, que permite a execução visual das instruções em um único Ciclo de Clock, servindo como base inicial da simulação da CPU. Após sua implementação, foram realizados testes para verificar a correta execução das instruções e a coerência na atualização dos sinais de controle, registradores e memória.

Posteriormente, foi realizada a implementação do Simulador Multiciclo, adaptando o simulador para o modelo Multiciclo por meio da inclusão de registradores intermediários e da execução sequencial das instruções, controlada por uma Máquina de Estados Finitos (FSM). Após essa etapa, foram aplicados testes específicos para garantir a correta lógica de controle e a execução passo a passo das instruções.

Concluídas as etapas de implementação e validação, a ferramenta foi disponibilizada via Web, acompanhada de um Manual do Usuário integrado à interface, descrevendo o funcionamento do simulador, seus modos de operação e os principais componentes da arquitetura. Por fim, foi elaborada a documentação técnica final do projeto, incluindo a descrição da arquitetura interna do simulador, a estrutura do código e orientações para manutenção e futuras extensões, garantindo sua reutilização em outros projetos.

4.2 Projeto do Simulador

Esta Seção apresenta o Projeto Conceitual do Simulador. A subseção 4.2.1 será dedicada ao Levantamento de Requisitos, enquanto a subseção 4.2.2 abordará as Ferramentas e Metodologias utilizadas no desenvolvimento. Na subseção 4.2.3 será apresentado o Diagrama de Classes que estrutura os principais componentes do sistema.

4.2.1 Levantamento de Requisitos

O Levantamento de Requisitos é uma etapa fundamental para garantir que o simulador atenda às necessidades dos usuários, especialmente no contexto educacional. Os requisitos foram classificados em funcionais e não funcionais, conforme proposto pela Engenharia de Software.

Os Requisitos Funcionais descrevem o comportamento esperado do sistema, ou seja, as funcionalidades que o simulador deve oferecer para cumprir seu propósito. A Tabela 6 apresenta esses requisitos, entre os quais se destacam a simulação passo-a-passo das instruções da arquitetura (RF01), a visualização gráfica do caminho de dados com destaque para os componentes ativos (RF02), e o controle da execução com comandos de pausa, avanço e reinício (RF03). O simulador também deve permitir a entrada de código em LM ou *Assembly* (RF04), realizar a conversão automática de *Assembly* para LM em binário (RF05), e exibir relatórios com o estado dos registradores e da memória (RF06).

Tabela 6 – Requisitos Funcionais do Simulador

ID	Requisito	Descrição
RF01	Simulação de instruções	Permitir a execução passo a passo das instruções da arquitetura.
RF02	Visualização Caminho de Dados	Exibir graficamente o caminho de dados, destacando componentes.
RF03	Controle de Execução	Permitir pausar, avançar e reiniciar a simulação.
RF04	Entrada de Código	Permitir inserção de código em LM ou *Assembly* via interface.
RF05	Integração com Montador	Converter automaticamente código *Assembly* em instruções binárias.
RF06	Relatório de Execução	Exibir valores de registradores e memória após cada instrução.

Os Requisitos Não Funcionais se referem às qualidades do sistema, como desempenho, usabilidade, portabilidade, entre outros. A Tabela 7 apresenta esses requisitos, entre os quais se destacam a necessidade de uma interface intuitiva voltada para discentes iniciantes (RNF01), o funcionamento Web em dispositivos móveis (RNF02), e a execução da simulação em tempo real, sem atrasos perceptíveis (RNF03). Também são importantes a utilização de tecnologias web para garantir portabilidade (RNF04), a modularização do código para facilitar sua manutenção (RNF05), a estabilidade diante de entradas inesperadas (RNF06) e a compatibilidade com navegadores mais usados (RNF07).

Tabela 7 – Requisitos Não Funcionais do Simulador

ID	Requisito	Descrição
RNF01	Usabilidade	Interface intuitiva, voltada para estudantes com pouca experiência.
RNF02	Responsividade	Deve funcionar Web em dispositivos móveis.
RNF03	Desempenho	A simulação deve ocorrer em tempo real, sem atrasos perceptíveis.
RNF04	Portabilidade	Desenvolvido com tecnologias web como HTML, CSS e JavaScript.
RNF05	Modularidade	Código modular para facilitar manutenção e expansão.
RNF06	Confiabilidade	Operação estável mesmo diante de entradas inesperadas.
RNF07	Compatibilidade	Compatível com os principais navegadores (Chrome, Firefox, Edge, Safari).

4.2.2 Ferramentas e Metodologias

O desenvolvimento de um simulador educacional exige a escolha de Ferramentas e Metodologias que garantam modularidade e facilidade de manutenção, ao mesmo tempo em que permitam uma boa experiência de uso e aprendizado. Neste contexto, cada decisão descrita abaixo baseou-se em critérios técnicos e pedagógicos.

A linguagem JavaScript foi escolhida para a implementação por ser amplamente suportada por navegadores modernos, permitindo o uso da aplicação sem necessidade de instalação ou configuração adicional. Isso está alinhado com o objetivo de criar uma ferramenta acessível tanto em desktop quanto em smartphones. Além disso, JavaScript oferece uma vasta gama de bibliotecas e *frameworks* que facilitam a criação de interfaces interativas e responsivas, fundamentais para o aspecto visual do simulador.

A utilização do Paradigma de Orientação a Objetos permite organizar o código de forma estruturada, promovendo modularidade, reutilização de componentes e facilidade de manutenção. Componentes como registradores, unidades funcionais, barramentos e instruções foram implementados como classes, permitindo abstrair comportamentos e facilitar a expansão futura. A Orientação a Objetos suporta o encapsulamento da lógica interna dos componentes, tornando o sistema confiável e flexível para atualizações.

O Repositório do Projeto encontra-se hospedado no *GitHub*, devido às vantagens em termos de controle de versão, rastreabilidade de mudanças e colaboração. Além disso, o *GitHub* permite a publicação da aplicação como uma página estática via *GitHub Pages*, atendendo diretamente ao requisito de acessibilidade via Web. A plataforma também facilitou a organização de tarefas, documentação e permitiu a disponibilização do projeto como ferramenta educacional de código aberto.

A Metodologia de Desenvolvimento seguiu um Modelo Iterativo e Incremental, com o desenvolvimento por etapas e a entrega gradual de funcionalidades. Essa abordagem é ideal para projetos acadêmicos, pois permite validação contínua, revisões periódicas e incorporação de *feedbacks* antes da conclusão do produto final.

Por fim, para garantir o alinhamento com os objetivos pedagógicos, a implementação foi guiada por Princípios de Usabilidade e Acessibilidade, considerando as limitações de tempo de aula, familiaridade dos estudantes com tecnologia, e clareza visual para o aprendizado conceitual da arquitetura de computadores.

4.2.3 Diagrama de Classes

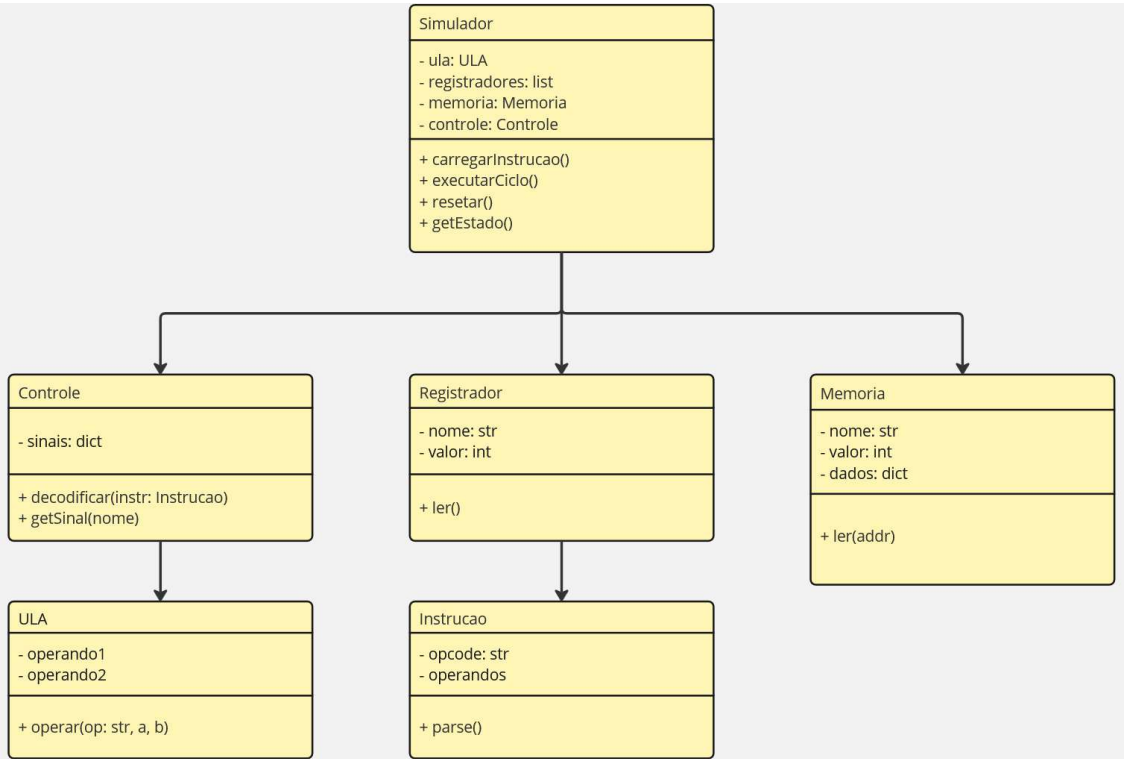
Um Diagrama de Classes é uma representação fundamental da estrutura do Software, o qual permite visualizar a relação entre os componentes principais do sistema e antecipar sua organização interna. A Figura 36 apresenta o Diagrama de Classes do Simulador, refletindo a organização atual do sistema. Ela é composta pelas seguintes classes principais: Simulador, Registrador, Memória, ULA, Controle e Instrução.

A Classe Simulador atua como a principal, concentrando os componentes por meio de seus atributos, incluindo a ULA, o conjunto de Registradores, a Memória, a Unidade de Controle, além do Clock e *Logs*. O Clock é responsável por coordenar a progressão temporal da simulação, enquanto os *logs* permitem o registro das informações que serão visualizadas durante a simulação.

Já a Classe Controle é responsável pelo gerenciamento dos Sinais de Controle que orientam o funcionamento das demais unidades. A ULA possui como atributos os operandos de entrada, operando1 e operando2, sendo encarregada da execução das operações aritméticas e lógicas. A Classe Instrução é composta pelo Opcode e pelos operandos, representando a estrutura básica das instruções processadas pelo simulador.

Os Registradores são definidos por um nome e um valor armazenado, enquanto a Memória possui atributos como nome, valor e dados, permitindo a simulação do armazenamento e acesso às informações.

Figura 36 – Diagrama de Classes do Simulador



Fonte: Elaborada pelo autor

4.3 Implementação do Simulador

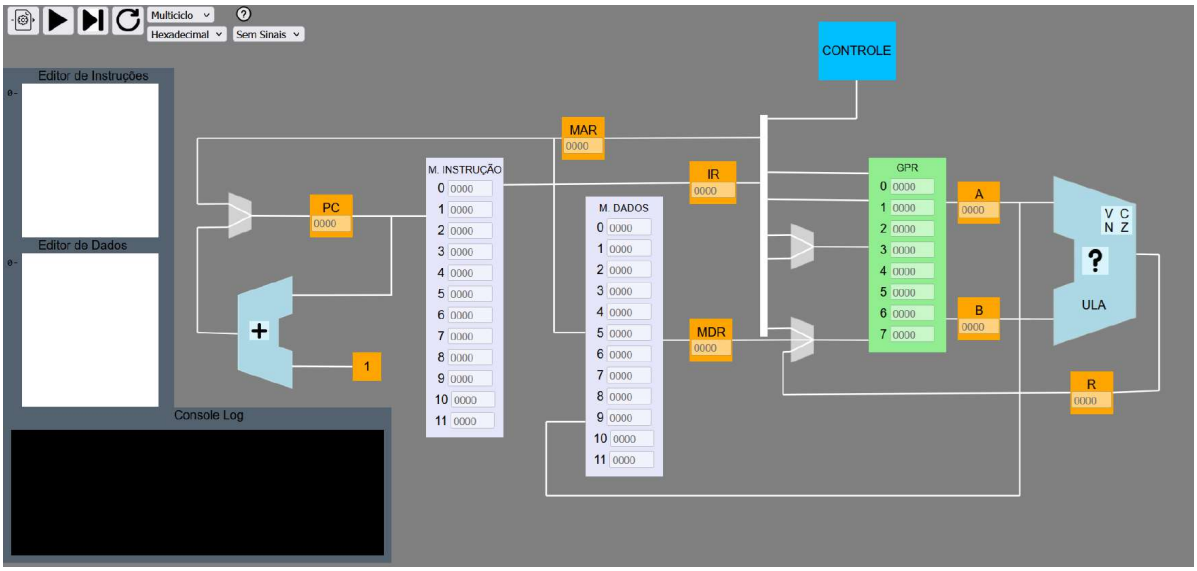
Esta seção apresenta a implementação do simulador de Caminho de Dados proposto. São descritos os principais módulos desenvolvidos, bem como suas funções no funcionamento da ferramenta. A implementação foi organizada de forma modular, contemplando a Interface Gráfica, o Montador de Assembly, a Simulação Textual e os Modos de Simulação.

4.3.1 Interface Gráfica

A Interface Gráfica do simulador é um dos elementos centrais para a experiência educacional, pois é por meio dela que o usuário interage com o Caminho de Dados, insere Instruções via progração *Assembly*, executa a simulação e compreende o fluxo interno da CPU. Foram projetadas três interfaces, para cada opção de Caminho de Dados, mas aqui será apresentada a versão Multiciclo. Cada uma dessas interfaces possui elementos em comum, mas também diferenças importantes na forma de execução e visualização. Abaixo,

podem ser observadas figuras com os esboços de duas interfaces: uma versão sem sinais de controle (Figura 37), uma com todos os sinais visíveis (Figura 38). Uma terceira interface com visualização parcial dos sinais também foi desenvolvida para reduzir a poluição visual, facilitando o entendimento.

Figura 37 – Interface do Modo Multiciclo (SEM Sinais de Controle)



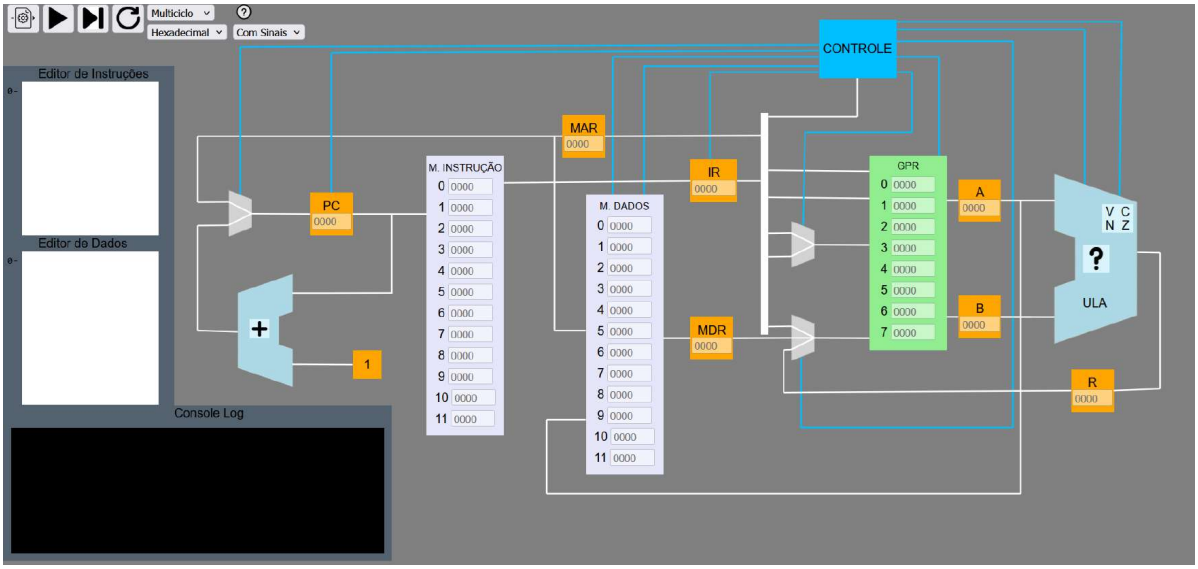
Fonte: Elaborada pelo autor

Conforme mencionado, as versões da interface compartilham uma estrutura comum. A Área de Entrada de Código permite que o usuário digite ou cole instruções em Linguagem *Assembly*, funcionando como ponto de partida da simulação. Abaixo há a área para Entrada de Dados, que permite a Declaração de Variáveis, assim como seus valores iniciais. Logo abaixo do Campo de Código, há um Console de Log que exibe mensagens de erro, avisos e informações relevantes sobre o andamento da execução, permitindo que o usuário acompanhe em tempo real o comportamento do sistema e receba *feedback* contínuo.

A parte central da interface é dedicada à Visualização Gráfica do Caminho de Dados, onde os blocos funcionais da CPU, como a ULA, os Registradores, a Memória de Instruções e a Memória de Dados são apresentados de forma clara e destacada. Esses elementos são dinamicamente atualizados para refletir o valor das informações em tempo de execução, a cada passo de cada instrução do programa, facilitando a compreensão do fluxo de informações dentro da CPU.

A interação com a simulação é realizada por meio de Botões de Controle que permitem ao usuário Montar o Programa, Executar o Programa por Completo, Avançar Passo-a-Passo cada Ciclo de Clock e Reiniciar o Sistema a qualquer momento. O botão Montar é responsável por traduzir o código inserido, preparando-o para a execução, cuja explicação é detalhada adiante no texto.

Figura 38 – Interface do Modo Multiciclo (COM Sinais de Controle)



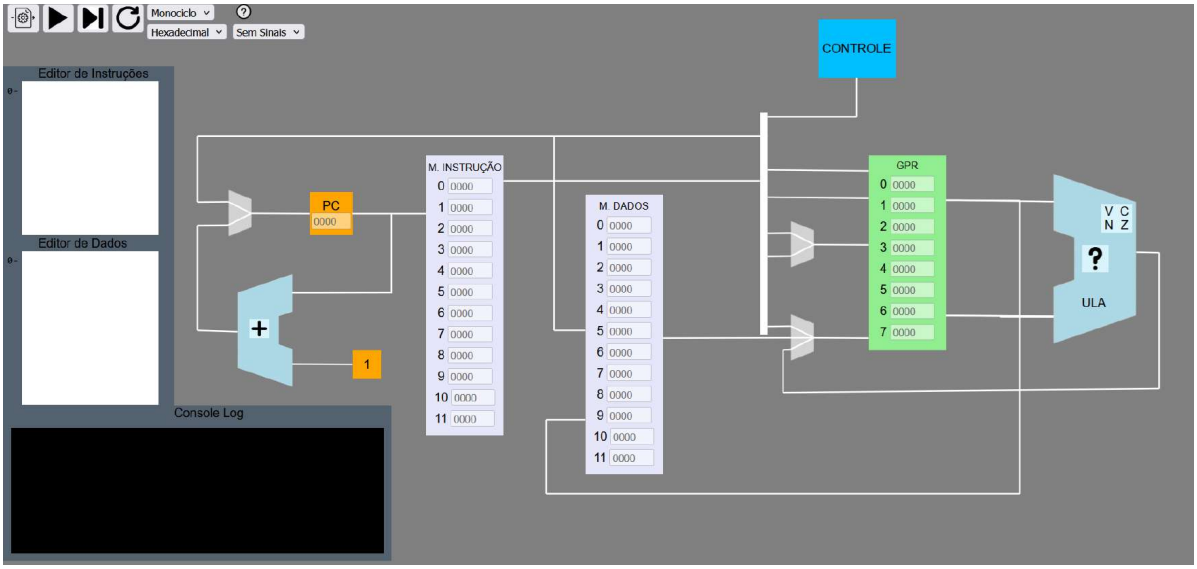
Fonte: Elaborada pelo autor

Além disso, a interface disponibiliza uma Barra de Configurações que possibilita ajustar diferentes aspectos da visualização e do funcionamento da simulação. Nela, o usuário pode selecionar o tipo de *Datapath* a ser utilizado (Monociclo ou Multiciclo). Ele também pode alternar entre diferentes Modos de Exibição, como a Visualização Completa, a Visualização Parcial ou a Ocultação dos Sinais de Controle. Ainda é possível definir a Base Numérica de Apresentação dos Valores, escolhendo entre Decimal e Hexadecimal. Informações representadas em Binário são exibidas apenas no log do sistema, permitindo uma análise mais detalhada do comportamento interno da Arquitetura simulada. Esta escolha foi tomada porque valores binários tornam os blocos muito grandes, poluem a visualização e reduzem o espaço para representação das movimentações de informação.

A Visualização do Fluxo de Dados usa setas animadas que mostram os Movimentos das Informações destacadas por cores nos Barramentos de Interconexão, ilustrando quais dados estão sendo transportados a cada instante. Cabe aqui destacar que, embora as conexões sejam representadas por fios únicos, cada uma delas é, na verdade, um Barramento de fios, o qual suporta a transmissão de informações em paralelo. A largura de cada barramento é compatível com o tamanho da informação transmitida (número fios igual número bits). Optou-se por não mostrar os Barramentos para não poluir a interface.

Essa abordagem visual é fundamental para que o aluno possa construir um modelo mental preciso e compreensível do funcionamento interno do processador, facilitando a aprendizagem dos conceitos de arquitetura computacional. Pode-se ver a movimentação das informações dentro da CPU simulada na Figura 39.

Figura 39 – Fluxo de Infos no Modo Monociclo (SEM Sinais de Controle)



Fonte: Elaborada pelo autor

As interfaces dos Modos Monociclo e Multiciclo apresentam diferenças estruturais que refletem diretamente o modelo de execução adotado. A principal distinção está na ausência de Registradores Intermediários no Modo Monociclo, uma vez que toda a execução da instrução ocorre em um único Ciclo de Clock, não havendo necessidade de armazenar resultados parciais entre etapas. Em contraste, no Modo Multiciclo, esses Registradores Intermediários são essenciais para preservar dados e Sinais de Controle ao longo dos diferentes ciclos que compõem a execução de uma instrução. Para facilitar o entendimento dessas diferenças, recomenda-se comparar a Figura 39, que corresponde ao *Datapath* Monociclo, com a Figura 38 que correspondente ao *Datapath* Multiciclo.

4.3.2 Montador (*Assembler*)

O Montador tem como principal finalidade traduzir o Código Escrito em Linguagem de Montagem (*Assembly*) para uma representação compreensível pelo simulador (Linguagem de Máquina), permitindo que as instruções possam ser executadas no *Datapath* selecionado. No contexto desta ferramenta, o montador foi implementado de forma simplificada, uma vez que o conjunto de instruções e a sintaxe do *Assembly* adotado são reduzidos e bem definidos. Dessa forma, seu funcionamento se aproxima de um *Parser*, responsável por ler o código-fonte, identificar instruções e operandos, e organizar essas informações em estruturas internas adequadas para a simulação.

Conforme pode-se observar na Figura 39, a Área de Código fornecida ao usuário é dividida em duas seções distintas. A parte de Código contém as instruções que serão simuladas pelo processador, enquanto que a parte de Dados é destinada à definição de valores que poderão ser acessados durante a execução do programa. O processo de Montagem ocorre, de forma resumida, em dois passos principais. No primeiro passo, o Montador desenvolvido analisa o texto de entrada, valida a sintaxe, identifica rótulos (*labels*) e separa corretamente as seções de Código e Dados. No segundo passo, as instruções já analisadas são traduzidas para sua representação interna, com a codificação dos Opcodes e Operandos, permitindo o carregamento correto nas memórias de instruções e de dados.

Para facilitar o entendimento, a Figura 40 apresenta um trecho de código em *Assembly*, correspondente a um programa simples de contador, e sua tradução realizada pelo Montador desenvolvido, evidenciando como uma instrução textual é convertida em sua forma codificada e pronta para execução na simulação.

Figura 40 – Exemplo de Montagem de Código

```
        LDA  CNT, R0          00001 000 CNT          00001 000 00000000
        LDA  END, R1          00001 001 END          00001 001 00000001
        LDAi   1, R2          00010 010 00000001     00010 010 00000001
CONTA   ADD  R0, R2, R0       00100 000 010 00 000   00100 000 010 00 000
        STA  R0, CNT          00011 000 CNT          00011 000 00000000
        SUB  R1, R0, R3       00101 001 000 00 011   00101 001 000 00 011
        JZ   FIM              01001 000 FIM          01001 000 00001000
        JMP  CONTA            01100 000 CONTA        01100 000 00000011
FIM     HLT                   01101 000 00000000     01101 000 00000000
```

(a) Código Assembly (b) Montagem Passo1 (c) Montagem Final

Fonte: Elaborada pelo autor

O Montador também é responsável por Identificar e Reportar Erros presentes no código *Assembly*, impedindo a execução de programas inconsistentes. Um exemplo típico ocorre quando um *Label* é utilizado na área de código sem ter sido previamente definido em nenhuma linha. O mesmo mecanismo de verificação é aplicado tanto a *Labels* de dados (utilizados como endereços de operandos nas Instruções `LDA` e `STA`), quanto a *Labels* de código (empregados como destinos nas Instruções de Desvio). Dessa forma, garante-se que todas as referências simbólicas estejam associadas a endereços válidos.

Além da Validação Semântica, o Montador realiza um pré-processamento do texto de entrada, no qual todas as linhas em branco são removidas e desconsideradas, não influenciando o resultado da montagem. Da mesma forma, diferenças de formatação não afetam a análise, uma vez que o uso de tabulações ou múltiplos espaços consecutivos é tratado como um único separador, permitindo maior flexibilidade na escrita do código sem comprometer sua correta interpretação.

4.3.3 Simulação Textual

A Etapa da Simulação Textual ocorre imediatamente após a Montagem. Nessa fase, o programa é executado/simulado integralmente de forma textual, sem qualquer animação gráfica. Seu o objetivo de registrar detalhadamente o comportamento da CPU ao longo do tempo. Como resultado, é gerado um conjunto estruturado de informações aqui referido como *Log*, ainda que não corresponda a um arquivo tradicional que descreve a evolução interna da simulação. Esse conjunto de dados serve posteriormente como base para a Ativação e Sincronização das Animações apresentadas na Interface Gráfica, garantindo que a visualização reflita a execução previamente simulada.

Para cada Ciclo de Clock, ou mais precisamente, para cada etapa associada à execução de uma instrução, são armazenadas informações do Estado do Sistema. Entre esses dados encontram-se os valores: do PC, da Instrução corrente, dos Registradores, das Memórias, bem como o Estado dos Sinais de Controle e os Resultados Intermediários produzidos pela ULA. No caso do Modo Multiciclo, essas informações correspondem às diferentes Etapas que compõem a execução de uma instrução, enquanto no Modo Monociclo todo esse estado é registrado em uma única etapa. Dessa forma, o simulador mantém um histórico completo da evolução do *Datapath*, ao longo da simulação do programa.

Internamente, essas informações são organizadas em Estruturas de Dados Sequenciais, nas quais cada elemento representa o estado completo da CPU em um determinado ciclo/etapa. Cada entrada agrupa os valores dos componentes e sinais relevantes, permitindo que o simulador avance na simulação durante a Visualização Gráfica. Para facilitar o entendimento, a Figura 41 apresenta um exemplo da estrutura de *log*, enquanto a Figura 42 apresenta um log real, gerado pelo simulador para uma instrução `LDA`. Dessa forma, evidencia-se a relação entre as etapas de execução, os Ciclos de Clock e as estruturas internas de armazenamento, bem como a forma como os dados produzidos pela simulação textual são posteriormente utilizados pela interface gráfica.

Figura 41 – Estrutura Padrão de Log

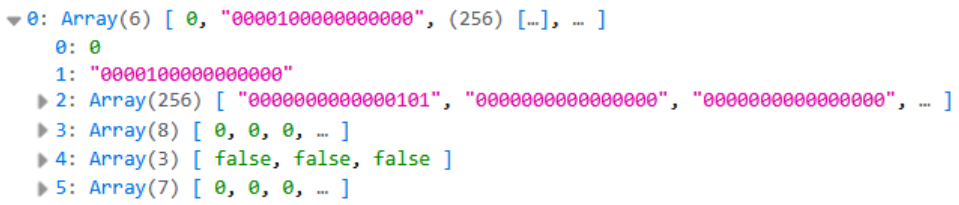
```
Log      = PC, Instrução, [Mémoria de Dados], [Registradores], [Flags], [Controle]
Flags    = [Zero, Carry, Negativo]
Controle = [muxPC, rdMem, wrMem, muxAReg, wrReg, muxDReg, opULA]
```

Fonte: Elaborada pelo autor

Como dito, as informações geradas durante a Simulação Textual são utilizadas como base para atualizar dinamicamente a Interface Gráfica, de modo que cada avanço na visualização corresponda a um estado previamente registrado do processador. A interface interpreta esses dados e atualiza os valores exibidos nos componentes do *Datapath*, nos Registradores, nas Memórias e nos Sinais de Controle, além de acionar as animações que

representam o Fluxo de Dados e a atividade dos blocos funcionais. É importante destacar que esse conjunto de informações é interpretado de forma distinta nos Modos Monociclo e Multiciclo. As próximas seções detalham tais Modos.

Figura 42 – Log gerado para uma Instrução `LDA`



Fonte: Elaborada pelo autor

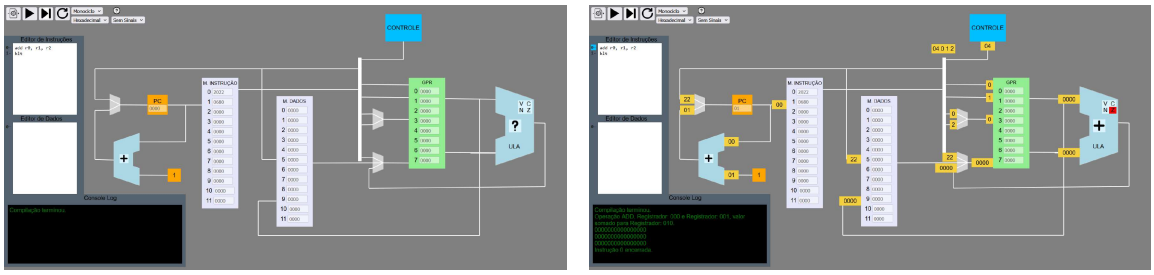
4.3.4 Modo de Simulação Monociclo

Na interface do Modo Monociclo, as informações da simulação são atualizadas com base em um único Ciclo de Clock por instrução. Internamente, o simulador considera que todas as etapas da execução, como Busca, Decodificação e Execução ocorrem de forma simultânea dentro desse único ciclo. No entanto, para fins didáticos e de visualização, o simulador organiza essas etapas de maneira sequencial, permitindo que o usuário acompanhe o Fluxo de Informações e a ativação dos Sinais de Controle de forma compreensível. Assim, embora o modelo teórico do Monociclo execute tudo ao mesmo tempo, a interface apresenta essas etapas uma após a outra, como se fossem passos de um mesmo ciclo.

Para ilustrar esse comportamento, considere a Figura 43 simulando uma instrução `ADD`. Do ponto de vista Arquitetural, essa instrução consome apenas um Ciclo de Clock e, portanto, para o usuário, será apresentada uma única tela animada correspondente a esse ciclo. Dentro dessa visualização, o simulador percorre internamente todas as etapas necessárias à instrução, atualizando os valores do PC, dos Registradores de origem e destino, da ULA e dos Sinais de Controle. A animação exibe essas atualizações de forma ordenada, destacando primeiro a Busca da instrução, em seguida a Decodificação e, por fim, a Execução. Dessa forma, o simulador mantém a fidelidade ao funcionamento do *Datapath* Monociclo, ao mesmo tempo em que oferece uma representação visual mais acessível ao usuário.

No Modo Monociclo, o aspecto mais relevante é compreender como o simulador organiza e consome as informações da simulação para atualizar a interface, e não o fato de que a instrução ocorre em um único ciclo. O simulador trabalha com um estado único por instrução, que consolida todos os efeitos da execução, e a interface utiliza esse estado como referência para atualizar, todos os componentes visuais do *Datapath*.

Figura 43 – Simulação Monociclo (Movimentações em UM Clock)



(a) INÍCIO (b) FINAL

Fonte: Elaborada pelo autor

Outro ponto importante é que, mesmo sem a existência de Registradores intermediários, o simulador mantém uma ordenação lógica das ações internas, permitindo que a interface destaque o fluxo de dados e a ativação dos Sinais de Controle de maneira coerente. Essa ordenação não representa ciclos adicionais, mas apenas uma sequência de atualização visual derivada de um único estado. Dessa forma, a interface não executa a instrução passo-a-passo, mas apenas interpreta e apresenta diferentes aspectos do mesmo estado, evitando ambiguidades entre cálculo, escrita e controle.

Por fim, cabe destacar que essa abordagem simplifica a sincronização entre simulação e interface. Como cada instrução corresponde exatamente a um avanço lógico, a navegação do usuário implica apenas selecionar o próximo estado armazenado. Essa característica reduz a complexidade da interface no modo Monociclo e reforça seu objetivo pedagógico, ao permitir que o foco do usuário esteja na compreensão do fluxo de dados e dos Sinais de Controle, e não na gestão de múltiplos estados temporais.

4.3.5 Modo de Simulação Multiciclo

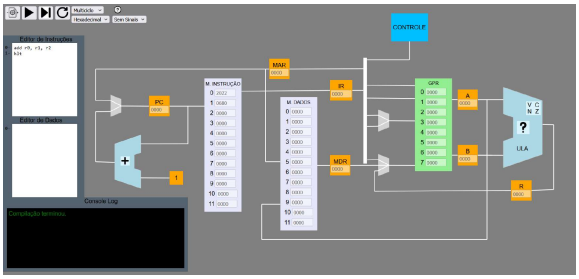
Na interface do Modo Multiciclo, a atualização das informações ocorre de forma diretamente associada ao avanço de cada Ciclo de Clock, sendo que, a cada ciclo, o simulador progride apenas uma etapa da execução da instrução. Essa organização permite que o mesmo conjunto de informações gerado na simulação textual seja reutilizado, já que cada etapa corresponde a um estado distinto do processador. Diferentemente do Monociclo, no Multiciclo a separação temporal entre as etapas é essencial, pois os valores intermediários precisam ser preservados entre ciclos para garantir a correta execução da instrução.

Do ponto de vista da interface, cada avanço de Clock seleciona um novo estado previamente registrado, resultando na atualização dos componentes visuais do *Datapath*, dos Registradores intermediários, das Memórias e dos Sinais de Controle. Assim, a interface reflete o comportamento do *Datapath* Multiciclo, no qual a instrução é naturalmente

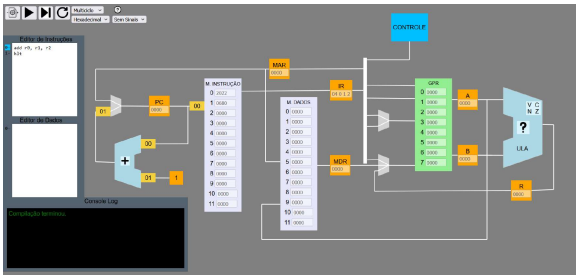
fragmentada em múltiplos ciclos, e cada um deles produz um estado observável.

Considerando a mesma instrução `ADD`, no modo Multiciclo sua execução é distribuída em cinco ciclos de *clock*. Para o usuário, isso se traduz na apresentação de cinco telas animadas distintas, cada uma representando uma etapa específica da instrução, como Busca, Decodifica, Lê Operandos, Cálculo ULA, e Escreve Resultado. Dessa forma, conforme pode-se observar na Figura 44, a interface evidencia claramente a progressão temporal da instrução, permitindo ao usuário acompanhar, ciclo a ciclo, como as informações e os Sinais de Controle evoluem.

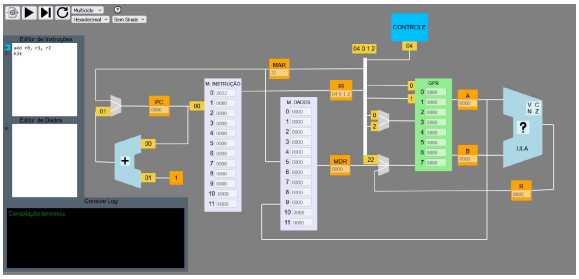
Figura 44 – Simulação Multiciclo (Movimentações a CADA Clock)



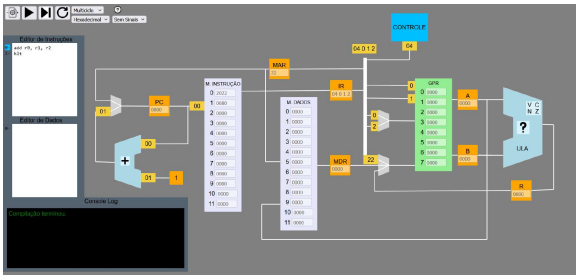
(a) INÍCIO



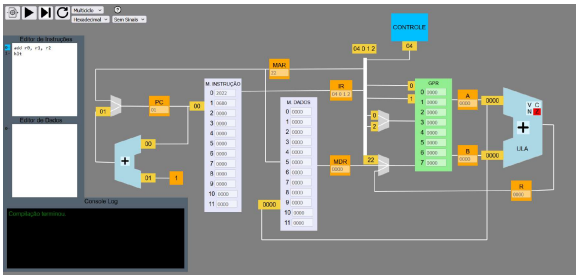
(b) Clock 1



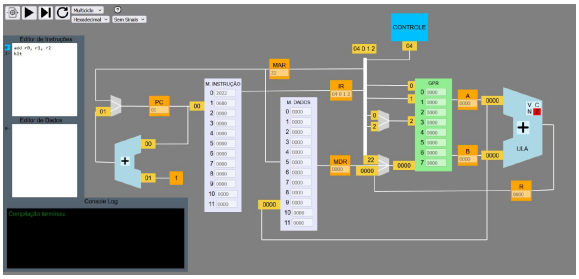
(c) Clock 2



(d) Clock 3



(e) Clock 4



(f) FINAL

Fonte: Elaborada pelo autor

5 Resultados Obtidos

Esta seção apresenta os Resultados Obtidos a partir da execução dos testes desenvolvidos para o simulador, com o objetivo de verificar o correto funcionamento da estrutura proposta, da ISA e dos Caminhos de Dados implementados. Os testes foram organizados de forma progressiva, iniciando por verificações da execução de Instruções de cada um dos tipos suportados (Movimentação de Dados, Aritméticas, Lógicas, Desvios), avançando a Ativação das *flags* da ULA e, por fim, com o emprego de Programas que simulam estruturas clássicas de programação e operações aritméticas. De modo geral, os resultados demonstram que as movimentações de dados, a execução das instruções e manipulação das memórias ocorreram conforme o esperado. As simulações permitiram validar tanto o fluxo de execução das instruções quanto o comportamento da ULA.

5.1 Acesso ao Simulador

Atualmente, o simulador conta com a implementação das versões Monociclo e Multiciclo, ambas acessíveis a partir de um link público da ferramenta em <https://simuladorcpu.github.io/Simulador-de-Caminho-de-Dados/>. Caso a versão final ainda não esteja disponível, uma versão temporária de demonstração pode ser utilizada para avaliar as principais funcionalidades do simulador e seu funcionamento geral. O Manual da Ferramenta, quando disponível, será disponibilizado com a versão final da ferramenta. O projeto é concebido como código aberto (ou encontra-se em processo de abertura), permitindo futuras contribuições, extensões e uso acadêmico.

A operação da ferramenta é simples e direta: o usuário digita ou cola o Código *Assembly* na área de edição, aciona o botão Montar para traduzir o programa e, em seguida, pode executar a simulação utilizando os controles de execução completa (*Play*) ou passo-a-passo, acompanhando a evolução do *Datapath* na interface. Durante a execução, é possível observar o código montado sendo carregado nas memórias correspondentes, bem como a atualização dos valores internos ao longo do tempo.

A ferramenta dispõe de manual de ajuda, acessível a partir da própria interface do simulador. Esse recurso fornece orientações básicas, suas principais funcionalidades, características da CPU simulada e modos de execução. O conteúdo do manual encontra-se disponível em <https://github.com/SimuladorCPU/Simulador-de-Caminho-de-Dados> e será progressivamente expandido e refinado juntamente com a evolução do simulador.

5.2 Testes de Instruções

Aqui serão apresentados três testes, o primeiro voltado às Instruções de Movimentação de Dados envolvendo a Memória, o segundo com Instruções Aritméticas e o terceiro com Instruções Lógicas, Note que os programas utilizados não realizam nenhum cálculo realmente util, sendo que forma elaborados com o único objetivo de testar e validar o funcionamento do simulador e das instruções da ISA.

Todas as simulações foram executadas em modo passo-a-passo, permitindo a verificação do fluxo correto das informações ao longo do Caminho de Dados. Entretanto, para fins de apresentação, são exibidos apenas os resultados correspondentes aos estados finais da memória. Caso contrário, muitas capturas de telas seriam geradas, e o volume de páginas deste documento ficaria excessivo.

Para facilitar a Análise dos Resultados, convencionou-se que todos os valores produzidos pelos testes são armazenados em posições específicas da memória, cujos *labels* correspondem ao valor esperado ao final da simulação. Ou seja, se ao final da simulação, as posições de Memória de Dados contiverem os valores iguais aos nomes usados, isso significa que o programa funcionou.

5.2.1 Simulação de Movimentação de Dados

A Simulação de Movimentação de Dados tem como objetivo validar o correto funcionamento das instruções `LDA`, `LDAi` e `STA`, bem como o uso de todos os Registradores da CPU. O código apresentado na Figura 45a foi desenvolvido para esse fim, realizando diversas operações de carregamento de dados da Memória para os Registradores (`LDA`), carregamento de valores imediatos (`LDAi`) e armazenamento do conteúdo dos Registradores de volta na Memória (`STA`).

Inicialmente, as três primeiras instruções utilizam o `LDA` para carregar valores previamente definidos na Memória. Em seguida, são executados pares de instruções `LDAi` e `STA`, garantindo que cada um dos oito Registradores seja utilizado ao menos uma vez. Para facilitar a verificação, todos os valores carregados nos Registradores são posteriormente gravados em posições específicas da Memória, identificadas por `labels` que indicam o valor esperado ao final da simulação (por exemplo, `v3p` para o valor 3 positivo e `v5n` para o valor 5 negativo). Note que foram testados tanto valores positivos quanto negativos, que devem ser representados em Complemento de 2 de 16 bits, incluindo os casos de maior e menor valor (32767 positivo e 32768 negativo).

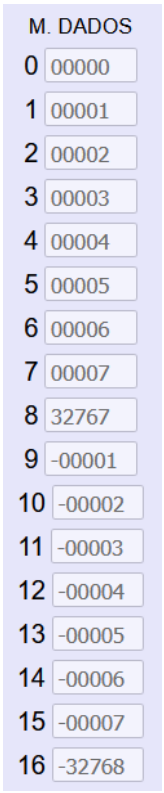
O resultado final pode ser observado na Figura 45b, que apresenta o conteúdo da Memória de Dados após a simulação do programa. A análise das figuras do Código e da Memória Final confirma que todos os valores foram corretamente armazenados, indicando que as instruções de Movimentação de Dados estão funcionando conforme o esperado.

Figura 45 – Teste de Movimentação de Dados

```
LDA  v0p, R0
LDA  v1p, R1
LDA  v1n, R2

LDAi  2, R2       STA  R2, v2p
LDAi  3, R3       STA  R3, v3p
LDAi  4, R4       STA  R4, v4p
LDAi  5, R5       STA  R5, v5p
LDAi  6, R6       STA  R6, v6p
LDAi  7, R7       STA  R7, v7p

LDAi -2, R2       STA  R2, v2n
LDAi -3, R3       STA  R3, v3n
LDAi -4, R4       STA  R4, v4n
LDAi -5, R5       STA  R5, v5n
LDAi -6, R6       STA  R6, v6n
LDAi -7, R7       STA  R7, v7n

LDAi  32767, R0   STA  R0, v32767p
LDAi -32768, R1   STA  R1, v32768n

HLT
```

```
v0p      0
v1p      1
v2p      0
v3p      0
v4p      0
v5p      0
v6p      0
v7p      0
v32767p  0
v1n     -1
v2n      0
v3n      0
v4n      0
v5n      0
v6n      0
v7n      0
v32768n  0
```

(a) Assembly de Entrada



(b) Mem. de Dados

Fonte: Elaborada pelo autor

5.2.2 Simulação de Instruções Aritméticas

A Simulação das Instruções Aritméticas tem como objetivo verificar o funcionamento correto das operações `ADD` e `SUB` no simulador. O código apresentado na Figura 46a foi elaborado para testar diferentes combinações de operandos, envolvendo valores positivos e negativos, a fim de garantir a correta geração dos resultados aritméticos. Para isso, em cada operação de soma ou subtração, o resultado é armazenado em uma posição específica da

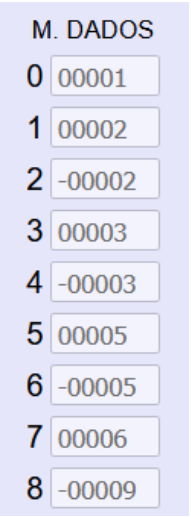
Memória de Dados por meio da instrução `STA`, sendo que o *label* dessa posição corresponde literalmente ao valor esperado ao final da execução.

Conforme indicado na Figura 46b, que apresenta o conteúdo da Memória de Dados após a simulação do programa, observa-se que todos os valores armazenados correspondem exatamente aos resultados esperados. Ressalta-se que, neste teste, foram utilizados apenas valores inteiros de pequena magnitude, uma vez que a verificação de *overflow* e também da Ativação das *flags* Aritméticas será tratada adiante, em programas específicos. Assim, a análise das figuras do Programa e da Memória Final confirma que os resultados obtidos são iguais aos valores esperados, validando o funcionamento das Instruções `ADD` e `SUB`.

Figura 46 – Teste de Instruções Aritméticas

```
LDAi  4, R4
LDAi  2, R2
ADD  R4, R2, R1    STA  R1, v6p
SUB  R4, R2, R1    STA  R1, v2p
SUB  R2, R4, R1    STA  R1, v2n
LDAi  3, R4
LDAi -2, R2
ADD  R4, R2, R1    STA  R1, v1p
SUB  R4, R2, R1    STA  R1, v5p
SUB  R2, R4, R1    STA  R1, v5n
LDAi -3, R4
LDAi -6, R2
ADD  R4, R2, R1    STA  R1, v9n
SUB  R4, R2, R1    STA  R1, v3p
SUB  R2, R4, R1    STA  R1, v3n
HLT
--------------------------------
v1p     0
V2p     0
v2n     0
V3p     0
v3n     0
v5p     0
v5n     0
V6p     0
v9n     0
```

(a) Assembly de Entrada



(b) Mem. de Dados

Fonte: Elaborada pelo autor

5.2.3 Simulação de Instruções Lógicas

A Simulação das Instruções Lógicas tem como objetivo validar o funcionamento correto das operações `AND`, `OR` e `NOT` no simulador. O código apresentado na Figura 47a foi desenvolvido seguindo a mesma estratégia adotada nos testes anteriores, realizando operações lógicas entre valores previamente carregados nos Registradores e armazenando

cada resultado em uma posição específica da Memória de Dados por meio da instrução `STA`. Os valores de entrada foram escolhidos como padrões de bits de 16 bits (`0000`, `1111`, `0101` e `1010`), de modo a facilitar a análise do comportamento das operações lógicas; entretanto, para simplificar a verificação, esses padrões são representados em decimal nos *Labels* escolhidos de Memória.

Conforme ilustrado na Figura 47b, que apresenta o conteúdo final da Memória de Dados após a simulação do programa, os resultados das operações `AND` e `OR` correspondem exatamente aos padrões binários esperados, assim como os resultados das instruções `NOT`, que geram o complemento bit-a-bit dos operandos de entrada. A comparação entre os valores armazenados na Memória e os *labels* correspondentes indica que todos os resultados obtidos são iguais aos valores esperados, confirmando o correto funcionamento das Instruções `AND`, `OR` e `NOT`.

Figura 47 – Teste de Instruções Lógicas

```
LDA   v255p, R0
LDA  v3855p, R1
LDA v23130p, R2

AND R0, R1, R3    STA R3, v15p
AND R0, R2, R4    STA R4, v90p

OR  R0, R1, R3    STA R3, v4095p
OR  R0, R2, R4    STA R4, v23295p

NOT R1, R3        STA R3, v3856n
NOT R2, R4        STA R4, v23131n
-------------------------------------
v255p      255      ; 0000 0000 1111 1111
v3855p    3855      ; 0000 1111 0000 1111
v23130p  23130      ; 0101 1010 0101 1010
v15p         0      ; 0000 0000 0000 1111
v90p         0      ; 0000 0000 0101 1010
v4095p       0      ; 0000 1111 1111 1111
v23295p      0      ; 0101 1010 1111 1111
v3856n       0      ; 1111 0000 1111 0000
v23131n      0      ; 1010 0101 1010 0101
```

(a) Assembly de Entrada

(b) Mem. de Dados

Fonte: Elaborada pelo autor

5.3 Testes de Desvios e *Flags*

Nesta seção são apresentados dois testes adicionais com foco no correto funcionamento do processador em situações de Controle de Fluxo e Sinalização de Estados internos da ULA. O primeiro teste é dedicado às Instruções de Desvio, verificando se os Saltos Condicionais e Incondicionais alteram corretamente o fluxo de execução do programa. O segundo teste tem como objetivo a verificação completa das *Flags* da ULA, assegurando que cada condição (`Zero`, `Negativo`, `Carry` e `Overflow`) seja corretamente ativada.

5.3.1 Simulação de Instruções de Desvios

A Simulação de Desvios tem como objetivo validar o funcionamento correto das Instruções de Salto, tanto Condicionais quanto Incondicionais, bem como a correta interpretação das *flags* Zero (`Z`) e Negativo (`N`). O código apresentado na Figura 48a foi construído de forma abrangente, a fim de testar todas as possibilidades de salto, o que pode resultar em um fluxo de execução aparentemente estranho ou pouco linear.

Nos testes com Saltos Condicionais (`JZ` e `JN`), o programa é organizado em grupos, nos quais: no primeiro caso nenhuma *flag* deve ser ativada; no segundo apenas `Z` deve ser ligada; e no terceiro apenas `N` deve ser ligada. Entre cada grupo de testes, é executada uma instrução de Salto Incondicional (`JMP`), garantindo a continuidade do fluxo de execução.

Caso qualquer uma das condições de desvio seja avaliada incorretamente, o programa salta imediatamente para o rótulo FIM, encerrando a execução com a variável OK permanecendo em Zero, o que indica falha no teste. Por outro lado, se todos os desvios forem corretamente tomados, o fluxo alcança o trecho TOK, no qual OK é setado para 1, indicando sucesso.

A verificação do acionamento das *flags* foi realizada visualmente, sendo possível observar na Figura 48b os momentos em que cada *flag* é ativada, lembrando que essas *flags* são setadas tanto nas instruções `LDA` e `LDAi`, quanto nas Instruções Aritméticas e Lógicas que usam a ULA. Conforme mostrado na figura do Programa e no conteúdo final da Memória, o valor armazenado em OK é igual a 1, confirmando que todos os saltos e condições de desvio foram corretamente executados.

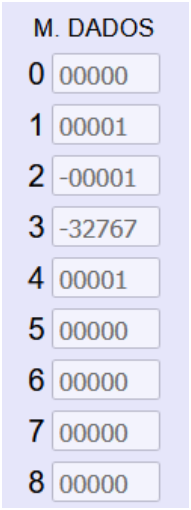
5.3.2 Simulação das *Flags* da ULA

A Simulação de *Flags* tem como objetivo verificar o comportamento correto das *flags Carry* (`C`) e *Overflow* (`V`) da ULA, complementando os testes anteriormente realizados para as *flags* `Z` e `N`. O código apresentado na Figura 49a foi elaborado para testar todas as possibilidades de ativação das *flags*, considerando diferentes Estudos de Caso.

Figura 48 – Teste de Instruções de Desvios

```
        LDAi  1, R1      JZ FIM   JN FIM
        LDAi  0, R2      JN FIM   JZ L01
L02     LDAi -1, R3      JZ FIM   JN L03
L06     ADD R1, R2, R4   JZ FIM   JN FIM
        ADD R1, R3, R5   JN FIM   JZ L07
L08     ADD R2, R3, R6   JZ FIM   JN L09
L03     LDA  v1p, R1     JZ FIM   JN FIM
        LDA  v0p, R2     JN FIM   JZ L04
L05     LDA  v1n, R3     JZ FIM   JN L06
L09     SUB R1, R2, R4   JZ FIM   JN FIM
        SUB R1, R4, R5   JN FIM   JZ L10
L11     SUB R2, R1, R6   JZ FIM   JN L12
L15     OR  R1, R2, R4   JZ FIM   JN FIM
        OR  R2, R5, R5   JN FIM   JZ L16
L17     OR  R1, R3, R6   JZ FIM   JN L18
L12     AND R1, R4, R4   JZ FIM   JN FIM
        AND R1, R2, R5   JN FIM   JZ L13
L14     AND R3, R6, R6   JZ FIM   JN L15
L18     LDA vX, R2
        NOT R2, R4       JZ FIM   JN FIM
        NOT R3, R5       JN FIM   JZ L19
L20     NOT R1, R6       JZ FIM   JN TOK
L01     JMP  L02
L04     JMP  L05
L07     JMP  L08
L10     JMP  L11
L13     JMP  L14
L16     JMP  L17
L19     JMP  L20
TOK     STA R1 OK
FIM     HLT
-----------------------------------
 v0p      0      ; 0000 0000 0000 0000
 v1p      1      ; 0000 0000 0000 0001
 v1n     -1      ; 1111 1111 1111 1111
 vX     -32767   ; 1000 0000 0000 0001
 OK       0
```

(a) Assembly de Entrada



(b) Mem. de Dados

Fonte: Elaborada pelo autor

Os Casos envolvem operações Aritméticas entre valores de mesmo sinal e de sinais opostos. Cada cenário foi construído de modo a provocar, ou não, a ativação de `C` e `V`, permitindo observar não apenas o seu acendimento, mas também a persistência do estado até que uma nova instrução altere seu valor.

Além disso, nos Casos Com *Overflow*, é esperado que o valor armazenado na Memória seja numericamente incorreto, refletindo o estouro de representação em Complemento de 2. Já nos Casos Sem *Overflow* o valor gravado permanece correto. Para facilitar a validação, os resultados são armazenados em posições de memória cujos *labels* indicam explicitamente o valor esperado, correto ou incorreto, ao final da execução.

Abaixo são apresentadas apenas as figuras dos estados relevantes, incluindo: (c) todas as *flags* desligadas; (d) *flag* Zero ligada; (e) *flag* Negativo ligada; (f) *flag Carry* ligada; e (g) *flag Overflow* ligada. A análise conjunta do Código e do conteúdo final da

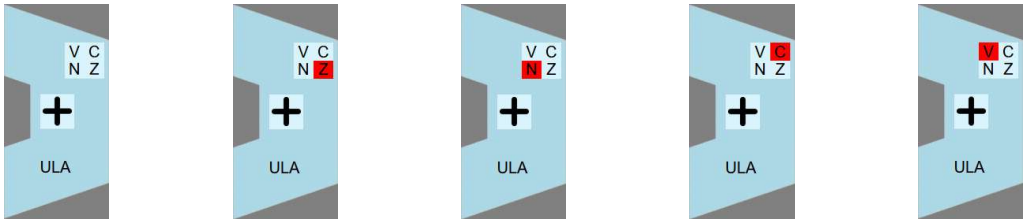
Memória da Figura 49b confirma que todas as *flags* foram corretamente ativadas nos cenários previstos, validando o funcionamento da ULA quanto às *flags*.

Figura 49 – Teste de Ativação de *Flags*

```
LDAi  32766, R1
LDAi -16400, R2
ADD  R1, R2, R1   STA R1, v16366p

LDAi  16400, R1
LDAi  16400, R2
ADD  R1, R2, R1   STA R1, v32736nV1
LDAi  32767, R2
ADD  R1, R2, R1   STA R1, v31pC1

LDAi -16400, R1
LDAi -16400, R2
ADD  R1, R2, R1   STA R1, v32736pV2
LDAi -16384, R2
ADD  R1, R2, R1   STA R1, v6352pC2
LDAi      1, R2
ADD  R1, R2, R1   STA R1, v32767p
ADD  R1, R2, R1   STA R1, v32768nV3

LDAi -32767, R1
LDAi     -1, R2
ADD  R1, R2, R1   STA R1, v32768n
ADD  R1, R2, R1   STA R1, v32767pV4
LDAi -15360, R2
ADD  R1, R2, R1   STA R1, v17407pC3
-------------------------------------
v16366p     0     ; 0011 1111 1110 1110
v32767p     0     ; 0111 1111 1111 1111
v32768n     0     ; 1000 0000 0000 0000
v32736nV1   0     ; 1000 0000 0010 0000   ; Aconteceu V
v32736pV2   0     ; 0111 1111 1110 0000   ; Aconteceu V
v32768nV3   0     ; 1000 0000 0000 0000   ; Aconteceu V
v32767pV4   0     ; 0111 1111 1111 1111   ; Aconteceu V
v31pC1      0     ; 0000 0000 0001 1111   ; Aconteceu C
v6352pC2    0     ; 0011 1111 1110 0000   ; Aconteceu C
v17407pC3   0     ; 0100 0011 1111 1111   ; Aconteceu C
```

(a) Assembly de Entrada

(b) Mem. de Dados



(c) F=0 (d) Z=1 (e) N=1 (f) C=1 (g) V=1

Fonte: Elaborada pelo autor

5.4 Testes de Programas Básicos

Nesta seção serão apresentados três programas distintos, cujo objetivo é demonstrar como Estruturas de Controle de alto nível podem ser implementadas utilizando exclusivamente as instruções disponibilizadas pela ISA desenvolvida. Inicialmente, são apresentados programas que simulam as estruturas FOR e IF-THEN-ELSE, evidenciando a capacidade da arquitetura de implementar laços de repetição e desvios condicionais. Posteriormente, é apresentado um programa de Multiplicação por Somas Sucessivas, uma vez que a ISA não possui uma instrução nativa de multiplicação.

5.4.1 Simulação Programa FOR

A Simulação do Código FOR tem como objetivo demonstrar como uma estrutura de repetição pode ser implementada utilizando apenas as instruções básicas da ISA, servindo como base para a construção de laços em geral. O programa apresentado na Figura 50a implementa um simples Contador, que pode ser diretamente associado ao comportamento de um FOR, sendo facilmente reutilizável sempre que uma repetição for necessária.

Figura 50 – Teste do Programa Contador

```
        LDA  CNT, R0
        LDA  END, R1
        LDAi   1, R2

CONTA   ADD  R0, R2, R0
        STA  R0, CNT

        SUB  R1, R0, R3
        JZ   FIM
        JMP  CONTA

FIM     HLT
--------------------------------
              ; TESTES
CNT    1      ;  1   0  -5  -7
END    5      ;  5   8  10  -1
```

(a) Assembly de Entrada

M. DADOS	
0	00005
1	00005
2	00000
3	00000
4	00000
5	00000
6	00000
7	00000
8	00000

(b) Mem. de Dados

Fonte: Elaborada pelo autor

O funcionamento do programa baseia-se nos valores de entrada `CNT` e `END`, onde `CNT` representa o valor inicial da contagem e `END` o valor final. A cada iteração do laço, o valor de CNT é incrementado em uma unidade por meio da instrução `ADD`, sendo posteriormente armazenado novamente na Memória. Em seguida, a instrução `SUB` é utilizada para comparar

CNT com END, e o desvio condicional JZ determina o encerramento do laço quando os valores se igualam. Caso contrário, o fluxo retorna ao início da repetição. Esse mesmo código pode ser adaptado para repetir qualquer conjunto de instruções, bastando inserir o trecho desejado antes da instrução SUB e configurar os valores de CNT e END.

Cabe destacar que o programa foi simulado com diversos conjuntos de valores de entrada, incluindo casos com valores positivos e negativos, e em todos os testes em que CNT < END o comportamento observado foi o esperado. O incremento correto do Contador até o valor final, pode ser observado na Figura 50b do conteúdo da Memória, para o caso da Contagem de 1 a 5. Dessa forma, conclui-se que a simulação funcionou corretamente para todas as entradas previstas, respeitando a limitação inerente de funcionamento apenas quando CNT é menor que END.

5.4.2 Simulação Programa IF-THEN-ELSE

A Simulação de um Código IF-THEN-ELSE tem como objetivo demonstrar como estruturas condicionais podem ser implementadas utilizando as instruções disponibilizadas pela ISA. O programa apresentado na Figura 51a emula o comportamento de uma estrutura de teste tradicional, podendo ser reutilizado ou adaptado para qualquer outro código condicional, bastando ajustar as comparações e os desvios envolvidos.

No código, dois valores de entrada, N1 e N2, são carregados nos registradores e comparados por meio da instrução SUB, cujo resultado é avaliado pelas *flags* Zero (Z) e Negativo (N). A partir dessa avaliação, três cenários distintos são possíveis: se N1<N2 a variável N1menorN2 é setada para 1; se N1>N2 a variável N1maiorN2 recebe 1; e caso N1=N2 a variável N1igualN2 é marcada com 1. O desvio para cada um desses casos é realizado pelas instruções JZ e JN, que direcionam o código para o bloco correspondente.

O programa foi simulado com diversos conjuntos de valores, incluindo combinações com números positivos, negativos e iguais, conforme indicado nos testes apresentados. A Figura 51b do conteúdo final da Memória confirmam que, no cenário analisado para valores 3 e 4, o desvio tomado corresponde ao resultado esperado da comparação. De forma análoga, todos demais conjuntos de entrada testados também produziram desvios corretos, validando o funcionamento da Estrutura IF-THEN-ELSE implementada.

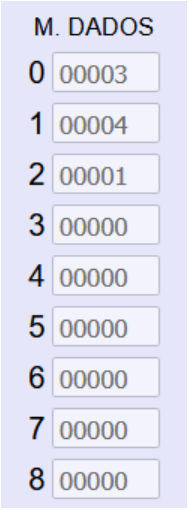
5.4.3 Simulação Programa Multiplicação

A Simulação do Programa de Multiplicação tem como objetivo demonstrar como uma operação de multiplicação pode ser implementada mesmo na ausência de uma instrução específica na ISA, que dispõe apenas das operações aritméticas ADD e SUB.

Figura 51 – Teste do Programa Comparador

```
        LDA  N1, R1
        LDA  N2, R2
        LDAi  1, R3

TESTA   SUB  R2, R1, R0
        JZ   CASO3
        JN   CASO2

CASO1   STA  R3, N1menorN2
        JMP  FIM

CASO2   STA  R3, N1maiorN2
        JMP  FIM

CASO3   STA  R3, N1igualN2

FIM     HLT
-----------------------------------
                ; TESTES
N1          3   ; 3  4 -3 -4  5  3
N2          4   ; 4  3 -4 -3  5 -4
N1menorN2   0
N1maiorN2   0
N1igualN2   0
```

(a) Assembly de Entrada



(b) Mem. de Dados

Fonte: Elaborada pelo autor

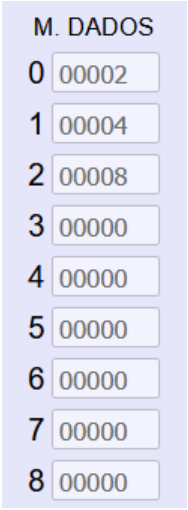
O código apresentado na Figura 52a realiza a multiplicação utilizando o método das Somas Sucessivas, no qual um dos operandos é somado repetidamente ao acumulador até que o número de repetições corresponda ao valor do outro operando. Para tornar o algoritmo mais eficiente, o programa compara inicialmente os valores de entrada `N1` e `N2`, escolhendo o menor deles como contador e o maior como valor a ser somado, garantindo assim o menor número possível de somas durante a execução.

Ressalta-se que esta implementação considera apenas valores naturais (inteiros positivos), embora o algoritmo possa ser estendido para tratar sinais, uma versão mais completa não foi desenvolvida devido às limitações de tempo. O programa foi testado com diversos conjuntos de valores de entrada, incluindo os exemplos apresentados, e a Figura 52b da simulação demonstra corretamente o resultado para o cenário com valores 2 e 4. Além disso, foi verificado que todas as demais simulações realizadas também produziram os resultados esperados, confirmando o funcionamento correto da multiplicação por somas sucessivas dentro das restrições da ISA.

Figura 52 – Teste do Programa Multiplicação

```
SEZERO    LDA  N1, R1     JZ  FIM
          LDA  N2, R2     JZ  FIM

          LDAi  1, R3
          LDA  RE, R4

TESTA     SUB  R1, R2, R7
          JN   MULTIP

INVERTE   LDA  N1, R2
          LDA  N2, R1

MULTIP    ADD  R4, R2, R4
          STA  R4, RE
          SUB  R1, R3, R1
          JZ   FIM
          JMP  MULTIP

FIM       HLT
-----------------------------------
                ; TESTES
N1        2     ; 2  4  0  2
N2        4     ; 4  2  4  0
RE        0
```

(a) Assembly de Entrada



(b) Mem. de Dados

Fonte: Elaborada pelo autor

As simulações realizadas contemplaram instruções de Movimentação de Dados, Operações Aritméticas e Lógicas, Desvios Condicionais e Incondicionais, além da verificação da Ativação de *flags*. A implementação de estruturas de controle e programas mais complexos também apresentaram os comportamentos esperados, como laços de repetição e multiplicação por somas sucessivas, sem que fossem identificados erros conceituais ou funcionais nos fluxos testados.

Após a realização dos testes, pode-se afirmar que os objetivos propostos foram atingidos de forma satisfatória. Todos os testes realizados ao longo das simulações indicam que a arquitetura implementada está funcional, com movimentação correta das informações pelo Caminho de Dados, execução adequada das instruções da ISA e comportamento coerente da Unidade de Controle e da ULA.

6 Conclusão

Esta seção reúne uma análise crítica dos resultados obtidos e das contribuições alcançadas ao longo do desenvolvimento do Simulador. Inicialmente, são discutidas as principais contribuições do trabalho, indicando quais objetivos foram totalmente ou parcialmente atingidos, bem como as limitações encontradas. Em seguida, é apresentada a conclusão geral, abordando os desafios enfrentados, os aprendizados adquiridos e as competências desenvolvidas ao longo do trabalho. Por fim, são discutidos trabalhos futuros, apontando possíveis melhorias e funcionalidades que podem ser incorporadas ao Simulador.

6.1 Contribuições

O principal objetivo deste trabalho foi o desenvolvimento de um Simulador de CPU, contemplando a visualização completa do seu Caminho de Dados, incluindo ULA, Registradores e Memórias, a partir de uma ISA própria. Esse objetivo foi totalmente atingido, uma vez que todos os componentes previstos foram implementados e integrados de forma funcional, permitindo a simulação de programas *Assembly*. A observação do Fluxo de Dados entre blocos e dos Sinais de Controle nas simulações se mostrou eficiente.

Outro objetivo consistiu em validar o funcionamento do Simulador por meio de testes, abrangendo instruções de Movimentação de Dados, Operações Aritméticas e Lógicas, Desvios Condicionais e Incondicionais, Verificação da Ativação das *Flags* da ULA. Também foram simulados programas mais complexos, como laços de repetição, estruturas condicionais e multiplicação via somas sucessivas. Esse objetivo também foi atingido, visto que todos os testes realizados apresentaram resultados coerentes com o comportamento esperado, validando a correta movimentação das informações e a execução das instruções.

O terceiro objetivo foi projetar uma Interface Gráfica com foco Educacional, capaz de auxiliar no entendimento do funcionamento interno da CPU e de servir como ferramenta de apoio ao Ensino de Arquitetura e Organização de Computadores. Esse objetivo foi parcialmente atingido, pois a ferramenta não foi submetida a testes com usuários finais. A validação junto a discentes não pôde ser realizada em função do tempo disponível para o desenvolvimento do projeto.

Por fim, destaca-se que não foram implementadas algumas funcionalidades previstas ou desejáveis, como a validação do Simulador com usuários, e a elaboração de um manual detalhado de uso. Essas atividades não foram realizadas principalmente devido a restrições de tempo, sendo indicadas como oportunidades de evolução em estudos futuros.

6.2 Conclusão

Um dos principais desafios enfrentados ao longo deste trabalho foi a complexidade do projeto e à implementação de uma arquitetura de computador completa, mesmo em um contexto educacional. A definição da ISA, o correto sequenciamento dos sinais de controle, a garantia da coerência no fluxo de informações entre os blocos e a depuração de erros sutis exigiram um processo iterativo constante de testes e refinamentos. Mesmo com um planejamento antecipado, ao longo do desenvolvimento, algumas decisões precisaram ser revistas e a soluções precisou ser ajustada, o que impactou diretamente no cronograma de desenvolvimento. Além disso, a necessidade de conciliar o trabalho com os prazos acadêmicos disponíveis impôs limitações quanto à implementação de funcionalidades adicionais e à realização de validações mais amplas, especialmente no que se refere à interface gráfica.

A realização deste projeto proporcionou um aprendizado significativo tanto do ponto de vista técnico quanto metodológico. Ao longo do desenvolvimento, foi possível aprofundar conhecimentos em Arquitetura e Organização de Computadores, projeto de Caminho de Dados, funcionamento de Unidades de Controle e execução de instruções em baixo nível. Também foram desenvolvidas habilidades em Engenharia de Software, como Modelagem, Testes Sistemáticos, Análise de Erros, além do aprimoramento da capacidade de transformar conceitos teóricos em uma Ferramenta prática e funcional.

O Simulador desenvolvido está disponível e encontra-se funcional, sendo capaz de simular corretamente os programas de teste propostos, com resultados consistentes e coerentes com a ISA definida. As movimentações de informações, a ativação dos Sinais de Controle e o fluxo de execução das instruções foram verificados por meio de simulações passo-a-passo e análise do estado final da Memória. Entretanto, embora funcione corretamente do ponto de vista técnico, o Simulador ainda não foi validado com discentes. Isso impede que ser tenha conclusões mais aprofundadas sobre aspectos de usabilidade, clareza visual, experiência de uso, e efetividade pedagógica, como desejado.

6.3 Trabalhos Futuros

Embora o Simulador tenha apresentado funcionamento correto nos cenários de teste realizados, testes mais extensivos ainda devem ser conduzidos, especialmente com programas mais longos e com combinações variadas de instruções, a fim de aumentar a confiabilidade da ferramenta. Até o momento, não foram identificados erros críticos de funcionamento, porém isso não elimina a possibilidade de falhas em casos específicos ainda não explorados. Assim, uma etapa futura importante consiste na ampliação do conjunto de testes, bem como na correção de eventuais erros encontrados.

Durante o desenvolvimento, algumas funcionalidades planejadas não puderam ser implementadas dentro do tempo disponível. Entre elas, destaca-se a adaptação completa da interface para dispositivos móveis, tornando o simulador responsivo. Também não foi possível desenvolver um Manual do Usuário, o que facilitaria o uso da ferramenta por novos usuários, especialmente estudantes com pouco contato prévio com Arquitetura e Organização de Computadores. Também o *Help* da ferramenta precisa ser melhorado visto que se encontra em uma versão resumida.

Além disso, podem ser incorporadas ao simulador funcionalidades voltadas à acessibilidade, como ajustes de cores, suporte a diferentes tipos e tamanhos de fonte, recursos de zoom e mecanismos de áudio, incluindo a narração dos *logs* da simulação. Estas funcionalidades são desejáveis devido ao seu potencial de tornar o Simulador adequado às necessidades de diferentes usuários.

Mesmo considerando que o Simulador atual atenda aos objetivos propostos, há amplo espaço para melhorias futuras. Uma possível evolução consiste na implementação de uma versão com Unidade de Controle Microprogramada. Outra melhoria relevante seria a visualização da lógica interna dos blocos funcionais e da Unidade de Controle, com uso de destaques visuais e zoom. Essas melhorias podem facilitar o acompanhamento do fluxo interno de sinais e dados, ampliando o valor didático do simulador e sua aplicabilidade em contextos educacionais mais avançados.

Referências

CS61C Staff. *Venus MIPS Simulator*. 2019. University of California, Berkeley. Disponível em: <https://venus.cs61c.org>. Citado 3 vezes nas páginas 16, 28 e 29.

David Patterson; John Hennessy. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. 2020. Citado 2 vezes nas páginas 25 e 26.

DAVIS, B. *MIPS 101: A Visual MIPS Processor Simulator*. 2018. <https://www3.ntu.edu.sg/home/smitha/fyp_gerald/index.html>. Citado 3 vezes nas páginas 16, 31 e 32.

Emulator.IO Team. *Online x86 Assembly Emulator*. 2014. Disponível em: <https://carlosrafaelgn.com.br/Asm86/>. Citado 3 vezes nas páginas 16, 29 e 30.

GRIEBLER, D. J. *Computação Paralela em Sistemas Embarcados*. 2010. Citado na página 13.

HENNESSY, D. P. J. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. 6. ed. [S.l.]: Morgan Kaufmann Publishers, 2020. Citado 2 vezes nas páginas 14 e 23.

MANO, M. M.; KIME, C. R. *Logic and Computer Design Fundamentals*. 3. ed. [S.l.]: Pearson, 2008. Citado na página 14.

MIPS Datapath Dev Team. *MIPS-Datapath Simulator*. 2015. <https://mi.eng.cam.ac.uk/~ahg/MIPS-Datapath/>. Citado 2 vezes nas páginas 16 e 32.

OneCompiler Team. *OneCompiler - Online Compiler and Interpreter*. 2019. Disponível em: <https://onecompiler.com/assembly>. Citado 3 vezes nas páginas 16, 29 e 30.

PEDERSEN, T. B. *Ripes: A Visual Computer Architecture Simulator*. 2018. <https://ripes.me/>. Citado 3 vezes nas páginas 16, 30 e 31.

SILVIUS, T. *Simple 8-bit Assembler Simulator*. 2014. Ferris State University. Disponível em: <https://schweigi.github.io/assembler-simulator/>. Citado 3 vezes nas páginas 16, 27 e 28.

TANENBAUM, A. S.; AUSTIN, T. *Structured Computer Organization*. 6. ed. [S.l.]: Pearson, 2013. Citado na página 14.

Universidade Federal de Pernambuco. *Infraestrutura de Hardware Revisão 1a Unidade*. s.d. <https://www.cin.ufpe.br/~if674cc/aulas/AulaInfraHW-RevisaoI.pdf>. Citado na página 23.

WebRISC-V Project. *WebRISC-V - Graphical RISC-V Pipeline Simulator*. 2022. <https://webriscv.dii.unisi.it/index.php>. Citado 2 vezes nas páginas 16 e 33.