

Diogo Valadares Reis dos Santos

DRISC-V: Uma Arquitetura Didática Integrando RISC-V Sobre RISC I

Campos dos Goytacazes, RJ

2024

Diogo Valadares Reis dos Santos

DRISC-V: Uma Arquitetura Didática Integrando RISC-V Sobre RISC I

Monografia apresentada ao Instituto Federal de Educação, Ciência e Tecnologia Fluminense como parte das exigências para a conclusão do Curso Bacharelado em Engenharia da Computação.

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA FLUMINENSE

Engenharia da Computação

Orientador: David Vasconcelos Corrêa da Silva

Campos dos Goytacazes, RJ

2024

Listas de ilustrações

Figura 1 – A Máquina de von Neumann original(TANENBAUM; AUSTIN, 2013, p.18)	13
Figura 2 – Um computador de 6 níveis. O método suportado para cada nível é indicado abaixo dele(junto com o programa que suporta). (TANENBAUM; AUSTIN, 2013, Adaptação p.5)	14
Figura 3 – Planejamento Monografia	28
Figura 4 – Modelo de simulação no Logisim Evolution.	35
Figura 5 – Resultado das pesquisas no <i>Google Ngram Viewer</i> (VALADARES, 2025)	39
Figura 6 – Exemplo da impressão de uma instrução realizada pelo <i>assembler</i>	51
Figura 7 – Arquivo de Registradores antes (esquerda) e depois (direita) da modificação.	54
Figura 8 – Encapsulamento do Arquivo de Registradores antes (esquerda) e depois (direita) da modificação.	54
Figura 9 – Primeira fase de decodificação de instruções. Antes (esquerda) e depois (direita).	55
Figura 10 – Segunda fase de decodificação de instruções. Antes (esquerda) e depois (direita).	56
Figura 11 – Formato de instrução do RISC-V para operações entre registradores (RISC-V, 2024b)	56
Figura 12 – Decodificador de imediato do RISC I(esquerda) em comparação com o novo decodificador(direita)	57
Figura 13 – Decodificador de condição antigo(esquerda) e novo(direita)	57
Figura 14 – Modelo original com modificações que foram executadas destacadas em amarelo, componentes que foram removidos cortados em vermelho e posições de novos componentes destacados em verde	58
Figura 15 – Resultado da primeira fase de modificações da arquitetura	59
Figura 16 – Temporização do RISC I (adaptado de Peek (1983))	60
Figura 17 – Demonstração da <i>pipeline</i> do RISC I.(STALLINGS, 1988)	60
Figura 18 – Demonstração da <i>pipeline</i> do RISC II.(STALLINGS, 1988)	61
Figura 19 – Primeira fase de decodificação durante a segunda fase de desenvolvimento	61
Figura 20 – Comparação entre o Decodificador de Imediato com <i>buffers tri-state</i> (esquerda) e multiplexadores (direita)	62
Figura 21 – Segunda fase de decodificação de instruções. Antes (esquerda) Depois (direita)	62
Figura 22 – Atualização no visual dos componentes	63
Figura 23 – Visualização dos componentes e suas conexões com os barramentos	64

Figura 24 – Comparação entre o Deslocador (acima) e o DataIO (abaixo) antes e depois	64
Figura 25 – Contador de Programa antes (esquerda) e depois (direita)	65
Figura 26 – Separação dos barramentos de dados e endereços	66
Figura 27 – Início da simulação do código SystemVerilog, com exibição simplificada do estado do sistema	67
Figura 28 – Fim da simulação do código SystemVerilog	68
Figura 29 – Contador em Anel, implementado com um Registrador de Deslocamento de 3 bits	68
Figura 30 – Dispositivos de entrada e saída do Logisim utilizados	69
Figura 31 – Seletor de Endereços	70
Figura 32 – Enter Caption	70
Figura 33 – Resultado da remoção de registradores de entrada, do Deslocador e da ULA de endereços	72
Figura 34 – Programa para interação com a simulação SV, denominado DRISC-V Interface	73
Figura 35 – Interface atualizada para interação com os dados do terminal	74
Figura 36 – Alteração no momento de atualização do registrador de instrução atual (<i>current_instruction</i>)	75
Figura 37 – Circuito do Controlador de CSR.	82
Figura 38 – Circuitos de fase, instrução atual e instrução decodificada.	82
Figura 39 – Representação simplificada do circuito de armazenamento e manipulação de CSRs.	83
Figura 40 – Circuitos de tratamento de <i>traps</i>	84
Figura 41 – Circuito de verificação de CSRs e controle de sinais de pulo e carregamento.	84
Figura 42 – Encapsulamento do Controlador de CSR	85
Figura 43 – Integração do Controlador de CSR na arquitetura	85
Figura 44 – Circuito do Dispositivo de Tempo Real	86
Figura 45 – Encapsulamento do Dispositivo de Tempo Real	87
Figura 46 – Registrador de Interrupção de Software	87
Figura 47 – Controlador de Operações com <i>Pipeline</i> Corrigida.	89
Figura 48 – Controlador de Operações com <i>Pipeline</i> Corrigida.	90
Figura 49 – Saída do programa de teste com melhorias na impressão.	92
Figura 50 – Formatos de instrução base do RISC-V (RISC-V, 2024b).	93
Figura 51 – Exemplo de código com destaque de sintaxe no Notepad++.	95
Figura 52 – Visualização da montagem de um programa de ordenação por inserção	97

Lista de tabelas

Tabela 1 – Arquiteturas encontradas na primeira fase de pesquisa.	33
Tabela 2 – Resultado das pesquisas do nome da arquitetura com termos adicionais para filtragem no <i>Google Scholar</i> (VALADARES, 2025)	39
Tabela 3 – Resultado relativo das pesquisas do nome da arquitetura com termos adicionais para filtragem no <i>Google Scholar</i> (VALADARES, 2025)	39
Tabela 4 – Lista de possíveis recursos e melhorias para implementação no <i>assembler</i>	41
Tabela 5 – Lista de possíveis recursos e melhorias para implementação no <i>assembler</i>	42
Tabela 6 – Nova <i>Pipeline</i> de 3 passos e 2 fases	76
Tabela 7 – Registradores de Controle e Estado implementados na arquitetura.	77
Tabela 8 – Exemplos de declarações de variáveis e seus valores em compilados	94

Lista de abreviaturas e siglas

DRISC	<i>Didactic Reduced Instruction Set Computer</i> (Computador Didático com Conjunto de Instruções Reduzido)
PC	<i>Program Counter</i> (Contador de Programa)
ULA	Unidade Lógica e Aritmética
CPU	<i>Central Processing Unit</i> (Unidade Central de Processamento)
FPU	<i>Floating-Point Processing Unit</i> (Unidade de Processamento de Pontos Flutuantes)
GPU	<i>Graphics Processing Unit</i> (Unidade de Processamento Gráfico)
TPU	<i>Tensor Processing Unit</i> (Unidade de Processamento de Tensores)
RAM	<i>Random Access Memory</i> (Memória de Acesso Aleatório)
ROM	<i>Read-Only Memory</i> (Memória Somente Leitura)
RISC	<i>Reduced Instruction Set Computer</i> (Computador com Conjunto de Instruções Reduzido)
CISC	<i>Complex Instruction Set Computer</i> (Computador com Conjunto de Instruções Complexo)
IDE	<i>Integrated Development Environment</i> (Ambiente de Desenvolvimento Integrado)
BNF	<i>Backus-Naur Form</i> (Forma de Backus-Naur)
DLX	<i>DeLuXe</i>
SAP	<i>Simple As Possible</i> (O mais simples possível)
MIPS	<i>Microprocessor without Interlocked Pipelined Stages</i> (Microprocessador sem estágios de <i>Pipeline</i> interligados)
FPGA	<i>Field Programmable Gate Array</i> (Matriz de portas programáveis em campo)
ISA	<i>Instruction Set Architecture</i> (Arquitetura do Conjunto de Instruções)
HDL	<i>Hardware Description Language</i> (Linguagem de Descrição de Hardware)

UART	<i>Universal Asynchronous Receiver-Transmitter</i> (Receptor-Transmissor Asíncrono Universal)
SPI	<i>Serial Peripheral Interface</i> (Interface Periférica Serial)
I ² C	<i>Inter-Integrated Circuit</i> (Circuito Inter-Integrado)
ALU	<i>Arithmetic Logic Unit</i> (Unidade Lógica e Aritmética)
HDL	<i>Hardware Description Language</i> (Linguagem de Descrição de Hardware)
VHDL	<i>VHSIC Hardware Description Language</i> (Linguagem de Descrição de Hardware VHSIC)
SV	SystemVerilog

Sumário

1	INTRODUÇÃO	10
1.1	Problema e Contexto	10
1.2	Objetivos	11
1.3	Justificativa	12
2	FUNDAMENTAÇÃO TEÓRICA	13
2.1	Arquitetura e Organização de Computadores	13
2.1.1	A Máquina de von Neumann	13
2.1.2	Características de Arquiteturas de Computadores	13
2.1.2.1	Níveis de Máquina	13
2.1.2.2	Barramentos	15
2.1.2.3	<i>Pipeline</i>	15
2.1.2.4	Conjunto de instruções	16
2.1.2.4.1	Modos de Operação	16
2.1.2.5	<i>Assembly</i>	17
2.2	Partes de uma Microarquitetura	17
2.2.1	Unidade Lógica e Aritmética(ULA)	17
2.2.1.1	IEEE 754	18
2.2.2	Memória	18
2.2.2.1	Registradores	19
2.2.2.2	Memória <i>Cache</i>	19
2.2.2.2.1	Arquitetura de Caches	20
2.2.2.3	Arquivo de Registradores	20
2.2.2.4	Modos de Endereçamento	21
2.2.3	Unidade de Controle	22
2.3	Arquitetura Berkeley RISC	22
2.3.1	RISC-V	23
2.4	CISC versus RISC	24
2.5	Simuladores	25
2.5.1	Linguagens de Descrição de Hardware (HDLs)	25
3	MÉTODOS E RECURSOS	26
3.1	Visão geral	26
3.2	Ferramentas Utilizadas	28
3.2.1	Simulador	29
3.2.2	IDE e Linguagem de Programação	29

3.2.3	Documentação	29
3.2.4	Controle de Versão	29
3.2.5	Assistentes Virtuais	29
4	TRABALHOS REALIZADOS	31
4.1	Pesquisa de Arquiteturas	31
4.1.1	Critérios da Pesquisa	31
4.1.2	Metodologia de Pesquisa	32
4.1.3	Trabalhos Encontrados	33
4.1.4	Motivações da Escolha da arquitetura RISC I	34
4.2	Modelo de Simulação Base	35
4.3	Assembler Base	36
4.4	Relatório Técnico	37
4.5	Bibliometria sobre arquiteturas	38
4.6	Listagem de Recursos Para Implementação	41
4.6.1	Recursos Para a Arquitetura	42
4.6.1.1	Remoção das Janelas de Registradores	42
4.6.1.2	Formato de Instrução Próximo ao RISC-V	42
4.6.1.3	Reorganização das Instruções	43
4.6.1.4	Padronização de Nomes	44
4.6.1.5	Padronização dos sinais de ativação entre componentes	44
4.6.1.6	Modo <i>Kernel</i> /Privilegiado	44
4.6.1.7	Novo Sistema de Interrupções e <i>Traps</i>	44
4.6.1.8	<i>Pipeline Flush</i>	45
4.6.1.9	Melhoria na Entrada e Saída de Informações para o Usuário	45
4.6.1.10	Mover os registradores dos próximos endereços do Arquivo de Registradores para o controlador	45
4.6.1.11	Multiplicação e Divisão	46
4.6.1.12	Pontos Flutuantes	46
4.6.1.13	<i>Pipeline</i> de 3 passos	46
4.6.1.14	Encaminhamento de dados para Instruções de <i>load</i>	46
4.6.1.15	<i>Pipeline</i> de 5 passos	47
4.6.1.16	Instruções <i>branch</i>	47
4.6.1.17	Predição de pulo	48
4.6.1.18	Conversão para HDL	48
4.6.1.19	Tornar compatível com FPGA	48
4.6.1.20	Interface de Expansão	49
4.6.1.21	<i>Multithreading</i>	49
4.6.1.22	Memória <i>Cache</i>	50
4.6.1.23	Paginação de Memória	50

4.6.2	Recursos Para a Assembler	51
4.6.2.1	Omissão de parâmetro	51
4.6.2.2	Pseudo-Instruções	51
4.6.2.3	<i>Debug Log</i> mais detalhado	51
4.6.2.4	Destaque de Sintaxe	52
4.6.2.5	Macros	52
4.6.2.6	Define	53
4.7	Implementação dos Recursos e Melhorias	53
4.7.1	Processo de Desenvolvimento - Arquitetura	53
4.7.1.1	Primeira Fase: Alterações Iniciais	53
4.7.1.2	Segunda Fase: <i>Pipeline</i> e Usabilidade	59
4.7.1.3	Terceira Fase: <i>SystemVerilog</i>	66
4.7.1.4	Quarta Fase: Entrada e Saída	69
4.7.1.5	Quinta Fase: Controlador de Registradores de Controle e Status	74
4.7.1.6	Sexta Fase: Pipeline	87
4.7.2	Processo de Desenvolvimento - Assembler	92
4.7.2.1	Adaptação para RISC-V	92
4.7.2.2	Variáveis	94
4.7.2.3	Sintaxe	95
4.7.2.4	Pré-processamento	96
4.7.2.5	Processo de montagem	97
5	RESULTADOS	99
6	CONCLUSÃO	100
	Considerações finais	101
	REFERÊNCIAS	102
	APÊNDICES	106
	APÊNDICE A – ARQUITETURA EM SYSTEMVERILOG - PRIMEIRA VERSÃO	107
B	- TESTE DE ENTRADA E SAÍDA	108
	ANEXOS	111

1 Introdução

1.1 Problema e Contexto

A Organização e Arquitetura de computadores é uma disciplina que explica o funcionamento de diversos componentes internos dos computadores, como estes se comunicam e como ocorre a execução de programas em baixo nível.(SOARES, 2015)(FÁVERO, 2011, p.12)(LOURENÇO; MIDORIKAWA, 2005)

Essa compreensão é fundamental para os alunos de computação, que frequentemente encontram desafios durante a introdução de conceitos de programação devido à falta de entendimento das abstrações físicas envolvidas. No entanto, esses conceitos costumam ser assimilados de forma mais completa à medida que os estudantes avançam para a disciplina de Arquitetura de Computadores.(RAABE; ZEFERINO, 2006)

Segundo Clements (2010), o conhecimento desta disciplina tem uma grande importância no entendimento de outras áreas dentro de cursos de computação. Ao usar uma arquitetura de processador específica, os estudantes têm uma compreensão melhor de problemas no mundo real.

Porém, como afirmado por Djordjevic, Milenkovic e Grbanovic (2000) um desafio significativo no ensino dessas arquiteturas é a dificuldade de estabelecer uma conexão cognitiva entre o conhecimento teórico e a aplicação prática. Isso se agrava porque há situações em que os modelos das operações internas das arquiteturas apresentados aos alunos frequentemente estão errados ou incompletos.(YEHEZKEL et al., 2001, p.60)

Alguns professores optam por utilizar algumas arquiteturas comerciais, mas Clements (1999, p.9)(2010) diz que a utilização de algumas destas arquiteturas no ensino pode complicar o trabalho do professor, pois elas geralmente são mais complexas ao tentar maximizar a entrada no mercado e o lucro das empresas que as criaram.

É possível a utilização de arquiteturas hipotéticas didáticas, mas isso pode desmotivar os alunos, que acabam sentindo que o uso destas arquiteturas não reflete problemas encontrados no mundo real. (CLEMENTS, 2010)

Uma boa solução seria a utilização de uma arquitetura didática baseada em uma arquitetura comercial, como realizado na arquitetura DLX (PATTERSON; HENNESSY, 1996, p. 96-108) que foi derivada sobre a arquitetura comercial MIPS, além de tirar inspiração de várias outras. Com uma arquitetura criada desta forma, mantém-se a vantagem de ser feita para o ensino, sendo mais simples, porém se mantendo realista ao usar como base arquiteturas reais.

O constante aumento de conteúdo na área de computação também se torna um obstáculo para o ensino. Uma possível solução é o aumentar o grau de abstração do conteúdo ensinado, no entanto, esta abstração só pode ser executada até certo nível antes que haja um prejuízo na compreensão e aprendizado dos estudantes. (WOLFFE et al., 2002, p.176)

Uma boa opção para o ensino é a utilização de simuladores, pois isso permite que os estudantes visualizem os eventos que ocorrem durante a execução de um programa em diferentes níveis de abstração. Os estudantes também conseguem criar seus próprios circuitos e arquiteturas através destes simuladores, ajudando o entendimento da operação de máquinas reais.(WOLFFE et al., 2002, p.176-177)

1.2 Objetivos

O objetivo geral deste trabalho é implementar uma arquitetura computacional didática voltada para alunos de cursos superiores na área de computação, fazendo uso de simuladores. Essa arquitetura abrange desde o planejamento do circuito base até a criação de um *assembler*. O propósito dessa implementação é fornecer um conteúdo educacional detalhado para auxiliar nos estudos, incluindo uma documentação completa da arquitetura e do processo de desenvolvimento.

Para alcançar esse objetivo é necessário completar os seguintes objetivos específicos:

- O1-** Catalogar arquiteturas já existentes que possuam baixa complexidade e que se aproximem de arquiteturas atuais.
- O2-** Executar uma bibliometria sobre uma arquitetura previamente escolhida em comparação a outras.
- O3-** Criar e disponibilizar um modelo de simulação de uma arquitetura, utilizando como base os trabalhos encontrados.
- O4-** Aprimorar o modelo de simulação adicionando recursos de interesse e priorizando a clareza e a compreensão didática.
- O5-** Criar um *assembler* para a compilação de programas específicos para a arquitetura implementada no modelo de simulação.
- O6-** Elaborar uma documentação didática que abranja todos os conceitos principais e os passos de criação da arquitetura.

1.3 Justificativa

Ao final deste trabalho, espera-se obter uma arquitetura e seu modelo de simulação completo e funcional, que seja não só fácil de entender, mas que também sirva como uma ferramenta educacional robusta.

A arquitetura desenvolvida integrará recursos relevantes para o mercado atual, proporcionando aos alunos acesso a conteúdos atualizados. Além disso, será simplificada para facilitar a compreensão. O modelo de simulação proposto permitirá que os alunos experimentem e compreendam conceitos avançados de arquitetura de computadores em um ambiente interativo, controlado e acessível.

A documentação detalhada e as ferramentas desenvolvidas servirão como um guia prático para professores e alunos, proporcionando um recurso valioso para o ensino e a aprendizagem da Arquitetura de Computadores.

Além disso, será crucial considerar a usabilidade na apresentação da arquitetura e documentação aos alunos. Se essa usabilidade for aplicada de forma adequada, a experiência de aprendizado e a motivação dos estudantes podem ser melhoradas, tornando o processo mais fácil e afetando positivamente seu progresso.(DIMITRIJEVIC; DEVEDZIC, 2020)

Portanto, uma arquitetura computacional didática bem documentada e com boa usabilidade tem valor como ferramenta de ensino em Arquitetura de Computadores. Este projeto visa proporcionar uma compreensão clara e prática dos conceitos fundamentais, preparando os alunos para suas futuras carreiras na área de computação.

2 Fundamentação teórica

2.1 Arquitetura e Organização de Computadores

Tanenbaum e Austin (2013, p.18) diz que o conjunto de tipos de dados, operações e recursos em todos os níveis de abstração de um computador é chamada arquitetura. A arquitetura lida com os aspectos visíveis ao usuário presente em certo nível. Aspectos de implementação como tecnologia utilizada para a produção do chip não faz parte da arquitetura. O estudo de como desenvolver as partes de um sistema computacional que são visíveis aos programadores é chamado Arquitetura e Organização de computadores

2.1.1 A Máquina de von Neumann

Segundo Tanenbaum e Austin (2013, p.18), atualmente, a Máquina de von Neumann é a base para quase todas as arquiteturas computacionais atuais, sendo um design com enorme influência.

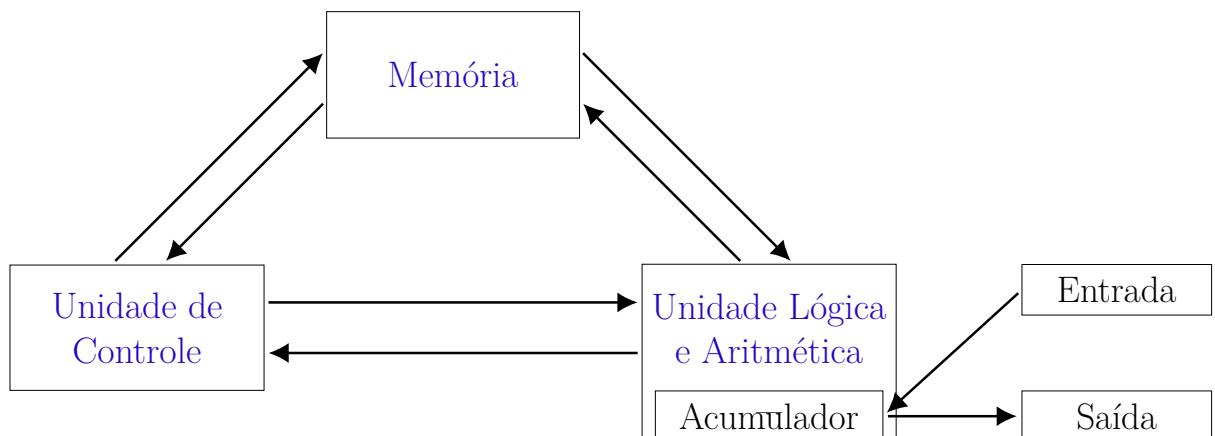


Figura 1 – A Máquina de von Neumann original(TANENBAUM; AUSTIN, 2013, p.18)

A Máquina de von Neumann tinha 5 partes básicas: a memória, a unidade lógica e aritmética(ULA), a unidade de controle e os dispositivos de entrada e de saída. Em computadores atuais, estas 5 partes são combinadas em um chip chamado unidade central de processamento(CPU).(TANENBAUM; AUSTIN, 2013, p.18)

2.1.2 Características de Arquiteturas de Computadores

2.1.2.1 Níveis de Máquina

Tanenbaum e Austin (2013, p.5) diz que a maioria dos computadores modernos consistem em dois ou mais níveis, como representado na Figura 2. Os níveis mais baixos

estão mais próximos do *hardware* real da máquina enquanto os mais altos são abstrações dos níveis abaixo, se aproximando do *software*. No nível mais baixo, o nível de lógica

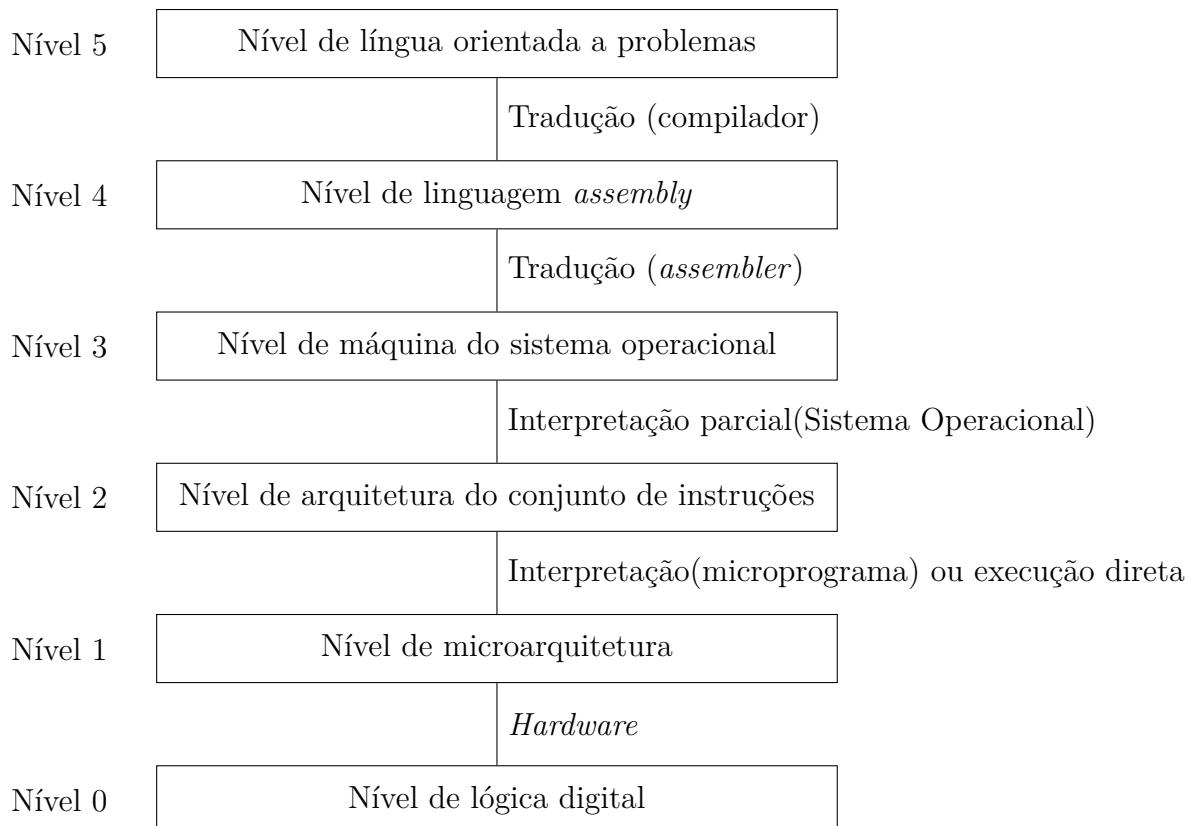


Figura 2 – Um computador de 6 níveis. O método suportado para cada nível é indicado abaixo dele(junto com o programa que suporta). ([TANENBAUM; AUSTIN, 2013](#), Adaptação p.5)

digital, se observa as portas lógicas que compõe o computador e como elas se conectam. Essas portas lógicas funcionam com entradas e saídas digitais, e realizam operações como "OU", "E" e "NÃO"; Quando combinadas entre si, essas portas lógicas podem realizar quaisquer operações matemáticas, ou podem também armazenar informações quando formam componentes chamados registradores.

O seguinte nível é o da microarquitetura. Nesse nível temos componentes mais complexos trabalhando em conjunto. Este engloba a ULA, memória e controle, e conecta eles através de um caminho de dados(*datapath*), o que permite as informações serem enviadas da memória pra ULA, que realiza alguma operação, e por fim devolve este resultado à memória.

O nível de Arquitetura do Conjunto de Instruções(ISA) contém todas instruções que a máquina pode executar, estas sendo representadas pela linguagens de máquina como sequências de números que dizem o que os componentes da máquina devem fazer.

O próximo nível é um nível híbrido, o nível de máquina do sistema operacional, onde a maioria das instruções também está presente no nível ISA. Este nível inclui novas

instruções, uma organização de memória diferente, a capacidade de executar múltiplos programas simultaneamente e outras características. As novas funcionalidades do nível 3 são executadas por um interpretador no nível 2, chamado de sistema operacional. Algumas instruções do nível 3 são executadas diretamente pelo microprograma ou controle por hardware, enquanto outras são interpretadas pelo sistema operacional.

Os dois níveis mais altos são os que os programadores comuns geralmente usam para a resolução de problemas. Estes níveis, diferentemente dos níveis mais baixos, são suportados apenas por tradução. Além disso, a forma com que esses programas são representados é diferente, trocando a forma numérica presente nos níveis mais baixos por uma forma mais compreensível para humanos, composta de palavras e abreviações significativas.

O 4º nível está mais diretamente conectado à linguagem de máquina, contendo a linguagem *assembly*, onde cada comando individual pode ser traduzido diretamente para uma instrução nos níveis inferiores. Esta tradução se dá por um programa chamado *assembler*.

Por fim, temos o 5º nível, que contém todas as linguagens em alto nível que precisam ser então traduzidas para os níveis 4 e 3 no processo chamado compilação.(TANENBAUM; AUSTIN, 2013, p.5-8)

2.1.2.2 Barramentos

Um barramento é um grupo físico de fios que possui uma função relacionada. Eles permitem a comunicação entre as várias partes da microarquitetura, como, por exemplo, um barramento de dados é o grupo de fios que permite a transferência de dados entre a memória e todos os subsistemas que compõem o computador.(CATSOULIS, 2002, p.11-12) Além do barramento de dados, existem o barramento de endereços, responsável por determinar o endereço de memória que está sendo acessado pelo processador, e o barramento de controle, responsável pelos sinais de comandos, que definem o que as partes de uma máquina devem fazer.

2.1.2.3 Pipeline

A *pipeline* de instruções é uma técnica que melhora o desempenho do processador ao dividir a execução de uma instrução em várias etapas, como busca, decodificação, cálculo de operandos, busca de operandos e execução. Semelhante a uma linha de montagem, onde diferentes estágios de produção ocorrem simultaneamente, a *pipeline* permite que várias instruções sejam processadas ao mesmo tempo em diferentes estágios. No entanto, a eficiência da *pipeline* pode ser afetada por fatores como o tempo de execução ser maior que o tempo de busca e instruções de desvio condicional que tornam o endereço da próxima instrução desconhecido. Para mitigar esses problemas, técnicas como a predição

de desvios são utilizadas, permitindo um aumento significativo na velocidade de execução das instruções. (STALLINGS, 1988, p.47-51)

2.1.2.4 Conjunto de instruções

Segundo Monteiro (2007, p.195), o conjunto de instruções (ISA) de um processador estabelece as operações básicas que a máquina executará e influencia diretamente o desempenho de suas atividades. Ele também define as especificações dos demais componentes, já que as instruções afetam praticamente todos os elementos do processador, como a ULA, os registradores e o barramento interno. O formato e os componentes das instruções determinam a organização e o desempenho do decodificador de instruções, bem como a largura do registrador de instruções.

Complementando essa perspectiva, o nível ISA é definido por como a máquina se apresenta ao programador de linguagem de máquina, ou mais realisticamente, ao compilador. Para que o compilador gere código compatível com a ISA, é necessário conhecer o modelo de memória, os registradores disponíveis, os tipos de dados suportados e o conjunto de instruções. Essa coleção de informações constitui a definição da ISA.(TANENBAUM; AUSTIN, 2013, p.345-346)

Assim, a partir da especificação das instruções de máquina, o restante do processador começa a ser delineado, sendo uma parte crucial do projeto de processadores e definindo sua arquitetura. Em arquiteturas modernas, a ISA é formalmente definida por documentos normativos que garantem compatibilidade entre diferentes implementações, permitindo que múltiplos fabricantes produzam chips funcionalmente idênticos.(TANENBAUM; AUSTIN, 2013, p.346)

Esses documentos especificam o comportamento esperado das instruções, modos de operação (como modo usuário e modo kernel), e incluem seções normativas e informativas que orientam tanto projetistas quanto desenvolvedores de compiladores. Portanto, a ISA não apenas guia o projeto físico do processador, mas também estabelece uma interface estável entre hardware e software, sendo essencial para garantir interoperabilidade, desempenho e previsibilidade na execução de programas. (TANENBAUM; AUSTIN, 2013, p.346-347)

2.1.2.4.1 Modos de Operação

Em arquiteturas modernas de processadores, os Modos de Operação representam diferentes níveis de privilégio que determinam o grau de acesso que o código em execução possui sobre os recursos do sistema. Essa separação é essencial para garantir segurança, estabilidade e controle adequado do hardware, especialmente em sistemas multitarefa e com suporte a sistemas operacionais.(RISC-V, 2024a, p.9)

Os modos mais comuns incluem o modo usuário, utilizado por aplicações comuns e caracterizado por acesso restrito ao hardware; o modo supervisor, geralmente reservado ao sistema operacional, com permissões para gerenciar memória, dispositivos e interrupções; e o modo de máquina (ou modo kernel), que possui o nível mais alto de privilégio e acesso irrestrito a todos os recursos do sistema. Esse último é utilizado para inicialização, configuração de segurança e controle de ambientes protegidos.(RISC-V, 2024a, p.9)

Durante a execução normal, o processador opera em modo usuário, executando aplicações. Quando uma operação privilegiada é necessária, como acessar o disco, configurar um temporizador ou manipular registradores de controle, ocorre uma *trap*, que é uma interrupção controlada que transfere a execução para um modo mais privilegiado. Essas transições podem ser verticais, quando há mudança de nível de privilégio, ou horizontais, quando ocorrem dentro do mesmo nível.(RISC-V, 2024a, p.10)

A existência de múltiplos modos permite o isolamento entre aplicações e o sistema operacional, protege contra falhas ou comportamentos maliciosos, viabiliza ambientes seguros e virtualizados, e oferece flexibilidade na execução de diferentes tipos de software. Algumas arquiteturas simplificadas podem implementar apenas o modo máquina, enquanto outras oferecem três ou mais níveis para suportar sistemas operacionais complexos e múltiplos usuários.(RISC-V, 2024a, p.10)

2.1.2.5 Assembly

Para facilitar a programação de computadores e o entendimento de código de máquina foram criadas as linguagens *assembly*, que fazem o uso de mnemônicos, como "ADD", "SUB" e "LDA", para representar as instruções que estão sendo utilizadas no programa. Diferentes famílias de processadores utilizam diferentes linguagens *assembly*, geralmente possuindo apenas algumas similaridades entre si. A transformação do código assembly para instruções de máquina se dá em compiladores especiais chamados *assemblers*.(CATSOULIS, 2002, p.6-7)

2.2 Partes de uma Microarquitetura

Como visto anteriormente nas seções 2.1.1 e 2.1.2.1, podemos separar a microarquitetura em diversos componentes. Cada microarquitetura pode possuir seus próprios componentes, porém eles geralmente seguem alguma variação da organização ULA-Memória-Controle.

2.2.1 Unidade Lógica e Aritmética(ULA)

A ULA é o circuito responsável por realizar operações matemáticas lógicas e aritméticas, como somas e subtrações, "E", "OU" e deslocamento de bits.(FÁVERO, 2011,

p.60) Alguns processadores porém, podem possuir unidades separadas para multiplicação e divisão, ou para o deslocamento de bits, como exemplo o RISC I([CATSOULIS, 2002, p.13](#))

2.2.1.1 IEEE 754

O padrão IEEE 754, estabelecido pela [IEEE \(1985\)](#), define regras para a representação e manipulação de números em ponto flutuante em sistemas computacionais. Ele foi criado para padronizar operações aritméticas que antes variavam entre fabricantes, dificultando a portabilidade e confiabilidade dos programas.

Esse padrão especifica:

- **Formatos de dados:** como números finitos, zeros com sinal, infinitos e valores especiais como *NaN* (Não é um Número).
- **Precisões:** incluindo formatos binários e decimais de 16, 32, 64, 128 e até 256 bits.
- **Regras de arredondamento:** para garantir consistência em cálculos.
- **Operações aritméticas:** como soma, subtração, multiplicação, divisão e funções matemáticas.
- **Tratamento de exceções:** como divisão por zero, estouro e operações inválidas.

Esse componente pode ser implementado separadamente da Unidade Lógica e Aritmética (ULA), por meio de uma Unidade de Ponto Flutuante (FPU), dedicada exclusivamente ao processamento de operações em ponto flutuante, o que permite maior desempenho e precisão em cálculos científicos e gráficos.

2.2.2 Memória

A memória em sistemas de computação desempenha um papel crucial no armazenamento e recuperação de informações essenciais para o funcionamento do sistema. Conceitualmente, a memória é simples, atuando como um repositório de dados. Entretanto, na prática, ela se configura como um subsistema complexo, composto por diferentes tipos de memórias, integradas para garantir desempenho e eficiência. ([MONTEIRO, 2007, p.79](#))

Segundo Monteiro (2007, p.79-80), a necessidade de múltiplos tipos de memória surge da crescente disparidade entre a velocidade dos processadores e o tempo de acesso às memórias. Enquanto a capacidade de processamento aumenta rapidamente, o avanço na velocidade de acesso à memória é mais lento. Além disso, o aumento no volume de dados e no tamanho dos programas exige maior capacidade de armazenamento.

Dessa forma, torna-se impraticável criar uma única memória que seja rápida, de grande capacidade e custo acessível. Para resolver esse problema, adota-se uma hierarquia de memórias, com diferentes tipos de componentes que equilibram velocidade, capacidade e custo, otimizando o desempenho do sistema computacional.

2.2.2.1 Registradores

Segundo [Catsoulis \(2002, p.14\)](#), os registradores são as unidades internas de armazenamento de um processador. Estes se comunicam diretamente com os outros componentes internos do processador. Estes registradores podem servir para funções como:

- **Acumuladores**

São usados em operações aritméticas. Em algumas arquiteturas, todos os registradores podem agir como acumuladores, enquanto em outras, estes servem apenas para armazenamento.

- **Registradores de Estado**

Servem para armazenar bits de estado que refletem resultados de operações anteriores, como zero, negativo ou interrupções.

- **Contadores de programa(PC)**

Estes são registradores especiais que apontam para a próxima instrução a ser buscada na memória e executada pelo processador. Quando a instrução é buscada, o sistema atualiza automaticamente o PC para armazenar o endereço da próxima instrução. Assim, o PC é essencial para o controle e sequenciamento da execução dos programas.([FáVERO, 2011](#))

- **Registradores de Controle**

Armazenam informações importantes para o controle, como código de operação ou resultado de sua decodificação.

2.2.2.2 Memória Cache

Um dos maiores desafios no projeto de computadores ao longo da história tem sido desenvolver sistemas de memória capazes de fornecer operandos ao processador na mesma velocidade em que ele os processa. Com o avanço acelerado da velocidade dos processadores, as memórias principais não acompanharam esse ritmo, tornando-se relativamente mais lentas a cada geração. Essa disparidade entre a velocidade da CPU e da memória tem limitado significativamente o desempenho dos sistemas computacionais, motivando pesquisas em soluções que contornem essa limitação ([TANENBAUM; AUSTIN, 2013, p.304](#)).

Os processadores modernos exigem muito dos sistemas de memória, tanto em termos de latência (tempo de resposta para fornecer um operando) quanto de largura de banda (quantidade de dados fornecidos por unidade de tempo). No entanto, essas duas características frequentemente entram em conflito: técnicas que aumentam a largura de banda tendem a elevar a latência. Por exemplo, o uso de *pipelining* em sistemas de memória permite o processamento simultâneo de múltiplas requisições, mas aumenta o tempo de resposta individual de cada operação (TANENBAUM; AUSTIN, 2013, p.304).

Uma das soluções mais eficazes para esse problema é o uso de **memória cache**. A cache é uma memória pequena e extremamente rápida que armazena os dados mais recentemente utilizados, permitindo acessos mais rápidos. Quando uma alta porcentagem dos dados requisitados está presente na cache, a latência efetiva do sistema de memória é significativamente reduzida (TANENBAUM; AUSTIN, 2013, p.304).

2.2.2.2.1 Arquitetura de Caches

Para melhorar simultaneamente a latência e a largura de banda, é comum utilizar múltiplos níveis de cache. Um modelo básico e eficiente é o uso de caches separadas para instruções e dados, conhecido como cache dividida (*split cache*). Essa abordagem permite que operações de memória sejam iniciadas de forma independente em cada cache, dobrando efetivamente a largura de banda.

Nos sistemas modernos, além das caches de instruções e dados (**L1**), há também caches de segundo nível (**L2**) e terceiro nível (**L3**). A arquitetura típica inclui:

- **Cache L1**: localizada no chip do processador, com tamanho entre 16 KB e 64 KB, separada para instruções (L1-I) e dados (L1-D).
- **Cache L2**: geralmente unificada, com capacidade entre 512 KB e 1 MB, conectada ao chip por um caminho de alta velocidade.
- **Cache L3**: localizada na placa do processador, composta por alguns megabytes de SRAM, mais rápida que a memória principal (DRAM).

Essas caches são geralmente inclusivas, ou seja, os dados presentes na L1 também estão na L2, e os da L2 estão na L3 (TANENBAUM; AUSTIN, 2013, p.305).

2.2.2.3 Arquivo de Registradores

O Arquivo de Registradores é um componente presente desde as primeiras arquiteturas RISC (PEEK, 1983, p.7-10), consistindo em um conjunto de registradores de uso geral disponíveis para operações durante a execução de instruções. Ele está presente em diversas arquiteturas RISC, como SPARC (SPARC International Inc, 1994, p.25),

MIPS ([MIPS Tech LLC, 2016](#), p.40-42), ARM ([ARM, 2014](#)), RISC-V ([RISC-V, 2024b](#), p.20-22), entre outras.

Apesar de variações entre arquiteturas, algumas características são recorrentes: a presença de 32 registradores acessíveis por vez, a conexão fixa do primeiro registrador ao valor zero, e o uso de janelas de registradores. Este último é um mecanismo que permite que apenas uma porção dos registradores físicos esteja visível ao programador em determinado momento.

Em arquiteturas como a RISC I ([PEEK, 1983](#), p.7-10), o Arquivo de Registradores é dividido em múltiplas janelas sobrepostas, cada uma contendo registradores locais, de entrada e de saída. Durante chamadas de procedimentos, a janela ativa é deslocada, permitindo que os registradores de saída de uma função se tornem os de entrada da próxima, reduzindo a necessidade de operações de carga e armazenamento na memória ([PEEK, 1983](#)).

Contudo, embora as janelas de registradores ofereçam acesso rápido a variáveis locais e minimizem o tráfego de memória, esse recurso tem sido gradualmente abandonado em arquiteturas mais recentes. Isso se deve à sua rigidez estrutural e ao uso potencialmente ineficiente de espaço, já que nem todas as janelas são utilizadas plenamente. Em contraste, *caches* modernos permitem uma alocação dinâmica de variáveis, adaptando-se melhor ao comportamento real do programa e oferecendo suporte tanto a variáveis locais quanto globais ([STALLINGS, 1988](#), p.43-44).

Além disso, com o avanço das técnicas de otimização em compiladores, tornou-se possível mapear eficientemente um número ilimitado de variáveis simbólicas em um conjunto fixo de registradores físicos. Essa abordagem reduz a necessidade de estruturas complexas como janelas de registradores, promovendo maior flexibilidade e melhor aproveitamento de espaço ([STALLINGS, 1988](#), p.44).

2.2.2.4 Modos de Endereçamento

[Catsoulis \(2002, p.15-16\)](#) diz que há diferentes formas que uma instrução pode referenciar um registrador ou locais de memória, estas formas sendo conhecidas como modos de endereçamento de um processador. Cada arquitetura possui suas próprias formas de endereçamento, dentre estas podemos citar([PATTERSON; HENNESSY, 1996, p.75](#))([CATSOULIS, 2002](#)):

- **Inerente**

A instrução lida puramente com registradores.

- **Imediatos**

A instrução possui um valor literal como um de seus operandos.

- **Direto ou Absoluto**

Parte da instrução possui um endereço que tem acesso parcial a memória. Este modo de endereçamento é comumente utilizado para reduzir a quantidade de acessos a memória, tirando a tanto a necessidade de uma busca adicional de endereços quanto reduzindo o código ao armazenar este endereço diretamente na instrução.

- **Registrador Indireto**

Acessa um local da memória, especificado por um endereço completo(ponteiro) que esta armazenado em um registrador.

- **Memória Indireta**

Similar ao registrador indireto, mas neste caso, o ponteiro esta contido na memória principal ao invés de um registrador.

- **Indexado**

Geralmente usado para ler vetores de valores, utilizando o ponteiro contido em um registrador como base, e adicionando o valor de outro registrador para obter o valor em um índice específico.

- **Relativo**

Este modo de endereçamento é parecido com o indexado, porém o valor base geralmente é o do contador de programas. Como exemplo, há instruções que podem utilizar o endereço do contador de programas para obter o endereço de uma instrução de pulo.

2.2.3 Unidade de Controle

O controlador, ou unidade de controle é a parte da microarquitetura que é responsável por comandar os outros componentes do sistema. Não há um padrão definido para o funcionamento do controle e é algo que se altera tanto com a arquitetura quanto pela implementação física da mesma. Como exemplo temos o RISC I, que segundo [Peek \(1983\)](#), foi implementado com 3 partes: O controle de instruções, o controle de tamanho de informações e o controlador da janela do Arquivo de Registradores. Outra possível implementação é integrar tudo em um mesmo componente de controle geral, como feito no SAP-1([MALVINO; BROWN, 1993](#))

2.3 Arquitetura Berkeley RISC

A criação das arquiteturas RISC (*Reduced Instruction Set Computer*) foi um marco significativo na história da computação. O projeto foi liderado pelos professores David Patterson e Carlo H. Séquin na Universidade de Berkeley, Califórnia. Este foi iniciado com

o objetivo de simplificar as arquiteturas de computadores, que estavam se tornando cada vez mais complexas. A ideia central era reduzir o conjunto de instruções para aumentar a eficiência e acelerar o processo de design.

Dois grupos de estudantes participaram do desenvolvimento das duas primeiras versões do RISC (Daniel T. Fitzpatrick et al., 1982). O grupo “Gold”, composto pelos estudantes Fitzpatrick, Foderaro, Peek, Peshkess, e Van Dyke, projetou o microprocessador RISC I (PEEK, 1983), enquanto grupo “Blue”, que incluía os estudantes Katevenis e Sherburne, introduziu diversas modificações no *datapath* e controle do RISC I, resultando no RISC II (KATEVENIS, 1985).

O RISC I foi projetado com um conjunto pequeno de instruções, de inicialmente 31 instruções sendo expandido para 39 ao fim do seu desenvolvimento (PATTERSON; SÉQUIN, 1982). O RISC I visava uma execução eficiente. A simplicidade do design permitiu que ele fosse implementado em um único chip, otimizando o uso de recursos limitados como transistores, área e consumo de energia.

O RISC II começou a ser criado em paralelo à produção do chip do RISC I, e introduziu um Arquivo de Registradores mais compacto, que requereu por sua vez, um sistema mais sofisticado de temporização (PATTERSON; SÉQUIN, 1982). Além disso, o RISC II introduziu diversas outras mudanças significativas como um sistema de instruções privilegiadas, melhorias e inclusões de novas interrupções, reorganização das instruções e mudanças estruturais de algumas conexões do *datapath*.

2.3.1 RISC-V

O RISC-V é uma arquitetura de conjunto de instruções (ISA) baseada nos princípios RISC, desenvolvida inicialmente em 2010 no Laboratório de Arquitetura de Computadores da Universidade da Califórnia, Berkeley. Diferente de seus predecessores acadêmicos, o RISC-V foi projetado com foco em extensibilidade, simplicidade e liberdade de uso, sendo disponibilizado como padrão aberto e livre de royalties, o que o torna altamente atrativo para pesquisa, ensino e aplicações comerciais.(PATTERSON; CHEN, 2016, p.1)

A arquitetura RISC-V mantém os fundamentos das arquiteturas RISC clássicas, como instruções de tamanho fixo, uso intensivo de registradores e separação clara entre instruções de carga/armazenamento e operações aritméticas. No entanto, ela introduz um sistema modular de extensões, permitindo que implementações sejam adaptadas conforme as necessidades específicas de cada projeto, desde microcontroladores até supercomputadores.(RISC-V, 2024b)

Entre suas principais características estão:

- **Base modular:** composta por um conjunto mínimo de instruções inteiras (RV32I

ou RV64I), ao qual podem ser adicionadas extensões como ponto flutuante (F), multiplicação e divisão (M), instruções atômicas (A), entre outras.

- **Suporte a múltiplas larguras de palavra:** incluindo variantes de 32, 64 e 128 bits.
- **Facilidade de implementação:** projetada para ser simples de decodificar e eficiente em hardware.
- **Ecossistema aberto:** com ferramentas, compiladores, simuladores e núcleos de processador disponíveis publicamente.

2.4 CISC versus RISC

As arquiteturas de processadores podem ser classificadas em duas abordagens principais: CISC (Computadores com Conjunto Complexo de Instruções) e RISC (Computadores com Conjunto Reduzido de Instruções) ([CATSOULIS, 2002](#), p.19), previamente discutido no tópico anterior. Cada abordagem oferece vantagens e desvantagens dependendo das exigências do sistema.

A arquitetura CISC é caracterizada por conjuntos de instruções complexas e diversificadas, capazes de realizar várias operações com uma única instrução. Essa complexidade reduzia o número de instruções necessárias, o que era vantajoso em um cenário onde a memória era cara e lenta. No entanto, essa abordagem levou a processadores mais complexos, com unidades de controle e decodificação mais lentas e maiores demandas de energia, gerando sobrecarga no design do hardware.

Por outro lado, a arquitetura RISC surgiu com a ideia de simplificar o processador, transferindo parte da complexidade para os compiladores. Processadores RISC possuem conjuntos de instruções reduzidos, otimizados para execução rápida e eficiente. Embora exijam mais instruções para realizar certas tarefas, a simplicidade da decodificação e execução resulta em um desempenho superior, com instruções que geralmente são completadas em um ou dois ciclos. Além disso, os processadores RISC utilizam grandes conjuntos de registradores e implementam uma arquitetura de carga/armazenamento, onde apenas instruções específicas acessam a memória, reduzindo o número de acessos à memória externa.

RISC também se destaca pelo uso de *pipelines*, permitindo a execução simultânea de várias instruções em diferentes estágios, o que aumenta ainda mais o desempenho. Devido ao baixo consumo de energia e alta eficiência, os processadores RISC são amplamente utilizados em sistemas embarcados e de baixa potência.

Apesar dessas distinções, as arquiteturas CISC e RISC têm se aproximado ao longo do tempo, com processadores CISC adotando características de RISC, como *pipelines* e

simplificação de instruções, e processadores RISC Incorporando recursos mais complexos típicos da CISC. A escolha entre essas arquiteturas depende das necessidades específicas de cada aplicação: enquanto RISC oferece vantagens em termos de eficiência energética, CISC pode ser preferível em sistemas com espaço limitado para armazenamento de programas, devido à sua maior densidade de instruções.(CATSOULIS, 2002, p.19-21)

2.5 Simuladores

Um simulador é um software com uma interface interativa que facilita a construção de modelos de simulação. Ele permite que os usuários criem modelos através de elementos básicos e os interliguem conforme o fluxo lógico do sistema. A maioria dos simuladores modernos minimiza a necessidade de programação, tornando o processo mais acessível e rápido.(CHWIF, 2014)

Os softwares de simulação permitem a criação de modelos computacionais que representam sistemas reais. Esses modelos podem ser usados para analisar e prever o comportamento do sistema sob diferentes condições. O processo envolve a definição de parâmetros, como tempos de chegada e atendimento, e a execução do modelo para obter medidas de desempenho, como a taxa de utilização de um atendente.(CHWIF, 2014)

2.5.1 Linguagens de Descrição de Hardware (HDLs)

As linguagens de descrição de hardware (*Hardware Description Languages* — HDL) são ferramentas fundamentais para o desenvolvimento e simulação de circuitos digitais. Ao invés de utilizar esquemáticos visuais, que se tornam impraticáveis à medida que a complexidade dos circuitos aumenta, HDLs permitem representar digitalmente a estrutura e o comportamento de sistemas eletrônicos por meio de uma linguagem textual.(KEIM, 2020)

Essa abordagem é especialmente útil na modelagem hierárquica de circuitos, que organiza transistores em portas lógicas, portas em módulos funcionais, e assim sucessivamente. Com HDLs como VHDL e Verilog, é possível expressar essas estruturas com precisão e eficiência, facilitando a síntese e simulação do circuito em plataformas como FPGAs.

Ao contrário das linguagens de programação tradicionais, HDLs descrevem operações que ocorrem em paralelo, refletindo a natureza simultânea do funcionamento dos circuitos digitais. Isso permite que diferentes partes do hardware sejam simuladas ou executadas ao mesmo tempo, mesmo que a descrição textual siga uma ordem sequencial.(KEIM, 2020)

3 Métodos e recursos

3.1 Visão geral

Para o desenvolvimento deste trabalho, foi necessário seguir os seguintes passos metodológicos:

1. Pesquisar arquiteturas simples e didáticas:

O primeiro passo consistiu em identificar arquiteturas relevantes que pudessem servir como base para o trabalho a ser desenvolvido. Isso requereu uma pesquisa na literatura acadêmica, livros, artigos e recursos online para encontrar essas arquiteturas.

2. Pesquisar documentação sobre a arquitetura escolhida:

Das arquiteturas pesquisadas anteriormente, uma foi escolhida para uma investigação mais aprofundada. Isso envolveu o estudo de manuais, relatórios técnicos, artigos e materiais relacionados para compreender os detalhes da arquitetura escolhida.

3. Compilar a documentação da arquitetura base:

As informações coletadas no passo anterior foram organizadas para criar uma documentação abrangente que descrevesse os componentes, instruções, registradores e fluxo de dados da arquitetura base.

4. Criar o modelo de simulação base:

Foi necessário criar uma implementação funcional da arquitetura. Isso envolveu a escolha de um software para prototipagem de circuitos lógicos e a criação dos circuitos, juntamente com anotações detalhadas do processo.

5. Criar um *assembler*:

Um montador (*assembler*) foi criado para traduzir a linguagem de montagem em instruções executáveis pela arquitetura. Isso foi fundamental para programar e executar códigos na arquitetura.

6. Documentar o trabalho desenvolvido:

Um registro detalhado de todas as etapas anteriores foi criado. Isso incluiu descrições das modificações feitas na arquitetura, decisões de projeto e aspectos relevantes do *assembler*.

7. Pesquisar métodos didáticos:

Foi necessário pesquisar abordagens pedagógicas eficazes para ensinar arquitetura e organização de computadores. Isso incluiu a criação de exemplos práticos, analogias, exercícios e estudos de caso.

8. Realizar uma bibliometria sobre arquiteturas computacionais:

Uma bibliometria ofereceu informações importantes sobre a relevância da arquitetura escolhida em comparação a outras de sua época e mais recente, de forma documentada e parametrizada.

9. Pesquisar recursos de interesse ausentes na arquitetura:

Para os próximos passos, foi necessário pesquisar recursos ou funcionalidades que não estavam presentes na arquitetura base. Isso incluiu instruções específicas, modos de endereçamento ou simplificações de componentes.

10. Listar recursos e otimizações de interesse:

Foram escolhidos os recursos a adicionar e as otimizações a aplicar. Foi importante considerar o impacto no aprendizado dos alunos ao selecionar essas melhorias, bem como a complexidade de implementação.

11. Planejar a implementação de um novo recurso:

Foi necessário traçar um plano detalhado para incorporar os recursos escolhidos à arquitetura. Isso envolveu investigar a documentação existente e criar notas sobre o processo a ser executado.

12. Implementar o recurso:

Após o planejamento, os recursos foram implementados no modelo de simulação, com modificações no *assembler* caso necessário.

13. Testar o funcionamento do recurso desenvolvido:

Testes foram essenciais para garantir que os recursos funcionassem conforme o esperado. Isso envolveu casos de teste, depuração e validação.

14. Documentar o recurso adicionado:

A documentação foi incrementada com informações detalhadas sobre os novos recursos. Isso incluiu como eles funcionavam, como eram utilizados e como foram implementados na arquitetura base.

15. Consolidação da documentação da arquitetura:

Todas as documentações criadas durante o trabalho foram compiladas em um único documento. Isso incluiu a descrição da arquitetura, o *assembler* e os recursos adicionados.

16. Organizar os resultados e a conclusão:

Os resultados da avaliação foram analisados, e foram tiradas conclusões sobre o projeto e finalizada a monografia.

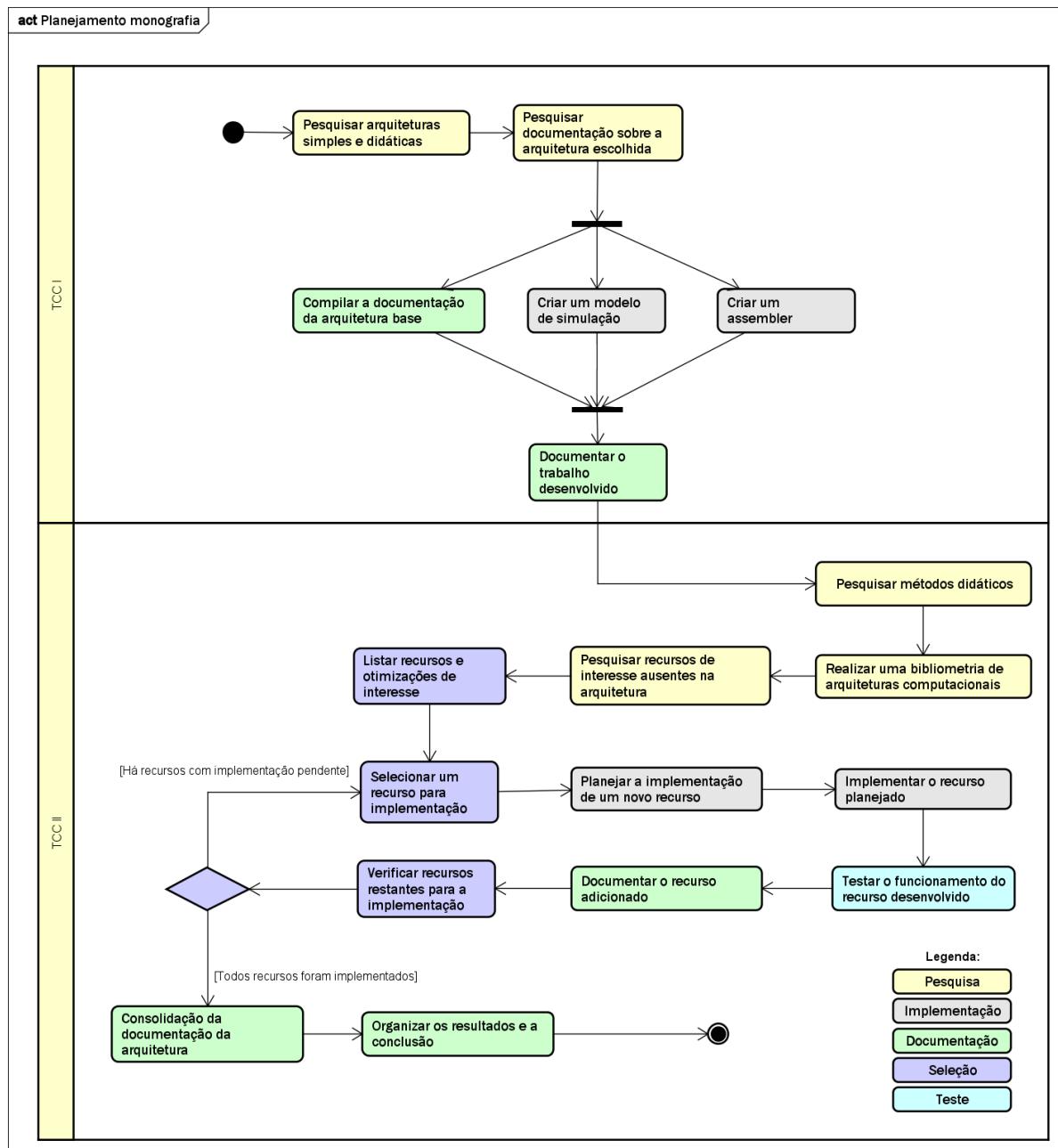


Figura 3 – Planejamento Monografia

Na Figura 3 observa-se os passos previamente descritos, assim como objetivos citados na seção 1.2 deste documento.

3.2 Ferramentas Utilizadas

Para o desenvolvimento deste trabalho, foram utilizados diversos recursos tecnológicos que facilitaram a criação, organização e documentação do projeto. A seguir, são descritos os principais recursos empregados:

3.2.1 Simulador

Para a simulação e validação dos circuitos lógicos, foram utilizados os simuladores Logisim Evolution [Burch et al. \(2014\)](#) e Icarus Verilog [Williams \(1998\)](#). O Logisim Evolution foi empregado para a montagem visual do circuito da arquitetura, além de permitir um modelo interativo. Já o Icarus Verilog atendeu à necessidade de simulação de linguagens de descrição de hardware, proporcionando testes mais eficientes e diretos, bem como a implementação da arquitetura em FPGA.

Ambos se destacaram pela simplicidade e por serem de código aberto, possibilitando o uso sem custos adicionais e garantindo flexibilidade na modificação de funcionalidades conforme a necessidade do projeto.

3.2.2 IDE e Linguagem de Programação

Para a programação do *assembler* do sistema, foi escolhida a IDE Visual Studio 2022, devido às suas ferramentas de desenvolvimento avançadas e suporte à linguagem C# na versão .NET 8. Essa combinação facilitou a escrita, depuração e manutenção do código.

A linguagem de descrição de hardware SystemVerilog foi utilizada para modelar a arquitetura que seria executada no Icarus Verilog. Para sua programação, foi utilizado o Visual Studio Code, com a extensão Vaporview(??)Vaporview) que oferece a visualização de formas de onda resultantes da simulação.

Por fim, o Notepad++ foi utilizado para a programação dos programas em assembly executados na arquitetura criada.

3.2.3 Documentação

A documentação do projeto foi elaborada utilizando diferentes ferramentas para atender a necessidades específicas. O *Google Sheets* foi utilizado para a criação e organização de tabelas, enquanto o editor LaTe $\mathrm{\acute{e}}\mathrm{x}$ Overleaf foi empregado na formatação e montagem do documento principal, incluindo esta monografia.

3.2.4 Controle de Versão

O controle de versão do projeto foi gerenciado através da plataforma GitHub. Esta escolha permitiu um acompanhamento detalhado das alterações no código e fácil acesso a versões anteriores do projeto, garantindo maior segurança e integridade do desenvolvimento.

3.2.5 Assistentes Virtuais

Para auxiliar na escrita dos textos da monografia e relatórios, com correções gramaticais e semânticas, foi utilizada a ferramenta Microsoft Copilot. Além disso, para

otimizar a escrita de códigos, a ferramenta GitHub Copilot foi empregada, fornecendo predições automáticas de código e melhorando a eficiência do desenvolvimento.

4 Trabalhos Realizados

Durante a primeira fase de criação da monografia, uma parte significativa do trabalho foi realizada, que incluiu a criação de uma base de trabalho que pudesse ser modificada durante a fase subsequente.

Nesta primeira fase, foram incluídas a pesquisa sobre arquiteturas viáveis para replicação e modificação, a implementação de um modelo de simulação dessa arquitetura no simulador escolhido e a criação de um *Assembler* com uma linguagem que possa ser montada para a linguagem da arquitetura desenvolvida.

Adicionalmente, foi escrita uma breve documentação, em formato de relatório técnico, detalhando como esses trabalhos foram realizados, que foram publicados abertamente na plataforma Github ([VALADARES, 2024](#)), acessível pelo endereço <<https://github.com/Diogo-Valadares/Didactic-RISC-I>>.

4.1 Pesquisa de Arquiteturas

O desenvolvimento de uma arquitetura didática requereu uma escolha cuidadosa de uma estrutura que fosse funcional e facilmente replicável e modificável para fins educacionais. Selecionar uma arquitetura adequada é fundamental para garantir que os conceitos possam ser ensinados de maneira clara e eficaz. Para isso, foi necessário realizar uma pesquisa abrangente que fornecesse uma visão geral das opções disponíveis, permitindo a identificação de arquiteturas que atendessem aos critérios específicos deste trabalho.

Esta seção detalha o processo realizado durante os dois primeiros passos metodológicos do trabalho: pesquisar arquiteturas simples e didáticas, e pesquisar documentação sobre a arquitetura escolhida.

4.1.1 Critérios da Pesquisa

Para iniciar esta pesquisa, foi essencial definir critérios que ajudassem a filtrar arquiteturas sem valor didático significativo ou excessivamente complexas. A simplicidade foi um fator importante, pois arquiteturas mais simples foram mais fáceis de replicar e entender. No entanto, simplicidade excessiva poderia resultar em uma falta de recursos didáticos, prolongando o processo de modificação necessário para fins educacionais.

Uma estratégia eficaz foi buscar arquiteturas mais antigas, que tendiam a ser mais simples e possuíam uma boa quantidade de documentação disponível. Arquiteturas populares entre as décadas de 70 e 90 foram analisadas, com resultados mais promissores

em lançamentos pós-1980. Essas arquiteturas eram menos complexas, tornando-as mais acessíveis e também ofereciam uma base sólida de conhecimento acumulado ao longo dos anos.

Além do ano de lançamento, foram consideradas características técnicas que indicavam o nível de complexidade da arquitetura. Entre essas características estavam a largura de endereços e dados, a quantidade de instruções, a existência de uma pilha (*stack*), o número de registradores, o suporte a operações de multiplicação e divisão, o suporte a números com ponto flutuante (*floats*) e a utilização de *pipeline*. Essas características ajudaram a determinar a adequação da arquitetura para fins didáticos, garantindo que ela fosse suficientemente robusta para ilustrar conceitos importantes sem se tornar excessivamente complexa.

Os últimos critérios foram mais subjetivos e incluíram a análise do contexto em que as arquiteturas foram utilizadas e se já tinham sido empregadas em ambientes educacionais. Saber onde e como uma arquitetura foi utilizada pode fornecer perspectivas sobre sua relevância e aplicabilidade no ensino de conceitos de arquitetura de computadores. Arquiteturas que já foram usadas em contextos didáticos tiveram a vantagem de serem mais facilmente adaptáveis para fins educacionais. Isso ocorreu porque muitas dessas arquiteturas já eram projetadas para fins educacionais ou foram reaproveitadas de produtos comerciais como objetos de estudo. Essas arquiteturas forneceram um ponto de partida concreto para o desenvolvimento de materiais de ensino, oferecendo exemplos práticos e documentações já testadas em sala de aula.

4.1.2 Metodologia de Pesquisa

A pesquisa sobre a arquitetura base foi dividida em duas etapas: uma pesquisa geral sobre diversas arquiteturas para selecionar a mais adequada às necessidades do trabalho, seguida por uma pesquisa detalhada sobre a arquitetura escolhida, visando obter o máximo de documentação disponível.

Na primeira fase, foram realizadas pesquisas iniciais em ferramentas como Google e Microsoft Copilot para identificar superficialmente arquiteturas que atendessem aos critérios do trabalho, criando uma lista preliminar. Em seguida, essa pesquisa foi aprofundada para obter detalhes técnicos de cada arquitetura, utilizando fontes acadêmicas como Google Scholar e Periódicos Capes. Esta investigação detalhada permitiu a coleta de informações críticas e a expansão da lista com arquiteturas relacionadas às inicialmente encontradas.

A segunda fase começou após a escolha da arquitetura e continuou durante o desenvolvimento do modelo de simulação. Pesquisas adicionais foram realizadas para preencher lacunas deixadas por estudos anteriores. Nesta etapa, além das ferramentas de busca já mencionadas, foi utilizada a ferramenta Research Rabbit para explorar trabalhos

relacionados aos já encontrados. Bibliotecas digitais da Universidade de Berkeley, Califórnia, onde a arquitetura RISC I foi desenvolvida, também foram consultadas.

4.1.3 Trabalhos Encontrados

Os trabalhos encontrados foram classificados em três categorias de importância. A primeira categoria incluiu arquiteturas que não foram relevantes para este estudo devido à sua complexidade, simplicidade excessiva ou por não atenderem a requisitos específicos. A segunda categoria abrangeu trabalhos que, embora não fossem sobre a arquitetura escolhida, eram relacionados e poderiam inspirar futuros desenvolvimentos. A última categoria consistiu em trabalhos diretamente relacionados à arquitetura selecionada. Todas as arquiteturas pesquisadas podem ser vistas na Tabela 1.

Arquitetura	Intel 4004	MOS Technology 6502	zilog z80	RISC I	RISC II	MIPS I	MIPS II	MIPS III	SAP	DLX
Ano	1971	1975	1976	1981-1984	1984	1985	1989	1991	1993	1994
Uso	Calculadoras	Sistemas embarcados, desktops(APPLE II) e portáteis	Sistemas embarcados, desktops e portáteis	Conceitual	Conceitual	Computadores de uso geral	Servidores	Computadores de uso geral	Educacional	Educacional
Largura de comunicação	4bits	8bits	8bits	32 bit	32 bit	32 bit	64bit	16 bits	32 bit	
Largura de endereços	12bits	16bits	16bits	32 bit	32 bit	32 bit	64bit	16 bits	32 bit	
Número de instruções	46	56	244	31-39	39	50	???	133	53	95
Número de Registradores	16	6	22	78	136	32	32	32	4	???
Pipeline	Não	Não	Não	2 estágios	3 estágios	5 estágios			Não	5 estágios
Stack	Sim	Sim	Sim	Como arquivo de registradores		Sim	Sim	Sim	A partir do SAP 3	Sim
Divisão e Multiplicação	Apenas por Bit Shifting(*2 ou /2)					Sim	Sim	Sim	Não	em algumas versões
Suporte a Float	Não	Não	Não	Não	Não	Como co-processador			Não	em algumas versões
Relação didática	Apesar de haver diversos simuladores, não são utilizados didaticamente				Criado por estudantes	Há implementações didáticas do MIPS			É uma arquitetura didática	

Tabela 1 – Arquiteturas encontradas na primeira fase de pesquisa.

Na primeira categoria, destacaram-se as arquiteturas Intel 4004 (1971), Zilog Z80 (1976) e MOS Technology 6502 (1975). Essas arquiteturas comerciais foram descartadas por serem ou muito simples ou muito complexas, além de não oferecerem conteúdo relevante para os objetivos deste trabalho.

A segunda categoria incluiu as arquiteturas SAP, MIPS e DLX. O SAP ([MALVINO; BROWN, 1993](#)) foi uma arquitetura teórica criada para ser a mais simples possível, ainda representando arquiteturas reais. Embora interessante, foi descartada por sua simplicidade excessiva, mas poderia servir de inspiração.

As arquiteturas MIPS e DLX ([PATTERSON; HENNESSY, 1996](#)) foram exemplos de arquiteturas RISC que expandiam os conceitos do RISC I. Apesar de serem mais complexas do que o desejado, com um *pipeline* de 5 fases, ofereceram uma base sólida de conhecimento. A DLX, desenvolvida com um objetivo didático, derivou da MIPS e combinou conceitos de várias outras arquiteturas, alinhando-se bem aos requisitos deste estudo. Trabalhos relacionados à DLX, por isso, podem ser úteis como referência.

A terceira categoria foi composta por trabalhos diretamente relacionados a arquitetura escolhida, o RISC I. O trabalho de [Peek \(1983\)](#) foi uma referência crucial, detalhando a implementação do RISC I em um relatório técnico. Este trabalho ofereceu diagramas dos circuitos e partes da arquitetura, além da descrição do funcionamento de

certos mecanismos, o que foi essencial para a replicação da arquitetura. Quase todo o modelo de simulação foi baseado neste trabalho. No entanto, muitas partes tiveram que ser decifradas ou recriadas do zero devido à falta de alguns detalhes ou às limitações do simulador em comparação à implementação física.

A dissertação de [Katevenis \(1985\)](#), embora focada no RISC II, teve grande importância ao preencher lacunas deixadas pelo trabalho de Peek. Informações cruciais fornecidas pelo trabalho de Katevenis incluíram o funcionamento da seleção dos bytes durante instruções de carregamento e armazenamento, o funcionamento circular da janela de registradores, o gerenciamento de interrupções e o uso do registrador imediato, além de várias outras informações gerais que são comuns entre o RISC II e o RISC I.

Finalmente, o artigo de [Stallings \(1988\)](#) sobre arquiteturas RISC contribuiu significativamente para esclarecer aspectos não abordados anteriormente. Esses aspectos incluem o pipeline, com comparações entre diferentes formas de divisão de tarefas, o formato de instrução, que não é completamente detalhado no trabalho de Peek, e a descrição de todas as instruções.

4.1.4 Motivações da Escolha da arquitetura RISC I

O RISC I foi escolhido por oferecer uma quantidade balanceada de recursos, não sendo difícil de entender e replicar, mas ainda complexo o suficiente para servir como uma boa base para o trabalho. A quantidade de instruções é pequena, totalizando 39 ao final do seu desenvolvimento. Muitas das instruções possuem mecanismos e ativações de controle semelhantes, o que facilitou a criação do circuito. A *pipeline* do RISC I oferece 2 fases, sendo fácil de entender, e, caso necessário, algumas pequenas alterações podem aumentar esse número para 3 fases, conforme implementado no RISC II.([KATEVENIS, 1985](#))

A arquitetura utiliza tanto largura de endereços quanto de dados de 32 bits, sendo similar a arquiteturas mais atuais. No quesito de operações matemáticas, o RISC I não oferece muitas opções, contendo apenas algumas operações lógicas, soma e subtração. Apesar disso, operações adicionais como multiplicação, divisão ou operações com ponto flutuante podem ser adicionadas facilmente, tanto pelas facilidades do simulador quanto pelo espaço para expansão no conjunto de instruções.([PEEK, 1983](#))

Outro recurso que o RISC I apresenta é a janela de registradores, que, embora não seja exatamente uma pilha (*stack*), possui uma função bem similar, além de ser um recurso interessante para apresentação didática. Esta janela de registradores acelera o processamento de programas no RISC I, reduzindo a necessidade de acesso à memória primária do sistema([PEEK, 1983](#)). No entanto, durante a segunda fase deste trabalho, esse recurso foi deprecado por razões que serão explicadas posteriormente.

Esta arquitetura foi criada por estudantes em um meio acadêmico, demonstrando

ser uma boa opção para uso didático. Além disso, o RISC I foi uma arquitetura altamente influente, servindo de referência para diversas arquiteturas atuais, como as arquiteturas ARM, sendo um ótimo ponto de entrada para quem deseja aprender estas arquiteturas.(TANENBAUM; AUSTIN, 2013)

4.2 Modelo de Simulação Base

A criação de um modelo de simulação base foi um dos passos metodológicos executados. O modelo desenvolvido possui todas as funcionalidades da arquitetura RISC I. Apesar de ter recebido modificações em relação à implementação original, devido a limitações do simulador e à falta de informações, não houve diferenças práticas no funcionamento geral do sistema.

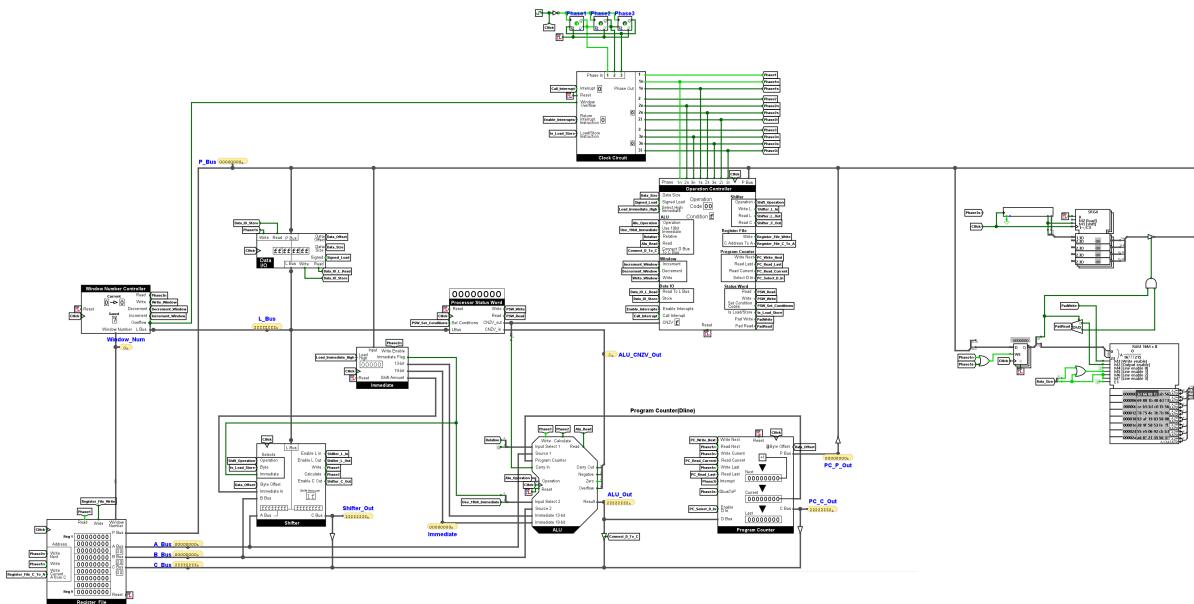


Figura 4 – Modelo de simulação no Logisim Evolution.

A Figura 4 mostra o modelo criado com todos os seus componentes. Para a execução de programas, os programas compilados podem ser carregados diretamente na memória RAM do sistema. A possibilidade de carregar programas diretamente no modelo de simulação simplificou seu teste e desenvolvimento. Na sua versão atual, o modelo não possui nenhum componente ou tela para exibir a saída de informações, sendo necessário observar diretamente o que está sendo gravado na memória e outros componentes para verificar seu funcionamento.

Um componente adicional que foi inserido no sistema é um componente para entrada de teclado, que grava informações inseridas pelo usuário e que podem ser acessadas pelo resto do sistema. A implementação deste componente é rudimentar, servindo apenas como prova de conceito, mas pode ser aprimorada nas etapas seguintes de desenvolvimento do

modelo, permitindo uma interação direta com o sistema sem a necessidade de manipulação da memória.

Apesar de o modelo de simulação estar completo, devido a limitações de tempo no desenvolvimento, não foi possível criar uma rotina de tratamento de *overflow* do contador de número da janela de registradores, um componente essencial para a execução de programas que exigem chamadas de funções na memória. Esta rotina, implementada na memória do sistema como um programa, permite que mais instruções de chamadas sejam realizadas sem sobrescrever e perder dados, garantindo que o usuário não precise interferir diretamente no seu programa criado.

Completando este passo metodológico, pode-se considerar o modelo de simulação um sucesso, apesar de algumas melhorias possíveis. Este modelo servirá como uma base sólida para o desenvolvimento de um novo modelo, objetivo da próxima etapa deste trabalho. As melhorias necessárias serão implementadas ao longo desse processo, garantindo que a arquitetura criada funcione como um ponto de partida eficiente para a criação de uma arquitetura didática e seu modelo de simulação.

4.3 Assembler Base

De forma paralela à criação do modelo de simulação, foi desenvolvido um *assembler* para facilitar a criação de programas a serem testados no modelo de simulação. Assim como o modelo de simulação, essa versão do *assembler* serviu como base para desenvolvimentos futuros, sendo necessário modificá-lo conforme o trabalho progredia.

Em sua primeira versão, o *assembler* foi capaz de traduzir todas as instruções da linguagem *assembly* criada para gerar código de máquina compatível com o RISC I. Além disso, o *assembler* possuía recursos como criação de variáveis, criação de rótulos (*labels*) para endereços e suporte a números hexadecimais.

A sintaxe aceita pelo *assembler* pode ser expressa pela seguinte notação BNF:

```

1 <instrução> ::= <formato1> | <formato2> | <rotulo> | <declaracaoPalavra>
2
3 <formato1> ::= <mnemônico> <regidorCondicao> <regidor>
4   <regidorValor>
5
6 <formato2> ::= <mnemônico> <regidorCondicao> <valor>
7
8 <rotulo> ::= ":" <enderecoDeRotulo>
9
10 <declaracaoPalavra> ::= ".word" [a-zA-Z][a-zA-Z0-9]* <valor>
11
12 <mnemônico> ::= "CALLI" | "CALL" | "JMP" | "CALLR" | "JMPC" | "SLL" |
13   "GETPSW" | "SRL" | "PUTPSW" | "SRA" | "LDBU" | "LDRBU" | "LDBS" |
```

```

    "LDRBS" | "LDW" | "LDRW" | "LDSU" | "LDRSU" | "LDSS" | "LDRSS" |
    "STS" | "STRS" | "STB" | "STRB" | "STW" | "STRW" | "AND" | "XOR" |
    "OR" | "SUB" | "SUBC" | "SUBR" | "SUBCR" | "ADD" | "ADDC" | "RET" |
    "RETI" | "GETLPC" | "LDHI"

12 <registradorCondicao> ::= <registrador> | <condição>
13
14 <registrador> ::= "R" <número> | "r" <número>
15
16 <registradorValor> ::= <registrador> | <valor>
17
18 <valor> ::= <número> | "#" <númeroHexadecimal> | ":" <enderecoDeRotulo>
19   | "." <enderecoDePalavra>
20
21 <condição> ::= "NEVER" | "GREATER" | "LESS_EQUAL" | "GREATER_EQUAL" |
22   "LESS" | "HIGHER" | "LOWER_SAME" | "CARRY_CLEAR" | "LOWER" |
23   "CARRY_SET" | "HIGHER_OR_SAME" | "POSITIVE" | "NEGATIVE" |
24   "NOT_EQUAL" | "EQUAL" | "OVERFLOW_CLEAR" | "OVERFLOW_SET" | "ALWAYS" |
25   | "NEV" | "GT" | "LE" | "GE" | "LT" | "HI" | "LOS" | "NC" | "LO" |
26   "C" | "HIS" | "PL" | "MI" | "NE" | "EQ" | "NV" | "V" | "ALW"
27
28 <número> ::= [0-9] +
29 <númeroHexadecimal> ::= [0-9a-fA-F] +
30
31 <enderecoDeRotulo> ::= [a-zA-Z][a-zA-Z0-9]*
32
33 <enderecoDePalavra> ::= [a-zA-Z][a-zA-Z0-9]*

```

Para utilizar o *assembler*, um arquivo .asm contendo o código do programa deve estar presente na pasta onde o *assembler* foi configurado. Ao ser executado, o *assembler* mostra todos os arquivos presentes e solicita a seleção de um para compilação. Mensagens de depuração de erros são exibidas e, ao final, o resultado é mostrado e gravado em disco. Este resultado pode então ser carregado diretamente no modelo de simulação, conforme descrito na Seção 4.2.

4.4 Relatório Técnico

Um Relatório Técnico no contexto acadêmico é um documento que detalha os métodos, resultados e conclusões de um projeto ou pesquisa. Na área de computação e arquitetura e organização de computadores, ele aborda desde a descrição de arquiteturas e algoritmos até a análise de desempenho e implementação de sistemas. Este modelo foi adotado por (PEEK, 1983) em sua apresentação do RISC I, o que inspirou a utilização deste modelo para a criação da documentação do que foi feito para este trabalho.

O relatório técnico criado, intitulado "Uma Implementação do RISC-I utilizando o simulador Logisim Evolution", focou em mostrar as diferenças de implementação entre a implementação original do RISC I e a implementação realizada no Logisim. Também foram detalhados alguns aspectos do RISC I além da implementação do *assembler*. Diferentemente de um artigo, um relatório técnico não possui normas técnicas exatas a serem seguidas, o que facilitou e acelerou o processo de escrita. No entanto, foram mantidos alguns padrões da norma ABNT de artigo para garantir que o documento permanecesse organizado.

Diversas partes do relatório técnico serviram de base para a documentação didática que foi criada nos próximos passos da monografia. Além disso, o relatório pode ser facilmente adaptado para uma publicação formal, caso haja interesse. Uma das grandes motivações para a realização deste relatório técnico foi a necessidade de documentar o que foi feito de maneira detalhada e precisa, o que se encaixa perfeitamente no último passo metodológico realizado durante o TCC 1.

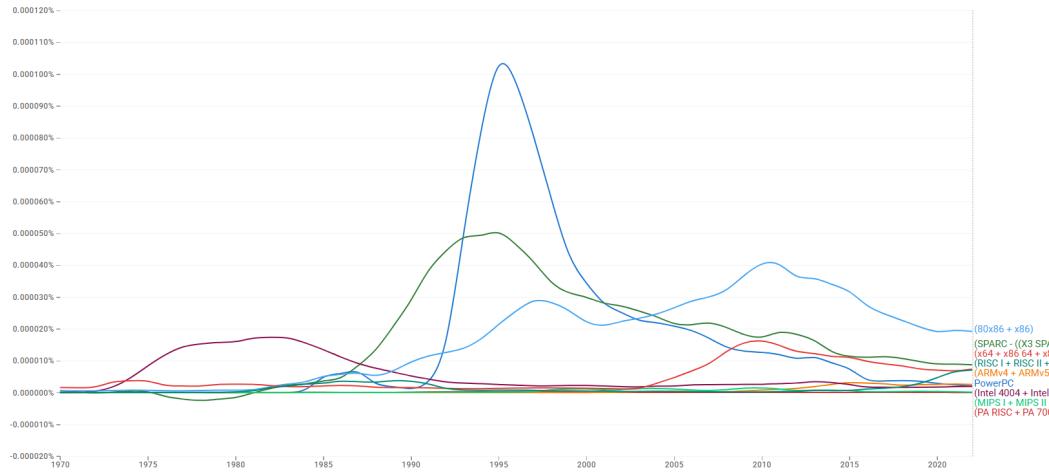
4.5 Bibliometria sobre arquiteturas

O primeiro passo da segunda parte deste trabalho foi a criação de um artigo intitulado "Uma Webibliometria Sobre a Relevância de Arquiteturas Computacionais" ([VALADARES, 2025](#)). Esta bibliometria foi realizada com base no trabalho de [Costa \(2010\)](#), porém com algumas alterações, sendo a maior delas a mudança na forma de pesquisa dos termos, resultando na utilização de dois tipos diferentes de pesquisa.

O primeiro tipo de pesquisa envolveu o uso da ferramenta *Google Ngram Viewer*, que implementa o trabalho de [Lin et al. \(2012\)](#). Em resumo, esta ferramenta permite a visualização da frequência do uso de um n-gram ao longo do tempo, onde n-gram é um conjunto de n palavras, números, símbolos e outros termos, com um limite de até 5 termos por n-gram. Esta pesquisa ofereceu uma visualização da relevância e popularidade de diversas arquiteturas ao longo dos anos, mas não permitiu uma comparação direta entre as arquiteturas pesquisadas devido à sua precisão e a certos problemas na filtragem([VALADARES, 2025](#)).

O segundo tipo de pesquisa foi similar ao de [Costa \(2010\)](#), porém com foco na pesquisa das arquiteturas em apenas duas bases de dados, o *Google Scholar* e o Periódicos CAPES, aplicando diferentes formas de filtragem e tabulando a quantidade de resultados. O objetivo dessa segunda pesquisa foi obter uma forma de comparação da relevância entre arquiteturas, complementando questões que não puderam ser analisadas durante a primeira pesquisa.

A Figura 5 mostra o resultado da pesquisa das arquiteturas no *Google Ngram Viewer*, enquanto as Tabelas 2 e 3 apresentam os resultados das pesquisas no *Google Scholar*. A Tabela 3 exibe uma porcentagem correspondente à quantidade de resultados

Figura 5 – Resultado das pesquisas no *Google Ngram Viewer* (VALADARES, 2025)

Arquitetura	Termo Individual	Architecture	Processor	Instruction Set	Computing	#	Média	Desvio Padrão	Embedded System	SoC	Microarchitecture
ARM	5850000	3240000	1630000	2180000	4340000	2847500	1198537	3090000	2660000	42800	
MIPS	649000	99000	94300	106000	208000	126825	54330	119000	60900	18800	
x86	171000	117000	109000	84200	129000	109800	18943	68500	30700	22400	
SPARC	261000	76000	62600	56500	113000	77525	24803	61900	64200	9100	
PowerPC	92300	61400	67700	55100	74200	64600	8211	45900	23000	8290	
x64	74700	25000	21600	15500	36500	24650	8824	15300	18200	1130	
RISC-V	26400	22200	22200	19000	22100	21375	1584	15700	11700	6420	
Intel(pré x86')	96300	16800	4850	18600	16800	14263	6332	6380	22700	70	
ARMv8	13000	10200	9190	7300	12000	9673	1963	7740	7260	2670	
DLX	49300	8830	3590	7220	17500	9285	5899	8620	16600	987	
PA-RISC	8870	7970	8330	7110	8180	7898	545	3020	1820	4110	
RISC I	11800	1330	1020	1160	1990	1375	429	572	1410	275	
Berkeley RISC	723	692	706	653	699	688	24	334	259	474	
RISC II	994	642	585	575	663	616	43	219	472	181	

Tabela 2 – Resultado das pesquisas do nome da arquitetura com termos adicionais para filtragem no *Google Scholar*(VALADARES, 2025)

Arquitetura	Termo Individual ²	% Architect	% Proces	% Instruction	% Comput	Média	Desvio Padrão	% Embedded Syst	% SoC	% Microarchitectu
Berkeley RISC	723	95,7%	97,6%	90,3%	96,7%	95,1%	3,3%	46,2%	35,8%	65,6%
PA-RISC	8870	89,9%	93,9%	80,2%	92,2%	89,0%	6,1%	34,0%	20,5%	46,3%
RISC-V	26400	84,1%	84,1%	72,0%	83,7%	81,0%	6,0%	59,5%	44,3%	24,3%
ARMv8	13000	78,5%	70,7%	56,2%	92,3%	74,4%	15,1%	59,5%	55,8%	20,5%
PowerPC	92300	66,5%	73,3%	59,7%	80,4%	70,0%	8,9%	49,7%	24,9%	9,0%
x86	171000	68,4%	63,7%	49,2%	75,4%	64,2%	11,1%	40,1%	18,0%	13,1%
RISC II	994	64,6%	58,9%	57,8%	66,7%	62,0%	4,3%	22,0%	47,5%	18,2%
ARM	5850000	55,4%	27,9%	37,3%	74,2%	48,7%	20,5%	52,8%	45,5%	0,7%
x64	74700	33,5%	28,9%	20,7%	48,9%	33,0%	11,8%	20,5%	24,4%	1,5%
SPARC	261000	29,1%	24,0%	22,4%	43,3%	29,7%	9,5%	23,7%	24,6%	3,5%
MIPS	649000	15,3%	14,5%	16,3%	32,0%	19,5%	8,4%	18,3%	9,4%	2,9%
DLX	49300	17,9%	7,3%	14,6%	35,5%	18,8%	12,0%	17,5%	33,7%	2,0%
Intel(pré x86')	96300	17,4%	5,0%	19,3%	17,4%	14,8%	6,6%	6,6%	23,6%	0,1%
RISC I	11800	11,3%	8,6%	9,8%	16,9%	11,7%	3,6%	4,8%	11,9%	2,3%

Tabela 3 – Resultado relativo das pesquisas do nome da arquitetura com termos adicionais para filtragem no *Google Scholar*(VALADARES, 2025)

com filtragem utilizando termos adicionais em relação aos resultados sem filtragem.

Após finalizada, a bibliometria ofereceu uma visão geral sobre a popularidade de algumas arquiteturas, o que permitiu uma escolha mais fácil de possíveis inspirações. Dentre as arquiteturas, duas se destacaram: ARM e RISC-V.

A família de arquiteturas ARM está consolidada em uma grande parte do mercado, totalizando cerca de 50% de todos os processadores disponíveis mundialmente (ARM,). As arquiteturas ARM também são RISC (ARM, 2014), tornando-as relevantes para este

trabalho devido a algumas semelhanças com a arquitetura Berkeley RISC utilizada no circuito base. Apesar disso, é uma arquitetura comercial que não é aberta, o que poderia dificultar o seu uso acadêmico.

Já o RISC-V ([RISC-V, 2024b](#)) não é exatamente uma arquitetura, mas um padrão aberto que especifica um conjunto de instruções modular. Apesar de não ser um padrão da indústria para computadores comerciais ou dispositivos móveis atualmente, ele tem ganhado rapidamente popularidade nos últimos anos devido à sua proposta, sendo já implementado em diversos microcontroladores e recebendo atualizações contínuas.

O padrão RISC-V é sucessor das arquiteturas Berkeley RISC (RISC I, RISC II, SOAR e SPUR ([RISC-V, 2024b, p.10](#))), surgindo também na Universidade de Berkeley, mas logo se tornando uma entidade própria.

A proposta do RISC-V é descrita em seu manual:

RISC-V (pronunciado "risk-five") é uma nova arquitetura de conjunto de instruções (ISA) que foi originalmente projetada para apoiar a pesquisa e a educação em arquitetura de computadores, mas que agora esperamos que também se torne um padrão livre e aberto para implementações industriais. Nossos objetivos ao definir o RISC-V incluem:

- Uma ISA completamente aberta que esteja livremente disponível para a academia e a indústria.
- Uma ISA real adequada para implementação direta em hardware nativo, não apenas simulação ou tradução binária.
- Uma ISA que evita "super-arquiteturas" para um estilo particular de microarquitetura (por exemplo, microcódigo, in-order, desacoplado, out-of-order) ou tecnologia de implementação (por exemplo, full-custom, ASIC, FPGA), mas que permita uma implementação eficiente em qualquer uma dessas.
- Uma ISA separada em uma pequena ISA base de inteiros, utilizável por si só como base para aceleradores personalizados ou para fins educacionais, e extensões padrão opcionais, para suportar o desenvolvimento de software de uso geral.
- Suporte para o padrão IEEE-754 de 2008 revisado para ponto flutuante. (ANSI/IEEE Std 754-2008, IEEE Standard for Floating-Point Arithmetic, 2008)
- Uma ISA que suporta extensões ISA extensivas e variantes especializadas.
- [...]

([RISC-V, 2024b](#)).

Esta proposta e os objetivos mostrados se alinham com os objetivos e passos metodológicos deste trabalho, e por isso o RISC-V foi escolhido como padrão principal a ser seguido durante a segunda fase deste trabalho.

4.6 Listagem de Recursos Para Implementação

Antes de começar a modificar a arquitetura base, foi feita uma listagem de possíveis recursos e melhorias, que pode ser vista nas tabelas 4 e 5. Essas listas foram montadas observando recursos das arquiteturas pesquisadas ao longo deste trabalho que não estavam presentes no RISC I.

A Tabela 4 está dividida em colunas que incluem o nome do recurso, a avaliação da prioridade dos recursos a serem implementados e uma última coluna exemplificando arquiteturas que implementam esse recurso. A avaliação de prioridade considera a dificuldade de implementação (A), a relevância do recurso para a arquitetura (B) e a relevância didática (C). Esses critérios foram definidos com os valores: Muito Baixa (1), Baixa (2), Média (3), Alta (4) e Muito Alta (5). Esses valores são utilizados para calcular uma pontuação total com a fórmula $(6-A)+B+C-3$, gerando valores de 0 a 12, onde quanto maior a pontuação, melhor.

Recursos/Melhorias	Dificuldade de Implementação	Relevância na Arquitetura	Relevância didática	Pontuação	Ocorrências
Remoção das janelas de registradores	Muito Baixa	Alta	Alta	10	ARM, MIPS, RISC-V
Data Forwarding para instruções de Load	Alta	Muito Alta	Muito Alta	9	-
Predição de pulo	Alta	Muito Alta	Muito Alta	9	-
Formato de Instrução próximo ao RISC-V	Média	Muito Alta	Alta	9	RISC-V
Instruções BRANCH	Média	Muito Alta	Alta	9	RISC-V
Padronização dos sinais de ativação entre componentes	Baixa	Muito Alta	Média	9	-
Pipeline Flush	Baixa	Muito Alta	Média	9	-
Pontos Flutuantes	Alta	Muito Alta	Muito Alta	9	x86,x64,ARM, RISC-V
Reorganização das Instruções	Baixa	Muito Alta	Baixa	8	-
Padronização de Nomes	Baixa	Média	Alta	8	-
Modo Kernel/Privilegiado	Alta	Muito Alta	Alta	8	RISC II, RISC-V, ARM
Multithreading	Muito Alta	Muito Alta	Muito Alta	8	RISC-V, ARM, X86, X65
Cache	Muito Alta	Muito Alta	Muito Alta	8	MIPS,x86,x64,ARM,RISC-V
Paginação de Memória	Muito Alta	Muito Alta	Muito Alta	8	-
Melhoria de Entrada e Saída de Informações para o Usuário	Média	Alta	Média	7	-
Novo Sistema de Interrupções e Traps	Alta	Alta	Alta	7	-
Mover os registradores dos próximos endereços do arquivo de registradores para o controlador	Muito Baixa	Média	Baixa	7	-
Conversão para HDL	Média	Baixa	Muito Alta	7	-
Proteção de Memória Física(PMP)	Média	Alta	Média	7	-
Multiplicação e Divisão	Média	Média	Média	6	MIPS,DLX,x86,x64,ARM, RISC-V
Pipeline de 3 passos	Alta	Alta	Média	6	RISC II
Tornar compatível com FPGA	Baixa	Baixa	Média	6	-
Interface de expansão	Muito Alta	Muito Alta	Média	6	MIPS
Pipeline de 5 passos	Alta	Alta	Baixa	5	MIPS, Intel486(x86)

Tabela 4 – Lista de possíveis recursos e melhorias para implementação no *assembler*

Já a Tabela 5 é similar a 4, porém é considerada a relevância para o *assembler*

no lugar da relevância na arquitetura e as ocorrências listadas são referentes a outras linguagens assembly.

Recursos/Melhorias	Dificuldade de Implementação	Relevância pro Assembler	Relevância didática	Pontuação	Ocorrências
Omissão de parâmetro	Baixa	Muito Alta	Alta	10	-
Pseudo-Instruções	Baixa	Muito Alta	Alta	10	-
Debug Log mais detalhado	Baixa	Alta	Muito Alta	10	-
Destaque de Sintaxe	Média	Média	Alta	7	-
Macros	Média	Média	Média	6	-
Define	Baixa	Média	Média	7	-

Tabela 5 – Lista de possíveis recursos e melhorias para implementação no *assembler*

4.6.1 Recursos Para a Arquitetura

Esta seção apresenta uma descrição de cada recurso da arquitetura na Tabela 4. Recursos que dependem de outros são descritos após suas dependências.

4.6.1.1 Remoção das Janelas de Registradores

As janelas de registradores são um sistema que, em teoria, melhora a performance do processador, além de tornar seu funcionamento mais próximo do *software*. Apesar das aparentes melhorias de desempenho, esse recurso não é encontrado em arquiteturas modernas como ARM ([ARM, 2014](#)), RISC-V ([RISC-V, 2024b](#)) ou até mesmo em arquiteturas um pouco mais antigas como MIPS ([MIPS Tech LLC, 2016](#)).

Isso se deve às suas desvantagens, que incluem a utilização de uma grande porção do circuito, redução da velocidade de *clock* devido aos longos caminhos de barramentos, e o fato de que, em programas que exigem muitas chamadas, ao exceder o número de janelas, as rotinas de salvamento dessas janelas acabam negando os ganhos de performance ([PATTERSON, 1985](#)).

Portanto, a remoção das janelas de registradores seria uma grande otimização para o circuito, além de simplificar a compreensão didática do Arquivo de Registradores.

4.6.1.2 Formato de Instrução Próximo ao RISC-V

O padrão RISC-V, como mencionado anteriormente na Seção 4.5, tem ganhado popularidade rapidamente, e por ser um padrão aberto, ele se torna uma ótima referência. Além disso, a forma como as instruções são organizadas traz diversas vantagens, que podemos listar:

- Posição dos endereços dos registradores consistente entre instruções, simplificando a decodificação ([RISC-V, 2024b](#)).
- Valores imediatos de algumas instruções alinhados aos *bytes*, facilitando a visualização dentro da instrução.

- Redução da quantidade de *opcodes* necessários, em favor da utilização de códigos adicionais que representam diferentes opções por instrução.
- A utilização de um padrão amplamente adotado, facilitando a disponibilidade de conteúdo sobre o tema.
- O RISC I, segundo [Patterson \(2017\)](#), é possivelmente a arquitetura mais próxima do RISC-V. Em sua comparação, ele cita as seguintes similaridades:
 - Um espaço de endereço endereçável por *bytes* de 32 bits.
 - Todas as instruções têm 32 bits de comprimento.
 - 31 registradores, com o registrador 0 fixo em zero, todos com 32 bits de largura.
 - Todas as operações são registrador-para-registrador (nenhuma é registrador-para-memória).
 - As mesmas operações aritméticas, lógicas e de deslocamento.
 - As mesmas instruções de carregar e armazenar *words*.
 - Versões com e sem sinal de carregar e armazenar *bytes* e *half words* (chamadas de "short" no RISC-I).
 - Opção imediata para todas as instruções aritméticas, lógicas e de deslocamento.
 - Imediatos são sempre estendidos com sinal.
 - Um modo de endereçamento de dados (registrador + imediato).
 - Endereçamento relativo ao contador de programa para saltos.
 - Sem instruções de multiplicação ou divisão.
 - Uma instrução para carregar um imediato largo na parte superior do registrador, para que uma constante de 32 bits ocupe apenas duas instruções.

[Patterson \(2017\)](#)

Estas semelhanças facilitam a aplicação do padrão RISC-V ao RISC I.

4.6.1.3 Reorganização das Instruções

A reorganização das instruções pode envolver a alteração dos *opcodes* de cada instrução, com o objetivo de simplificar tanto a decodificação quanto a visualização do conjunto. Uma abordagem eficiente seria adotar a organização utilizada no RISC II, que apresenta uma ordenação mais direta e uma divisão mais clara entre os diferentes grupos de instruções. Essa melhoria torna-se implícita na implementação do padrão RISC-V, que já incorpora tais princípios de organização.

4.6.1.4 Padronização de Nomes

No momento, o circuito criado não possuía um padrão bem definido de nomes, o que dificultava a compreensão da funcionalidade de certas entradas e saídas de alguns componentes. A padronização definiria como esses nomes devem ser escritos para a melhoria da compreensão da função dos componentes e para a melhoria da consistência geral.

4.6.1.5 Padronização dos sinais de ativação entre componentes

A temporização do circuito não está padronizada. Há alguns componentes que se ativam durante a descida do sinal de *clock* enquanto outros se ativam durante a subida. Estes sinais devem ser alterados para melhoria da consistência no funcionamento do circuito e para a prevenção de erros.

4.6.1.6 Modo *Kernel*/Privilegiado

A inclusão de um Modo *Kernel* em uma arquitetura didática é fundamental para ilustrar conceitos essenciais de segurança, controle e organização do sistema. Esse recurso permite demonstrar, de forma clara, como o processador diferencia entre código confiável e não confiável, como ocorrem as transições entre aplicações e o sistema operacional, e como se estabelece o isolamento entre diferentes camadas de software.

Mesmo em ambientes simplificados, compreender os modos de operação ajuda os estudantes a visualizar a estrutura real dos sistemas computacionais modernos e a reconhecer a importância da proteção de recursos e da gestão de privilégios. Além disso, esse conceito reforça a noção de que o sistema operacional atua como uma camada intermediária crítica entre o *hardware* e as aplicações, sendo responsável por garantir o funcionamento seguro e eficiente do sistema.

O Recurso 4.6.1.2 especifica o funcionamento do modo *kernel*, e poderia ser seguido para esta implementação, resultando em um controle mais padronizado. Esse seria um ótimo recurso para ser ensinado, pois evidencia um dos principais mecanismos de *hardware* para o gerenciamento de um sistema operacional.

4.6.1.7 Novo Sistema de Interrupções e *Traps*

A implementação de um sistema de *traps* em uma arquitetura didática é essencial para demonstrar como o processador lida com eventos excepcionais e solicitações de serviços do sistema operacional. Esse mecanismo permite que o fluxo de execução seja transferido de forma controlada para rotinas específicas, como chamadas de sistema ou tratamento de erros, garantindo que o sistema possa responder adequadamente a situações inesperadas ou a requisições de alto nível feitas por programas em modo usuário.

Além de reforçar a separação entre os modos de operação, o sistema de *traps* exemplifica como o *hardware* e o sistema operacional colaboram para manter a estabilidade, a segurança e a previsibilidade da execução. Para fins didáticos, esse recurso é valioso por permitir que os estudantes compreendam o papel das interrupções e exceções no funcionamento de sistemas reais, e como essas estruturas são fundamentais para o controle de acesso a recursos privilegiados.

4.6.1.8 Pipeline Flush

O *pipeline flush* é uma técnica utilizada para limpar a *pipeline* do processador quando ocorre uma mudança no fluxo de controle, como um pulo condicional ou uma interrupção. Quando uma instrução de pulo é executada, as instruções que foram buscadas e decodificadas, mas ainda não executadas, podem não ser mais válidas, pois o fluxo de execução mudou para um novo endereço.

Para evitar a execução de instruções inválidas, o *pipeline flush* limpa todas as instruções que estão atualmente na *pipeline*, garantindo que apenas as instruções do novo fluxo de controle sejam executadas. Isso é feito invalidando as instruções nas etapas da *pipeline* e reiniciando o processo de busca de instruções a partir do novo endereço de destino.

A implementação do *pipeline flush* na arquitetura envolveria a adição de lógica de controle que detecta quando uma mudança no fluxo de controle ocorre e sinaliza para as etapas da *pipeline* que as instruções devem ser invalidadas. Essa técnica é essencial para manter a integridade do fluxo de execução e evitar erros causados pela execução de instruções incorretas.

4.6.1.9 Melhoria na Entrada e Saída de Informações para o Usuário

Durante a replicação do RISC I, foram exploradas algumas formas rudimentares de interface de entrada e saída. No entanto, é desejável desenvolver mecanismos mais eficientes de interação com o sistema, tanto para fins didáticos, demonstrando o funcionamento dos dispositivos de entrada, quanto para aplicações práticas.

Para entrada de dados pelo usuário, podem ser utilizados dispositivos como teclado, *joystick* e *sliders*. Já para saída, opções incluem um terminal e uma tela gráfica simples.

4.6.1.10 Mover os registradores dos próximos endereços do Arquivo de Registradores para o controlador

Esta mudança tem como foco a exclusão dos registradores de endereço localizados dentro do Arquivo de Registradores. Com a adoção do padrão RISC-V, esses registradores já seriam modificados a favor das diferenças no funcionamento de determinadas instruções.

No entanto, a principal motivação foi a simplificação da forma como as instruções em execução são armazenadas no controlador.

Em vez de distribuir uma instrução por diferentes partes do processador, é possível concentrar todas as informações em um único controlador. Essa abordagem facilita o gerenciamento e a visualização das instruções em execução, além de centralizar os sinais de controle de maneira mais eficiente.

4.6.1.11 Multiplicação e Divisão

Multiplicação e divisão são operações que geralmente não são abordadas em arquiteturas didáticas. A adição dessas operações pode oferecer uma perspectiva sobre o funcionamento delas dentro de uma arquitetura. O RISC-V possui uma extensão, denominada M, que implementa ambas as operações e pode ser seguida para a implementação. No entanto, caso a divisão se revele difícil para implementação, é possível adicionar apenas as operações de multiplicação, que no RISC-V estão dentro do sub-conjunto da extensão M, denominado Zmmul.

4.6.1.12 Pontos Flutuantes

Similar a multiplicação e divisão, operações com pontos flutuantes deveriam ser abordadas dentro de arquiteturas didáticas. Estas operações são definidas em diversas extensões do RISC-V, porém o foco principal seria a extensão F, que define as operações com precisão única(32 bit), descrita pelo padrão IEEE 754-2008([RISC-V, 2024b](#)). As extensões que definem outros tamanhos de ponto flutuantes podem ser consideradas para implementações futuras, porém a implementação mínima apenas requer precisão única.

4.6.1.13 Pipeline de 3 passos

O RISC I possui uma *pipeline* de 2 passos, onde o primeiro é a coleta da instrução e decodificação enquanto a segunda é a execução. Esta *pipeline* pode ser otimizada ao adicionar um terceiro passo, que executa o segundo ciclo de execução de instruções de **load** e **store** em paralelo com os outros ciclos.

Este tipo de *pipeline* foi executada no RISC II, e melhora execução destas instruções, porém ela pode introduzir dependência de dados que devem ser tratadas por software ou hardware.

4.6.1.14 Encaminhamento de dados para Instruções de load

Data forwarding ou Encaminhamento de dados é uma técnica utilizada para resolver dependências de dados em pipelines via hardware, especialmente em instruções de **load**. No caso da *Pipeline* de 3 passos mencionada anteriormente, as instruções de **load** demoram

um ciclo adicional para a gravação do dado no registrador, o que causa a instrução que vem a seguir utilizar o antigo valor desse registrador.

Ao encaminhar dados diretamente das etapas de carregamento para as etapas de execução de instruções subsequentes, podemos evitar atrasos causados por acessos à memória, melhorando a eficiência geral do *Pipeline* e evitando problemas. Caso este recurso seja difícil para implementação, um aviso deve ser adicionado no *assembler*, indicando quaisquer conflitos de dados.

4.6.1.15 *Pipeline* de 5 passos

Uma *pipeline* de 5 passos é uma evolução da *pipeline* de 3 passos, adicionando mais estágios para aumentar o paralelismo e, consequentemente, o desempenho do processador. Os cinco estágios típicos são:

- **Busca de Instrução (IF):** Localiza e carrega a próxima instrução da memória.
- **Decodificação (ID):** Interpreta a instrução e identifica os operandos necessários.
- **Execução (EX):** Realiza a operação indicada pela instrução.
- **Acesso à Memória (MEM):** Lê ou escreve dados na memória, quando necessário.
- **Escrita de Resultado (WB):** Armazena o resultado da operação nos registradores.

Esse tipo de *pipeline* é implementado em arquiteturas como a MIPS([MIPS Tech LLC, 2016](#)) e fornece uma melhoria na performance. No entanto, pode aumentar a complexidade do circuito, o que pode dificultar o entendimento didático do funcionamento da *pipeline*. Caso seja implementada, esse aspecto deve ser considerado.

Vale destacar que os estágios de busca de instrução e acesso à memória podem gerar conflitos quando executados simultaneamente, já que ambos requerem acesso à memória principal. Para evitar esse problema, é necessário empregar técnicas que permitam o acesso paralelo ou escalonado à memória, garantindo o funcionamento correto da *pipeline*.

4.6.1.16 Instruções *branch*

As instruções de pulo condicional do RISC I exigem que primeiro uma operação seja executada como subtração ou soma para que códigos de condição sejam definidos. Esses códigos são então testados utilizando uma condição selecionada na instrução do pulo.

Este processo pode ser simplificado por um tipo de operação chamada de *branch*, que executa uma subtração seguida dos testes de condição, tudo dentro de um ciclo da mesma instrução, tornando o processo bem mais direto. Este tipo de pulo condicional é

definido no conjunto de instrução base do RISC-V junto com instruções para pulos não condicionais.

4.6.1.17 Predição de pulo

Predições de pulo, assim como o *data forwarding*, são formas de otimizar a *pipeline* e evitar *hazards* na arquitetura. Na predição de pulo, durante a fase de coleta da próxima instrução, se a instrução atual for um pulo, a instrução a ser coletada deve ser a do endereço definido pelo pulo.

Esse processo pode ser simples para pulos não condicionais, pois eles sempre são executados. No entanto, para instruções de pulos condicionais, esse processo é mais complexo, pois só é possível saber se o pulo será tomado após a verificação da condição, que geralmente ocorre ao fim do ciclo, deixando pouco ou nenhum tempo para obter dados do endereço do pulo.

Uma possível solução é coletar a instrução tanto do endereço seguinte ao pulo quanto do endereço definido pelo pulo. No entanto, isso pode exigir um aumento na quantidade de passos por ciclo da *pipeline*, reduzindo a performance geral.

O mecanismo de predição de pulo possui uma alta relevância didática, pois mostra uma das possíveis otimizações que são implementadas em *pipelines*. Caso implementada, a predição de pulo torna o recurso de *pipeline flush*(4.6.1.8) obsoleto.

4.6.1.18 Conversão para HDL

A conversão da arquitetura para linguagens de descrição de hardware como SystemVerilog ou VHDL permite a implementação e simulação em nível de hardware de forma mais eficiente que a simulação em simuladores com interface gráfica como o Logisim.

Isso é essencial para validar a funcionalidade e desempenho da arquitetura de forma automatizada, bem como para realizar síntese e implementação em FPGA o que exige a descrição por essas linguagens.

4.6.1.19 Tornar compatível com FPGA

Tornar a arquitetura compatível com FPGA envolve não só a descrição por linguagens HDL, mas também uma adaptação do design para que possa ser sintetizado e implementado em dispositivos FPGA. Isso inclui a otimização do uso de recursos de hardware disponíveis no FPGA, remoção de componentes que não são compatíveis com FPGAs e a verificação da funcionalidade por meio de simulações e testes no hardware.

Uma interface física também deve ser implementada para que o usuário possa interagir com o processador, o que pode requerer a implementação de protocolos de comunicação de dispositivos de entrada e saída como I²C, SPI, UART ou USB.

A realização desse passo agrega valor ao projeto por diversos motivos. Primeiramente, permite a validação prática da arquitetura em um ambiente físico, essencial para verificar o funcionamento real do sistema além da simulação, fortalecendo a confiabilidade dos resultados obtidos.

Além disso, a implementação em FPGA possibilita a interação direta com periféricos reais, ampliando o escopo do projeto e permitindo demonstrar aplicações concretas. Essa etapa também evidencia o domínio de ferramentas e práticas utilizadas na indústria de sistemas digitais, tornando o projeto mais robusto e relevante tanto em contextos acadêmicos quanto profissionais.

4.6.1.20 Interface de Expansão

Uma interface de expansão permitiria a adição modular de componentes ao processador, como co-processadores (por exemplo, FPUs, TPUs ou GPUs). Essa interface é essencial para aumentar a flexibilidade e a capacidade de processamento do sistema, diferindo da interface com dispositivos de entrada e saída, pois é projetada para comunicação direta e de alta velocidade entre o processador principal e os módulos de expansão.

Para implementar a interface de expansão, é necessário definir um protocolo de comunicação eficiente entre o processador e os módulos de expansão. Esse protocolo deve incluir a definição de comandos específicos para inicialização, leitura e escrita de dados, interrupções e os sinais e barramentos que serão compartilhados entre o processador e os dispositivos de expansão.

4.6.1.21 *Multithreading*

Do ponto de vista educacional, a implementação de multithreading em uma arquitetura de processador oferece uma oportunidade para explorar conceitos fundamentais de concorrência, paralelismo e gerenciamento de contexto. Ao lidar com múltiplos fluxos de execução, o estudante é exposto a desafios reais enfrentados em arquiteturas modernas, como escalonamento de tarefas, sincronização e compartilhamento de recursos.

Do ponto de vista arquitetural, implementar multithreading exige a criação de mecanismos para gerenciamento de contexto, como registradores duplicados, controle de escalonamento e sincronização entre threads. Embora isso aumente a complexidade do design, também proporciona uma oportunidade valiosa de explorar conceitos avançados de arquitetura de processadores.

A inclusão de multithreading agrega valor ao projeto por permitir a análise de desempenho em cenários reais, além de demonstrar a escalabilidade da arquitetura. Academicamente, essa funcionalidade evidencia o domínio de técnicas modernas de projeto de sistemas computacionais e aproxima o trabalho das práticas adotadas em processadores

comerciais.

4.6.1.22 Memória Cache

A memória *cache* é um recurso presente na maioria dos sistemas computacionais modernos, sendo essencial para aumentar a eficiência de acesso à memória principal pelo processador.

Além de acelerar a leitura e escrita de dados, a separação entre dados e instruções proporcionada pelo *cache* L1 viabiliza a implementação de uma pipeline de cinco estágios, ao eliminar conflitos de acesso à memória durante a busca de instruções e operações com dados.

4.6.1.23 Paginação de Memória

Para implementar a paginação de memória no *hardware*, é necessário incluir uma Unidade de Gerenciamento de Memória (MMU). A MMU é responsável por traduzir endereços virtuais em endereços físicos. Os passos que estariam envolvidos nesse processo são:

1. **Adicionar uma MMU:** Esse componente ficaria entre o processador e a memória. Ele traduz os endereços virtuais gerados pelos programas em endereços físicos na memória RAM.
2. **Configurar Tabelas de Páginas:** Tabelas de páginas seriam criadas para mapear os endereços virtuais para os endereços físicos. Cada entrada na tabela de páginas contém o endereço físico correspondente e permissões de acesso.
3. **Gerenciar Falhas de Página:** Um mecanismo para lidar com falhas de página teria que ser implementado. Quando uma página solicitada não estiver na memória física, a MMU deve sinalizar uma falha de página, e o sistema operacional deve carregar a página do disco rígido para a memória.
4. **Permissões e Proteção:** As permissões de acesso na tabela de páginas devem ser configuradas para garantir que cada processo tenha acesso apenas às suas próprias páginas, evitando erros.

Esse recurso seria relativamente complexo e depende de vários outros como o modo *kernel* e *cache*, porém devido à sua importância, é um ótimo recurso a ser considerado caso haja tempo para a sua implementação.

4.6.2 Recursos Para a Assembler

4.6.2.1 Omissão de parâmetro

A omissão de parâmetro é um recurso que facilita a criação de código assembly, permitindo que o programador não precise preencher parâmetros que não são usados pela instrução. Isso simplifica a escrita do código, tornando-o mais legível e menos propenso a erros. Por exemplo, em uma instrução que não utiliza um segundo operando, o programador pode omitir esse parâmetro, e o *assembler* tratará a ausência de forma adequada, utilizando um valor padrão ou ignorando o parâmetro omitido.

4.6.2.2 Pseudo-Instruções

Para auxiliar na programação do código assembly, pseudo-instruções podem ser criadas. Esse é um recurso presente em diversas linguagens assembly que traz a possibilidade de emular instruções não existentes na arquitetura a partir de outras.

Diversos exemplos de pseudo-instruções são mostrados no manual do RISC-V ([RISC-V, 2024b](#)), como a instrução de negação (NOT), que pode ser sintetizada por um ou exclusivo (XOR), ou instruções de desvio condicional (BLT, BGE), que podem ser invertidas ao alterar a ordem dos operandos (BGT, BLE).

Para que seja possível a implementação de pseudo-instruções, uma reestruturação do *assembler* pode ser necessária. Esse recurso pode também completar a omissão de parâmetros, uma vez que pseudo-instruções podem ser definidas com diferentes quantidades de parâmetros necessários.

4.6.2.3 Debug Log mais detalhado

```

1 Current line: AND R1 R1 #FFFFFFC7F
2
3 10000000000000000000000000000000 Instruction AND(2164260864) added to
   [27] with:
4 00000000000100000000000000000000 Destination: 00080000
5 00000000000000010000000000000000 Source 1: 00004000
6 0000000000000001110001111111 Immediate: 00001c7f

```

Figura 6 – Exemplo da impressão de uma instrução realizada pelo *assembler*

Durante a primeira versão do *assembler*, informações sobre a montagem do código eram exibidas. No entanto, essas informações não eram intuitivas e eram incompletas, conforme pode ser observado na Figura 6.

O *assembler* deveria fornecer informações mais diretas e apresentar mensagens de erro claras. Algumas melhorias que poderiam ser implementadas incluem:

- Impressão de uma linha por tradução, que deve exibir o resultado da tradução e comparação com o código original.
- Uma melhor detecção de erros sintáticos.
- Adição de detecção de erros semânticos, que incluiria avisos ao utilizar recursos não inicializados ou acesso a endereços inválidos.
- Impressão dos passos de criação e tradução de rótulos e variáveis.
- Detecção de *hazards* na pipeline

4.6.2.4 Destaque de Sintaxe

O destaque de sintaxe é uma funcionalidade que melhora a legibilidade e a compreensão do código-fonte ao aplicar diferentes cores e estilos de formatação a elementos distintos da linguagem de programação. Isso inclui palavras-chave, variáveis, strings, comentários e outros componentes sintáticos.

A principal vantagem do destaque de sintaxe é facilitar a identificação de diferentes partes do código, tornando mais fácil detectar erros e entender a estrutura e a lógica do programa. Além disso, o destaque de sintaxe pode acelerar o processo de escrita de código, pois ajuda os programadores a reconhecer rapidamente os elementos da linguagem e a evitar erros comuns.

Implementar o destaque de sintaxe em qualquer IDE que possa ser utilizada para programar o código assembly pode melhorar significativamente a experiência do usuário, tornando a escrita e a leitura do código assembly mais intuitivas e eficientes. Isso é especialmente útil em ambientes educacionais, onde a clareza e a facilidade de uso são essenciais para o aprendizado.

4.6.2.5 Macros

Macros são uma ferramenta em linguagens de montagem que permitem a definição de sequências de instruções que podem ser reutilizadas em diferentes partes do código. Elas funcionam como uma espécie de função ou procedimento, onde um conjunto de instruções é agrupado sob um nome específico. Quando esse nome é chamado no código, o conjunto de instruções associado é inserido no lugar.

As macros ajudam a reduzir a repetição de código, facilitam a manutenção e melhoram a legibilidade do código assembly. Elas são especialmente úteis para operações complexas ou frequentemente repetidas, permitindo que o programador defina a sequência de instruções uma vez e a reutilize conforme necessário.

A implementação de macros no *assembler* envolve a criação de um pré-processador que expande as macros antes da montagem do código. Isso significa que, durante a fase de

pré-processamento, o *assembler* substitui todas as chamadas de macros pelas instruções definidas na macro, gerando o código final que será montado.

4.6.2.6 Define

O recurso *define* é uma funcionalidade encontrada em *assembly* e outras linguagens de programação que permite a definição de constantes ou macros. Em um *assembler*, o *define* é usado para atribuir um nome a um valor constante ou a uma sequência de instruções, facilitando a leitura e a manutenção do código.

Por exemplo, em um *assembler*, você pode definir uma constante para um endereço de memória específico:

1 `#define START_ADDRESS 0x1000`

Isso permite que você use *START_ADDRESS* em vez de 0x1000 em todo o seu código, tornando-o mais legível e fácil de modificar. Se o endereço precisar ser alterado, você só precisará atualizar a definição, em vez de procurar e substituir todas as ocorrências do valor no código.

O uso de *define* melhora a clareza e a manutenção do código, permitindo a criação de nomes significativos para valores constantes.

4.7 Implementação dos Recursos e Melhorias

Apesar do planejamento na metodologia indicar a implementação de cada recurso em ciclos separados, pelo tipo de relação entre alguns desses recursos, parte das implementações ocorreu em paralelo. Nesta seção será descrito em ordem de implementação todos os recursos que foram adicionados. Caso um recurso descrito em 4.6 seja citado, a seção que o descreve será exibida entre parênteses, como exemplo o cache(4.6.1.22);

4.7.1 Processo de Desenvolvimento - Arquitetura

4.7.1.1 Primeira Fase: Alterações Iniciais

As primeiras melhorias focaram em tornar o sistema compatível com o conjunto de instruções RISC-V (4.6.1.2), começando pelo *assembler* descrito na Seção 4.7.2. Após as adaptações realizadas no *assembler*, a primeira melhoria implementada foi a remoção das janelas de registradores (4.6.1.1), que simplificou significativamente o circuito necessário e eliminou a necessidade de rotinas de tratamento de interrupções relacionadas ao *overflow* do número de janelas. Isso também economizou tempo ao remover a necessidade de adaptar esses recursos para o novo conjunto de instruções.

Para a remoção das janelas de registradores, três componentes precisaram ser modificados: o Arquivo de Registradores, a Palavra de Status do Processador e o Controlador - enquanto o Controlador do Número de Janela foi removido totalmente.

O Arquivo de Registradores sofreu a maior modificação, que envolveu a remoção do controle relacionado às janelas, deixando apenas uma versão expandida dos registradores globais, renomeada como *Registers Bank* (Banco de Registradores). Adicionalmente, os registradores de endereço foram movidos para o controlador (4.6.1.10), aproveitando que ambos os componentes já estavam sendo modificados.

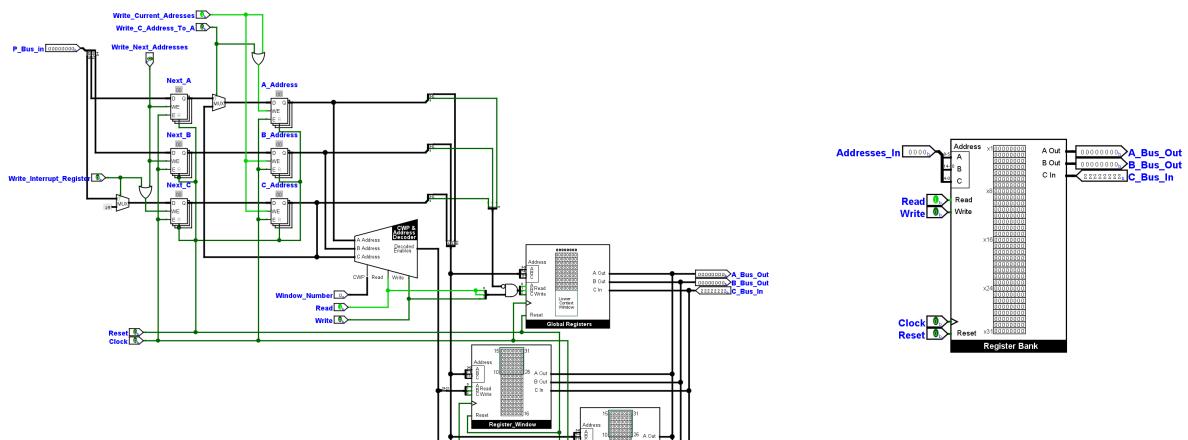


Figura 7 – Arquivo de Registradores antes (esquerda) e depois (direita) da modificação.

A Figura 7 mostra o resultado das modificações realizadas no Arquivo de Registradores. É possível notar que, neste ponto, o Banco de Registradores possui as mesmas entradas e saídas que o Arquivo de Registradores, o que o torna apenas um encapsulador. Isso foi feito considerando que o Arquivo de Registradores poderia ser expandido em alguma etapa seguinte para suportar pontos flutuantes, criando a necessidade de outro Banco de Registradores dentro do Arquivo de Registradores.

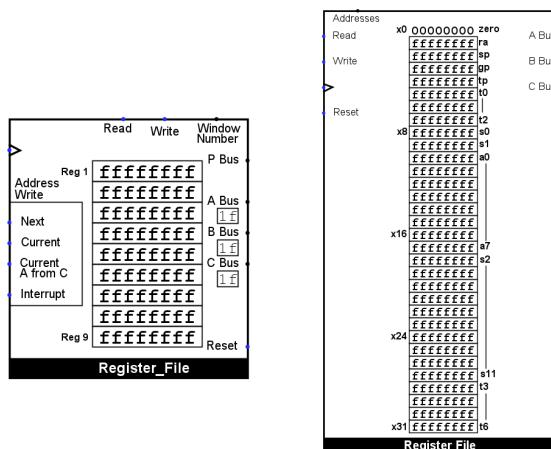


Figura 8 – Encapsulamento do Arquivo de Registradores antes (esquerda) e depois (direita) da modificação.

A Figura 8 mostra as alterações feitas no encapsulamento do componente. A remoção das janelas permitiu que todos os registradores do arquivo fossem mostrados. Foram atribuídos nomes aos registradores que indicam o uso recomendado, seguindo a especificação do RISC-V (2024b). Com todas as modificações, a interface do Arquivo de Registradores também foi simplificada, facilitando o controle necessário para utilizá-lo.

A Palavra de Status do Processador (PSW) no RISC-I servia apenas a duas funções: armazenar um espelhamento dos números de janela e o código CNZV (*carry*, negativo, zero e *overflow*) resultante da ALU. Com a remoção do Controlador do Número de Janela, parte da função do PSW foi eliminada, e como as instruções do RISC-V não necessitam do armazenamento dos códigos CNZV, a PSW tornou-se um componente desnecessário para o sistema, permitindo sua remoção.

Para adaptar o Controlador às novas modificações, seus registradores de instrução foram alterados, e a decodificação e transmissão de sinais para controle do número de janela e do PSW foram removidas. Em paralelo, o conjunto de instruções do RISC-V (4.6.1.2, consequentemente 4.6.1.3) estava sendo adicionado ao modelo. Ao adaptar o conjunto de instruções do RISC-I para o RISC-V, notou-se que o circuito necessário para o controle tornou-se bem mais simples, como pode ser visto nas Figuras 9 e 10.

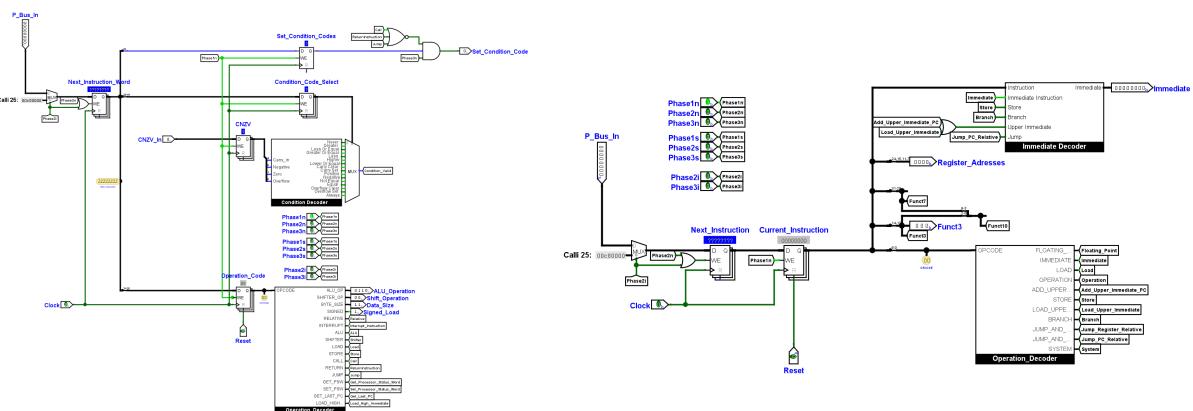


Figura 9 – Primeira fase de decodificação de instruções. Antes (esquerda) e depois (direita).

Essa simplificação se deve a alguns fatores. O primeiro fator é que o conjunto de instruções base do RISC-V possui uma quantidade menor de instruções em comparação com o RISC-I, o que simplifica o decodificador de instruções. Apesar de possuir menos instruções, as capacidades desse conjunto são equivalentes às do RISC-I, pois cada instrução pode possuir mais funções, que são definidas por códigos dentro da palavra da instrução, o que nos leva ao segundo fator.

Os códigos de função, por estarem separados do *opcode*, podem ser passados diretamente para os componentes do processador, com o mínimo de decodificação necessário. Esses códigos são sempre definidos nas mesmas posições, facilitando o processo de extração. Isso pode ser visto na Figura 11, que mostra a posição do *funct3* e do *funct7*.

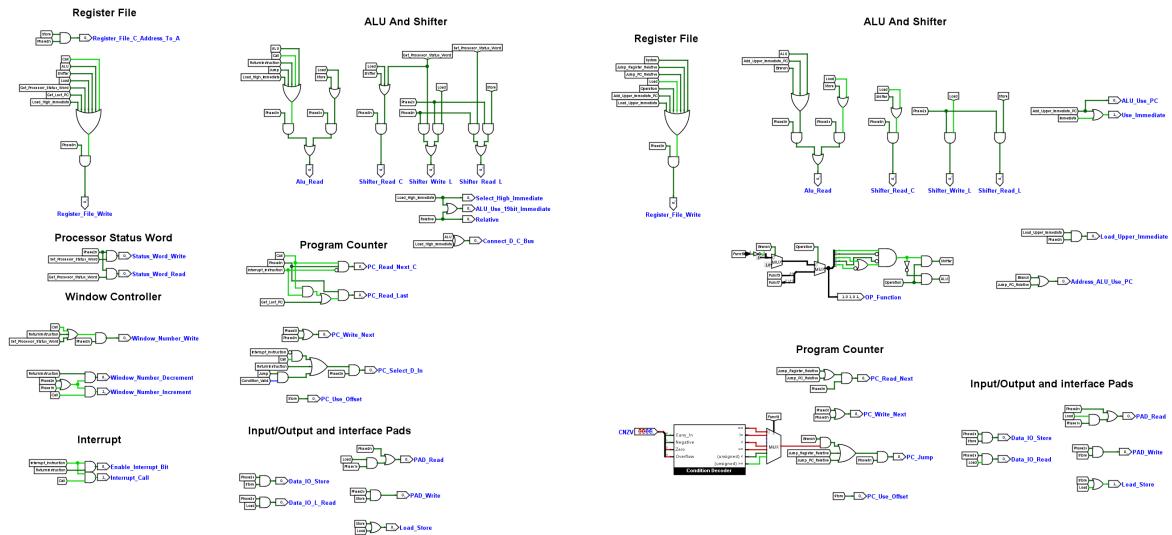


Figura 10 – Segunda fase de decodificação de instruções. Antes (esquerda) e depois (direita).

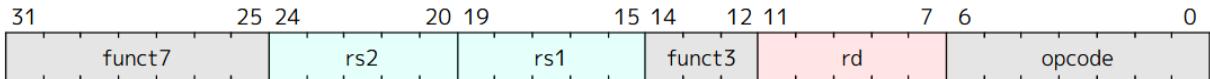


Figura 11 – Formato de instrução do RISC-V para operações entre registradores ([RISC-V, 2024b](#))

Nem todas as instruções e componentes necessitam do *funct7* para funcionar, por isso há uma separação. Exemplos do que esses códigos podem representar são: operações lógico-aritméticas (adição, subtração, *or*, *and*, deslocamento de bits, etc.), tamanho e sinalização de um valor (*byte*, *short*, *word*, com ou sem sinal) e condições (maior que, igual, diferente, etc.).

Outra alteração necessária para tornar o sistema compatível com o RISC-V é a forma como os valores imediatos são salvos e tratados. No RISC-I, há um componente exclusivo para o armazenamento e decodificação do valor imediato. Na nova versão, o registrador da instrução atual é aproveitado para obter o valor imediato (como pode ser visto na Figura 9), que é então decodificado seguindo a especificação do RISC-V.

Espero que essa versão expandida esteja mais clara e coesa. Se precisar de mais alguma modificação, estou à disposição para ajudar!

A Figura 12 mostra as diferenças entre os decodificadores. Além da alteração da forma de decodificação, a forma com que os valores são passados para os componentes também foi alterada, onde antes cada tipo tinha seu próprio barramento separado, agora todos os valores imediatos são estendidos para 32 bits e passados para um barramento unificado que passa em paralelo com os barramentos A, B e C, simplificando as conexões e controles necessários entre os componentes.

O recurso de pulsos condicionais([4.6.1.16](#)) também foi adicionado durante esta fase.

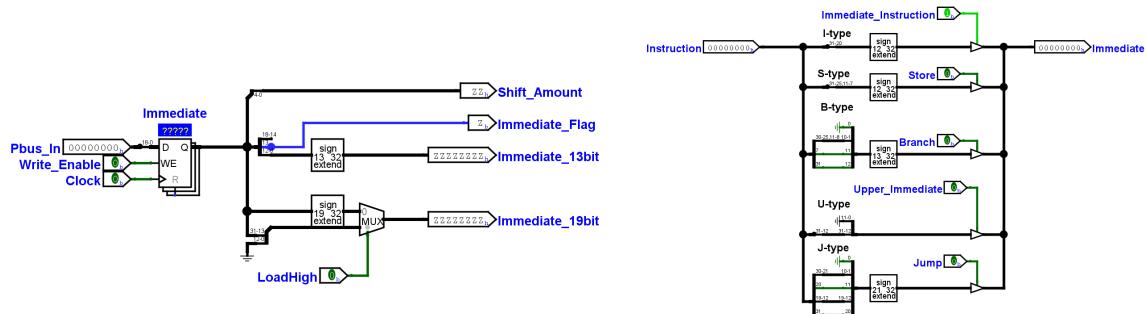


Figura 12 – Decodificador de imediato do RISC I(esquerda) em comparação com o novo decodificador(direita)

O RISC I possuía pulsos condicionais que dependiam de duas instruções, uma para o cálculo do código CNZV(geralmente subtração), e outra para decodificar este código e realizar o pulo caso resulte em um valor verdadeiro(instrução JUMP). O RISC-V, por outro lado, utiliza uma forma mais direta de pulo condicional utilizando a instrução BRANCH. Nessa instrução tanto a subtração quanto o teste de condição e execução do pulo ocorrem no mesmo ciclo. Para que isso fosse possível, uma segunda ULA com apenas adição conectada ao Contador de Programa foi criada e nomeada da ULA de Endereços(*Address ALU*). Além disso o código CNZV é passado da ULA diretamente para o Controlador, onde um novo decodificador de condição é utilizado.

O novo Decodificador de Condição possui menos condições que o do RISC I, pois diversas das condições podem ser sintetizadas alterando a ordem dos operadores. Essa diferença reduz o circuito necessário para decodificação e controle, o que pode ser observado nas Figuras 10 e 13;

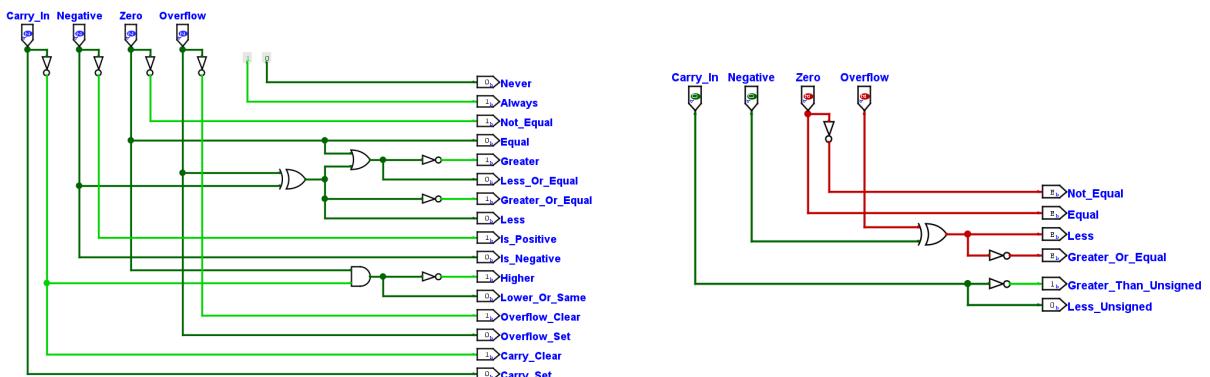


Figura 13 – Decodificador de condição antigo(esquerda) e novo(direita)

Outras modificações foram executadas durante essa fase para adaptar o circuito

ao novo controle. A ULA, não precisa mais da entrada *carry in* e os valores imediatos agora são passados por apenas uma entrada, operações de Multiplicação e Divisão(4.6.1.11) foram adicionadas, apesar de que, nos passos seguintes a divisão foi removida, e finalmente o seu controle foi atualizado.

O Deslocador sofreu uma pequena alteração na entrada de operação(que também foi alterada na ULA), e teve a sua saída para o barramento L movida para fora do componente, o que será pertinente para alterações em passos seguintes.

Por fim foi removido o controle da entrada de endereço do Contador de Programa, que no momento, recebia o valor a ULA de Endereços. Todas as alterações dessa fase podem ser vistas na Figura 14, com o resultado final mostrado na Figura 15. Vale notar que nesse momento a implementação do conjunto de instruções não estava completa, pois ainda não possuía instruções de sistema(*SYSTEM*), além de diversos recursos privilegiados da especificação.

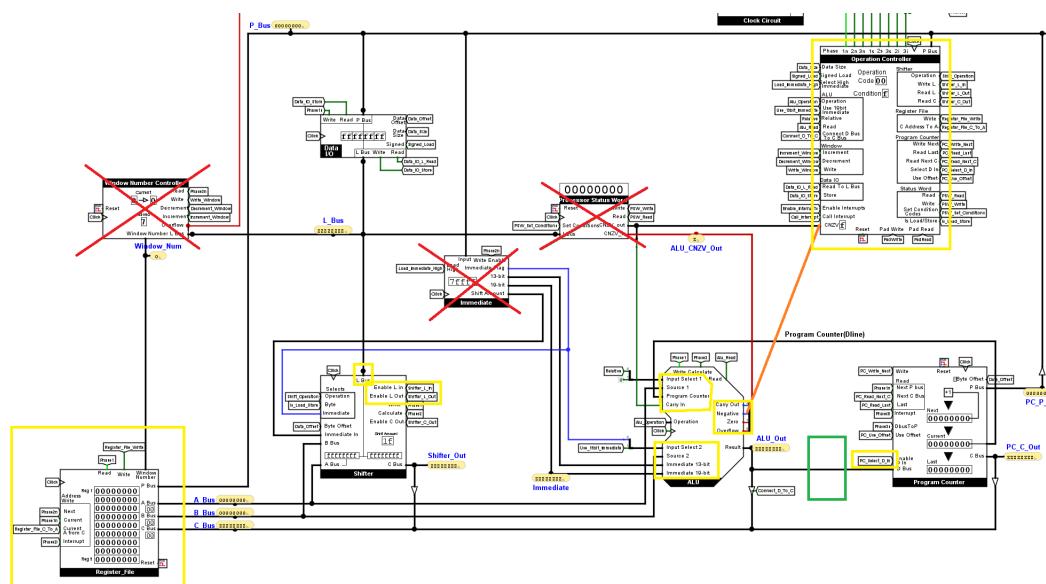


Figura 14 – Modelo original com modificações que foram executadas destacadas em amarelo, componentes que foram removidos cortados em vermelho e posições de novos componentes destacados em verde

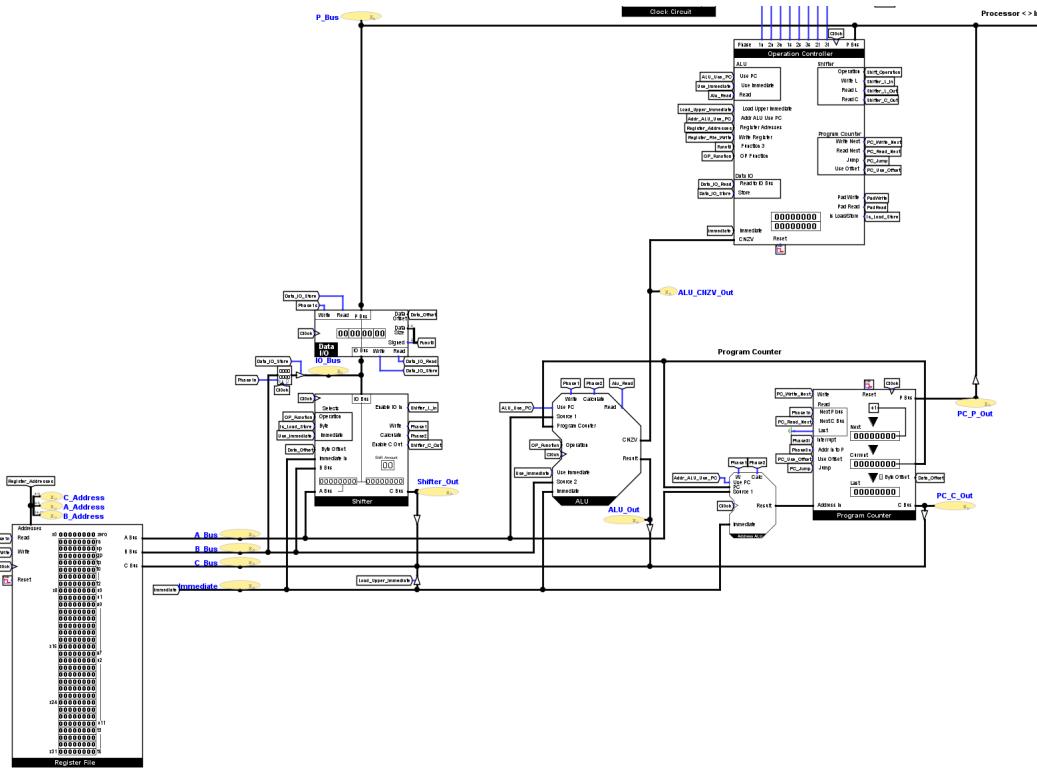


Figura 15 – Resultado da primeira fase de modificações da arquitetura

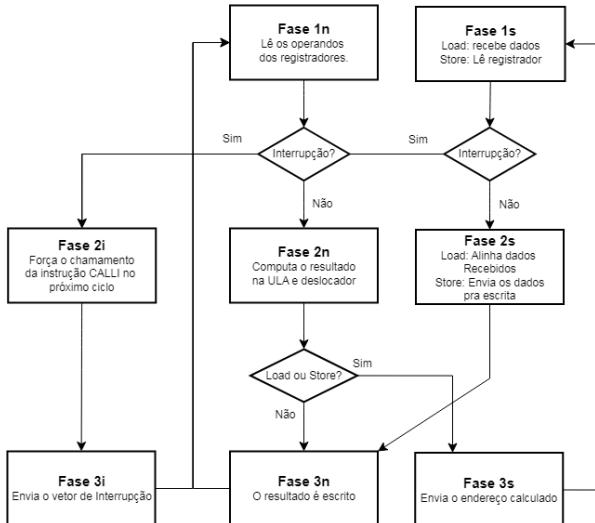
4.7.1.2 Segunda Fase: Pipeline e Usabilidade

Após as primeiras modificações para adaptar a arquitetura ao conjunto de instruções RISC-V, a *Pipeline* da arquitetura foi analisada para melhorias. No RISC I (PEEK, 1983), a *Pipeline* funciona em uma divisão de dois estágios que ocorrem durante três ciclos de *clock*, chamados de fases.

No primeiro estágio (I), ocorre a coleta da próxima instrução (*Instruction Fetch* ou IF) durante a fase 2n e a decodificação da instrução coletada anteriormente (*Instruction Decode* ou ID) durante a fase 1n. Após este estágio (E), uma sequência de operações ocorre, começando pela passagem de operandos, que saem do Arquivo de Registradores para os outros componentes através dos barramentos A e B, que então podem ser usados em operações matemáticas durante a fase 2n (*Execution* ou EX).

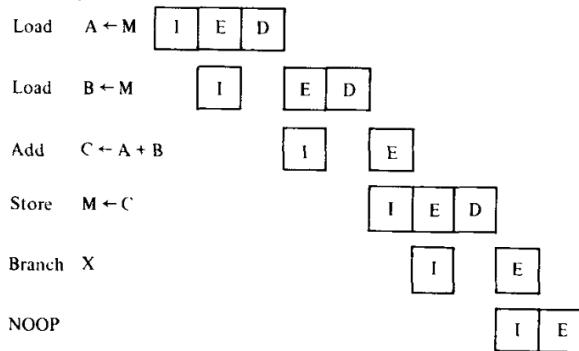
Para qualquer instrução que não acesse a memória, o segundo estágio termina na fase 3n, onde os resultados de quaisquer operações são escritos no Arquivo de Registradores através do barramento C (*Write Back* ou WB). Porém, caso a instrução acesse a memória, o segundo estágio é estendido. Essa extensão (D) envia o endereço que deve ser acessado na memória e então carrega ou envia os dados que precisam ser trocados (*Memory* ou MEM).

Tudo isso ocorre durante as fases que possuem o sufixo S, que podem ser vistas no fluxograma na Figura 16, mostrando todas as possíveis fases que ocorrem durante o

Figura 16 – Temporização do RISC I (adaptado de [Peek \(1983\)](#))

segundo estágio. As fases adicionais, com sufixo I, referem-se ao tratamento de interrupções; porém, como estas foram removidas durante a fase de desenvolvimento anterior, elas podem ser ignoradas.

A Figura 17 mostra como as instruções são divididas e executadas utilizando a *Pipeline* do RISC I. Dentro das divisões mostradas, o I corresponde ao primeiro estágio (IF + ID), o E corresponde ao segundo estágio (EX + WB) enquanto o estágio D corresponde à extensão que ocorre durante as instruções de carregamento e armazenamento (MEM).

Figura 17 – Demonstração da *pipeline* do RISC I. ([STALLINGS, 1988](#))

É visível que, em instruções que acessam a memória, a extensão de carregamento atrasa a coleta, decodificação e execução em um estágio da pipeline, tornando acessos à memória inefficientes na execução de programas. Durante o desenvolvimento do RISC II, isso foi corrigido, o que fez a extensão de acesso à memória ser executada em paralelo com os outros estágios da pipeline, conforme mostrado na Figura 18.

Neste momento, essa modificação pareceu simples o suficiente para ser implementada na arquitetura, o que levou à sua execução (4.6.1.13). A *Pipeline* poderia ser revisitada no futuro, dividindo os passos de IF e ID, e os passos de EX e WB, tornando-a uma *Pipeline*

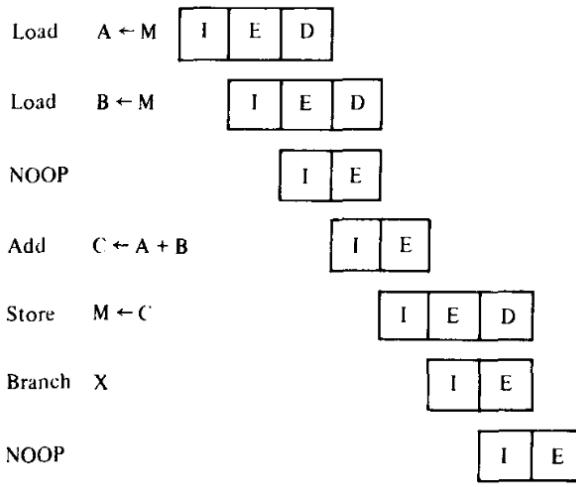


Figura 18 – Demonstração da *pipeline* do RISC II.(STALLINGS, 1988)

de cinco estágios.

O primeiro passo dessa modificação foi remover completamente o componente de Controle de *Clock*. Este componente já não possuía mais função na arquitetura, pois, com o sistema de interrupções removido e a eliminação da extensão nas instruções **load** e **store**, as únicas fases utilizadas seriam as fases 1n, 2n e 3n, que podem ser alimentadas diretamente por um gerador de fase externo (neste momento, três componentes de *Clock* com tempo e fase diferentes).

O controlador recebeu um novo registrador, que armazena a instrução executada anteriormente. Essa instrução é então decodificada entre **load** e **store**, gerando dois novos sinais de controle. Esses sinais substituem as ativações que ocorriam durante as fases S, além de gerarem uma substituição temporária do código *funct3*, que define o tamanho da informação a ser carregada pelo **load**, e do endereço dos registradores, que é modificado para conter o endereço que receberá a informação de **load**.

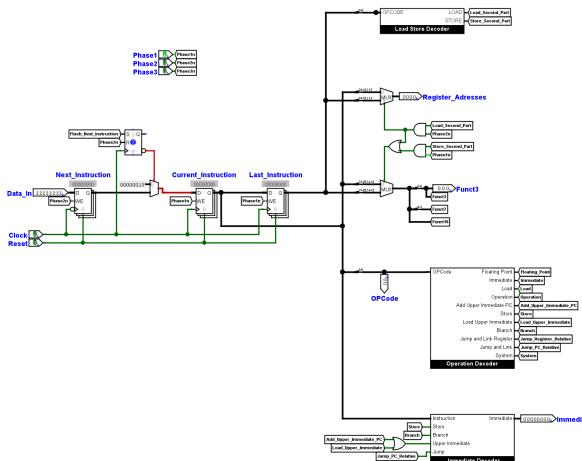


Figura 19 – Primeira fase de decodificação durante a segunda fase de desenvolvimento

A Figura 19 mostra o resultado das modificações realizadas na primeira fase de

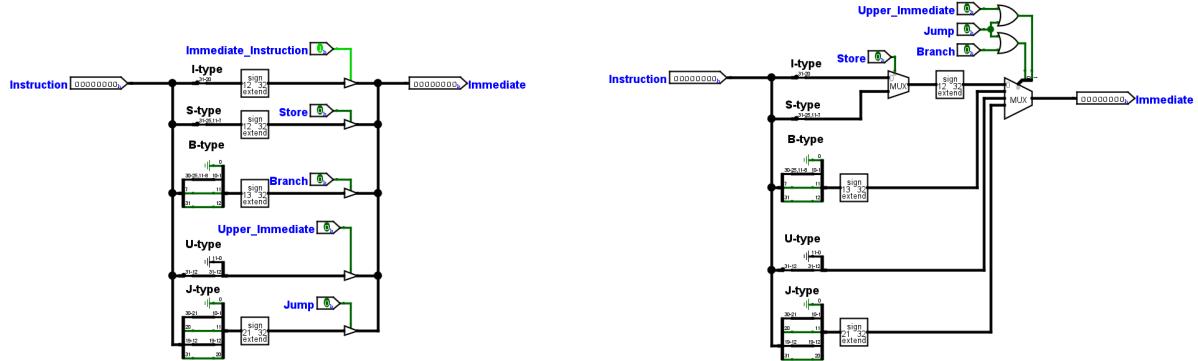


Figura 20 – Comparação entre o Decodificador de Imediato com *buffers tri-state* (esquerda) e multiplexadores (direita)

decodificação do Controlador. A organização dos componentes foi alterada para melhor separação e visualização. Além disso, vestígios do controle relacionado a interrupções foram removidos, e os Decodificadores de Operação e Imediato foram levemente modificados.

O Decodificador de Operação teve apenas seu encapsulamento alterado, com as saídas completamente visíveis. O Controlador de Imediato sofreu alterações mais significativas, com a substituição dos *buffers tri-state* por multiplexadores para a seleção do valor de saída. Esta mudança foi feita porque FPGAs geralmente não suportam *buffers tri-state* internamente. Além disso, permite que um valor padrão esteja sempre disponível; no caso, o imediato de 12 bits das instruções OP-IMM. A Figura 20 mostra o resultado dessas mudanças.

A Figura 21 mostra as alterações na segunda etapa de decodificação. Modificações adicionais relacionadas à *Pipeline* incluíram o descapsulamento do decodificador de condição e alterações no controle do Deslocador e Data IO, que foram separados durante essa fase.

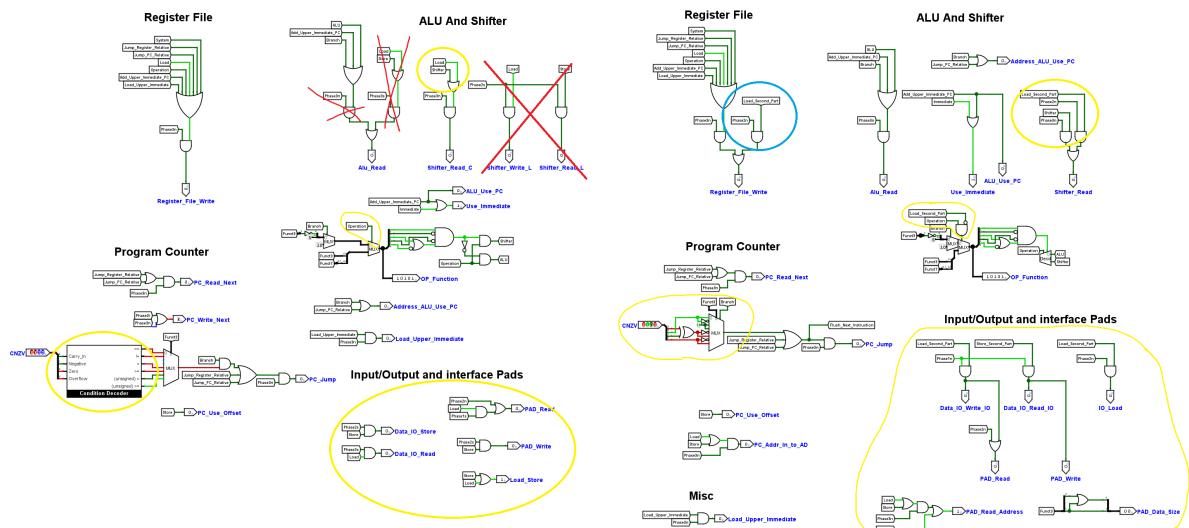


Figura 21 – Segunda fase de decodificação de instruções. Antes (esquerda) Depois (direita)

Neste estágio do desenvolvimento, mesmo após testes, um erro na decodificação do

OP_Function passou despercebido. O sinal da segunda fase do carregamento afetava a *Pipeline* durante toda a fase, quando deveria afetar apenas o segundo ciclo de *clock*. Esse erro foi corrigido em etapas posteriores.

O recurso de *Pipeline Flush* (4.6.1.8) foi adicionado para eliminar os *hazards* relacionados à execução de saltos como pode ser visto nas Figuras 19 e 21 (Acima da saída *PC_Jump*). No entanto, considerou-se que esse recurso poderia ser removido em favor de soluções mais eficientes, como predição de salto.

Para tornar a visualização da arquitetura mais amigável, foi realizada uma melhoria visual dos componentes, destacando e separando melhor as entradas e exibindo registradores que antes só poderiam ser vistos inspecionando os componentes. Todas as alterações podem ser vistas na Figura 22.

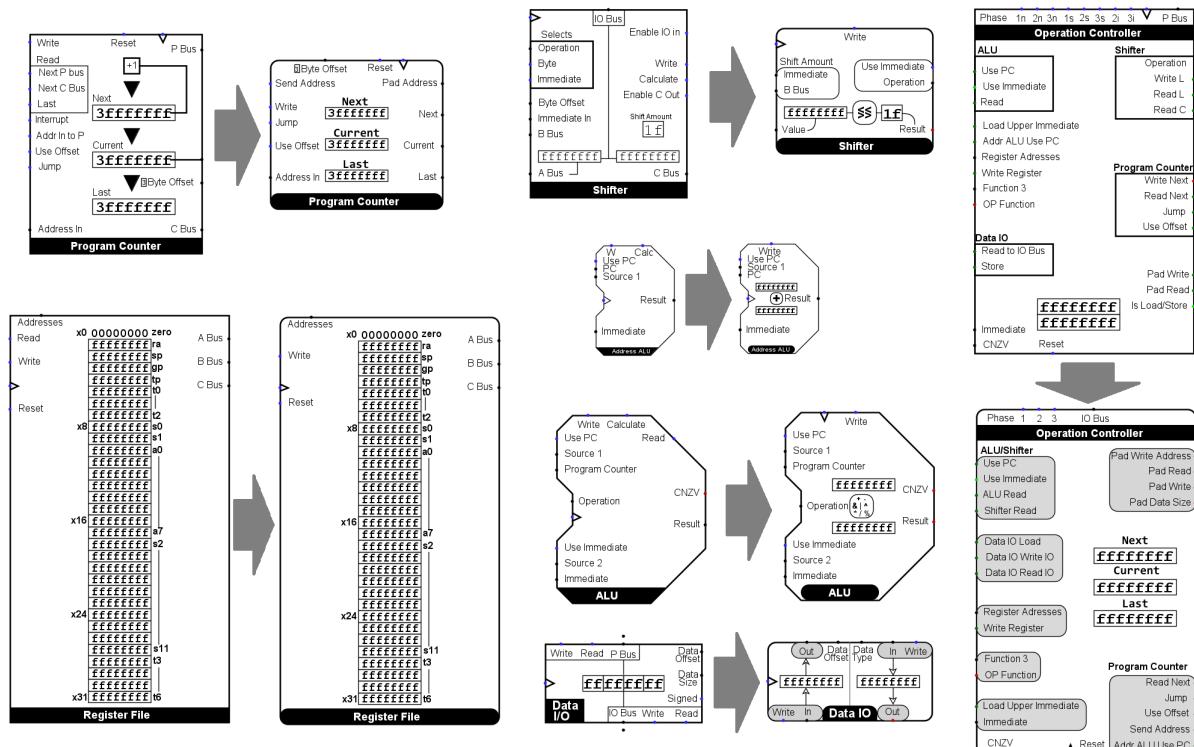


Figura 22 – Atualização no visual dos componentes

A Figura 22 também mostra a remoção dos sinais de cálculo e leitura. A ULA e o Deslocador possuíam registradores que armazenavam o resultado das operações, mas essa armazenagem não era necessária, pois o valor fica disponível até que os registradores de entrada sejam atualizados, permitindo tempo suficiente para ser escrito no Arquivo de Registradores. Essa remoção simplificou tanto a estrutura interna quanto os sinais de controle necessários para a execução.

Os sinais de leitura foram movidos para fora dos componentes, controlando agora uma série de multiplexadores conectados ao barramento C. A motivação dessa alteração foi a mesma realizada no Decodificador de Imediato, tornando o sistema mais compatível

com FPGAs. As modificações podem ser vistas na Figura 23.

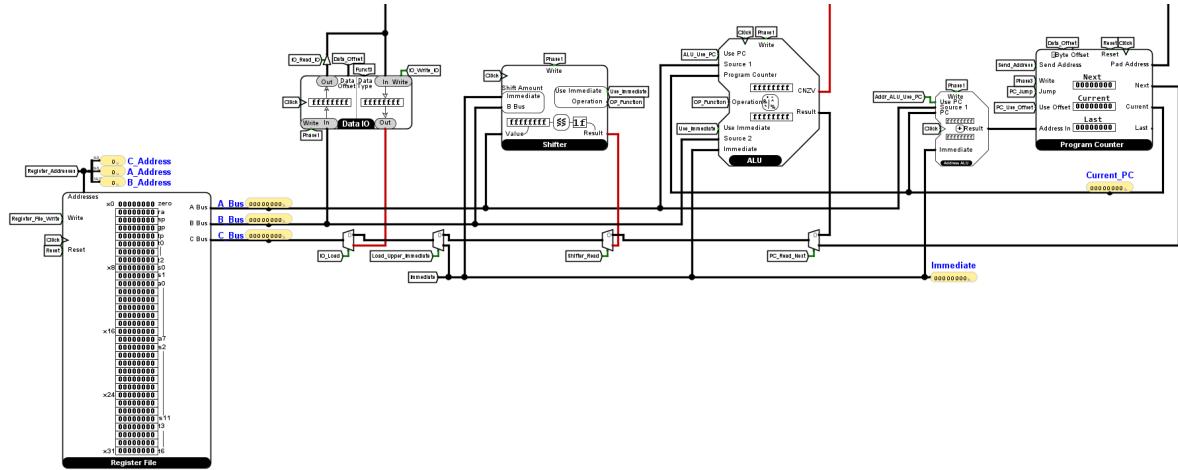


Figura 23 – Visualização dos componentes e suas conexões com os barramentos

Além disso, foi realizada a separação entre o *DataIO* e o Deslocador. No RISC I, esses componentes trabalham em conjunto durante as instruções de `load` e `store`, pois a informação recebida nas instruções de carregamento às vezes precisa ser deslocada. No entanto, essa integração adiciona complexidade ao controle e pode causar problemas futuros.

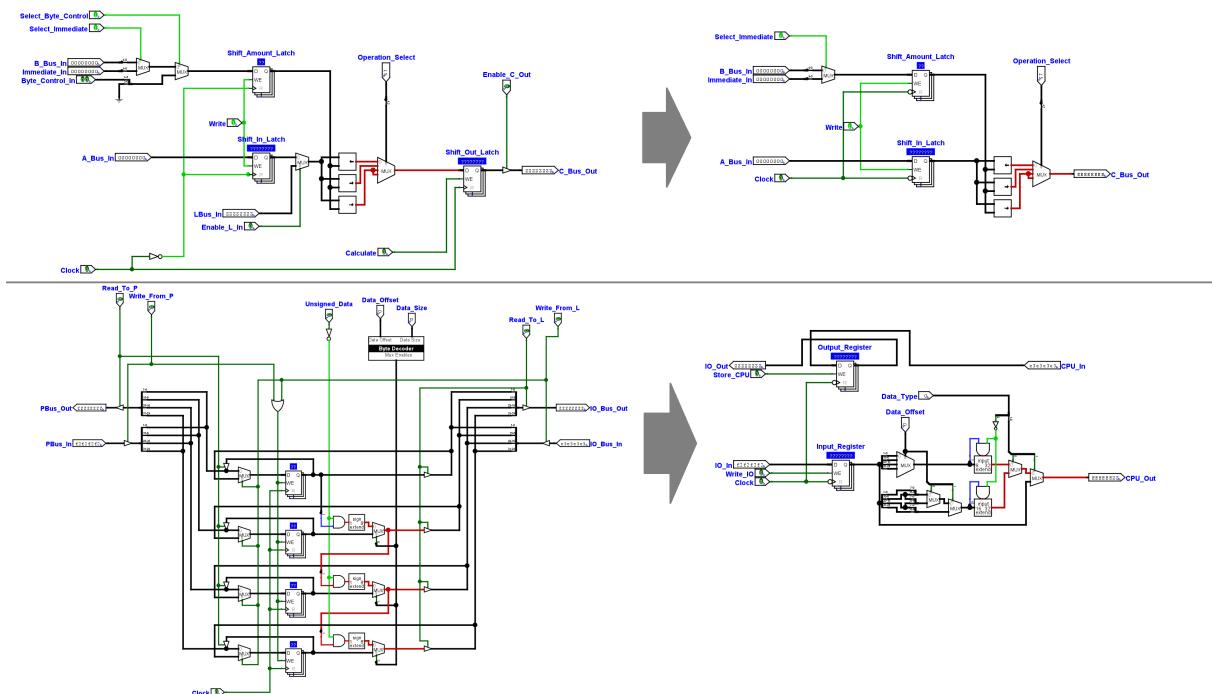


Figura 24 – Comparação entre o Deslocador (acima) e o DataIO (abaixo) antes e depois

A Figura 24 mostra que houve uma grande redução no circuito do Deslocador, enquanto o *DataIO*, apesar da redução visual, pode ter aumentado em complexidade devido à necessidade de mais multiplexadores com um maior número de entradas. Agora, o *DataIO* precisa realizar o deslocamento necessário.

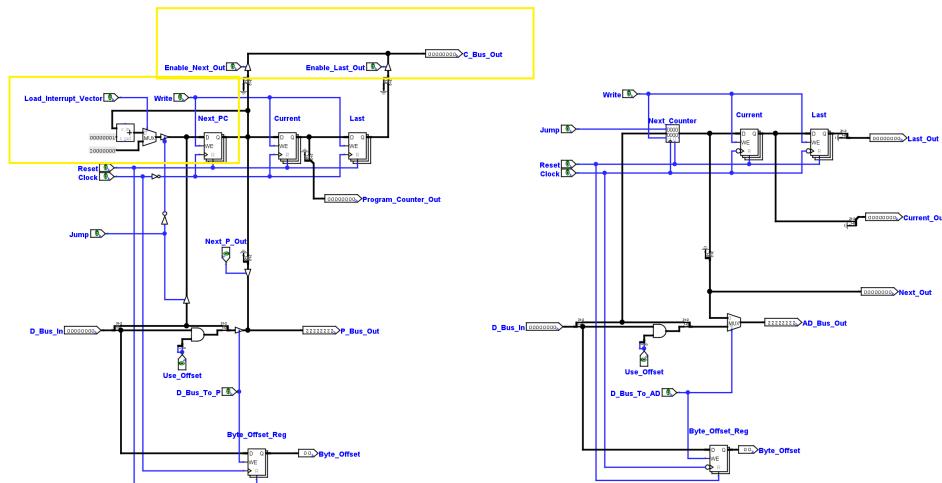


Figura 25 – Contador de Programa antes (esquerda) e depois (direita)

Além dessa separação, o *DataIO* teve os registradores utilizados durante o *load* e *store* e os caminhos dos dados carregados e salvos separados. Essa alteração, embora não estritamente necessária, facilita a compreensão do fluxo de dados no circuito.

A saída dos dados para o barramento P (agora chamado barramento de IO ou barramento de dados) ainda possuía um *buffer tri-state*, mas a remoção desse componente estava planejada.

No Contador de Programas, pequenas alterações foram feitas. O primeiro registrador foi alterado para um contador, o que simplifica visualmente os componentes, facilitando o entendimento do que está acontecendo. A saída para o barramento C foi trocada por uma saída para cada registrador, transferindo o controle de seleção para componentes externos. Essas alterações podem ser vistas na Figura 25.

Finalmente, durante essa etapa, os barramentos de endereço e dados foram separados (Figura 26), permitindo que dados e endereços sejam usados simultaneamente e evitando problemas na nova pipeline.

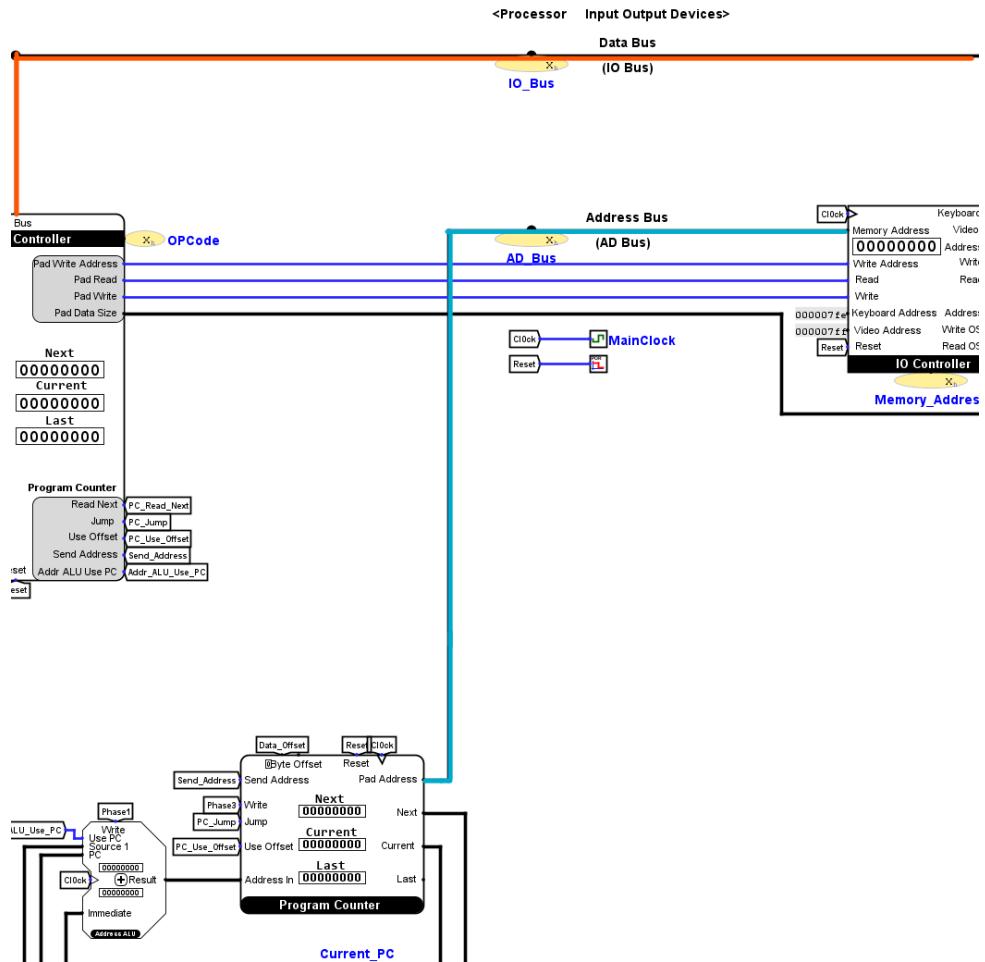


Figura 26 – Separação dos barramentos de dados e endereços

4.7.1.3 Terceira Fase: *SystemVerilog*

A terceira fase de desenvolvimento focou na conversão da arquitetura criada para uma especificação em uma linguagem de descrição de hardware (4.6.1.18), especificamente o SystemVerilog (SV). O processo de conversão consistiu na tradução direta, componente por componente, da arquitetura. O resultado final possibilitou a criação de testes mais rápidos e robustos para a arquitetura, além de facilitar futuras alterações para uma interação mais rápida com o processador.

O primeiro componente a ser codificado foi a ULA, seguida pela ULA de Endereços, Contador de Programa, Arquivo de Registradores, Data IO, Controlador, o processador com todos os componentes anteriores e, finalmente, a memória RAM. Todo o código da primeira versão pode ser visto no Apêndice A.

Cada componente possuía um arquivo de teste individual, mas esses arquivos deixaram de ser utilizados após a criação do arquivo de teste principal, que integra o componente do processador à memória RAM e inclui código para imprimir o estado atual do sistema.

Para executar um programa via simulação no SV, os programas montados pelo *assembler* são carregados na memória RAM através da rotina \$readmemh do SV. Essa rotina recebe o nome de um arquivo, que é lido e copiado para os registradores do componente. A formatação do arquivo é similar à dos arquivos criados pelo *assembler*, mas é necessário remover os endereços e cabeçalhos.

Para permitir que o usuário troque o programa, o nome do arquivo pode ser passado como parâmetro durante a instanciação do componente no arquivo principal de testes. O arquivo principal também inclui parâmetros para controlar a largura do endereço de memória, o tamanho do programa em bytes, o tempo de oscilação do *Clock*, o tempo limite de simulação e uma variável para selecionar uma das duas formas de impressão do estado do sistema.

A Figura 27 mostra o início da execução da simulação, utilizando a exibição simplificada do estado do processador. Nesta exibição, os valores são exibidos ao final de cada fase da *Pipeline* (a cada 3 ciclos de *Clock*), e informações sobre a instrução atual são apresentadas. Um processo simples de decompilação é executado para exibir os nomes legíveis da instrução e função atuais.

test_drisc.sv:24: warning: Port 4 (address_bus) of drisc expects 32 bits, got 12.						
test_drisc.sv:24: : Padding 20 high bits of the port.						
Time	Instruction	PC	addr	addr	addr	Instruction
30	00000013	0	[zero]:00000000	[zero]:00000000	[zero]	0 OP_IMM ADD
60	00000013	0	[zero]:00000000	[zero]:00000000	[zero]	0 OP_IMM ADD
90	0e802283	1	[zero]:00000000	[s0]:00000000	[t0]	232 LOAD WORD
120	0ec02303	2	[zero]:00000000	[a2]:00000000	[t1]	236 LOAD WORD
150	000f0137	3	[t5]:00000000	[zero]:00000000	[sp]	983040 LUI -----
180	00028513	4	[t0]:00000100	[zero]:00000000	[a0]	0 OP_IMM ADD
210	00000593	5	[zero]:00000000	[zero]:00000000	[a1]	0 OP_IMM ADD
240	40530633	6	[t1]:00000110	[t0]:00000100	[a2]	1029 OP SUB
270	ffff6013	7	[a2]:00000010	[t6]:00000000	[a2]	-1 OP_IMM ADD
300	0080000f	8	[zero]:00000000	[s0]:00000000	[ra]	8 JAL -----
330	00000013	8	[zero]:00000000	[zero]:00000000	[zero]	0 OP_IMM ADD
360	04c5da63	10	[a1]:00000000	[a2]:0000000f	[s4]	84 BRANCH GREATER_EQUAL
390	ff010113	11	[sp]:000f0000	[a6]:00000000	[sp]	-16 OP_IMM ADD
420	00112623	12	[sp]:000efff0	[ra]:00000024	[a2]	12 STORE WORD
450	01312423	13	[sp]:000efff0	[s3]:00000000	[s0]	8 STORE WORD
480	01212223	14	[sp]:000efff0	[s2]:00000000	[tp]	4 STORE WORD
510	00912023	15	[sp]:000efff0	[s1]:00000000	[zero]	0 STORE WORD
540	00058493	16	[a1]:00000000	[zero]:00000000	[s1]	0 OP_IMM ADD

Figura 27 – Início da simulação do código SystemVerilog, com exibição simplificada do estado do sistema

Embora essa simulação não seja interativa neste momento, ela ofereceu uma maneira alternativa de identificar erros no sistema. Ao intercalar a análise entre as simulações do Logisim Evolution e do Icarus Verilog, foi possível solucionar problemas que surgiam ao executar o algoritmo de ordenação *quicksort*.

O fim da simulação pode ocorrer quando o tempo limite é alcançado, quando um salto para o próprio endereço é identificado ou quando uma instrução ilegal é detectada. Após isso, o conteúdo do Arquivo de Registradores e da Memória RAM é impresso, com destaque para os espaços de memória que sofreram alterações, como mostra a Figura 28.

Durante a criação do sistema, foi observado que a maioria das detecções de *Clock*

```

43020 | 0000006f | 9 | [zero]:00000000 [zero]:00000000 [zero] |          0 |      JAL -----
Infinite loop detected, exiting simulation

Register values:
R[0] ( zero ) = 00000000
R[1] ( ra ) = 00000024
R[2] ( sp ) = 000f0000
R[3] ( gp ) = 00000000
R[4] ( tp ) = 00000000
R[5] ( t0 ) = 00000011
R[6] ( t1 ) = 00000009
R[7] ( t2 ) = 0000000a
R[8] ( s0 ) = 00000000
R[9] ( s1 ) = 00000000
R[10] ( a0 ) = 00000010
R[11] ( a1 ) = 00000010
R[12] ( a2 ) = 0000000f
R[13] ( a3 ) = 00000009
R[14] ( a4 ) = 00000000
R[15] ( a5 ) = 00000000
R[16] ( a6 ) = 00000000
R[17] ( a7 ) = 00000000
R[18] ( s2 ) = 00000000
R[19] ( s3 ) = 00000000
R[20] ( s4 ) = 00000000
R[21] ( s5 ) = 00000000
R[22] ( s6 ) = 00000000
R[23] ( s7 ) = 00000000
R[24] ( s8 ) = 00000000
R[25] ( s9 ) = 00000009
R[26] ( s10) = 00000000
R[27] ( s11) = 00000000
R[28] ( t3 ) = 00000109
R[29] ( t4 ) = 0000010a
R[30] ( t5 ) = 00000022
R[31] ( t6 ) = 00000011

RAM values:
Address 00000000: 13 00 00 00.83 22 80 0e.03 23 c0 0e.37 01 0f 00
Address 00000010: 13 85 02 00.93 05 00 00.33 06 53 40.13 06 f6 ff
Address 00000020: ef 00 80 00.6f 00 00 00.63 da c5 04.13 01 01 ff
Address 00000030: 23 26 11 00.23 24 31 01.23 22 21 01.23 20 91 00
Address 00000040: 93 84 05 00.13 09 06 00.ef 00 80 03.93 89 06 00
Address 00000050: 93 85 04 00.13 86 f9 ff.ef f0 1f fd.93 85 19 00
Address 00000060: 13 06 09 00.ef f0 5f fc.83 24 01 00.03 29 41 00
Address 00000070: 83 29 81 00.83 20 c1 00.13 01 01 01.67 80 00 00
Address 00000080: b3 02 c5 00.83 82 02 00.13 83 f5 ff.93 83 05 00
Address 00000090: 93 0c f6 ff.63 c8 7c 02.b3 0e 75 00.83 8f 0e 00
Address 000000a0: 13 00 00 00.63 dc 5f 00.13 03 13 00.33 0e 65 00
Address 000000b0: 03 0f 0e 00.23 80 ee 01.93 83 13 00
Address 000000c0: 6f f0 5f fd.13 03 13 00.33 0e 65 00.03 0f 0e 00
Address 000000d0: b3 0e c5 00.83 8f 0e 00.23 80 ee 01.23 00 fe 01
Address 000000e0: 93 06 03 00.67 80 00 00.00 01 00 00.10 01 00 00
Address 000000f0: 00 00 00 00.00 00 00 00.00 00 00 00.00 00 00 00 00 00
Address 00000100: 88 99 aa bb.cc dd ee ff.00 11 22 33.44 55 66 77
Address 00000110: 00 00 00 00.00 00 00 00.00 00 00 00.00 00 00 00 00 00
Address 00000120: 00 00 00 00.00 00 00 00.00 00 00 00.00 00 00 00 00 00
Address 00000130: 00 00 00 00.00 00 00 00.00 00 00 00.00 00 00 00 00 00
Address 00000140: 00 00 00 00.00 00 00 00.00 00 00 00.00 00 00 00 00 00
    ...

```

Figura 28 – Fim da simulação do código SystemVerilog

utilizadas para atualizar os registradores ocorria durante a borda de descida. A ideia inicial era que faria mais sentido enviar os dados durante a borda de subida do *Clock* e escrevê-los durante a borda de descida. No entanto, após alguns testes, decidiu-se manter todas as ativações na borda de subida (4.6.1.5), pois isso não alterava significativamente o funcionamento do sistema. Essa alteração de temporização também foi aplicada ao modelo de simulação no Logisim.

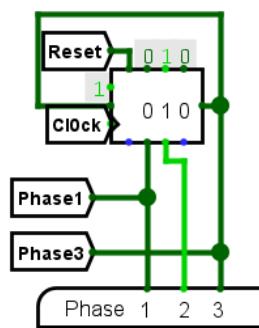


Figura 29 – Contador em Anel, implementado com um Registrador de Deslocamento de 3 bits

Enquanto os sinais de ativação estavam sendo definidos, a entrada de fases do processador foi alterada para um gerador de fases interno. Esse gerador funciona com um contador em anel, que é atualizado durante a borda positiva do *Clock*, conforme mostra a Figura 29.

Finalmente, durante essa fase, o sinal de *overflow* calculado pela ULA foi alterado tanto no DRISC quanto no circuito do RISC I. Esse sinal apresentava alguns problemas anteriormente, embora não afetasse o funcionamento de alguns programas de teste.

4.7.1.4 Quarta Fase: Entrada e Saída

Durante a replicação do RISC I, alguns experimentos foram realizados para adicionar componentes que permitissem a interação do usuário. No entanto, esses experimentos se limitaram a testes simples, que não acrescentaram muito ao trabalho realizado. Nesta fase, um dos principais focos foi definir uma forma para permitir que os estudantes pudessem interagir com o sistema criado (4.6.1.9).

No Logisim, alguns componentes foram escolhidos para permitir essa interação (Figura 30), destacando-se o teclado e a tela. O teclado permite que o usuário digite caracteres, que são então transmitidos para sua saída com seus respectivos valores ASCII. A tela é um retângulo que, a partir de sinais de posição, cor e escrita, modifica seus pixels, permitindo a renderização de gráficos computados pelo processador.

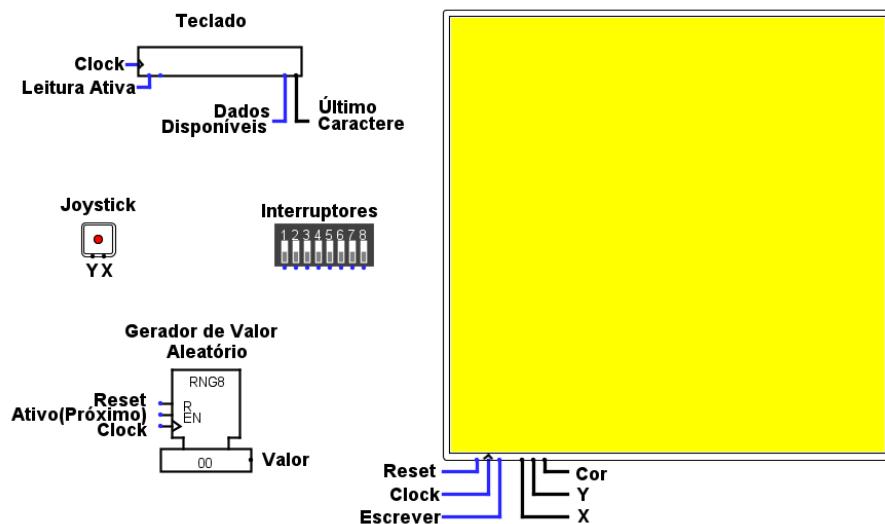


Figura 30 – Dispositivos de entrada e saída do Logisim utilizados

Para conectar esses componentes ao processador, foi criado um componente que transmite os sinais de escrita e leitura caso o endereço atual esteja dentro de uma faixa de valores. Esse componente (Figura 31) funciona como um controlador rudimentar de dispositivos de entrada e saída, ajudando a evitar conflitos no barramento de dados.

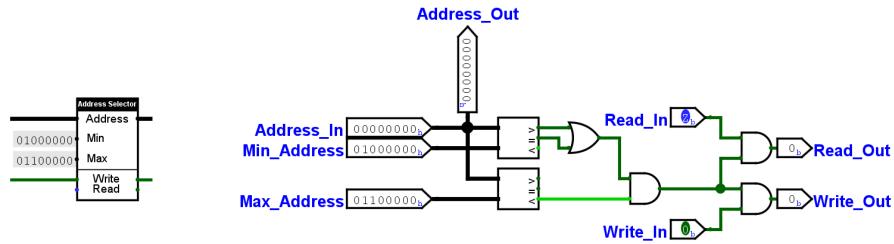


Figura 31 – Seletor de Endereços

Com este novo componente, foram criadas alocações fixas para cada dispositivo de entrada e saída, conforme mostrado na Figura 32. Considerando o último valor como exclusivo, a memória RAM ficou alocada entre os endereços 0x00000000 e 0x00fffffc; um gerador de número aleatório, 8 interruptores, um *joystick* e o teclado ficaram com os endereços 0x00fffffc a 0x01000000, e a tela com os endereços 0x01000000 a 0x01100000.

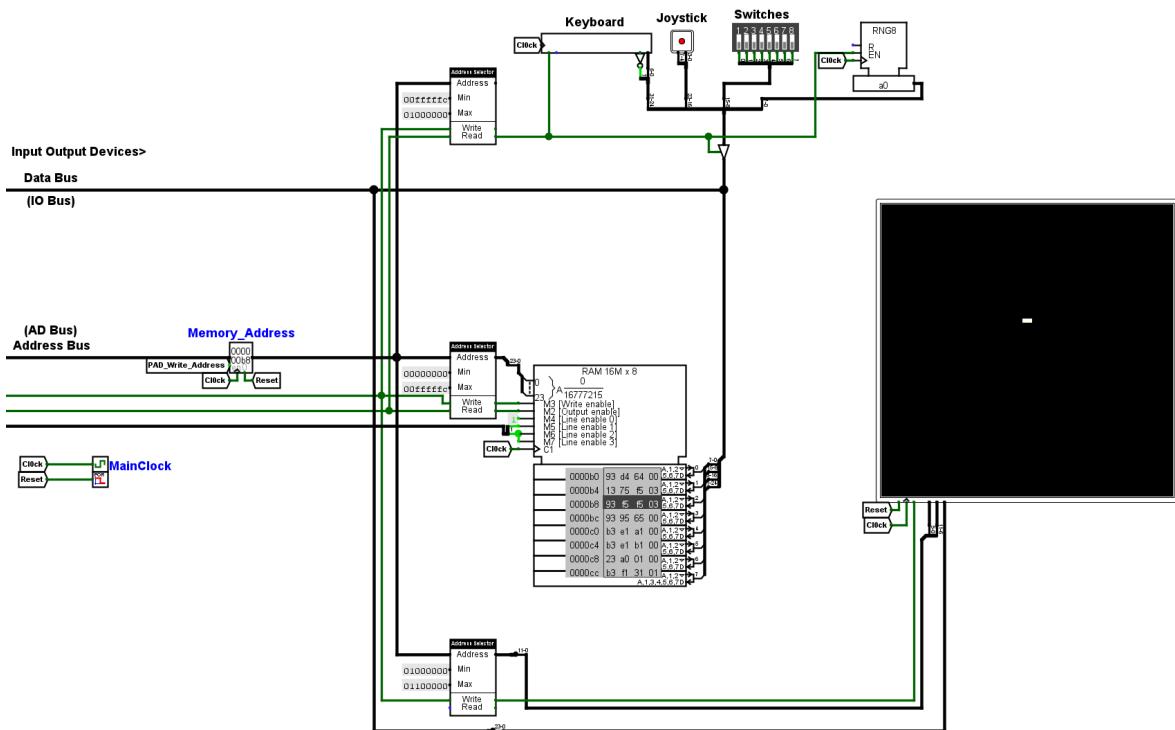


Figura 32 – Enter Caption

Como cada dispositivo de entrada emite apenas um byte de informação, os 4 dispositivos de entrada foram colocados em uma palavra, o que permite o carregamento dos valores de todos os dispositivos simultaneamente ou o carregamento de um dos valores utilizando a instrução `lbu`(load byte unsigned). Por exemplo, a partir do endereço 0x01000000 armazenado no registrador `gp`, podemos obter o valor do teclado com a instrução "`lbu rd gp -1`"(carregar byte de `[gp - 1]` para o registrador `rd`), e o valor do *joystick* com a instrução "`lbu rd gp -2`"(carregar byte de `[gp - 2]` para o registrador `rd`).

Já a tela foi configurada de modo que cada pixel possuísse seu próprio endereço, com os 6 bits menos significativos representando a posição horizontal, enquanto os 6 bits subsequentes representam a posição vertical. Esse tamanho pode ser alterado para aumentar ou diminuir o tamanho da tela, porém, 6 bits por coordenada foram definidos como padrão por fins práticos.

Ao escrever em qualquer um dos endereços da tela, o pixel correspondente troca para a cor que foi enviada, que deve estar no formato RGB 888 (Em hexadecimal 0x00BBGGRR). Outros formatos podem ser escolhidos, porém estes fornecem menos flexibilidade e não possuem alinhamento aos *bytes* por canal de cor, o que pode dificultar o seu controle.

Para testar a entrada e saída, um programa simples para o DRISC foi desenvolvido (Apêndice B), onde um ponto em movimento é desenhado na tela e, utilizando as teclas W, A, S e D, o usuário pode modificar a direção desse ponto.

Após a definição das interfaces de entrada e saída, foi notado que os registradores presentes no Deslocador, na ULA e na ULA de endereços não eram necessários, isso porque os barramentos A e B contêm a informação necessária durante todo o ciclo de execução e escrita do arquivo de registradores.

Essa remoção dos registradores também tirou a necessidade de um sistema de encaminhamento de dados (4.6.1.14), pois assim que o arquivo de registradores recebe os dados da memória, os dados atualizados não precisam ser repassados para os outros componentes.

Sem os registradores, a ULA de endereços e o Deslocador se tornaram muito simples para continuarem como componentes próprios, então o Deslocador foi fundido à ULA e a ULA de endereços foi fundida ao Contador de Programa. Os resultados dessas alterações podem ser vistos na Figura 33.

Após essas alterações, as interfaces de entrada e saída foram implementadas no código SV, porém com alguns desafios. O maior deles foi que simuladores como o Icarus Verilog não oferecem uma forma fácil de interação em tempo real com a simulação. Para contornar isso, algumas opções foram analisadas, como a compilação do código utilizando o Verilator ou o recurso DPI (Interface de Programação Direta) do SystemVerilog.

O Verilator (SNYDER, 2003) é um programa que transforma código Verilog ou SystemVerilog em código C++ ou SystemC para simulação. Ele verifica o código, faz checagens de erros e pode adicionar pontos de análise. O resultado é um conjunto de arquivos que podem ser compilados para criar um modelo simulável. Esse modelo permite testar e simular o design do hardware, integrando-o com código C customizado.

Apesar de sua proposta promissora, o Verilator possui uma grande limitação: a falta de suporte para sistemas operacionais Windows. Soluções para isso envolveriam compilar o Verilator manualmente ou utilizar máquinas virtuais. No entanto, como o objetivo é

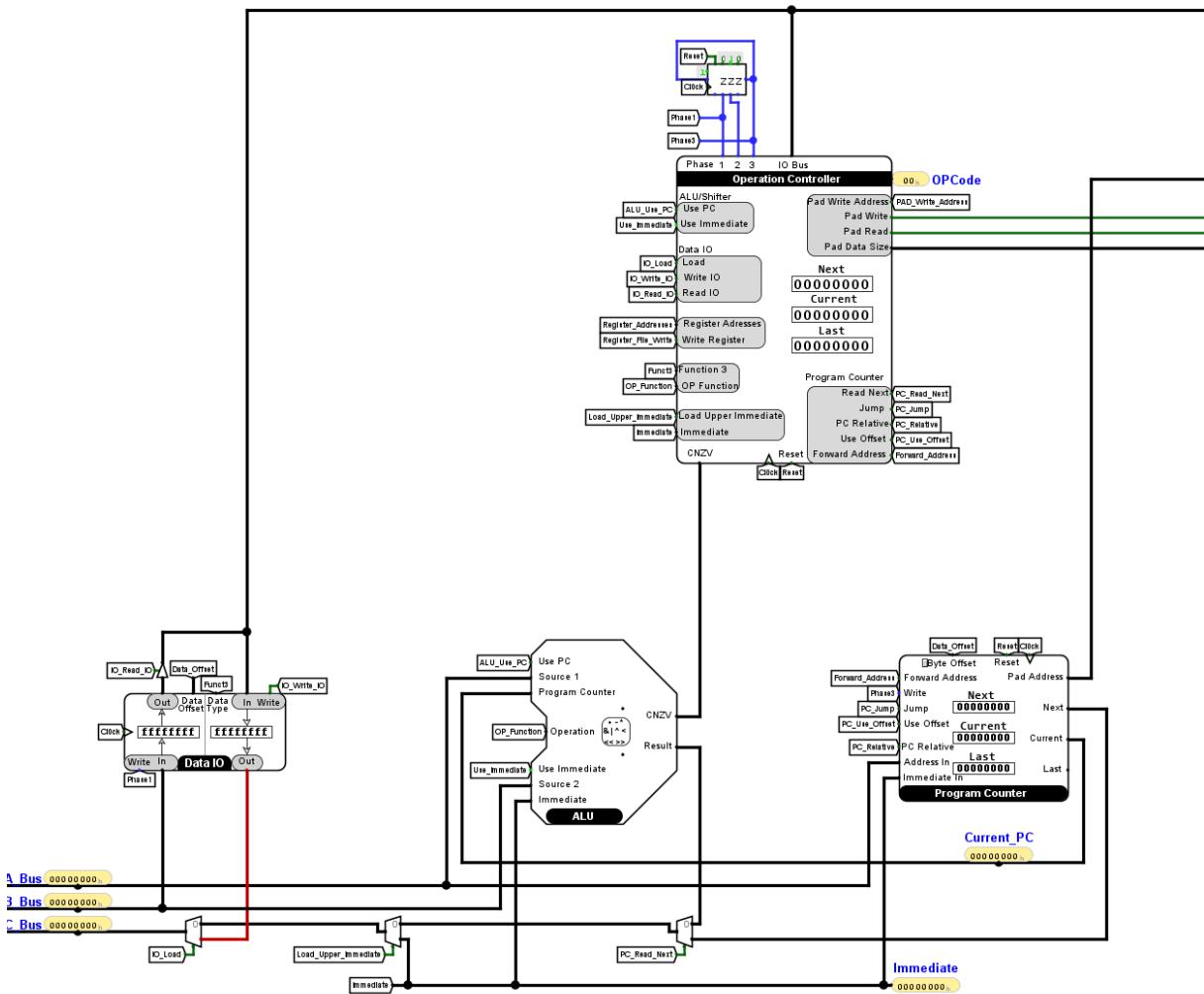


Figura 33 – Resultado da remoção de registradores de entrada, do Deslocador e da ULA de endereços

obter um ambiente didático fácil de ser utilizado, essa solução foi descartada por ser muito complexa.

O recurso DPI, por outro lado, oferece uma forma de comunicação direta com códigos escritos em C. Esses códigos devem ser compilados como bibliotecas que são anexadas à simulação. Funções criadas no código SV e no código C são exportadas de um lado e importadas pelo outro. Isso permitiria, por exemplo, que uma função no lado do SV criasse um aplicativo com uma interface gráfica no lado do C. Esse tipo de integração poderia ser facilmente utilizado, mas o Icarus Verilog não possui suporte para o recurso DPI.

Uma última solução foi encontrada, utilizando funções do SystemVerilog para ler e modificar arquivos. Esses arquivos podem então ser utilizados como uma interface entre a simulação e outros programas. Apesar dessa solução possuir vários pontos negativos, ela

ainda é simples o suficiente para não precisar de uma configuração complexa.

O primeiro componente a ser codificado foi a tela (Disponível no Apêndice A). O funcionamento base desse componente é similar a uma memória RAM sem função de leitura, ou seja, o componente possui apenas entradas que permitem a escrita no seu conjunto de registradores. Esse conjunto é então periodicamente copiado para o arquivo externo, caso modificações tenham sido feitas na memória.

Após a criação do componente de tela, desenvolveu-se um novo programa em C#, responsável pela leitura dos dados da tela, como ilustrado na Figura 34. Cada número presente no arquivo de tela é interpretado como um pixel pelo programa, gerando o mesmo resultado visual observado na simulação com o Logisim.

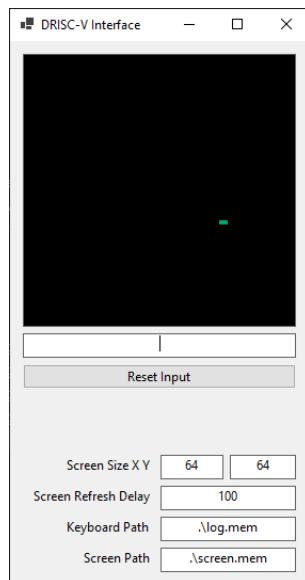


Figura 34 – Programa para interação com a simulação SV, denominado DRISC-V Interface

Foi incorporada também uma entrada para teclado, onde cada caractere digitado é convertido em um número hexadecimal e adicionado ao final do arquivo de entrada. Um componente de entrada na simulação acessa esse arquivo a cada sinal de leitura. Outros tipos de entrada ainda não foram implementados, com exceção do gerador de números aleatórios, o qual foi codificado diretamente no componente de entrada.

Na primeira versão do *DRISC-V Interface*, era possível configurar parâmetros como o tamanho da tela, a frequência de leitura do arquivo de tela e os nomes dos arquivos de entrada e saída.

Já nas etapas finais do desenvolvimento, adicionou-se um novo dispositivo de saída: o Terminal. Esse componente foi projetado para a impressão rápida de texto, evitando a necessidade de renderizar individualmente cada caractere durante a exibição de informações.

No Logisim, o terminal possui uma entrada de 7 bits, permitindo a impressão de

qualquer caractere da tabela ASCII, além de um sinal adicional para limpeza do conteúdo. Os endereços designados para o terminal foram 0x01100000 para escrita e 0x01100001 para limpeza.

No código da simulação SV, a implementação do terminal seguiu um conceito semelhante ao da tela. Contudo, em vez de um valor de pixel por linha, o arquivo final contém uma lista de bytes correspondentes ao buffer de dados do terminal. Esses dados são interpretados pela interface, que foi atualizada conforme mostrado na Figura 35.

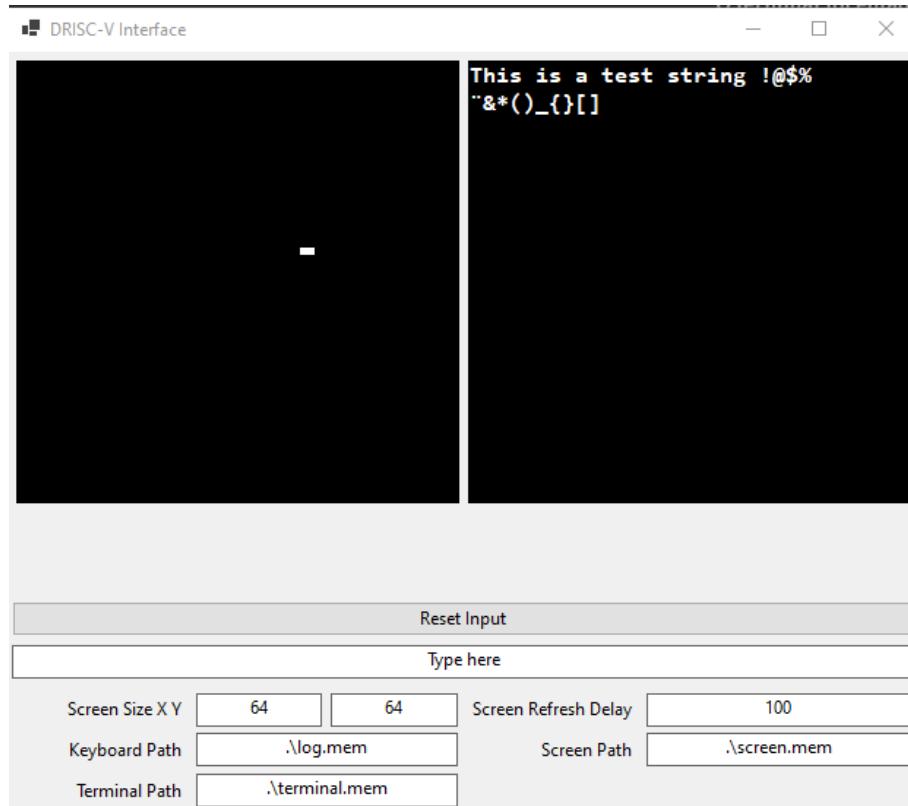


Figura 35 – Interface atualizada para interação com os dados do terminal

4.7.1.5 Quinta Fase: Controlador de Registradores de Controle e Status

Com a base dos dispositivos de E/S prontos, foi decidida a adição de um novo sistema de interrupções(4.6.1.7). O sistema antigo do RISC I havia sido removido por não possuir mais funções válidas na nova versão do processador; porém, o RISC-V possui uma especificação sobre exceções e interrupções que puderam ser implementadas, presente no volume II do manual de especificação RISC-V (2024a).

Para atender à especificação, a expansão *Zicsr* de registradores de controle e estado(CSRs) teve de ser implementada. Essa expansão adiciona registradores fundamentais para o tratamento de interrupções, além de ser requerida pela arquitetura privilegiada do RISC-V (2024a), podendo ser aproveitada para a implementação dos modos de Máquina(Kernel) e Usuário (4.6.1.6).

Durante a implementação da expansão Zicsr foram implementadas atualizações no funcionamento da pipeline. A primeira modificação envolveu a redefinição do momento de atualização dos registradores referentes à instrução atual e à última instrução. Originalmente, essa atualização era realizada na borda de descida do *clock*, o que gerava uma assincronia entre a fase e os registradores. Esse descompasso dificultava a análise e compreensão dos eventos, além de aumentar a propensão a erros de sincronismo durante o desenvolvimento.

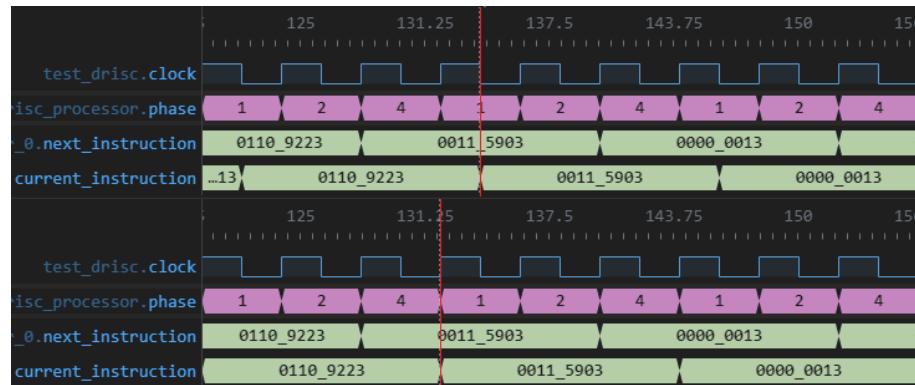


Figura 36 – Alteração no momento de atualização do registrador de instrução atual (*current_instruction*)

Para corrigir esse problema, o momento de atualização dos registradores foi alterado: em vez de ocorrer na borda de descida do *clock* durante a fase 1, passou a ser realizado na borda de subida durante a fase 3. O resultado dessa modificação pode ser observado na Figura 36. É importante destacar que, como a atualização dos registradores depende da fase, e a fase é influenciada pelo *clock*, a fase utilizada deve preceder a subida do *clock*. Por esse motivo, a fase escolhida é a 3, e não a 1.

A segunda alteração, porém, foi de maior impacto: a remoção de uma das fases da pipeline. Essa remoção não só simplificou a execução das instruções, mas também envolveu a remoção de registradores e a simplificação de circuitos de controle, além de um aumento efetivo de cerca de 33% no tempo de processamento de instruções.

A ideia sobre essa alteração foi que, após todas as mudanças feitas até esse momento na arquitetura, diversos espaços ficaram livres entre as fases da pipeline, tornando a execução desnecessariamente mais demorada. Com isso em mente, foram analisados os possíveis conflitos entre os sinais durante cada parte da pipeline, e todos os sinais foram movidos para a primeira fase possível, resultando na *Pipeline* vista na Tabela 6.

Nessa nova pipeline, a instrução de **store** é executada durante 2 passos, contrário aos 3 passos que utilizava anteriormente; porém, a instrução de (load) foi mantida durante o terceiro passo como uma forma de lidar com atrasos de leitura de informação da memória, mesmo que este caso não aconteça durante a simulação.

O aumento de velocidade se deve ao fato de que uma nova instrução é processada

Passo Da Pipeline	Instrução/Passo	Fase 1	Fase 2
1	Coleta de Instrução e Decodificação	RAM → Próxima Instrução	Próxima Instrução → Instrução Atual Endereço da Próxima Instrução → Barramento de Endereços
2	Todas Instruções		Resultado → Arquivo de Registradores
	Load	Endereço do dado → Barramento de Endereços	Dispositivo de E/S → DataIO
	Store		DataIO → Dispositivo de E/S
	Jump		Endereço do Pulo → Próximo PC
3	Branch		Se verdade: Endereço do Pulo → Próximo PC
3	Load	Resultado → Arquivo de Registradores	

Tabela 6 – Nova *Pipeline* de 3 passos e 2 fases

a cada 2 ciclos de *clock*, ao invés dos 3 ciclos anteriores, e foi verificado a partir da cronometragem da execução do programa de entrada e saída, onde o tempo de escrita de uma linha na tela foi de 4 segundos para em torno de 2,67 segundos.

Com essas alterações sobre a pipeline, o Controlador de CSRs pôde ser construído. Diferentemente dos outros componentes, este foi inicialmente criado através do código SV. Isso foi feito pois este componente possui um certo nível de complexidade no seu comportamento que é melhor expresso por código, além de que os testes sobre o código são mais simples de serem executados.

O primeiro passo para a criação desse controlador foi a escolha de quais CSRs seriam utilizados. Para isso, a especificação privada do [RISC-V \(2024a\)](#) foi consultada. Alguns dos CSRs são obrigatórios para a implementação, como o `mvendorid`, que contém um identificador JEDEC do fabricante do processador; porém, no nosso caso, podemos apenas colocar valores de exemplo nesses registradores.

A Tabela 7 mostra todos os CSRs que foram implementados na arquitetura:

Nome	Resumo	Notas
M VENDOR ID	Identifica o fabricante	Valor fixo 0x00af001d
M ARCH ID	Identifica a arquitetura	Valor fixo 0x00000000
M IMP ID	Identifica a implementação	Valor fixo para cada atualização da arquitetura.
M HART ID	ID do núcleo de hardware	Valor fixo 0x00000000 (núcleo único)
M STATUS	Contém alguns bits relacionados ao estado do processador	Bits que foram implementados: Privilégio Anterior (PP), Interrupção Ativa (IE) e Interrupção Ativa Anterior (PIE)
M ISA	Indica as expansões disponíveis e ativas do processador	Não foi implementada uma forma de desativar expansões. As expansões implementadas foram E, I, M (sem divisão) e U
M I E	Bits que permitem a ativação de interrupções	Apenas foram implementadas interrupções de <i>software</i> , <i>timer</i> e externas a nível de máquina.
M T VEC	Registrador que contém o endereço com tratamento de interrupções e exceções	O valor padrão é 0x80000001, o que significa que o endereço base é 0x80000000 e cada tipo de interrupção é levado para um endereço diferente

M STATUS H	Bits superiores do M STATUS	O único bit que contém informação relevante é o de endianismo do modo usuário, que é fixo em 0 nessa implementação.
M SCRATCH	Registro temporário	Usado para armazenamento temporário de informações durante o tratamento de <i>traps</i> .
M E PC	Contador de programa de exceção	Indica o último endereço de memória acessado antes de uma <i>trap</i> .
M CAUSE	Causa da <i>trap</i>	Além das interrupções, a causa também pode ser relacionada a algumas exceções, como as de endereços desalinhados ou instruções ilegais.
M T VAL	Valor da <i>trap</i>	Contém o valor que causou a <i>trap</i> , podendo ser uma instrução ou um endereço calculado.
M I P	Interrupção pendente	O seu valor é controlado por sinais de dispositivos externos ao processador, e seus bits são equivalentes aos bits do MIE
CYCLE	Contador de ciclos	Um contador que atualiza a cada ciclo de <i>clock</i> . Cada fase da <i>Pipeline</i> dura dois ciclos de <i>clock</i> .
TIME	Contador de tempo	Ao ser lido, ativa uma instrução de carregamento de um registrador endereçado na memória.
INST RET	Contador de instruções executadas	Incrementa por instrução iniciada a cada dois ciclos de <i>clock</i> . Não incrementa durante exceções.
CYCLE H	Bits superiores do CYCLE	—
TIME H	Bits superiores do TIME	—
INSTRET H	Bits superiores do INSTRET	—

Tabela 7 – Registradores de Controle e Estado implementados na arquitetura.

Após a escolha e o estudo detalhado das especificações dos registradores, sua implementação foi realizada de forma individual, atendendo aos requisitos técnicos necessários.

Inicialmente, o foco esteve na implementação das instruções responsáveis pela manipulação dos registradores. Essas instruções realizam a leitura de um CSR-alvo enquanto simultaneamente modificam seu conteúdo por meio de uma das três operações disponíveis: escrita, definição ou limpeza. A lógica geral para a implementação dessas instruções está descrita nos Algoritmos 1 e 2.

O Algoritmo 1 ilustra como as instruções são implementadas na rotina síncrona do CSR, que é executada em cada borda de subida do *clock*. Este processo garante a operação correta dos registradores e suas interações com as instruções associadas. Tentativas de escrita para CRSSs que só permitem leitura são ignoradas; porém, tentativas de leitura ou escrita de CSRs que exigem um nível de operação superior ao que está sendo utilizado emitem um sinal de CSR ilegal e não são executadas.

Algoritmo 1 Rotina Síncrona do Controlador de CSR

```

1: se é reset então
2:   Inicializar os registradores com os valores iniciais
3: senão se é fase 2 então
4:   se é Interrupção OU Exceção então
5:     Executar rotina de trap
6:   senão se é instrução MRET então
7:     Executar rotina do MRET
8:   senão se é instrução de manipulação de CSR E CSR é válido então
9:     CSR selecionado ← NovoValorCalculado()
10:  fim se
11: se NÃO é Exceção então
12:   Incrementar contador INSTRET: INSTRET ← INSTRET + 1
13: fim se
14: senão se é fase 1 E é instrução de manipulação de CSR E CSR é válido então
15:   Barramento C ← valor do registrador selecionado
16: fim se
17: se NÃO é reset então
18:   Incrementar contador CYCLE: CYCLE ← CYCLE + 1
19: fim se

```

Já o Algoritmo 2 apresenta a lógica utilizada para calcular o novo valor que será gravado nos CSRs durante a execução das instruções de manipulação.

Algoritmo 2 Função NovoValorCalculado()

Entrada: Registrador, Dado

Saída: Novo valor do CSR

```

1: se modo de operação é escrita então
2:   retorna Dado
3: senão se modo de operação é definição então
4:   retorna Registrador OU Dado
5: senão se modo de operação é limpeza então
6:   retorna Registrador E NÃO Dado
7: senão
8:   retorna nenhum valor
9: fim se

```

Com a manipulação dos CSRs devidamente implementada, o próximo passo foi a integração das funcionalidades de Exceções e Interrupções. Para isso, foram criados os seguintes sinais:

- **Interrupção:** Indica a ocorrência de uma interrupção.
- **Exceção:** Indica a ocorrência de uma exceção.
- **Causa (da trap):** Contém a causa da interrupção ou exceção.

- **Valor (da trap):** Armazena dados adicionais utilizados no tratamento de determinadas exceções.

Esses sinais são gerenciados de forma assíncrona por meio do decodificador, cuja lógica está detalhada no Algoritmo 3.

Algoritmo 3 Rotina Assíncrona do Controlador de CSR

```

1: Inicialização dos Sinais Assíncronos:
2:   Interrupção ← falso
3:   Exceção ← falso
4:   Causa ← Nenhuma
5:   Valor ← 0
6:
7: se interrupções estão ativas OU é Modo Usuário então
8:   se interrupção  $X$  está ativa E pendente então
9:     Interrupção ← verdadeiro
10:    Causa ← Interrupção  $X$ 
11:   fim se
12: fim se
13:
14: se Interrupção = verdadeiro então
15:   se MTVEC está no modo veteado então
16:     Endereço Alvo ← MTVEC + (Causa × 4)
17:   senão
18:     Endereço Alvo ← MTVEC
19:   fim se
20: senão se Alguma condição para exceção é verdadeiro então
21:   Exceção ← verdadeiro
22:   Causa ← Código da exceção
23:   Valor ← Instrução Atual, Endereço Calculado ou 0
24: senão se é uma instrução MRET então
25:   Endereço Alvo ← MEPC
26: fim se
27:
28: se Exceção = verdadeiro então
29:   Endereço Alvo ← MTVEC
30: fim se

```

É importante destacar que há uma ordem de prioridade codificada conforme especificado por RISC-V (2024a). Essa ordem segue o seguinte esquema:

1. Interrupções:

- a) Externa (prioridade mais alta).
- b) Software.
- c) Timer.

2. Exceções:

- a) Instrução Ilegal.
- b) Endereço de Instrução Desalinhado.
- c) Breakpoint.
- d) ECALL a partir do Modo Máquina.
- e) ECALL a partir do Modo Usuário.
- f) Endereço de Carregamento Desalinhado.
- g) Endereço de Salvamento Desalinhado.

Vale observar que alguns eventos possuem a mesma prioridade ou não podem ocorrer simultaneamente.

Os sinais são então utilizados durante a rotina de execução das *traps*, bem como na gravação dos CSRs relacionados, conforme apresentado no Algoritmo 4. Esta rotina faz parte da execução síncrona detalhada anteriormente no Algoritmo 1.

Algoritmo 4 Rotina de Trap (Porção Síncrona)

- 1: Interrupção Ativa Anterior \leftarrow Interrupção Ativa
 - 2: Interrupção Ativa \leftarrow **falso**
 - 3: Privilégio Anterior \leftarrow Privilégio
 - 4: Privilégio \leftarrow **Modo Máquina (Kernel)**
 - 5: MEPC \leftarrow Contador de Programa Atual
 - 6: MCAUSE \leftarrow Causa (da *trap*)
 - 7: MVAL \leftarrow Valor (da *trap*)
-

Inicialmente, a rotina foca em desativar as interrupções para evitar que novas *traps* sejam acionadas enquanto as interrupções pendentes estão sendo tratadas. Além disso, o Modo Máquina é ativado, permitindo que o sistema em execução acesse recursos privilegiados, como CSRs específicos que são essenciais para o tratamento de exceções e interrupções.

Ao final da rotina, o contador de programa é salvo no CSR MEPC, garantindo que o retorno ao programa ocorra após o término do tratamento da exceção ou interrupção. Além disso, os CSRs responsáveis por armazenar a causa e o valor da *trap* são atualizados, proporcionando informações essenciais para o processamento dessas situações no nível de sistema.

Para retornar aos programas após uma rotina de tratamento, a instrução MRET deve ser executada. Essa instrução permite o retorno ao privilégio anterior e reativa as interrupções previamente habilitadas. Além disso, ela faz com que o contador de programa salte para o endereço que estava sendo executado antes da rotina de tratamento. A

porção síncrona da MRET está detalhada no Algoritmo 5, enquanto a parte assíncrona, apresentada no Algoritmo 3, é responsável por atribuir o endereço de retorno ao sinal Endereço Alvo.

Algoritmo 5 Rotina do MRET (*Machine Return*) (Porção Síncrona)

- 1: Interrupção Ativa \leftarrow Interrupção Ativa Anterior
 - 2: Interrupção Ativa Anterior \leftarrow **verdadeiro**
 - 3: Privilégio \leftarrow Privilégio Anterior
 - 4: Privilégio Anterior \leftarrow **Modo Usuário**
-

O sinal Endereço Alvo é um dos principais sinais da interface do controlador. Ele é transmitido ao Contador de Programa, onde pode ser utilizado como endereço de pulo ou endereço de carregamento. Para viabilizar o uso do Endereço Alvo no Contador de Programa, foram criados dois sinais adicionais: o Pulo de Sistema (*System_Jump*) e o Carregamento de Sistema (*System_Load*).

O Pulo de Sistema funciona de maneira semelhante ao sinal de pulo do Controlador de Operações, mas possui prioridade mais alta. Isso significa que, caso ambos os sinais sejam ativados simultaneamente, o Pulo de Sistema terá preferência. Por outro lado, o Carregamento de Sistema é um sinal que atua tanto no Controlador de Operações quanto no Contador de Programa. No Controlador de Operações, ele é inserido na decodificação da instrução atual, resultando na execução de uma instrução *load* no Endereço Alvo, que é selecionado no Contador de Programa quando o sinal de Carregamento de Sistema está ativo.

O Pulo de Sistema é ativado na fase 2 quando ocorre uma exceção, interrupção ou instrução MRET. Já o Carregamento de Sistema é ativado durante a leitura de CSRs mapeados na memória, como os registradores TIME e TIMEH nesta implementação.

O controlador de CSR foi testado continuamente ao longo de seu desenvolvimento, empregando diversas técnicas, incluindo testes em SystemVerilog e análise de sinais por meio de arquivos VCD. Esses arquivos registram o histórico de ativação de todos os sinais da simulação, permitindo uma verificação detalhada do comportamento do sistema. Após a conclusão da implementação funcional, o componente desenvolvido foi replicado na simulação do Logisim, conforme ilustrado na Figura 37.

O circuito do controlador foi projetado seguindo uma ordem lógica semelhante à utilizada no desenvolvimento do código. A primeira etapa envolveu a criação dos registradores e a implementação das funcionalidades de manipulação de dados.

O Circuito 1 processa a entrada da fase e da instrução atual, separando-a em seus respectivos parâmetros. Em paralelo, o Circuito 7 recebe a instrução atual já decodificada, originada diretamente do Controlador de Operações. Neste circuito, alguns sinais são transmitidos para módulos subsequentes, enquanto dois sinais específicos são gerados:

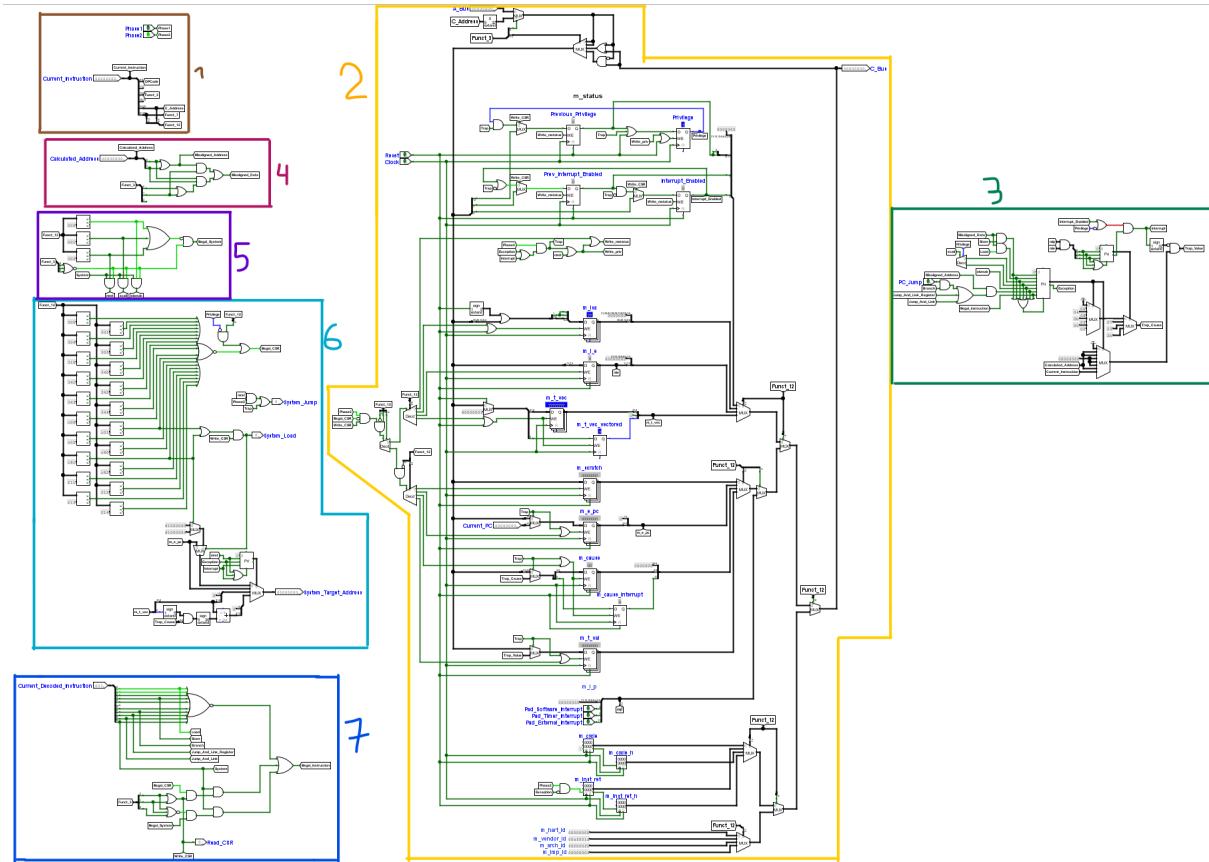


Figura 37 – Circuito do Controlador de CSR.

o sinal de manipulação de CSR e o sinal de instrução ilegal. Ambos os circuitos estão ilustrados na Figura 38.

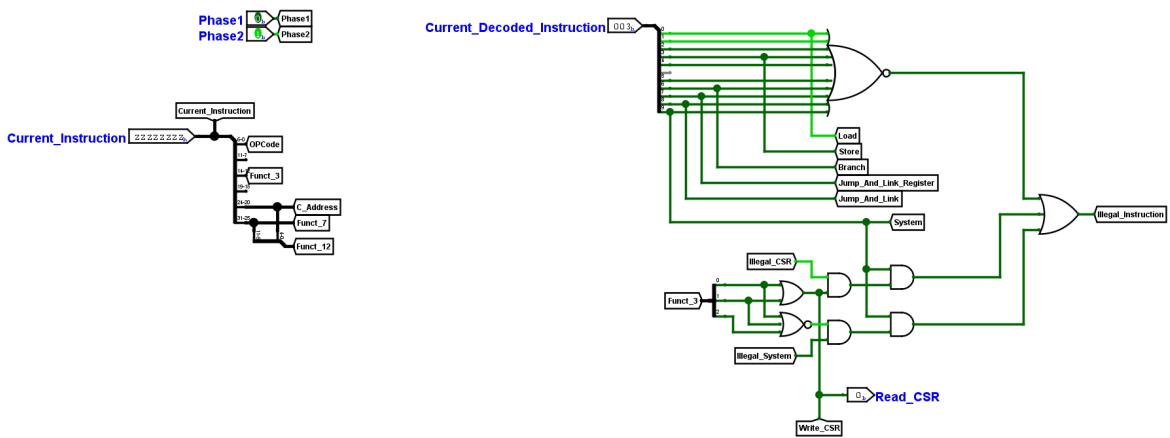


Figura 38 – Circuitos de fase, instrução atual e instrução decodificada.

O Circuito 2 é responsável pelo armazenamento e manipulação dos CSRs. Sua lógica de funcionamento simplificada é representada na Figura 39. Cada registrador possui sua escrita ativada por meio de um decodificador conectado a um sinal de controle. Para evitar a criação de um decodificador excessivamente grande, a implementação dividiu esse componente em módulos menores, utilizando apenas alguns dos 12 bits do endereço. Além

disso, certos registradores, como aqueles de leitura exclusiva, não são conectados a um decodificador.

As saídas dos registradores são encaminhadas para multiplexadores, que, de maneira análoga à lógica de escrita, utilizam apenas alguns bits para selecionar os registradores apropriados. A saída do último demultiplexador é direcionada tanto para o barramento C quanto para o circuito responsável pelo cálculo do novo valor do CSR.

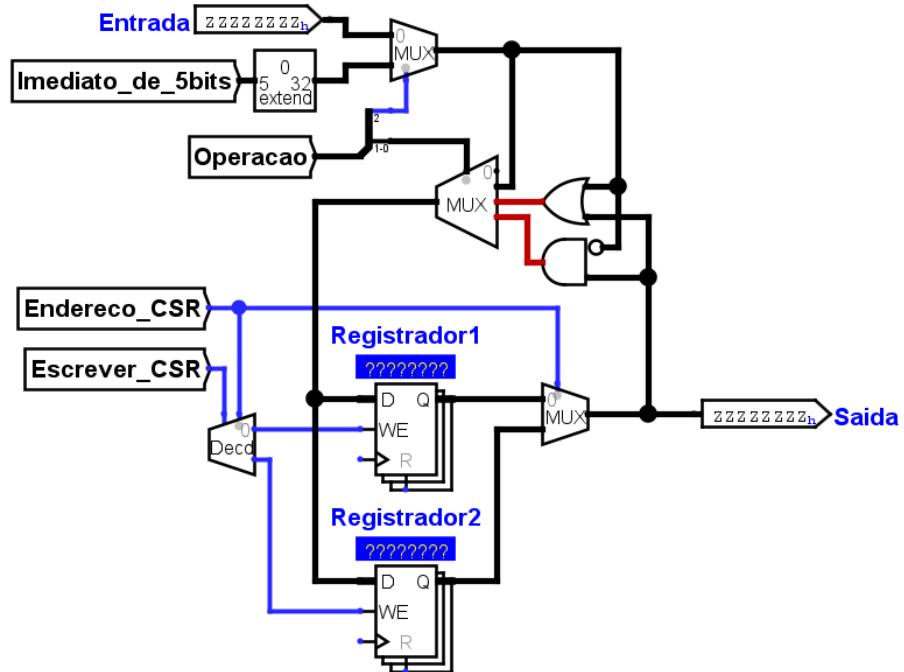


Figura 39 – Representação simplificada do circuito de armazenamento e manipulação de CSRs.

O cálculo do novo valor do CSR segue a lógica estabelecida no Algoritmo 2. O resultado é conectado a todos os registradores, sendo sua entrada proveniente do barramento A ou de um valor imediato de 5 bits (sem sinal), extraído do campo Rs2 da instrução.

Os Circuitos 3, 4, 5 (Figura 40) e parte do Circuito 6 são responsáveis pela lógica assíncrona do tratamento de *traps*. O Circuito 4 detecta endereços desalinhados gerados pelo contador de programas, identificando desalinhamentos para instruções de salto, carregamento e armazenamento. Já o Circuito 5 utiliza os campos `func_3` e `funct_12` (últimos 12 bits da instrução) para gerar sinais indicativos de instrução de sistema ilegal, além de instruções `EBREAK`, `ECALL` e `MRET`.

Uma parte significativa do Circuito 6 é dedicada à verificação da existência do endereço do CSR da instrução atual, emitindo um sinal de CSR ilegal caso o endereço não seja válido. Esse circuito também identifica tentativas de acesso a CSRs que requerem privilégios elevados, agregando essa informação ao sinal de CSR ilegal.

Por fim, o Circuito 3 é, em sua essência, uma implementação da lógica descrita no Algoritmo 3. Ele gera os sinais de exceção e interrupção, bem como os sinais de causa e

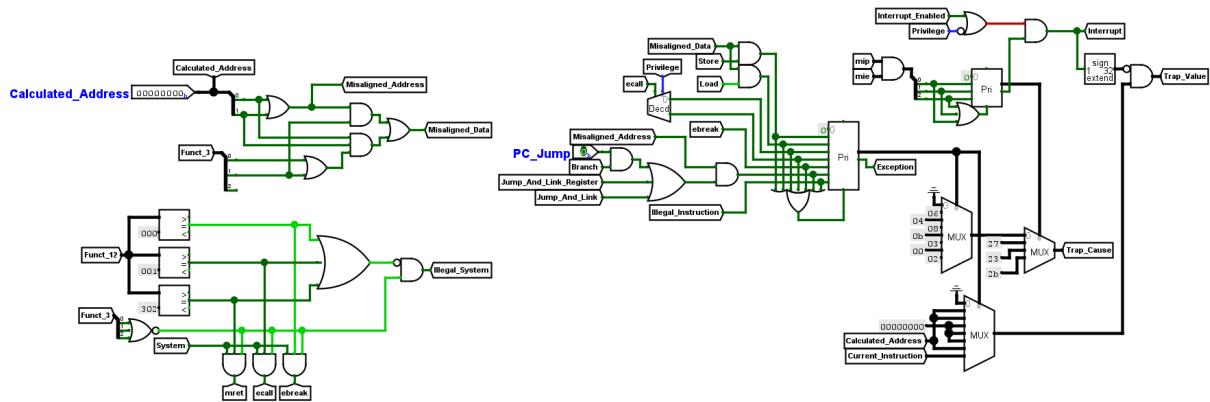


Figura 40 – Circuitos de tratamento de traps.

valor da *trap*. Este circuito incorpora um novo componente do Logisim, o codificador de prioridade, que recebe os vários sinais das possíveis exceções e interrupções e determina automaticamente o endereço do evento com maior prioridade.

A Figura 41 apresenta o Circuito 6. A porção superior está relacionada à detecção de CSRs ilegais, enquanto a parte inferior gerencia os sinais de pulo e carregamento de sistema, além de definir o endereço alvo enviado ao contador de programas.

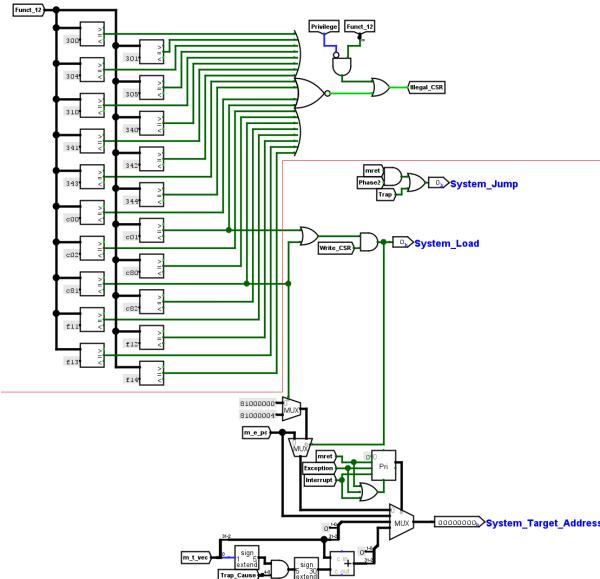


Figura 41 – Circuito de verificação de CSRs e controle de sinais de pulo e carregamento.

A Figura 42 apresenta o encapsulamento visual do controlador. Nela, é possível observar os valores de todos os CSRs escrevíveis. Bits não utilizados (sempre 0) não são exibidos. Pequenas legendas foram adicionadas aos registradores MISA e MIE para indicar o significado dos valores. O bit de privilégio foi posicionado ao lado do registrador MSTATUS, pois, apesar de não estar incluído no CSR, trata-se de uma informação relevante.

A interface com o Controlador de Operações foi mantida à esquerda, enquanto a interface com o Contador de Programas permaneceu na parte inferior do componente. Os

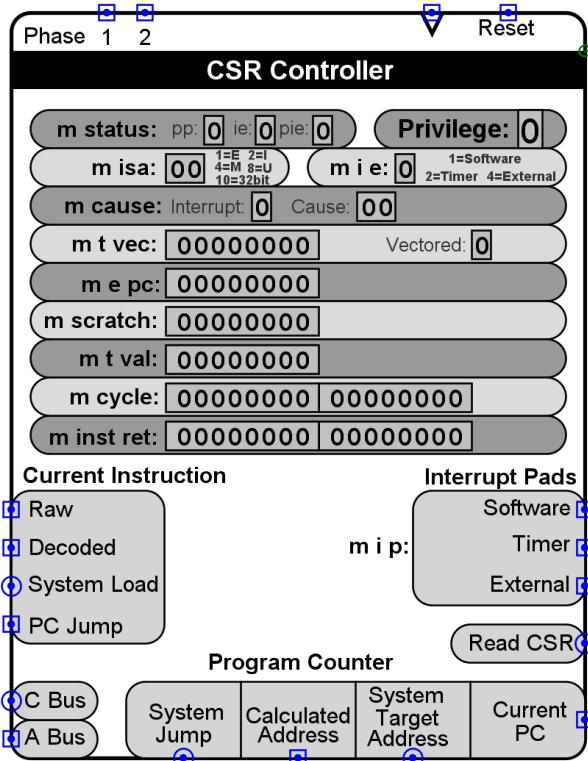


Figura 42 – Encapsulamento do Controlador de CSR

sinais de interrupção foram posicionados à direita, permitindo uma melhor organização do circuito geral, conforme ilustrado na Figura 43. Durante o desenvolvimento do novo componente, foram utilizados os mesmos programas de teste empregados na simulação SV.

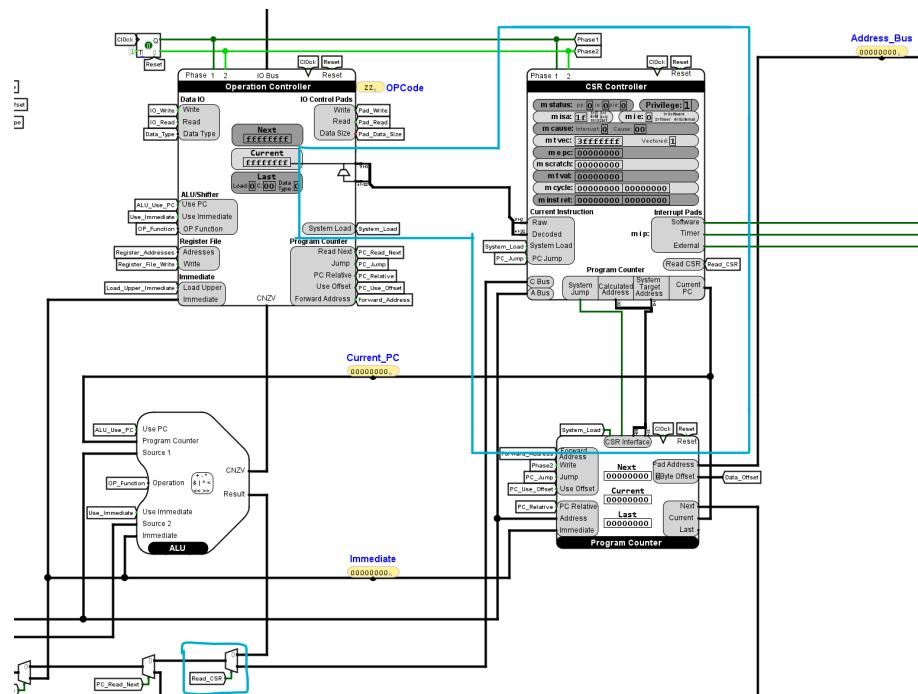


Figura 43 – Integração do Controlador de CSR na arquitetura

Um dos CSRs mencionados anteriormente, o TIME (e, consequentemente, o TI-

MEH), é mapeado na memória do processador, e não no próprio processador. Isso significa que, para acessá-lo, é necessário executar uma instrução de carregamento ou salvamento. Como mostrado anteriormente, a leitura do CSR pode ser realizada através de uma instrução de manipulação de CSR, utilizando o mecanismo criado de carregamento de sistema.

Esse CSR desempenha um papel crucial no mecanismo de interrupção por tempo do processador. Ele opera por meio de dois registradores de 64 bits: um que armazena o tempo atual (`m_time`) e outro que define o tempo da próxima interrupção (`m_cmp_time`) (RISC-V, 2022, p.5). Quando `m_time` ultrapassa `m_cmp_time`, um sinal de interrupção é enviado ao processador por meio de um fio de controle.

Na implementação pela simulação SV, o componente responsável pelo armazenamento e controle desses registradores recebeu um sinal de clock secundário, encarregado da incrementação de `m_time`. Esse clock pode ser configurado com uma frequência maior ou menor que o restante da simulação. Frequências mais altas, em até 10 vezes o clock do processador, não impactam significativamente a performance. No entanto, frequências excessivamente elevadas causaram degradação no desempenho.

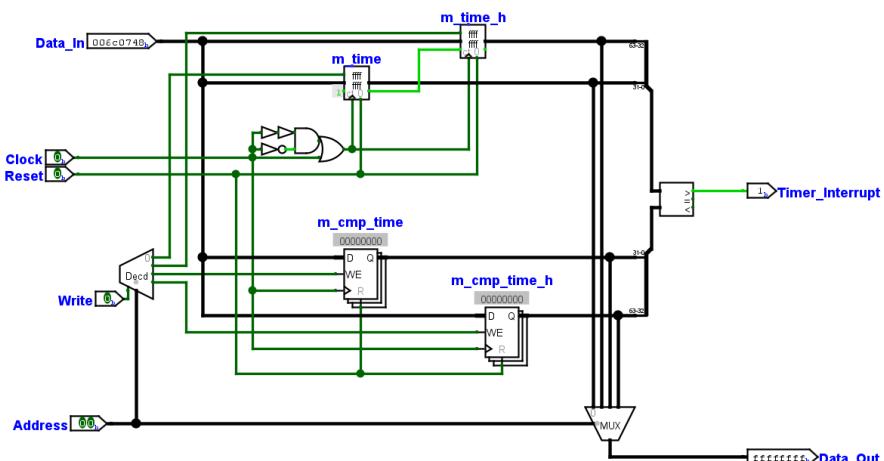


Figura 44 – Circuito do Dispositivo de Tempo Real

Já no Logisim (Figura 44), não é possível gerar um clock com frequência diferente sem a criação de componentes customizados. A redução da frequência do processador também não foi viável, pois a simulação do Logisim já operava em uma frequência bastante baixa, entre 30 e 60 Hertz. Para simular frequências mais altas, foi utilizado um gerador de pulso ativado na borda de descida do `clock`. Esse mecanismo, combinado com o sinal de `clock` normal, efetivamente dobra os sinais responsáveis pela incrementação de `m_time`.

O gerador de pulso consiste em dois *buffers*, uma porta `NÃO` e uma porta `E`. Quando o sinal de `clock` é recebido, uma das entradas da porta `E` é atualizada primeiro, ativando temporariamente a porta antes que a outra entrada seja ajustada.

Os registradores são mapeados contiguamente na memória a partir do endereço

0x81000000 e são divididos em porções de 32 bits, permitindo operações de leitura e escrita. A Figura 45 ilustra o componente desenvolvido e suas interfaces. A entrada de endereços recebe os bits 2 e 3 do endereço de memória, garantindo o mapeamento correto de cada registrador em blocos de 4 bytes.

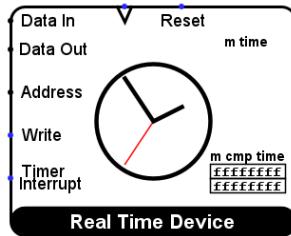


Figura 45 – Encapsulamento do Dispositivo de Tempo Real

O registrador responsável pela interrupção por software (RISC-V, 2022, p.8) foi implementado de maneira simplificada, utilizando apenas um *flip-flop*. Esse registrador pode ser modificado por meio de instruções de armazenamento no endereço 0x81000010, logo após o último endereço do Dispositivo de Tempo Real. A Figura 46 apresenta a implementação do registrador.

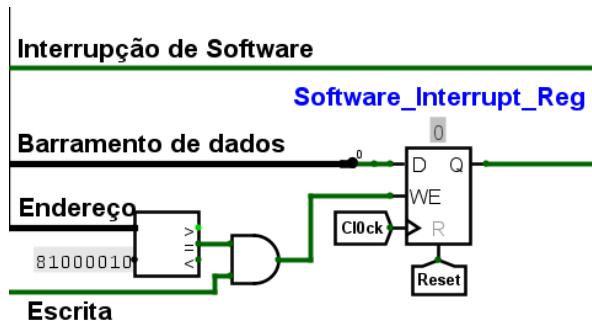


Figura 46 – Registrador de Interrupção de Software

4.7.1.6 Sexta Fase: Pipeline

Perto do fim do desenvolvimento do controlador de CSR, foi criado um programa para a arquitetura, com o objetivo de processar as possíveis exceções e interrupções que podem ocorrer durante a execução de outros programas. Esse programa é o *Trap Handler*, o qual é executado assim que uma exceção ocorre ou no ciclo seguinte à ocorrência de um sinal de interrupção. Na implementação atual, ele deve estar armazenado no endereço indicado pelo CSR `mtvec`, inicializado como 0x80000000.

O *Trap Handler* foi inicialmente implementado de forma simples, lidando apenas com exceções e imprimindo-as no terminal. O objetivo era tanto apresentar um exemplo básico de como os erros são processados no sistema quanto viabilizar uma forma prática de testar se todas as exceções estavam sendo tratadas corretamente.

Durante o desenvolvimento, verificou-se que, apesar de o *Trap Handler* ser chamado e executado corretamente, o processador continuava a executar as instruções até o fim, comportamento incorreto que poderia causar outros erros. Esse problema fazia com que instruções de carregamento, salvamento e desvio fossem executadas completamente, mesmo quando geravam exceções, resultando em modificações indesejadas nos registradores e na memória.

Para corrigir esse problema, um sinal de exceção deveria ser transmitido do controlador de CSR para o controlador de operações, que então deveria desativar os sinais correspondentes às instruções afetadas. Durante esse processo, decidiu-se implementar uma correção que alteraria a forma como esse recurso seria integrado. Essa correção diz respeito a um equívoco cometido na implementação da *Pipeline* do processador, presente desde sua primeira versão.

O erro consistia em posicionar o registrador da instrução atual antes da etapa de decodificação. Embora esse equívoco já tivesse sido identificado anteriormente, não havia sido corrigido por não causar problemas nas simulações. No entanto, em um processador real, isso poderia limitar o tempo disponível para decodificar a instrução antes de sua execução, comprometendo a frequência máxima de operação e eliminando qualquer ganho proporcionado pela pipeline.

Inicialmente, as alterações foram planejadas apenas no controlador de operações, já que o controlador de CSR possui grande complexidade e não é essencial para o funcionamento básico do processador. O planejamento buscou utilizar o menor número possível de registradores, alocados o mais próximo possível dos sinais de controle, imediatamente antes das portas lógicas conectadas aos sinais de fase.

O registrador da instrução atual foi mantido, tanto por conter informações que não exigem decodificação lógica quanto por ser utilizado pelo controlador de CSR. Além disso, foram adicionados registradores associados diretamente a instruções específicas, como **load**, **store**, **branch**, **LUI** (Load Upper Immediate), **AUIPC** (Add Upper Immediate to PC), bem como registradores compartilhados entre múltiplas instruções, como para desvios não condicionais (**JAL** e **JALR**), instruções relativas ao endereço atual (**Branch** e **JAL**), dois registradores para instruções da ULA, e um registrador para controle de escrita no arquivo de registradores e um registrador para o valor imediato.

O controlador de programa também inclui três registradores introduzidos em etapas anteriores, os quais substituem o registrador da última instrução. Esses registradores são específicos para instruções de carregamento, únicas a possuírem um terceiro ciclo na pipeline. São eles: o registrador de **load** anterior, o endereço do registrador c alvo e o registrador de tipo de dado (tamanho e sinal).

Alguns sinais de controle não utilizaram registradores ou foram alocados ainda

com uma ou duas portas lógicas após os registradores. Isso se deve ao fato de que esses sinais são gerados durante a fase de execução ou precisariam de mais um registrador que aumentaria a complexidade do circuito sem melhorias significativas na velocidade de decodificação. A Figura 47 ilustra o novo controlador de operações, com as devidas correções à pipeline.

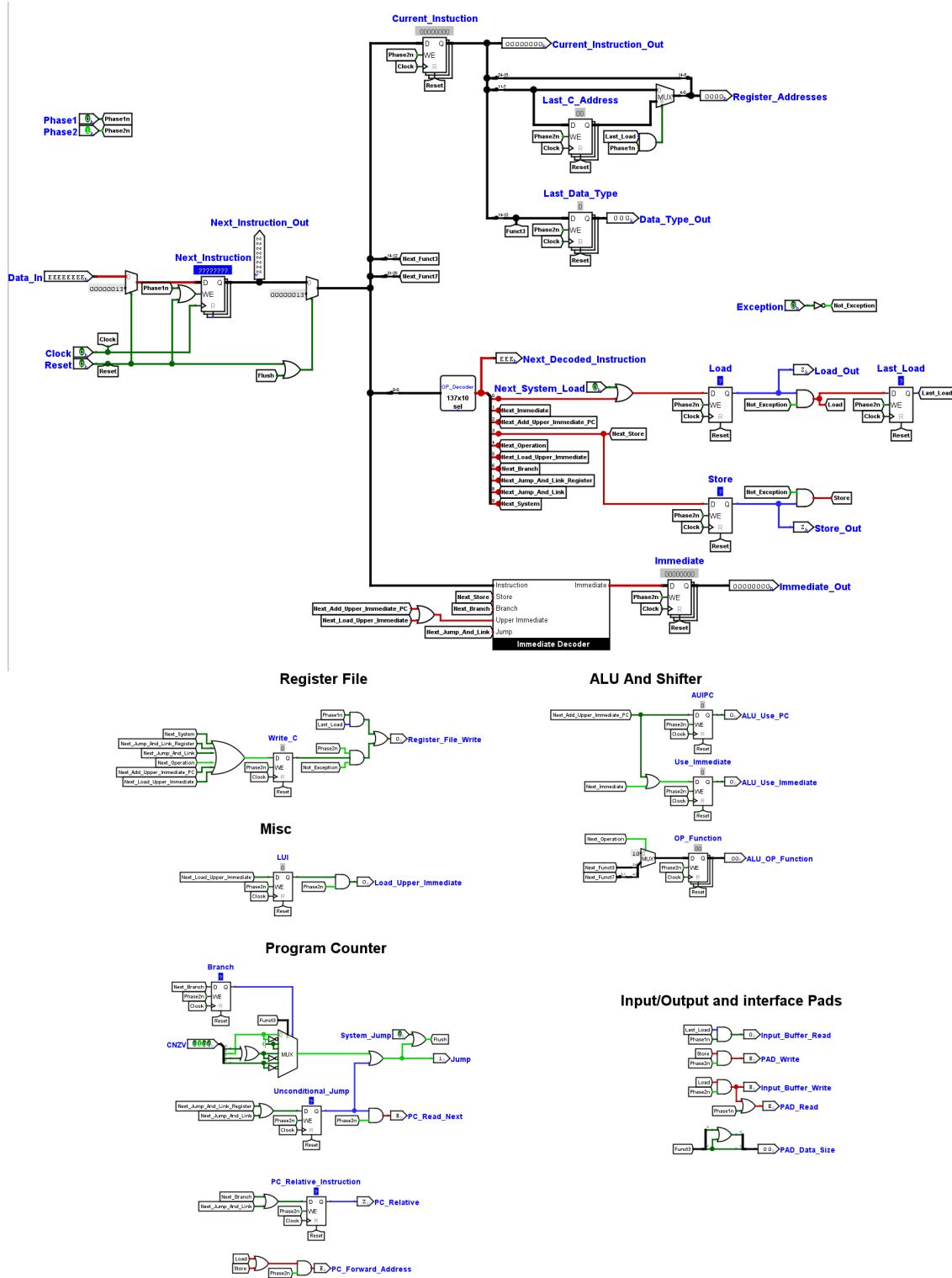


Figura 47 – Controlador de Operações com *Pipeline* Corrigida.

Além dos registradores adicionados, é possível ver a implementação do sinal de exceção, que assim que ativado, impede que os sinais de carregamento, salvamento e escrita no Arquivo de Registradores continuem ativados. Isso evita os efeitos colaterais provenientes de instruções que causam exceções.

Com o controlador de operações corrigido, o próximo passo foi a revisão do controlador de CSR. Devido à sua complexidade, as alterações iniciais foram realizadas por meio do código em SV. O foco dessas modificações foi o mesmo aplicado ao controlador de instruções: mover o máximo possível da etapa de decodificação para o primeiro ciclo da pipeline. Para isso, realizou-se primeiramente uma análise do circuito, visando identificar quais sinais são gerados na fase de decodificação e quais são processados durante a execução da instrução.

Os sinais marcados para decodificação no controlador de CSR são aqueles relacionados às instruções seguintes, incluindo se a próxima instrução é uma instrução de sistema (`mret`, `ecall`, `ebreak` e `csrrw`) ou se é uma instrução ilegal. As demais informações necessárias são fornecidas pelo controlador de operações e abrangem o valor da palavra da próxima instrução, o valor da instrução atual, a próxima instrução já decodificada, e se a instrução atual corresponde a um `load`, `store` ou se está executando um desvio.

A Figura 48 apresenta os circuitos que receberam registradores adicionais no controlador de CSR. Nota-se que, em comparação com versões anteriores, quase todos os sinais de entrada foram modificados para refletirem valores referentes à próxima instrução.

A única exceção a essa regra é o teste de privilégio ao acessar os CSRs, pois a única maneira de o nível de privilégio ser alterado é mediante uma *trap* ou através da instrução `mret`. Após um desses eventos ocorrer, um desvio é executado, e como desvios são sempre seguidos por um ciclo de instrução vazio (`nop`), torna-se impossível que o privilégio seja modificado entre duas instruções consecutivas.

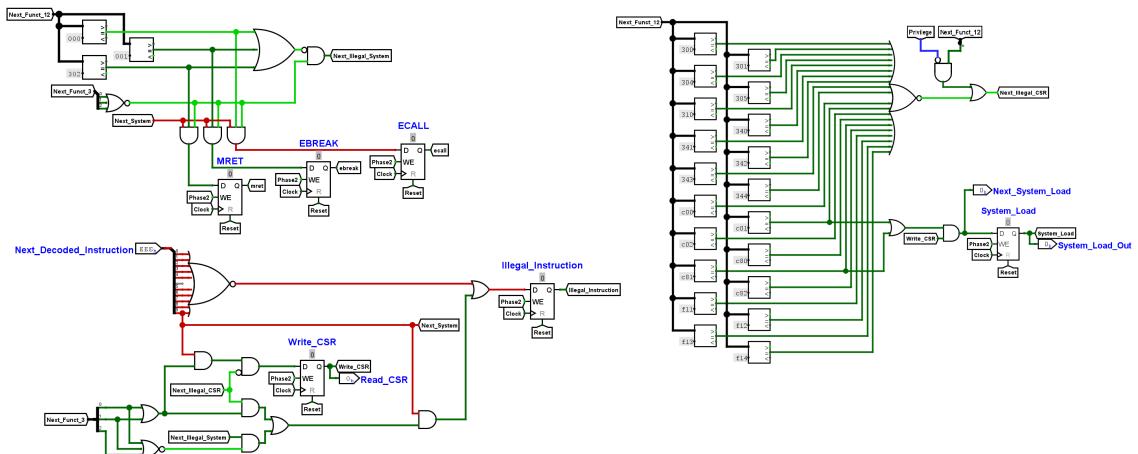


Figura 48 – Controlador de Operações com *Pipeline Corrigida*.

Durante as correções executadas, outro componente que foi modificado foi o

Contador de Programa. Foi observado que os sinais de pulo eram sempre ativados durante a segunda fase do pipeline. No entanto, não era necessário combinar o sinal de fase nos controladores, pois o próprio Contador de Programa já recebia esse sinal para atualizar seu registrador interno por meio do sinal `write`.

Dessa forma, o sinal `write` foi combinado diretamente com os sinais de desvio dentro do próprio contador, o que resultou na simplificação de algumas partes do circuito. As modificações realizadas no Contador de Programa estão detalhadas no Código 4.1.

```

1 //antes:
2     if(reset) [...]
3     else if(system_jump) next_reg <= system_address_target[31:2];
4     else if(jump) next_reg <= calculated_address[31:2];
5     else if(write) next_reg <= next_reg + 1;
6
7     if(write)
8         last_reg <= current_reg;
9         current_reg <= next_reg;
10    end
11 //depois:
12    if(reset) [...]
13    else if(write) begin
14        if(system_jump) next_reg <= system_address_target[31:2];
15        else if(jump) next_reg <= calculated_address[31:2];
16        else next_reg <= next_reg + 1;
17
18        current_reg <= next_reg;
19    end

```

Listing 4.1 – Alterações realizadas no Contador de Programa

Por fim, outra modificação realizada foi a remoção do registrador responsável por armazenar a última instrução no Contador de Programa. Após alterações estruturais em passos anteriores, esse registrador deixou de ter relevância na arquitetura e, portanto, pôde ser eliminado sem impactos funcionais.

Durante o processo de desenvolvimento dessa fase, melhorias foram realizadas no código SV para impressão de mensagens mais informativas. Alterações inclusas foram a adição de uma coluna para exibir valores decimais com sinal ao lado de cada registrador e a adição de uma coluna com valores ASCII correspondentes a cada byte da memória RAM, que podem ser vistos na Figura 49.

Também foi adicionado uma opção para desativar a geração do controlador de CSR no processador no código SV, permitindo uma execução mais simplificada de programas que não utilizam interrupções ou exceções.

Register values:	RAM values:	
R[0] (zero) = 00000000 0	address data (hex)	data (ascii)
R[1] (ra) = 00000024 36	00000000: 13 00 00 00.83 22 80 0e.03 23 c0 0e.37 01 0f 00	? ? ? ? à " Ç ? # ? l ? ? ?
R[2] (sp) = 000f0000 983040	00000010: 13 85 02 00.93 05 00 00.33 06 53 40.13 06 f6 ff	? à ? ? ô ? ? ? 3 ? 5 @ ? ? ?
R[3] (gp) = 00000000 0	00000020: ef 00 80 00.6f 00 00.63 da c5 04.13 01 01 ff	' ? Ç ? o ? ? ? c ↕ + ? ? ?
R[4] (tp) = 00000000 0	00000030: 23 26 11 00.23 24 31 01.23 22 21 01.23 20 91 00	# & ? ? # \$ 1 ? # " ! ? # æ ?
R[5] (tf) = 00000064 100	00000040: 93 84 05 00.13 09 00.ef 00 80 03.93 89 06 00	ô à ? ? ? ? ? ? ' ? Ç ? ô è ? ?
R[6] (t1) = 0000000d 13	00000050: 93 85 04 00.13 86 9f ff.ef f0 1f fd.93 85 19 00	ô à ? ? ? ? à " ' ? ? à ? ?
R[7] (t2) = 0000000d 13	00000060: 13 06 09 00.ef 0f 5f fc.83 24 01 00.03 29 41 00	? ? ? ? ' ? ? ? ?) A ?
R[8] (s0) = 00000000 0	00000070: 83 29 81 00.83 20 c1 00.13 01 01.67 88 00 00	â) û ? à I ? ? ? ? ? g Ç ? ?
R[9] (s1) = 00000000 0	00000080: b3 02 c5 00.83 82 00.13 83 f5.93 83 05 00	? + ? à é ? ? ? à § ô à ? ?
R[10] (a0) = 00000100 256	00000090: 93 0c f6 ff.63 c8 7c 02.b3 0e 75 00.83 8f 0e 00	ô ? + c ¶ ? u ? à Å ? ?
R[11] (a1) = 00000010 16	000000a0: 13 00 00 00.63 dc 5f 00.13 03 13 00.33 0e 65 00	(Zeroes Omitted) ? ? ? ? c ■ - ? ? ? ? ? e ?
R[12] (a2) = 0000000f 15	000000b0: 03 0f 0e 00.23 00 fe 01.23 80 ee 01.93 83 13 00	? ? ? ? # ? ■ ? # Ç " ? ô à ? ?
R[13] (a3) = 0000000d 13	000000c0: 6f f0 5f fd.13 03 13 00.33 0e 65 00.03 0f 0e 00	o x ? ? ? ? 3 ? e ? ? ? ? ?
R[14] (a4) = 00000000 0	000000d0: b3 0e c5 00.83 8f 0e 00.23 88 ee 01.23 00 fe 01	? + ? à Å ? ? # Ç " ? # ? ■ ?
R[15] (a5) = 00000000 0	000000e0: 93 06 03 00.67 88 00.00.00 01 00.00.10 01 00 00	ô ? ? ? g Ç ? ? ? ? ? ? ? ? ? ? ?
R[16] (a6) = 00000000 0		
R[17] (a7) = 00000000 0		
R[18] (a8) = 00000000 0		
R[19] (s3) = 00000000 0	00000100: 30 31 32 33.34 35 36 37.38 39 61 62.63 64 65 66	0 1 2 3 4 5 6 7 8 9 a b c d e f
R[20] (s4) = 00000000 0	(Zeroes Omitted)	
R[21] (s5) = 00000000 0	00000fa0: 0b 00 00 00.0d 00 00 00.0b 00 00 00.68 00 00 00	? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
R[22] (s6) = 00000000 0	00000fb0: 0b 00 00 00.0e 00 00 00.0e 00 00 00.5c 00 00 00	? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
R[23] (s7) = 00000000 0	00000fc0: 0b 00 00 00.0f 00 00 00.0f 00 00 00.5c 00 00 00	? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
R[24] (s8) = 00000000 0	00000fd0: 0b 00 00 00.0f 00 00 00.0a 00 00 00.68 00 00 00	? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
R[25] (s9) = 0000000c 12	00000fe0: 0b 00 00 00.0f 00 00 00.08 00 00 00.68 00 00 00	? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
R[26] (s10) = 00000000 0	00000ff0: 00 00 00 00.00 00 00 00.00 00 00 00.24 00 00 00	? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
R[27] (s11) = 00000000 0		
R[28] (t3) = 0000010d 269		
R[29] (t4) = 0000010d 269		
R[30] (t5) = 00000064 100		
R[31] (t6) = 00000064 100		

Figura 49 – Saída do programa de teste com melhorias na impressão.

4.7.2 Processo de Desenvolvimento - Assembler

O desenvolvimento do *assembler* foi realizado paralelamente ao desenvolvimento da arquitetura. Inicialmente, o objetivo era obter uma forma de criar programas para que pudessem ser executados nos simuladores. Esse objetivo se tornou menos relevante quando o padrão RISC-V foi adotado, pela existência de outros *assemblers* mais completos que poderiam ser utilizados.

Logo, o objetivo foi obter um *assembler* mais simples, que pudesse exibir como que as instruções são traduzidas para o código de máquina, e que fornecesse informações importantes para o *debugging* de programas, como o endereço de cada instrução.

Além de melhorias no código de tradução de instruções, foram realizadas melhorias no código de pré processamento de código, com aperfeiçoamentos na remoção de espaços em branco e comentários e a adição de suporte para *macros* e *defines*, que facilitam a criação de código repetitivo e definições que facilitam a leitura do código.

4.7.2.1 Adaptação para RISC-V

A primeira versão do *assembler* apresentava limitações significativas, pois utilizava um processo de decodificação fixo para todas as instruções, o que dificultava a adição ou modificação de formatos diferentes. No caso do RISC I, isso não representava um problema, uma vez que havia apenas dois formatos de instrução: **OPCode <rd|cond> <rs1> <rs2|Imediato13>** e **OPCode <rd> <imediato-19>**. Contudo, no RISC-V existem diversos outros formatos, além de pseudoinstruções definidas especificamente para *assemblers*(RISC-V, 2025).

Para solucionar essa limitação, foi realizada uma reescrita do *assembler*. Em vez de adotar uma tradução fixa, a nova versão passou a utilizar *dicionários* de funções de

tradução. Assim, para adicionar novas instruções, basta incluí-las no dicionário, usando o nome como chave e associando a ele uma função que recebe uma lista de parâmetros como entrada e retorna o código de máquina correspondente como saída.

Essa abordagem não apenas ampliou a flexibilidade na criação de novas instruções, mas também possibilitou a visualização direta de todas as instruções disponíveis no *assembler*, por meio de uma lista no próprio código.

Nas versões mais recentes, esse comportamento foi aprimorado para permitir que uma pseudoinstrução pudesse ser traduzida em múltiplas instruções. Para isso, adotou-se um dicionário de funções geradoras, responsáveis por criar as funções de tradução necessárias durante o processo de montagem.

Por exemplo, ao acessar uma determinada instrução no dicionário, obtém-se uma função `geradorDeTradutor(string[] parâmetros)`, que por sua vez retorna uma lista de funções. Cada função dessa lista, quando executada, gera o valor numérico de uma instrução necessária para a implementação completa da pseudoinstrução.

Embora a maioria das instruções seja traduzida para apenas uma instrução de máquina, existem pseudoinstruções que podem gerar mais de uma, como `li`, `la` e `call`.

As funções de tradução foram implementadas a partir de funções genéricas, responsáveis por converter os formatos comuns de instrução especificados por [RISC-V \(2024b\)](#), conforme ilustrado na Figura 50.

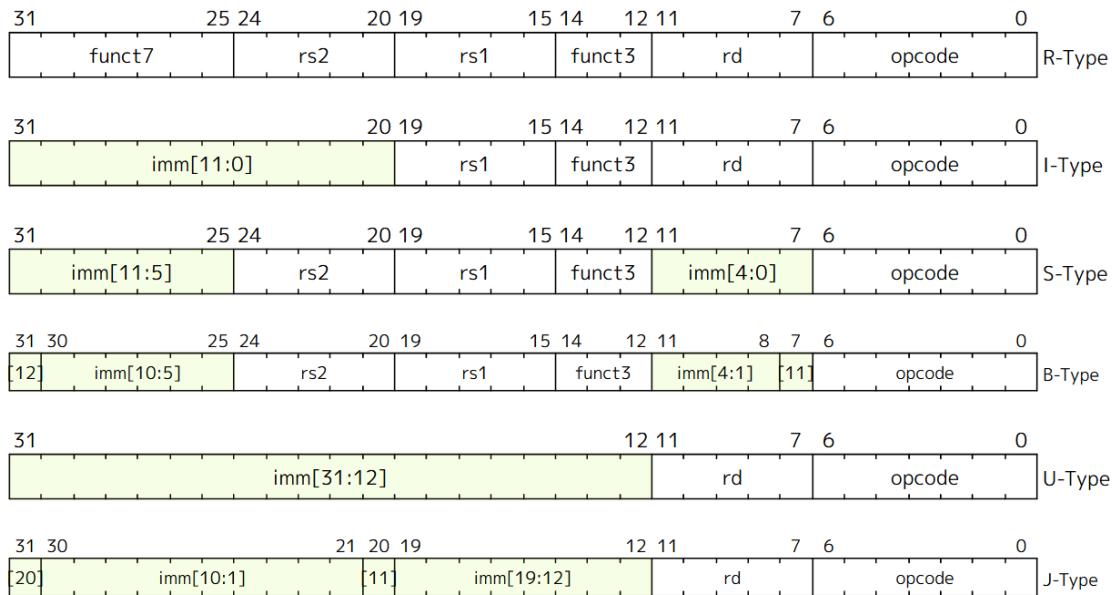


Figura 50 – Formatos de instrução base do RISC-V ([RISC-V, 2024b](#)).

Para utilizar essas funções genéricas, é necessário fornecer determinados parâmetros. Todos os parâmetros relacionados ao tipo de instrução foram definidos como enumeráveis, facilitando a tradução e garantindo melhor legibilidade. Entre eles estão: instruções base,

nomes de registradores, operações matemáticas, tipos de comparação para instruções de desvio, tamanhos de dados, funções de sistema e nomes de CSR.

Já os parâmetros correspondentes a valores numéricos são fornecidos como texto, de forma que possam ser processados por um método específico capaz de diferenciar automaticamente valores hexadecimais e decimais.

4.7.2.2 Variáveis

Foi adicionado suporte à declaração de variáveis com tipos numéricos de 8, 16 e 32 bits, além de *strings*. Essas variáveis são decodificadas por meio de um dicionário com estrutura semelhante ao utilizado para instruções, permitindo que cada variável possua um nome opcional e/ou uma sequência de valores contíguos na memória.

A Tabela ?? apresenta exemplos de declarações válidas:

Tabela 8 – Exemplos de declarações de variáveis e seus valores em compilados

Tipo	Declaração	Valor Compilado
<code>.string</code>	<code>.string teste "uma string"</code>	756b6120-73747269-6e670000
	<code>.string "a string"</code>	61207374-72696e67-00000000
	<code>.string "1"</code>	00000031
<code>.word</code>	<code>.word umInteiro 0x03bb00ff</code>	03bb00ff
	<code>.word umVetor 1 2 3</code>	00000001-00000002-00000003
<code>.half</code>	<code>.half umHalf 0xffff</code>	0000ffff
	<code>.half umVetorHalf 0xffff 0xeeee 0xCCCC</code>	eeeeffff-0000CCCC
	<code>.half 0xffff</code>	0000ffff
<code>.byte</code>	<code>.byte umByte 128</code>	00000080
	<code>.byte umVetorbyte 0xff 0xee 0xbb</code>	00bbeeff
	<code>.byte 0xff 0xee 0xbb</code>	00bbeeff

Devido ao funcionamento alinhado a palavras de 32 bits do sistema, variáveis com tamanho inferior a esse limite são automaticamente completadas com zeros à esquerda. Por exemplo, ao declarar um byte com o valor `0xff`, o valor armazenado será `000000ff`.

No caso de *strings*, os caracteres são agrupados sequencialmente em palavras de 32 bits, respeitando a ordem dos bytes. Como toda *string* deve ser finalizada com o caractere nulo (`0x00`), uma palavra adicional contendo esse valor é inserida quando o último caractere ocupa exatamente o último byte da última palavra. Isso garante que o terminador esteja sempre presente na memória, mesmo quando a *string* ocupa múltiplos de 4 bytes.

Quando uma lista de valores é declarada, os dados são organizados em sequência na memória, e o nome associado à variável corresponde ao endereço do primeiro elemento da lista.

Em *assemblers* desenvolvidos para sistemas reais, as variáveis são geralmente montadas em uma seção separada após o código. No entanto, para fins didáticos e para facilitar a compreensão do funcionamento interno, nesta versão as variáveis são inseridas diretamente no ponto em que foram declaradas.

4.7.2.3 Sintaxe

A sintaxe adotada para o novo *assembler* é semelhante à utilizada em outros *assemblers* RISC-V, porém apresenta algumas simplificações. Por exemplo, não é necessário utilizar vírgulas para separar os parâmetros de uma instrução. Assim, a instrução “`addi a5,a5,31`” torna-se “`addi a5 a5 31`”.

Outra modificação refere-se à forma de indicar deslocamentos de memória, aproximando-se mais da maneira como a instrução é traduzida para código de máquina. Em um código *assembly* RISC-V tradicional, o deslocamento aparece antes do endereço, que fica entre parênteses, como em “`sb a5,3(s0)`” (armazenar o byte no endereço contido em `s0 + 3`). No novo formato, essa mesma instrução é escrita como “`sb a5 s0 3`”.

Há também diferenças na forma de declarar e chamar rótulos e variáveis. Para declarar um rótulo, o caractere “`:`” é adicionado no início do nome, e não ao final. Por exemplo, `procedimento teste:` torna-se `:procedimento teste`. Já nas chamadas, os rótulos são sempre precedidos por “`:`”, enquanto variáveis são precedidas por “`.`”.

```

46 .macro addVector xRegR yRegR xReg1 yReg1 Xreg2 Yreg2
47     add xRegR xReg1 Xreg2
48     add yRegR yReg1 Yreg2
49 .endmacro
50
51     li sp 0x00010000
52     li videoRegister videoAddress
53     li colorRegister white //colorReg white
54     li s3 0xffff0000 //address bits
55     li s4 1 //snake size
56     li s5 0 //current head offset
57     setVector s0 s1 0x1f 0x1f //setting starting position to center of the display
58
59     setVector t2 t3 1 0 //setting initial speed
60
61 :loopStart
62     lb t1 gp -1
63     blt t1 zero :draw //if there is no input the value will be negative
64
65 #REGION "switch" (input)
66     beqi t1 0x77 :w a0
67     beqi t1 0x61 :a a0
68     beqi t1 0x73 :s a0
69     beqi t1 0x64 :d a0
70     beqi t1 0x71 :loopEnd a0
71
72 :w
73     setVector t2 t3 0 -1
74     jump :draw
75 :a
76     setVector t2 t3 -1 0
77     jump :draw
78 :s
79     setVector t2 t3 0 1
80     jump :draw
81 :d
82     setVector t2 t3 1 0
83 #ENDREGION

```

Figura 51 – Exemplo de código com destaque de sintaxe no Notepad++.

Para melhorar a legibilidade, foi criado um arquivo de destaque de sintaxe para o

editor Notepad++, que pode ser importado facilmente. Esse destaque oferece suporte a tema escuro, retração de blocos de *macros* e definição de regiões personalizadas por meio do comando “#REGION”. A Figura 51 ilustra um exemplo de código de teste exibido com essa configuração.

4.7.2.4 Pré-processamento

O pré-processamento do código pode ser dividido em quatro etapas principais. A primeira consiste na remoção de espaços em branco e comentários, com o objetivo de realizar uma limpeza inicial do conteúdo. Em seguida, todas as declarações de *defines* são extraídas do código e armazenadas em um dicionário. Após essa coleta, os *defines* são removidos do código original e substituídos por seus respectivos valores nas chamadas correspondentes.

Na mesma função responsável pela substituição dos *defines*, ocorre também a substituição de macros. Esse processo segue uma lógica semelhante, porém leva em consideração os parâmetros de entrada das macros. Para isso, os parâmetros devem ser armazenados no dicionário da macro, permitindo que sejam corretamente inseridos durante a expansão da chamada.

O Código 4.2 apresenta um exemplo de macro que realiza um desvio condicional caso um registrador seja igual a um valor imediato. Para a declaração da macro, utiliza-se a diretiva “.macro”, seguida do nome da macro (no caso, `beqi`) e dos parâmetros que serão usados. O corpo da macro contém as instruções desejadas e é finalizado pela diretiva “.endmacro”.

```

1 .macro beqi reg immediate jumpTarget auxReg
2     li auxReg immediate
3     beq reg auxReg jumpTarget
4 .endmacro
5
6 // Exemplo de chamada da macro
7     beqi a0 0 :enderecoPulo t0
8
9 // Resultado da expansão da macro
10    li t0 0
11    beq a0 t0 :enderecoPulo

```

Listing 4.2 – Exemplo de macro

Após as macros, vem o último passo: a indexação dos endereços de variáveis e rótulos em um dicionário. Os rótulos são removidos do código, e os endereços correspondentes — tanto de rótulos quanto de variáveis — são traduzidos nos pontos em que são referenciados, concluindo o processo de pré-processamento.

4.7.2.5 Processo de montagem

Ao executar o *assembler*, um terminal é aberto para o usuário, exibindo todos os códigos *assembly* encontrados como opções em uma lista. Após o usuário digitar o número correspondente a uma das opções, a montagem é iniciada. Com exceção da remoção de espaços em branco e comentários, todos os passos do pré-processamento são exibidos no terminal, incluindo tanto a indexação quanto a substituição de textos.

Concluído o pré-processamento, inicia-se a tradução de cada linha do código. As informações exibidas ao usuário incluem: a linha da instrução no código já pré-processado, o endereço esperado para a instrução, o valor binário resultante e a instrução original que gerou esse código.

O valor binário e a instrução são impressos com cores distintas para cada parte da instrução, o que facilita a identificação de problemas e a compreensão de como cada componente da instrução é traduzido. A Figura 52 apresenta o resultado da montagem de um programa que realiza *insertion sort* em uma lista de *bytes*.

[Line]	[Memory_Address]	Translated_Binary	Original_Code
[0][0]		000000000000000000000000000010011	nop
[1][4]		0000010110000000010001010000011	lw t0 zero 88
[2][8]		0000010111000000010001100000011	lw t1 zero 92
[3][c]		00000000000011110000000100110111	lui sp 0xf0
[4][10]		000000000000001010000101000100111	mv a0 t0
[5][14]		010000000101001100000101101100111	sub a1 t1 t0
[6][18]		0000000010000000000000000011101111	call 8
[7][1c]		0000000000000000000000000011011111	jump 0
[8][20]		00000000000000000000000010100100111	addi t0 zero 0
[9][24]		0000000000001001010000010100100111	inc t0 1
[10][28]		0000000000000010100000110001000111	mv t1 t0
[11][2c]		000000101011001011010100011000111	bge t0 a1 40
[12][30]		000000001100010100000111011000111	add t2 a0 t1
[13][34]		1111111111100111000111000000011	lb t3 t2 -1
[14][38]		000000000000001110000111010000011	lb t4 t2 0
[15][3c]		11111110011000001010100111000111	ble t1 zero -24
[16][40]		111111111100111011010010111000111	ble t3 t4 -28
[17][44]		000000111000011100000000010000111	sb t3 t2 0
[18][48]		111111111101001110001111101000111	sb t4 t2 -1
[19][4c]		11111111111001100000011000100111	dec t1 1
[20][50]		11111110000111111110000011011111	jal zero -32
[21][54]		000000000000000000010000000001100111	ret
[22][58]		00000000000000000000000000001110000	.word basePointer 112
[23][5c]		00	.word arrayEnd 128
[24][60]		00	.word 0 0 0 0
[64]		00	
[68]		00	
[6c]		00	
[25][70]		10101010100110011100110001110111	.word elemento0 0xAA99CC77
[26][74]		11101110001100111111110000000000	.word elemento1 0xEE33FF00
[27][78]		1101110100100010010001001000100	.word elemento2 0xDD112244
[28][7c]		10001000010101011011011011000110	.word elemento3 0x8855BB66

Figura 52 – Visualização da montagem de um programa de ordenação por inserção

Ao final da execução, um arquivo contendo os valores hexadecimais é gerado, podendo ser utilizado diretamente no Logisim Evolution. Caso o usuário deseje utilizar o programa em uma simulação Verilog, esse arquivo deverá ser ajustado, pois contém informações adicionais de endereçamento e cabeçalho que precisam ser removidas.

O *assembler* também possui mecanismos para a detecção de alguns erros simples de sintaxe e semântica. No entanto, devido à grande variedade de possíveis equívocos que podem ser cometidos durante a escrita do código, ainda existem casos que não são tratados adequadamente. Nessas situações, podem ser exibidas mensagens de erro geradas pela própria linguagem C#, as quais, em sua maioria, são pouco descritivas e não oferecem informações suficientes para auxiliar na resolução do problema.

5 Resultados

6 Conclusão

Considerações finais

Referências

- ARM. *Building the Future of Computing*. Disponível em: <<https://www.arm.com/>>. Citado na página 39.
- ARM. *ARM Cortex-A Series Programmer's Guide for ARMv7-A*. 2014. Disponível em: <<https://developer.arm.com/documentation/den0013/d>>. Citado 3 vezes nas páginas 21, 39 e 42.
- BURCH, C. et al. *Logisim Evolution*. 2014. Disponível em: <<https://github.com/logisim-evolution/logisim-evolution>>. Citado na página 29.
- CATSOULIS, J. *Designing embedded hardware*. 1st ed. ed. Sebastopol, CA: O'Reilly, 2002. OCLC: ocm51226672. ISBN 978-0-596-00362-3. Disponível em: <<https://archive.org/details/designing-embedded-hardware/page/n5/mode/1up>>. Citado 7 vezes nas páginas 15, 17, 18, 19, 21, 24 e 25.
- CHWIF, L. *Modelagem e simulação de eventos discretos: teoria & aplicações*. 4a. ed. [S.l.]: Elsevier Brasil, 2014. ISBN 978-85-352-7933-7. Citado na página 25.
- CLEMENTS, A. Selecting a processor for teaching computer architecture. *Microprocessors and Microsystems*, v. 23, n. 5, p. 281–290, out. 1999. ISSN 01419331. Disponível em: <<https://linkinghub.elsevier.com/retrieve/pii/S0141933199000496>>. Citado na página 10.
- CLEMENTS, A. ARMs for the poor: Selecting a processor for teaching computer architecture. In: *2010 IEEE Frontiers in Education Conference (FIE)*. [s.n.], 2010. p. T3E-1–T3E-6. ISSN: 2377-634X. Disponível em: <<https://ieeexplore.ieee.org/abstract/document/5673541>>. Citado na página 10.
- COSTA, H. Modelo para webibliomining: proposta e caso de aplicação. v. 13, p. 115–126, 2010. Citado na página 38.
- Daniel T. Fitzpatrick et al. A RISCy approach to VLSI. *ACM Sigarch Computer Architecture News*, v. 10, n. 1, p. 28–32, jan. 1982. MAG ID: 2048226925. Citado na página 23.
- DIMITRIJEVIC, S.; DEVEDZIC, V. Usability Evaluation in Selecting Educational Technology. In: . Zrenjanin, Serbia: [s.n.], 2020. ISBN 978-86-7672-341-6. Disponível em: <https://www.researchgate.net/publication/346403516_Usability_Evaluation_in_Selecting_Educational_Technology>. Citado na página 12.
- DJORDJEVIC, J.; MILENKOVIC, A.; GRBANOVIC, N. An integrated environment for teaching computer architecture. *IEEE Micro*, v. 20, n. 3, p. 66–74, maio 2000. ISSN 1937-4143. Disponível em: <<https://ieeexplore.ieee.org/document/846311>>. Citado na página 10.
- FÁVERO, E. *Organização e arquitetura de computadores*. Pato Branco: Edutfpr, 2011. ISBN 978-85-7014-082-1. Disponível em: <https://redeetec.mec.gov.br/images/stories/pdf/eixo_infor_comun/tec_inf/081112_org_arq_comp.pdf>. Citado 3 vezes nas páginas 10, 17 e 19.

- IEEE. *IEEE Standard for Binary Floating-Point Arithmetic*. IEEE, 1985. ISBN: 9780738111650. Disponível em: <<http://ieeexplore.ieee.org/document/30711/>>. Citado na página 18.
- KATEVENIS, M. Reduced instruction set computer architectures for VLSI. jan. 1985. MAG ID: 2171342458 S2ID: f2e99c1a499176ffe665c9b523080b1ac65665a0. Citado 2 vezes nas páginas 23 e 34.
- KEIM, R. *What Is a Hardware Description Language (HDL)? - Technical Articles*. 2020. Disponível em: <<https://www.allaboutcircuits.com/technical-articles/what-is-a-hardware-description-language-hdl/>>. Citado na página 25.
- LIN, Y. et al. Syntactic Annotations for the Google Books NGram Corpus. jul. 2012. Citado na página 38.
- LOURENÇO, A. E.; MIDORIKAWA, E. T. Ensino de arquitetura de computadores utilizando simuladores completos. 2005. ISSN 1413-215X. Disponível em: <<https://repositorio.usp.br/item/001458472>>. Citado na página 10.
- MALVINO, A. P.; BROWN, J. A. *Digital computer electronics*. 3. ed. ed. Lake Forest, Ill.: Glencoe, 1993. ISBN 978-0-02-800594-2 978-0-07-462235-3. Citado 2 vezes nas páginas 22 e 33.
- MIPS Tech LLC. *MIPS32 Architecture For Programmers*. 6. ed. [S.l.: s.n.], 2016. v. 1-2. Citado 3 vezes nas páginas 21, 42 e 47.
- MONTEIRO, M. *Introdução À Organização De Computadores*. [S.l.]: Ltc-Livros Tecnicos E Cientificos Editora Lda, 2007. ISBN 978-85-216-1543-9. Citado 2 vezes nas páginas 16 e 18.
- PATTERSON, D. *How close is RISC-V to RISC-I? / ASPIRE*. 2017. Disponível em: <<https://aspire.eecs.berkeley.edu/2017/06/how-close-is-risc-v-to-risc-i/>>. Citado na página 43.
- PATTERSON, D.; SéQUIN, C. *Design and Implementation of RISC I*. Berkeley, 1982. 25 p. Disponível em: <<http://www2.eecs.berkeley.edu/Pubs/TechRpts/1982/5449.html>>. Citado na página 23.
- PATTERSON, D. A. Reduced instruction set computers. *Communications of The ACM*, v. 28, n. 1, p. 8–21, jan. 1985. MAG ID: 2018426736. Citado na página 42.
- PATTERSON, D. A.; CHEN, T. *RISC-V Geneology*. Berkeley, 2016. 8 p. Disponível em: <<https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-6.pdf>>. Citado na página 23.
- PATTERSON, D. A.; HENNESSY, J. L. *Computer architecture : a quantitative approach*. 2nd ed. ed. San Francisco: Morgan Kaufmann Publishers, 1996. ISBN 978-1-55860-329-5 978-1-55860-372-1. Disponível em: <<http://archive.org/details/computerarchitec00patt>>. Citado 3 vezes nas páginas 10, 21 e 33.
- PEEK, J. *The VLSI Circuitry of RISC I*. Berkeley, 1983. 59 p. Disponível em: <<https://www2.eecs.berkeley.edu/Pubs/TechRpts/1983/6347.html>>. Citado 10 vezes nas páginas 2, 20, 21, 22, 23, 33, 34, 37, 59 e 60.

- RAABE, A.; ZEFERINO, C. *Processadores para Ensino de Conceitos Básicos de Arquitetura de Computadores*. Workshop sobre Educação em Arquitetura de Computadores, 2006. Disponível em: <https://www.researchgate.net/publication/266878017_Processadores_para_Ensino_de_Conceitos_Basicos_de_Arquitetura_de_Computadores>. Citado na página 10.
- RISC-V. *RISC-V Advanced Core Local Interruptor Specification*. Version 1.0-rc4. RISC-V, 2022. Disponível em: <<https://riscv.org/technical/specifications/>>. Citado 2 vezes nas páginas 86 e 87.
- RISC-V. *The RISC-V Instruction Set Manual Volume I: Unprivileged Architecture*. Version 20240411. RISC-V, 2024. I. Disponível em: <<https://riscv.org/technical/specifications/>>. Citado 5 vezes nas páginas 16, 17, 74, 76 e 79.
- RISC-V. *The RISC-V Instruction Set Manual: Volume II: Privileged Architecture*. Version 20240411. RISC-V, 2024. II. Disponível em: <<https://riscv.org/technical/specifications/>>. Citado 11 vezes nas páginas 2, 3, 21, 23, 40, 42, 46, 51, 55, 56 e 93.
- RISC-V. *RISC-V Assembly Programmer's Manual*. Version v0.0.0. [s.n.], 2025. Disponível em: <<https://riscv.org/technical/specifications/>>. Citado na página 92.
- SNYDER, W. *Verilator*. 2003. Disponível em: <<https://www.veripool.org/verilator/>>. Citado na página 71.
- SOARES, L. Seminary, *Simuladores e ferramentas educacionais no ensino de arquitetura de computadores*. São Paulo: [s.n.], 2015. Disponível em: <https://www.dcce.ibilce.unesp.br/~aleardo/cursos/arqcomp/Semin_EnsinoArq.pdf>. Citado na página 10.
- SPARC International Inc. *The SPARC architecture manual*. Version 9. Englewood Cliffs, NJ: Prentice Hall, 1994. ISBN 978-0-13-825001-0. Citado na página 20.
- STALLINGS, W. Reduced instruction set computer architecture. *Proceedings of the IEEE*, v. 76, n. 1, p. 38–55, jan. 1988. ISSN 1558-2256. Disponível em: <<https://ieeexplore.ieee.org/document/3287>>. Citado 6 vezes nas páginas 2, 16, 21, 34, 60 e 61.
- TANENBAUM, A. S.; AUSTIN, T. *Structured computer organization*. 6. ed. ed. Boston, Mass. Munich: Pearson, 2013. (Always learning). ISBN 978-0-13-291652-3. Citado 8 vezes nas páginas 2, 13, 14, 15, 16, 19, 20 e 35.
- VALADARES, D. *Didactic-RISC-I*. 2024. Disponível em: <<https://github.com/Diogo-Valadares/Didactic-RISC-I>>. Citado na página 31.
- VALADARES, D. *Uma Implementação do RISC I Utilizando o Simulador Logisim Evolution*. [S.l.], 2025. Disponível em: <https://github.com/Diogo-Valadares/Didactic-RISC-I/blob/main/Documentation/Technical%20Report%20RISC-I/Uma_Implementa%C3%A7%C3%A3o_do_RISC_I_utilizando_o_simulador_Logisim_Evolution.pdf>. Citado 4 vezes nas páginas 2, 4, 38 e 39.
- WILLIAMS, S. *Icarus Verilog*. 1998. Disponível em: <<https://bleyer.org/icarus/>>. Citado na página 29.

- WOLFFE, G. S. et al. Teaching computer organization/architecture with limited resources using simulators. *SIGCSE Bull.*, v. 34, n. 1, p. 176–180, fev. 2002. ISSN 0097-8418. Disponível em: <<https://doi.org/10.1145/563517.563408>>. Citado na página 11.
- YEHEZKEL, C. et al. Three simulator tools for teaching computer architecture: Little Man computer, and RTLSim. *J. Educ. Resour. Comput.*, v. 1, n. 4, p. 60–80, dez. 2001. ISSN 1531-4278. Disponível em: <<https://doi.org/10.1145/514144.514732>>. Citado na página 10.

Apêndices

APÊNDICE A – Arquitetura em SystemVerilog - Primeira Versão

B Teste de Entrada e Saída

```

1 .macro drawPoint videoAddress xReg yReg colorReg resolutionBits
2     yResolutionOffset addressBits
3     //masking the correct bits
4     andi xReg xReg resolutionBits
5     andi yReg yReg resolutionBits
6
7     slli yReg yReg yResolutionOffset
8
9     //generating the pixel address
10    or videoAddress videoAddress xReg
11    or videoAddress videoAddress yReg
12
13    //writing the pixel
14    sw colorReg videoAddress 0
15
16    //reset video address and yReg to its original value
17    and videoAddress videoAddress addressBits
18    srl yReg yReg yResolutionOffset
19 .endmacro
20
21 .macro beqi reg immediate position auxReg //branches if reg==immediate
22     li auxReg immediate
23     beq reg auxReg position
24 .endmacro
25
26 .macro setVector xReg yReg xVal yVal
27     li xReg xVal
28     li yReg yVal
29 .endmacro
30
31 .macro copyVector xRegTarget yRegTarget xRegSource yRegSource
32     mv xRegTarget xRegSource
33     mv yRegTarget yRegSource
34 .endmacro
35
36 .macro storeVector address xRegSource yRegSource
37     sh xRegSource address 0
38     sh yRegSource address 2
39 .endmacro
40
41 .macro addVector xRegR yRegR xReg1 yReg1 Xreg2 Yreg2
42     add xRegR xReg1 Xreg2
        add yRegR yReg1 Yreg2

```

```
43 .endmacro
44     nop
45     li sp 0x00010000
46     //video address 0x01000000 will be stored at the global pointer
47     //adicionally, the keyboard is the videoAddress-1, which should
        save resources.
48     li gp 0x01000000
49     li s2 0xffffffff //colorReg white
50     li s3 0xff000000 //address bits
51     li s4 1 //snake size
52     li s5 0 //current head offset
53     setVector s0 s1 0x1f 0x1f //setting starting position to center of
        the display
54
55     //first we set a loop that first test if user has set any
        input(loaderd to t1).
56     //if so we set the point speed, then move the position based on the
        speed.
57     //the loop ends once the input key is 'q'(t6)
58
59     setVector t2 t3 1 0 //setting initial speed
60
61 :loopStart
62     lb t1 gp -1
63     nop
64
65     blt t1 zero :draw //if there is no input the value will be negative
66
67 #REGION "switch"(input)
68     beqi t1 0x77 :w a0
69     beqi t1 0x61 :a a0
70     beqi t1 0x73 :s a0
71     beqi t1 0x64 :d a0
72     beqi t1 0x71 :loopEnd a0
73
74     :w
75     setVector t2 t3 0 -1
76     jump :draw
77     :a
78     setVector t2 t3 -1 0
79     jump :draw
80     :s
81     setVector t2 t3 0 1
82     jump :draw
83     :d
84     setVector t2 t3 1 0
```

```
85 #ENDREGION
86
87 :draw
88 copyVector a0 a1 s0 s1          //save old position
89
90 addVector s0 s1 s0 s1 t2 t3      //apply velocity to position
91
92 inc s2 0x010203 //update color
93
94 drawPoint gp s0 s1 s2 0x3f 6 s3 //draw head
95
96
97 drawPoint gp a0 a1 zero 0x3f 6 s3 //erase tail
98
99 jump :loopStart
100 :loopEnd
101
102 jump 0
```

Anexos