Implementação de Emulador Funcional de Sistema Computacional baseado em Arquitetura RISC-V

Gabriel Faustino de Freitas Silva[1], **João Pedro de Melo Roberto**[1], **Rodrigo Benfatti Aragão Leite**[1]

[1]Universidade Federal de Mato Grosso do Sul

g.faustino@ufms.br, joao_melo@ufms.br, rodrigo.benfatti@ufms.br

Abstract. *This work presents the development of a RISC-V architecture emulator in software. The RISC-V architecture, characterized by simplicity and modularity, was chosen for its relevance and openness to academic research. The emulator incorporates essential components such as Central Process Unit (CPU), memory and an Grafic Process Unit (GPU), validating its operation through tests and visual demonstrations. Among the demos, we highlight the rendering of the symbol monument of UFMS and the response to keyboard events for switching colors, evidencing the emulator's ability to handle interrupts and graphic. The emulator offers a flexible platform for architectural studies and experimentation in the RISC-V environment, demonstrating potential for future expansion.*

Resumo. *Este trabalho apresenta o desenvolvimento de um emulador de arquitetura RISC-V em software. A arquitetura RISC-V, caracterizada pela simplicidade e modularidade, foi escolhida pela sua relevância e abertura para pesquisa acadêmica. O emulador incorpora componentes essenciais como Unidade Central de Processamento (CPU), memória e uma Unidade de Processamento Gráfico (GPU), validando seu funcionamento por meio de testes e demonstrações visuais. Entre as demonstrações, destacam-se a renderização do monumento símbolo do UFMS e a resposta a eventos de teclado para troca de cores, comprovando a capacidade do emulador de manipular interrupções e gráficos. O emulador oferece uma plataforma flexível para estudos de arquitetura e experimentação no ambiente RISC-V, demonstrando potencial para futuras expansões.*

Contents

1. Introdução e História RISC-V

1.1. Introdução

Aqui, definimos como emulador o software que permite o funcionamento de um sistema de tal maneira como se estivesse utilizando um conjunto de hardware distinto daquele em que o sistema operacional está sendo aplicado. Trata-se de um software que foi programado para execução em um hardware com características diferentes e incompatíveis do hardware em que se deseja executá-lo[IBM Documentation 2023].

O objetivo deste trabalho é o projeto e a implementação de um emulador de um sistema computacional contendo os componentes de memória, barramento de comunicação, dispositivos de entrada e saída (placa gráfica e teclado) e processador. Dentre todas as opções de arquiteturas analisadas, decidiu-se pela arquitetura RISC, implementando o conjunto de instruções RISC-V.

1.2. História do RISC-V

O processador RISC-V ("risc five" em sua pronúncia na língua inglesa) é uma arquitetura de conjunto de instruções (ISA) aberta e de uso livre. Ela surgiu no ambiente universitário no ano de 2010, na UC Berkeley, pela equipe de pesquisadores da instituição, em especial pelo ilustre teórico David Patterson. A princípio, o objetivo da pesquisa era criar uma arquitetura simples, adaptável e de fácil aplicabilidade, com propósitos acadêmicos e educativos [Asanović and Patterson 2014, Hennessy and Patterson 2017].

Anteriormente ao RISC-V, a equipe trabalhou com outras versões desse mesmo processador, visando otimizar, a cada versão, o desempenho e a eficiência do conjunto de instruções dessa ferramenta, de tal modo que se tornou pioneira no desenvolvimento da plataforma RISC. Esta arquitetura recebeu atenção especial pela sua capacidade de atender demandas comerciais, principalmente para setores que exigem um elevado nível de personalização e eficiência.

Em síntese, o modelo RISC foi introduzido formalmente nos anos 1980 por Patterson, o qual observou que, ao simplificar o conjunto de instruções de um processador, era possível alcançar uma implementação mais eficiente e mais rápida. Em 1982, sob a orientação de Patterson, em conjunto com o professor Carlo Séquin e a equipe da UC Berkeley, um dos primeiros processadores RISC foi construído, tendo recebido a alcunha de RISC-I. Este possuía apenas 32 instruções e 78 mil transistores, auferindo notoriedade em razão de sua natureza descomplicada. Diante do resultado exitoso, outras arquiteturas foram desenvolvidas baseando-se nas características da arquitetura RISC proposta por Patterson, tais como o ARM e o MIPS. A arquitetura ARM, produzida pela empresa britânica Acorn Computers, é um dos maiores sucessos comerciais da arquitetura RISC, sendo aplicado primordialmente em dispositivos móveis. A arquitetura MIPS, por sua vez, elaborada pelo cientista John Hennessy, da Universidade de Stanford, se mantém em uso em inúmeros aparelhos e sistemas embarcados na atualidade. Outrossim, a arquitetura SPARC, fabricada pela empresa Sun Microsystems, se evidencia como outra tecnologia fundada na arquitetura RISC, utilizada principalmente em âmbito de servidores [Hennessy and Patterson 2014].

É inequívoco assinalar que a diferença substancial do RISC-V com relação às demais arquiteturas encontra-se no fato de sua ISA ser livre e aberta, enquanto a maioria

das outras arquiteturas, como ARM, x86 e MIPS, são arquiteturas proprietárias, as quais exigem licenciamento para serem utilizadas. Nesse passo, o RISC-V, ao ser livre de royalties, facilita a sua adoção, viabilizando inovações e customizações.

Em decorrência da possibilidade de seu livre uso e demais atribuições, surgiu ao redor da arquitetura um ecossistema cujo crescimento se observa de maneira gradativa, o qual conta com ferramentas de software e suporte, como compiladores, sistemas operacionais e ambientes de desenvolvimento [Hennessy and Patterson 2017]. Diante desse ecossistema, empresas diversas do ramo da tecnologia investem na arquitetura RISC-V, a exemplo disso, temos a multinacional e líder do mercado de placas gráficas, NVIDIA, utilizando-se do RISC-V para desenvolver o projeto FALCON [Sijstermans 2017].

O futuro do RISC-V é, portanto, promissor. A arquitetura RISC-V está se estabelecendo como uma alternativa viável às ISAs proprietárias, com potencial para impactar diferentes setores de infraestrutura tecnológica. A liberdade e flexibilidade proposta dispõem da capacidade de impulsionar cada vez mais inovações tecnológicas nos próximos anos.

2. Conceitos de Arquitetura de Computadores e RISC-V

2.1. Conceitos Básicos

A arquitetura de hardware é o conjunto de regras e métodos que descrevem, corretamente, a funcionalidade, organização e implementação de processamento de dados. A arquitetura define a forma como o hardware do computador é projetado, interligado e gerido para executar programas de software com eficiência. Diante dessa definição, alguns exemplos de componentes que estruturam a arquitetura de um sistema são a CPU, memória, barramento de dados e unidades de processamento paralelo, como a GPU.

A CPU é responsável por executar instruções e processar dados. A via de dados do processador, também chamada de microarquitetura, é variável, porém dentre os conceitos centrais, é incluso a técnica do *pipeline*, a qual divide a execução de uma instrução em múltiplas etapas, como busca, decodificação e execução, as quais permitem que diversas instruções sejam processadas simultaneamente.

Além do *pipeline*, é de conhecimento da arquitetura a unidade de controle e a unidade aritmética e lógica - ALU. A fim de diferenciar ambas, a primeira coordena as operações da CPU enquanto a ALU executa as operações matemáticas e lógicas fundamentais do processador [Hennessy and Patterson 2014].

A Arquitetura de Harvard Modificada define um caminho alternativo entre a memória de instrução e o processador central de forma que permita que as palavras na memória de instrução possam ser tratadas como dados somente de leitura, não apenas de instrução. Isso permite que os dados constantes, particularmente tabelas de dados ou textos, sejam alcançados sem ter que primeiramente ser copiado para a memória de dados, liberando assim mais memória de dados para variáveis de leitura/gravação [Fernandez 2015].

A memória mais conhecida por usuários comuns é a memória secundária, que abrange os dispositivos de armazenamento permanentes, como disco rígidos (*Hard Drive* - HD) e SSDs, os quais mantêm os dados quando o sistema não está ativo. Essa memória

possui maior capacidade de armazenamento de informações, porém a velocidade de acesso da CPU para esses dados é mais lenta.

Para além das memórias secundárias, a memória principal de um sistema computacional é a memória RAM, pois fornece acesso rápido e temporário aos dados usados ativamente pelos programas do computador. A característica principal dessa memória é sua volatilidade, uma vez que os dados armazenados nela são perdidos quando o sistema é desligado.

No entanto, por mais que a memória RAM seja uma memória de rápido acesso à CPU, existe a memória cache a qual é utilizada para armazenar justamente as instruções que serão processadas pela CPU a curto prazo, a fim de fornecer uma melhor eficiência na execução de processos. O cache é dividida em níveis, conhecidos como L1, L2 e L3.

Um conceito mais utilizado na arquitetura de computadores modernos é a incorporação de unidades de processamento paralelo, visando maximizar o desempenho em aplicações que exigem grande quantidade de cálculos simultâneos. Utilizando milhares de núcleos, as GPUs comerciais modernas otimizam a paralelização da execução de operações matemáticas, auxiliando o trabalho da CPU para esses cálculos. A aplicação mais comum para a GPU, além da parte visual, é para simulações matemáticas complexas e algoritmos que possuem alto nível de paralelismo, como os encontrados em aplicações de aprendizado de máquina.

Para que a CPU consiga se comunicar com todos os dispositivos exernos a ela, é necessário o uso de um barramento de comunicação. Os barramentos são sistemas de comunicação entre dispositivos distintos, permitindo a troca de dados entre eles. Comumente, o barramento é utilizado para comunicação entre CPU, memória e periféricos.

O método mais comum de comunicação entre a CPU e os dispositivos é a entrada/saída mapeada em memória (MMIO), que associa endereços físicos do barramento aos registradores, portas e regiões de memória dos dispositivos conectados. No MMIO, áreas específicas do espaço de endereçamento de memória são reservadas para os dispositivos de E/S. Entre as principais características do MMIO estão o compartilhamento do espaço de endereçamento entre a memória RAM e os dispositivos de E/S, a reserva de endereços específicos dentro desse espaço, e o fato de que, no MMIO, as operações de E/S são realizadas utilizando as mesmas instruções de *load* e *store* que o processador emprega para acessar a RAM [Song et al. 2016, Hennessy and Patterson 2014].

Outra forma de comunicação dos dispositivos com a CPU é através de interrupções. A depender da microarquitetura da CPU, diferentes tipos de interrupção podem existir (de hardware, de software, alta prioridade, baixa prioridade, entre outros). Além disso, diferentes microarquiteturas possuem diferentes métodos de como responder à uma interrupção. Uma interrupção de baixa prioridade, por exemplo, pode ser gerada por um clique de mouse ou com o pressionar de uma tecla.

2.2. Instruções e Ferramentas RISC-V

O RISC-V segue a arquitetura RISC, ou seja, é baseada em um conjunto de instruções simplificado em que cada instrução realiza uma operação básica e uniforme, facilitando a implementação de um hardware com execução eficiente delas pela CPU.

O conjunto de instruções RV32I, que é a base da arquitetura RISC-V para sistemas

de 32 bits e operações inteiras, define o mínimo necessário de operações que uma CPU compatível com RISC-V precisa implementar para funcionar. Esse conjunto opera com registradores e endereços de 32 bits que obtém a nomenclatura de x0 a x31. O registrador x0 tem valor fixado como 0, enquanto os demais registradores são utilizados para armazenar dados temporários, além de ponteiros e resultados de operações. A característica principal do RV32I é a base de operações com instruções inteiras, não incluindo operações de ponto flutuante. A simplicidade é um dos princípios desse conjunto de instruções, o que facilita a implementação de CPUs eficientes e de baixo custo.

A modularidade do RISC-V é um de seus grandes diferenciais em relação a outras arquiteturas RISC, pois permite que desenvolvedores escolham apenas os conjuntos de instruções necessários, o que otimiza tanto o desempenho quanto o consumo de energia e recursos. A arquitetura define um conjunto base de instruções que é essencial e obrigatório, no qual podem ser adicionadas várias extensões opcionais para atender as diferentes necessidades [Waterman et al. 2014, Waterman et al. 2015, Waterman 2016]. Algumas dessas várias extensões são:

a) A extensão “I” é a base do RISC-V para sistemas de 32 bits que suportam operações inteiras. Contém o núcleo das instruções necessárias para operações aritméticas, controle de fluxo, entre outras. É o ponto de partida para qualquer implementação da arquitetura;

b) A extensão “E” é uma variante da extensão base “I” com um conjunto de registradores reduzido, projetada especificamente para dispositivos embarcados de baixo custo, com apenas 16 registradores em vez de 32. O objetivo seria reduzir a área de chip e o consumo de energia;

c) A extensão “M” adiciona suporte para operações de multiplicação e divisão inteiras. Ela é crucial para aplicações que requerem processamento intensivo, como processamento digital de sinais;

d) A extensão “F” introduz operações de ponto flutuante de precisão simples (32 bits). Ela é utilizada em aplicações que envolvem cálculos de números precisos, como gráficos e simulações científicas;

f) A extensão “A” adiciona suporte para operações atômicas, fundamentais em ambientes multiprocessados e de concorrência. Operações de leitura-modificação-escrita precisam ocorrer sem interferência de outras instruções.

As extensões mencionadas são algumas das diversas existentes, o RISC-V se destaca das outras ISAs RISC por sua natureza modular, flexível e aberta, permitindo a criação de sistemas customizados e altamente otimizados.

As instruções RISC-V são organizadas em diferentes categorias, de acordo com as operações que realizam, sendo estas: instruções aritméticas e lógicas, instruções de carregamento e armazenamento - LOAD and STORE, instruções de controle de fluxo e instruções de imediatos. Tendo em vista que está sendo utilizado formatos binários de 32 bits, existem campos que indicam cada tipo de operação, registradores envolvidos, deslocamentos e valores imediatos, para tal, os formatos de cada uma dessas ações são nomeados como fomatos R, I, S, B, U e J [Hennessy and Patterson 2017].

2.2.1. Formato R

As instruções do tipo R operam com dois registradores-fonte, rs1 e rs2, e um registrador de destino - rd.

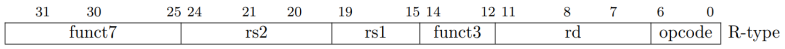

Figure 1. Campos do conjunto R-Type

2.2.2. Formato I

As instruções do formato I envolvem o rs1, rd e um valor imm(imediato).

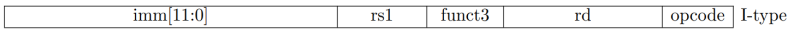

Figure 2. Campos do conjunto I-Type

2.2.3. Formato S

As instruções do Tipo S são utilizadas para operações de armazenamento.

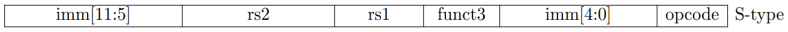

Figure 3. Campos do conjunto S-Type

2.2.4. Formato B

As instruções de desvio condicional do tipo B comparam dois registradores e desviam o fluxo de controle para um endereço baseado no valor imediato, caso a condição para isso seja verdadeira.

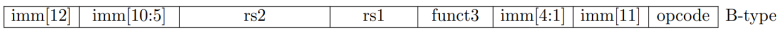

Figure 4. Campos do conjunto B-Type

2.2.5. Formato U

Para o formato U, o lui e auipc são instruções que envolvem a carga de valores imediatos de 20 bits em registrados.

Figure 5. Campos do conjunto U-Type

2.2.6. Formato J

As instruções de tipo J, como o jal, realizam o desvio incondicional para o endereço calculado pela CPU.

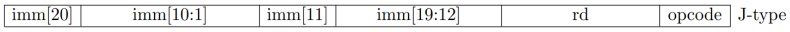

Figure 6. Campos do conjunto J-Type

2.2.7. Descrição das Instruções

As instruções aritméticas e lógicas possuem versões nos formatos R e I, como as instruções em assembly na Tabela1:

Table 1. Algumas instruções aritméticas e lógicas

Instrução	Sintaxe	Descrição
ADD	`add rd, rs1, rs2`	Soma os valores dos registradores `rs1` e `rs2` e armazena o resultado em `rd`.
SUB	`sub rd, rs1, rs2`	Subtrai o valor do registrador `rs2` do valor de `rs1` e armazena o resultado em `rd`.
AND	`and rd, rs1, rs2`	Executa a operação lógica "E" entre `rs1` e `rs2`, armazenando o resultado em `rd`.
OR	`or rd, rs1, rs2`	Executa a operação lógica "OU" entre `rs1` e `rs2`, armazenando o resultado em `rd`.
SLL	`sll rd, rs1, rs2`	Desloca os bits do valor em `rs1` para a esquerda pela quantidade de posições especificadas por `rs2`, armazenando o resultado em `rd`.
ADDI	`addi rd, rs1, imm`	Soma o valor do registrador `rs1` com o valor imediato `imm`, armazenando o resultado em `rd`.

Na arquitetura RISC-V, o processador utiliza o modelo load/store para acessar e manipular dados na memória. O comando *load* carrega as informações da memória para os registradores, enquanto o comando *store* salva dados dos registradores nos endereços de memória. No conjunto de instruções RV32I, essas operações usam os fomatos I e S, na Tabela2:

Table 2. Algumas instruções Load and Store

Instrução	Sintaxe	Descrição
LW	`lw rd, offset(rs1)`	Carrega uma palavra (32 bits) da memória, cujo endereço é o valor de `rs1` somado ao deslocamento `offset` e armazena o resultado em `rd`.
SW	`sw rs2, offset(rs1)`	Armazena o valor de `rs2` na memória no endereço calculado como o valor de `rs1` somado ao deslocamento `offset`.

Outro grupo de instruções existente é o de controle de fluxo, ou *Branch/Jump*, permitindo o desvio da execução para um endereço específico, dependendo da condição. A seguir, as instruções foram especificadas na Tabela3:

Table 3. Algumas instruções de Controle de Fluxo

Instrução	Sintaxe	Descrição
BEQ	`beq rs1, rs2, offset`	Se o valor de `rs1` for igual ao valor de `rs2`, o programa desvia para o endereço de memória calculado como `PC + offset`.
BNE	`bne rs1, rs2, offset`	Se o valor de `rs1` for diferente de `rs2`, o programa desvia para `PC + offset`.
JAL	`jal rd, offset`	Salta para o endereço `PC + offset` e armazena o endereço do próximo `PC` em `rd`. Utilizado para chamar funções.
JALR	`jalr rd, offset(rs1)`	Salta para o endereço calculado como `rs1 + offset` e armazena o valor do próximo `PC` em `rd`.

Por fim, o grupo de instruções de imediatos são variações das instruções aritméticas, porém utiliza-se os valores imediatos, constantes, para realizar os cálculos. As instruções estão especificadas na Tabela4:

Table 4. Algumas instruções de imediatos

Instrução	Sintaxe	Descrição
LUI	`lui rd, imm`	Carrega os 20 bits superiores do imediato `imm` no registrador `rd` e zera os 12 bits inferiores. Usado para construir endereços.
AUIPC	`auipc rd, imm`	Soma o imediato `imm` ao valor do contador de programa (`PC`) e armazena o resultado em `rd`. Usado para construção de endereços relativos.

3. Emulador RISC-V

Com base nos conceitos minuciados nos capítulos anteriores acerca da arquitetura de computadores, sobretudo relacionados à arquitetura RISC-V, esta seção detalhará o desenvolvimento do emulador que foi projetado para simular a execução de programas. O objetivo principal será descrever as etapas de implementação do emulador, a fim de abordar as escolhas técnicas, tais como as principais funcionalidades.

O emulador projetado tem como propósito oferecer um local que permita a execução de instruções RISC-V em um ambiente controlado, servindo para fins didáticos e, também, para demonstrar o potencial computacional dessa arquitetura em diferentes cenários e configurações do processador.

Em primeiro lugar, é fundamental abordar o ambiente de desenvolvimento e as ferramentas utilizadas para a construção do emulador. A linguagem de programação selecionada foi o Java, uma linguagem orientada a objetos escolhida devido ao seu amplo ecossistema e à disponibilidade de bibliotecas bem documentadas e atualizadas, o que facilitou a implementação de diferentes componentes do projeto. Além disso, o conhecimento prévio dos membros da equipe sobre essa linguagem contribuiu significativamente para uma curva de aprendizado mais rápida e eficiente, otimizando o desenvolvimento do projeto.

A fim de facilitar o trabalho de compartilhamento e colaboração, a IDE escolhida foi o IntelliJ, desenvolvido e distruibuido pela empresa JetBrains. Ela possui ferramentas importantes e descomplicadas, como o versionamento através do GIT. Este, é um sistema de controle de versão, ou VCS, que monitora o histórico de alterações à medida que as pessoas colaboram com o projeto [Scott Chacon and Ben Straub 2024]. Como possui versionamento, é possível recuperar versões anteriores do projeto a qualquer momento. Para utilizar dessa ferramenta, foi usado o GITHUB, uma hospedagem de repositórios GIT que fornece aos desenvolvedores ferramentas diretamente da IDE.

Por fim, uma vez que se detalhou as ferramentas utilizadas, a seção a seguir explorará a estrutura interna do emulador, explicando como foi realizada a divisão das pastas, as suas principais funcionalidades e como as instruções são executadas e monitoradas.

A fim de melhor visualizar a interação entre os módulos, a figura 7 mostra o diagrama de classes entre eles:

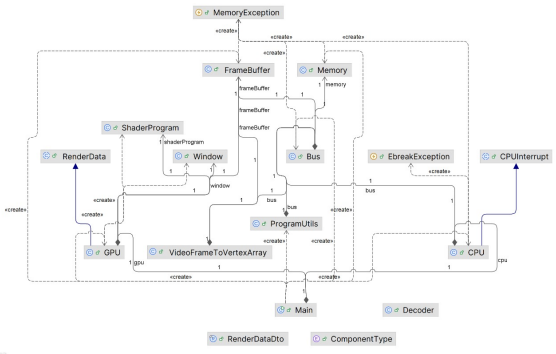

Figure 7. Diagrama de Classes do Emulador

3.1. Processador

A arquitetura de processadores é uma área central da Engenharia de Computação, em especial a implementação de CPUs capazes de executar instruções eficientemente e precisas. O emulador proposto simula uma CPU simplificada, implementada em linguagem Java, atingindo funcionalidades essenciais da arquitetura RISC, como manipulação de registradores, acesso à memória, tratamento de interrupções e execução de diversas instruções.

A CPU simulada contém 32 registradores de uso geral, além de registradores de Controle e Status (CSR) para gerenciar interrupções e exceções. O objetivo principal deste trabalho é explorar os desafios e soluções associados à simulação de uma CPU operante similarmente a arquiteturas reais, a fim de proporcionar uma visão prática sobre a construção de processadores e gerenciamento eficiente de recursos computacionais.

Diante disso, neste capítulo será detalhada a implementação das classes necessárias para o desenvolvimento da CPU do emulador. A figura 8 exibe o diagrama entre os métodos.

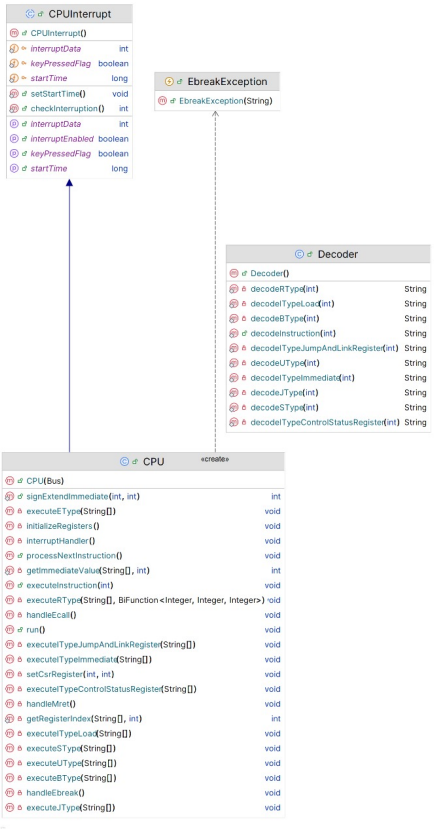

Figure 8. Diagrama de Classes da CPU

3.1.1. Classe `Decoder`

A classe Decoder é responsável pela decodificação de instruções de uma CPU. A classe utiliza a arquitetura RISC-V como base para realizar essa decodificação, interpretando diversos tipos de instruções, sendo elas os tipos R, I, S, B, U e J.

O método principal da classe Decoder é o decodeInstruction. A função dele é receber a instrução de 32 bits, e, com base nos opcode, determina qual tipo de instrução deve ser decodificada.

Listing 1. Método decodeInstruction()

```
1 public static String decodeInstruction(int instruction);
```

Para a decodificação, existem três campos significativos e utilizados na classe decoder, sendo estes os campos OPCODE, Funct3 e Funct7.

O campo OPCODE ocupa os 7 primeiros bits (menos significativos) de uma instrução de 32 bits. A principal função é identificar o tipo geral de operação que o processador deve realizar. Em outras palavras, informa ao processador a categoria da instrução, se é uma operação de cálculo aritmético, uma operação lógica, uma transferência de dados entre memória e registradores ou uma operação de controle de fluxo.

Os diferentes valores do opcode mapeiam para tipos distintos de instrução. Na Tabela5, temos:

Table 5. Tipos de Instruções do Processador

Código Hexadecimal	Tipo de Instrução	Descrição
0x33	Tipo R	Instrução de operações aritméticas e lógicas com registradores.
0x03	Tipo I	Instrução que envolve imediatos ou carga de memória.

Na classe Decoder, o opcode é extraído da instrução da seguinte forma:

Listing 2. Definição do opcode

```
int opcode = instruction & 0x7F; // Extrai os 7 bits menos
    significativos
```

Cada valor de opcode define um grupo de operações e, dependendo do tipo da instrução, os demais campos de instruções, funct3 e funct7, ajudam a definir a operação exata.

Para além do OPCODE, o funct3 é um campo de 3 bits localizado nos bits 12 a 14 da instrução de 32 bits. Este campo é usado para diferenciar operações específicas dentro de uma mesma categoria de instruções, como aritméticas, lógicas ou de controle.

Para uma instrução do tipo R, cujas operações dependem de registradores, o funct3 diferencia operações como adição, subtração, AND, OR, entre outras. Na Tabela6, em uma instrução com opcode = 0x33, R-Type, o campo funct3 pode ter os seguintes valores:

Table 6. Exemplo de instruções baseadas no campo funct3

funct3	Operação	Descrição
000	Adição (add) ou Subtração (sub)	Executa adição quando `funct7` = 0000000 e subtração quando `funct7` = 0100000. Ambas operações são realizadas com registradores.
111	AND	Realiza a operação lógica E (AND) bit a bit entre dois registradores.
110	OR	Realiza a operação lógica OU (OR) bit a bit entre dois registradores.
001	Shift lógico à esquerda (SLL)	Desloca os bits de um registrador à esquerda logicamente, preenchendo com zeros à direita.

Assim, na implementação do funct3, é declarado como:

Listing 3. Definição do funct3

```
int funct3 = (instruction >> 12) & 0x7; // Extrai o funct3 dos
   bits 12 a 14
```

Por fim, o funct7 é um campo de 7 bits localizado nos bits 25 a 31 da instrução. A principal função é especificar variações de uma operação a qual foi parcialmente identificada pelos campos opcode e funct3. O campo funct7 é frequentemente utilizado para diferenciar variantes de uma mesma operação. A exemplo disso, para o opcode 0x33, R-Type, e funct3 é 000, o campo funct7 diferencia entre:

a) funct7 = 0000000: Adição (add);

b) funct7 = 0100000: Subtração (sub).

O campo funct7 permite que mais operações sejam expresas com o mesmo opcode e funct3, a fim de garantir uma maior variedade sem aumentar o número de opcodes.

Diante disso, a configuração do funct7 na classe é feita da seguinte maneira:

Listing 4. Definição do funct7

```
int funct7 = (instruction >> 25) & 0x7F; // Extrai o funct7 dos
   bits 25 a 31
```

A depender do opcode da instrução, os métodos decodeRType, decodeIType, decodeSType, decodeBType, decodeUType e decodeJType são chamados. Cada método utiliza operações bit a bit para extrair os campos apropriados da instrução de 32 bits.

Em suma, a classe Decoder é construída de forma modular, com um método principal que identifica o tipo de instrução baseado no opcode e chama o método auxiliar adequado para decodificá-la. Essa construção de classe facilita a extensão para suportar novos tipos de instruções ou arquiteturas.

Abaixo, está especificado como cada método é instanciado dentro da classe Decoder:

Listing 5. Método decodeRType()

```
private static String decodeRType(int instruction);
```

Listing 6. Método decodeITypeLoad()

```
1 private static String decodeITypeLoad(int instruction);
```

Listing 7. Método decodeITypeImmediate()

```
1 private static String decodeITypeImmediate(int instruction);
```

Listing 8. Método decodeITypeControlStatusRegister()

```
1 private static String  private static String
     decodeITypeControlStatusRegister(int instruction);
```

Listing 9. Método decodeSType()

```
1 private static String decodeSType(int instruction);
```

Listing 10. Método decodeBType()

```
1  private static String decodeBType(int instruction);
```

Listing 11. Método decodeUType()

```
1  private static String decodeUType(int instruction);
```

Listing 12. Método decodeJType()

```
1   private static String decodeJType(int instruction)
```

3.1.2. Classe `CPUInterrupt`

A classe CPUInterrupt é uma classe abstrata que trata a interrupção da CPU em um programa. A estrutura da classe CPUInterrupt é desenhada a fim de gerenciar interrupções de maneira eficiente, separando a lógica de interrupção baseado em tempo e eventos, este é, por exemplo, uma tecla pressionada. Valendo-se de váriaveis estáticas para gerenciar o estado global da interrupção, garante que qualquer modo de interrupção seja tratado uniformemente dentro do sistema. As subclasses da CPUInterrupt podem estender essa classe para implementar comportamentos específicos de interrupção.

Para o funcionamento da classe, o método CheckInterruption() foi implementado a fim de verificar se há alguma interrupção a ser tratada, retornando um código númerico diferente dependendo do tipo de interrupção. O método calcula o tempo decorrido desde que a interrupção foi ativada, utilizando o atributo startTime e a função System.currentTimeMillis().

O *timer* é desenvolvido considerando se tempo decorrido é maior ou igual ao valor do clock obtido do método Main.getClockSpeed(). A depender do valor, uma interrupção do *timer* ocorre, desabilitando a interrupção e reiniciando o tempo.

Para verificar se existe alguma tecla pressionada, a flag *keyPressedFlag* permanece ativa, caso haja uma interrupção, é desativada e a tecla é processada, retornando o valor *interruptData* + 2.

Listing 13. Método checkInterruption()

```
1    public static int checkInterruption();
```

Outro método implementado na classe é o setStartTime, este inicializa o atributo startTime com o tempo atual do sistema. Esse método é chamado, geralmente, quando a interrupção é habilitada para iniciar a contagem.

Listing 14. Método setStartTime()

```
1    public static void setStartTime();
```

3.1.3. Classe `CPU`

A classe CPU simula o funcionamento de uma CPU RISC-V ao implementar a execução de instruções, leitura e escrita na memória, além de tratar interrupções. As instruções são decodificadas e executadas por meio de métodos específicos que seguem a arquitetura proposta. Além disso, gerencia interrupções, salvando o estado e ajustando o programCounter para a posição da rotina de interrupção.

A classe CPU estende CPUInterrupt, ou seja, herda a responsabilidade pelo tratamento de interrupções do sistema. Os atributos da classe estão minuciados na Tabela 7.

Table 7. Atributos e Registradores Utilizados

Atributo	Descrição
registers[]	Array de 32 registradores de propósito geral.
CSRRegisters[]	Array de 4096 registradores CSR.
programCounter	Aponta para a próxima instrução a ser executada.
MIE, MTVEC, MEPC, MCAUSE, MTVAL, MIP	Registradores da arquitetura RISC-V para controle de interrupções e status.

O construtor da classe inicializa os registradores com valores padrão e conecta a CPU a um barramento (Bus) para acessar a memória e periféricos. A implementação do construtor está exemplificada abaixo:

Listing 15. Construtor da Classe CPU

```
1    public CPU(final Bus bus);
```

O construtor da CPU depende do método initializeRegisters, o qual inicializa alguns registradores da CPU com valores padrão. A exemplo disso, temos:

a) Registrador 2 - Stack Pointer: Definido para o topo da memória (0x3E8000);

b) Registrador 3 - Global Pointer: Definido para o meio da memória (0x1F4000);

c) Registrador 8 - Frame Pointer: É utilizado com o mesmo propósito do registrador 2.

O método initializeRegisters() é implementado da seguinte maneira:

Listing 16. Método initializeRegisters()

```
1     private void initializeRegisters();
```

O método principal da classe CPU é o run. Este é executado continuamente até ser interrompido. Ela convoca o método processNextInstruction() para buscar e executar instruções na memória, sendo implementado como:

Listing 17. Método run()

```
1     public void run();
```

A seguir, o método processNextInstruction busca na memória a próxima instrução e decide se alguma interrupção necessita de tratamento. Caso alguma interrupção esteja habilitada, é convocado o método interruptHandler(), caso contrário, lê a próxima instrução da memória e envia para execução. Esse método é instânciado dessa forma:

Listing 18. Método processNextInstruction()

```
1     public void processNextInstruction();
```

Para além desse, o método executeInstruction decodifica e executa a instrução. A depender da operação, um método específico para cada tipo de instrução é chamado da classe Decoder para esse trabalho. Caso a operação seja desconhecida, uma exceção é disparada. Abaixo, segue sua instanciação:

Listing 19. Método executeInstruction()

```
1     public void executeInstruction(int instruction) throws
          MemoryException {...}
```

Caso seja necessário lidar com interrupções, o método interruptHandler é convocado. Esse método salva o estado do PC no atributo MEPC e define o novo valor do PC para a rotina de interrupção.

Listing 20. Método interruptHandler()

```
1     private void interruptHandler();
```

Além disso, a depender da especificidade da interrupção, existem três métodos possíveis para utilização. Caso seja uma chama de sistema, ecall, utiliza-se o método handleEcall(). Para pontos de interrupção, ebreak, convoca-se o handleEbreak() e, por fim, para retornos de interrupção, mret, é chamado o método handleMret().

Listing 21. Método handleEcall()

```
1     private void handleEcall();
```

Listing 22. Método handleEbreak()

```
1     private void handleEbreak();
```

Listing 23. Método handleMret()

```
1     private void handleMret();
```

A partir do princípio que a classe CPU é a classe mãe na implementação da CPU RISC-V, existem funções utilitárias para auxiliar no processamento. Elas são:

a) getRegisterIndex: Essa função extrai o índice de um registrador a partir de uma instrução decodificada;

Listing 24. Função getRegisterIndex()

```
1       private static int getRegisterIndex(String[] parts, int
           partIndex);
```

b) getImmediateValue: Extrai um valor imediato, imm, de uma instrução;

Listing 25. Função getImmediateValue()

```
1    private static int getImmediateValue(String[] parts, int
        partIndex);
```

c) signExtendImmediate: Realiza a extensão de sinal de um valor imediato a fim de garantir a correta interpretação de bits negativos.

Listing 26. Função signExtendImmediate()

```
1   public static int signExtendImmediate(int immediate, int
       bitWidth);
```

As classes criadas para isso são herdadas pela CPU a fim de manter o funcionamento adequado das funções.

3.2. Memória

A implementação de um sistema de memória eficiente é crucial no desenvolvimento de um emulador, haja vista que a fidelidade ao comportamento do hardware original e o desempenho são primordiais. No contexto do emulador, a memória virtualizada necessita replicar o funcionamento do hardware real de maneira precisa, oferencendo uma base para o armazenamento e a recuperação de dados durante a execução de instruções.

A arquitetura da memória de um emulador envolve diversos aspectos, entre eles, os diferentes tipos de memória, a simulação de mapeamentos de memória e a manipulação correta dos acessos de leitura e escrita. Para garantir a integridade dos dados, o emulador deve lidar com exceções de memória, como acessos fora do intervalo permitido, o *overflow*, e tentativas de escrita em áreas de memória de somente leitura. Tais erros necessitam ser interceptados e tratados de maneira adequada, a fim de evitar comportamentos inesperados durante a emulação.

Em suma, o desenvolvimento de um subsistema de memória no emulador é a espinha dorsal para a execução precisa e eficiente dos programas simulados. Além disso, a escolha de algoritmos para gerenciamento de memória impacta diretamente na performance e na precisão do emulador. Nesse contexto, as classes e métodos implementadas seguem no objetivo de cumprir com tais requisitos.

Na figura 9, o diagrama de classe do módulo da memória é exibido.

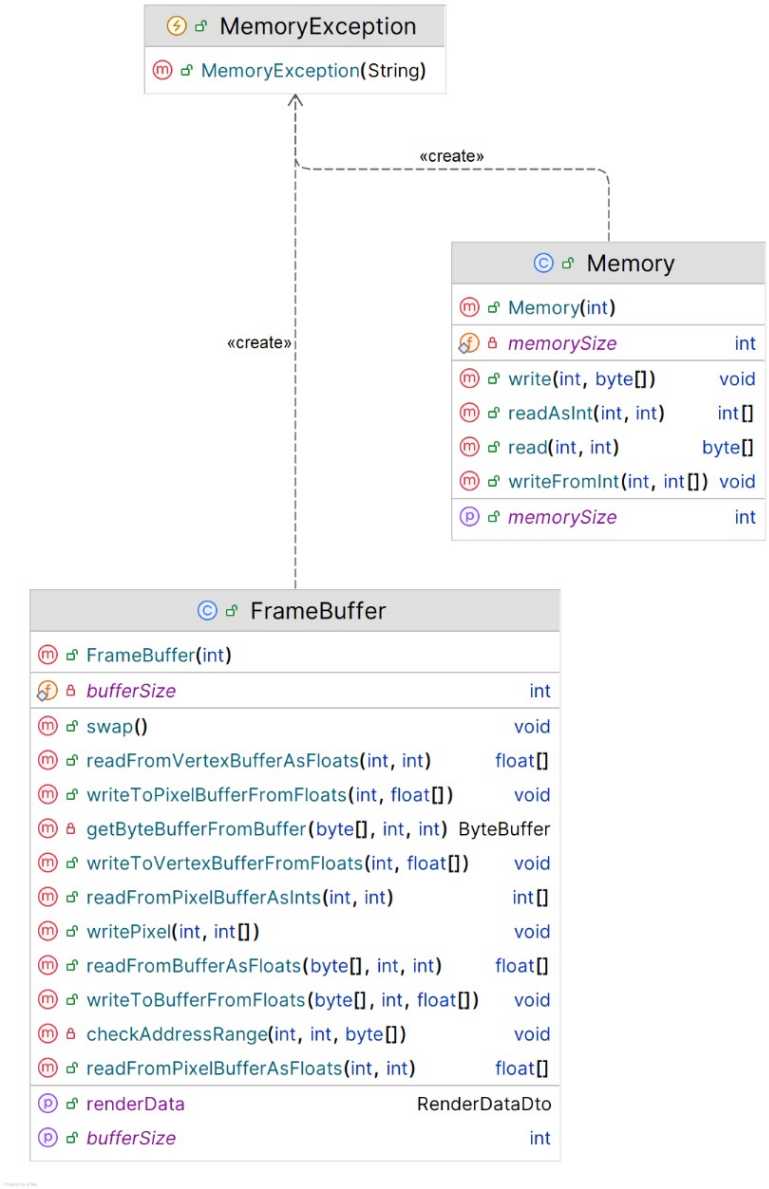

Figure 9. Diagrama de Classes da Memória

3.2.1. Classe `FrameBuffer`

Para permitir que a GPU renderize gráficos no sistema enquanto a CPU emulada opera com uma interface de pixels em 2D, o emulador utiliza uma abordagem de buffer duplo através da classe FrameBuffer. Esta classe gerencia dois buffers principais para renderizar a saída visual, *front* e *back* buffer, permitindo que a GPU desenhe gráficos com uma atualização fluída. A renderização é realizada na interface de hospedagem, a GPU processa as informações do frontBuffer, usando as APIs do OpenGL, para manipular os dados gráficos diretamente.

A arquitetura emulada, portanto, permite um equilíbrio entre a flexibilidade do OpenGL para renderização e a simplicidade da manipulação direta de pixels, proporcio-

nando um ambiente gráfico realista para o software que está sendo emulado. Para isso, a classe trabalha com dois tipos de buffers, explicados na Tabela 8:

Table 8. Tipos de Buffer Utilizados no Emulador

Tipo de Buffer	Descrição
Buffer de vértices	Armazena informações sobre as posições dos vértices no espaço 3D.
Buffer de pixels	Responsável pelo armazenamento das cores dos pixels.

Na Tabela 9, elencaremos os atributos da classe:

Table 9. Buffers Utilizados na Classe

Tipo de Buffer	Descrição
bufferSize	Valor estático, compartilhado por todas as instâncias da classe, serve para definir o tamanho de cada buffer.
frontPixelBuffer	Armazena os dados de pixel que estão sendo exibidos atualmente.
backPixelBuffer	Armazena os dados de pixels para a próxima renderização.
frontVertexBuffer	É o buffer de vértices atualmente em uso.
backVertexBuffer	É o buffer de vértices para a próxima atualização.

A classe FrameBuffer possui um construtor que inicializa os buffers de vértices e de pixels com o tamanho especificado. O *size* do framebuffer é multiplicado por 8, a fim de suportar múltiplos bytes por vértice/pixel. Portanto, o tamanho do buffer, *bufferSize*, é definido com o dobro do valor fornecido.

Listing 27. Construtor da classe FrameBuffer

```
1     public FrameBuffer(final int bufferSize);
```

Para além do construtor, existem os métodos da classe FrameBuffer. Eles são:

A) writePixel: Escreve os dados de pixels, representados como inteiros, no backPixelBuffer. Este método calcula as coordenadas normalizadas dos pixels e transforma as cores de cada pixel em valores de ponto flutuante para uso em operações gráficas. Para exemplificar melhor, o fluxo do método é transformar cada cor (inteiro) em componentes RGB e as normaliza. Após isso, escreve os dados de cores no backPixelBuffer e os vértices correspondentes no backVertexBuffer;

Listing 28. Método writePixel()

```
1   public void writePixel(int beginAddress, final int[] data)
       throws MemoryException {...}
```

B) writeToPixelBufferFromFloats: Converte um array de float para bytes e escreve os dados no backPixelBuffer;

Listing 29. Método writeToPixelBufferFromFloats()

```
1  public void writeToPixelBufferFromFloats(final int beginAddress
      , final float[] data) throws MemoryException {...}
```

C) writeToVertexBufferFromFloats: Similiarmente ao anterior, converte um array de float para bytes e escreve no backVertexBuffer;

Listing 30. Método writeToVertexBufferFromFloats()

```
public void writeToVertexBufferFromFloats(final int
    beginAddress, final float[] data) throws MemoryException
    {...}
```

D) writeToBufferFromFloats: É um método auxiliar usado pelos métodos acima para converter arrays float para bytes e escrever os dados em qualquer buffer. Esse método utiliza a função do java ByteBuffer para fazer a conversão de tipos;

Listing 31. Método writeToBufferFromFloats()

```
public void writeToBufferFromFloats(final byte[] buffer, final
    int beginAddress, final float[] data) throws MemoryException
     {...}
```

E) checkAddressRange: Verifica se o intervalo de endereços para escrita é valido. O método dispara uma exceção caso os limites sejam excedidos;

Listing 32. Método checkAddressRange()

```
private void checkAddressRange(int beginAddress, int data, byte
    [] backBuffer) throws MemoryException {...}
```

F) swap(): Realiza a troca entre os buffers frontais e traseiros de vértices e pixels, promovendo o backBuffer para o frontBuffer, para exibição;

Listing 33. Método swap()

```
   public void swap();
```

G) getRenderData: Retorna os dados de renderização a partir dos frontBuffers. Os dados são encapsulados em um objeto, *RenderDataDto*, que inclui tanto dados de vértices quanto de pixels;

Listing 34. Método getRenderData()

```
public RenderDataDto getRenderData() throws MemoryException
    {...}
```

H) readFromPixelBufferAsFloats: Lê uma seção do frontPixelBuffer e retorna um array de floats, representando os dados armazenados;

Listing 35. Método readFromPixelBufferAsFloats()

```
  public float[] readFromPixelBufferAsFloats(final int
     beginAddress, final int endAddress) throws MemoryException
     {...}
```

I) readFromVertexBufferAsFloats: Lê uma seção do frontVertexBuffer e retorna um array de floats;

Listing 36. Método readFromVertexBufferAsFloats()

```
1  public float[] readFromVertexBufferAsFloats(final int
       beginAddress, final int endAddress) throws MemoryException
       {...]
```

J) readFromBufferAsFloats: Função auxiliar que lê qualquer buffer, seja de vértices ou pixels, e retorna os dados em formato de float;

Listing 37. Método readFromBufferAsFloats()

```
1  public float[] readFromBufferAsFloats(final byte[] buffer, final
       int beginAddress, final int endAddress) throws
     MemoryException {...}
```

K) readFromPixelBufferAsInts: Lê uma seção do frontPixelBuffer e retorna um array de inteiros;

Listing 38. Método readFromPixelBufferAsInts()

```
1  public int[] readFromPixelBufferAsInts(final int beginAddress,
     final int endAddress) throws MemoryException {...}
```

L) getByteBufferFromBuffer: Retorna um ByteBuffer posicionado em um intervalo específico do buffer dado. Esta função é usada para ler ou modificar dados no buffer de bytes.

Listing 39. Método getByteBufferFromBuffer()

```
1  private ByteBuffer getByteBufferFromBuffer(final byte[] buffer,
       final int beginAddress, final int endAddress) throws
     MemoryException {...}
```

3.2.2. Classe `MemoryException`

A classe MemoryException é uma exceção personalizada, utilizada para tratar erros relacionados a operações de memória na aplicação. Ela estende a classe RuntimeException, que é uma exceção não verificada no Java.

Essa classe possui um único construtor, o qual permite que uma mensagem de erro seja passada quando a exceção é lançada. Esse construtor possui o parâmetro message, esse sendo uma descrição detalhada do motivo pela qual a exceção foi disparada. Além disso, o método super(message) invoca o construtor da classe RunTimeException, o qual armazena essa mensagem e a torna acessível posteriormente por métodos como getMessage(). Segue abaixo a forma em que o construtor é implementado:

Listing 40. Construtor da Classe MemoryException()

```
1  public MemoryException(String message) {
2  super(message); // Call superclass constructor with the provided
       message
3      }
```

Em suma, essa classe é utilizada em operações de leitura e escrita de memória para sinalizar erros. Ao invés de utilizar exceções genéricas como ArrayIndexOutOfBoundsException, esse método fornece mais clareza e contexto sobre o tipo de erro específico ocorrido na memória.

3.2.3. Classe `Memory`

A classe Memory representa um modelo de memória simplificado para armazenar e recuperar dados. É semelhante ao funcionamento de um módulo de memória em hardware, onde a memória é representada por um array de bytes. Essa classe possui dois atributos importantes, sendo eles exibidos na Tabela 10.

Table 10. Atriutos de Memória Utilizados na Classe

Atributos	Descrição
memorySize	Um inteiro que armazena o tamanho total da memória. Este valor é acessível de forma pública através do método getter.
memory[]	Um array de bytes que efetivamente representa a memória. Todos os dados são armazenados neste array.

Na classe Memory, o construtor inicializa o array de memória com o tamanho especificado. Define o valor do atributo memorySize como o tamanho total da memória alocada. Abaixo, segue sua implementação:

Listing 41. Construtor da Classe Memory()

```
1  public Memory(final int memorySize) {
2     this.memory = new byte[memorySize]; // Allocate memory
3         Memory.memorySize = memorySize;
4     }
```

Os métodos pertencentes à ela são:

A) write: Este método grava um array de bytes na memória, inicializando da posição especificada, beginDataPosition. O método itera sobre o array de value e grava byte por byte na memória. Além disso, verifica se a gravação ultrapassa o tamanho da memória e, caso positivo, dispara uma exceção;

Listing 42. Método write()

```
1  write(int beginDataPosition, byte[] value);
2      }
```

B) writeFromInt: Escreve um array de inteiros na memória, convertendo cada inteiro em quatro bytes antes de realizar a escrita. Esse método utiliza o ByteBuffer para converter os inteiros em bytes e escreve na memória de maneira similar ao método anterior. Além disso, verifica se a posição de escrita excede o tamanho da memória;

Listing 43. Método writeFromInt()

```
1  writeFromInt(int beginDataPosition, int[] value);
2      }
```

D) read: O método lê uma faixa de bytes na memória entre duas posições, beginDataposition e endDataposition, e retorna um array de bytes. Ou seja, verifica se as posições de leitura são válidas e copia os bytes correspondentes da memória e retorna o array de bytes;

Listing 44. Método read()

```
read(int beginDataPosition, int endDataPosition);
    }
```

E) readAsInt: O método analisa uma faixa de bytes da memória e converte-os em um array de inteiros. É similar ao método read, porém converte os bytes lidos em inteiros de 32 bits. Além disso, utiliza o ByteBuffer para realizar a conversão de bytes para inteiros.

Listing 45. Método readAsInt()

```
readAsInt(int beginDataPosition, int endDataPosition);
    }
```

A classe Memory utiliza a classe MemoryException para realizar os disparos de exceções quando ocorre algum erro.

3.3. Barramento

Durante a implementação do emulador, o barramento, *bus*, desempenha um papel crucial ao simular a comunicação entre diferentes componentes de hardware, como memória e dispositivos de entrada/saída. No contexto do projeto, a implementação de um barramento no emulador é fundamental para garantir que os dados possam ser transmitidos de forma eficaz e precisa entre os módulos.

O barramento é responsável por gerenciar a leitura e escrita de dados, coordenando a transferência de informações entre a memória e os demais componentes do emulador, como o FrameBuffer. Através do bus, os endereços são mapeados e direcionados ao destino correto utilizando MMIO. Além disso, abstrair a complexidade da comunicação é mais uma de suas responsabilidades. O código do barramento não precisa se preocupar em manipular diretamente os diferentes módulos, haja vista que ele realiza a interface entre eles de forma transparente. Por fim, outro de seus compromissos é garantir a integridade dos dados e além de roteá-los, o barramento verifica se os endereços estão corretos, disparando exceções em casos de endereços inválidos ou operações fora dos limites de memória. Portanto, ao implementar esse mecanismo no emulador, é possível criar um ambiente mais realista e modular.

3.3.1. Classe `Bus`

A classe Bus é responsável justamente por gerenciar a comunicação entre os dois componentes principais - Memory e FrameBuffer. O barramento atua como intermediário, determinando a qual componente as operações de leitura e escrita se referem, de acordo com os endereços fornecidos. Para que tal função seja cumprida, os atributos da classe Bus, evidenciados na Tabela 11, são baseadas nas classes Memory e Framebuffer.

Table 11. Buffers e Atributos de Memória Utilizados na Classe

Atributos	Descrição
frameBuffer	Armazena uma referência ao componente FrameBuffer, responsável pelo armazenamento de dados de imagem e renderização.
frameBufferSize	Armazena o tamanho do FrameBuffer, obtido estaticamente através do método FrameBuffer.getBufferSize().
memory	Semelhante ao frameBuffer, armazena uma referência ao componente Memory, sendo responsável pelo armazenamento e gerenciamento de dados gerais durante a execução.
memorySize	O atributo armazena o tamanho da memória, obtido estaticamente com Memory.getMemorySize().

O construtor do barramento inicializa os componente framebuffer e memory com base nos paramêtros recebidos. Ele cria uma instância e prepara os buffers para comunicação.

Listing 46. Construtor Bus()

```
1 public Bus(final FrameBuffer frameBuffer, final Memory memory) {
2         this.frameBuffer = frameBuffer;
3         this.memory = memory;
4     }
```

Os métodos da classe Bus são destinadas à comparação de componentes para saber se a escrita da instrução será realizada na memória da CPU ou da GPU. Esses, são:

A) whichComponentType: Determina qual componente deve ser utilizado com base no endereço de memória recebido. Se o endereço estiver dentro do intervalo da memória, retorna MEMORY, caso contrário retonra FRAME_BUffer. Se o endereço for inválido lança um MemoryException;

Listing 47. Método whichComponentType()

```
1  public ComponentType whichComponentType(final int address);
```

B) write: Este método gerencia a escrita de dados, a partir da determinação de qual componente deve ser feita a escrita, utilizando o método anterior para isso. Sendo o FrameBuffer, calcula o endereço correto e, dependendo do valor, realiza um swap dos buffers ou escreve dados. No entanto, sendo Memory, chama o método writeFromInt() para gravar os dados no endereço da memória;

Listing 48. Método write()

```
1   public void write(final int address, final int[] value);
```

C) read: Responsável por ler dados. O método realiza a leitura com base no endereço fornecido, a depender do componente determinado pelo método citado mais acima. Sendo a memória, chama readAsInt() para recuperar os dados de memória, caso contrário ajusta o endereço e lê os dados do buffer de pixels.

Listing 49. Método read()

```
1 public int[] read(int address, final int endDataPosition);
```

Nesta classe, o uso de switch e do método wichComponentType garante que os acessos à memória e ao framebuffer sejam realizados corretamente, evitando erros de segmentação ou corrupção de dados.

3.4. GPU

A capacidade de processar grandes volumes de dados visuais em tempo real depende da eficiência das GPUs, dispositivos especializados na execução paralela de cálculos matemáticos intensivos. Nos últimos anos, com o aumento da demanda por gráficos em alta resolução e renderização complexa, a pesquisa em otimização e customização de GPUs tornou-se essencial para atender aos requisitos de desempenho das aplicações contemporâneas [Cook 2012].

Neste projeto, foi desenvolvida uma GPU simplificada, capaz de realizar a renderização gráfica por meio da biblioteca OpenGL, amplamente utilizada em aplicações interativas em 3D e 2D [Khronos Group 2024]. A GPU visa não apenas melhorar a eficiência dos cálculos gráficos, mas também permitir que seja possível implementar funções específicas otimizadas para determinadas aplicações, como simulações de física ou aprendizado de máquina . A utilização de bibliotecas como OpenGL, em conjunto com APIs modernas como GLFW, proporciona uma interface robusta para a comunicação entre hardware e software, facilitando a criação e manipulação de janelas de renderização, gerenciamento de texturas e buffers de quadros.

No emulador proposto, é inclusa a construção de uma janela de renderização interativa utilizando a biblioteca GLFW, que gerencia os eventos de interface gráfica e as interações de teclado e mouse, enquanto o OpenGL é responsável pelo desenho de cenas. Além disso, a conversão de vídeos em tempo real para renderização de vértices através da GPU será explorada, utilizando o framework FFmpeg para a captura de frames, demonstrando o potencial de uma arquitetura customizada para otimizar tarefas gráficas específicas [Cheng et al. 2012]. Na figura 10, segue o diagrama de classes do módulo GPU para melhor visualização do funcionamento:

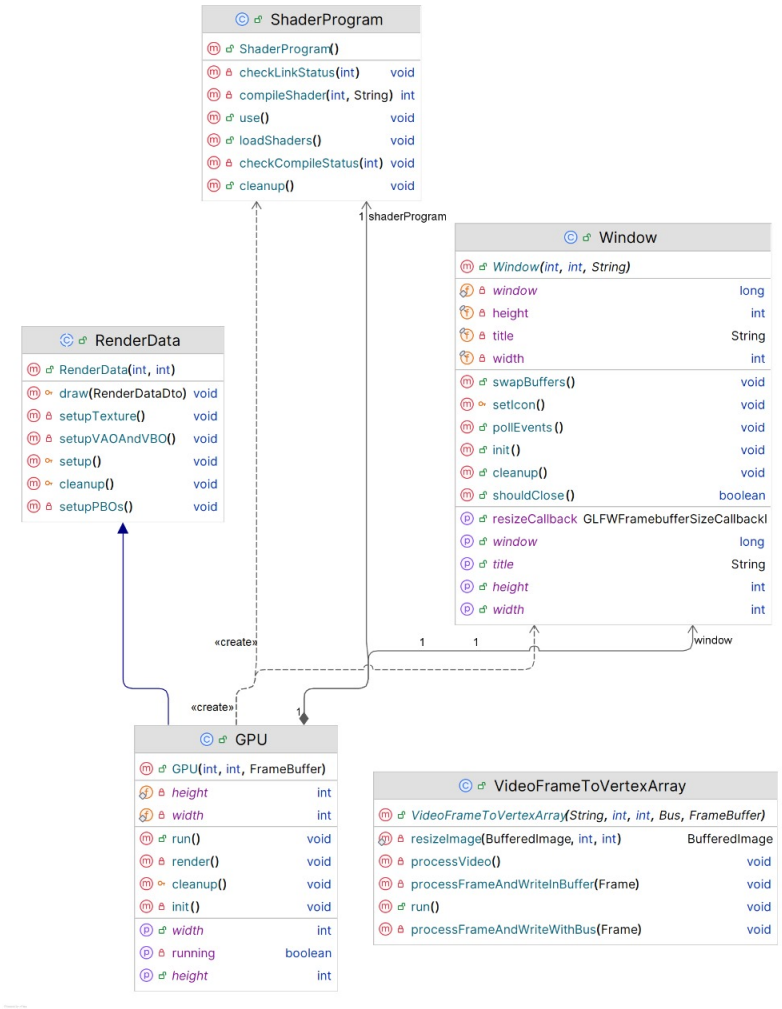

Figure 10. Diagrama de classes da GPU

A fim de detalhar o processo de construção da GPU para o emulador, será descrito a estrutura de cada classe, os métodos utilizados nela e sua devida responsabilidade.

3.4.1. Classe `RenderData`

A princípio, inicia-se a descrição da classe `RenderData`, a qual gerencia a configuração, atualização e desenhos de dados de renderização para o OpenGL. Esta é responsável por preparar os dados de vértices, texturas, buffers e objetos relacionados que serão utilizados pela GPU a fim de renderizar os gráficos em uma janela. A Tabela 12 mostra os atributos utilizados.

Table 12. Atributos e Buffers Utilizados na Renderização

Atributo	Descrição
Width	Dimensão horizontal da textura a ser renderizada. Este valor é definido como parâmetro no construtor e utilizado para configurar as texturas OpenGL.
Height	Dimensão vertical da textura a ser renderizada. Este valor é definido como parâmetro no construtor e utilizado para configurar as texturas OpenGL.
BufferSize	Tamanho do buffer definido pelo FrameBuffer. Representa a quantidade de pixels a serem processados.
numVertices	O número de vértices que serão desenhados. Como cada vértice possui 8 componentes, entre eles a posição e cor, o cálculo é feito dividindo o tamanho do buffer por 8.
vao, *vbo*, *textureId*	Identificadores para o Vertex Array Object (VAO), Vertex Buffer Object (VBO) e a textura. Esses serão utilizados pela biblioteca OpenGL para organizar os dados de vértices e texturas.
pboIds	Um array de identificadores de Pixel Buffer Objects (PBOs), que são buffers usados para transmissões eficientes de texturas.
nextPboIndex	Um índice que controla qual PBO será chamado na próxima operação, utilizando-se do algoritmo round-robin.

Listing 50. Construtor da classe RenderData

```
1 public RenderData(final int width, final int height);
```

Dentro dessa classe, os métodos se valem dos atributos para implementar as funções de inicialização do OpenGL. Nesse passo, oportuno destacar a definição de cada uma delas:

A) setup: Configura o ambiente OpenGL para renderização, habilita texturas, ajusta o alinhamento de pixels e convoca métodos de configurações de textura, como VAO/VBO e PBOs;

Listing 51. Método setup()

```
1 public void setup();
```

B) setupTexture: Inicializa as configurações de textura e aloca memória para ela. Além disso, gera o identificador da textura através da função glGentextures(). Configura parâmetros de textura, como comportamento de repetição e filtragem, a fim de definir como será mapeada a textura e se a exibição será ampliada ou reduzida. Por fim, valendo-se da função glTexImage2D(), utiliza as dimensões especificadas;

Listing 52. Método setupTexture()

```
1 private void setupTexture();
```

C) setupVAOandVBO: Configura o VAO e VBO, utilizados pelo OpenGL, a fim de organizar e armazenar dados de vértices, ou seja, gera e vincula o VAO e VBO, além

de configurar os atributos do vértice para posição, cor e coordenadas de textura usando a função glVertexAttribPointer();

Listing 53. Método setupVAOAndVBO()

```
1 private void     private void setupVAOAndVBO();
```

D) setupPBOs: Criando dois PBOs, um duplo buffer, e alocando espaço para ele, permite que o próximo quadro esteja preparado enquanto o quadro atual ainda é renderizado, assim configura o PBO utilizados para melhorar a performance da transmissão de texturas para a GPU;

Listing 54. Método setupPBOs()

```
1     private void setupPBOs();
```

E) Draw: Este método lida com o desenho real dos dados de vértices e a atualização da textura, ou seja, atualiza o VBO com os novos dados através do objeto RenderDataDto, seleciona e usa um dos PBOs gerados para realizar o carregamento de textura, a fim de renderizar sem bloqueio, mapeia na janela a textura no OpenGL com glTexSubImage2D() e desenha os vértices com a função glDrawArrays();

Listing 55. Método draw(RenderDataDto dataDto)

```
1 public void draw(RenderDataDto dataDto);
```

F) cleanup(): Por fim, esse método realiza a limpeza dos recursos alocados no OpenGL através da exclusão dos buffers e da textura, liberando os recursos da GPU e evita, dessa forma, o vazamento de memória após o uso.

Listing 56. Método cleanup()

```
1  public void cleanup();
```

Em suma, a classe RenderData gerencia a configuração e execução da renderização gráfica do OpenGL. A classe utiliza de técnicas avançadas como PBOs para otimizar o carregamento e a renderização, com o objetivo de garantir uma performance eficiente. Tais funcionalidades são fundamentais para qualquer aplicação gráfica envolvendo renderização 2D ou 3D.

3.4.2. Classe `ShaderProgram`

A classe *ShaderProgram* gerencia a compilação, conexão e uso do programa de shaders no OpenGL. Um *shaders* é um programa que roda na GPU o qual realiza operações gráficas, tal qual o cálculo de cores e transformações geométricas.

O único atributo dessa classe é o programID, este sendo o identificador do programa de sombreamento, o qual será criado a partir de shaders individuais, o *vertex* e *fragment*.

Os métodos presentes nessa classe são para realizar os cálculos necessários para as operações gráficas. Sendo eles:

A) loadShaders: Este método carrega e realiza a compilação dos shaders de vértices e fragmentos, conectando-os em um único programa, o qual pode ser usado pela GPU. Para tal, existem etapas a ocorrerem para a compilação, sendo essas:

1) Compilação do vertex shader: Processa os vértices de um objeto, tomando como entrada a posição e a cor do vértice, assim como as coordenadas da textura, gerando a posição final e suas variáveis de saída;

2) Compilação do *fragment shader*: Este processa o fragmento, ou pixel, da imagem, determinando a cor final. Nessa etapa, o shader faz uso de uma textura e multiplica seus valores pela cor dos vértices, a fim de calcular a cor final de cada fragmento;

3) Criação do programa: Após as etapas anteriores, o método cria um programa de *shaders* e anexa ambos os *shaders* compilados a ele;

4) Conexão do programa: O método loadShaders() busca a função glLinkProgram() para integrar os *shaders* em um só programa executável pela GPU. Esse ciclo é indispensável para agregar o comportamento dos dois *shaders* em um único *pipeline* gráfico;

5) A última etapa é a limpeza, uma vez que finalizado os processos anteriores, os *shaders* são deletados para economizar recursos.

Listing 57. Método loadShaders()

```
1    public void loadShaders();
```

B) compileShader: Método responsável por compilar um *shader*, seja um *vertex* ou *fragment* shader. Utilizando a função glCreateShader() é criado um identificador de tipo especificado. Após isso, o código é repassado como argumento e atribuído ao *shader* criado com a função glShaderSource(). Por fim, ocorre a compilação desse *shader* através do chamado da função glCompilerShader() e a verificação com o método checkCompileStatus();

Listing 58. Método compileShader()

```
1    private int compileShader(int type, String source);
```

C) checkLinkStatus : Este é responsável por verificar se ocorreu de maneira devida a integração do programa de *shaders*. Aplica a função glGetProgramiv() para verificar o status da *"linkagem"* (conexão), e se houver erro, dispara uma exceção detalhada do erro utilizando glGetProgramInfoLog();

Listing 59. Método checkLinkStatus()

```
1     private void checkLinkStatus(int program);
```

D) checkCompileStatus: É empregado para verificar o status de compilação de um *shader* individual. Similar ao método anterior, no entanto utiliza a função glGetShaderiv() para obter a conjuntura e, da mesma forma, dispara exceção em caso de erro;

Listing 60. Método checkCompileStatus()

```
1    private void checkCompileStatus(int shader);
```

E) use: O método liga o programa de shaders compilado e ”linkado”. Ao acionar a função glUseProgram(programID), a GPU é instruida a usar o programa de shaders específico em operações subsequentes de renderização;

Listing 61. Método use()

```
1    public void use();
```

F) cleanup: Assim como na classe anterior, esse método deleta os recursos associados ao shaders, com a função glDeleteProgram (programID), evitando, assim, vazamento de memória.

Listing 62. Método cleanup()

```
1    public void cleanup();
```

Diante disso, a clase ShaderProgram encapsula a funcionalidade de carregar, compilar e gerenciar *shaders* em OpenGL. O processo inclui a verificação de erros durante o processo, para garantir o funcionamento correto antes de ser utilizada para renderização.

3.4.3. Classe `VideoFrameToVertexArray`

A classe VideoFrameToVertexArray foi utilizada para testar os métodos da GPU de forma isolada do funcionamento da CPU. É responsável por processar um arquivo de vídeo e converter cada *frame* em um array de vértices para renderização. O processamento é feito em uma thread separada, o que permite que o vídeo seja processado de forma paralela ao restante do emulador. Na Tabela 13, defini-se os atributos inéditos.

Table 13. Atributos Relacionados ao Processamento de Vídeo

Tipo de Atributo	Descrição
videoFilePath	O diretório do arquivo a ser processado.
converter	Objeto da classe Java2DFrameConverter, com o objetivo de converter os quadros de vídeo para o formato necessário.
bus	O atributo do sistema de comunicação (Barramento) para permitir a escrita dos dados convertidos em outro componente.

Para que exista maior concepção do propósito de cada método pertencente a classe, irá ser mantido uma explicação contextual de cada uma dessas, logo:

A) ResizeImage: Este redimensiona uma imagem para as dimensões especificadas. Utilizando o Graphics2D para redesenhar a imagem em um novo tamanho com o método Graphics2D.drawImage(), da biblioteca OpenGL;

Listing 63. Método resizeImage()

```
1    private static BufferedImage resizeImage(BufferedImage
        originalImage, int targetWidth, int targetHeight);
```

B) run: O método sobrescreve o método run() da classe Thread. É o inicio da thread que começa o processamento de vídeo. Quando iniciada, o método processVideo() é chamado para iniciar o processamento do vídeo;

Listing 64. Método run()

```
1    public void run();
```

C) processVideo: É o principal para processar o vídeo. Utiliza o FFmpegFrameGrabber, da biblioteca FFmpeg, para abrir o arquivo de vídeo e processar cada quadro capturado [FFmpeg Developers 2024]. Para inicializar o processo de captura de *frames*, utiliza o método grabber.start(), após isso, a função grabber.grabImage() captura os *frames* individualmente. Para cada *frame*, o método processFrameAndWriteWithBus() é convocada, a fim de realizar sua função e escrever os dados no atributo bus. Ao fim, o método processVideo calcula o tempo necessário para manter a taxa de quadros em 60 por segundo, (60 *fps*) e aguarda o tempo restante;

Listing 65. Método processVideo()

```
1    private void processVideo();
```

D) processFrameAndWriteWithBus: Processa o quadro individual ao redimensionar a imagem e mapear os dados de cor de cada pixel presente no barramento (bus). Ao ser chamada, converte o *frame* em uma imagem, utilizando o Java2DFrameConverter. Após, o método resizeImage() é chamado e, quando finalizado, ocorre a escrita no barramento de forma que itera cada pixel da imagem redimensionada. Aplicando o método bus.write() para escrever no barramento a cor da cada pixel, ou seja, o endereço do pixel na memória é atualizado conforme os pixels são processados;

Listing 66. Método processFrameAndWriteWithBus()

```
1    private void processFrameAndWriteWithBus(Frame frame)
```

E) processFrameAndWriteInBuffer: Similar ao método anterior, porém ao invés de escrever diretamente no barramento, realiza a escrita no framebuffer. Logo, como no processFrameAndWriteWithBus(), converte o quadro para bufferedImage e redimensiona a imagem para as dimensões desejadas. Após isso, normatiza as coordenadas, ou seja, para cada pixel, as coordenadas x e y são ajustadas para valores entre -1 e 1. Essas coordenadas são essenciais para criar um array de vértices. Esses dados são escritos no framebuffer em um formato específico e, ao finalizar o alocamento, o método convoca a função framebuffer.swap() para alterar os buffers, garantindo, então, que a próxima operação de renderização utilizará um novo conjunto de dados.

Listing 67. Método processFrameAndWriteInBuffer()

```
1    private void processFrameAndWriteInBuffer(Frame frame)
        throws MemoryException{...}
```

Resumidamente, a classe VideoFrameToVertexArray processa o arquivo de vídeo em tempo real e converte cada quadro em dados a serem utilizados para renderização. Essa abordagem permite que o processamento de vídeo e a conversão para dados de vértice sejam feitos em uma thread separada, a fim de garantir que a aplicação continue a operar enquanto os quadros são convertidos em um módulo de teste.

3.4.4. Classe `Window`

A classe Window gerencia a criação e operação de uma janela de renderização utilizando a ferramenta GLFW, integrando-a com o OpenGL [Khronos Group 2024]. Essa classe possui dois novos atributos antes não definidos, mostrados na Tabela 14.

Table 14. Atributos da Janela

Tipo de Atributo	Descrição
title	O título exibido na barra de título da janela.
window	É o handle da janela criada. Representa a janela no contexto da biblioteca GLFW.

A classe window implementa funções especializadas na criação dessa tela, tais como:

A) init: Inicialização e configuração da janela. Através do uso do método GLFW.glfwCreateWindow, ela é inicializada com a largura, altura e título especificados pelos atributos. Além disso, ao utilizar a função glfwGetVideoMode, armazena a dimensão do monitor principal e posiciona a janela ao centro dela. Para a integração com o OpenGL, utiliza do método GLFW.glfwMakeContextCurrent para contextualizar o openGL com o GLFW e do GLFW.glfwMakeContextCurrent(1) para ativar o V-Sync. A exibição da janela é feita com a chamada do método GLFW.glfwShowWindow() e, por fim, inicializa o openGL com GL.createCapabilities();

Listing 68. Método init()

```
1     public void init();
```

B) setResizeCallback: É utilizada quando a janela é redimensionada. A partir do momento em que há alteração na dimensão, é chamado o método glViewPort() para centralizar novamente a tela de acordo com a nova dimensão.

Listing 69. Método setResizeCallback()

```
1     public void setResizeCallback(GLFWFramebufferSizeCallbackI
          callback);
```

C) setIcon: Define o ícone da janela. Ele utiliza um bloco de memória temporário para carregar a imagem e, valendo-se do método da biblioteca STB, retona um buffer contendo os dados do arquivo PNG utilizado. O ícone é definido utilizando o GLFW e a memória temporária é liberada utilizando outro método STB;

Listing 70. Método setIcon()

```
1     protected void setIcon();
```

D) shouldClose: Verifica se a janela deve ser fechada, retornando true se o método GLFW.glfwWindowShouldClose() indicar que o evento de fechamento foi gerado;

Listing 71. Método shouldClose()

```
1     public boolean shouldClose();
```

E) pollEvents: Processa todos os eventos pendentes da janela, como cliques e pressionamentos de teclas, com o método GLFW.glfwPollEvents(), garantindo, assim, que o sistema de eventos do GLFW esteja atualizado;

Listing 72. Método pollEvents()

```
1       public void pollEvents();
```

F) cleanup: Por fim, este método comum nas classes da GPU, libera os recursos associados à janela.

Listing 73. Método cleanup()

```
1       public void cleanup();
```

Essa abordagem descrita acima permite que a janela seja controlada eficientemente para aplicações gráficas interativas.

3.4.5. Classe `GPU`

A classe GPU é a principal na construção da GPU do emulador. Ela interliga e comanda o momento em que cada classe filha será chamada. A classe GPU herda de thread, o que significa que a GPU será executada em uma thread separada para lidar com operações gráficas, sem bloquear o restante do sistema. A classe controla todos os outros processos necessários para a execução da GPU.

Os atributos existentes na classe são conhecidos, uma vez que a definição deles já foi realizada. No entanto, é válido citá-los na Tabela 15 para melhor explicar os métodos pertencentes à GPU.

Table 15. Atributos da Janela de Renderização

Tipo de Atributo	Descrição
Width	Atributo estático que representa a dimensão horizontal da janela de renderização. Acessível de fora da classe.
Height	Atributo estático que representa a dimensão vertical da janela de renderização. Acessível de fora da classe.
frameBuffer	Um objeto da classe FrameBuffer. Armazena os dados da imagem que serão renderizados pela GPU.
shaderProgram	Objeto da classe ShaderProgram. Responsável por carregar e usar os shaders para a renderização.
renderData	Objeto da classe RenderData. Armazena as informações de renderização.
window	Objeto da classe Window. Representa a janela onde a renderização ocorrerá. Ela é inicializada com as dimensões especificadas.

O construtor inicializa esses atributos acima. Tais valores são usados para configurar a janela de renderização e processar dados gráficos.

Listing 74. Construtor da classe GPU

```
1 public GPU(final int width, final int height, final FrameBuffer
     frameBuffer);
```

Os métodos dessa classe são baseadas em implementações e chamadas das classes citadas anteriormente. Dentre elas, estão os métodos:

A) run: Implementação principal do loop da thread que executa a GPU. Esse método incia a GPU, chamando os métodos a seguir a fim de executar de maneira devida.

Listing 75. Método run()

```
1         public void run();
```

B) init: Inicializa os componentes essenciais da GPU, como o GLFW. Cria a janela de renderização e configura o *callback* para redimensionamento da janela. Carrega e ativa os *shaders*. Inicializa a classe RenderData com a largura e altura da janela e configura os dados para renderização e, por fim, configura a cor de fundo da janela com glClearColor;

Listing 76. Método init()

```
1         public void init();
```

C) isRunning: Verifica se a janela deve permanecer em aberto, retornando *true* enquanto a janela não for fechada. Essa verificação mantém o ciclo de renderização ativo;

Listing 77. Método isRunning()

```
1         public void isRunning();
```

D) render: Esse método lida com a lógica de renderização a cada quadro. Ele limpa o buffer de cores e profundidade para preparar a janela para o próximo frame, utilizando o glClear. Além disso, desenha os dados renderizados ao chamar o método *draw* da classe RenderData, transmitindo os dados gráficos do frameBuffer. Por fim, atualiza a tela realizando a troca dos buffers e processa eventos como interações do usuário com a janela;

Listing 78. Método render()

```
1         public void render();
```

E) cleanup: Garante que todos os recursos alocados sejam liberados de forma adequada.

Listing 79. Método cleanup()

```
1         public void cleanup();
```

Em síntese, a classe GPU gerencia todo o ciclo de vida da renderização gráfica. Inicia o ambiente gráfico com o GLFW e OpenGL, cria a janela e configura shaders e dados de renderização. Ademais, cada componente é isolado em sua própria classe, para seguir boas práticas da programação modular.

4. Validação e Demonstrações

Com base na arquitetura implementada, incluindo o barramento, a GPU e o gerenciamento de memória, este capítulo busca avaliar o funcionamento do emulador em diversos cenários, abordando tanto sua funcionalidade quanto a precisão da emulação em reproduzir o comportamento de uma arquitetura RISC-V.

A validação inclui a análise das classes implementadas, através de testes de precisão de execução, reprodução de instruções e operações gráficas pelo sistema. Para tanto, foram executadas operações que testam a eficiência do barramento, as quais verificam a integridade dos dados transmitidos entre memória e GPU, e garantem a capacidade da CPU emulada de manipular gráficos através de uma interface 2D de pixels, gerando a saída gráfica por meio do OpenGL. Os testes de desempenho e precisão foram fundamentais para evidenciar as potencialidades e limitações do emulador, destacando tanto os casos de sucesso quanto os possíveis gargalos identificados.

Neste capítulo, é necessário a explicação de como foi válidado cada módulo desenvolvido no projeto, a fim de evidenciar o funcionamento devido de cada classe e a aplicabilidade dos métodos. Essa abordagem permitirá uma avaliação completa das capacidades da emulação, destancando os benefícios e os desafios enfrentados na implementação e possibilitando uma análise crítica sobre o funcionamento e a aplicabilidade do emulador desenvolvido.

4.1. Validações

4.1.1. Memória e Barramento

Para garantir o correto funcionamento de sistemas de emulação de hardware, como GPUs, CPUs e sistemas embarcados, é essencial validar os módulos de memória e barramento, haja vista que esses componentes formam a base para a troca de dados entre todos os módulos do emulador.

No código desenvolvido para a emulação do sistema, o barramento conecta a Memory e o FrameBuffer à GPU, possibilitando que dados sejam transferidos eficientemente entre a memória principal e a unidade de processamento gráfico. A classe Bus foi testada em conjunto com a GPU, enquanto o módulo de memória foi validado de forma isolada com uma classe de teste específica chamada MemoryTest.

A classe MemoryTest executa uma série de operações fundamentais para verificar o correto funcionamento da Memory na leitura, escrita e tratamento de exceções. Esta classe de teste utiliza o framework JUnit para definir um conjunto de testes automatizados e assegura que os métodos principais da Memory estejam funcionando conforme o esperado.

Dentro da classe de teste, o método writeAndReadMemoryTest realiza as seguintes etapas:

a) Primeiramente, uma instância de Memory é criada com o tamanho predefinido de 1024 bytes. Em seguida, é definido um array de endereços (addresses), contendo todos os valores de 0 a 1023, a fim de simular um *range* completo de endereços válidos de memória;

b) Três arrays de dados, data1, data2 e data3, são inicializados com tamanhos de 5, 10, 15 bytes, respectivamente. Estes arrays são preenchidos com valores aleatórios gerados para simular diferentes blocos de dados que serão escritos e lidos em diferentes posições da memória. Os valores são gerados de forma pseudoaleatória, assegurando que os testes não dependam de valores estáticos e possam validar a consistência dos dados em diferentes execuções;

c) A Memory é testada para garantir que os dados escritos em posições específicas possam ser retornadas corretamente. Cada bloco de dados é escrito em uma posição de memória específica, e o teste subsequente compara os valores retornados pela função read() com os dados originalmente escritos. Esse processo assegura que tanto a write() quanto a read() estão funcionando corretamente, preservando a integridade dos dados;

d) São realizados testes adicionais para verificar o comporatmento da memória em condições de erro. Estes testes incluem tentativas de leitura e escrita em posições de memória fora dos limites, por exemplo caso esteja acima de 1024 e índices negativos. O método assertThrows() é utilizado para assegurar que exceções do tipo MemoryException sejam lançadas nestas circustâncias, indicando que a Memory possui verificações adequadas de limites e que está preparada para tratar acessos inválidos.

A abordagem acima promove maior segurança e robustez no módulo, pois cada operação crítica foi testada individualmente. Além disso, os testes de exceção provam que o módulo de memória lida com erros de forma adequada, garantindo que o sistema de emulação possa evitar falhas graves durante o processamento de dados em condições adversas.

Diante do exposto, a MemoryTest cumpre um papel essencial na validação do sistema, oferencendo uma base segura para o desenvolvimento e integração dos demais módulos, como a GPU e o Bus.

4.1.2. Processador

A validação do processador dá-se, principalmente, pela leitura correta das instruções. Para atingir esse objetivo, foi criada uma classe de teste unitário em Java a fim de validar a funcionalidade do método decodeInstruction da classe Decoder, presente na CPU, que interpreta instruções RISC-V binárias.

Ao utilizar a biblioteca JUnit 5, a classe garante que as instruções binárias sejam decodificadas corretamante para suas representações em strings legíveis. Nesta seção, será analisado o funcionamento da classe e dos métodos.

A princípio, deve-se comentar sobre as importações e as definições primárias da classe, exibidas na Tabela 16:

Listing 80. Importações

```
1 import org.junit.jupiter.api.Test;
2 import static br.faustech.cpu.Decoder.decodeInstruction;
3 import static org.junit.jupiter.api.Assertions.assertEquals;
```

Dentro da classe DecoderTest, seu único método é o decoderTest. Este executa

Table 16. Anotações e Métodos de Teste

Tipo de Atributo	Descrição
@Test	Anotação do JUnit que marca métodos de teste unitário.
decodeInstruction	Método estático importado da classe Decoder que transforma instruções binárias em representações legíveis.
assertEquals	Método que compara o resultado da função testada com o valor esperado para verificar a validade do teste.

uma série de instruções binárias específicas e, em cada caso, verifica se o resultado da decodificação corresponde ao esperado. A exemplo disso, temos:

Listing 81. Exemplo do método decoderTest()

```
@Test
public void decoderTest() {
    //Set instruction ADD
    int instruction = 0b00000000000000011000001010110011;
    String type = decodeInstruction(instruction);
    assertEquals("add␣rd=5,␣rs1=3,␣rs2=0", type);
    ...
}
```

Desse modo, é possível perceber que cada instrução é definida como um número binário, ou seja:

a) instruction = 0b00000000000000011000001010110011: Define uma instrução binária específica, neste caso, a instrução ADD.

Além disso, o método decodeInstruction é chamado para interpretar a instrução e retorna uma string com representação textual da instrução, indicando o tipo da operação e os registradores/valores envolvidos.

A fim de validar a saída, o método assertEqual verifica se a saída do método anterior corresponde à string esperada. Caso os valores não coincidiram, o teste falha, indicando que a decodificação está incorreta.

A função principal dos testes em DecoderTest é verificar se o decodeInstruction consegue traduzir as instruções binárias para representações legíveis em alto nível, essenciais para a compreensão e execução de comandos pelo emulador RISC-V. O sucesso de cada teste indica que o método da classe Decoder interpreta corretamente a estrutura e retorna os valores esperados, validando a funcionalidade e confiabilidade do emulador.

4.1.3. GPU

Durante a criação da Classe GPU, foi desenvolvido o método VideoFrameToVertexArray(), descrita como um método para teste da GPU de maneira independente dos outros módulos.

Para a validação da GPU, a classe de teste GPUTeste é fundamental para avaliar o funcionamento dos componentes principais da GPU em contraste com apenas a utilização

do método citado acima. No entanto, a inclusão do barramento, FrameBuffer, a Memory e o processamento de vídeo, são feitos utilizando a VideoFrameToVertexArray().

A classe GPUTeste contém configurações iniciais e variáveis estáticas que definem o caminho do vídeo, VIDEO_PATH, a largura e altura da tela, WIDTH e HEIGHT, e o tamanho do FrameBuffer e da Memory. Estas definições são essenciais para inicializar os componentes de forma compatível com a resolução e quantidade de memória necessárias para o teste de emulação da GPU e processamento de vídeo.

Listing 82. Variáveis da classe GPUTest

```
1     private static final String VIDEO_PATH = "";
2
3     private static final int WIDTH = 320;
4
5     private static final int HEIGHT = 240;
6
7     private static final int FRAME_BUFFER_SIZE = WIDTH * HEIGHT
          * 4;
8
9     private static final int MEMORY_SIZE = 4096;
```

O método gpuTest dentro da classe é responsável por criar instâncias dos componentes principais e iniciar a execução do teste.

Listing 83. Método gpuTest()

```
1  @Test
2     public void gpuTest(){
3     ...
4     }
```

A instância FrameBuffer é inicializada com o tamanho de memória necessário para armazenar uma imagem de vídeoo, calculado com base nas dimensoes e profundidade de cor da tela, por exemplo 320x240 e 4 bytes por pixel. O FrameBuffer age como um buffer de saída de vídeo que será processador pela GPU e visualizado na interface.

O Bus é inicializado com o FrameBuffer e uma nova instância de Memory. Este barramento serve como canal de comunicação entre a memória principal, onde os dados do programa e do vídeo são armazenados, e o FrameBuffer. Em testes de emuladores, o barramento é essencial para validar a troca de dados entre CPU, GPU e e memória[Hennessy and Patterson 2017].

Depois da instânciação dos métodos acima, o método VideoFrameToVertexArray é utilizado e, após isso, a GPU é criada com as mesmas dimensões do FrameBuffer e iniciada em uma Thread separada. Durante a execução, a GPU lê os dados do FrameBuffer e os renderiza na tela. A fim de verificar se a thread da GPU ainda está ativa, existe um loop de monitoramento. Se a GPU estiver em um estado TERMINATED, o método verifica o estado do videoProcessor e interrompe qualquer thread que ainda esteja ativa, garantindo que todos os processos sejam encerrados.

O teste avalia a integração entre os componentes da GPU e o sistema de emulação, verificando se a GPU consegue renderizar os dados corretamente a partir do FrameBuffer.

Ela também assegura que o barramento está transmitindo dados de forma adequada entre a memória e o framebuffer para a renderização gráfica.

4.2. Demonstrações

Após as etapas de validação e construção do emulador, foram desenvolvidas duas demonstrações visuais para ilustrar o funcionamento do emulador. A primeira focou na reprodução do monumento símbolo da UFMS, uma imagem simplificada para o teste. O processo envolveu o uso de uma imagem de entrada e a transformação dela em dados que pudessem ser utilizadas diretamente pelo emulador para exibir a logo na tela. Para isso, utilizou-se um script em Python que processa a imagem da logo e gera o código C correspondente, no qual cada pixel da imagem é associado a valores de cor e posição que serão processadas pelo emulador.

4.2.1. Monumento Símbolo da UFMS

Esse script tem como propósito principal simplificar a imagem em um número menor de cores, identificar intervalos de pixels com cores iguais e, então, gerar um código C otimizado para preencher esses pixels em um emulador.

O script é composto por diversas funções responsáveis por carregar, processar e gerar o código C correspondente à imagem fornecida. Abaixo, segue a explicação de cada uma dessas:

A) quantitize_image: A função reduz o número de cores da imagem, um processo conhecido como quantização. Neste caso, a imagem é carregada e convertida para o modo RGBA para preservar as informações de transparência, redimensionada para 320x240 pixels, a resolução do emulador, e reduzida para um número limitado de cores, definido pelo usuário, com um valor padrão de 16 cores. Essa quantização é importante, pois reduz a complexidade de processamento e o tamanho do código C gerado;

B) get_image_matrix: Esta função converte a imagem quantizada em uma matriz de valores hexadecimais representando as cores, usando a biblioteca NumPy para aplicar essa transformação de maneira eficiente. Cada cor é convertida para um valor hexadecimal no formato 0xRRGGBB, que representa o código RGB. A função também "comprime" a matriz para obter uma lista linear de cores, o que facilita o cálculo de intervalos de pixels consecutivos com a mesma cor;

C) count_intervals: Responsável por calcular intervalos consecutivos de pixels com a mesma cor. Ela percorre a lista de cores e, quando encontra uma sequência de pixels com a mesma cor, registra o intervalo de início e fim. O objetivo é agrupar os pixels por cor, o que torna o código final mais otimizado e reduz o número de operações necessárias no código C

D) merge_intervals: A função aprimora ainda mais a compactação dos intervalos, unindo os consecutivos de pixels que compartilham a mesma cor em um único intervalo;

E) generate_c_code: Gera o código C baseado nos intervalos de cores e posições dos pixels obtidos nas etapas anteriores. A função é projetada para otimizar o código de saída, utilizando três estratégias diferentes. A primeira estratégia é para um único pixel, o qual gera uma atribuição direta. A segunda, para intervalos de até 3 pixels, realizando,

da mesma forma, atribuições diretas sem loop. Por fim, a terceira estratégia é para intervalos maiores, utilizando um loop for, atribuindo o valor de cada cor para cada pixel no intervalo, o que torna o código mais compacto. O código gerado define o endereço de cada pixel na memória e aplica as cores correspondentes. Ao final, uma instrução para o swap_address atualiza a tela, exibindo a imagem no emulador;

F) save_to_file e main: São funções complementares e principais. A primeira salva o código C gerado em um arquivo com o nome especificado pelo usuário, este pode ser compilado e executado no ambiente de emulação para visualizar a logo. A função main integra todas as etapas, solicitando ao usuário o caminho da imagem e o número de cores desejados para a quantização. A partir desses dados, a função gera a lista de cores, conta os intervalos, aplica a fusão de intervalos e, por fim, salva o código C gerado em um arquivo de saída.

Dessa forma, a imagem pode ser exibida no emulador, possibilitando uma visualização do "paliteiro" da UFMS e evidenciando a capacidade do emulador em processar gráficos básicos. Esse processo é fundamental para validar a exibição de imagem no ambiente de simulação.

Figure 11. Renderização Monumento Símbolo da UFMS

4.2.2. Visualização das Cores RGB atráves de Interrupções

Para além dessa demonstração, foi desenvolvida uma solução para o teste das interrupções com a janela gráfica. Esta foi construída com o intuito de alternar entre as cores vermelha, verde e azul a partir do momento em que as teclas R, G e B do teclado são pressionadas.

A fim de atingir esse objetivo, um código em Assembly do RISC-V foi desenvolvido para testar a resposta do sistema a interrupções do teclado. Essa funcionalidade

é implementada por meio de uma Tabela de vetores de interrupção que redireciona as chamadas para os manipuladores específicos, um para o temporizador, *timer*, e outro para o teclado, *keyboard*.

A configuração inicial é realizada através do rótulo boot, na qual as interrupções são desativadas temporariamente via CSR mie(Machine Interrupt Exception), este controla quais interrupções podem ser habilitadas, permitindo que a CPU responda a eventos externos, como temporizadores e sinais periféricos. Após isso, o vetor de interrupção no registrador CSR mtvec é configurado, ele contém o endereço onde a CPU deve buscar o manipulador (*handler*) de interrupções quando uma interrupção ou exceção ocorre. Além disso, o registrador CSR mstatus, que é usado para armazenar e gerenciar o estado do modo máquina, reativa as interrupções. Após essa inicialização, o código executa a função main.

A função main configura a cor inicial da tela, preenchendo uma região da memória com o valor RGB em hexadecimal. Após o preenchimento da memória para o plano de fundo, o endereço swap_address é atualizado para exibir o novo estado na tela gráfica.

Como é utilizado uma tabela de interrupções, no bloco table, a primeira linha, j timer_interrupt, define o manipulador de interrupções do temporizador, enquanto a segunda, j_key_interrupt define o manipulador de interrupções do teclado.

Os manipuladores de interrupção do temporizador não são o foco principal na construção da demonstração, porém exibe o salvamento e restauração de registro, esses sendo necessários em rotinas de interrupção a fim de garantir que o contexto original seja mantido. A função timer_interrupt salva os registradores na pilha, incrementa o valor no endereço de memória 1024 e restaura o contexto dos registradores.

O manipulador key_interrupt é responsável por alterar a cor da janela gráfica conforme a tecla pressionada. Primeiramente, ele lê o valor de um registrador CSR (CSR 0x343) que indica a tecla pressionada, armazenando-a no registrador de uso geral a6. Em seguida, são feitas comparações para verificar qual tecla foi pressionada e, com base na tecla, o valor de cor é alterado, evidenciado na Tabela 17.

Table 17. Definições de Cores Baseadas nas Teclas Pressionadas

Tecla Pressionada	Código Hexadecimal	Cor Definida
R (82)	**0xffff0000**	Define a cor como vermelho.
G (71)	**0xff00ff00**	Define a cor como verde.
B (66)	**0xff0000cc**	Define a cor como azul.

Caso a tecla pressionada seja o código de interrupção 256 o código executa a instrução ebreak, comumente utilizada em execução para debug. Por fim, o manipulador restaura o estado dos registradores.

É pertinente observar que para a tecla B foi utilizado o código hexadecimal 0xff0000cc, pois ao ser aplicado um valor acima desse, no padrão RGB, o valor azul não era exibido na tela do emulador, uma anomalia que não foi cabível de resolução.

O código descrito acima é eficiente para o objetivo inicial. A implementação utiliza os registros CSR para compreender as entradas do teclado e determinar qual tecla foi

pressionada, permitindo, assim, a mudança de cor sem uma estrutura condicional complexa.

Em síntese, a primeira demonstração, com a geração do logo da UFMS, evidenciou a capacidade do emulador em lidar com renderização gráfica, convertendo imagens em código C otimizado para o display simulado. O processo de quantização e geração do código ilustrou a funcionalidade do emulador para gerar gráficos complexos com base em dados pré-definidos, o qual oferece controle detalhado sobre a representação visual.

Por sua vez, a segunda demonstração é voltada para a gestão de interrupções com mudança de cores mediante comandos do teclado, demonstrando, portanto, a precisão e robustez do sistema em responder a sistemas externos. Ao interpretar as teclas de entrada para alterar entre as cores vermelho, verde e zul, o emulador comprovou a funcionalidade de seu controlador de interrupções e destacou a eficiência do código de interrupção em Assembly para manipular e restaurar registros, bem como para responder às operações de I/O.

Por fim, as demonstrações validaram as principais funcionalidades do emulador, confirmando sua adequação e eficácia em tarefas de controle gráfico e tratamento de interrupções - uma base sólida para futuros aprimoramentos e aplicações mais complexas.

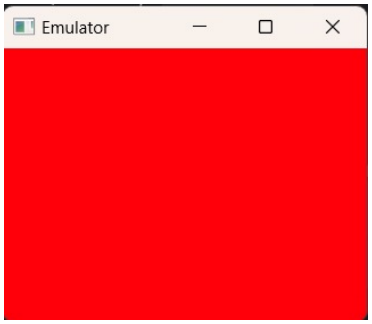

Figure 12. Tecla R: Vermelho

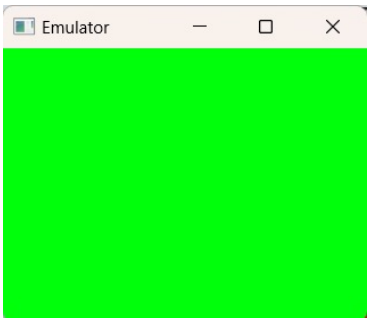

Figure 13. Tecla G: Verde

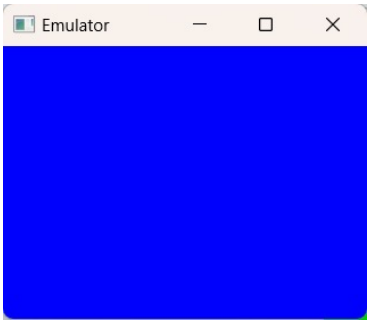

Figure 14. Tecla B: Azul

Figure 15. Cores exibidas pelo emulador ao interpretar a interrupção

5. Conclusão

A emulação em software preservou a fidelidade dos componentes da arquitetura RISC-V e permitiu a realização de testes detalhados de funcionalidade, abrangendo aspectos críticos como a manipulação da memória, interrupções do teclado e demonstrações gráficas, que incluem a renderização do logotipo da UFMS e a mudança de cores da janela em resposta a eventos de teclado. Essas funcionalidades não só validaram a eficácia da emulação, mas também destacaram o potencial educacional do projeto.

O estudo inicialmente teve como objetivo a implementação de um emulador para a arquitetura RISC-V em um ambiente de hardware, visando o desenvolvimento em um System-on-Chip (SoC) implementado em FPGA. Contudo, devido à alta complexidade associada à validação e ao tempo disponível, foi necessário redirecionar o foco para uma implementação em software. Essa mudança estratégica possibilitou uma maior agilidade no desenvolvimento, garantindo a finalização do projeto dentro do prazo estabelecido, sem comprometer os objetivos e o mérito da pesquisa.

A transição para um ambiente de software não apenas simplificou o processo de desenvolvimento, mas também proporcionou uma plataforma robusta e flexível para simular o funcionamento da arquitetura RISC-V. A nova abordagem permitiu a implementação de módulos essenciais, como CPU, memória e barramento, além de uma interface gráfica baseada em OpenGL para a emulação da GPU. Essa interface não apenas enriqueceu a experiência do usuário, mas também facilitou a visualização dos processos de renderização e manipulação de dados.

Dessa forma, o projeto não apenas cumpriu o objetivo de criar uma simulação que captura a essência e a estrutura de um ambiente RISC-V, mas também se estabeleceu como uma ferramenta educativa e experimental valiosa. Essa ferramenta pode ser expandida ou modificada para novos desafios de pesquisa, servindo como um ponto de partida para estudos mais aprofundados em arquitetura de computadores e design de sistemas. Assim, o projeto representa um passo significativo na compreensão e exploração das capacidades da arquitetura RISC-V em contextos acadêmicos e de desenvolvimento tecnológico [Emulador 2024].

References

Asanović, K. and Patterson, D. A. (2014). Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*.

Cheng, Y., Liu, Q., Zhao, C., Zhu, X., and Zhang, G. (2012). Design and implementation of mediaplayer based on ffmpeg. In *Software Engineering and Knowledge Engineering: Theory and Practice: Volume 2*, pages 867–874. Springer.

Cook, S. (2012). *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes.

Emulador, G. (2024). Emulator - risc-v emulator project. `https://github.com/gabrielf4ustino/emulator`. Accessed: 2024-12-04.

Fernandez, M. P. (2015). Arquitetura de computadores.

FFmpeg Developers (2024). Ffmpeg documentation. Accessed: 2024-10-10.

Hennessy, J. L. and Patterson, D. (2014). *Arquitetura de computadores: uma abordagem quantitativa*, volume 5. Elsevier Brasil.

Hennessy, J. L. and Patterson, D. A. (2017). Computer organization and design risc-v edition: The hardware software interface.

IBM Documentation (2023). *Emulators in AIX 7.3*. Accessed: 2024-10-10.

Khronos Group (2024). Opengl reference pages. Accessed: 2024-10-10.

Scott Chacon and Ben Straub (2024). *Pro Git Book, 2nd Edition - Capítulo: Começando - O Básico do Git*. Acessado em: 2024-10-29.

Sijstermans, F. (2017). Risc-v core implementation in nvidia falcon. Presentation at the 6th RISC-V Workshop, NVIDIA Corporation, 2017.

Song, N. Y., Son, Y., Han, H., and Yeom, H. Y. (2016). Efficient memory-mapped i/o on fast storage device. *ACM Transactions on Storage (TOS)*, 12(4):1–27.

Waterman, A., Lee, Y., Avizienis, R., Patterson, D. A., and Asanovic, K. (2015). The risc-v instruction set manual volume ii: Privileged architecture version 1.7. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-49*.

Waterman, A., Lee, Y., Patterson, D. A., and Asanovic, K. (2014). The risc-v instruction set manual, volume i: User-level isa, version 2.0. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54*, page 4.

Waterman, A. S. (2016). *Design of the RISC-V instruction set architecture*. University of California, Berkeley.

Glossary

CPU Unidade Central de Processamento. 1, 4–6, 8, 10, 11, 14, 15, 17, 18, 24, 30, 35, 36, 38, 41, 43

CSR registradores de Controle e Status. 11, 15, 41

GPU Unidade de Processamento Gráfico. 1, 4, 5, 18, 24–26, 28–30, 33–38, 43

MMIO entrada/saída mapeada em memória. 5, 23

RISC-V Arquitetura de conjunto de instruções (ISA) aberta e de uso livre. 1, 3–6, 8–11, 15, 17, 35–37, 42, 43