Implementação de Interface Gráfica em Java para Emulador de Sistema Computacional baseado em Arquitetura RISC-V

José Othávio Alves Monteiro[1]

[1]Faculdade de Computação – Universidade Federal de Mato Grosso do Sul

jose.o@ufms.br

Abstract. *This work presents a Graphical User Interface (GUI) for a RISC-V architecture emulator. The emulator was written in Java and offers a flexible platform for architecture studies and experimentation in the RISC-V environment. The interface was developed using the Swing Toolkit, which includes a sophisticated set of GUI components that create a rich and interactive user interface. The emulator has different functionalities, such as a Register Tab and RISC-V instruction visualization, which offers the user a flexible platform and a greater capacity to develop, perform tests, and validate information, with potential for future expansions.*

Resumo. *Este trabalho apresenta uma Interface Gráfica do Usuário (GUI) para um emulador de arquitetura RISC-V. O emulador foi feito na linguagem Java e oferece uma plataforma flexível para estudos de arquitetura e experimentação no ambiente RISC-V. A interface foi desenvolvida utilizando o framework Swing, pelo conjunto sofisticado de componentes de GUI que criam uma interface rica e interativa ao usuário. O emulador conta com diferentes funcionalidades como Aba de Registradores e visualização de instruções em RISC-V, que oferecem ao usuário uma plataforma flexível e uma capacidade maior de desenvolver, realizar testes e validar informações, também com um potencial para futuras expansões.*

Sumário

1. Introdução e História da Swing

Neste trabalho iremos definir a estrutura de uma Interface Gráfica do Usuário (GUI) que foi desenvolvida para um emulador de instruções da arquitetura RISC-V. Essa interface foi projetada para integrar-se ao emulador, proporcionando uma experiência visual que facilita o processo de depuração, execução e análise das instruções.

O principal objetivo do projeto é construir uma interface que permita ao desenvolvedor visualizar, interagir e controlar o comportamento do emulador de maneira clara e intuitiva. Para isso, foram desenvolvidas janelas que interagem com os principais elementos do sistema emulado, como a unidade gráfica e o processador, utilizando os recursos da linguagem Java. A biblioteca gráfica escolhida foi o Swing, devido ao seu framework leve, funcional e de fácil integração.

1.1. Motivação e Objetivos do Trabalho

O presente trabalho propõe o desenvolvimento de uma interface gráfica amigável para um emulador de instruções da arquitetura RISC-V, com o objetivo de tornar sua utilização mais acessível e didática. Essa interface busca proporcionar uma visualização clara da execução do código, acompanhamento dos registradores e interação com a simulação em tempo real.

Entre as contribuições esperadas estão: (i) a ampliação da usabilidade do emulador existente; (ii) a facilitação do ensino e aprendizado dos conceitos da arquitetura RISC-V; e (iii) a criação de um ambiente visual que favoreça a experimentação prática por parte de estudantes e professores.

Assim, este trabalho não se limita à implementação técnica de uma interface, mas propõe uma solução detalhada que pode ser integrada a práticas pedagógicas, contribuindo com o ensino de arquiteturas de computadores em ambientes acadêmicos.

1.2. História da Swing

Swing não é uma sigla. O nome representa a escolha colaborativa de seus designers quando o projeto foi iniciado no final de 1996. Swing, na verdade, faz parte de uma família maior de produtos Java conhecida como Java Foundation Classes (JFC), que incorpora muitos dos recursos das Internet Foundation Classes (IFC) da Netscape, bem como aspectos de design da divisão Taligent da IBM e da Lighthouse Design. Swing está em desenvolvimento ativo desde o período beta do Java Development Kit (JDK) 1.1, por volta da primavera de 1997. As APIs Swing entraram em beta no segundo semestre de 1997 e foram lançadas inicialmente em março de 1998. Quando lançadas, as bibliotecas Swing 1.0 continham quase 250 classes e 80 interfaces [Marc Loy 2002].

Inicialmente, o Java utilizava o AWT (Abstract Window Toolkit) para criar interfaces gráficas e então Swing surgiu como uma alternativa mais flexível, poderosa e principalmente uma alternativa pura, ou seja, escrita e manipulada exclusivamente em Java. Em 2010, após a Oracle adquirir a Sun, uma nova biblioteca chamada JavaFX foi projetada para, gradualmente substituir a Swing [Deitel and Deitel 2016]. Essa nova biblioteca possuía ferramentas mais modernas como integração com CSS, suporte a video e audio, entre outras funcionalidades. Porém o engajamento com essa nova biblioteca foi menor do que o esperado e interfaces desenvolvidas se mantiveram com o Kit Swing, por-

tanto essa substituição nunca ocorreu completamente e até hoje em dia, Swing se mantêm uma boa opção no desenvolvimento de interfaces gráficas.

Outra vantagem fundamental do Java Swing é seu mecanismo abrangente de tratamento de eventos, que permite aos desenvolvedores responder às interações do usuário, como cliques em botões, movimentos do mouse e entradas de teclado. Isso é alcançado por meio do uso de ouvintes de eventos e métodos de callback, que permitem aos desenvolvedores definir ações personalizadas a serem executadas quando eventos específicos ocorrem. Por exemplo, os desenvolvedores podem anexar um objeto da classe *ActionListener* a um componente de botão para manipular eventos de clique de botão ou um objeto da classe *MouseListener* a um componente para responder a eventos do mouse, como cliques, arrastamentos e movimentos do mouse. Essa arquitetura orientada a eventos facilita a criação de interfaces gráficas de usuário responsivas e interativas que engajam os usuários e proporcionam uma experiência de usuário fluida [Botwright 2024].

2. Trabalhos Relacionados

Nessa seção, iremos citar projetos que possuem relação com o presente trabalho, seja pelo foco na emulação da arquitetura RISC-V ou pelo desenvolvimento de interfaces gráficas para ambientes educacionais.

2.1. Emulador RISC-V desenvolvido no TCC de alunos da FACOM/UFMS (2024)

O trabalho de conclusão de curso desenvolvido por alunos da FACOM/UFMS consistiu no desenvolvimento de um emulador funcional da arquitetura RISC-V [de Freitas Silva et al. 2024]. O projeto contempla diversos componentes de um sistema computacional, como memória, barramento de comunicação, dispositivos de entrada e saída (teclado e placa gráfica) e processador.

A proposta foi implementada em Java, utilizando as bibliotecas GLFW e OpenGL para a criação de uma janela de renderização interativa. Essa janela gerencia eventos relacionados à interface gráfica, incluindo interações via teclado e mouse, além de realizar o desenho das cenas. O objetivo do projeto foi oferecer uma ferramenta didática que auxilie no ensino de arquitetura de computadores com base no padrão RISC-V [de Freitas Silva et al. 2024].

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 [de Freitas Silva et al. 2024].

Figura 1. Renderização Monumento Símbolo da UFMS

2.2. Emuladores baseados na arquitetura MIPS

A arquitetura MIPS (Microprocessor without Interlocked Pipeline Stages) é amplamente adotada no contexto educacional, sendo uma das arquiteturas mais tradicionais para o ensino de organização e arquitetura de computadores. Assim como a arquitetura RISC-V, a MIPS pertence à categoria das arquiteturas RISC (Reduced Instruction Set Computer), caracterizadas por um conjunto reduzido e eficiente de instruções [Patterson and Hennessy 2014].

Entre os emuladores mais utilizados para a arquitetura MIPS destaca-se o MARS (MIPS Assembler and Runtime Simulator), uma ferramenta desenvolvida em Java com propósitos educacionais [Vollmar and Kidd 2006]. O MARS combina um montador com um ambiente de simulação, oferecendo recursos para edição, montagem e execução de programas escritos em linguagem assembly. Essa ferramenta é amplamente empregada em disciplinas introdutórias ao estudo de sistemas computacionais e linguagens de baixo nível.

A origem da arquitetura MIPS remonta a 1981, quando foi desenvolvida como parte de um projeto de pesquisa em VLSI (Very Large Scale Integration) na Universidade de Stanford, sob a liderança do professor John Hennessy. O emulador MARS, por sua vez, foi desenvolvido por Pete Sanderson e Ken Vollmar, enquanto atuavam como docentes nas universidades Otterbein e Missouri State, respectivamente.

2.3. RARS

Com o surgimento e crescente adoção da arquitetura RISC-V no meio acadêmico, foi desenvolvido o RARS (RISC-V Assembler and Runtime Simulator), um emulador educacional que visa facilitar o ensino e o aprendizado da linguagem assembly RISC-V. A ferramenta oferece uma interface gráfica simples e acessível, permitindo a edição, montagem e execução de programas, de forma semelhante ao que é feito com o MARS para a arquitetura MIPS [Landers 2024].

O RARS é um projeto mais recente, idealizado por Benjamin Landers, e tem sido amplamente utilizado por instituições de ensino que optaram por migrar para a arquitetura RISC-V, aproveitando seu caráter aberto e modular. Além de simular a execução de instruções, o RARS também disponibiliza funcionalidades adicionais, como visualização

de registradores, memória, e suporte a chamadas de sistema, contribuindo significativamente para o processo de aprendizado [Landers 2024].

3. Framework Swing para Interfaces Gráficas

3.1. Conceitos básicos

Swing é uma framework que faz parte da biblioteca padrão do Java. Essas ferramentas são utilizadas para desenvolver Interfaces Gráficas com o Usuário (GUIs). Os componentes da Swing podem ser estendidos, modificados ou agrupados para criar interfaces com aparência e comportamento personalizados [Marc Loy 2002].

Dentre os componentes essenciais da Swing, podemos citar o *JFrame*, *JPanel*, *JTable*, *JMenuBar*, entre outros. O *JFrame* é a principal classe de janela, pois representa a própria janela principal da aplicação, incluindo atributos como título, tamanho, botões de controle (fechar, minimizar, etc.), bordas, entre outros.

Para manipular diretamente os elementos visuais da interface, utiliza-se o *ContentPane*, que representa o painel de conteúdo interno do JFrame. É nele que são adicionados os demais componentes gráficos, como listas, tabelas, menus e painéis (*JPanel*).

Outra classe importante é a *SwingWorker*. Como as interações com o usuário em Swing ocorrem na Thread de Despacho de Eventos (EDT), todas as ações são por padrão síncronas. Isso significa que se uma operação demorada for executada nessa thread, a interface pode travar. Considerando que uma das funcionalidades propostas foi a exibição em tempo real dos valores dos registradores, para que essa atualização ocorra sem comprometer a responsividade da interface, utilizou-se um *SwingWorker*. Ele permite executar tarefas em segundo plano (background thread) e atualizar a interface de forma segura, permitindo que a thread secundária copie constantemente os dados dos registradores da CPU e os exiba na janela de depuração.

Em testes realizados, foram consideradas outras frameworks, porém a biblioteca Swing - apesar de mais antiga e mais simples — mostrou-se mais adequada às necessidades do projeto.

3.2. Classes e componentes da Swing

Dentro dessa framework, existem diversas classes que realizam ações ou exibem informações em um formato específico. Como a framework contém diversos componentes e nem todos foram utilizados, mencionaremos apenas os relevantes para a interface desenvolvida.

3.2.1. JTextField

Essa é a classe utilizada para definir algum texto, ela funciona recebendo uma uma *String* e pode ser adicionada em outras classes para exibir a informação.

```
JTextField memoryField = new JTextField("0x00000000");
```

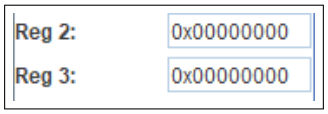

Figura 2. Componente JTextField

3.2.2. JFormattedTextField

Esse é um componente que é combinado com diferentes classes para definir algum valor com formatação customizada, na interface utilizamos o componente para exibir um número, portanto a classe recebe uma instância de um tipo inteiro obtido pela classe *NumberFormat*.

```
NumberFormat amountFormat = NumberFormat.getIntegerInstance();
JFormattedTextField widthField = new JFormattedTextField(
    amountFormat);
widthField.setValue(320);
```

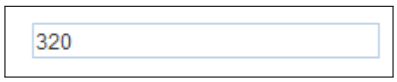

Figura 3. Componente JFormattedTextField

3.2.3. JLabel

Essa é a classe utilizada para definir algum rótulo, ela funciona recebendo uma *String* e pode ser adicionada em outras classes como *JPanel* para exibir o texto.

```
JLabel widthLabel = new JLabel("Width:");
```

Figura 4. Componente JLabel

3.2.4. JCheckBox

Essa é a classe utilizada para definir uma caixa de seleção, ela funciona sendo criada e então definindo uma função de callback que será chamada quando a opção for selecionada. Na interface utilizamos principalmente para definir a configuração de Dark Mode.

```
JCheckBox darkModeCheckbox = new JCheckBox();
darkModeCheckbox.setSelected(parent.getDarkModeEnabled().get());
darkModeCheckbox.addActionListener(e -> {});
```

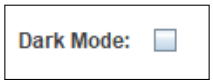

Figura 5. Componente JCheckBox

3.2.5. GridBagConstraints

Essa é uma ferramenta que é combinada com diferentes classes para definir configurações para uma organização customizada, na interface utilizamos em alguns lugares quando precisamos customizar o layout de alguma tela. Portanto é aplicado os valores específicos nessa instânciação e então o objeto é adicionado com algum outro componente.

```
GridBagConstraints gbc = new GridBagConstraints();
gbc.insets = new Insets(5, 5, 5, 5);
gbc.gridx = 0; gbc.gridy = 1; gbc.fill = GridBagConstraints.NONE
   ; gbc.weightx = 0;
content.add(widthLabel, gbc);
```

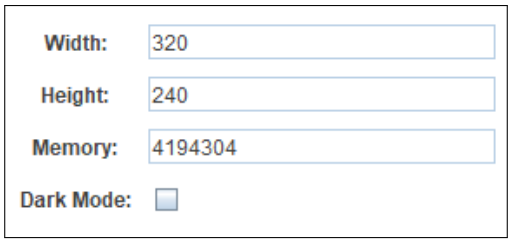

Figura 6. Classe GridBagConstraints

3.2.6. BorderLayout

Essa é a classe auxiliar utilizada para definir um layout, existem diferentes tipos como *GridLayout*, *FlowLayout*, porém geralmente utilizamos o *BorderLayout* que apenas define a direção da ferramenta atual em seu componente pai, as direções podem ser *NORTH*, *EAST*, etc.

```
getContentPane().setLayout(new BorderLayout());
getContentPane().add(createActionBar(), BorderLayout.NORTH);
```

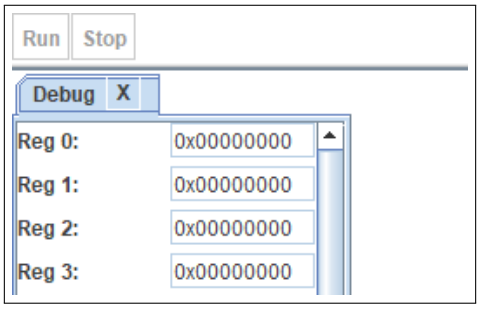

Figura 7. Classe BorderLayout

3.2.7. JMenuItem

Esse é o componente utilizado para definir um botão que será clicável em algum menu, ele é criado passando uma *String* com o texto que será visível no botão e então é preciso definir uma função de callback que realizará alguma ação após o click.

```
JMenuItem openItem = new JMenuItem("Open");
openItem.addActionListener(e -> openFile());
```

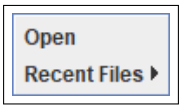

Figura 8. Componente JMenuItem

3.2.8. JMenu

Esse é o componente utilizado para definir um dropdown em alguma barra de ações que pode ser selecionado para exibir diferentes botões. Ele é criado passando uma *String* que servirá como texto para a opção e então é preciso definir os itens que serão clicáveis.

```
JMenu fileMenu = new JMenu("File");
fileMenu.add(openItem);
```

Figura 9. Componente JMenu

3.2.9. JMenuBar

Esse é o componente utilizado para definir uma barra de ações em alguma janela. Ele é criado sem nenhum parâmetro e então é preciso adicionar as listas de dropdown que serão exibidas e clicáveis.

```
JMenuBar menuBar = new JMenuBar();
menuBar.add(fileMenu);
```

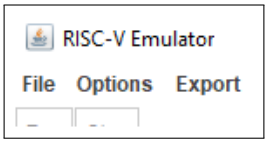

Figura 10. Componente JMenuBar

3.2.10. JButton

Esse é o componente utilizado para definir um botão que pode ser clicável para ser realizado alguma ação similar a um *JMenuItem*. A diferença nesse caso é que ele pode ser aplicado em diferentes lugares e nessa interface ele é utilizado na seção especial de ações que possuem botões como *Run* e *Stop*. O componente também é criado com uma *String* para o texto que será exibido e então é preciso criar uma funçaõ de callback que irá realizar a ação.

```
JButton runButton = new JButton("Run");
runButton.addActionListener(e -> startEmulator());
```

Figura 11. Componente JButton

3.2.11. JPanel

Essa é a classe base para definir algum conteúdo ou janela, ela funciona como uma caixa para uma tela e recebe a organização ou definição dos dados.

```
JPanel contentPanel = new JPanel(new BorderLayout());
```

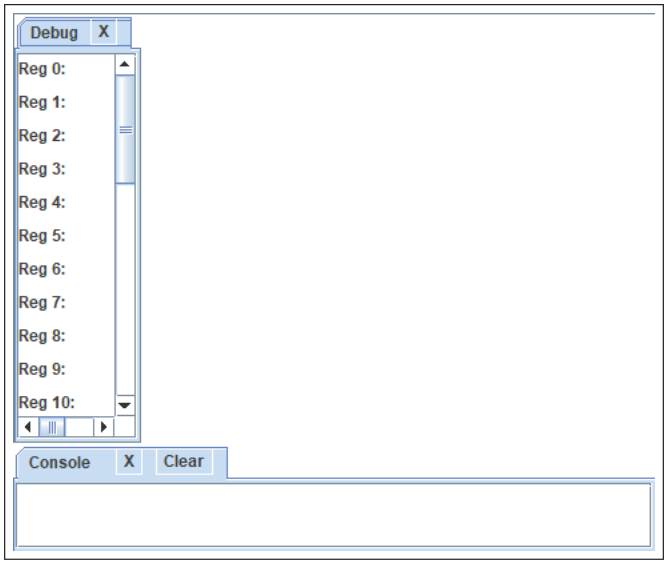

Figura 12. Componente JPanel

3.2.12. JTabbedPane

Essa é a classe que pode ser utilizada em conjunto customizar o layout de uma janela. Essa classe em específico pode receber diversos *JPanel* ou outras classes similares e então organiza os mesmos como se fossem abas na tela.

```
JTabbedPane centerTabbedPane = new JTabbedPane();
```

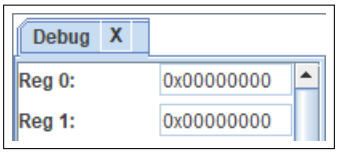

Figura 13. Componente JTabbedPane

3.2.13. JSplitPane

Essa é uma classe similar ao *JTabbedPane*, porém ao invés de organizar os *JPanel* em abas, essa ferramenta divide a janela em 2 na direção horizontal ou vertical. Ela é criada recebendo diretamente os componentes que ficaram dentro, juntamente com a configuração da direção.

```
JSplitPane centerSplitPane = new JSplitPane(JSplitPane.
   HORIZONTAL_SPLIT, centerTabbedPane, contentPanel);
```

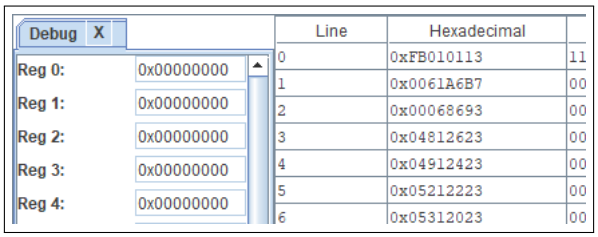

Figura 14. Componente JSplitPane

3.2.14. TableRowSorter

Esse é um componente especial que é utilizado para filtrar e ordenar textos. Para definir o componente, é preciso passar como parâmetro o *model* utilizado para a criação do *contentPanel* e então criar um *JTextField* para receber o texto de busca. No final é adicionado um *DocumentListener* no *filterField* que irá enviar o texto para o *TableRowSorter*. Podemos visualizar a seguir as funções mencionadas.

```
TableRowSorter<DefaultTableModel> sorter = new TableRowSorter<>(
   model);

filterField.getDocument().addDocumentListener(new
   DocumentListener() {
    public void insertUpdate(DocumentEvent e) { applyFilter(); }
}
```

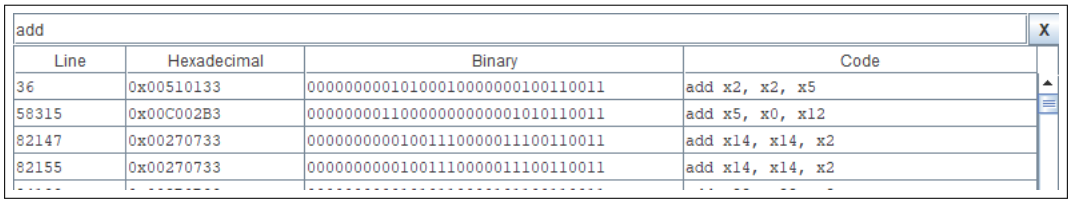

Figura 15. Componente TableRowSorter

3.2.15. InputMap

Esse é outro componente especial utilizado na interação com o desenvolvedor, ele aplica uma função de callback quando ocorrer alguma sequência de botões apertados no teclado enquanto a janela está em foco. Isso é utilizado para que o filtro seja dinâmico e seja exibido após o desenvolvedor clicar nas teclas ”Control”+ ”F”.

```
InputMap inputMap = contentPanel.getInputMap(JComponent.
   WHEN_IN_FOCUSED_WINDOW);

inputMap.put(KeyStroke.getKeyStroke(KeyEvent.VK_F, InputEvent.
   CTRL_DOWN_MASK), "showFilterField");
```

4. Interface Gráfica em Java

Com base no emulador e na framework Swing que foram minuciados nos capítulos anteriores, esta seção detalhará o desenvolvimento da interface que foi projetada para controlar e agrupar informações do emulador. O objetivo principal será descrever as etapas de implementação do emulador, a fim de abordar as escolhas técnicas, tais como as principais funcionalidades.

A interface tem como propósito auxiliar na configuração, desenvolvimento e depuração do emulador. Considerando que no trabalho anterior algumas das informações eram aplicadas diretamente no código, a GUI apresenta novas funcionalidades que auxiliam nesse uso.

Foram utilizadas as mesmas ferramentas de desenvolvimento do trabalho que implementou o emulador RISC-V, ou seja, para o desenvolvimento do programa foi utilizado a IDE IntelliJ e para versionamento do código foi utilizado Git em conjunto com Github para hospedagem.

Para auxiliar no entendimento da interface, foi utilizado o formato de Diagrama de Classes para desenvolver uma imagem que elabore visualmente a estrutura das janelas principais da GUI. A Figura 16 mostra a estrutura da interface gráfica que é composta por diversas janelas, botões e painéis interconectadas.

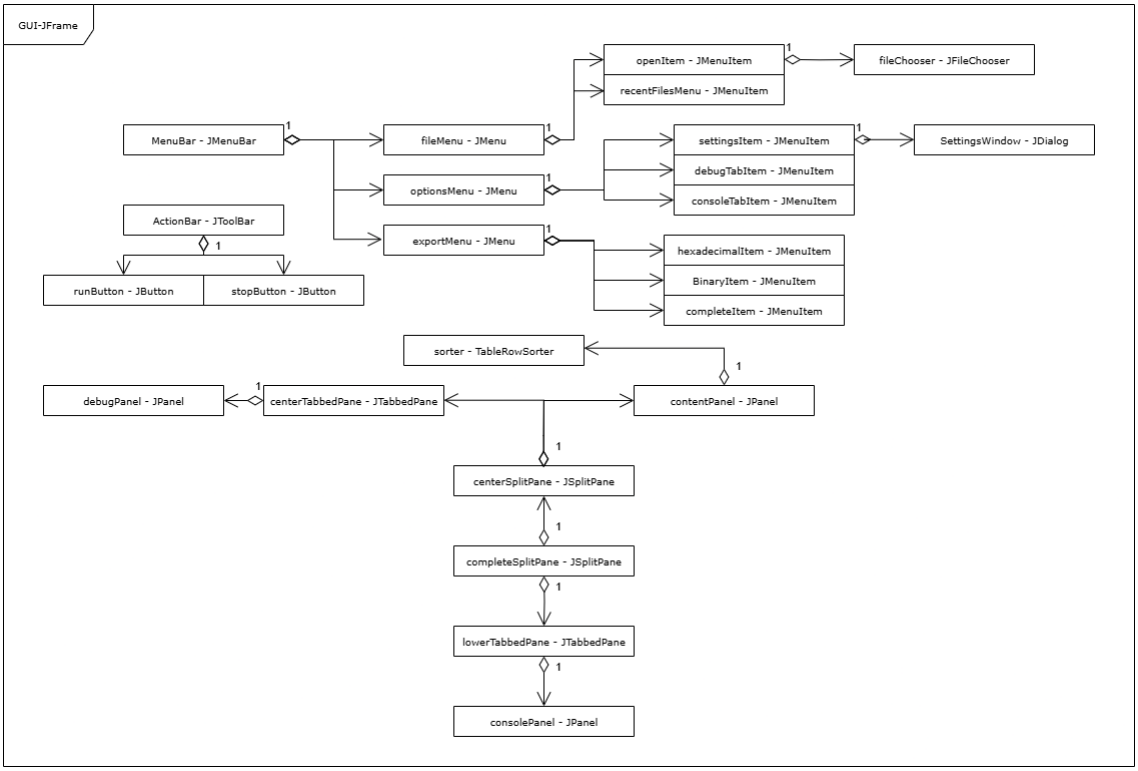

Figura 16. Diagrama de classes representando janelas da GUI

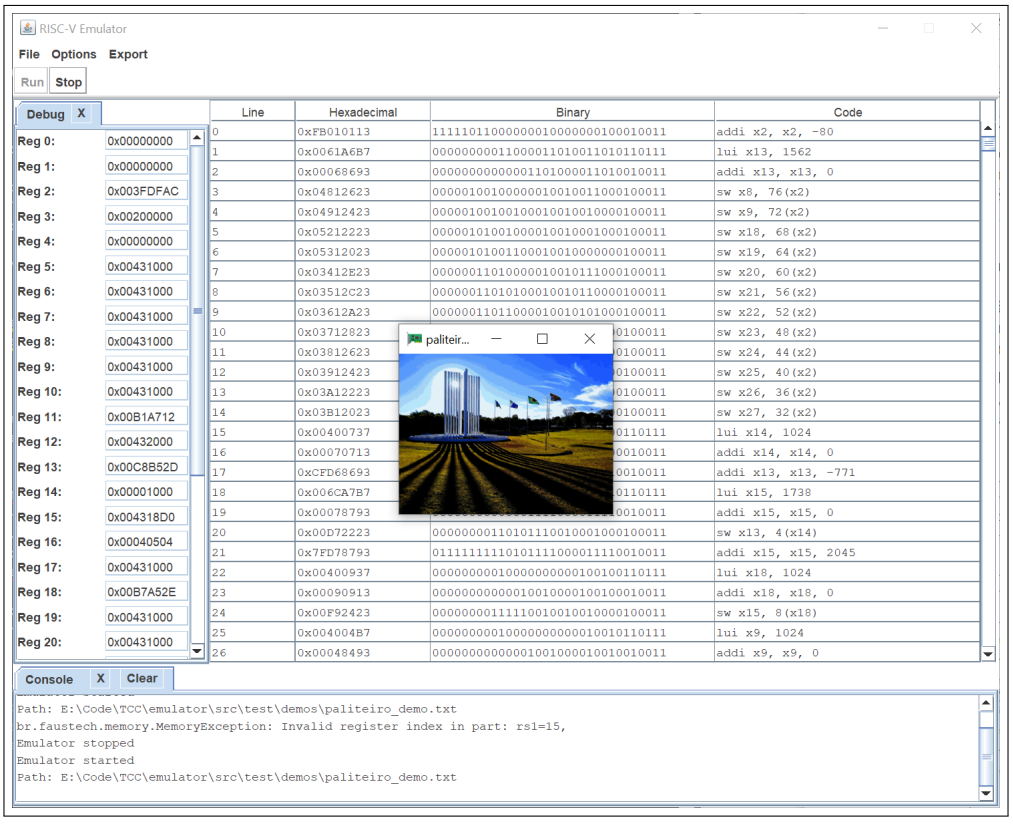

Figura 17. Interface Gráfica

4.1. contentPanel

O *ContentPanel* foi uma das principais áreas desenvolvidas, o intuito dessa seção é exibir visualmente para o desenvolvedor as instruções que foram decodificadas do binário. Conforme pode ser validado na Figura 18, para apresentar o máximo de dados possíveis para o desenvolvedor, foram criados 4 colunas selecionáveis em uma tabela, a coluna *Line* que exibe o número da linha em questão, *Hexadecimal* exibindo o número correspondente a instrução em formato Hexadecimal, *Binary* que apresenta os 32 dígitos binários referente a instrução e *Code* que representa a instrução completa em RISC-V.

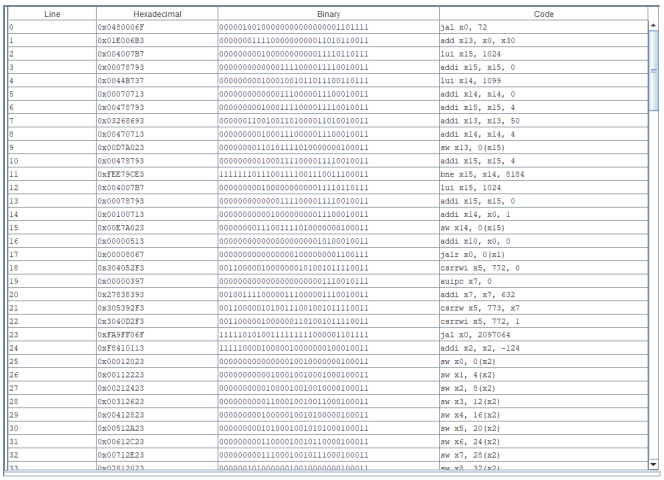

Figura 18. Janela com o conteúdo do código

Por se tratar de uma seção que é exibida imediatamente na interface após a inicialização, mas que tem seu conteúdo preenchido apenas após a seleção de um programa, sua criação é realizada por meio da definição de um *JPanel* vazio. Esse painel é armazenado em um atributo de classe, permitindo seu acesso posterior, conforme ilustrado no Listing 1.

O conteúdo da janela é carregado posteriormente, por meio da chamada à função apresentada no Listing 2. Essa função pode ser invocada em dois momentos: quando o usuário seleciona um programa por meio da opção *File* → *Open*, que permite a escolha de um arquivo, o segundo caso é pela opção *File* → *Recent Files*, que exibe uma lista com os dez últimos programas abertos.

Listing 1. Criação do `contentPanel`

```
contentPanel = new JPanel(new BorderLayout());
```

Listing 2. Método `openFileContentTab()`

```
private void openFileContentTab(File file);
```

As instruções decodificadas inicialmente eram criadas utilizando uma simples exibição textual com uma barra de rolagem, porém esse formato começou a se mostrar ineficiente e imprático ao usuário. Portanto a estrutura da janela é feita com uma tabela que contêm a seguinte estrutura: as quatro colunas com os respectivos valores citados

anteriormente, as linhas correspondendo as instruções do programa e a barra de rolagem faz a interação com as 4 colunas paralelamente. Esse formato foi desenvolvido utilizando a classe chamada *DefaultTableModel*, que recebe uma lista de nomes para o cabeçalho e uma matriz de objetos da classe *Object*. Mesmo que no momento sejam utilizados apenas valores textuais, foi utilizado a classe *Object* para o conteúdo pois a classe referente ao modelo *DefaultTableModel* espera esse tipo de valor e ao utilizar o mesmo, existe uma maior flexibilidade e facilidade em adicionar outros tipos de variáveis. Portanto esse objeto foi inicializado utilizando a linha Listing 3.

Listing 3. Criação do `DefaultTableModel`

```
model = new DefaultTableModel(tableData, columnNames);
```

4.2. debugPanel

O *debugPanel* é a funcionalidade desenvolvida que interage diretamente com a CPU, essa aba que auxilia na depuração pode ser aberta ao clicar no menu em *Options* → *Open Debug Tab*. Após a aba ser aberta, conforme pode ser verificado na Figura 19, irá ser exibido uma lista com 32 registradores listados de 0 a 31 e após o programa ser selecionado e inicializado, a GUI irá fazer uma cópia constante de cada registrador da CPU, buscando os dados e exibindo para o desenvolvedor em tempo real. Essa janela pode ser fechada clicando no mesmo botão que foi utilizado na abertura ou também pode ser utilizado o botão ”X”que existe ao lado do título da aba.

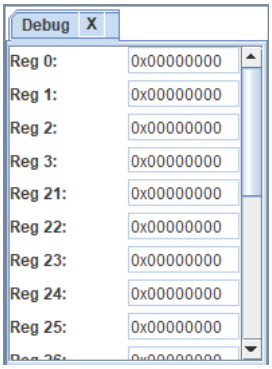

Figura 19. Janela com o conteúdo de depuração

Pelo seu formato especial, essa aba é criada no momento em que o botão *Open Debug Tab* é clicado e é utilizado um atributo da classe denominado *centerTabbedPane*, o mesmo é criado sem valores conforme pode ser verificado no Listing 4, porém o seu conteúdo é preenchido através de uma classe auxiliar chamada *RegisterUpdater* que estende a classe apresentada anteriormente chamada *SwingWorker*. O objeto que é denominado *updater* é instanciado no momento em que a interface começa a rodar o programa e a CPU é criada, após isso ocorrer a CPU faz a chamada da função da GUI Listing 5 enviando os registradores como parâmetro. Essa função então cria o *updater* passando como parâmetro a referência para os registradores e para a lista de componentes *JTextField* que exibe os seus valores, para que então seja executado em plano de fundo uma função que aplicará os dados dos registradores em seus respectivos campos na interface.

Listing 4. Criação do `centerTabbedPane`

```
centerTabbedPane = new JTabbedPane();
```

Listing 5. Método `setRegisterUpdater()`

```
public void setRegisterUpdater(int[] registers);
```

Para melhor visualização dos dados referente aos registradores, é criado um *JPanel* que irá conter as informações com o layout apresentado no listing 6. Após isso é utilizado um simples *JTextField* para conter os 32 campos e a referência desses objetos é eventualmente enviada para a classe que atualiza os dados em tempo real. Também é importante mencionar que o formato do conteúdo de cada registrador é em Hexadecimal e sempre serão completados com o dígito 0 a esquerda, portanto um exemplo de um valor que o registrador pode apresentar é ”0x000000FF”.

Listing 6. Criação do `registerMemoryPanel`

```
JPanel registerMemoryPanel = new JPanel(new GridLayout(32, 2, 5,
    5));
```

4.3. consolePanel

O *consolePanel* foi desenvolvido para apresentar informações para o desenvolvedor sobre o status atual do programa. Essa janela pode ser aberta clicando no menu em *Options* → *Open Console Tab*. Quando o usuário realizar algumas das operações essenciais como: abrir um programa, inicializar um programa, parar um programa ou até quando algum erro ocorrer, irá ser exibido essa informação na janela conforme pode ser visualizado na Figura 20. É importante mencionar que é exibido para o usuário uma lista com as 20 últimas mensagens. Assim que uma nova mensagem for adicionada e a quantia total ultrapassar 20 mensagens a última mensagem da lista será removida. O usuário também pode clicar no botão *Clear* ao lado do título da aba para remover todas as mensagens da listagem. Para fechar a janela, pode ser clicado no mesmo botão de abertura ou no ”X”que fica entre o botão *Clear* e o título *Console*.

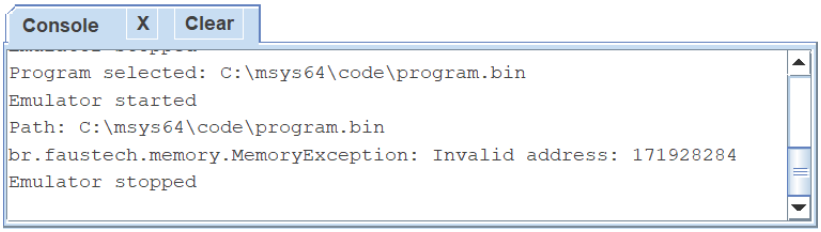

Figura 20. Janela com o conteúdo do console

Similarmente às janelas anteriores, o conteúdo da aba de console só é exibido após a interação do usuário com o botão *Open Console Tab*. Essa aba, apresentada anteriormente na Figura 20, é inicialmente criada com um conteúdo vazio, como demonstrado no Listing 7.

Após a abertura da aba pelo usuário, o console é preenchido por meio de um atributo da interface do tipo *JTextArea*, denominado *consoleTextArea*. Essa variável é responsável por armazenar mensagens que indicam os eventos ocorridos durante a execução do programa. Essas mensagens são adicionadas com a chamada da função ilustrada no Listing 8, que é invocada tanto na interface gráfica (GUI) quanto na simulação da CPU, a fim de exibir possíveis erros ou informações relevantes ao usuário.

Listing 7. Criação do `lowerTabbedPane`

```
lowerTabbedPane = new JTabbedPane();
```

Listing 8. Método `consoleInfo()`

```
public void consoleInfo(String message);
```

Para compor o painel interno da aba de console, é utilizada uma instância da classe *JScrollPane*, a qual recebe como parâmetro o componente de texto responsável pela exibição das mensagens. A utilização do *JScrollPane* permite incorporar barras de rolagem automaticamente, facilitando a navegação e a leitura de conteúdos extensos. A criação desse componente está apresentada no Listing 9.

Listing 9. Criação do `scrollPane`

```
JScrollPane scrollPane = new JScrollPane(consoleTextArea);
```

4.4. actionBar

A *actionBar* é uma barra com botões que foram criados para aprimorar a interação entre desenvolver e os eventos do emulador, no momento existem 2 botões conforme pode ser validado na Figura 21, o *Run* que inicializa a execução aonde o emulador começa a rodar as instruções de código de máquina e *Stop* que finaliza o emulador limpando qualquer estrutura pendente.

Figura 21. Barra de ações

Devido à sua estrutura simplificada, a barra de ações é criada diretamente por meio da chamada de um método que referencia o conteúdo da janela e aplica uma função responsável pela criação dos botões, como ilustrado no Listing 10.

Para que um botão execute uma ação ao ser pressionado, é necessário associar a ele uma função que será invocada nesse evento. Portanto é enviado para o botão que inicializa a emulação e para o botão que finaliza a mesma, as respectivas funções conforme pode ser visualizado no Listing 11.

Nesse caso a função *startEmulator* salva algumas informações no console e chama o método de inicialização do emulador criado na Main. Após essa etapa, o botão *Run* é desabilitado e o botão *Stop* é ativado, de forma a evitar interações indevidas durante a execução. Em contraste, a função *StopEmulator* interage diretamente com as estruturas internas do sistema, incluindo a finalização da execução da CPU, para depois alterar os status dos botões de forma oposta a função anterior.

Listing 10. Criação do `actionBar`

```
getContentPane().add(createActionBar(), BorderLayout.NORTH);
```

Listing 11. Método de callback `startEmulator()` e `stopEmulator()`

```
runButton.addActionListener(e -> startEmulator());
stopButton.addActionListener(e -> stopEmulator());
```

Para facilitar o entendimento da interação entre a interface gráfica e o emulador, detalharemos o funcionamento dos botões de controle da execução. Como mencionado anteriormente, o botão responsável por iniciar a emulação é o *Run*. Uma de suas ações mais relevantes é a desativação de si mesmo e a ativação do botão *Stop*, com o objetivo de evitar interações indevidas durante a execução do programa.

Entretanto, a principal funcionalidade do botão *Run* consiste na chamada de um método de callback, previamente definido na classe *Main*, como ilustrado no Listing 12. É dentro desse método que ocorre a criação dos componentes essenciais do emulador, como o *FrameBuffer*, o *Bus*, entre outros.

Por outro lado, a função associada ao botão *Stop*, denominada *stopEmulator*, realiza o processo inverso em relação ao estado dos botões. Sua etapa mais crítica consiste na interação direta com as estruturas internas do sistema, especialmente com a finalização da execução da CPU, conforme demonstrado no Listing 13.

Listing 12. Chamada do método `onArgsSelected`

```
listener.onArgsSelected(path);
```

Listing 13. Chamada da interrupção da CPU

```
cpu.interrupt();
```

4.5. settingsWindow

Considerando que o objetivo dessa janela é apresentar um menu com possíveis configurações, a *settingsWindow* foi desenvolvida em um formato diferente dos anteriores. Para facilitar a interação do usuário com a janela, foi criada uma classe separada chamada *SettingsWindow* que estende a classe *JDialog*, essa janela pode ser exibida acessando no menu *Options* $\rightarrow$ *Settings* e irá ser exibido as configurações conforme pode ser visualizado na Figura 22. Paralelamente foi desenvolvida uma funcionalidade extra onde o emulador salva as configurações em um arquivo de texto local com os dados no formato JSON, para que no momento em que o emulador for reaberto, as mesmas configurações serão aplicadas. Esse arquivo pode ser visualizado na Figura 23. No momento as configurações existentes são: resolução da janela em que o programa irá rodar, a quantidade de memória do emulador e a opção *Dark Mode*. As opções além do tema escuro estão desativas e foram aplicados os valores padrões.

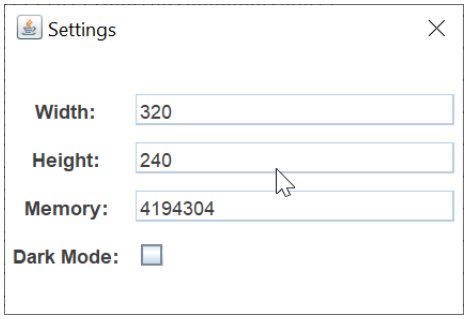

Figura 22. Conteúdo do arquivo de configuração


```
{
  "recentFiles": [
    "C:\\msys64\\code\\color_test.bin",
    "C:\\msys64\\code\\interrupt_test.bin",
    "E:\\Code\\TCC\\emulator\\src\\test\\demos\\interrupt_demo.txt",
    "C:\\msys64\\code\\program.bin",
    "E:\\Code\\TCC\\emulator\\src\\test\\demos\\color_demo.txt"
  ],
  "darkMode": true
}
```


Figura 23. Janela de configuração

Considerando a estrutura especial da janela de configurações, sua instância não é criada durante a inicialização do emulador. A criação desse componente ocorre somente mediante interação do usuário, ao pressionar o botão *Settings*. Nesse momento, a função de callback associada ao botão é responsável por instanciar um objeto da classe correspondente, conforme ilustrado no Listing 14.

Após sua criação, a janela torna-se visível e recebe como parâmetro uma referência à interface principal, passada via construtor. Essa referência é essencial para permitir a comunicação entre a janela de configurações e a interface geral, especialmente no que diz respeito à aplicação do tema visual da GUI.

Embora essa janela não seja instanciada na inicialização do sistema, algumas variáveis relacionadas ao arquivo de configuração são definidas nesse estágio inicial. Dentre elas, destaca-se a variável *recentFiles*, responsável por armazenar o histórico dos últimos 10 programas abertos no emulador. O carregamento dessas informações pode ser observado no trecho de código apresentado no Listing 15.

Listing 14. Criação do `settingsWindow`

```
SettingsWindow settingsWindow = new SettingsWindow(this);
```

Listing 15. Método `loadHistory()`

```
configFile.loadHistory(recentFiles, darkModeEnabled);
```

4.6. export

Uma das funcionalidades extras que foram desenvolvidas para a interface para auxiliar no processo de desenvolvimento, é o botão de exportar o código que pode ser clicado no

menu em *Export* → *Code*. Existem 3 opções nesse sentido que são referentes as colunas do conteúdo da interface, *Hexadecimal Code*, *Binary Code* e *Complete Code*, também é possível selecionar a célula específica e copiar a instrução ou o seu código. Assim como outros botões são definidos, podemos validar no Listing 16 a seção que cria o componente.

Listing 16. Criação do `hexadecimalItem`

```
JMenuItem hexadecimalItem = new JMenuItem("Hexadecimal Code");
hexadecimalItem.addActionListener(e -> exportHexadecimalCode());
```

Após o botão ser pressionado, o texto referente ao mesmo é passado para a área de transferência, podemos visualizar como o texto é retornado com base no Listing 17.

Listing 17. Retorno do código

```
StringSelection selection = new StringSelection(sb.toString());
Toolkit.getDefaultToolkit().getSystemClipboard().setContents(
   selection, null);
```

4.7. sorter

Outra funcionalidade extra desenvolvida, foi a ordenação e barra de filtragem para as instruções decodificadas. Conforme foi apresentado anteriormente, foi utilizado a classe *TableRowSorter* e então definido um campo de filtragem denominado *filterField*. Para exibir esse campo, foi utilizado as classes *InputMap* e *ActionMap* para adicionar o filtro quando o desenvolvedor pressiona ”Control”+ ”F”. Para remover esse filtro, foi criado um simples botão ”X”no final do campo.

Com esses componentes foi aplicado um callback que filtra quando algum texto é escrito no *filterField*, podemos visualizar o método onde a filtragem é aplicada ou removida no Listing 18.

Listing 18. Método `applyFilter`

```
private void applyFilter();
```

5. Desenvolvimento de Programas para a arquitetura RISC-V

Nesta seção, iremos detalhar os passos utilizados no processo de desenvolvimento de um programa na arquitetura RISC-V, assim como apresentar outras ferramentas modernas e acessíveis que auxiliam em cada etapa. A seguir, descrevem-se os principais passos:

5.1. Escrita do Código Fonte

A primeira etapa do desenvolvimento é criar um arquivo com as instruções no formato da arquitetura RISC-V. Isso pode ser feito diretamente ou por meio de ferramentas que compilam o código de uma linguagem para uma arquitetura específica. Neste trabalho, o site Compiler Explorer (godbolt.org) foi utilizado como ferramenta principal para essa etapa, permitindo escrever o programa de teste na linguagem C, compilar para a arquitetura RISC-V e visualizar em tempo real o código assembly gerado.

Conforme pode ser visualizado na Figura 24, há uma aba à esquerda onde podemos escrever o código na linguagem desejada e uma aba à direita onde é possível visualizar o código assembly gerado. Para este trabalho, estamos utilizando a versão específica

”RISC-V (32-bits) gcc 15.1.0”com a flag de otimização -O2”. Outras versões da arquitetura podem ser utilizadas; contudo, a flag de otimização demonstrou ter impactos positivos na eficiência do programa gerado.

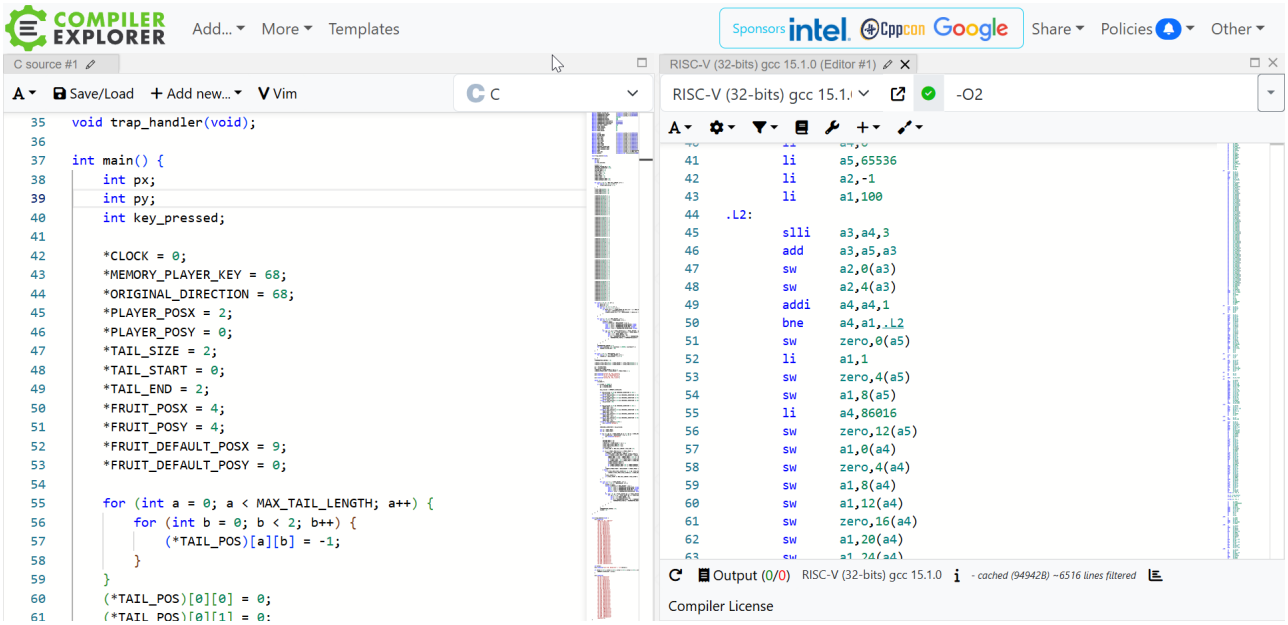

Figura 24. Janela do site Compiler Explorer (godbolt)

5.2. Compilação do Código Assembly em Código de Máquina

A segunda etapa do desenvolvimento consiste em utilizar o código assembly em um compilador para gerar o binário. Para isso, podemos utilizar ferramentas como o *RARS*, já citado. Com esse programa, é possível inserir o código assembly e extrair o binário no formato desejado. No entanto, neste trabalho foram utilizados os ambientes de desenvolvimento *MSYS2* (Minimal System 2) e *UCRT64* (Universal C Runtime).

Esses ambientes foram utilizados por diversos motivos. Os principais são a familiaridade do integrante do trabalho com as ferramentas e o fato de que o sistema operacional utilizado foi o Windows. Devido a problemas técnicos relacionados à virtualização, não foi possível utilizar outros ambientes como o *Windows Subsystem for Linux*. Dessa forma, o ambiente *MSYS2* se mostrou mais estável por não utilizar virtualização, mas sim adaptações de bibliotecas específicas.

Assim, optou-se pelo uso do ambiente *UCRT64* em conjunto com o pacote `mingw-w64-ucrt-x86_64-riscv64-unknown-elf-gcc`, que oferece suporte completo à compilação cruzada para a arquitetura RISC-V. A escolha foi motivada pela melhor compatibilidade com os requisitos do projeto, como, por exemplo, o suporte a instruções específicas como *mret*.

A Listagem 19 apresenta o comando utilizado para chamar sequencialmente o *assembler*, o *linker* e o conversor de binário (*objcopy*), responsáveis por transformar o código assembly em um arquivo binário executável compatível com o emulador desenvolvido.

Listing 19. Comando de Compilação

```
riscv64-unknown-elf-as -march=rv32im_zicsr -mabi=ilp32 -o
   program.o program.s && riscv64-unknown-elf-ld -m elf32lriscv
   -o program.elf program.o && riscv64-unknown-elf-objcopy -O
   binary program.elf program.bin
```

```
Admin@DESKTOP-L78QBUQ UCRT64 /code/demo
$ ls
program.s

Admin@DESKTOP-L78QBUQ UCRT64 /code/demo
$ riscv64-unknown-elf-as -march=rv32im_zicsr -mabi=ilp32 -o program.o program.s && riscv64-unknown-elf
-ld -m elf32lriscv -o program.elf program.o && riscv64-unknown-elf-objcopy -O binary program.elf progr
am.bin

Admin@DESKTOP-L78QBUQ UCRT64 /code/demo
$ ls
program.bin  program.elf  program.o  program.s

Admin@DESKTOP-L78QBUQ UCRT64 /code/demo
$
```

Figura 25. Janela do Ambiente *UCRT64*

5.3. Testando o Programa

Após realizar os passos anteriores, é possível selecionar o programa na interface do emulador, habilitar as janelas de depuração e console, iniciar o programa e acompanhar em tempo real a execução, conforme ilustrado na Figura 26.

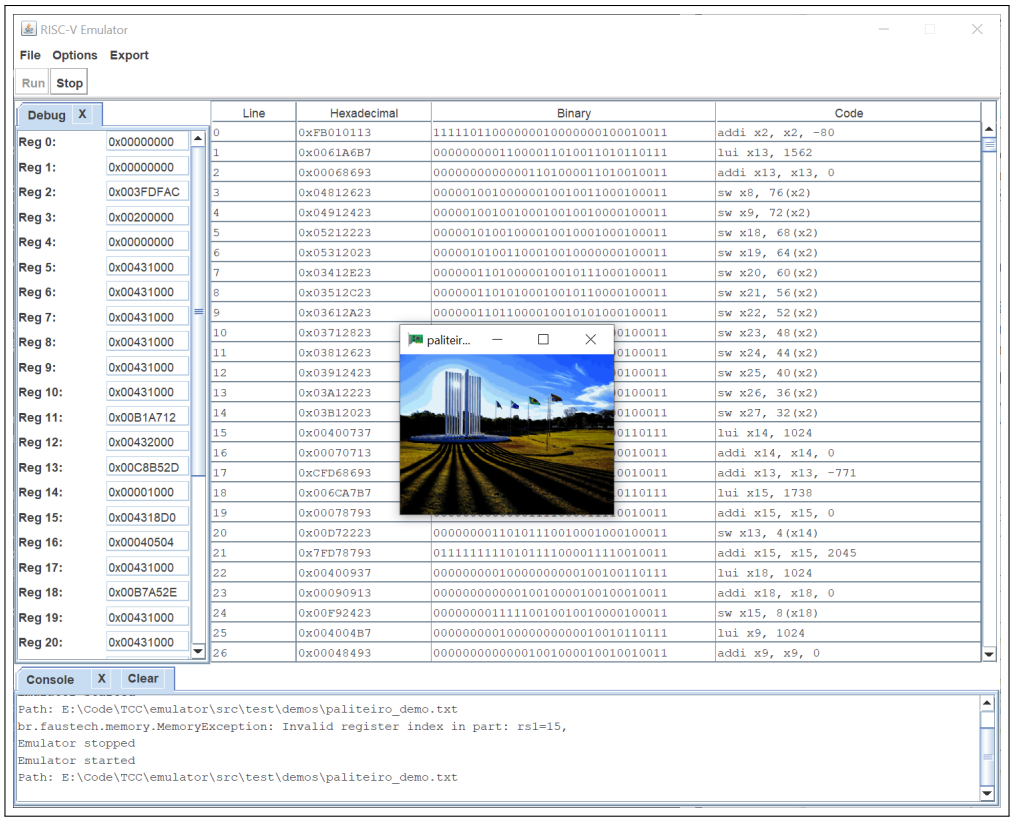

Figura 26. Emulando o Programa

6. Demonstrações

Nesta seção, será apresentado o principal programa de teste desenvolvido até o momento. Utilizando as ferramentas mencionadas anteriormente e as funcionalidades do próprio emulador, foi desenvolvido um jogo simples do tipo ”Snake”na linguagem C. Foram aplicadas também algumas técnicas para resolver problemas encontrados e melhorar a estabilidade do programa. A seguir, descrevem-se alguns aspectos relevantes da demonstração:

6.1. Variáveis na Memória

Inicialmente, foram utilizadas variáveis estáticas no formato em C. No entanto, ao longo dos testes, foi percebida uma maior instabilidade com esse formato. Por isso, algumas informações mais críticas foram separadas e definidas utilizando posições específicas na memória, conforme o formato apresentado na Listagem 20.

Para trabalhar com essas variáveis, é preciso definir uma posição específica dentro da memória total, que atualmente equivale a 4194304 bytes, ou 0x00400000. Após isso, é possível acessá-las diretamente, de forma semelhante a um ponteiro, utilizando o nome definido com a diretiva *#define*. Esse funcionamento pode ser verificado na Listagem 21.

Listing 20. Definição de variáveis na memória

```
#define MEMORY_PLAYER_KEY ((volatile uint32_t *)0x00151000)
#define CLOCK             ((volatile uint32_t *)0x00151004)
#define PLAYER_POSX       ((volatile uint32_t *)0x00151008)
#define PLAYER_POSY       ((volatile uint32_t *)0x0015100C)
```

Listing 21. Acesso de variáveis na memória

```
*CLOCK = 0;
*MEMORY_PLAYER_KEY = 68;
*PLAYER_POSX = 2;
*PLAYER_POSY = 0;
```

6.2. Geração de Fruta

O objetivo do jogo é capturar blocos vermelhos que representam ”frutas”e após o jogador conseguir alcançá-la, o sprite do personagem (a cobra) aumentaria de tamanho em 1 unidade. Para tornar o jogo mais dinâmico, é preciso posicionar a ”fruta”de forma ”aleatória”no tabuleiro e garantir que a posição é correta, ou seja, a fruta não é colocada na mesma posição do jogador.

Considerando as dificuldades técnicas das funcionalidades e a dificuldade de gerar números ”pseudo-aleatórios”, foram aplicados cálculos matemáticos que consideram valores rotativos e a posição do jogador conforme pode ser visualizado no Listing 23

Listing 22. Definição de variáveis referentes a ”fruta”

```
#define FRUIT_POSX         ((volatile uint32_t *)0x00151024)
#define FRUIT_POSY         ((volatile uint32_t *)0x00151028)
#define FRUIT_DEFAULT_POSX ((volatile uint32_t *)0x00151030)
#define FRUIT_DEFAULT_POSY ((volatile uint32_t *)0x00151034)
```

```
*FRUIT_POSX = 6;
*FRUIT_POSY = 4;
*FRUIT_DEFAULT_POSX = 9;
*FRUIT_DEFAULT_POSY = 0;
```

Listing 23. Geração de posição da ”fruta”

```
while ((*GRID)[*FRUIT_POSY * GRID_HEIGHT + *FRUIT_POSX] != 0) {
 *FRUIT_POSX = (px + *FRUIT_POSX) * 2 + 3;
 if (*FRUIT_POSX >= GRID_WIDTH) {
  *FRUIT_POSX -= GRID_WIDTH;
  if (*FRUIT_POSX >= GRID_WIDTH) {
   *FRUIT_POSX -= GRID_WIDTH;
  }
 }
 *FRUIT_POSY = (py + *FRUIT_POSY) * 2 + 2;
 if (*FRUIT_POSY >= GRID_HEIGHT) {
  *FRUIT_POSY -= GRID_HEIGHT;
  if (*FRUIT_POSY >= GRID_HEIGHT) {
   *FRUIT_POSY -= GRID_HEIGHT;
  }
 }
 if (*FRUIT_POSX < 0 || *FRUIT_POSX >= GRID_WIDTH) *FRUIT_POSX =
     *FRUIT_DEFAULT_POSX;
 if (*FRUIT_POSY < 0 || *FRUIT_POSY >= GRID_HEIGHT) *FRUIT_POSY
    = *FRUIT_DEFAULT_POSY;
 (*FRUIT_DEFAULT_POSX)++;
 (*FRUIT_DEFAULT_POSY)++;
 if (*FRUIT_DEFAULT_POSX < 0 || *FRUIT_DEFAULT_POSX >=
    GRID_WIDTH) *FRUIT_DEFAULT_POSX = 0;
 if (*FRUIT_DEFAULT_POSY < 0 || *FRUIT_DEFAULT_POSY >=
    GRID_WIDTH) *FRUIT_DEFAULT_POSY = 0;
};
(*GRID)[*FRUIT_POSY * GRID_HEIGHT + *FRUIT_POSX] = 2;
```

6.3. Gerenciamento da Cauda

No jogo, o jogador é representado por uma posição principal (a ”cabeça”da cobra) e por uma lista de posições que compõem a ”cauda”. Inicialmente, foi considerada a manipulação constante de todas as posições da cauda, mas essa abordagem mostrou-se rapidamente ineficiente.

A estrutura atual é gerenciada da seguinte maneira: existe uma lista capaz de armazenar até 100 posições da cauda (valor definido por `MAX_TAIL_LENGTH`). Há também duas variáveis que indicam a posição inicial e final da cauda. A posição final é sempre atualizada com a nova posição da cabeça, enquanto a posição inicial é utilizada para liberar espaço caso a cabeça não encontre uma fruta.

Listing 24. Definição da cauda

```
#define TAIL_POS    ((volatile uint32_t (*)[MAX_TAIL_LENGTH][2])0
   x00160000)
```

```
#define TAIL_START ((volatile uint32_t *)0x0015101C)
#define TAIL_END   ((volatile uint32_t *)0x00151020)
```

Listing 25. Utilização da posição final da cauda

```
(*TAIL_POS)[*TAIL_END][0] = px;
(*TAIL_POS)[*TAIL_END][1] = py;
(*TAIL_END)++;
if (*TAIL_END >= MAX_TAIL_LENGTH) *TAIL_END = 0;
```

Listing 26. Utilização da posição inicial da cauda

```
if ((*TAIL_POS)[*TAIL_START][0] != -1 && (*TAIL_POS)[*TAIL_START
   ][1] != -1) {
 (*GRID)[(*TAIL_POS)[*TAIL_START][1] * GRID_HEIGHT + (*TAIL_POS)
    [*TAIL_START][0]] = 0;
}
(*TAIL_START)++;
if (*TAIL_START >= MAX_TAIL_LENGTH) *TAIL_START = 0;
```

6.4. Píxeis da Tela

Para melhorar a visualização, foi definido um grid com base na resolução original da tela, que é 320x240. Esse grid foi dividido em uma matriz de 10x10, fazendo com que cada ”píxel lógico”represente uma área de 32x24 píxeis reais. Assim, é necessário calcular a posição correta de cada área na memória da GPU para que a imagem seja apresentada corretamente.

A matriz possui valores que indicam o tipo de conteúdo: 0 para área vazia (preto), 1 para jogador (branco) e 2 para fruta (vermelho).

Listing 27. Definição da Grid

```
#define FRAMEBUFFER_CONTROL         ((volatile uint32_t *)0
   x00400000)
#define FRAMEBUFFER_START           ((volatile uint32_t *)0
   x00400004)
#define FRAMEBUFFER_WIDTH  320
#define FRAMEBUFFER_HEIGHT 240
#define PIXEL_WIDTH        32
#define PIXEL_HEIGTH       24
#define GRID_WIDTH         10
#define GRID_HEIGHT        10
#define GRID               ((volatile uint32_t (*)[GRID_WIDTH *
   GRID_HEIGHT])0x00155000)
```

Listing 28. Manipulação da memória da GPU para exibição da imagem

```
for (int j = 0; j < GRID_HEIGHT; j++) {
 for (int i = 0; i < GRID_WIDTH; i++) {
  uint32_t color;
  switch ((*GRID)[j * GRID_HEIGHT + i]) {
```

```
    case 0: color = FRAMEBUFFER_COLOR_BLACK; break;
    case 1: color = FRAMEBUFFER_COLOR_WHITE; break;
    case 2: color = FRAMEBUFFER_COLOR_RED; break;
    default: color = FRAMEBUFFER_COLOR_BLACK; break;
   }
   for (py = 0; py < PIXEL_HEIGTH && j * PIXEL_HEIGTH + py <
      FRAMEBUFFER_HEIGHT; py++) {
    for (px = 0; px < PIXEL_WIDTH && i * PIXEL_WIDTH + px <
       FRAMEBUFFER_WIDTH; px++) {
     int x = i * PIXEL_WIDTH + px;
     int y = j * PIXEL_HEIGTH + py;
     if ((y * FRAMEBUFFER_WIDTH + x) < FRAMEBUFFER_END) {
      FRAMEBUFFER_START[y * FRAMEBUFFER_WIDTH + x] = color;
     }
    }
   }
  }
}
*FRAMEBUFFER_CONTROL = 0;
```

6.5. Interrupção do teclado

Essa funcionalidade está relacionada à implementação do emulador RISC-V usado [de Freitas Silva et al. 2024]. Para utilizá-la em C, é necessário usar a instrução *asm volatile()* para inserir diretamente os comandos de baixo nível no código gerado.

Para configurar as interrupções, é preciso:

- Desabilitar a flag global (CSR 772);
- Atribuir à CSR 773 o endereço da função de interrupção;
- Reabilitar a flag global.

A função que será chamada pode ser escrita normalmente em C. Contudo, é preciso realizar a troca de contexto manualmente (salvar e restaurar os registradores) e, ao final, utilizar a instrução *mret* para retornar do modo de interrupção.

Dentro da função, é possível acessar a tecla pressionada lendo a posição de memória 835 e, com base no valor, atualizar o estado do jogo.

Listing 29. Configuração da interrupção

```
asm volatile("csrrwi x5, 772, 0\n\t");
asm volatile("la x7, trap_handler\n\t"
             "csrrw x5, 773, x7\n\t");
asm volatile("csrrwi x5, 772, 1\n\t");
```

Listing 30. Função de callback da interrupção

```
void trap_handler(void) {
 asm volatile(
  "addi x2, x2, -124\n\t"
  "sw x0, 0(x2)\n\t"
```

```
  "sw x1, 4(x2)\n\t"
  ...
  "sw x30, 120(x2)\n\t"
  "sw x31, 124(x2)\n\t");

 int mtval;
 asm volatile("csrr %0, mtval\n\t" : "=r"(mtval));

 if (mtval == 87 || mtval == 65 || mtval == 83 || mtval == 68 ||
     mtval == 256) {
  *MEMORY_PLAYER_KEY = mtval;
 }

 asm volatile(
  "lw x0, 0(x2)\n\t"
  "lw x1, 4(x2)\n\t"
  ...
  "lw x30, 120(x2)\n\t"
  "lw x31, 124(x2)\n\t"
  "mret\n\t");
}
```

6.6. Demonstração

A seguir é apresentado duas figuras referentes ao jogo em execução. Em uma delas o jogador está andando para baixo até uma ”fruta”e na outra o jogador está indo para direita na direção de uma ”fruta”.

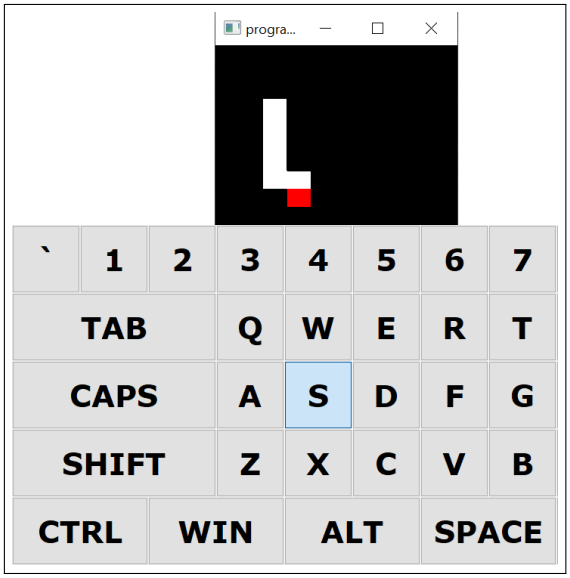

Figura 27. Jogo em andamento com a tecla ”S”pressionada

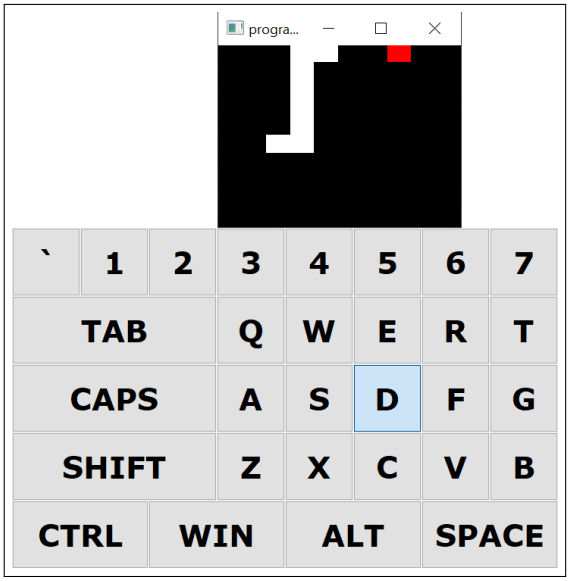

Figura 28. Jogo em andamento com a tecla ”D”pressionada

7. Trabalhos Futuros

Ao longo do planejamento e desenvolvimento deste trabalho, foram consideradas diversas funcionalidades adicionais. No entanto, devido à limitação de tempo, elas foram retiradas do escopo atual. Assim, nesta seção são apresentados alguns dos possíveis aprimoramentos ou componentes como sugestões para trabalhos futuros.

Algumas funcionalidades, como o tratamento de interrupções por teclado ou a visualização da tela em nível de pixels, exigem que certas instruções sejam implementadas diretamente no código assembly, o que pode ser desnecessariamente complexo neste momento. Uma possível solução seria o desenvolvimento de bibliotecas específicas que encapsulem essas funcionalidades. Além disso, recursos úteis como a impressão de texto no terminal por meio de instruções dedicadas também não foram abordados neste projeto e podem ser considerados em futuras extensões.

Projetos mais avançados incluem a reimplementação do emulador em outras linguagens, como C++ ou Python. No entanto, isso exigiria adaptações significativas, especialmente na integração com bibliotecas gráficas e de interface.

Outra melhoria relevante seria a implementação de um sistema de depuração (*debugger*) com suporte a *breakpoints*, permitindo a execução passo a passo do programa e a inspeção do estado interno do emulador durante a simulação.

8. Conclusão

A interface gráfica desenvolvida preserva todas as funcionalidades do emulador da arquitetura RISC-V original, ao mesmo tempo em que amplia significativamente a capacidade de desenvolvimento. Isso se reflete na simplificação de etapas como a visualização das instruções decodificadas, a inicialização de programas, o acompanhamento dos valores nos registradores, a identificação de erros durante a execução e a alternância entre diferentes programas em tempo de simulação.

O principal objetivo deste trabalho foi analisar e implementar funcionalidades que aprimorassem a usabilidade do emulador da arquitetura RISC-V. Para isso, foi desenvolvida uma GUI (Interface Gráfica de Usuário) integrada ao sistema, além da criação de novos programas de teste voltados à validação e demonstração das capacidades técnicas do projeto. Em um primeiro momento, houve a intenção de iniciar o desenvolvimento de um segundo emulador, utilizando a linguagem C++ e aplicando técnicas modernas de engenharia de software. No entanto, considerando a complexidade da proposta e as limitações de tempo, optou-se por concentrar os esforços nos objetivos essenciais definidos inicialmente.

Essa reorientação permitiu dedicar atenção integral à construção e aprimoramento da interface gráfica, bem como à elaboração de programas de teste mais complexos, como, por exemplo, a implementação de uma versão do jogo Snake. Dessa forma, o trabalho final não apenas apresenta funcionalidades que ilustram a versatilidade e robustez do emulador, como também contribui para a melhoria do fluxo de desenvolvimento, oferecendo uma base sólida e extensível para futuras pesquisas ou expansões do sistema.

Referências

Botwright, R. (2024). *Java Swing Programming*. Rob Botwright.

de Freitas Silva, G. F., de Melo Roberto, J. P., and Leite, R. B. A. (2024). Implementac¸ao de emulador funcional de sistema computacional baseado em arquitetura risc-v. Disponível em: https://repositorio.ufms.br/handle/123456789/10402.

Deitel, P. and Deitel, H. (2016). *Java: Como Programar*. Pearson Education do Brasil, 10 edition.

Landers, B. (2024). Rars - risc-v assembler and runtime simulator. https://github.com/TheThirdOne/rars. Acessado em: 18 junho 2025.

Marc Loy, Robert Eckstein, D. W. (2002). *Java Swing*. O'Reilly Media, Inc., 2th edition.

Patterson, D. A. and Hennessy, J. L. (2014). *Organização e Projeto de Computadores: A Interface Hardware/Software*. LTC, Rio de Janeiro, 5 edition.

Vollmar, K. and Kidd, P. (2006). Mars: An education-oriented mips assembler and runtime simulator. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, pages 239–243. ACM.

Glossário

CPU Unidade Central de Processamento. 7, 16, 18, 19

GPU Unidade de Processamento Gráfico. 26

GUI Interface Gráfica do Usuário. 1, 4, 7, 13, 14, 16, 18, 20, 29

RISC-V Arquitetura de conjunto de instruções baseada no paradigma RISC. 1–6, 13, 15, 21, 22, 27, 29

Swing Biblioteca de componentes para interfaces gráficas em Java. 1, 2, 4, 5, 7, 13