UNIVERSIDADE DE SÃO PAULO

Escola Politécnica

Graduação em Engenharia de Computação

VisuARM

PCS3732 - Laboratório de Processadores

Julia Castro Nunes Terrigno - 13679673

Juliana Mitie Hosne Nakata - 13679544

Tássyla Lissa Lima - 13684471

São Paulo,

16 de Agosto de 2025

Sumário

1. Motivação

O estudo de arquitetura de computadores e processadores representa, muitas vezes, um desafio para os estudantes que estão tendo seus contatos iniciais com a matéria. Durante a disciplina PCS3732, uma das dificuldades das participantes do grupo foi visualizar e compreender em detalhes como o código em linguagem de alto nível se transforma em instruções de máquina, bem como a forma como esse código é executado internamente pelo processador. Com essa dificuldade em vista, o grupo buscou formas alternativas de compreender em profundidade a matéria de forma visual e prática. No entanto, existem poucas ferramentas que conseguem atingir esses objetivos.

Diante desse cenário, surgiu a ideia de desenvolver uma ferramenta que auxilia na visualização e no entendimento dos processos envolvidos, tornando a aprendizagem mais intuitiva e prática. O projeto **VisuARM** foi concebido com a finalidade de permitir ao estudante observar, de maneira interativa, a relação entre o código escrito em C, o Assembly gerado em diferentes níveis de otimização e a execução desse código em nível de máquina. Além disso, a possibilidade de simular o uso de memória (stack, heap, dados e instruções) contribui para que conceitos antes teóricos passem a ter uma representação visual clara, favorecendo a fixação do conteúdo. A inspiração inicial veio de experiências anteriores com ferramentas como o **ARM Click & Disasm**, que demonstraram o potencial de recursos interativos para o ensino.

2. Introdução

O projeto VisuARM é uma plataforma didática interativa, desenvolvida como uma Single-Page Application (SPA), voltada ao ensino de conceitos de arquitetura de processadores ARM. Ele permite que o usuário escreva programas em C e visualize o código Assembly correspondente. O VisuARM tem a capacidade de comparar visualmente o Assembly gerado com diferentes flags de otimização (como -O0 e -O2), evidenciando as estratégias do compilador para melhorar a performance. Também há uma aba que exibe a tabela de símbolos e identifica a seção de memória que cada uma está localizada.

Além disso, a plataforma disponibiliza os principais arquivos gerados pelo processo de compilação para um ambiente bare-metal (como .elf, .map e .s), aproximando o estudante do funcionamento real do processador sem a abstração de um sistema operacional. Essa

abordagem prática complementa o aprendizado teórico e contribui para o desenvolvimento de uma compreensão sólida sobre como softwares interagem diretamente com o hardware.

3. Desenvolvimento

O desenvolvimento do projeto VisuARM teve início a partir da definição das funcionalidades centrais que a plataforma deveria oferecer. Primeiramente, foi estabelecida a necessidade de criar uma versão mínima viável, capaz de compilar código escrito em C e exibir o Assembly correspondente. Essa etapa foi fundamental para validar o fluxo básico de comunicação entre front-end e back-end, confirmando que o compilador poderia ser integrado de maneira funcional.

A partir dessa estrutura inicial, foi possível implementar melhorias na interface. O grupo optou por utilizar **React** com **Vite** para a construção do front-end, a fim de garantir maior desempenho no carregamento e facilidade de modularização dos componentes. Para o back-end, foi utilizado **Python**.

3.1. Arquitetura da Aplicação

O VisuARM foi projetado com uma arquitetura cliente-servidor para separar as responsabilidades e otimizar a performance. O backend foi desenvolvido em Python utilizando o microframework Flask. Ele é responsável por toda a lógica de compilação e simulação, expondo uma API REST com um endpoint principal (/compile). Este endpoint recebe o código C, o compila utilizando a toolchain ARM GCC (arm-none-eabi-gcc), e em caso de sucesso, utiliza o framework de emulação **Unicorn** para executar o binário *.elf* gerado. O resultado, incluindo o Assembly, os artefatos da compilação e o traço da simulação, é serializado em JSON e retornado ao cliente.

O frontend é uma Single-Page Application (SPA) construída com React e Vite, o que garante uma experiência de usuário rápida e responsiva. A interface, estilizada com Tailwind CSS, é organizada em um sistema de abas:

- **Código**: Utiliza o Monaco Editor para oferecer uma experiência de edição de código.
- **Assembly & Otimização**: Emprega a biblioteca react-diff-view para renderizar uma comparação visual e intuitiva entre os códigos Assembly.
- **Tabela de Símbolos**: Gerencia o estado da simulação, permitindo a animação passo a passo e destacando as alterações nos registradores a cada instrução.

- **Arquivos Gerados**: Apresenta os resultados da compilação de forma organizada, permitindo a análise e o download.

3.2. Compilação do código C para Assembly

A ideia da tela principal era apresentar a estrutura de um editor de texto, em que o usuário adiciona o código em C que deve ser compilado. Para isso, foi utilizado o Monaco Editor, da Microsoft, que trouxe ao projeto uma experiência semelhante ao Visual Studio Code, tanto em termos de aparência quanto de funcionalidades básicas de edição. Desse modo, foi possível tornar a escrita de código mais intuitiva e próxima de um ambiente real de desenvolvimento, como mostra a Figura 1:

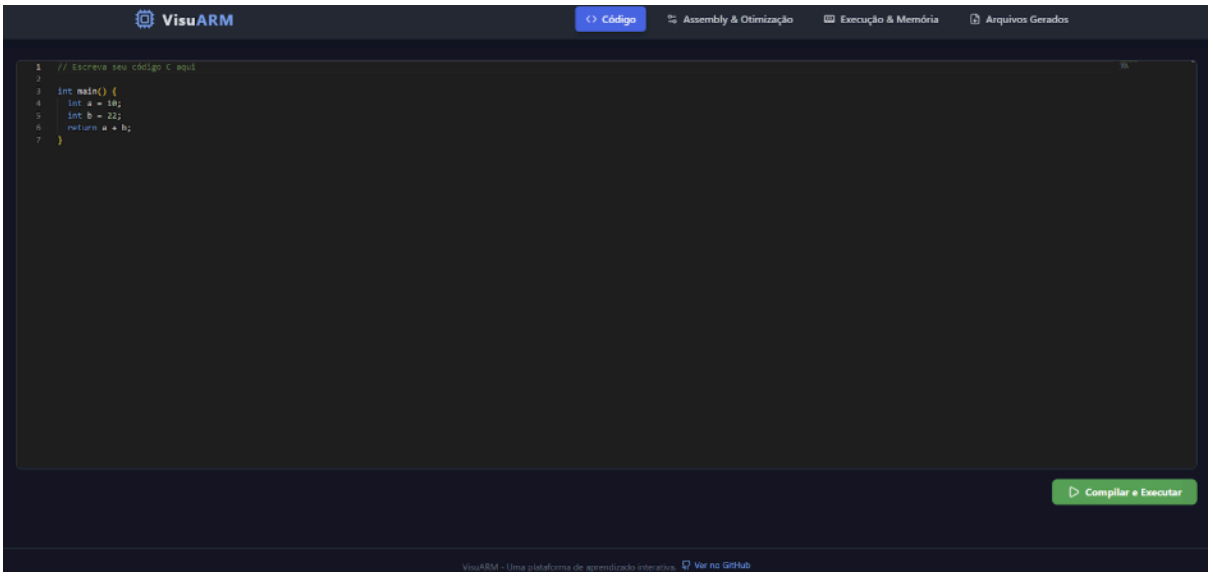

Figura 1: Aba para inserção de código C

Ao clicar em "Compilar e Executar", o código é enviado ao backend. O servidor anexa um pequeno startup code que define o ponto de entrada (_start) e chama a função main, simulando um ambiente sem sistema operacional. O código é então compilado com arm-none-eabi-gcc para a arquitetura Cortex-M3.

Na sequência, foram adicionadas novas abas à aplicação. A seção de comparação, denominada **Assembly & Otimização** (Figura 2), permite observar os resultados da compilação sob diferentes níveis de otimização (-O0, -O1, -O2, -O3), possibilitando que o estudante compreenda como o compilador modifica as instruções de acordo com o objetivo de desempenho ou economia de recursos.

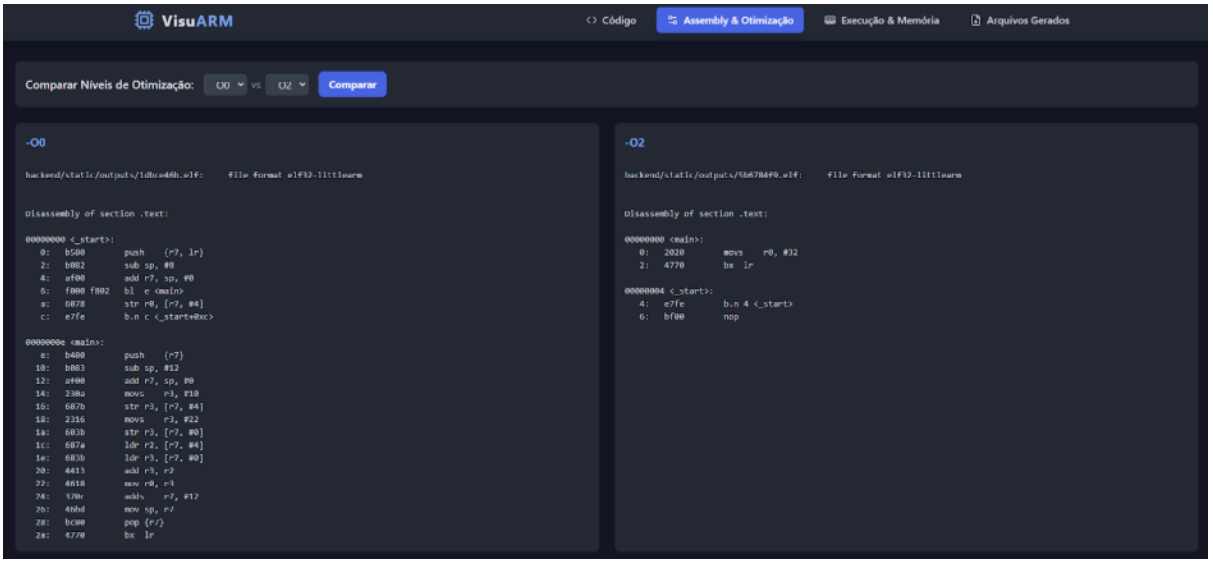

Figura 2: Aba para a comparação de códigos Assembly com diferentes otimizações

Nessa página, o usuário pode escolher quais otimizações ele quer visualizar e os respectivos códigos aparecerão lado a lado, facilitando a comparação.

3.3. Tabela de Símbolos Interativa

A aba da **Tabela de Símbolos** (Figura 3) oferece uma visão detalhada de como o programa compilado é organizado na memória do processador. Após a compilação, o backend executa um parser customizado que analisa o arquivo *.map* gerado pelo linker. Este parser extrai informações cruciais sobre cada seção do programa, como .text (código), .data (dados inicializados) e .bss (dados não inicializados), incluindo seus endereços de início e tamanhos.

Esses dados são enviados ao frontend, que renderiza uma visualização interativa composta por uma tabela detalhada que lista todos os símbolos (nomes de funções e variáveis globais) extraídos do arquivo de mapa. O usuário pode pesquisar e filtrar os símbolos para encontrar rapidamente o endereço e o tamanho de uma função ou variável específica.

Essa funcionalidade permite que o estudante entenda a estrutura de um programa executável e visualize como o código e os dados são mapeados no espaço de endereçamento do hardware, um conceito fundamental no desenvolvimento de sistemas embarcados.

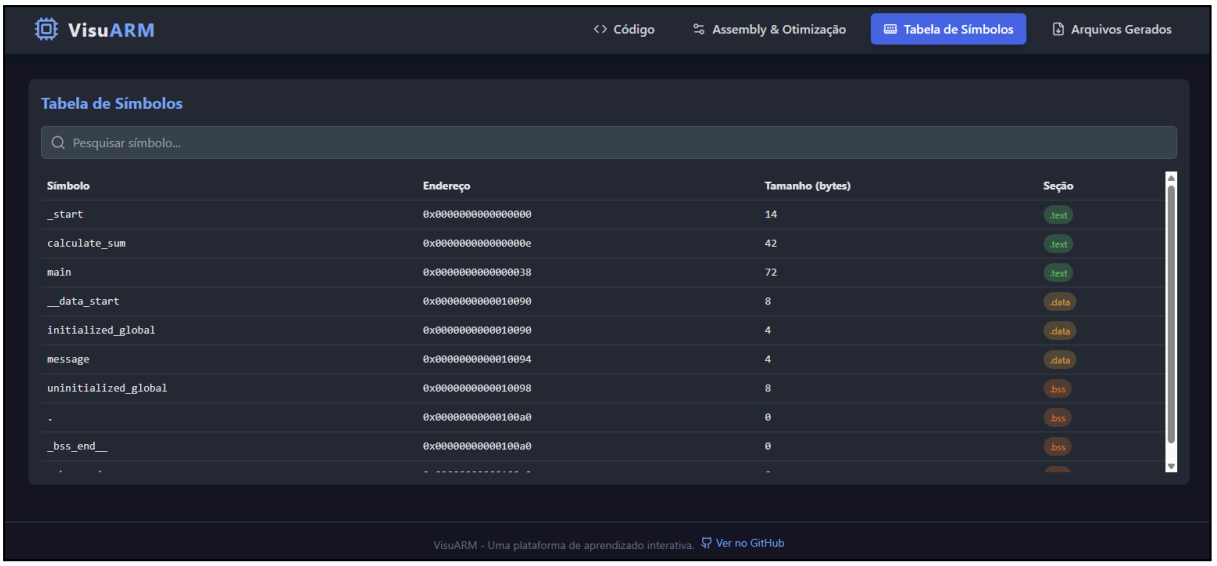

Figura 3: Aba que exibe a Tabela de Símbolos

3.4. Arquivos da compilação

A aba de **Arquivos Gerados** (Figura 4) foi implementada para listar e disponibilizar para download os artefatos resultantes da compilação, como .elf, .s e .map. Além de reforçar o caráter prático da ferramenta, essa funcionalidade aproxima o estudante do fluxo de desenvolvimento bare-metal, permitindo que os arquivos possam ser utilizados em outros ambientes de teste. Abaixo, encontra-se um detalhamento sobre cada um dos arquivos gerados:

- **.elf** (Executable and Linkable Format): O arquivo binário final que contém o código executável, dados e metadados.
- **.s** (Assembly): O código Assembly gerado pelo objdump, permitindo uma análise textual das instruções.
- **.map** (Map File): Um arquivo de texto que descreve como o linker alocou as seções de código e dados na memória.

  Além desses, o código .c original também fica disponível para download.

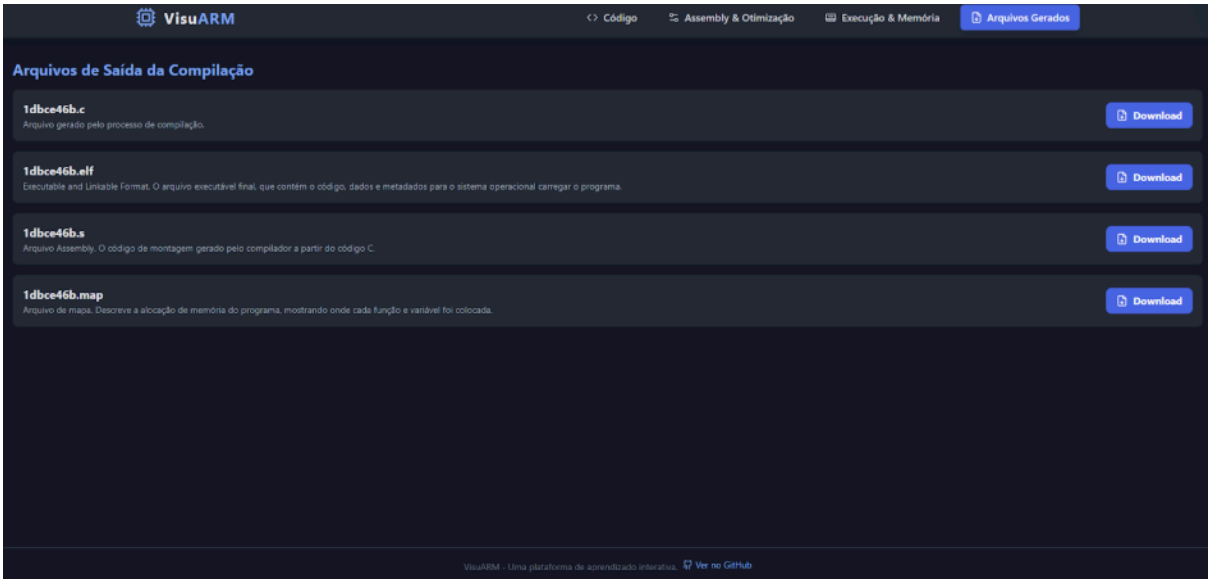

Figura 4: Aba de arquivos gerados

4. Tutorial de Instalação e Execução

4.1. Pré-requisitos

- **Git:** Para clonar o repositório. Instruções de instalação.
- **Python (versão 3.8 ou superior):** Para executar o backend. Instruções de instalação.
- **Node.js (versão 18 ou superior) e npm:** Para executar o frontend. Instruções de instalação.
- **ARM GCC Toolchain (arm-none-eabi-gcc):** Essencial para a compilação do código C para ARM.

  → **Linux/WSL (Ubuntu/Debian):**

```
sudo apt update
sudo apt install gcc-arm-none-eabi
```

4.2. Clonando o Repositório

Clone o repositório do VisuARM do GitHub para a sua máquina local. Abra um terminal e execute o comando abaixo:

```
git clone https://github.com/tassyla/VisuARM.git
```

4.3. Configurando e Executando o Backend (Servidor)

- Navegue até a pasta do backend: *cd backend*
- Crie e ative um ambiente virtual:

```
python3 -m venv venv
source venv/bin/activate
```

- Instale as dependências Python:

```
pip install Flask Flask-CORS pyelftools
```

- Inicie o servidor backend:

```
python3 app.py
```

Se tudo ocorrer bem, você verá uma mensagem indicando que o servidor Flask está rodando, geralmente em *http://127.0.0.1:5000*. Deixe este terminal aberto.

4.4. Configurando e Executando o Frontend

Abra um segundo terminal e siga estes passos:

- Navegue até a pasta do frontend (a partir da raiz do projeto):

```
cd frontend
```

- Instale as dependências do Node.js: Este comando lerá o arquivo package.json e baixará todas as bibliotecas necessárias (React, Monaco Editor, etc.).

```
npm install
```

- Inicie o servidor de desenvolvimento do frontend:

```
npm run dev
```

O terminal mostrará uma mensagem indicando que o servidor de desenvolvimento está rodando e fornecerá uma URL local, geralmente http://localhost:5173.

4.5. Acessando a Aplicação

Com os dois servidores (backend e frontend) em execução, abra seu navegador de internet e acesse a URL fornecida pelo servidor do frontend:

```
http://localhost:5173
```

A aplicação VisuARM deve carregar, e você estará pronto para escrever, compilar e analisar seu código C. Para parar a aplicação, pressione Ctrl + C em cada um dos terminais. Para sair do ambiente virtual do Python, digite deactivate.

5. Conclusão

O projeto VisuARM atingiu os objetivos propostos de desenvolver uma ferramenta didática para o ensino de arquitetura de processadores ARM, permitindo a análise integrada do ciclo de compilação e execução de programas em C. A plataforma viabilizou a visualização do código Assembly em diferentes níveis de otimização, a exploração das regiões de memória e o acesso aos artefatos gerados pelo processo de compilação, como arquivos .elf, .s e .map.

A utilização de tecnologias como React com Vite no front-end e Python no back-end garantiu modularidade, desempenho e flexibilidade na implementação, possibilitando a integração de funcionalidades interativas, como a comparação de códigos Assembly e a representação gráfica da memória. Tais recursos contribuem para uma compreensão mais aprofundada das transformações realizadas pelo compilador e da forma como o software interage com o hardware em um ambiente bare-metal.

Os resultados obtidos demonstram que o VisuARM é capaz de suprir lacunas presentes em ferramentas existentes, fornecendo um ambiente unificado para visualização do fluxo completo entre código-fonte e execução em nível de máquina. Como trabalhos futuros, destacam-se a implementação da execução passo a passo e a simulação dinâmica, visando ampliar ainda mais a aplicabilidade didática da plataforma. Finalmente, o projeto consolidou-se como uma abordagem técnica, interativa e alinhada às necessidades pedagógicas da disciplina.