Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Shell interativo com carregador para RISC-V em FPGA

Luiz Carlos da Silva Néto Vartuli

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Marcus Vinicius Lamar

Brasília
2025

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenador: Prof. Dr. Marcelo Grandi Mandelli

Banca examinadora composta por:

Prof. Dr. Marcus Vinicius Lamar (Orientador) — CIC/UnB
Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo — CIC/UnB
Prof. Dr. Ricardo Pezzuol Jacobi — CIC/UnB
Prof. Dr. Marcelo Grandi Mandelli — CIC/UnB

CIP — Catalogação Internacional na Publicação

Vartuli, Luiz Carlos da Silva Néto.

Shell interativo com carregador para RISC-V em FPGA / Luiz Carlos da Silva Néto Vartuli. Brasília : UnB, 2025.
10 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2025.

1. RISC-V, 2. FPGA, 3. RS232, 4. shell, 5. loader

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Shell interativo com carregador para RISC-V em FPGA

Luiz Carlos da Silva Néto Vartuli

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Marcus Vinicius Lamar (Orientador)
CIC/UnB

Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo
CIC/UnB

Prof. Dr. Ricardo Pezzuol Jacobi
CIC/UnB

Prof. Dr. Marcelo Grandi Mandelli
CIC/UnB

Prof. Dr. Marcelo Grandi Mandelli
Coordenador do Bacharelado em Ciência da Computação

Brasília, 03 de julho de 2025

Dedicatória

Dedico esse trabalho à minha família, que sempre incentivou meus estudos; à minha parceira, Isabel Pierre Starling, que me apoiou durante o fim desta jornada; aos meus amigos, que me acompanharam durante várias partes da minha vida; e aos meus professores, que fomentaram minha curiosidade.

Agradecimentos

Agradeço ao meu orientador, Prof. Dr. Marcus Vinicius Lamar, por tirar todas as minhas dúvidas e prover as ferramentas necessárias para este trabalho.
Agradeço à minha parceira por todas as releituras do trabalho.
Finalmente, agradeço aos meus pais pela revisão gramatical.

Resumo

Este trabalho apresenta o desenvolvimento de um *shell* para um processador RISC-V implementado em FPGA, com o objetivo de possibilitar o carregamento e a execução de programas externos por meio de uma interface serial RS232 com um computador. A proposta visa substituir o método tradicional de carregamento pelo *In-System Memory Content Editor* do Intel Quartus® Prime, oferecendo uma solução mais rápida e automatizada. O sistema implementa uma interface textual baseada no modelo REPL, com comandos básicos (`echo`, `clear`, `help`, `exit`) e suporte ao comando `exec`, responsável por carregar programas e dados de arquivos binários diretamente na memória do FPGA. As chamadas de sistema de arquivos foram implementadas de forma simplificada, apenas utilizando comunicação serial para acessar arquivos no computador. Os testes demonstraram a funcionalidade completa do *shell*, incluindo tratamento de erros, *scroll* automático da tela e compatibilidade com o simulador RARS.

Palavras-chave: RISC-V, FPGA, RS232, shell, loader

Abstract

This article presents the development of a shell for a RISC-V processor implemented on an FPGA, with the aim of enabling the loading and execution of external programs via an RS232 serial interface connected to a computer. The solution replaces the traditional loading method using the In-System Memory Content Editor tool from Intel Quartus® Prime, providing a faster and more automated alternative. The system offers a text interface based on the REPL model, with basic commands (`echo`, `clear`, `help`, `exit`) and support for the `exec` command, which loads code and data from binary files directly into the FPGA's memory. File system calls were implemented by simply using serial communication to access files on the connected computer. Testing confirmed the shell's full functionality, including error handling, automatic screen scrolling, and compatibility with the RARS simulator.

Keywords: RISC-V, FPGA, RS232, shell, loader

Shell interativo com carregador para RISC-V em FPGA

Luiz Carlos da Silva Néto Vartuli
Departamento de Ciência da Computação
Universidade de Brasília
Brasília, Brazil
luizcsnvartuli@gmail.com

Marcus Vinicius Lamar
Departamento de Ciência da Computação
Universidade de Brasília
Brasília, Brazil
lamar@unb.br

***Resumo*—Este trabalho apresenta o desenvolvimento de um *shell* para um processador RISC-V implementado em FPGA, com o objetivo de possibilitar o carregamento e a execução de programas externos por meio de uma interface serial RS232 com um computador. A proposta visa substituir o método tradicional de carregamento pelo *In-System Memory Content Editor* do Intel Quartus® Prime, oferecendo uma solução mais rápida e automatizada. O sistema implementa uma interface textual baseada no modelo REPL, com comandos básicos (`echo`, `clear`, `help`, `exit`) e suporte ao comando `exec`, responsável por carregar programas e dados de arquivos binários diretamente na memória do FPGA. As chamadas de sistema de arquivos foram implementadas de forma simplificada, apenas utilizando comunicação serial para acessar arquivos no computador. Os testes demonstraram a funcionalidade completa do *shell*, incluindo tratamento de erros, *scroll* automático da tela e compatibilidade com o simulador RARS.**

***Abstract*—This article presents the development of a shell for a RISC-V processor implemented on an FPGA, with the aim of enabling the loading and execution of external programs via an RS232 serial interface connected to a computer. The solution replaces the traditional loading method using the In-System Memory Content Editor tool from Intel Quartus® Prime, providing a faster and more automated alternative. The system offers a text interface based on the REPL model, with basic commands (`echo`, `clear`, `help`, `exit`) and support for the `exec` command, which loads code and data from binary files directly into the FPGA's memory. File system calls were implemented by simply using serial communication to access files on the connected computer. Testing confirmed the shell's full functionality, including error handling, automatic screen scrolling, and compatibility with the RARS simulator.**

***Index Terms*—RISC-V, FPGA, RS232, shell, loader**

I. Introdução

FPGAs (*Field-Programmable Gate Arrays*) são circuitos integrados reconfiguráveis que têm ampla utilização no meio acadêmico, especialmente no ensino e pesquisa de arquitetura de computadores. Uma de suas principais vantagens é a possibilidade de projetar e testar arquiteturas de processadores, explorando desde conceitos básicos de microarquitetura até sistemas completos. No contexto da disciplina de Organização e Arquitetura de Computadores (OAC), ministrada na Universidade de Brasília (UnB), utilizam-se FPGAs do modelo DE1-SoC [8], que permitem a implementação prática de processadores desenvolvidos em sala de aula.

Entre os projetos utilizados no laboratório da disciplina, destaca-se o processador baseado na arquitetura RISC-V, uma ISA (*Instruction Set Architecture*) aberta e extensível, implementado sob a orientação do professor Marcus Vinicius Lamar com a colaboração de diversos alunos. Essa implementação tem servido como base para experimentos com a execução de programas diretamente no FPGA. A versão em foco nesse trabalho é a ISA RV32IMF multiciclo.

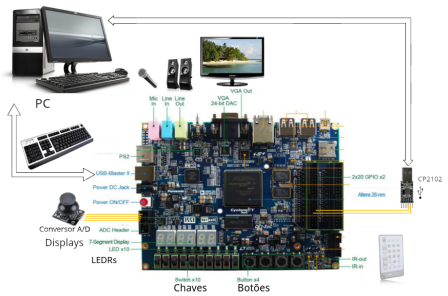

Figura 1. Ambiente de laboratório da disciplina de OAC

O ambiente de laboratório possui diversos recursos de hardware já integrados ao kit DE1-SoC (Figura 1), como a saída de vídeo via VGA, a saída de áudio analógico via conector P2, a entrada para teclado no padrão PS/2 e um conversor analógico-digital. Além disso, há uma interface serial RS232, que possibilita a transmissão de dados entre o processador RISC-V e o PC por meio de um conversor USB–RS232 (CP2102). É também disponibilizado para os alunos um sistema operacional simples e mínimo, que provê algumas chamadas de sistema, como entrada e saída pelo teclado e pelo VGA, respectivamente.

Contudo, a forma tradicional de carregar programas para execução nesse processador exige o uso da ferramenta *In-System Memory Content Editor*, disponibilizada pelo ambiente Intel Quartus® Prime. Esse procedimento apresenta limitações consideráveis em termos de usabilidade. O usuário precisa abrir manualmente a ferramenta, selecionar o bloco de memória adequado, localizar e carregar arquivos no formato .mif ou .hex e, por fim, pressionar a tecla F7 para escrever os dados/programas nas memórias do FPGA. Esse processo é repetitivo, propenso a erros e pouco intuitivo para uso

frequente ou aplicações mais dinâmicas.

Diante disso, o objetivo deste trabalho é o desenvolvimento de um *shell* simples, capaz de ser executado diretamente no processador RISC-V implementado no FPGA, com o propósito de facilitar a execução de programas externos. Esse *shell* se comunica com um computador com o sistema operacional Windows por meio de uma conexão serial RS232, adaptada via USB, permitindo o carregamento interativo de arquivos binários armazenados em um computador para a memória do processador RISC-V.

Essa abordagem propõe uma alternativa mais flexível e automatizada em comparação ao fluxo tradicional com o *In-System Memory Content Editor*. Para viabilizar esse sistema, foram definidos os seguintes objetivos específicos:

- Desenvolver um *shell* funcional, com capacidade de leitura interativa de comandos e execução local no processador RISC-V.
- Implementar chamadas de sistema de arquivos, possibilitando a interação com arquivos armazenados no PC.
- Criar um carregador de programas que interprete o conteúdo dos arquivos enviados pelo computador e os armazene corretamente nas memórias de instruções e dados do FPGA.

A seção II aborda os conhecimentos básicos necessários para melhor compreensão desse artigo. A seção III apresenta em detalhes a metodologia utilizada para atingir esses objetivos, descrevendo tanto os aspectos técnicos da implementação quanto as decisões de projeto adotadas ao longo do desenvolvimento. A seção IV evidencia os resultados obtidos no estudo. Por fim, as seções V e VI concluem o artigo e apresentam possíveis trabalhos futuros.

II. Referencial Teórico

A. Shell e interpretação de comandos em sistemas interativos (REPL)

Sistemas interativos baseados no modelo *Read-Eval-Print Loop* (REPL) são caracterizados por um ciclo contínuo em que o sistema lê a entrada textual do usuário, avalia ou interpreta o comando recebido, apresenta o resultado (caso exista) e retorna ao estado de espera por nova entrada [3]. Esse modelo é amplamente adotado em interfaces de linha de comando, linguagens interpretadas e ambientes interativos de desenvolvimento.

O conceito de *shell*, nesse contexto, refere-se a uma interface textual que interpreta comandos do usuário e executa ações diretamente no sistema. *Shells* tradicionais, como os compatíveis com o *Portable Operating System Interface* (POSIX), implementam linguagens de comando completas, com suporte a variáveis, estruturas de controle, redirecionamento e execução de *scripts* [4]. No entanto, *shells* mais simples, utilizados em contextos específicos ou com restrições de recursos, podem limitar-se à execução de comandos básicos, mantendo ainda assim a estrutura interativa essencial ao modelo REPL.

A principal função de um *shell* é oferecer um meio direto de interação entre o usuário e o sistema [1, pp. 42-43], independentemente da complexidade dos comandos suportados. Em ambientes embarcados, acadêmicos ou experimentais, o *shell* pode assumir um papel específico de controle, permitindo a realização de tarefas como inicialização de subsistemas, teste de funcionalidades ou carregamento de programas.

B. Comunicação Serial RS232

O padrão RS232 (*Recommended Standard* 232) [7] é um protocolo de comunicação serial assíncrona originalmente desenvolvido nos anos 1960 para interligação de terminais e modems. Apesar de suas limitações em velocidade e distância, o RS232 continua sendo amplamente utilizado em sistemas embarcados, devido à sua simplicidade, ampla documentação e suporte em microcontroladores e dispositivos de *hardware* diversos.

A comunicação serial baseada em RS232 ocorre utilizando três sinais principais: transmissão (TX), recepção (RX) e *ground* (GND), além de sinais de controle opcionais [5]. Como é assíncrona, a sincronização entre os dispositivos é garantida por meio de parâmetros comuns, como taxa de transmissão (*baud rate*), número de bits de dados, bit de paridade e bit de parada.

Em projetos acadêmicos com FPGAs, o uso de RS232 é comum por oferecer uma via de comunicação simples e direta entre um processador implementado no *hardware* e um computador. Essa interface permite, por exemplo, o envio e recebimento de dados, comandos ou arquivos, sem a necessidade de protocolos complexos ou pilhas de software. Além disso, seu uso é facilitado pela disponibilidade de conversores USB-serial compatíveis com o padrão, o que garante a interoperabilidade com computadores modernos.

A escolha pela utilização do protocolo RS232 neste trabalho se deu principalmente pela disponibilidade de uma implementação já existente na plataforma do FPGA, desenvolvida por alunos em turmas anteriores da disciplina de Organização e Arquitetura de Computadores (OAC). Essa implementação foi baseada no blog FPGA4Fun[1], que descreve uma interface serial mapeada em memória.

C. Chamada de sistema

Chamadas de sistema (*system calls*) são mecanismos que permitem que programas em modo usuário solicitem serviços ao sistema operacional ou ao ambiente supervisor, como acesso a dispositivos de entrada e saída, arquivos, memória ou controle de processos. Elas estabelecem uma interface controlada entre os diferentes níveis de privilégio do sistema, evitando que aplicações tenham acesso direto a recursos críticos do *hardware* [1, pp. 47-50].

Em processadores modernos, chamadas de sistema são normalmente implementadas por meio de instruções especiais, como a `ecall` na arquitetura RISC-V, que geram uma exceção ou interrupção controlada. Essa exceção altera o fluxo de execução, redirecionando-o para um endereço previamente configurado no registrador de vetor de exceção (como o `mtvec` em RISC-V), onde reside a rotina responsável por tratar a chamada [2].

[1] https://www.fpga4fun.com/SerialInterface.html

A estrutura típica de uma chamada de sistema envolve a passagem de argumentos por registradores (como `a0`–`a6` em RISC-V), a identificação do serviço desejado por meio de um número associado à chamada (passado no registrador `a7` em RISC-V) e o retorno de resultados ao programa solicitante, também via registradores (`a0` e `a1` em RISC-V). No contexto da disciplina de OAC, há um procedimento chamado `ExceptionHandler`, que é responsável pelo tratamento de exceções. Ao identificar que a exceção é uma chamada de sistema, ele salva o estado dos registradores na pilha, executa a chamada de sistema desejada, restaura os registradores e, finalmente, utiliza a instrução `mret` para retornar da chamada de sistema.

D. Manipulação de arquivos em sistemas operacionais

A manipulação de arquivos é uma das funcionalidades fundamentais de um sistema operacional, responsável por permitir o armazenamento, acesso, leitura e escrita de dados persistentes em dispositivos de memória secundária. Os sistemas operacionais modernos fornecem uma abstração de arquivos que oculta detalhes físicos dos dispositivos de armazenamento, oferecendo uma interface uniforme para os programas de usuário [1, pp. 38-41].

Essa interface consiste em um conjunto de chamadas de sistema responsáveis por realizar operações sobre arquivos. Entre essas operações estão: `Open`, `Read`, `Write`, `Seek` e `Close`, que permitem, respectivamente, abrir arquivos, ler dados de forma sequencial ou aleatória, escrever ou modificar seu conteúdo e fechá-los adequadamente após o uso [1, pp. 262-263]. Internamente, o sistema operacional mantém estruturas de controle como descritores de arquivos, isto é, inteiros que identificam arquivos abertos, e tabelas de arquivos abertos, que associam os descritores utilizados pelo processo às informações reais sobre a localização e o estado do arquivo.

E. Carregador e seções de memória

O carregamento de programas em um sistema computacional envolve a leitura de um arquivo executável e a transferência de seu conteúdo para regiões apropriadas da memória principal, de modo que o processador possa executá-lo [6]. Esses arquivos executáveis, como o formato ELF (*Executable and Linkable Format*) utilizado em sistemas Unix, organizam o programa em seções distintas que incluem código, dados inicializados, dados não inicializados e informações auxiliares para o carregamento. A tarefa de interpretar esse conteúdo e alocá-lo corretamente na memória é desempenhada por um componente chamado carregador.

Em arquiteturas clássicas, essas seções de memória são bem definidas: as seções `text`, `data` e `bss` armazenam, respectivamente, o código executável, os dados inicializados e os dados não inicializados. O carregador é responsável por posicionar cada uma dessas seções em seus respectivos endereços, conforme especificado durante a montagem ou ligação do programa.

F. RARS

O RARS (*RISC-V Assembler and Runtime Simulator*[2]) é uma ferramenta educacional desenvolvida para facilitar o aprendizado da arquitetura RISC-V por meio da escrita, montagem e simulação de programas em *assembly*. Inspirado no MARS (utilizado no ensino da arquitetura MIPS), o RARS provê uma interface gráfica que permite editar código, montar instruções, simular a execução passo a passo e visualizar o conteúdo dos registradores e da memória.

Internamente, o RARS funciona como montador, traduzindo o código-fonte em *assembly* para instruções binárias compatíveis com a arquitetura RISC-V, e como simulador, executando essas instruções em um ambiente controlado que emula o comportamento do processador. Para fins didáticos, o simulador oferece um conjunto de chamadas de sistema que proveem entrada e saída padrão, manipulação de arquivos e outras funcionalidades.

Além disso, o RARS permite exportar programas montados para diversos formatos de arquivos, como binário, mif e hexadecimal em ASCII, que podem ser utilizados fora do simulador. Os arquivos mif, por exemplo, podem ser usados para carregar um programa pela ferramenta *In-System Memory Content Editor*. No contexto dos arquivos binários, diferentemente de montadores tradicionais, como o *GNU Assembler*, o RARS não os gera em formatos executáveis padrão (como ELF ou PE). Ao invés disso, o RARS só é capaz de gerar arquivos com o conteúdo de uma seção de memória, sendo necessária a geração de dois arquivos (um da seção `text` e um da seção `data`) para obter um programa completo.

III. METODOLOGIA

Esta seção descreve os métodos e estratégias empregados no desenvolvimento do sistema proposto, cuja execução ocorre diretamente no processador RISC-V implementado no FPGA. O ambiente de interação com o usuário é realizado por meio de um monitor externo conectado ao FPGA, no qual é exibida a interface do *shell*. O PC participa apenas como um auxiliar para o acesso a arquivos, sendo utilizado por meio de um programa em linguagem C que se comunica via RS232 com o FPGA e atende às chamadas de sistema solicitadas pelo processador.

O trabalho foi estruturado em três partes principais. Primeiramente, a Subseção III-A apresenta a implementação do *shell*, que inclui mecanismos para leitura interativa de comandos, reconhecimento de instruções simples, execução de comandos internos (como `echo`, `clear`, `help`, `exit` e `exec`) e suporte a rolagem automática da tela.

Em seguida, a Subseção III-B aborda o sistema de arquivos, detalhando a comunicação serial RS232 tanto no computador quanto no FPGA, além da criação do programa em C responsável por processar as chamadas de sistema de manipulação de arquivos. Essa abordagem permite que o processador RISC-V utilize operações como `Open`, `Read` e `Write` mesmo sem um sistema de arquivos.

[2]https://github.com/TheThirdOne/rars

Por fim, a Subseção III-C apresenta o carregador de programas, que organiza a memória entre a área reservada para o código do usuário e a área destinada ao *kernel*, o qual, neste trabalho, designa especificamente o conjunto formado pelo *shell* e pelas rotinas de chamadas de sistema. Nessa parte, também são detalhadas a implementação do comando `exec`, responsável por carregar programas externos para execução no FPGA, e as modificações realizadas na chamada de sistema `exit`, de modo a assegurar o retorno ao *shell* após a finalização de um programa.

A. Shell

1) Leitura interativa de string

Para implementar a funcionalidade de REPL (*Read-Eval-Print Loop*) do *shell*, foi desenvolvida uma função que permite a leitura interativa da entrada do usuário, exibindo os caracteres digitados e permitindo sua edição. Essa função foi implementada utilizando um laço onde cada caractere é lido individualmente. Caso seja um *backspace* (0x8), o último caractere é apagado. Caso seja uma quebra de linha (0xA), o laço termina. Caso contrário, o caractere é impresso na tela e o cursor passa para a próxima posição.

Essa função foi implementada como uma chamada de sistema para ser utilizada por outros alunos.

2) Parsing de comandos

Como o *shell* desenvolvido possui poucos comandos e não oferece suporte a funcionalidades mais complexas, como uso de variáveis e redirecionamento de entrada e saída, não foi necessário implementar um *parser* completo. A identificação dos comandos é feita por meio da função `starts_with`, que recebe duas *strings* e retorna verdadeiro se a segunda for prefixo da primeira. Isso permite verificar se a entrada corresponde a um comando válido.

3) Comandos implementados: `echo`, `clear`, `exit`, `help` e `exec`

Foram implementados os seguintes comandos para o *shell*:

- `echo` - imprime o argumento fornecido na linha seguinte.
- `clear` - limpa a tela e posiciona o cursor na primeira linha.
- `exit` - encerra a execução do *shell*.
- `help` - exibe os comandos disponíveis e outras informações.
- `exec` - executa o programa especificado

A seção III-C2 documenta a implementação do comando `exec` com mais detalhes.

4) Rolagem automática

Quando o conteúdo atinge a última linha da tela, o *shell* executa uma rolagem automática para liberar espaço. Isso é feito por uma função que desloca o conteúdo da tela duas linhas para cima.

B. Sistema de Arquivos

1) Comunicação RS232 no PC

A comunicação serial RS232 foi implementada em C, utilizando a biblioteca de Teuniz van Beelen[3]. A biblioteca original oferece apenas funções básicas (`PollComport`, `SendByte` e `SendBuf`). Por isso, foram implementadas funções adicionais de leitura blocante: `ReadByte`, `ReadInt`, `ReadString`, `ReadBuf`, além da função `SendInt`.

Das funções de leitura, a função `ReadByte` é a única que invoca diretamente uma função da biblioteca, sendo essa a função `PollComport`. Enquanto isso, a função `SendInt` utiliza a função `SendBuf` da biblioteca. Todas as funções de transferência de inteiro deste trabalho são *big-endian*, ou seja, os *bytes* mais significativos são transferidos primeiro.

2) Comunicação RS232 no FPGA (RISC-V)

No FPGA, a comunicação RS232 foi implementada diretamente em *Assembly* RISC-V, utilizando uma interface mapeada em memória. As funções desenvolvidas foram: `ReadByte`, `ReadInt`, `ReadBuf`, `SendByte`, `SendInt`, `SendString` e `SendBuf`. Apenas as funções `ReadByte` e `SendByte` interagem diretamente com a interface mapeada em memória. Todas as outras funções utilizam a `ReadByte` ou `SendByte` em *loop*.

Com base nessas duas rotinas elementares, foram criadas funções de nível mais alto capazes de lidar com unidades maiores de informação, como inteiros e blocos de dados. As funções `ReadInt`, `ReadBuf`, `SendInt`, `SendString` e `SendBuf` operam repetidamente sobre `ReadByte` ou `SendByte`, compondo leituras e escritas byte a byte.

3) Programa em C para chamadas de sistema de arquivos

Foi criado um programa em C que é executado no PC e aguarda comandos enviados pelo RISC-V via RS232. O programa interpreta o *byte* recebido para identificar a chamada desejada, coleta os argumentos por meio das funções de leitura implementadas e executa a função correspondente da biblioteca `stdio.h`. Após a execução, os valores de retorno são enviados de volta ao RISC-V. Esse padrão pode ser visto na função `syscall_write`, vista na Listagem 1.

```
void syscall_write() {
    int fd = RS232_ReadInt(cport_nr);
    printf("fd: %d\n", fd);
    int n = RS232_ReadInt(cport_nr);
    printf("n: %d\n", n);
    char buf[n];
    RS232_ReadBuf(cport_nr, buf, n);
    int count = fwrite(buf, 1, n, files[fd]);
    printf("%d bytes escritos\n", count);
    RS232_SendInt(cport_nr, count);
}
```

Listing 1. Implementação da função `syscall_write`.

Enquanto a biblioteca `stdio.h` utiliza ponteiros de arquivo (`FILE*`) para suas operações, as chamadas de sistema do RISC-V utilizam apenas descritores de arquivos (`int`).

[3]https://www.teuniz.net/RS-232/

Sendo assim, foi necessário um mapeamento entre descritores e ponteiros de arquivos. Para isso, foi utilizado o vetor `FILE *files[1024]`. A função `syscall_open`, invocada para executar a chamada `Open`, armazena o ponteiro do arquivo na posição indexada pelo seu descritor, como visto na Listagem 2.

```
void syscall_open() {
    ...
    int fd;
    FILE *f = fopen(filepath, mode);
    if (!f) {
        fprintf(stderr, "Erro ao abrir o
            arquivo %s: %s\n", filepath,
            strerror(errno));
    fd = -1;
    } else {
        fd = fileno(f);
        files[fd] = f;  /* salva o stream do
            arquivo aberto no indice do
            descritor do arquivo */
        printf("Arquivo %s aberto no descritor
            %d com sucesso\n", filepath, fd);
    }
    ...
}
```

Listing 2. Trecho da função `syscall_open`, evidenciando o mapeamento entre descritores e ponteiros de arquivos.

4) Chamadas de sistema de arquivos no FPGA

As chamadas de sistema `Open`, `Close`, `LSeek`, `Read` e `Write` foram implementadas com a mesma semântica das chamadas de sistema presentes no simulador RARS. Sua implementação no processador RISC-V foi feita de forma simples: o RISC-V envia o byte identificador da chamada seguido dos argumentos via RS232 e aguarda a resposta do PC com os valores de retorno, conforme mostrado na Listagem 3.

```
########################################
# Read
# a0 = descritor do arquivo
# a1 = endereco do buffer onde os bytes serao
    escritos
# a2 = quantidade de bytes a serem lidos
########################################
# retorna
# a0 = quantidade de bytes lidos
########################################
Read: DE1(s8, Read.DE1)
    ecall
    ret
Read.DE1: addi sp, sp, -4
    sw ra, 0(sp)

    mv s0, a0
    li a0, READ
    jal RS232_SendByte  # manda o codigo da
        chamada pro pc
    mv a0, s0
    jal RS232_SendInt   # manda descritor do
        arquivo pro pc
    mv a0, a2
    jal RS232_SendInt   # manda quantidade de
        bytes a serem lidos pro pc

    jal RS232_ReadInt   # le quantos bytes
        foram lidos
    mv s0, a0
    mv a0, a1
    mv a1, s0
    jal RS232_ReadBuf
    mv a0, s0

    lw ra, 0(sp)
    addi sp, sp, 4
    ret
```

Listing 3. Implementação da chamada de sistema Read no FPGA.

C. Carregador

1) Separação da memória

Para garantir a execução correta do código do usuário, é necessário que ele seja carregado no mesmo endereço em que foi montado. O montador do RARS, no entanto, sempre monta o código a partir do endereço `0x00400000`. Sendo assim, para impedir que o código do usuário sobrescreva o *kernel*, surge a necessidade de montá-lo a partir de um outro endereço. Foi então escolhido o endereço `0x0041000` como endereço inicial do código do *kernel* (seção `ktext`). Similarmente, foi escolhido o endereço `0x10030000` como o endereço inicial da seção de dados do *kernel* (seção `kdata`). Estes endereços foram definidos a partir das restrições de quantidade de memória interna existente no FPGA, onde 128KiB são reservados ao segmento `data` e 64KiB ao segmento `text`

Para possibilitar a montagem do *kernel* nesses endereços, foram desenvolvidas duas novas diretivas no montador do RARS: `ktext` e `kdata`. Essas diretivas funcionam de forma análoga às diretivas `text` e `data`, sinalizando para o montador que o trecho de código ou de dados a seguir deve ser montado a partir do endereço inicial da seção correspondente à diretiva.

Com isso, é necessário apenas um pequeno programa no início da seção `text`, com a função de configurar o endereço do ktext no registrador utvec e transferir a execução para o *shell*. Esse programa pode ser visto na Listagem 4.

```
.text
la   tp, ExceptionHandling   # carrega em tp o
    endereco base das rotinas do sistema ECALL
csrw    tp, utvec        # seta utvec para o
    endereco tp
csrsi   ustatus, 1
j shell.clear_cmd
```

Listing 4. Programa inicial necessário para configurar o registrador utvec e iniciar a execução do shell.

2) Comando `exec`

Com a seção de programa do usuário reservada, foi adicionado o comando `exec` ao *shell*, responsável por carregar o código. Como o RARS não gera arquivos em formato executável (ELF, PE, etc.), é necessária a geração de dois arquivos binários separados: `programa.bin` (código) e `programa.dat` (dados), com `programa` sendo o nome do programa do usuário.

Inicialmente, o carregador utiliza a chamada `Open` para abrir o arquivo `programa.bin`. Caso não o encontre, o *shell* imprime uma mensagem de erro comunicando isso ao usuário, e volta ao REPL. Depois, o carregador utiliza a chamada

`LSeek` para determinar o tamanho do arquivos. Caso exceda o espaço disponível, outra mensagem de erro é exibida. Caso contrário, o arquivo será lido.

Se o *shell* estiver sendo executado no processador RISCV, o arquivo é lido diretamente na seção `text`. Se o *shell* estiver rodando no RARS, isso não é possível, então o conteúdo do arquivo é temporariamente carregado na seção `data` e copiado instrução por instrução para a seção correta. Após a leitura do arquivo de código, o mesmo procedimento é feito para o arquivo de dados. Caso obtenha sucesso no carregamento, os arquivos são fechados e a execução é transferida para o início da seção `text`.

3) Chamada de sistema `exit`

Para que o *shell* permaneça funcional após a execução de um programa do usuário, foi necessário modificar a chamada de sistema `exit`. O comportamento original dessa chamada variava conforme o ambiente de execução: no simulador RARS, ela invocava a rotina de encerramento do próprio emulador; já na execução no processador no FPGA, a chamada resultava em um laço infinito. Em ambos os casos, não havia um retorno ao *shell*.

A nova implementação da chamada `exit`, apresentada na Listagem 5, unifica o comportamento para os dois ambientes. Após a finalização do programa do usuário, a mensagem "Pressione qualquer tecla para continuar" é exibida na parte inferior da tela e o sistema aguarda a entrada de um caractere por meio da chamada `readChar`. Essa pausa permite que o usuário visualize a saída final do programa antes que o controle retorne ao *shell*.

Em seguida, o ponteiro de pilha (`sp`) é restaurado ao estado anterior à chamada de sistema. Por fim, o endereço do procedimento de `clear` do *shell* é carregado no registrador `uepc`, e a instrução `uret` é executada. Esse procedimento garante o retorno correto ao fluxo de execução do *shell* após a chamada de sistema, respeitando a convenção de exceções e interrupções da arquitetura RISC-V.

```
goToExit: la a0, press_key_msg
    li a1, 0
    li a2, 232
    li a3, 0x00FF
    li a4, 0
    jal printString
    jal readChar
    addi    sp, sp, 264   # reseta a pilha
    li t0, SHELL_CLEAR_ADDRESS
    csrw     t0, uepc     # coloca no
        registrador uepc
    uret
```

Listing 5. Chamada de sistema `exit` nova.

IV. Resultados

Os componentes do sistema foram projetados com atenção ao uso eficiente da memória, de modo a garantir o funcionamento do *shell*, das chamadas de sistema e ainda reservar espaço adequado para os programas de usuário. O *shell* completo, incluindo os comandos internos e a interface de entrada interativa, ocupa 1140 *bytes*, enquanto a implementação das chamadas de sistema utiliza 7548 *bytes*. Em contraste, o espaço reservado para os programas de usuário corresponde a 64 KiB (65536 *bytes*), valor significativamente maior do que o do *kernel* (8688 *bytes*). Essa distribuição evidencia que a maior parte da memória está disponível para execução de aplicações externas, ao passo que o *kernel* minimalista permanece compacto, respeitando os recursos limitados do FPGA e facilitando sua reutilização em projetos futuros.

O código do *shell* e das chamadas de sistema está disponível em um repositório do *Github*[4]. Vídeos demonstrando o funcionamento do trabalho estão disponíveis no *YouTube*[5,6].

A. Execução do Shell no Processador RISC-V

A execução do sistema inicia com o carregamento do programa do *shell* na memória do FPGA pelo *In-System Memory Content Editor*, seguido da configuração do registrador utvec e da transferência de controle para o endereço inicial do *shell*. A interface textual é exibida diretamente na tela conectada ao FPGA e permite a interação com o usuário por meio de um teclado conectado ao sistema. A Figura 2 mostra uma foto do *shell* em execução no ambiente do FPGA.

Figura 2. Foto da execução do *shell* no FPGA

O *shell* implementa os quatro comandos internos: `echo`, `clear`, `help` (Figura 3) e `exit`. Cada comando foi testado individualmente no processador, utilizando entrada por teclado e exibição direta em vídeo. Comandos inválidos são reconhecidos, e o *shell* informa o usuário que o comando é inválido, retornando imediatamente ao estado de espera por nova entrada.

[4]https://github.com/ziul123/risc-v-loader-shell
[5]https://youtu.be/2BpYoFsBRDc
[6]https://youtu.be/fo6uGAqDrww

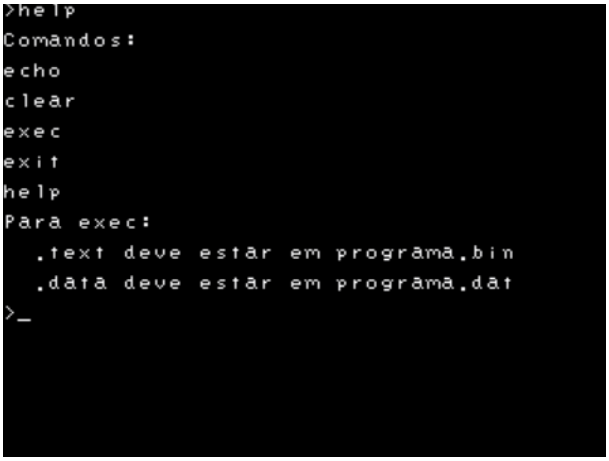

Figura 3. Comando `help`.

B. Rolagem automática da tela

O sistema de exibição textual conta com suporte a rolagem automática. Quando o conteúdo exibido alcança a última linha visível da tela, uma rotina de *scroll* é acionada, deslocando as linhas para cima e liberando espaço para novos comandos e saídas.

C. Chamadas de Sistema de Arquivos

Cada chamada de sistema implementada foi submetida a testes utilizando diferentes parâmetros de entrada e em todas as situações observou-se o comportamento correto esperado. Após essa validação funcional, também foi realizada uma avaliação de desempenho das chamadas `Read` e `Write`, medindo o tempo necessário para a transferência de blocos de dados de 1 KiB.

Para garantir resultados consistentes, o procedimento foi repetido 100 vezes e, a partir desses dados, calculou-se a média dos tempos medidos. Destaca-se que, em todo o desenvolvimento deste trabalho, a comunicação serial RS232 foi configurada para operar a uma taxa de transmissão (*baud rate*) de 115200 b/s. A Tabela I apresenta os valores obtidos nesse teste de desempenho.

Operação	Tempo médio de execução	Taxa de transferência
Read 1 KiB	114,90 ms	8,91 KB/s
Write 1 KiB	122,09 ms	8,38 KB/s

Tabela I
TEMPO DE EXECUÇÃO E TAXA DE TRANSFERÊNCIA DAS CHAMADAS READ E WRITE PARA BLOCOS DE 1 KIB.

Essas chamadas são essenciais para o funcionamento do carregador de programas, permitindo que os arquivos binários e de dados sejam lidos do computador e transferidos para a memória interna do FPGA.

D. Carregamento de Programas Externos

O carregamento de programas externos, ambos no FPGA via RS232 e no RARS, foi bem sucedido. Casos de erro, como ausência do arquivo ou excesso de tamanho, são tratados com mensagens adequadas na interface do *shell*.

E. Chamada de sistema `exit` no programa de usuário

A Figura 4 apresenta a tela após a execução de um programa de usuário. A chamada `exit` está esperando uma tecla ser digitada, possibilitando a leitura da mensagem impressa pelo programa do usuário antes da tela ser apagada pelo procedimento de `clear` do *shell*.

Figura 4. Tela após a execução de um programa de usuário e da chamada de sistema `exit`.

V. CONCLUSÃO

Este trabalho apresentou o desenvolvimento de um *shell* funcional para um processador RISC-V implementado em FPGA, com o objetivo de facilitar o carregamento e a execução de programas externos a partir de um computador conectado via interface serial RS232. A motivação principal foi superar as limitações impostas pelo método tradicional de carregamento de programas diretamente na memória utilizando o *In-System Memory Content Editor* do Quartus®, propondo uma solução mais automatizada.

O sistema implementado demonstrou ser eficaz ao oferecer uma interface textual baseada no modelo REPL, capaz de interpretar comandos básicos como `echo`, `clear`, `help` e `exit`, além de possibilitar o carregamento de arquivos binários por meio do comando `exec`. A implementação das chamadas de sistema de arquivos, com comunicação RS232 entre FPGA e PC, permitiu a transferência de dados e programas de forma eficiente.

VI. TRABALHOS FUTUROS

Como trabalhos futuros, diversas extensões podem ser exploradas para ampliar a funcionalidade do sistema desenvolvido. Uma possibilidade é a implementação de funcionalidades adicionais no *shell*, como histórico de comandos, variáveis de ambiente e redirecionamento de entrada e saída. Além disso, novas chamadas de sistema podem ser adicionadas para permitir uma maior integração entre os programas executados e o próprio *shell*, como, por exemplo, leitura e escrita equivalente ao uso de `stdin` e `stdout` em terminais comuns.

Outra extensão relevante seria a introdução de um mecanismo de escalonamento, possibilitando a execução concorrente de múltiplos programas no ambiente do *shell*, com

troca de contexto e gerenciamento de processos. Também se destaca a implementação de um sistema de paginação em memória real, que permitiria a execução de programas maiores do que a memória física disponível no FPGA, por meio do carregamento sob demanda de páginas de código e dados.

Por fim, o carregador poderia ser modificado para carregar arquivos ELF, possibilitando assim o uso de montadores tradicionais. É possível também aprimorar o programa residente no computador que responde às chamadas de sistema para oferecer funcionalidades mais próximas de um sistema de arquivos remoto, como listagem de arquivos no diretório atual ou mudança de diretório.

Referências

[1] Andrew Tanenbaum; Herbert Bos. *Modern Operating Systems*. 4ª ed. Pearson Education, 2014. ISBN: 013359162X; 9780133591620.

[2] RISC-V International Privileged Horizontal Committee. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. Technical Report Version 20250508. Latest ratified volume II (Privileged ISA). RISC-V International, mai. de 2025, pp. 39–40.

[3] Gerald Jay Sussman Harold Abelson. *Structure and Interpretation of Computer Programs - 2nd Edition*. 2ª ed. MIT Electrical Engineering and Computer Science. The MIT Press, 1996, p. 7. ISBN: 0262011530; 9780262011532.

[4] IEEE and The Open Group. *POSIX.1-2017: IEEE Std 1003.1-2017, Shell Command Language*. https://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html. 2018.

[5] Lakeview Research Jan Axelson. *Serial Port Complete: Programming and Circuits for Rs-232 and Rs-485 Links and Networks*. Lakeview Research, 2002, p. 121. ISBN: 9780965081979; 0965081974.

[6] John R. Levine. *Linkers & Loaders*. 1st. The Morgan Kaufmann Series in Software Engineering and Programming. Morgan Kaufmann, 1999, pp. 10–12. ISBN: 9781558604964; 1558604960.

[7] Telecommunications Industry Association (TIA). *Interface Between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange*. Reaffirmed 2002, 2012. Telecommunications Industry Association (TIA), 1997.

[8] Terasic Technologies. *DE1-SoC Development Kit*. https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=165&No=836#contents. Accessed: 2025-06-29. 2025.