Desenvolvimento do Simulador do Processador RISC-V
Organização e Arquitetura de Computadores - Turma 03

Gabriel Lucas França do Nascimento
190107111
nascimento.franca@aluno.unb.br

25/11/2024

1 Introdução

Este projeto envolve a criação de um simulador para a arquitetura RV32I utilizando uma linguagem de alto nível, como C, C++ ou Python. As funções essenciais de busca e decodificação de instruções já estão implementadas (em C), porém, para este trabalho em específico foram recriadas em Python. O objetivo é implementar a função de execução (execute()) para as instruções do subconjunto especificado. O programa binário a ser simulado deve ser gerado usando o montador RARS, acompanhado dos dados correspondentes. O simulador precisa ser capaz de ler arquivos binários que contenham tanto o segmento de código quanto o segmento de dados, carregá-los na memória e executar o programa.

2 Descrição do Trabalho

Conforme descrito na especificação do trabalho, as instruções e dados devem vir a partir dos arquivos gerados pelo RARS. Para isso, criamos um arquivo de teste chamado Testador.asm, e realizamos o Dump de memória, gerando assim 2 arquivos: code.bin e data.bin.

Foi utilizada a configuração "Compact, Text at Address 0". As instruções implementadas no trabalho foram:

- LW (Load Word)
- LB (Load Byte)
- LBU (Load Byte Unsigned)
- SW (Store Word)
- SB (Store byte)
- ADD (Sum)
- ADDi (Add Immediate)
- SUB (Subtract)
- AND
- OR
- XOR (Exclusive OR)
- SLT (Set Less Than)
- SLTU (Set Less Than Unsigned)
- SLLI (Shift Left Logical Immediate)
- SRAI (Shift Right Arithmetic Immediate)
- SRLI (Shift Right Logical Immediate)
- JAL (Jump and Link)
- JALR (Jump and Link Register)
- AUIPC (Add Upper Immediate to PC)
- LUI (Load Upper Immediate)
- BEQ (Branch if Equal)

- BNE (Branch if Not Equal)
- BGE (Branch if Greater or Equal)
- BGEU (Branch if Greater or Equal Unsigned)
- BLTU (Branch if Less Than Unsigned)
- BLT (Branch if Less Than)
- ORi (OR Immediate)
- ANDi (AND Immediate)
- ECALL (System call)

A memória é estruturada para comportar 4096 palavras de 32bits, utilizando 14 bits para endereçamento, e os dados são carregados a partir do endereço 0x2000. O programa é carregado a partir do endereço 0.

```
# Configuração da memória e registradores
MEM_SIZE = 16384  # 16 KB de memória
mem = np.zeros(MEM_SIZE, dtype=np.uint8)

# Registradores RISC-V
reg = np.zeros(32, dtype=np.int32)  # Registradores x0-x31
pc = 0  # Program Counter (PC)
reg[0] = 0 # Registrador x0 sempre vale zero
```

3 Descrição da Implementação

A arquitetura do processador é baseada nas seguintes etapas principais:

1. **Fetch:** O PC busca a instrução da memória de instrução. A cada instrução, o PC é incrementado para a próxima.
2. **Decode:** A instrução é decodificada, e os registradores relevantes são lidos. São atribuídos os registradores e campos de identificação da instrução.
3. **Execute (EX)**: É realizada toda a operação das instruções, incluindo leitura e escrita de registradores, memoria e PC.

3.1 Da implementação

Todos os testes fornecidos pelo professor foram implementados com sucesso. O primeiro deles:

1. Código em Assembly RISC-V

```
    teste1:
        li t1, -2          # Testa ADD
        li t2, 3
        add t3, t1, t2
        li a1, 1
        li t6, 1
        beq t3, t6, t1_ok
        jal FAIL
        j teste2
t1_ok:
        jal OK
```

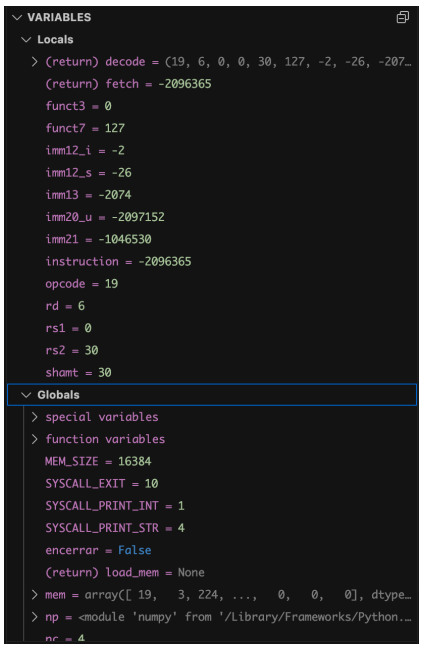

Figure 1: Campos após a decodificação das instruções

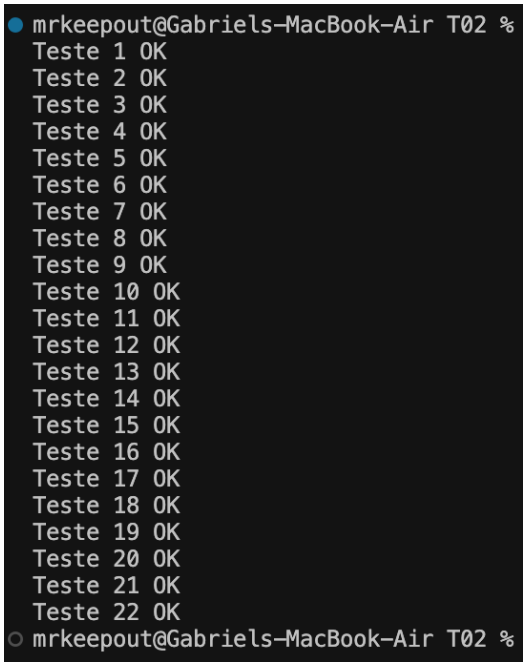

Figure 2: Todos os testes do arquivo Testador.asm passaram com sucesso.

4 Detalhes da implementação e execução

Para implementação e execução do código foi utilizado Python 3.13.0 com o VSCode 1.95.3 (ARM64 Edition - Apple Silicon). Para debug, foi utilizado o debug padrão do Python com o VSCode.

- Sistema operacional: MacOS Sequoia 15.1.1
- Processador: M1 Apple Silicon
- IDE: VSCode 1.95.3