Processador RISC-V de Ciclo Único em Verilog

Victor Gomes ferreira - 23.1.8010, Gabriel Calili Nunes Biciate - 23.2.8020

Resumo

Este documento apresenta a implementação de um processador RISC-V de 32 bits com um caminho de dados de ciclo único, desenvolvido como parte da disciplina de CSI211 - Fundamentos de Organização e Arquitetura de Computadores da Universidade Federal de Ouro Preto (UFOP). O design foi baseado no diagrama de caminho de dados da Figura 4.21 do livro texto e suporta um subconjunto de 7 instruções do padrão RISC-V. São detalhados a estrutura modular do projeto, os procedimentos de compilação e execução, e os resultados de testes que validam a correção funcional do processador.

1 Visão Geral

Este projeto consiste na implementação de um processador RISC-V de 32 bits com um caminho de dados de ciclo único, desenvolvido como parte do trabalho prático da disciplina de **CSI211 - Fundamentos de Organização e Arquitetura de Computadores** na Universidade Federal de Ouro Preto (UFOP).

O design do processador foi baseado no diagrama de caminho de dados apresentado na Figura 4.21 do livro texto da disciplina, adaptado para um subconjunto específico de instruções.

1.1 Conjunto de Instruções Implementado (Grupo 32)

O processador foi projetado para decodificar e executar corretamente o seguinte conjunto de 7 instruções do padrão RISC-V:

Tabela 1: Conjunto de Instruções Implementado

Tipo	**Instrução**	**Descrição**
Load	`lh`	Carrega uma meia-palavra (16 bits) da memória.
Store	`sh`	Armazena uma meia-palavra (16 bits) na memória.
Tipo-R	`sub`	Subtrai o conteúdo de dois registradores.
Tipo-R	`or`	Realiza a operação OU bit a bit.
Tipo-I	`andi`	Realiza a operação E bit a bit com um imediato.
Tipo-R	`srl`	Realiza o deslocamento lógico de bits para a direita.
Tipo-SB	`beq`	Realiza um desvio condicional se dois registradores forem iguais.

2 Estrutura do Projeto

O design foi modularizado em vários arquivos Verilog para organização e clareza, representando os principais blocos lógicos de um processador:

- `riscv_processor.v`: O módulo de topo que conecta todos os outros componentes.
- `control_unit.v`: A unidade de controle principal, responsável por decodificar o opcode da instrução e gerar os sinais de controle.
- `alu_control.v`: Unidade de controle secundária que gera o sinal específico para a ULA com base no `ALUOp` e nos campos `funct`.
- `alu.v`: A Unidade Lógica e Aritmética, que executa as operações de cálculo.
- `reg_file.v`: O banco de registradores que armazena o estado dos 32 registradores do processador.
- `memory.v`: Um módulo que simula uma memória unificada para instruções e dados.

- `datapath_components.v`: Contém componentes auxiliares como multiplexadores, o gerador de imediato e somadores.
- `testbench.v`: O ambiente de simulação usado para verificar a funcionalidade do processador, pré-carregando um programa e monitorando a execução.

3 Como Compilar e Executar

Pré-requisito: Tenha o Icarus Verilog instalado e configurado no seu sistema.

3.1 Primeira forma de rodar o código:

1. **Rodar o arquivo .bat:** Nesse repositório existe um arquivo chamado `compila_e_executa_iterativo.bat`. Ao rodar ele você escolhe que tipo de teste fará de acordo com os dois testes que estão na pasta `/testes`.

3.2 Segunda forma de rodar o código:

1. **Preparar o Teste:** Certifique-se de que o arquivo `program.mem` contém o programa em hexadecimal que você deseja executar. Você pode usar o script `compila_e_executa_iterativo.bat` para escolher entre os testes pré-definidos.
2. **Compilar:** No terminal, na pasta raiz do projeto, execute o comando:

```
iverilog -o meu_processador.vvp testbench.v riscv_processor.v memory.v
    control_unit.v alu_control.v datapath_components.v reg_file.v alu.v
```

Listing 1: Comando de compilação

3. **Executar:** Após a compilação bem-sucedida, execute a simulação com:

```
vvp meu_processador.vvp
```

Listing 2: Comando de execução

A saída da simulação, incluindo o estado final dos registradores, será exibida no terminal.

4 Verificação e Testes

Dois programas de teste foram criados para validar o design:

- `Teste_simples.txt`: Realiza uma verificação fundamental das operações aritméticas (`or`, `andi`, `sub`) e do ciclo completo de acesso à memória (`sh` e `lh`).
- `Teste_todos_comandos.txt`: Um teste de estresse que utiliza todas as 7 instruções implementadas de forma interdependente, validando a lógica de deslocamento de bits (`srl`) e o controle de fluxo com desvios condicionais (`beq`).

Ambos os testes foram executados com sucesso, e os resultados nos registradores corresponderam aos valores esperados, validando a correção funcional do processador.

5 Relatório de Verificação do Processador RISC-V (Grupo 32)

Para validar o processador RISC-V de ciclo único que implementei, desenvolvi uma suíte de testes com o objetivo de verificar a correção funcional de cada componente e do caminho de dados como um todo. A seguir, detalho os dois principais programas de teste que foram executados com sucesso, demonstrando a robustez do design.

A simulação foi realizada em um ambiente onde o registrador `x5` é pré-carregado com o valor `100` (`0x64`) pelo `testbench`, servindo como ponto de partida para a geração de valores não-nulos.

5.1 Teste 1: Verificação Fundamental de Aritmética e Memória

O primeiro teste (`Teste_simples.txt`) foi projetado para ser uma verificação rápida e essencial do fluxo de dados principal, focando nas operações aritméticas básicas e no caminho de acesso à memória.

5.1.1 Código Assembly (Teste_simples.txt)

```
# Teste de verifica  o simples de SUB, ANDI, OR e acesso    mem ria

# Carrega 100 (de x5) em x6
or x6, x5, x0

# Cria a constante 4 em x7 (resultado de 100 & 20)
andi x7, x5, 20

# Calcula 100 - 4 = 96 e armazena em x8
sub x8, x6, x7

# Armazena o resultado (96) na mem ria no endere o 100
sh x8, 0(x5)

# L   o valor (96) de volta da mem ria para o registrador x9
lh x9, 0(x5)
```

Listing 3: Código Assembly - Teste Simples

5.1.2 Código Hexadecimal (Teste_simples.txt)

```
0002e333
0142f393
40730433
00829023
00029483
```

Listing 4: Código Hexadecimal - Teste Simples

5.1.3 Objetivo do Teste

O propósito deste teste era garantir que os componentes mais críticos do processador estavam funcionando em sequência.

1. **Validação da ULA:** Verifiquei se as instruções `or`, `andi` e `sub` eram executadas corretamente pela ULA.
2. **Integridade do Banco de Registradores:** Confirmei que o resultado de uma operação (`or`) era corretamente escrito em um registrador (`x6`) e podia ser lido como operando para a instrução seguinte (`sub`).
3. **Caminho de Dados da Memória:** O mais importante, este teste validou todo o ciclo de acesso à memória. A instrução `sh` precisava que o endereço (calculado pela ULA e vindo de `x5`) e o dado (vindo de `x8`) chegassem corretamente à memória. Em seguida, a instrução `lh` precisava ler o mesmo endereço, buscar o dado da memória e garantir que o `mux` de write-back selecionasse a memória como fonte para escrever o valor de volta em `x9`. O sucesso neste teste provou que o coração do processador estava funcional.

5.2 Teste 2: Verificação Completa do Conjunto de Instruções

O segundo teste (`Teste_todos_comandos.txt`) foi o "teste de estresse"final. O objetivo era criar um único programa que utilizasse **todas as 7 instruções** implementadas (`lh`, `sh`, `sub`, `or`, `andi`, `srl`, `beq`) de forma interdependente.

5.2.1 Código Assembly (Teste_todos_comandos.txt)

```
# Teste final de verifica  o para todas as instru  es do Grupo 32

# Bloco 1: Prepara  o e L gica
or x6, x5, x0          # x6 = 100 (Copia valor inicial)
andi x7, x5, 12         # x7 = 4   (Cria constante)
```

```
sub x8, x6, x7          # x8 = 96  (Teste de SUB)

# Bloco 2: Teste de Mem ria
sh x8, 0(x5)            # Mem ria[100] = 96 (Teste de SH)
lh x9, 0(x5)            # x9 = 96 (Teste de LH)

# Bloco 3: Teste de Deslocamento e Desvio
or x10, x9, x0          # x10 = 96 (Copia valor da mem ria)
andi x11, x5, 4         # x11 = 4  (Cria constante para deslocamento)
srl x12, x10, x11       # x12 = 6  (Teste de SRL: 96 >> 4)
beq x8, x9, 8           # Pula se 96 == 96 (Teste de BEQ)

# Bloco Final (s   executa se o BEQ falhar)
sub x13, x5, x5         # Esta linha deve ser pulada
or x0, x0, x0           # NOP para preencher espa o
```

Listing 5: Código Assembly - Teste Completo

5.2.2 Código Hexadecimal (`Teste_todos_comandos.txt`)

```
0002e333
00c2f393
40730433
00829023
00029483
0004e533
0042f593
00b55633
00940463
40d686b3
00006033
```

Listing 6: Código Hexadecimal - Teste Completo

5.2.3 Objetivo do Teste

Este programa foi desenhado como um "procurador de falhas"para validar a totalidade do meu design:

1. **Confirmação da Lógica e Memória:** Os primeiros blocos revalidam as operações `or`, `andi`, `sub`, `sh`, e `lh`, garantindo a base do processador.
2. **Validação da Instrução de Deslocamento:** A instrução `srl` foi testada para garantir que a ULA e o Controle da ULA pudessem lidar com operações de deslocamento de bits, que usam uma combinação diferente de sinais de controle.
3. **Validação do Controle de Fluxo:** O passo mais crucial foi testar a instrução `beq`. Este teste não apenas verificou se a ULA podia fazer a subtração para a comparação, mas também se o sinal `Zero` era gerado corretamente, se o `branch_condition` era calculado, e se o `mux` do PC selecionava o endereço de desvio correto, alterando o fluxo de execução do programa. O fato de o registrador `x13` ter permanecido `0` ao final da simulação provou que o desvio foi tomado com sucesso, validando todo o mecanismo de controle de fluxo do processador.

6 Resultados dos testes práticos:

6.1 Saídas da Simulação

Aquei nessa sessão iremos demonstrar por meio e imagens os resultados dos testes execudaots no cógigo, seguindo a ordem de demonstração do testes:

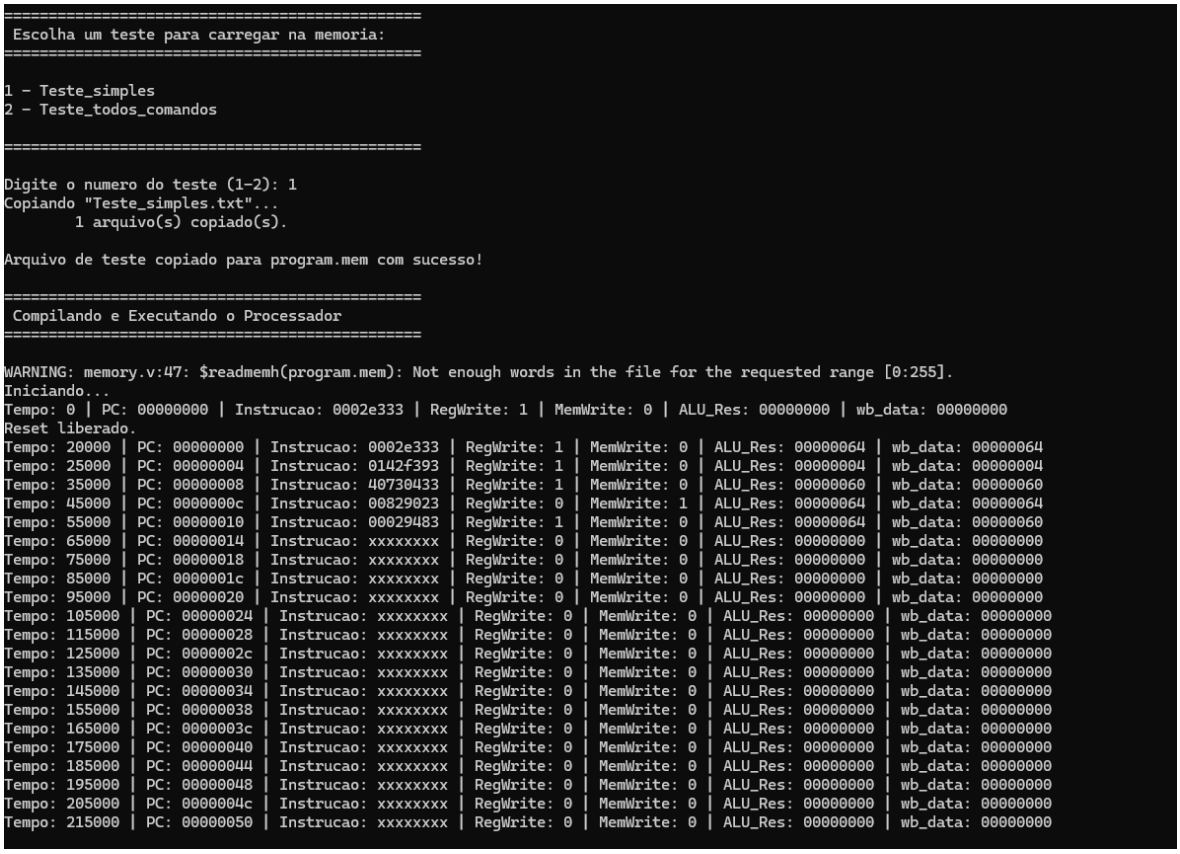

Figura 1: Saída da simulação do Teste Simples.
Esta imagem mostra o log de execução do Icarus Verilog para o programa `Teste_simples.txt`, detalhando o estado do PC, instrução, registradores e memória em cada passo.

Figura 2: Estado final dos registradores após o Teste Simples.
Esta imagem exibe os valores finais de todos os registradores após a execução completa do programa `Teste_simples.txt`, validando os resultados esperados.

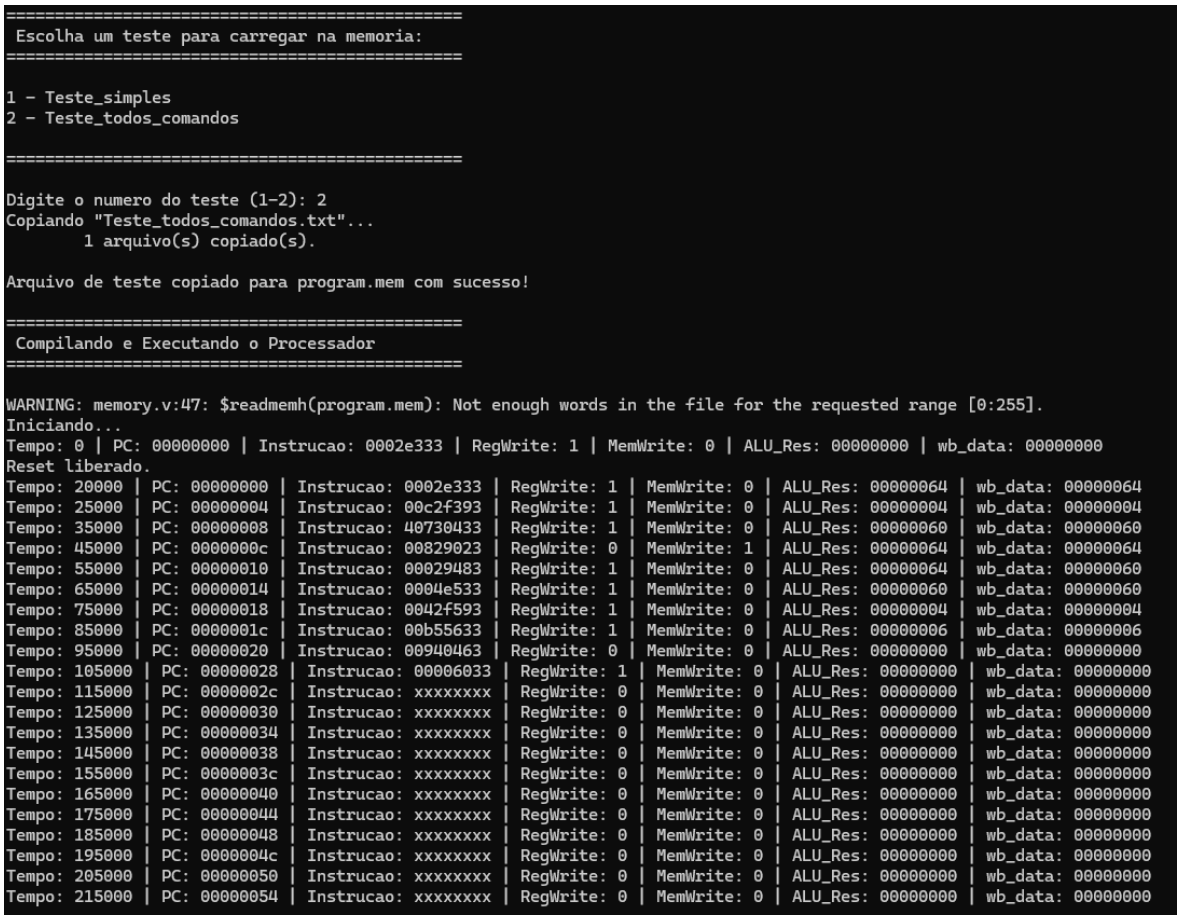

Figura 3: Saída da simulação do Teste Completo.
Esta imagem mostra o log de execução do Icarus Verilog para o programa `Teste_todos_comandos.txt`, detalhando o estado do PC, instrução, registradores e memória em cada passo.

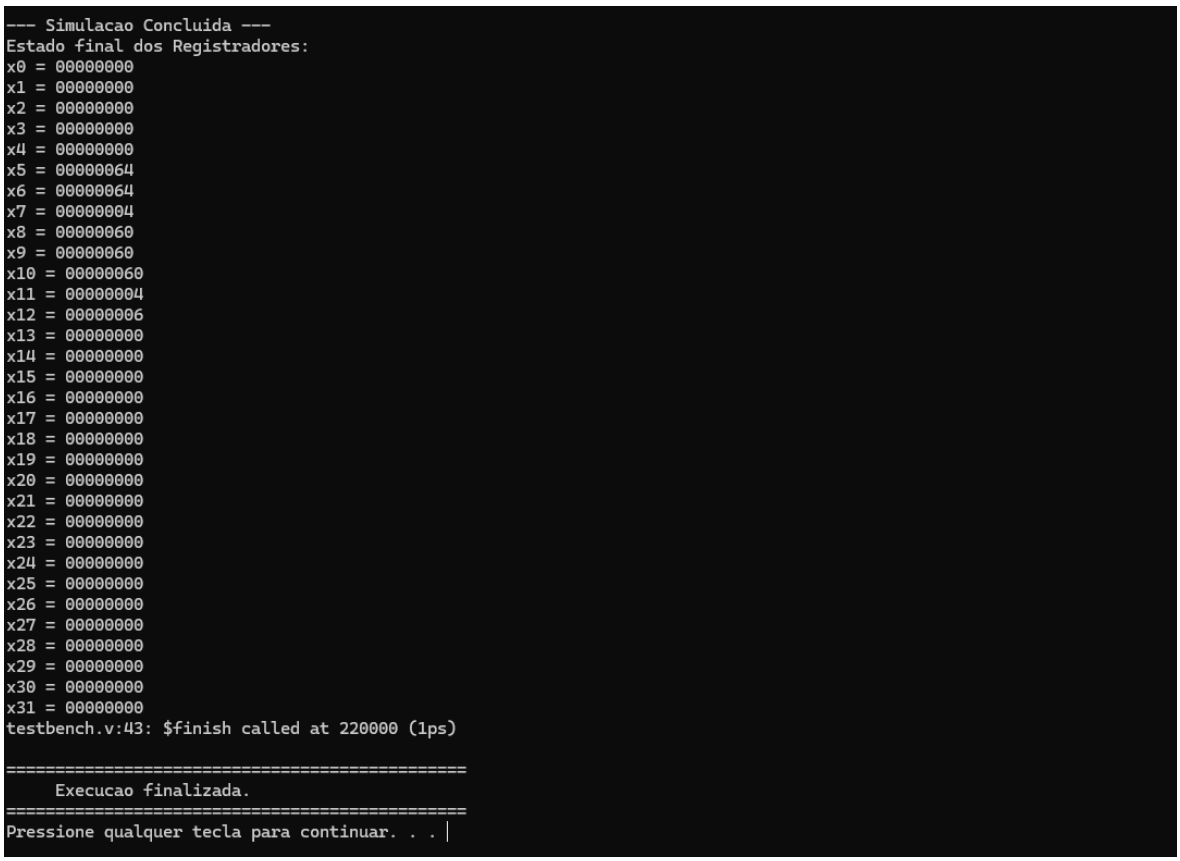

Figura 4: Estado final dos registradores após o Teste Completo.
Esta imagem exibe os valores finais de todos os registradores após a execução completa do programa `Teste_todos_comandos.txt`, validando os resultados esperados.