Caminho de Dados RiscV - Versão Simplificada Iverilog

Ayko B. Colaço -5785 [1], **Pedro Elecio F. Realino - 5786**[2]

[1]Instituto de Ciências Exatas e Tecnológicas – Universidade Federal de Visoça (UFV - Campus Florestal)

ayko.colaco@ufv.br, pedro.realino@ufv.br

Resumo. *Descrição do segundo Trabalho Prático de Organização de Computadores, o qual tinha como objetivo a implementação de uma versão simplificada de alguns caminhos de dados do RISC-V.*

Abstract. *Description of the second Practical Assignment for Computer Organization, which aimed at implementing a simplified version of some RISC-V data paths.*

1. Introdução

O propósito deste trabalho é desenvolver uma versão simplificada de um caminho de dados do processador RISC-V utilizando Verilog, (linguagem de descrição de hardware).

Cada grupo foi encarregado de sete instruções diferentes do caminho de dados e tinha como objetivo elaborar sua implementação em Verilog. Por fim, é necessário mostrar no terminal o mapa com registradores e seus valores após as operações.

Para a realização desse trabalho, além do material disponibilzado pelo professor no Moodle, também foi usado de auxílio vídeos, fóruns e ferramentas da internet . Todo o trabalho foi desenvolvido e testado em um notebook com sistema operacional Windowns 11. A IDE utilizada foi o Visual Studio Code, com auxílio de algumas extensões específicas para Verilog, além claro, do compilador Icarus Verilog que já nos era mais familiar.

2. Execução do Codigo

Para executar a simulação do caminho de dados RISC-V, primeiro é necessário instalar o compilador Icarus Verilog, além da ferramenta gtkwave para a visualização da simulação. Para o codigo em si é nescessario, uma instrução, ou lista de instruções RISC-V em binario, que devem ser escritas dentro do arquivo (Instruction-Memory), por padrão deixamos 8 intruções. Caso seja preciso usar outras instruções, seguir o formato: Instruction-Mem[N] = 32'bInstrução em binario, para N instruções.

Para executar a simulação é preciso estar no diretorio (TP2OC-CAMINHO-DE-DADOS-RISC-V) então digitar o comando make, que executara automaticamente o codigo, para a visualização de ondas, digitar (gtkwave exec.vcd).

3. Desenvolvimento

Para podermos implementar o caminho de dados, nós separamos o caminho em algumas partes, e implementamos por modulos(ALU.v, Instruction-Memory, Imm-gem, Data-Memory, Program-Counter, Banco-de-Registradores, Unit-Control e Caminho-de-Dados), a seguir um resumo de cada modulo:

3.1. ALU.v

O módulo ALU é projetado para realizar operações aritméticas e lógicas em dois operandos de 32 bits, X e Y. Ele recebe um sinal de controle de 3 bits, ALUop, que determina qual operação a ALU deve executar. As operações que são: AND , OR , ADD, SUB e SRL. O resultado é armazenado na saída Resultado-ALU, e um sinal zero indica se o resultado é zero.

3.2. Instruction-Memory

O módulo Instruction-Memory é responsável por fornecer instruções a partir de uma memória de instruções com base em um endereço de 32 bits. A memória contém 256 posições(0 a 255), cada uma armazenando uma instrução.

Quando um endereço é fornecido como entrada, o módulo usa os 8 bits menos significativos desse endereço para acessar a posição correspondente na memória. A instrução armazenada nessa posição é então disponibilizada na saída Instrucao.

3.3. Imm-gem

O módulo Imm-gen é projetado para gerar valores imediatos de 32 bits a partir de uma instrução de 32 bits, dependendo do tipo da instrução(Tipo I, S e B). O módulo usa um case para identificar o tipo de instrução com base nos 7 bits menos significativos da instrução para selecionar o valor imediato correspondente. Se a instrução não corresponder a um dos tipos esperados, o módulo define o valor imediato como zero.

3.4. Data-Memory

Este módulo implementa uma memória de dados simplificada, permitindo operações de leitura e escrita em um vetor de memória, ele le os dados de um arquivo (Memoria-Dados.bin) e armazena no vetor.

A escrita na memória é controlada por um sinal (Mem-Write). Quando esse sinal está ativado (1), e ocorre uma borda de subida no clock, o valor em Write-Data é gravado na posição de memória especificada pelo endereço(Endereco).

A leitura da memória é controlada por umsinal Mem-Read. Quando esse sinal está ativado (1), o valor armazenado na posição de memória indicada pelo endereço é lido e passado para Read-Data. Se o sinal Mem-Read estiver desativado, Read-Data é definido como zero.

3.5. Program-Counter

O módulo Program-Counter é responsável por armazenar o endereço da próxima instrução. Em cada borda de subida (posedge) do sinal de clock , se o reset não estiver ativo, o valor de pc é atualizado para o valor de pc-next (endereco da prxima instrução)

3.6. Banco-de-Registradores

O módulo Banco-de-Registradores vria um vetor de 32 registradores de 32 bits, permitindo leitura e escrita de valores nesses registradores, permitindo que diferentes instruções acessem e modifiquem os valores dos registradores conforme necessário durante a execução do programa.

3.7. Unit-Control

O módulo Unit-Control gera sinais de controle necessários para a execução das instruções. A partir dos valores de opcode, funct3, e funct7, ele determina como a unidade de execução deve operar, configurando sinais como ALUop, Mem-Read, Mem-Write, Reg-Write, ALU-src, Mem-to-Reg, e branch. O módulo inicializa os sinais de controle zerados e, em seguida, usa um bloco case para definir os sinais corretos com base no opcode.

3.8. Caminho-de-Dados

Esse módulo integra diversos componentes fundamentais, cada um responsável por uma parte do processo de execução de instruções, ele instancia os outros modulos, e integra as componentes para o codigo funcionar.

4. Resultados

Para verificação de resultados, foram utilizados um conjunto de instruções RISC-V em binário (Figure 1), e o vetor de memoria de dados foi preenchido com os dados de um arquivo Memoria-Dados.bin (Figure2), o resultado da execução e valores dos registradores podem ser vistos na (Figure 3) e a visualização de ondas na (Figure 4).

```
//Instruções que deixamos de padrão.
 Instruction_Mem[0] = 32'b00000000101100000000000001110010011; // addi x7, x0, 11
 Instruction_Mem[1] = 32'b00000000000010000000000000010000011; // lb x1, 1(x0)
 Instruction_Mem[2] = 32'b00000000000100000100000100100011; // sb x2, 2(x1)
 Instruction_Mem[3] = 32'b01000000001000001000000110110011; // sub x3, x1, x2
 Instruction_Mem[4] = 32'b00000000000010001011100100110011; // and x4, x2, x1
 Instruction_Mem[5] = 32'b00000000000010000111000101010011; // ori x5, x1, 1
 Instruction_Mem[6] = 32'b00000000000010001010100110010011; // srl x6, x2, 1
 Instruction_Mem[7] = 32'b00000000000010001000000000011100011; // beq x1, x2, 1
```

Figure 1. Arquivo de entrada(Inttruções)

```
00000000000000000000000000000001
00000000000000000000000000000010
00000000000000000000000000000011
00000000000000000000000000000100
00000000000000000000000000000101
00000000000000000000000000000110
00000000000000000000000000000111
00000000000000000000000000001000
00000000000000000000000000001001
00000000000000000000000000001010
00000000000000000000000000001011
00000000000000000000000000001100
00000000000000000000000000001101
00000000000000000000000000001110
00000000000000000000000000001111
00000000000000000000000000010000
00000000000000000000000000010001
00000000000000000000000000010010
00000000000000000000000000010011
00000000000000000000000000010100
00000000000000000000000000010101
00000000000000000000000000010110
00000000000000000000000000010111
00000000000000000000000000011000
00000000000000000000000000011001
00000000000000000000000000011010
00000000000000000000000000011011
00000000000000000000000000011100
00000000000000000000000000011101
00000000000000000000000000011110
00000000000000000000000000011111
00000000000000000000000000100000
00000000000000000000000000100001
00000000000000000000000000100010
00000000000000000000000000100011
00000000000000000000000000100100
00000000000000000000000000100101
00000000000000000000000000100110
```

Figure 2. Memoria de Dados

```
Teste completo.
==================INICIO==================
Registrador[0]  0
Registrador[1]  2
Registrador[2]  0
Registrador[3]  2
Registrador[4]  0
Registrador[5]  3
Registrador[6]  0
Registrador[7]  11
Registrador[8]  0
Registrador[9]  0
Registrador[10] 0
Registrador[11] 0
Registrador[12] 0
Registrador[13] 0
Registrador[14] 0
Registrador[15] 0
Registrador[16] 0
Registrador[17] 0
Registrador[18] 0
Registrador[19] 0
Registrador[20] 0
Registrador[21] 0
Registrador[22] 0
Registrador[23] 0
Registrador[24] 0
Registrador[25] 0
Registrador[26] 0
Registrador[27] 0
Registrador[28] 0
Registrador[29] 0
Registrador[30] 0
Registrador[31] 0
==================FIM==================
```

Figure 3. Resultado no terminal

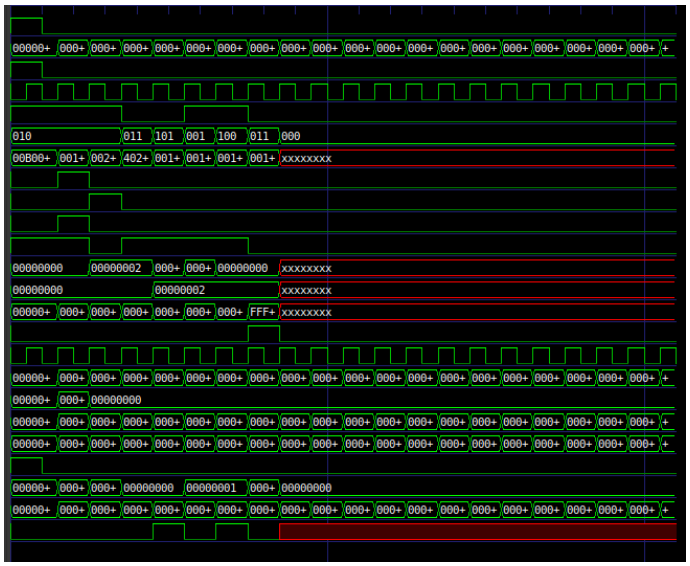

Figure 4. Resultado no gtkwave

5. Repositorio GitHub

O Nosso codigo esta desponivel no seguinte Repositorio GitHub:
https://github.com/oc-ufv/-tp01-5785-5786

6. Referencias

https://gemini.google.com/app
https://chat.openai.com/
https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/
https://awari.com.br/string-funcoes-de-string-em-python/
https://bleyer.org/icarus/
https://docs.google.com/presentation/d/1JOFFnvNb5eB2nUv1TNRIbNGpcxBhNR0QgvWHinsLBiQ/edits
Patterson, D.A.; Hennessy J.L. Computer Organization and Design: RISC-V Edition, 2ª
Edição, Editora Morgnan Kaufman, 2021.