Implementação do Caminho de Dados do RISC-V

Alice Ladeira Gonçalves
Sabrina Bruni de Souza Faria

[1]Universidade Federal de Viçosa - Campus Florestal (UFV)
Minas Gerais – MG – Brasil

Abstract. *This documentation contains information about the implementation of the RISC-V data path, that is, from reading a file containing the instructions, the program makes the data path for each instruction. In addition to explaining how it works, information about the implementation choices and how this influences the results obtained will also be present here.*

Resumo. *Essa documentação contém informações acerca da implementação do caminho de dados do RISC-V, ou seja, a partir da leitura de um arquivo contendo as instruções, o programa faz o caminho de dados para cada instrução. Além de explicar o seu funcionamento, aqui também estarão presentes informações sobre as escolhas de implementação e como isso influencia nos resultados obtidos.*

1. Introdução

Esse trabalho consiste na implementação do caminho de dados do processador RISC-V de 32 bits, na qual cada instrução, codificada em sua respectiva sequência de bits, é lida e seus comandos são executados. A implementação do caminho de dados é composta por dois arquivos em verilog, sendo eles o testbench (”testbench.v”) e o caminho de dados (”caminho_dados.v”) em si, o arquivo com as instruções em linguagem natural (”instrucoes.asm”), arquivo com as instruções codificadas em binário (”binario.asm”). Vale ressaltar que os resultados obtidos no TP 1 foram usados e ao longo dessa documentação dissertaremos de forma mais detalhada sobre, bem como sobre as escolhas de implementação e como isso influencia nos resultados finais.

2. Desenvolvimento

O trabalho passou por várias etapas ao longo do seu desenvolvimento. A fim de facilitar a compreensão, a explicação será dividida entre: Formas de entrada e saída, Funcionamento geral do caminho de dados e Detalhes de implementação.

2.1. Formas de entrada e saída

Como entrada o “caminho_dados.v” recebe um arquivo contendo instruções em linguagem de máquina. É importante destacar que esse arquivo foi gerado pelo primeiro trabalho prático, ou seja, o montador que fizemos no primeiro tp recebe as instruções em linguagem natural e gera um arquivo com as instruções em binário. Após ler cada instrução do arquivo e executar seu respectivo caminho de dados, o resultado, o qual gerou modificações nos registradores ou memória, a depender do tipo de instrução, é printado no terminal no formato pré-definido. A cada instrução, tanto os registradores quanto a memória são printados, mesmo que não tenham sofrido alterações. Cabe destacar que tanto a leitura do arquivo de entrada quanto o print de saída são feitos no “testbench.v”.

2.2. Funcionamento geral do caminho de dados

A implementação do caminho de dados está separado entre dois arquivos em verilog, o "caminho_dados.v" e o "testbench.v". O primeiro arquivo apresenta a descrição de um sistema com vários módulos interconectados. Cada módulo tem uma função específica no sistema. Abaixo estão listados os módulos, juntamente com uma descrição da sua função no código.

1) O módulo PC tem duas partes principais. A primeira é sensível ao sinal de reset e atribui 0 ao sinal de saída linha quando ocorre uma borda de subida no sinal de reset. A segunda parte é sensível à borda de descida do sinal de clock. Se line for igual a 0, linha recebe o valor de sum. Caso contrário, linha é incrementado em 1. Quando o clock vale 0, a leitura de uma linha e mudança para a próxima são realizadas e quando há mudança de intrução, o rd, rs1, rs2 e opcode dela são separados e guardados.

2) O módulo Control é sensível à borda de subida do sinal de clock. Ele atribui valores apropriados às saídas com base no valor de opcode. Existem várias condições if e else if para diferentes valores de opcode. Cada condição configura as saídas de acordo com a função específica da instrução correspondente. Dentro desse módulo, o "opcodefield" e "funct3" são utilizados para identificar o tipo de operação das instruções do tipo imediato, exceto load. A variável "control" é usada para validação do final da comparação entre instruções lida e seu respectivo tipo.

3) O módulo Registradores tem três partes principais. A primeira parte copia os valores de entrada Registers_entrada para as saídas Registers_saida. A segunda parte é sensível à borda de subida do sinal "mem" e realiza a escrita no registrador, sendo copiada da memória ou resultado da ALU. Já a terceira parte é sensível à borda de subida do sinal control e atribui valores apropriados às saídas data1, data2, REG e WriteMem, dependendo dos sinais de controle e dos valores de entrada.

4) O módulo ALU tem duas partes principais. A primeira parte é sensível à borda de subida do sinal REG e executa a operação apropriada da ALU com base no valor de ALUcontrol. A segunda parte é sensível à borda de subida do sinal alu realizando o desvio a depender se é do tipo branch e se os valores comparados são iguais. Além disso, atribui um valor para line, o qual indica ao pc se haverá um desvio ou continuará operando as instruções em sequência.

5) O módulo Memory tem duas partes principais. A primeira parte copia os valores de entrada Memoria_entrada para as saídas Memoria_saida. A segunda parte é sensível à borda de subida do sinal alu, caso seja de escrever na memória, atribui o valor do registrador correspondente, se for apenas de leitura, atribui o valor que será escrito no registrador.

As variáveis "mem", "control", "alu", "reg" e "line" são utilizadas como variáveis de controle dentro dos módulos.

O caminho de dados é responsável por realizar as operações aritméticas e lógicas, o acesso à memória e a manipulação de registradores. Já o arquivo "testbench.v" implementa o caminho de dados. O código pode ser dividido em seções:

1. Declaração de variáveis: São declaradas várias variáveis inteiras e registradores para armazenar os sinais e dados usados no processador.

2. Declaração dos módulos: São instanciados os módulos, como PC (Program Counter), Control, Registers, ALU (Arithmetic Logic Unit) e Memory. Esses módulos representam os componentes do processador e são conectados através dos sinais e entradas/saídas definidos.

3. Blocos “always”: Existem vários blocos ”always” que são executados quando os sinais especificados mudam de valor. Esses blocos são responsáveis por atualizar os registradores, exibir valores na saída e realizar a leitura de instruções de um arquivo.

4. Bloco “initial”: O bloco “initial” é executado apenas uma vez no início da simulação. Ele inicializa o reset e varia o clock até atingir o fim do arquivo.

Em resumo, o código implementa a lógica de controle e o caminho de dados de um processador simples. Ele busca instruções em um arquivo binário, executa essas instruções por meio do caminho de dados e exibe os valores dos registradores e memória durante a execução.

2.3. Alterações para funcionamento na FPGA

Algumas alterações foram feitas nos códigos entregues na primeira parte para que os resultados fossem apresentados na FPGA. Em primeiro lugar, é importante destacar que os arquivos foram renomeados e agora são ”nome_arquivo.sv”, visto que a extensão do Verilog, SystemVerilog fornece recursos mais avançados.

Em relação aos componentes da FPGA, o botão 0 foi destinado ao clock e o botão 1 ao reset, o swap 0 é para registrador e swap 1 para memória, o led 1 indica que os valores mostrados são referentes ao registrador e o led 2 representa a memória. Por fim, em relação aos displays, dividimos da seguinte forma: hex 7 e hex 6 indicam a linha, hex 5 e hex 4 a posição do registrador ou memória e os 4 últimos, hex 3, hex 2, hex 2 e hex 0, servem para mostrar o valor armazenado na memória ou registrador.

As mudanças feita no testebench foram as interações com os displays, para que os resultados pudessem ser apresentados. Já no arquivo ”caminho_dados.sv”, o modulo instructions é responsável por armazenar as instruções em um vetor chamado ”instructions”. As instruções são inicializadas dentro de um bloco ”always”acionado por um sinal de reset. Também há uma lógica que seleciona uma instrução do vetor com base na entrada ”linha”.

Por último, foi criado um arquivo exlusivamente para manipulação dos displays, o ”Display.sv”. O código é um módulo chamado ”SevenSegmentConverter”que converte um número decimal de 2 dígitos em sua representação de display de sete segmentos. Ele possui entradas para o número de entrada e saídas para os segmentos do display.

O bloco ”always”no módulo contém a lógica de conversão, verificando o número de entrada e determinando os valores dos segmentos de saída com base nos dígitos das dezenas e unidades. Se o número estiver fora do intervalo válido, os segmentos de saída serão definidos como ”1111111”.

A segunta parte é uma extensão do primeiro módulo, chamada ”SevenSegmentConverter2”. Ela possui entradas e saídas adicionais para controlar dois conjuntos de números de entrada, bem como displays adicionais e uma saída para um LED.

Apesar do funcionamento se apresentar como esperado no testbench, não obtive-

mos os resultados esperados na FPGA, visto que a linha não era incrementada e as demais partes do código dependem dela, direta ou indiretamente.

Abaixo podemos ver registros do funcionamento do código no testebench.

Figura 1. Display indicando a linha de determinada instrução

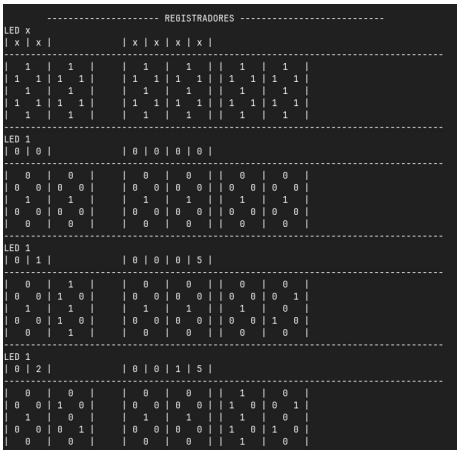

Figura 2. Displays indicando os valores armazenados nos registradores

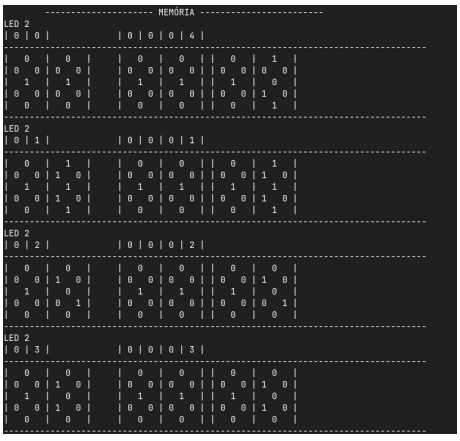

Figura 3. Displays indicando os valores armazenados na memória

2.4. Detalhes de implementação

É válido ressaltar que o caminho de dados suporta, além das instruções pedidas: load, store, sub, and, ori, srl e beq, também é capaz de operar o caminho de dados de algumas instruções extra: OR, ANDI e ADD. Além disso, o código possui registrador e memória de entrada (Registers_entrada e Memoria_entrada) e registrador e memória de saída (Registers_saida e Memoria_saida). Essa escolha de implementação se fez necessária por se tratarem de vetores e ser necessário manipulá-lo tanto na entrada, para receber dados, quanto na saída, para fornecer dados.

3. Resultados

Os resultados obtidos foram condizentes com o esperado, visto que foi possível atender a todos os critérios pedidos na especificação. Ademais, foi possível, através da realização do trabalho, compreender melhor o funcionamento acerca do caminho de dados. Nas imagens abaixo é possível ver o resultado obtido após a execução das instruções listadas.

```
add x1, x2, x3
lb x2, 0(x10)
sub x4, x2, x5
and x8, x4, x6
ori x2, x7, 8
add x6, x8, x4
beq x4, x8, -4
srl x10, x10, x4
sb x8, 0(x10)
```

Figura 4. Conjunto de instruções utilizadas para teste

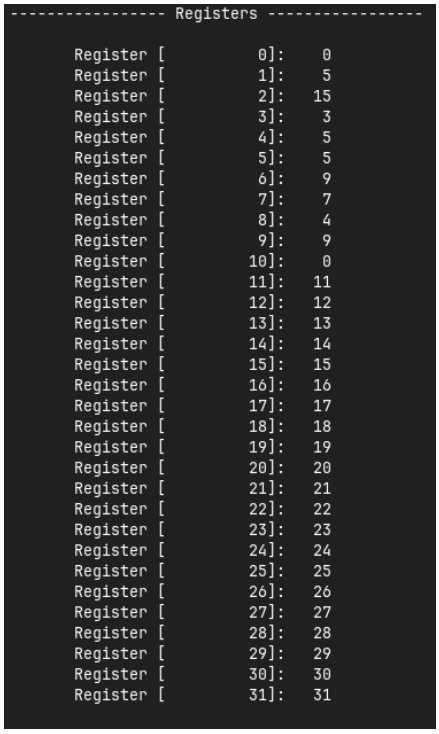

Figura 5. Resultado dos registradores pós execução das instruções

4. Considerações finais

Apesar do sucesso final, algumas dificuldades surgiram ao longo do desenvolvimento, entre elas a sincronização, sendo necessário criar variáveis para verificação do

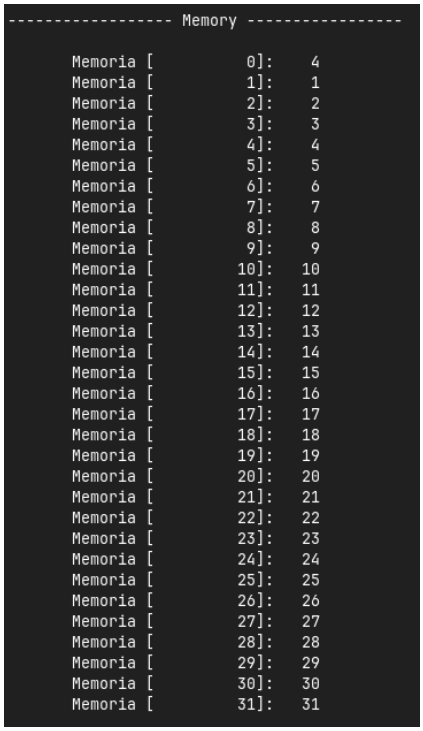

Figura 6. Resultado da memória pós execução das instruções

término ou não da execução de cada módulo e a própria linguagem, já que foi necessário realizar tarefas mais complexas, como leitura de arquivos, e as informações de como fazê-lo não estavam disponíveis na documentação da linguagem.

5. Referências

[1] Site da documentação do Verilog: https://www.chipverify.com/verilog/verilog-file-io-operations