Relatório Trabalho Prático - Caminho de Dados RISC-V

Erian Alves[1], **Guilherme Sérgio**[2], **Pedro Cardoso**[3]

[1]Instituto de Ciências Exatas e Tecnológicas,
Universidade Federal de Viçosa, Florestal, MG, Brasil

1. Introdução

Um Caminho de Dados é um elemento do processador constituído de todos os componentes responsáveis pela execução das operações elementares sobre os dados (transformações nos dados).

O objetivo do trabalho é implementar uma versão simplificada do caminho de dados do RISC-V. A imagem a seguir ilustra o caminho.

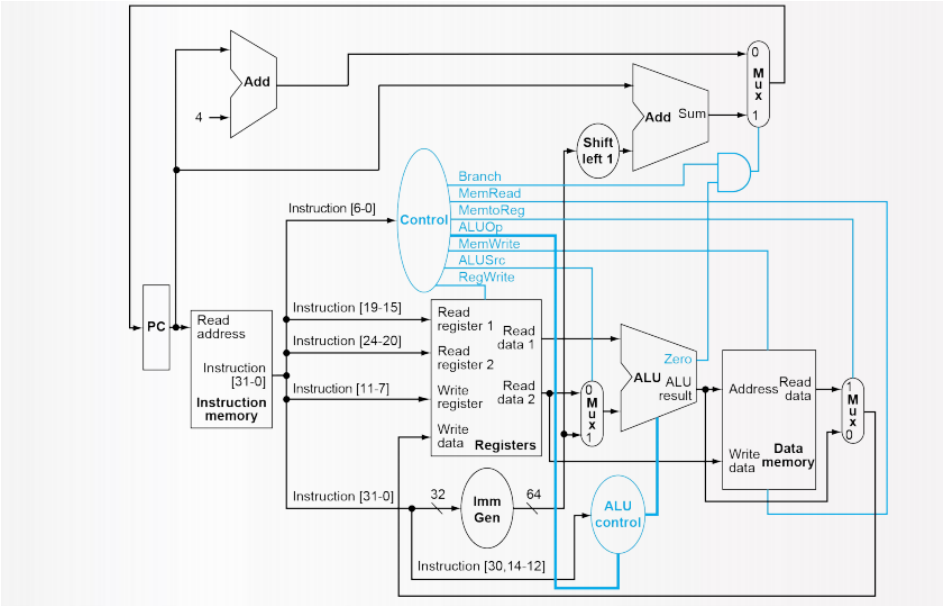

Figura 1. Caminho de dados.

2. Desenvolvimento

O caminho de dados solicitado neste trabalho foi construído de maneira a suportar a execução das seguintes instruções: add, sub, or, and, ld, sd e beq. Antes de realizar a devida implementação fez-se necessário compreender como ocorre o fluxo de execução das instruções e os elementos que compõe o caminho.

2.1. Comportamento caminho de dados

Para o entendimento do funcionamento do caminho de dados iremos nos basear nos seguintes passos de execução de uma instrução: busca, execução e resultado. Por fim iremos explicar como o fluxo do caminho de dados funciona.

Busca: Para realizar a busca são necessários três componentes: dois elementos de estado e um somador. Um dos elementos de estado armazena instruções (memória de

instruções) e o outro armazena o endereço da próxima instrução a ser executada (Program Counter - PC). O terceiro componente calcula o endereço da próxima instrução a ser executada (somador).

O PC é um registrador de 64 bits que é atualizado a cada ciclo de clock. A tarefa do somador é incrementar o valor do PC. Ele pode ser visto como uma unidade lógica aritmética (ALU) muito simples que só realiza operação de soma. A ALU é um hardware capaz de realizar de somas, subtrações e, geralmente, operações lógicas AND e OR.

Execução: Na parte de efetuar as operações, há seis componentes que atuam neste estágio: o banco de registradores, um elemento de estado que consiste em um grupo de registradores que podem ser lidos e escritos; o extensor de sinal, o qual tem a função de aumentar o tamanho do item de dado replicando o bit de maior ordem; a unidade de controle, responsável por comandar o funcionamento de outros componentes por meio de sinais estabelecidos de acordo com a instrução; um multiplexador de entrada dois, incumbido de selecionar um dos dados inseridos, de acordo com um sinal; a unidade de controle da ALU, a qual indica para a ALU qual a operação a ser executada com base na em informações da instrução; e a unidade lógica aritmética, aqui desempenhando as quatro funções descritas anteriormente.

Resultado: Finalizando o caminho de dados, temos as seguintes unidades: a memória de dados, a qual armazena informações e é acessada por instruções load e store; dois multiplexadores, um somador, uma porta lógica AND e um elemento para realizar um deslocamento para a esquerda em um valor númerico.

Funcionamento: Depois de conhecido os componentes que cada passo contém, podemos entender o funcionamento do caminho de dados. O fluxo começa com a busca, onde o PC é atualizado e manda o enderço da instrução correta a ser executada para a memória de instruções. Após isso entramos na parte de execução, onde já conhecemos qual instrução executar. Nessa etapa os registradores corretos são selecionados no banco de registradores e o controle manda informações para os Muxes e o controle da ALU, assim a ALU pode realizar a sua devida operação e passar o resultado adiante. Por fim, chegamos na etapa do resultado, no qual o valor gerado na execução é escrito na memória de dados, em um registrador ou acontece um desvio.

2.2. Implementação

No quesito da implementação, foi utilizada a linguagem de descrição de hardware Verilog e a ferramenta de simulação e síntese Icarus Verilog, a qual opera como um compilador para o código fonte escrito na linguagem. Com o objetivo de proporcionar uma melhor organização na construção do caminho de dados, o trabalho foi desenvolvido em módulos separados, sendo eles: *SettingPC*, *InstructionMemory*, *Control*, *RegisterFile*, *Extending-Sign*, *Mux2*, *ALUControl*, *RISCVALU*, *MemoryData*, *ShiftRight1* e *NextPC*. Além disso, eles foram separados em três arquivos, Inicio.v, PreparingToExecute.v e ExecutingOperations.v, seguindo o paralelo estabelecido na seção anterior, e também o simulacao.v para a simulação.

No processo de codificação, foi utilizado construtos tanto de lógica combinacional, quanto sequencial. Relevante ressaltar que o uso dessa última possibilitou que determinados trechos executassem apenas nos momentos corretos, de acordo com o fluxo estabelecido pelo caminho de dados, devido a natureza da execução de uma linguagem

de descrição de hardware ser simultânea. Em relação a metodologia de clock, seguiu-se o padrão adotado no livro texto, no qual todas a mudanças de estado ocorrem na borda do clock.

Os módulos *Control*, *Mux2*, *ALUControl*, *RISCVALU* e *ShiftRight1* são equivalentes aos seguintes componentes do caminho de dados: unidade de controle, multiplexadores, unidade de controle da ALU, unidade lógica e aritmética e o elemento encarregado de realizar divisões por 2. Devido a natureza de suas funcionalidades, realizar cálculos e definição de sinais, não exigiram um desenvolvimento além do que a teoria apresenta, por isso uma descrição de suas estruturas seria uma repetição do conteúdo abordado anteriormente. Já se tratando dos demais, há pontos e detalhes importantes de serem comentados, os quais possibilitaram um correto funcionamento da simulação, além da condensação de alguns funções em um único módulo.

O *SettingPC*, incumbido de atualizar o valor do PC a cada subida de clock, é um dos módulos que recebe o sinal de "reset", e tem por objetivo definir o valor do registrador contador de programa como zero, para que ele tenha um local inicial a qual se referir na memória de instrução. Outros que respondem a esse sinal são a *MemoryData*, o *RegisterFile* e a *InstructionMemory*, os quais o utilizam para definir o momento em que os registradores que formam as suas unidades armazenadoras de dados devem ser inicializados. A sintaxe para estes itens, usando o tipo de dado mencionado é a seguinte:

reg [TamanhoDoDado] memoria [QuantidadeDeUnidades]

O ponto a ser destacado aqui é a maneira como a matriz de registradores é inicializada em cada módulo. Enquanto nos dois primeiros é utilizado um laço de repetição *for*, estabelecendo todas as posições com um mesmo valor. Na *InstructionMemory*, para que fosse possibilitado a simulação de diversas instruções sem a necessidade de definir a sua identificação de cada uma manualmente, fez-se o uso de uma função da linguagem Verilog para leitura de arquivos e armazenamento de seu conteúdo nos registradores. É possível observar o comando na figura a seguir.

```
$readmemb("bin_memory_file.mem", memory_array, [start_address], [end_address])
```

Figura 2. Comando de leitura de um arquivo.

Outro detalhe a ser ressaltado se encontra na *MemoryData*. Para que fosse possível exibir as 32 primeiras posições de memória, houve a necessidade de que um fio armazenase todo o conteúdo da memória devido a uma limitação no Verilog, o qual não permite que matrizes sejam passados como parâmetros de módulos. A fim de efetuar a tarefa citada utilizou-se de um recurso da linguagem chamado *vector bit-select and part-select addressing*, o qual possibilita, seguindo a sintaxe a seguir, uma atribuição automática do conteúdo da memória para o fio.

fio[TamanhoDoElemento*j +: TamanhoDoElemento] = MemoriaDeDAdos[j];

Importante destacar que o tamanho das instruções trabalhadas são de 32 bits, enquanto todos os cálculos envolvendo endereços e a unidade de lógica aritmética são de 64 bits. Em relação ao módulo *ExtendingSign*, esse incumbido de extrair o imediato

da instrução e convertê-lo para 64 bits, foram utilizados os recursos de concatenação e replicação da linguagem para que a manipulação dos bits da instrução fosse mais prática. Por fim, a estrutura que completa o caminho de dados implementado neste trabalho, é o *NextPC*. Esse módulo é o responsável por realizar diversas tarefas, as quais finalizam o ciclo de execução de uma instrução. Nele é exercida a operação de incremento do contador de programa para a instrução seguinte e também com o imediato da instrução, além disso, é também feita a verificação com sinais de saída da unidade de controle e da ALU para identificar se ocorrerá um desvio.

Sabendo como o código foi criado, é importante citar uma inconveniência adicionada no código. Como o caminho de dados manuseia as memórias de uma forma diferente da presente neste trabalho, o Program Counter teve que sofrer algumas adaptações. Para realizar a mudança de instrução, tanto para quando não há desvios quanto para quando existe algum desvio, dividiu-se o offset do Program Counter por 4, assim o endereçamento é feito de forma sequencial na matriz de instruções. Dessa forma, uma instrução como *beq x0, x0, 16*, se torna *beq x0, x0, 4*, seguindo 4 instruções à frente normalmente. Conhecendo o fato de que o montador já realiza um *shift right* no imediato de inistruções de desvio, a divisão por 4 foi feita adicionando um *shift right* no lugar do *shift left* padrão do caminho de dados. Dessa forma foi possível receber o binário gerado por algum montador sem maiores problemas.

3. Resultados

A partir do código proposto é interessante tentar observar se o design do hardware está correto, uma boa opção é realizar um testbench, inserindo inputs selecionados e verificar se os outputs correspondem com o esperado. Para ter uma melhor visualização utilizamos também o software Gtkwave, que nós fornece informações sobre as ondas, valores entre outros.

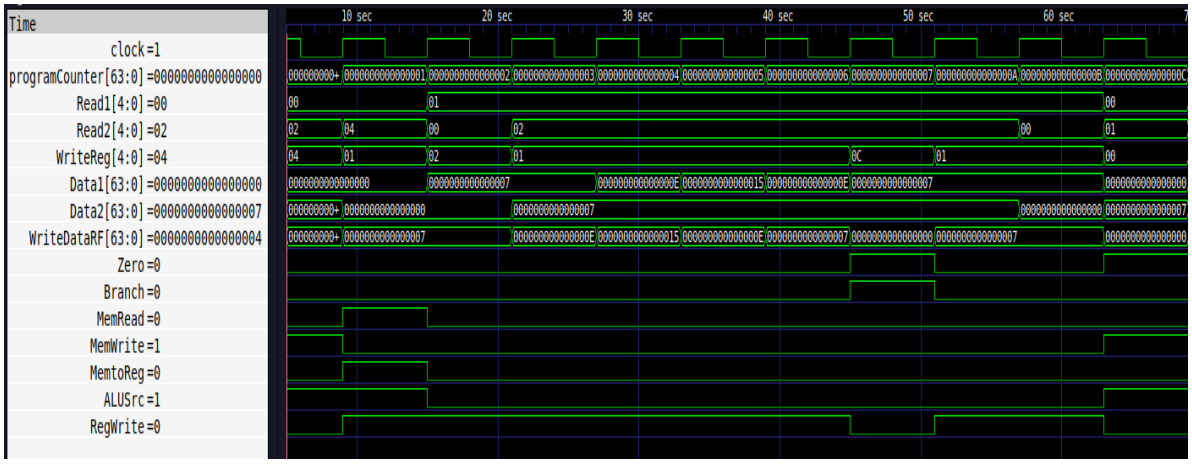

Figura 3. Ondas produzidas pelo código criado

A Figura 3 mostra as ondas para uma seleção predefinida de instruções, que foi passada como exemplo na especificação do trabalho. A região em branco mostra os valores do primeiro ciclo de clock, que é um pouco difícil de visualizar nas ondas.

Para a realização do teste, como o caminho de dados não abrange instruções com imediatos, os registradores foram inicializados manualmente. A inicialização manual

foi feita colocando 0 no registrador como seu valor. Exceto pro registrador x2, que foi iniciado com o valor 7, para questões de teste. Vale citar também que na Figura 3 não contém todos os sinais, foram selecionados apenas os mais importantes de se mostrar, como o número do registradores (Read1 e Read2) que são usados e seus valores (Data1 e Data2), os fios da unidade de controle (O branch representa o PCSrc), o valor de escrita selecionado pelo último MUX (writeDataRF) e coisas mais gerais, como o clock e o valor do PC.

Na figura 4 é possível observar as 32 posições da memória de dados para o mesmo conjunto de instruções.

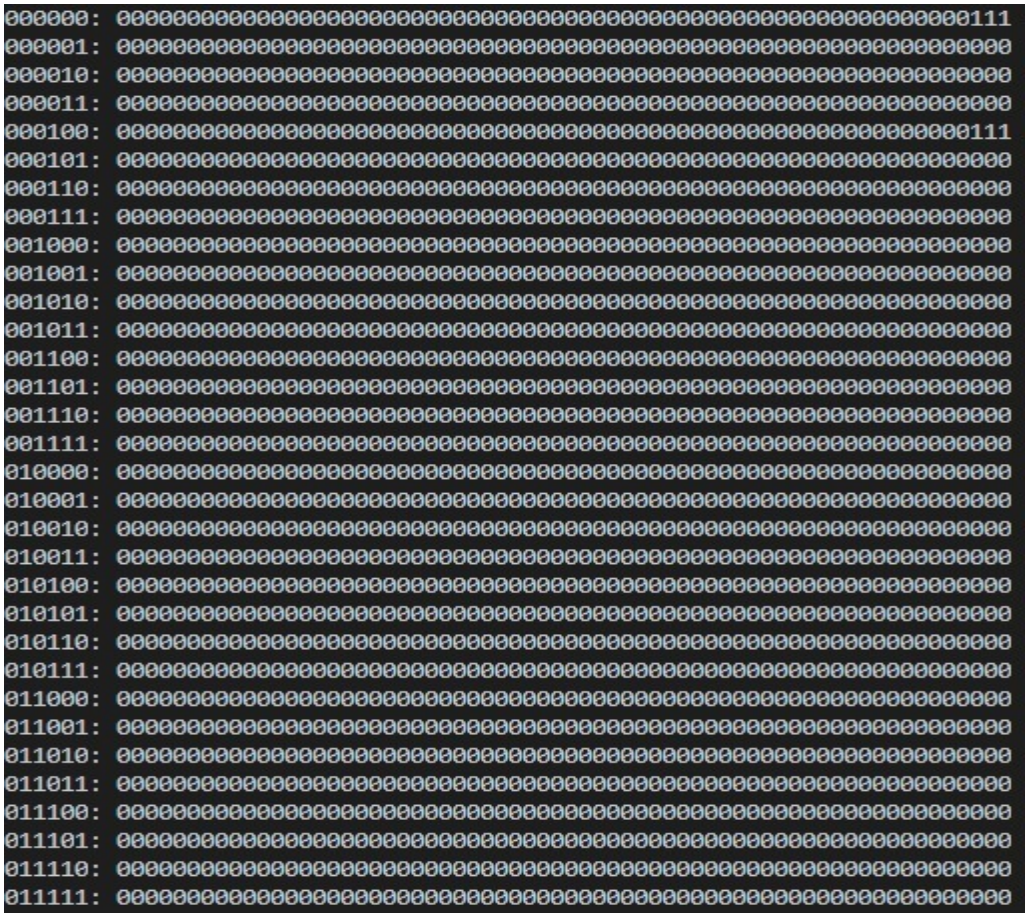

Figura 4. Conteúdo da Data Memory.

Outra coisa importante de se falar é que apesar do Branch seguir o enderçamento do PC padrão de byte em byte, a memória de instruções foi representanda sendo sequencial e indo de 1 em 1, por questões de simplicidade. Assim o Branch com o valor 12, que é o valor real da instrução de teste, moveu o PC três instruções a frente, como é possível ver ocorrendo no diagrama de ondas.

Para que seja possível realizar testes com diferentes valores nos registradores, é necessário definir tais valores antes da execução do código. A fim de efetuar esse ajuste, deve-se abrir o arquivo PreparingToExecute.v e ir até o módulo *RegisterFile*. Nele há um bloco de código o qual é utilizado para inicializar os registradores. A figura a seguir mostra esse trecho e também mostra um exemplo de como realizar a definição de valores.

```
module RegisterFile (Read1, Read2, WriteReg, WriteData, RegWrite, Data1, Data2, MemtoReg, reset, clock);
    input [4:0] Read1, Read2, WriteReg;
    input [63:0] WriteData;
    input RegWrite, MemtoReg, reset, clock;
    output [63:0] Data1, Data2;
    reg [63:0] RF [31:0];
    integer i;
    // Para definir valores para regs para teste, insira no always abaixo -> exemplo: RF[posicao] = valor
    always @ (reset) begin
        // O registrador referente a posicao 0 ira inicializar com o valor 2
        RF[0] = 64'b10;
    end

    assign Data1 = RF[Read1];
    assign Data2 = RF[Read2];
    always  @(posedge clock) begin
        if (RegWrite) begin
            RF[WriteReg] <= WriteData;
        end
    end
endmodule
```

Figura 5. Exemplo de inicialização do banco de registradores.

4. Considerações finais

A partir da produção deste trabalho foi possível realizar uma maior absorção e aprofundamento do conteúdo visto em aula. Conceitos como o caminho de dados, o funcionamento em single clock, o princípio de design sobre regularidade foram fundamentais no processo de implementação e, por meio da prática, houve uma melhor fixação. Ademais, outro importante aprendizado está relacionado ao uso de linguagem de descrição de hardware, Verilog; pois foi necessário muita pesquisa para realizar o seu uso nessa tarefa. Portanto, sendo uma experiência enriquecedora. Além disso, apesar de ser uma implementação simplificada do caminho de dados do RISC-V, sem pipeline e com um número de instruções reduzido, o trabalho não foi menos desafiador. Logo, tal fato reforça a complexidade do tema e a importância da dedicação no processo de estudo.

5. Referências

[1] David A. Patterson, John L. Hennessy. Computer Organization and Design RISC-V Edition: The Hardware Software Interface. Morgan Kaufmann. 1ª edição, 2017.

[2] Initialize Memory in Verilog. Project F - FPGA Development, 2020. Disponível em: <https://projectf.io/posts/initialize-memory-in-verilog/>. Acesso em: 13 de nov. 2020.