



CCF 252 - Organização de Computadores I  
Trabalho Prático 03 - Caminho de Dados RISC-V

Mateus Aparecido - 3858, Artur Papa - 3886, Luciano Belo - 3897

23 de novembro de 2020

# 1 Introdução

O trabalho em questão tem como objetivo abordar os conteúdos trabalhados na disciplina de Organização de Computadores 1, em específico, como visto no módulo 3, a utilização do Caminho de dados na arquitetura Single-Cycle RISC-V. O caminho de dados - Datapath [2] - é um conjunto de unidades funcionais que realizam operações de processamento de dados.

O caminhos de dados, juntamente com a unidade de controle, constituem a CPU (unidade de processamento central) de um sistema de computador. Trata-se de uma arquitetura load/store, ou seja, a leitura e a escrita de operandos armazenados na memória são feitos através de duas instruções dedicadas: ld e sd. As unidades funcionais (ALU, MUX, Instruction Memory, entre outros) não acessam diretamente a memória, mas, registradores internos que armazenam temporariamente seus operandos.

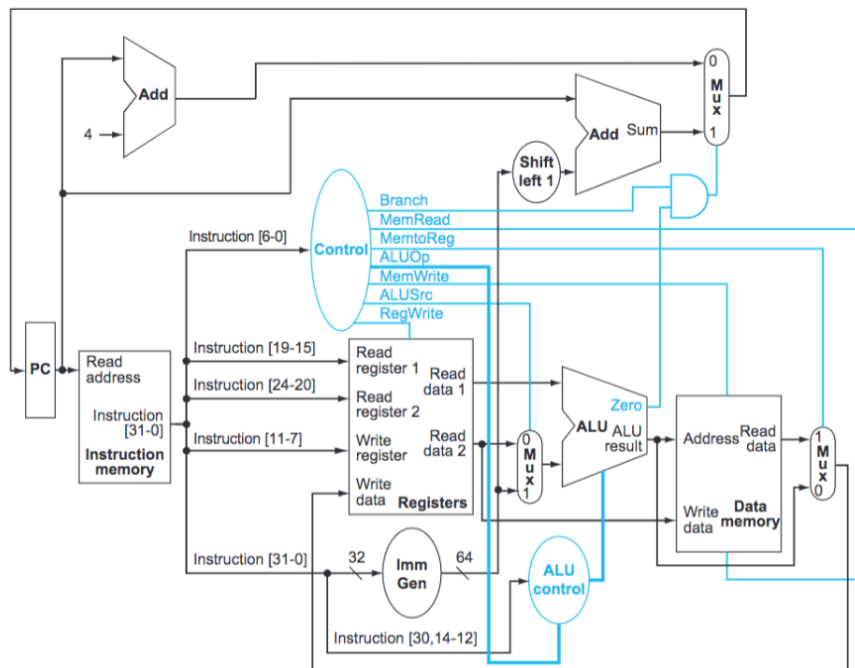


Figura 1: Caminho de dados

## 2 Caminho de dados

Essa seção irá abordar as unidades funcionais implementadas durante o desenvolvimento do projeto, além de explicá-las previamente.

### 2.1 PC

Contém o endereço da memória cujo conteúdo deve ser interpretado como a próxima instrução. O contador de programa é automaticamente incrementado para cada ciclo de instrução de forma que as instruções são normalmente executadas sequencialmente a partir da memória, sendo que o contador de programa deve ser colocado a zero no início da execução do mesmo. No Datapath Risc-V, o pc vai gerar o endereço de leitura para a memória de instruções.

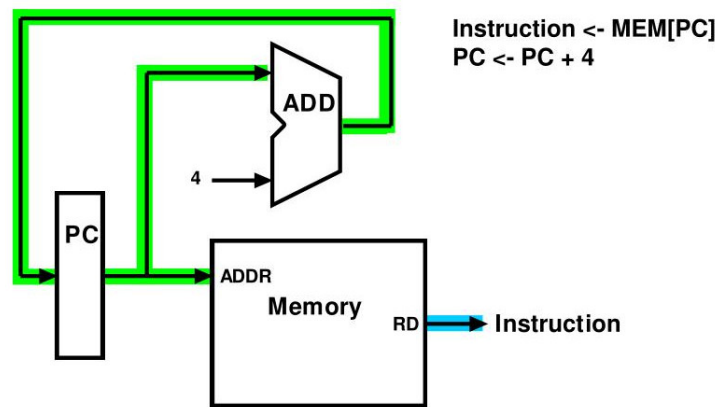


Figura 2: Program Counter

```
module program_counter(clock, reset, data_in, data_out);
    input clock;
    input reset;
    input [31:0] data_in;
    output reg [31:0] data_out;

    always @ (posedge clock or posedge reset) begin
        if (reset) begin
            data_out ≤ 0;
        end
        else begin
            data_out ≤ data_in;
        end
    end
end
endmodule
```

Figura 3: Módulo do PC

## 2.2 Memória de instrução (Instruction Memory)

A memória de instrução [3] é usada para armazenar e fornecer instruções dado um endereço. Ela pega uma entrada de endereço de instrução de 32 bits, lê os dados desse endereço para a saída de dados de leitura, já que no caminho de dados não escrevemos instruções. Uma vez que a memória de instrução é apenas para leitura, podemos tratá-la como lógica combinacional, embora seja um elemento de lógica sequencial.

No módulo desenvolvido, foi utilizado o parâmetro 'WIDTH' para definir a quantidade de instruções a serem lidas, além disso, é no arquivo 'instrucoes.bin' que estarão contidas as instruções em binário para a execução dos testes.

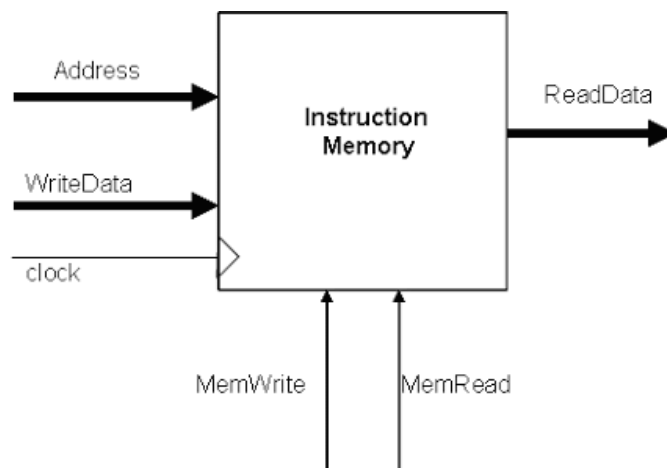


Figura 4: Memória de instrução

```
module InstructionMemory(endereco, instrucaoOut);  
  
    input [31:0] endereco;  
  
    output [31:0] instrucaoOut;  
  
    parameter WIDTH = 4; //Mudar a variável de acordo com o tamanho da memória  
  
    reg [31:0] RAM [0:WIDTH-1];  
  
    initial  
    begin  
        $readmemb("instrucoes.bin", (RAM));  
    end  
  
    assign instrucaoOut = RAM[endereco/4];  
  
endmodule
```

Figura 5: Módulo Instruction Memory

## 2.3 Unidade de controle (control)

No caminho de dados, cada etapa é modificada por sinais de controle que configuram as direções de fluxo de dados em barramentos de comunicação e selecionam a ALU e funções da memória. Os sinais de controle[1] são gerados por uma unidade de controle que consiste em uma ou mais máquinas de estados finitos.

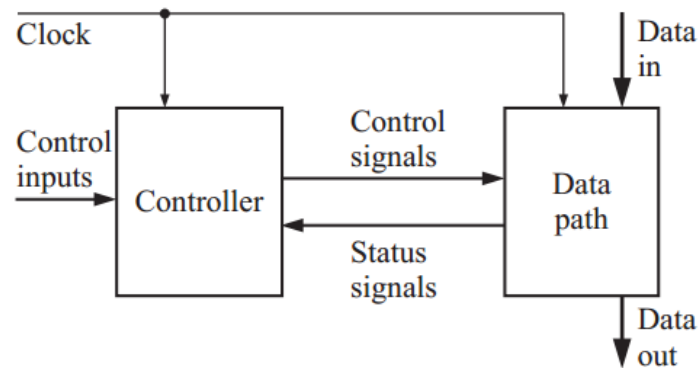


Figura 7: Unidade de controle

```
parameter R_Format = 7'b0110011;
parameter LD = 7'b0000011;
parameter SD = 7'b0100011;
parameter BEQ = 7'b1100011;

always @(*)
begin
    case (opcode)
        R_Format:
            begin
                Branch = 1'b0;
                MemRead = 1'b0;
                MemtoReg = 1'b0;
                MemWrite = 1'b0;
                alusrc = 1'b0;
                regwrite = 1'b1;
                aluop = 2'b10;
                RegDst = 1'b1;
            end
    end
end
```

Figura 8: Parte do módulo control

## 2.4 Registradores (Register File)

O Register File é uma matriz de registros de processador em uma unidade de processamento central (CPU). É um elemento de estado que consiste de um conjunto de registradores que pode ser lido e escrito fornecendo um número de registro a ser acessado, faz parte da arquitetura e é visível para o programador, ao contrário do conceito de caches transparentes.

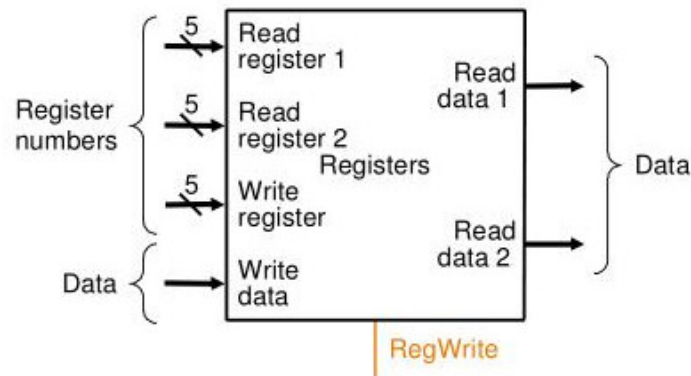


Figura 9: Banco de registradores

```
integer i;

always @ (posedge reset or posedge clock) begin

    if (reset) begin
        for (i=0; i<32; i=i+1) array[i] ≤ 64'b0;
    end

    else if (writereg) begin
        array [rd] = writedata;
    end

    readdata1 = array[rs1];
    readdata2 = array[rs2];
end
```

Figura 10: Parte do módulo regfile

## 2.5 Extensor de sinal (signal extension)

O Extensor de sinal é um componente usado para aumentar o tamanho dos dados. O bit mais significativo é replicado de forma a manter o sinal do dado original.

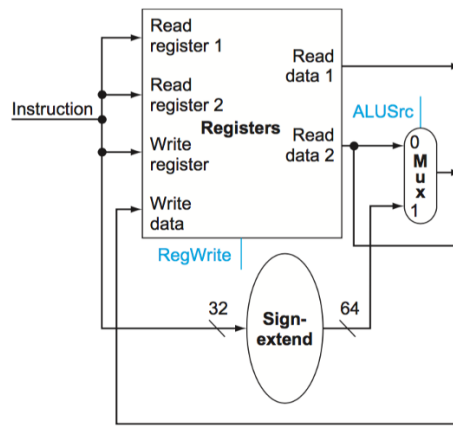


Figura 11: Extensor de sinal

```

module extensao_sinal(
    in,
    extensor
);
    input [31:0] in;
    output[31:0] extensor;
    reg[31:0] IMM_OUT;
    wire[6:0] opcode;
    wire[2:0] funct3;

    assign extensor = IMM_OUT;
    assign opcode = in[6:0];
    assign funct3 = in[14:12];
    always @(in)
    case(opcode)
        7'b0100011: IMM_OUT ≤ { {21{in[31]}}, in[30:25], in[11:8], in[7]}; // SD
        7'b0000011: IMM_OUT ≤ { {21{in[31]}}, in[30:25], in[24:21], in[20]}; // LD
        7'b1100011: IMM_OUT ≤ { {20{in[31]}}, in[7], in[30:25], in[11:8], {1{1'b0}}};
        default: IMM_OUT ≤ 32'bx;
    endcase
endmodule

```

Figura 12: Módulo do extensor de sinal

## 2.6 Deslocamento para Esquerda (Shift Left)

O Deslocamento para Esquerda é um roteamento dos sinais entre a entrada e a saída que adiciona dois zeros à extremidade inferior do campo de deslocamento estendido de sinal. Nenhum hardware de deslocamento real é necessário, uma vez que a quantidade de 'deslocamento' é constante. Como sabemos que o deslocamento foi estendido de sinal de 12 bits, serão descartados apenas os bits de sinal.

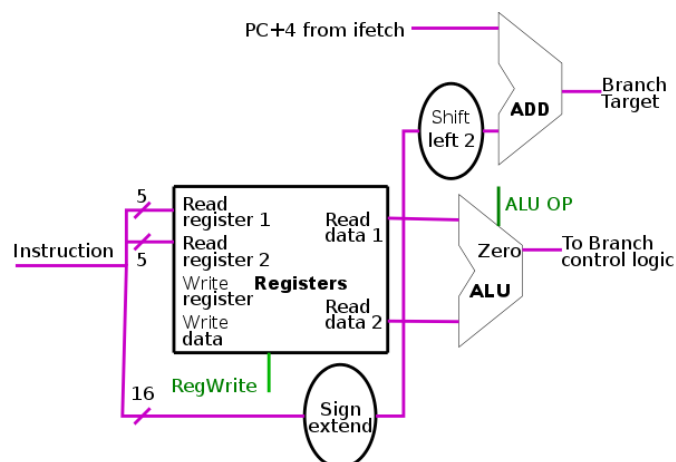


Figura 13: Deslocamento pra esquerda integrado a outras partes do datapath

```
module ShiftLeft2(ValSignExtend, Result);
    input wire [31:0]ValSignExtend;
    output reg [31:0]Result;
    always @ (*) begin
        Result ≤ (ValSignExtend << 2);
    end
endmodule
```

Figura 14: Módulo do shiftright

## 2.7 MUX

Multiplexadores [5] são também conhecidos como seletores de dados. Este tipo de circuito lógico trabalha com vários dados de entrada, efetuando uma seleção em um determinado instante. Em outras palavras, ele atua como uma chave digital de várias posições, onde de acordo com as entradas de seleção, irá controlar qual será o chaveamento para a saída.

No caminho de dados do RISC-V[4] os multiplexadores são usados para definir qual será a entrada correta para as várias fontes nas diferentes instruções, assim como é ilustrado no diagrama a seguir:

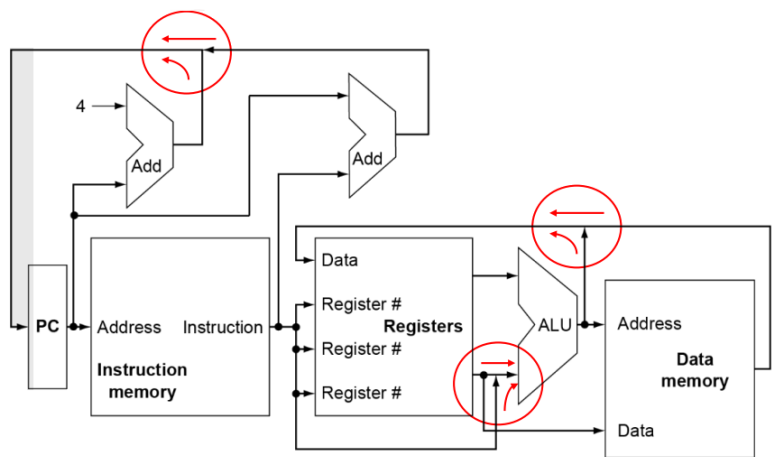


Figura 15: Lugares aonde é necessário ter um mux

```
module mux1 (data0,data1,select,out);
    input wire [4:0] data0, data1;
    input wire select;
    output reg out;
    always @(data0, data1, select) begin
        case (select)
            0: out ≤ data0;
            1: out ≤ data1;
        endcase
    end
endmodule
```

Figura 16: Módulo do mux



## 2.8 ALU

A ALU [6] (em português ULA - Unidade Lógica Aritmética) é a unidade central do caminho de dados . A ALU é o conjunto de circuitos de computação responsável pela execução das instruções aritméticas (adição, subtração, comparação, multiplicação e divisão), de lógica booleana (OR, XOR, AND) e manipulação de bits (deslocamentos). A ALU executa as instruções RISC-V, sendo todos os seus operandos vindos do banco de registradores ou de valores imediatos, constantes passadas pelo compilador (programador) por meio de instruções específicas.

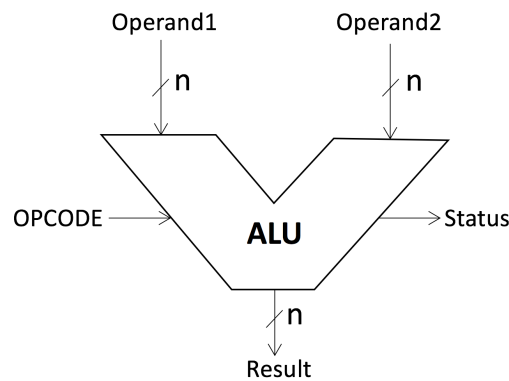


Figura 17: ULA(ALU)

```
module alu(alu_control,a,b,alu_out,zero);
    input [3:0] alu_control;
    input [31:0] a,b;
    output reg [31:0] alu_out;
    output zero;

    assign zero = (alu_out == 0);

    always @(alu_control, a, b)
    begin
        case (alu_control)
            0: alu_out ≤ a & b; // AND
            1: alu_out ≤ a | b; // OR
            2: alu_out ≤ a + b; // ADD
            6: alu_out ≤ a - b; // SUB
            7: alu_out ≤ a < b ? 1 : 0;
            12: alu_out ≤ ~( a | b );
            default: alu_out ≤ 0;
        endcase
    end
endmodule
```

Figura 18: Módulo da ALU

## 2.9 Alu Control

A Alu Control especifica qual instrução a ALU irá executar baseada no tipo da operação e em seus códigos de controle.

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

Figura 19: Códigos de controle da ALU, com as linhas de controle e o ALUOp

```

module alu_control(funct7,funct3,alu_operation,alu_ctr);
    input [6:0]funct7; // 7bits
    input [2:0]funct3; // 3bits
    input [1:0]alu_operation; //2bits
    output reg [3:0]alu_ctr; //4bits

    always @*begin
        if(
            alu_operation == 2'b00 &&
            funct7 == 7'bxxxxxxx &&
            funct3 == 3'b011
        ) begin //alu_ctr => ld
            alu_ctr ≤ 4'b0010;
        end
    end

```

Figura 20: Módulo da ALU control

## 2.10 Memória de dados (Data Memory)

A memória de dados deve ser escrita em instruções de armazenamento, por isso a memória de dados tem controle de leitura e gravação, ela possui uma entrada com o endereço da instrução e uma com os dados de gravação, e uma única saída para leitura do resultado.

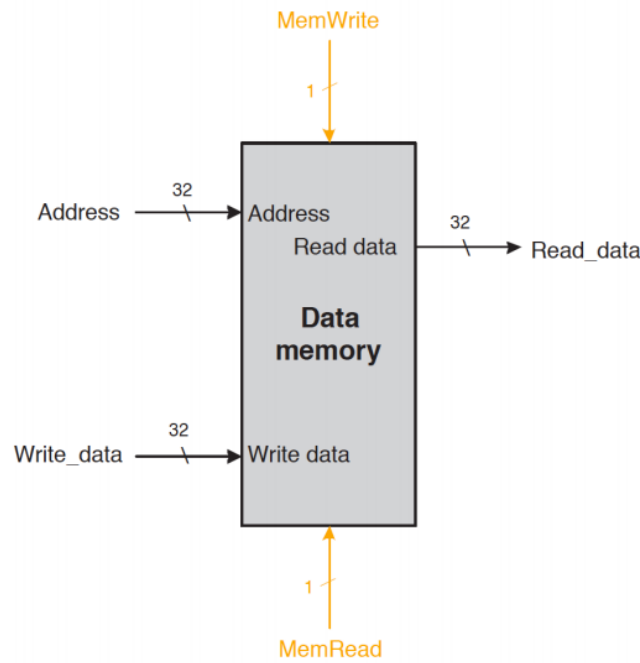


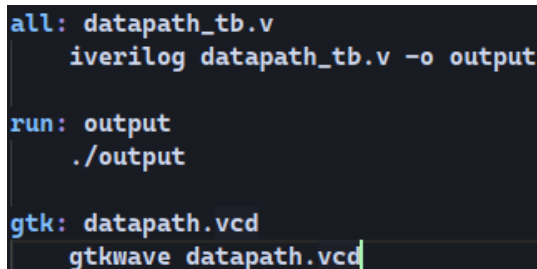
Figura 21: Data memory

```
module data_memory (  
    mem_write,  
    mem_read,  
    address,  
    write_data,  
    result,  
    reset,  
    clock  
);  
  
    input wire mem_write, mem_read, reset, clock;  
    input wire [31:0] address;  
    input wire [31:0] write_data;  
    output reg [31:0] result;  
    reg [31:0] dataMemory [31:0];  
  
    always @ (posedge clock) begin // write the reg  
        if (reset) begin  
            dataMemory[0] ≤ 32'd0;  
        end  
    end  
endmodule
```

Figura 22: Módulo do DataMemory

## 2.11 Execução do Trabalho

Para a execução do caminho de dados o grupo optou por adicionar um arquivo makefile contendo as seguintes instruções:



```
all: datapath_tb.v
    iverilog datapath_tb.v -o output

run: output
    ./output

gtk: datapath.vcd
    gtkwave datapath.vcd
```

Figura 23: Makefile

Assim, para compilar o caminho de dados basta executar no terminal o comando ‘make’, para executar usamos o comando ‘make run’ e para gerar o formato de ondas no software GTKWave executamos o comando ‘make gtk’.

Obs: Todos os códigos do trabalho estarão no diretório ‘src’ assim como já explicado no README do repositório no Github.

## 3 Conclusão

Desta forma, podemos concluir que o trabalho em questão foi desenvolvido conforme o esperado, atingindo todas as especificações requeridas na descrição do mesmo, já que a função principal do caminho de dados foi feita, a implementação de elementos que processam dados e endereços na CPU.

Por conseguinte podemos ressaltar que o livro-texto da disciplina, [7] foi de suma importância para o desenvolvimento do projeto. Módulos como ALU, ALU Control e Instruction Memory foram embasados nos conteúdos elucidados no material feito por Patterson e Hennessy, o que foi de grande ajuda para o grupo na escolha das imagens e também na construção do código em verilog.

Em adição, é válido dizer que apesar das dificuldades na implementação do código, o grupo foi capaz de superar e corrigir quaisquer erros no desenvolvimento do algoritmo. Por fim, verificou-se a assertiva para o objetivo do projeto em implementar uma versão simplificada do caminho de dados do RISC-V.

## Referências

- [1] control:[https://www.eng.auburn.edu/~nelsovp/courses/elec5200\\_6200/ELEC5200\\_6200\\_datapath\\_control.pdf](https://www.eng.auburn.edu/~nelsovp/courses/elec5200_6200/ELEC5200_6200_datapath_control.pdf).
- [2] datapath:<https://www.computerhope.com/jargon/d/datapath.htm>.
- [3] Instruction:<http://web-ext.u-aizu.ac.jp/~yliu/teaching/comparch/lecture7.html>.
- [4] multiplexers:[http://algo.ing.unimo.it/people/andrea/Didattica/Architettura/SlidesPDF/Chapter\\_04-RISC-V.pdf](http://algo.ing.unimo.it/people/andrea/Didattica/Architettura/SlidesPDF/Chapter_04-RISC-V.pdf).
- [5] Mux:<https://eletronworld.com.br/eletronica/multiplexadores/#:~:text=Multiplexadores%20s%C3%A3o%20tamb%C3%A9m%20conhecidos%20como,sele%C3%A7%C3%A3o%20em%20um%20determinado%20instante>.
- [6] ula:<https://repositorio.ufpe.br/bitstream/123456789/26036/1/DISSERTA%C3%87%C3%83O%20Cecil%20Accetti%20Resende%20de%20Ata%C3%ADde%20Melo.pdf>.
- [7] David A. Patterson John L. Hennessy. *Computer Organization and Design*. 2017.