

Caminho de Dados do RISC-V

Grupo 1 - Alan Araújo dos Reis (5096), Gabriel Rodrigues Marques (5097)

Instituto de Ciências Exatas e Tecnológicas

Universidade Federal de Viçosa - Campus Florestal (UFV-CAF)

Florestal – MG – Brasil

{alan.a.reis, gabriel.r.marques}@ufv.br

Resumo. Esta documentação apresenta o desenvolvimento e implementação do caminho de dados simplificado do RISC-V, capaz de executar um subconjunto de instruções.

Repositório: <https://github.com/gabridulol/RISC-V-Datapath>

1. Introdução

O trabalho consiste no desenvolvimento de uma versão simplificada do caminho de dados do RISC-V. O caminho de dados deve executar um subconjunto de instruções e exibir os resultados, simulando o funcionamento de um processador. A linguagem de descrição de hardware Verilog foi utilizada para implementação. O caminho de dados segue o que foi apresentado no livro *Computer Organization and Design RISC-V Edition: The Hardware Software Interface* (Figura 1).

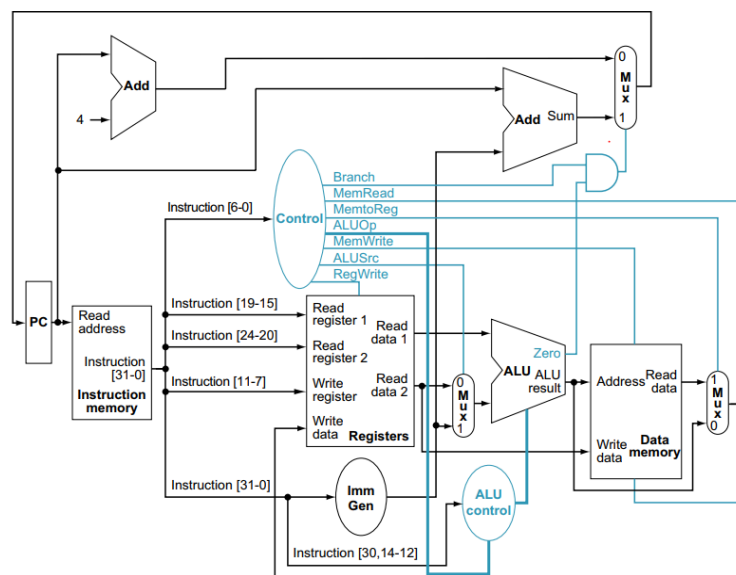


Figura 1. Caminho de Dados do RISC-V

2. Desenvolvimento

O caminho de dados implementado por cada grupo deveria suportar um subconjunto de instruções. É importante ressaltar que o caminho de dados foi desenvolvido e implementado especificamente para este subconjunto de instruções (Figura 2). Não há suporte para outras instruções fora desse subconjunto.

lb	sb	add	and	ori	sll	bne
----	----	-----	-----	-----	-----	-----

Figura 2. Instruções do Grupo 1

lb: leitura de um valor da memória de dados para o registrador.

sb: escrita de um valor do registrador para memória de dados.

add: soma o valor de dois registradores e armazena o resultado em um terceiro registrador.

and: realiza uma operação AND bit a bit entre dois registradores e armazena o resultado em um terceiro registrador.

ori: realiza uma operação OR bit a bit entre um registrador e um valor imediato (constante), e armazena o resultado em um segundo registrador.

sll: desloca os bits de um registrador para a esquerda por um número especificado de posições em um registrador, e armazena o resultado em um terceiro registrador.

bne: salta para um endereço especificado se os valores de dois registradores não forem iguais.

O desenvolvimento consistiu em modularizar cada “caixinha” do caminho de dados e descrever seu funcionamento. Os módulos e seu funcionamento estão resumidos brevemente logo abaixo.

Add: o módulo Add realiza a soma de dois valores de entrada, produzindo o resultado como saída.

ALU: o módulo ALU realiza operações lógicas aritméticas entre dois valores de entrada, produzindo o resultado da operação como saída. A operação realizada é determinada por um sinal de controle gerado pelo ALUControl.

ALUControl: o módulo ALUControl gera como resultado de saída o sinal de controle que determina a operação lógica aritmética realizada pela ALU. O sinal de controle gerado é determinado pelo sinal de controle ALUOp de Control e pela funct3 da instrução.

Control: o módulo Control é responsável por gerar os sinais de controle necessários para orientar o caminho de dados: Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc e RegWrite. Os sinais de controle gerados são determinados pelo opcode da instrução.

DataMemory: o módulo DataMemory implementa a memória de dados, responsável por armazenar os dados na memória durante a execução das instruções.

ImmGen: o módulo ImmGen gera o valor do imediato a partir de uma instrução.

InstructionMemory: o módulo InstructionMemory implementa a memória de instruções, responsável por armazenar as instruções que serão executadas.

Mux: o módulo Mux seleciona um dos valores de entrada, determinado por um sinal de controle.

PC: o módulo PC é responsável por manter o contador de programa, que contém o endereço de memória da próxima instrução a ser executada. A cada mudança de clock o endereço é atualizado, caso o sinal de reset esteja ativo o contador de programa é reiniciado.

Registers: o módulo Registers implementa o banco de registradores, responsável por armazenar os dados dos registradores durante a execução das instruções.

Datapath: o módulo Datapath realiza a conexão entre todos os módulos implementados.

```
1 module Datapath (
2     input wire clk, reset
3 );
4
5     wire [31:0] PC;
6     wire [31:0] instruction;
7     wire [31:0] immediate;
8     wire [31:0] readData1, readData2, readData3;
9     wire [31:0] muxout0, muxout1, muxout2;
10    wire [31:0] addout0, addout1;
11    wire [31:0] ALUResult;
12    wire [3:0] ALUControl;
13    wire [2:0] ALUOp;
14    wire Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite;
15    wire zero;
16
17    Add add0(PC, 4, addout0);
18    Add add1(PC, immediate, addout1);
19    ALU alu(readData1, muxout0, ALUControl, ALUResult, zero);
20    ALUControl alucontrol(instruction[14:12], ALUOp, ALUControl);
21    Control control(instruction[6:0], Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite);
22    DataMemory datamemory(MemWrite, MemRead, ALUResult, readData2, readData3);
23    ImmGen immgen(instruction, immediate);
24    InstructionMemory instructionmemory(PC, instruction);
25    Mux mux0(ALUSrc, readData2, immediate, muxout0);
26    Mux mux1(Branch & zero, addout0, addout1, muxout1);
27    Mux mux2(MemtoReg, ALUResult, readData3, muxout2);
28    PC pc(clk, reset, muxout1, PC);
29    Registers registers(RegWrite, instruction[19:15], instruction[24:20], instruction[11:7], muxout2, readData1, readData2);
30
31 endmodule
```

Figura 2. Módulo Datapath

Datapath_Testbench: o módulo Datapath_Testbench executa a simulação do caminho de dados e exibe os resultados. A memória de dados, memória de instruções e registradores são inicializados com os arquivos de entrada. O estado inicial e o estado atualizado a cada execução de uma nova instrução são exibidos.

```
1  module Datapath_Testbench;
2      reg clk, reset;
3
4      Datapath datapath(clk, reset);
5
6      initial begin
7          $readmemb("Verilog/Input/DataMemory.mem", datapath.datamemory.memory);
8          $readmemb("Verilog/Input/InstructionMemory.mem", datapath.instructionmemory.memory);
9          $readmemb("Verilog/Input/Registers.mem", datapath.registers.registers);
10
11         for (integer i = 0; i < 32; i = i + 1) begin
12             $display("DataMemory [-] = %d", i, datapath.datamemory.memory[i]);
13         end
14         $display();
15         for (integer i = 0; i < 32; i = i + 1) begin
16             $display("InstructionMemory [-] = %h", i, datapath.instructionmemory.memory[i]);
17         end
18         $display();
19         for (integer i = 0; i < 32; i = i + 1) begin
20             $display("Register [-] = %d", i, datapath.registers.registers[i]);
21         end
22         $display();
23         $display("Program Counter = -", datapath.pc.PCOut);
24         $display("Instruction = %h", datapath.instructionmemory.instruction);
25         $display();
26     end
27
28     always @(datapath.instructionmemory.instruction) begin
29         for (integer i = 0; i < 32; i = i + 1) begin
30             $display("DataMemory [-] = %d", i, datapath.datamemory.memory[i]);
31         end
32         $display();
33         for (integer i = 0; i < 32; i = i + 1) begin
34             $display("Register [-] = %d", i, datapath.registers.registers[i]);
35         end
36         $display();
37         $display("Program Counter = -", datapath.pc.PCOut);
38         $display("Instruction = %h", datapath.instructionmemory.instruction);
39         $display();
40     end
41
42     initial begin
43         clk = 0;
44         reset = 1;
45         #2 reset = 0;
46     end
47
48     always #1 clk = ~clk;
49
50 endmodule
```

Figura 3. Módulo Datapath_Testbench

Mais detalhes de implementação dos módulos podem ser vistos no código fonte do projeto, disponível no repositório.

3. Resultados


O código em RISC-V Assembly (Figura 4) foi utilizado para realizar os testes no caminho de dados. O montador desenvolvido no trabalho anterior foi utilizado para converter as instruções para a linguagem de máquina binária correspondente.



```
1  bne x0, x0, 0
2  lb x1, 0(x0)
3  add x2, x1, x1
4  and x3, x1, x1
5  ori x4, x1, 32
6  sll x5, x1, x1
7  add x6, x3, x2
8  add x7, x5, x4
9  add x8, x7, x6
10 lb x8, 1(x0)
```

Figura 4. Código RISC-V Assembly

O caminho de dados pode ser executado via terminal com comando *make*. Ao executar é exibido o estado inicial e o estado atualizado a cada execução de uma nova instrução, além do contador de programa e instrução executada. Os resultados podem ser encontrados nas figuras abaixo (Figura 5, 6, 7, 8, 9, 10).



1	DataMemory [0] =	4
2	DataMemory [1] =	0
3	DataMemory [2] =	0
4	DataMemory [3] =	0
5	DataMemory [4] =	0
6	DataMemory [5] =	0
7	DataMemory [6] =	0
8	DataMemory [7] =	0
9	DataMemory [8] =	0
10	DataMemory [9] =	0
11	DataMemory [10] =	0
12	DataMemory [11] =	0
13	DataMemory [12] =	0
14	DataMemory [13] =	0
15	DataMemory [14] =	0
16	DataMemory [15] =	0
17	DataMemory [16] =	0
18	DataMemory [17] =	0
19	DataMemory [18] =	0
20	DataMemory [19] =	0
21	DataMemory [20] =	0
22	DataMemory [21] =	0
23	DataMemory [22] =	0
24	DataMemory [23] =	0
25	DataMemory [24] =	0
26	DataMemory [25] =	0
27	DataMemory [26] =	0
28	DataMemory [27] =	0
29	DataMemory [28] =	0
30	DataMemory [29] =	0
31	DataMemory [30] =	0
32	DataMemory [31] =	0

Figura 5. Memória de Dados (INICIAL)




```
1  InstructionMemory [ 0] = 00001067
2  InstructionMemory [ 1] = 00000083
3  InstructionMemory [ 2] = 00108133
4  InstructionMemory [ 3] = 0010f1b3
5  InstructionMemory [ 4] = 0200e213
6  InstructionMemory [ 5] = 001092b3
7  InstructionMemory [ 6] = 00218333
8  InstructionMemory [ 7] = 004283b3
9  InstructionMemory [ 8] = 00638433
10 InstructionMemory [ 9] = 008000a3
```

Figura 6. Memória de Instruções



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

Figura 7. Registradores (INICIAL)



1	DataMemory [0] =	4
2	DataMemory [1] =	112
3	DataMemory [2] =	0
4	DataMemory [3] =	0
5	DataMemory [4] =	0
6	DataMemory [5] =	0
7	DataMemory [6] =	0
8	DataMemory [7] =	0
9	DataMemory [8] =	0
10	DataMemory [9] =	0
11	DataMemory [10] =	0
12	DataMemory [11] =	0
13	DataMemory [12] =	0
14	DataMemory [13] =	0
15	DataMemory [14] =	0
16	DataMemory [15] =	0
17	DataMemory [16] =	0
18	DataMemory [17] =	0
19	DataMemory [18] =	0
20	DataMemory [19] =	0
21	DataMemory [20] =	0
22	DataMemory [21] =	0
23	DataMemory [22] =	0
24	DataMemory [23] =	0
25	DataMemory [24] =	0
26	DataMemory [25] =	0
27	DataMemory [26] =	0
28	DataMemory [27] =	0
29	DataMemory [28] =	0
30	DataMemory [29] =	0
31	DataMemory [30] =	0
32	DataMemory [31] =	0

Figura 8. Memória de Dados (FINAL)



```
1 Register [ 0] =      0
2 Register [ 1] =      4
3 Register [ 2] =      8
4 Register [ 3] =      4
5 Register [ 4] =     36
6 Register [ 5] =     64
7 Register [ 6] =     12
8 Register [ 7] =    100
9 Register [ 8] =    112
10 Register [ 9] =      0
11 Register [10] =      0
12 Register [11] =      0
13 Register [12] =      0
14 Register [13] =      0
15 Register [14] =      0
16 Register [15] =      0
17 Register [16] =      0
18 Register [17] =      0
19 Register [18] =      0
20 Register [19] =      0
21 Register [20] =      0
22 Register [21] =      0
23 Register [22] =      0
24 Register [23] =      0
25 Register [24] =      0
26 Register [25] =      0
27 Register [26] =      0
28 Register [27] =      0
29 Register [28] =      0
30 Register [29] =      0
31 Register [30] =      0
32 Register [31] =      0
```

Figura 9. Registradores (FINAL)

Init Value	Register	Decimal	Hex	Binary
0	x0 (zero)	0	0x00000000	0b00000000000000000000000000000000
<input type="text" value="0"/>	x1 (ra)	4	0x00000004	0b00000000000000000000000000000100
<input type="text" value="0"/>	x2 (sp)	8	0x00000008	0b00000000000000000000000000001000
<input type="text" value="0"/>	x3 (gp)	4	0x00000004	0b00000000000000000000000000000100
<input type="text" value="0"/>	x4 (tp)	36	0x00000024	0b000000000000000000000000000100100
<input type="text" value="0"/>	x5 (t0)	64	0x00000040	0b0000000000000000000000000001000000
<input type="text" value="0"/>	x6 (t1)	12	0x0000000c	0b000000000000000000000000000001100
<input type="text" value="0"/>	x7 (t2)	100	0x00000064	0b0000000000000000000000000001100100
<input type="text" value="0"/>	x8 (s0/fp)	112	0x00000070	0b0000000000000000000000000001110000
<input type="text" value="0"/>	x9 (s1)	0	0x00000000	0b00000000000000000000000000000000
<input type="text" value="0"/>	x10 (a0)	0	0x00000000	0b00000000000000000000000000000000
<input type="text" value="0"/>	x11 (a1)	0	0x00000000	0b00000000000000000000000000000000
<input type="text" value="0"/>	x12 (a2)	0	0x00000000	0b00000000000000000000000000000000
<input type="text" value="0"/>	x13 (a3)	0	0x00000000	0b00000000000000000000000000000000
<input type="text" value="0"/>	x14 (a4)	0	0x00000000	0b00000000000000000000000000000000
<input type="text" value="0"/>	x15 (a5)	0	0x00000000	0b00000000000000000000000000000000
<input type="text" value="0"/>	x16 (a6)	0	0x00000000	0b00000000000000000000000000000000
<input type="text" value="0"/>	x17 (a7)	0	0x00000000	0b00000000000000000000000000000000
<input type="text" value="0"/>	x18 (s2)	0	0x00000000	0b00000000000000000000000000000000
<input type="text" value="0"/>	x19 (s3)	0	0x00000000	0b00000000000000000000000000000000
<input type="text" value="0"/>	x20 (s4)	0	0x00000000	0b00000000000000000000000000000000
<input type="text" value="0"/>	x21 (s5)	0	0x00000000	0b00000000000000000000000000000000
<input type="text" value="0"/>	x22 (s6)	0	0x00000000	0b00000000000000000000000000000000
<input type="text" value="0"/>	x23 (s7)	0	0x00000000	0b00000000000000000000000000000000
<input type="text" value="0"/>	x24 (s8)	0	0x00000000	0b00000000000000000000000000000000
<input type="text" value="0"/>	x25 (s9)	0	0x00000000	0b00000000000000000000000000000000
<input type="text" value="0"/>	x26 (s10)	0	0x00000000	0b00000000000000000000000000000000
<input type="text" value="0"/>	x27 (s11)	0	0x00000000	0b00000000000000000000000000000000
<input type="text" value="0"/>	x28 (t3)	0	0x00000000	0b00000000000000000000000000000000
<input type="text" value="0"/>	x29 (t4)	0	0x00000000	0b00000000000000000000000000000000
<input type="text" value="0"/>	x30 (t5)	0	0x00000000	0b00000000000000000000000000000000
<input type="text" value="0"/>	x31 (t6)	0	0x00000000	0b00000000000000000000000000000000

Figura 10. RISC-V Interpreter

4. Conclusão

O desenvolvimento do caminho de dados simplificado do RISC-V proporcionou uma experiência prática enriquecedora e um entendimento aprofundado do funcionamento e das nuances de sua implementação. Foi possível perceber a importância da modularização e da correta integração entre os diferentes componentes do caminho de dados. O caminho de dados demonstrou ser capaz de executar de maneira correta o subconjunto de instruções definidas, atendendo aos requisitos funcionais solicitados. O seu código-fonte está disponível para futuras melhorias e ampliações.

Referências

David A. Patterson and John L. Hennessy. Computer Organization and Design RISC-V Edition: The Hardware Software Interface.

RISC-V Interpreter. Disponível em:

<<https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/#>>

RISC-V Graphical Datapath Simulator. Disponível em:

<<https://jesse-r-s-hines.github.io/RISC-V-Graphical-Datapath-Simulator/>>