

Caminho de Dados em Arquitetura RISC-V

Álvaro Gomes - (5095)¹, Esdras Araújo - (5102)²

alvaro.g.neto@ufv.br, esdras.araujo@ufv.br

¹Universidade Federal de Viçosa (UFV)
Campus Florestal – MG – Brazil

Abstract. *The main objective of this work is to synthesize the knowledge related to the data path inside a processor in the RISC-V architecture. The data path is a fundamental part of a processor, as it is in this part where the logical and arithmetic operations are performed to process the instructions inside the computer. The data path passes through several sequential stages. At each stage the instructions are fetched, decoded, read and written in the registers and the control unit coordinates where the instructions should be sent and what operations and modules should be executed.*

Resumo. *O objetivo principal deste trabalho é sintetizar os conhecimentos relacionados ao caminho de dados dentro de um processador na arquitetura RISC-V. O caminho de dados é uma parte fundamental de um processador, pois é nessa parte onde são feitas as operações lógicas e aritméticas para processar as instruções dentro do computador. O caminho de dados passa por diversos estágios sequenciais. Em cada estágio as instruções são buscadas, decodificadas, lidas e escritas nos registradores e a unidade de controle coordenada para onde devem ser mandadas as instruções e quais operações e módulos devem ser executadas.*

1. Execução do Projeto

Para executar o projeto, siga as seguintes etapas:

1. Preencha o arquivo `instrucao.asm` com as instruções assembly que serão testadas. O montador não aceita rótulos, então instruções de desvio devem receber como argumento números imediatos.
2. Execute o comando `Python3 montador.py` para gerar as instruções no formato binário e salvá-las no arquivo `Instrucoes.bin`.
3. Caso haja algum erro no montador, as instruções podem ser escritas diretamente em binário no arquivo `instrucoes.bin`.
4. Utilize o arquivo `makefile` fornecido para facilitar a compilação e execução do projeto.
5. Para compilar e executar o projeto, utilize o comando `make`.
6. Caso deseje visualizar os resultados no GTKWave, utilize o comando `make gtk`.
7. Se o `makefile` não estiver instalado, você pode seguir a sequência de comandos a seguir:
 - Execute `iverilog -o testbench testBench.v` para compilar o código.
 - Execute `vvp testbench` para executar o projeto.

- Execute `gtkwave RiscV.vcd` para abrir o arquivo VCD no GTKWave e visualizar os resultados.
8. Os registradores e a memória são inicializados a partir de arquivos, nomeados de `Registradores.bin` e `DataMemory.bin`, respectivamente.

2. Estrutura do Projeto

2.1. Diagrama do Caminho de Dados

Para exemplificar melhor o caminho de dados implementado, o diagrama de dados abaixo fornece uma visualização mais clara do fluxo de dados dentro do processador. Com ele, é possível analisar os blocos funcionais, as unidades lógicas e aritméticas, pontos de controle, desvios e transformações. Além disso, por meio do diagrama, é possível ter a percepção de como os blocos estão relacionados e como os desvios e cálculos são feitos. Essa foi uma parte fundamental para a execução do projeto.

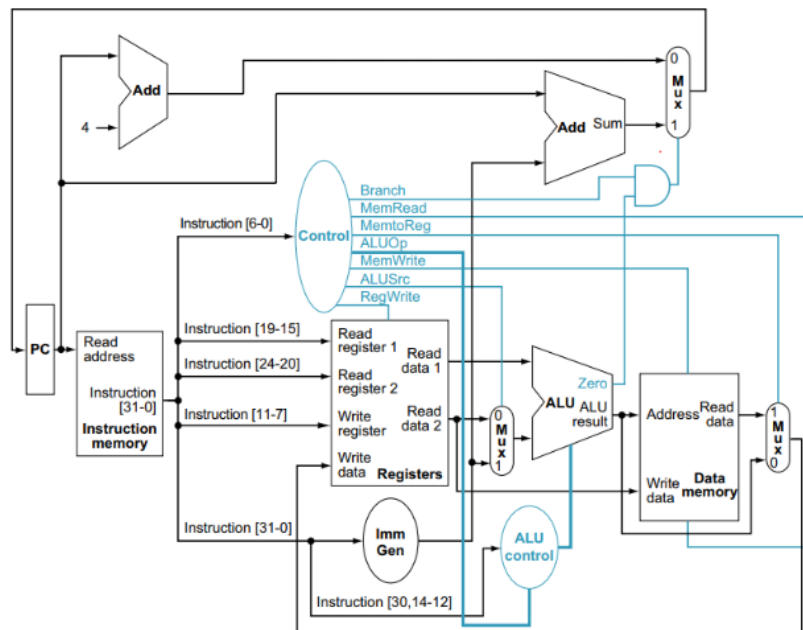



Figure 1. Diagrama do Caminho de Dados

2.2. RegisterFile

Os registradores são responsáveis pelo armazenamento temporário dos dados executados pelas instruções. Em nosso projeto, criamos uma estrutura de 32 registradores com 32 bits cada. Primeiramente, cada *wire* é atribuído a um valor dos registradores que foi lido de um arquivo "Registros.bin". Em todo momento, os valores armazenados nos registradores são passados para *ReadData1* e *ReadData2* conforme especificado pela instrução. Em cada borda de subida do clock, é verificado se o sinal *RegWrite* está ativo e, se estiver, o valor calculado durante o ciclo da instrução é escrito no registrador de destino.

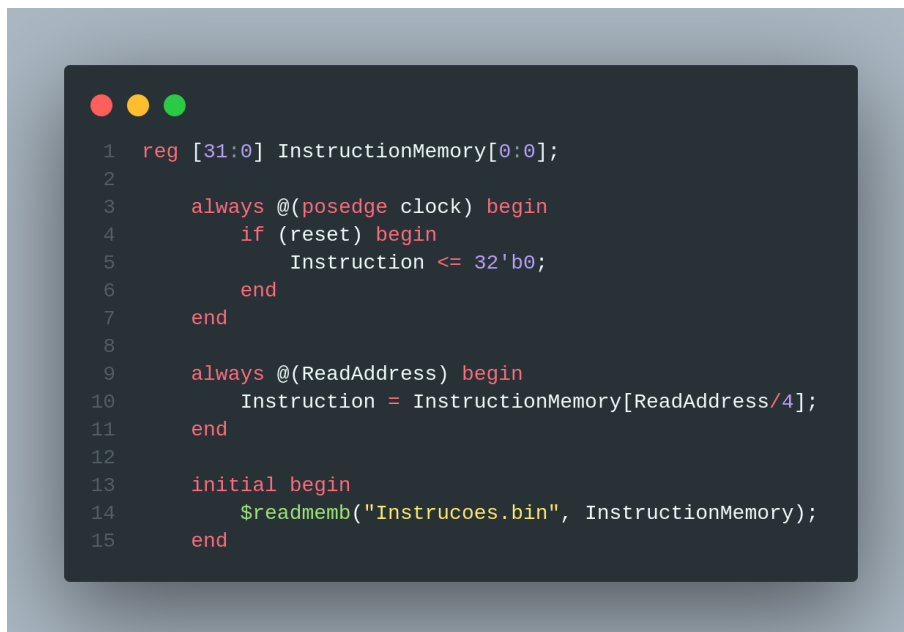
A screenshot of a code editor with a dark background and light-colored text. The code is Verilog for a RegisterFile module. It includes line numbers from 1 to 15 on the left. The code assigns initial values to registers x0, x1, and x2, followed by an ellipsis. It then contains two 'always' blocks: one for writing to the register array on the rising edge of the clock when RegWrite is active, and another for reading from the register array to ReadData1 and ReadData2.

```
1  assign x0 = register[0];
2      assign x1 = register[1];
3      assign x2 = register[2];
4      ...
5
6      always @(posedge clock) begin
7          if (RegWrite) begin
8              register[WriteReg] <= WriteData;
9          end
10     end
11
12     always @* begin
13         ReadData1 = register[ReadReg1];
14         ReadData2 = register[ReadReg2];
15     end
```

Figure 2. Trecho do módulo RegisterFile

2.3. InstructionMemory

O módulo InstructionMemory é responsável por armazenar as instruções que serão executadas pelo processador. Ele possui como entrada o clock, reset e *ReadAddress*, que é o endereço da instrução que deve ser lida, e a saída é a instrução do endereço informado. As instruções são lidas de um arquivo "Instrucoes.bin", que contém todas as instruções no formato binário. As instruções são armazenadas no InstructionMemory, que é uma espécie de array de instruções. Sempre que o endereço é atualizado, o módulo atribui à saída *Instruction* a nova instrução do endereço informado. Como os endereços são atualizados de 4 em 4, é necessário fazer a divisão por 4 para obter o índice correto dentro do InstructionMemory.

A screenshot of a code editor with a dark background and light-colored text. The code is written in Verilog and is numbered from 1 to 15 on the left side. The code defines a 32-bit register named InstructionMemory, which is updated on the rising edge of the clock. It also includes a reset condition where the Instruction register is set to 32'b0. The Instruction register is updated with the value from InstructionMemory at the address specified by ReadAddress divided by 4. Finally, the InstructionMemory is initialized with the contents of the file "Instrucoes.bin".

```
1  reg [31:0] InstructionMemory[0:0];
2
3  always @(posedge clock) begin
4      if (reset) begin
5          Instruction <= 32'b0;
6      end
7  end
8
9  always @(ReadAddress) begin
10     Instruction = InstructionMemory[ReadAddress/4];
11 end
12
13 initial begin
14     $readmemb("Instrucoes.bin", InstructionMemory);
15 end
```

Figure 3. Trecho do módulo InstructionMemory

2.4. ALU (Unidade Lógica Aritmética)

O módulo ALU é responsável por executar operações aritméticas e lógicas em dois operandos de entrada, *ReadData1* e *ReadData2*, com base na operação especificada por *Operation*. Ele possui como saída o resultado da operação (*Result*) e um sinal indicando se o resultado é zero (*Zero*). As operações suportadas incluem adição (ADD), subtração (SUB), deslocamento lógico à esquerda (SLL) e operações lógicas OR, AND e ORI.

```
1  always @* begin
2      case (Operation)
3          4'b0000: begin // LD e SD
4              Result <= ReadData1 + ReadData2;
5          end
6          4'b0001: begin // BEQ
7              Result <= ReadData1 - ReadData2;
8          end
9          4'b0010: begin // ADD
10             Result <= ReadData1 + ReadData2;
11         end
12         4'b0011: begin // SUB
13             Result <= ReadData1 - ReadData2;
14         end
15         4'b0100: begin // SLL
16             Result <= ReadData1 << ReadData2[4:0];
17         end
18         4'b0101: begin // OR
19             Result <= ReadData1 | ReadData2;
20         end
21         4'b0110: begin // AND
22             Result <= ReadData1 & ReadData2;
23         end
24         4'b0111: begin // ORI
25             Result <= ReadData1 | ReadData2;
26         end
27     endcase
28
29     Zero <= (Result == 0);
```

Figure 4. Trecho do módulo ALU

2.5. ALUControl

O módulo ALUControl é responsável por gerar o sinal de controle (*Operation*) para a Unidade Lógico-Aritmética (ALU) com base nos sinais de controle *ALUOP*, *Funct3* e *Funct7*. Ele possui como saída o sinal *Operation*, que especifica a operação a ser executada pela ALU. O *ALUOP* fornece o formato da instrução e o *Funct3* e *Funct7* determinam exatamente qual operação deve ser executada.

```
1  always @(*) begin
2      case (ALUOP)
3          3'b000:
4              begin
5                  Operation = 4'b0000; // LD e SD
6              end
7
8          3'b001:
9              begin
10                 Operation = 4'b0001; // BEQ
11             end
12
13         3'b010: // R-Type
14             begin
15                 case(Funct3)
16                     3'b000:
17                         begin
18                             case(Funct7)
19                                 7'b0000000:
20                                     begin
21                                         Operation = 4'b0010; // ADD
22                                     end
23
24                                     7'b0100000:
25                                         begin
26                                             Operation = 4'b0011; // SUB
27                                         end
28                                 endcase
29                             end
30                         end
31                     3'b001:
32                         begin
33                             Operation = 4'b0100; // SLL
34                         end
35                     3'b110:
36                         begin
37                             Operation = 4'b0101; // OR
38                         end
39                     3'b111:
40                         begin
41                             Operation = 4'b0110; // AND
42                         end
43                     end
44                 endcase
45             end
46         end
47
48         3'b011:
49             begin
50                 Operation = 4'b0101; // ORI
51             end
52         endcase
53     end
```

Figure 5. Trecho do módulo ALUControl

2.6. Control

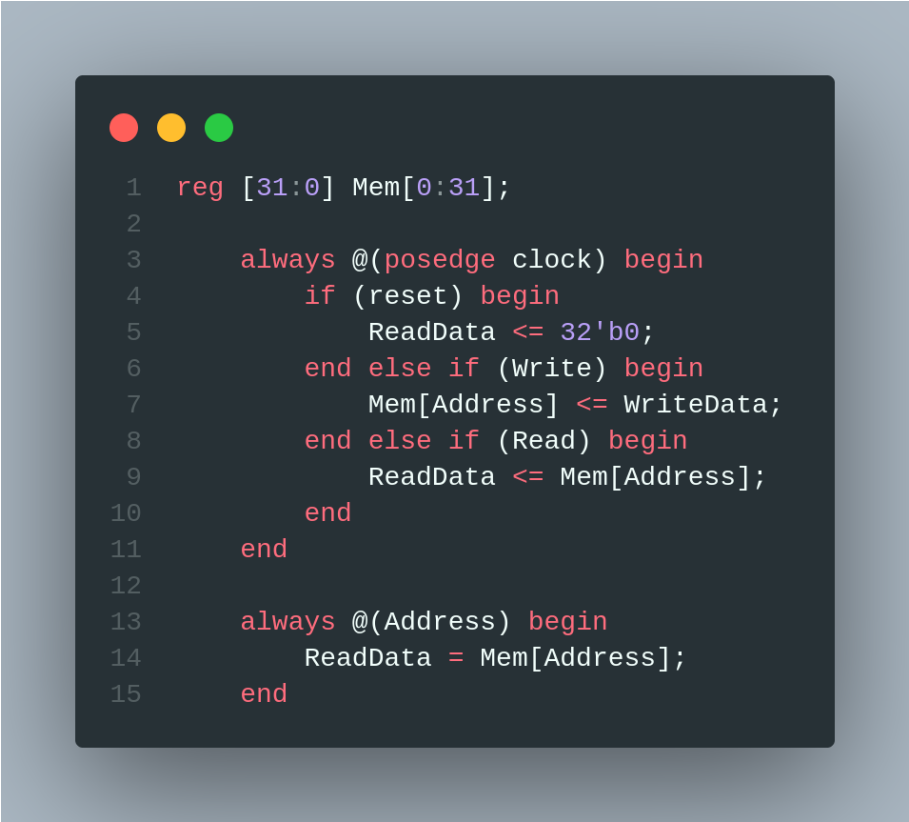
O módulo Control é responsável por gerar os sinais de controle necessários para coordenar as diferentes etapas do processador e controlar a execução das instruções. Ele recebe como entrada os sinais *opcode* e *Funct3*, que indicam o tipo e a operação da instrução atual. Com base nessas entradas, o módulo Control gera uma série de sinais de controle, como *MemRead*, *MemWrite*, *BranchZero*, *BranchNotZero*, *ALUOp*, *WriteSrc*, *ALUSrc* e *RegWrite*, que são responsáveis por controlar o acesso à memória, a fonte de dados para escrita em registradores, desvios condicionais e a operação a ser executada pela Unidade Lógico-Aritmética (ALU). Esses sinais de controle são essenciais para garantir a correta execução das instruções e o correto encaminhamento dos dados no processador.

```
1  case (Opcode)
2      7'b1100011: begin //SB-Type
3          case (funct3)
4              3'b000: begin //BEQ
5                  BranchNotZero = 1'b0;
6                  BranchZero = 1'b1;
7              end
8              3'b001: begin //BNE
9                  BranchNotZero = 1'b1;
10                 BranchZero = 1'b0;
11             end
12         endcase
13     end
14     default: begin
15         BranchNotZero = 1'b0;
16         BranchZero = 1'b0;
17     end
18 endcase
```

Figure 6. Trecho do módulo Control

2.7. DataMemory

O módulo DataMemory é responsável por armazenar os dados na memória e realizar operações de leitura e escrita nesses dados. Ele possui como entradas o clock, reset, *Address*, *WriteData*, *Write* e *Read*, e a saída é o *ReadData*, que representa o dado lido da memória. O módulo utiliza uma memória chamada "Mem", que é um array de 32 palavras de 32 bits cada. Caso o sinal *Write* esteja ativado, o módulo realiza a escrita do dado *WriteData* no endereço especificado pelo sinal *Address*. Caso o sinal *Read* esteja ativado, o módulo lê o dado presente no endereço especificado por *Address* e o armazena em *ReadData*.

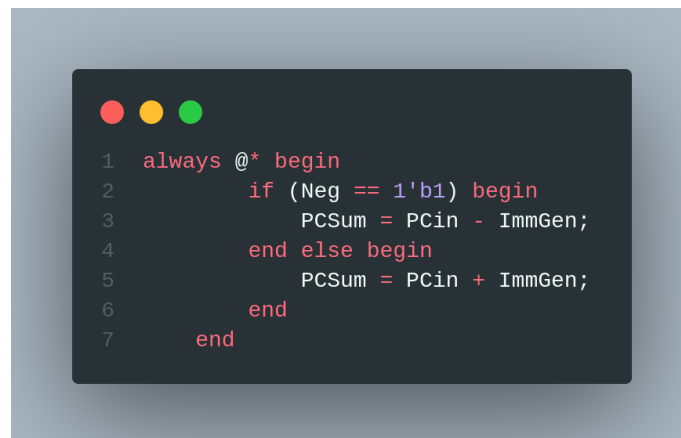
A screenshot of a code editor with a dark background and light-colored text. The code is Verilog for a DataMemory module. It features a 32-bit register 'Mem' and two always blocks. The first block is triggered by the clock (posedge) and handles reset, write, and read operations. The second block is triggered by the address and assigns the value from memory to ReadData. Line numbers 1 through 15 are visible on the left side of the code block.

```
1  reg [31:0] Mem[0:31];
2
3  always @(posedge clock) begin
4      if (reset) begin
5          ReadData <= 32'b0;
6      end else if (Write) begin
7          Mem[Address] <= WriteData;
8      end else if (Read) begin
9          ReadData <= Mem[Address];
10     end
11 end
12
13 always @(Address) begin
14     ReadData = Mem[Address];
15 end
```

Figure 7. Trecho do módulo DataMemory

2.8. PC Counter

O módulo PC Counter é responsável por controlar o Program Counter (PC) dentro do processador. Ele garante a sequência correta de execução das instruções. O PC Counter recebe sinais de clock, reset e *Branch*. Durante a execução, o PC é incrementado em 4 para apontar para a próxima instrução. Em casos de desvio condicional, o PC é atualizado para o endereço de destino determinado pela instrução. Essa atualização do PC permite o controle do fluxo de execução do programa, garantindo a correta sequência de instruções a serem executadas pelo processador.

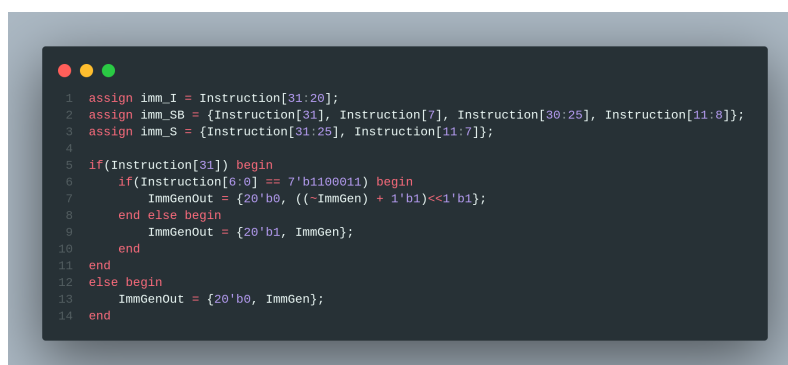
A screenshot of a code editor showing a Verilog code snippet for the PC Counter module. The code is written in a dark-themed editor with line numbers 1 through 7 on the left. The code defines an 'always' block that checks the 'Neg' signal. If 'Neg' is 1'b1, it calculates 'PCSum' as 'PCin - ImmGen'. Otherwise, it calculates 'PCSum' as 'PCin + ImmGen'.

```
1 always @* begin
2     if (Neg == 1'b1) begin
3         PCSum = PCin - ImmGen;
4     end else begin
5         PCSum = PCin + ImmGen;
6     end
7 end
```

Figure 8. Trecho do módulo PC Counter

2.9. Immediate Generator

O módulo Immediate Generator (ImmGen) é responsável por gerar o valor imediato (*immediate*) necessário para a execução de algumas instruções. Ele recebe como entrada o campo imediato da instrução e realiza a extensão de sinal adequada para obter o valor imediato completo. A extensão de sinal é necessária para garantir que o valor imediato seja representado corretamente, seja ele positivo ou negativo. O valor imediato gerado é utilizado em operações aritméticas, lógicas e de desvio condicional, permitindo que as instruções sejam executadas corretamente com os valores adequados.

A screenshot of a code editor showing a Verilog code snippet for the Immediate Generator module. The code is written in a dark-themed editor with line numbers 1 through 14 on the left. The code assigns 'imm_I' to 'Instruction[31:20]', 'imm_SB' to '{Instruction[31], Instruction[7], Instruction[30:25], Instruction[11:8]}', and 'imm_S' to '{Instruction[31:25], Instruction[11:7]}'. It then uses an 'if' statement to check 'Instruction[31]'. If it's 1, it checks 'Instruction[6:0]'. If 'Instruction[6:0]' is 7'b100011, it calculates 'ImmGenOut' as {20'b0, ((~ImmGen) + 1'b1) << 1'b1}. Otherwise, it calculates 'ImmGenOut' as {20'b1, ImmGen}.

```
1 assign imm_I = Instruction[31:20];
2 assign imm_SB = {Instruction[31], Instruction[7], Instruction[30:25], Instruction[11:8]};
3 assign imm_S = {Instruction[31:25], Instruction[11:7]};
4
5 if(Instruction[31]) begin
6     if(Instruction[6:0] == 7'b100011) begin
7         ImmGenOut = {20'b0, ((~ImmGen) + 1'b1) << 1'b1};
8     end else begin
9         ImmGenOut = {20'b1, ImmGen};
10    end
11 end
12 else begin
13     ImmGenOut = {20'b0, ImmGen};
14 end
```

Figure 9. Trecho do módulo Immediate Generator

2.10. Clock

O módulo Clock é responsável por gerar um sinal de clock utilizado para sincronizar as operações no processador. Esse sinal de clock é uma sequência de pulsos regulares que define o ritmo de funcionamento do processador, controlando a execução das instruções e o fluxo de dados entre os diferentes componentes. O sinal de clock é essencial para garantir a correta sincronização das operações e evitar conflitos de dados. Ele é gerado de acordo com uma frequência específica e é utilizado como referência temporal para a execução das tarefas dentro do processador.

3. Fluxo dos Módulos

O caminho de dados implementado para a execução das instruções segue um fluxo bem definido. A execução começa com o contador de programa (PC) fornecendo o endereço da próxima instrução ao módulo de memória de instruções (InstructionMemory). O InstructionMemory busca a instrução correspondente ao endereço fornecido e a retorna.

Em seguida, a instrução é decodificada pelo módulo InstructionDecoder. Esse módulo identifica os sinais de controle necessários para as operações subsequentes. Com base nos sinais de controle, o módulo RegisterFile lê os valores dos registradores necessários para a execução da instrução. Se a instrução requer um valor imediato, o módulo ImmGen entra em ação. Ele gera o valor imediato a partir dos campos relevantes da instrução.

O módulo MuxAlu desempenha um papel importante na seleção da fonte correta de dados para a ALU. Dependendo dos sinais de controle, o MuxAlu escolhe entre o valor do registrador ou o valor imediato gerado pelo ImmGen, como entrada para a ALU. Em seguida, o módulo ALUControl utiliza os campos da instrução e o ALUOP para determinar a operação que a ALU deve realizar. Com base nas informações do ALUControl, a ALU executa a operação especificada.

Após a execução da ALU, a memória de dados (DataMemory) entra em ação quando necessário. Ela realiza operações de escrita ou leitura na memória, utilizando o endereço fornecido e os dados apropriados, sendo dependente dos sinais Write e Read fornecidos pelo Control. Se houver necessidade, os resultados finais são armazenados de volta nos registradores pelo módulo RegisterFile. Isso permite que os dados sejam preservados para uso posterior.

Por fim, o PC é atualizado com base nas condições de ramificação e saltos, determinadas pelo módulo JumpControl, que analisa se haverá a necessidade de desvio ou não, variando de acordo se as instruções lidas são do tipo SB. Essa atualização define o sinal para o cálculo do novo endereço a ser passado no próximo ciclo.

Esse fluxo contínuo de busca, decodificação, execução e armazenamento de instruções permite que o processador execute programas sequencialmente, seguindo as instruções contidas na memória de instruções. Cada módulo desempenha um papel específico na manipulação dos dados e controle do fluxo de execução, garantindo a correta interpretação e operação das instruções.

4. Resultados dos Testes

Durante o desenvolvimento deste trabalho, foram realizados testes abrangendo diferentes operações lógicas e aritméticas definidas para o grupo. Os resultados obtidos foram consistentes, demonstrando a correta execução das instruções e o armazenamento dos valores esperados nos registradores e memória de dados. O caminho de dados mostrou eficiência no processamento das instruções, cumprindo os requisitos de desempenho estabelecidos. Esses testes confirmaram a funcionalidade do caminho de dados implementado, validando o sucesso do projeto. A sequência de instruções executadas para os testes está ilustrada na Figura 10, e os resultados obtidos para cada registrador estão apresentados na Figura 11. Além disso, as formas de ondas obtidas após a simulação no GTKWave podem ser visualizadas na Figura 12.

```
add x5, x5, x5
sb x5, 0(x0)
lb x3, 0(x0)
sll x5, x5, x30
sub x6, x5, x3
ori x7, x3, 1
and x4, x3, x7
or x8, x3, x7
```

Figure 10. Sequência de instruções executadas para testes

Registrador 0:	0
Registrador 1:	0
Registrador 2:	0
Registrador 3:	2
Registrador 4:	2
Registrador 5:	4
Registrador 6:	2
Registrador 7:	3
Registrador 8:	3
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:	1
Registrador 31:	0

Figure 11. Resultados obtidos para cada registrador

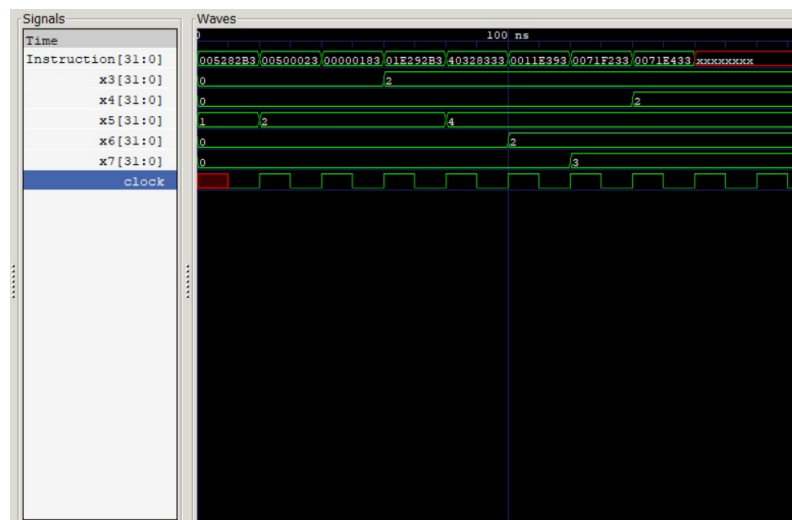


Figure 12. Formas de ondas obtidas após a simulação no GTKWave

Conclusão

Este trabalho teve como objetivo sintetizar os conhecimentos relacionados ao caminho de dados em um processador na arquitetura RISC-V. Durante o projeto, foram implementados módulos essenciais, como RegisterFile, InstructionMemory, ALU, ALUControl, Control, DataMemory, PC Counter e Immediate Generator, que desempenham papéis específicos na execução das instruções, desde a busca e decodificação até a execução e armazenamento dos resultados.

O fluxo de dados foi estabelecido de forma a garantir a sincronização correta das operações e o encaminhamento adequado dos dados entre os componentes. A interpretação do diagrama do caminho de dados proporcionou uma visão clara do fluxo e da interação entre os blocos funcionais, permitindo compreender a estrutura do processador e como as instruções são processadas em cada etapa.

A implementação em Verilog e a execução de testes com instruções assembly validaram o funcionamento correto do caminho de dados. Essa abordagem prática contribuiu para uma melhor compreensão dos conceitos teóricos.

Em conclusão, este trabalho proporcionou uma sólida compreensão do caminho de dados em um processador RISC-V, desde a busca e decodificação até a execução e armazenamento dos resultados.

References

- [1] Patterson, D. A., & Hennessy, J. L. (2017). *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann.
- [2] Asanović, K., & Patterson, D. (2014). *The RISC-V Instruction Set Manual: Volume I: User-Level ISA Version 2.2*. Technical Report.
- [3] Hines, J. (2023). *RISC-V Graphical Datapath Simulator*. Retrieved from: <https://jesse-r-s-hines.github.io/RISC-V-Graphical-Datapath-Simulator/>