
UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
ENGENHARIA DA COMPUTAÇÃO

MARIA GABRIELLA DA SILVA MEDEIROS
MÁRIO DANIEL TELES DA SILVA FILHO
MIRIAM GONZAGA DA SILVA SANTOS

# PROJETO DE LABORATÓRIO PROCESSADOR RISC-V (RV32I) EM SYSTEMVERILOG

RECIFE
2025

MARIA GABRIELLA DA SILVA MEDEIROS
MÁRIO DANIEL TELES DA SILVA FILHO
MIRIAM GONZAGA DA SILVA SANTOS

# PROJETO DE LABORATÓRIO PROCESSADOR RISC-V (RV32I) EM SYSTEMVERILOG

Relatório apresentado ao Curso de Engenharia da Computação do Centro de Informática da Universidade Federal de Pernambuco, como avaliação na disciplina CIN 0012 - Laboratório de Organização e Arquitetura de Computadores, em atendimento ao requisito parcial para a conclusão do Projeto de Laboratório intitulado Processador RISC-V (RV32I) em *SystemVerilog*.

Professora: Edna Natividade da Silva Barros
Professor: Victor Wanderley Costa de Medeiros

RECIFE
2025

# SUMÁRIO

## 1. INTRODUÇÃO

O pipeline é um mecanismo essencial para aumentar o desempenho de processadores, permitindo que várias instruções sejam executadas simultaneamente em estágios distintos. A divisão em cinco etapas (IF - busca, ID - decodificação, EX - execução, MEM - acesso à memória e WB - escrita no registrador) possibilita maior aproveitamento do hardware, à medida que reduz o tempo médio de processamento por instrução.

O presente projeto tem como objetivo implementar um processador RISC-V de 32 bits com pipeline de cinco estágios, desenvolvido em SystemVerilog. A arquitetura foi projetada de forma modular, abrangendo um subconjunto significativo do conjunto de instruções RV32I. Dessa forma, o projeto possibilita compreender, na prática, os princípios de execução paralela, controle de fluxo e organização interna de processadores modernos.

Nos tópicos a seguir, são detalhadas as principais decisões de projeto referentes a cada módulo e às classes de instruções implementadas. Para isso, as instruções do conjunto RV32I foram organizadas de acordo com seus formatos: R-type, I-type, S-type, B-type, U-type e J-type.

# 2. IMPLEMENTAÇÃO DAS INSTRUÇÕES

## 2.1. Instruções R-Type

### 2.1.1. Instruções SUB, SLT, XOR e OR

Para implementar as instruções do tipo R, o **Controller** é o primeiro módulo responsável por interpretar o opcode da instrução. Para instruções do tipo **R-type**, o opcode é 0110011, e ao reconhecê-lo o Controller ativa sinais compatíveis com esse formato: habilita escrita em registrador (RegWrite = 1), indica que o operando B vem de um registrador (ALUSrc = 0) e define ALUOp = 2'b10, informando que a operação da ULA deve ser decidida com base nos campos funct3 e funct7. Ou seja, para instruções R-type, o Controller apenas classifica o tipo da instrução e delega ao ALUController a escolha da operação aritmética ou lógica correta.

```
// controle dos dois bits da ALUOp (do menos significativo pro mais significativo)
assign ALUOp[0] = (Opcode == BR || Opcode == JAL || Opcode == JALR); // bit menos significativo do ALUOp (01 = Branch, 11 = JAL/JALR)
assign ALUOp[1] = (Opcode == R_TYPE || Opcode == IMM || Opcode == JAL || Opcode == JALR); // bit mais significativo do ALUOp (10 = Rtype/IMM, 11 = JAL/
```

Imagem 1: Trecho do código que define o valor de ALUOp.

O **ALUController** recebe ALUOp do módulo Controller e, no caso de ALUOp = 2'b10, analisa os campos **funct3** e **funct7** da instrução R-type para definir qual operação deve ser executada. Dessa forma, ele distingue entre todas as instruções aritméticas: **ADD**, **SUB**, **SLL**, **SLT**, **XOR**, **SRL**, **SRA**, **OR**, **AND** possivelmente operadas. A partir dessa combinação, o ALUController gera um código de controle de 4 bits chamado Operation, que é enviado diretamente para a ULA.

```
2'b10: begin // ALUOp = 2'b10 (R-type)
    case(Funct3) //tratar operações com mesmo funct3

        3'b000: begin // (sub, add, addi)

            if (Funct7 == 7'b0100000) //sub
                Operation = 4'b0001;
```

Imagem 2: Sinal de Operation é definido no módulo ALUController (Instrução SUB).

```
3'b010: // (slt, slti)
    Operation = 4'b0101;
//3'b011 são operação de sltu e sltiu - nao foram implementadas :)

3'b100: // (xor)
    Operation = 4'b0100; // (xor)
```

Imagem 3: Definição do Operation das instruções SLT, XOR a serem realizadas na ALU.

A **ALU** recebe os operandos SrcA e SrcB vindos do banco de registradores e o sinal Operation gerado pelo ALUController. Com base nesse código de controle, ela executa a operação correspondente e devolve o resultado à ALUResult. Para instruções R-type, isso permite que cada instrução execute exatamente o comportamento esperado, de forma coordenada entre os três módulos.

```
3'b110: // (or)
    Operation = 4'b0011;
```

Imagem 4: Definição do Operation da instrução  OR a ser realizada na ALU.

## 2.2. Instruções I-Type

### 2.2.1. Instruções I-Type e LOADs

As instruções do tipo I utilizam dois registradores (rd, rs1) e um **imediato** como operando. Esse grupo inclui tanto as instruções aritméticas imediatas **(ADDI, SLTI, SLLI, SRLI, SRAI**), quanto as instruções de leitura da memória **LOAD** (LH, LB, LBU). Embora instruções aritméticas e loads tenham finalidades diferentes, ambos os formatos compartilham a característica fundamental de usar um imediato estendido como operando B da ULA (que é determinado no módulo Controller através do **ALUSrc**).

A primeira etapa do processamento ocorre no **Controller**, que identifica o opcode correspondente. Para instruções aritméticas imediatas, o opcode é 0010011, enquanto para LOADs é 0000011. Em ambos os casos, o Controller ativa sinais essenciais: habilita a escrita em registrador (RegWrite = 1), seleciona o imediato como operando B da ULA (ALUSrc = 1), e define o ALUOp. No caso das instruções aritméticas imediatas, ALUOp = 2'b10, indicando que a operação deve ser selecionada a partir de funct3 e funct7. Já para LOADs, ALUOp = 2'b00, pois a ULA deve apenas calcular o endereço de memória (soma de rs1 + imediato). Assim, o papel do Controller é classificar o tipo de operação, ajustar os sinais de escrita e memória e encaminhar o fluxo correto para o ALUController.

O **ALUController** recebe o sinal ALUOp e, dependendo do valor, decide qual operação deve ser executada pela ULA. Para instruções I aritméticas (ALUOp = 2'b10), ele interpreta funct3 e, quando necessário, funct7, para distinguir entre ADDI, SLTI, SLLI, SRLI e SRAI. Para LOADs (ALUOp = 2'b00), o ALUController simplesmente gera o código de operação de soma, pois todo LOAD precisa somar o registrador base rs1 ao imediato deslocado. O resultado dessa decodificação é o sinal Operation, de 4 bits, enviado diretamente para a ULA.

```
3'b001: // (sll, slli)
    Operation = 4'b1001;

3'b010: // (slt, slti)
    Operation = 4'b0101;
```

Imagem 5: Definição do Operation das instruções SLL, SLLI, SLT e SLTI  a serem realizadas na ALU.

```
3'b101: begin// (srli, srai)

    if (Funct7 == 7'b0000000) // srli
        Operation = 4'b1010;
    else if (Funct7 == 7'b0100000) // srai
        Operation = 4'b1011;
end
```

Imagem 6: Definição do Operation das instruções SRLI, SRAI  a serem realizadas na ALU.

A **ALU** recebe o operando vindo de rs1 (SrcA), o imediato estendido (SrcB, gerado pelo módulo de imediato), e o sinal Operation. Para instruções tipo I aritméticas, ela executa a lógica ou aritmética correspondente (adição, deslocamentos, comparações e operações lógicas).

```
always_comb
begin
    case(Operation) //cada operação é um possível caso:

        4'b0000:        // AND
                ALUResult = SrcA & SrcB;
        4'b0001:        // SUB
                ALUResult = SrcA - SrcB;
        4'b0010: //ADD E ADDI
                ALUResult = SrcA + SrcB;
        4'b0011:        // OR
                ALUResult = SrcA | SrcB;
        4'b0100:        // XOR
                ALUResult = SrcA ^ SrcB;
        4'b0101: //SLT e SLTI (usado também pelo blt, bge)
                ALUResult = ($signed(SrcA) < $signed(SrcB)) ? 32'h00000001 : 32'h00000000; //signed (considera valores negativos)
        4'b1000:       //  Equal (usado pelo beq e bne)
                ALUResult = (SrcA == SrcB) ? 32'h00000001 : 32'h00000000;
        4'b1001:       //  SLLI
                ALUResult = SrcA << SrcB[4:0];
        4'b1010:       //  SRLI
                ALUResult = SrcA >> SrcB[4:0];
        4'b1011:       //  SRAI
                ALUResult = $signed(SrcA) >>> SrcB[4:0]; //levar em consideração o sinal

        default: //se não for nenhum dos casos válidos descritos acima:
                ALUResult = 32'b0; //resultado = 0

    endcase
end
```

Imagem 7: Organização da ALU com as operações aritméticas baseadas nos ‘operations’.

Para LOADs, a ULA calcula o endereço efetivo de memória, que é enviado ao módulo de memória (datamemory). O valor resultante é enviado para o Write Back, armazenando no registrador destino exatamente o conteúdo lido da memória (quando MemtoReg = 1).

```
assign MemtoReg = (Opcode == LW); // valor da memória é carregado para registradores (loads)
// regwrite escreve em registrador (guarda pc+4)
assign RegWrite = (Opcode == R_TYPE || Opcode == LW || Opcode == IMM || Opcode == JAL || Opcode == JALR); // instruções que escrevem em registradores
```

Imagem 8: Organização para acesso e escrita na memória pelos registradores.

No caso dos LOADs, o módulo **datamemory** interpreta o Funct3 para decidir se a leitura será de um byte, halfword ou word e se terá ou não extensão de sinal. Esse dado é então enviado ao registrador rd. Assim, para instruções tipo I aritméticas e LOADs, a interação entre Controller, ALUController e ULA garante que o imediato seja processado corretamente, seja para computações internas ou para cálculos de endereços de memória.

```
if (MemRead) begin
  case (Funct3)
    3'b000: begin //LB
      case (a[1:0]) //determinar qual byte deve ser lido
        2'b00: rd = {{24{Dataout[7]}}, Dataout[7:0]};    // byte 0
        2'b01: rd = {{24{Dataout[15]}}, Dataout[15:8]};  // byte 1
        2'b10: rd = {{24{Dataout[23]}}, Dataout[23:16]}; // byte 2
        2'b11: rd = {{24{Dataout[31]}}, Dataout[31:24]}; // byte 3
      endcase
    end
    3'b001: begin //LH
      case (a[1:0]) //determinar o halfword a ser lido
        2'b00: rd = {{16{Dataout[15]}}, Dataout[15:0]};    // halfword 0
        2'b10: rd = {{16{Dataout[31]}}, Dataout[31:16]};  // halfword 1
        default: rd = 32'b0; //quando o endereço estiver desalinhado
      endcase
    end
    3'b010:  //LW
      rd = Dataout;
    3'b100: begin //LBU
      case (a[1:0])
        2'b00: rd = {24'b0, Dataout[7:0]};    // byte 0
        2'b01: rd = {24'b0, Dataout[15:8]};  // byte 1
        2'b10: rd = {24'b0, Dataout[23:16]}; // byte 2
        2'b11: rd = {24'b0, Dataout[31:24]}; // byte 3
      endcase
    end
    default: rd = Dataout;
  endcase
```

Imagem 9: Atribuição dos endereços de leitura bit a bit

## 2.3. Instruções S-Type

### 2.3.1. Instruções SB, SH

As instruções de acesso à memória (para escrita) do conjunto RV32I utilizam o **formato S-type**, cujo imediato é formado pelos campos inst[31:25] e inst[11:7]. Para instruções **SB** (Store Byte) e **SH** (Store Halfword), esse imediato define o deslocamento aplicado ao endereço base contido em um registrador, permitindo armazenar apenas parte da palavra (word) na memória.

O imediato S-type é gerado pelo módulo **imm_Gen**, que realiza a extensão de sinal e organiza corretamente os 12 bits utilizados para o cálculo do endereço (após ser tratado no imm_Gen, o imediato passa a ter 32 bits). Esse imediato é somado ao registrador base na **ALU**, resultando no endereço efetivo de memória. No estágio **MEM**, o módulo de memória utiliza o campo **Funct3**, os bits menos significativos do endereço (**a[1:0]**) e os sinais de controle para determinar quais bytes devem ser escritos, como os dados devem ser alinhados dentro da palavra de 32 bits e se a operação de escrita é válida ou deve ser ignorada em casos de desalinhamento.

A instrução **SB** armazena apenas **8 bits** (wd[7:0]) no endereço calculado. Como a memória é organizada em palavras de 32 bits, o módulo determina **qual dos quatro bytes** deve ser escrito analisando os bits menos significativos do endereço (a[1:0]). Para cada caso, ativa-se apenas **um bit de Wr**, habilitando a escrita de um único byte:

```
case (Funct3)
  3'b000: begin  //SB
    case(a[1:0])
      2'b00: begin
        Wr = 4'b0001; //escrever o byte 0
        Datain = {24'b0, wd[7:0]};
      end
      2'b01: begin
        Wr = 4'b0010; //escrever o byte 1
        Datain = {16'b0, wd[7:0], 8'b0};
      end
      2'b10: begin
        Wr = 4'b0100; //escrever o byte 2
        Datain = {8'b0, wd[7:0], 16'b0};
      end
      2'b11: begin
        Wr = 4'b1000; //escrever o byte 3
        Datain = {wd[7:0], 24'b0};
      end
    endcase
  end
```

Imagem 10: Escrita de um único byte em Datain

A instrução **SH** armazena **16 bits** (wd[15:0]) na memória, e por isso só é válida quando o endereço está alinhado a 2 bytes (últimos bits 00 ou 10).

```
3'b001: begin //SH
  case(a[1:0])  //determina o endereço da operação
  2'b00: begin
    Wr = 4'b0011; //escreve halfword 0;
    Datain = {16'b0, wd[15:0]};
  end
  2'b10: begin
    Wr = 4'b1100; //escreve halfword 1;
    Datain = {wd[15:0], 16'b0};
  end
  default: begin
    Wr = 4'b0000; //quando o endereço estiver desalinhado
    Datain = 32'b0;
  end
  endcase
end
```

Imagem 11: Escrita de dois bytes em Datain

## 2.4. Instruções B-Type

### 2.4.1. Instruções BEQ, BNE, BLT, BGE

As instruções de desvio condicional (**BEQ**, **BNE**, **BLT** e **BGE**) são implementadas no processador por meio da integração entre o módulo **ALU**, o **BranchUnit** e os sinais de controle gerenciados no módulo **Datapath**. Todas essas instruções seguem o formato **B-type**, no qual o imediato representa um deslocamento a ser somado ao valor atual do PC para calcular o endereço de destino do branch. A ULA é responsável por realizar a comparação entre os registradores rs1 e rs2, produzindo um resultado cujo bit menos significativo indica o resultado do teste lógico.

```
7'b1100011:  /*B-type*/
  Imm_out = {
    {19{inst_code[31]}},
    inst_code[31],
    inst_code[7],
    inst_code[30:25],
    inst_code[11:8],
    1'b0
    };
```

Imagem 12: Formato do Imediato das Instruções B-Types (Módulo imm_Gen)

Para **BEQ**, o desvio é tomado quando rs1 e rs2 são iguais, o que corresponde a (ALUResult[0] = 1), já em **BNE**, o desvio ocorre quando esses valores são diferentes, utilizando o valor negado desse mesmo bit. As instruções **BLT** e **BGE** recorrem à operação de "set-less-than" (SLT) da ULA: em **BLT**, o branch é tomado quando rs1 < rs2 (ALUResult[0] = 1), enquanto em **BGE** o desvio só é realizado quando rs1 ≥ rs2, representado pelo valor negado desse bit.

```
2'b01: begin//operações de branche (beq, bne, blt, bge)
    case(Funct3) //tratar operações conforme o funct3

        3'b000: // beq
            Operation = 4'b1000; // equal na ula
        3'b001: // bne
            Operation = 4'b1000; // equal na ula
        3'b100: // blt
            Operation = 4'b0101; // slt na ula
        3'b101: // bge
            Operation = 4'b0101; // slt na ula

    endcase
end
```

Imagem 13: Definição do Operation das instruções de branch a serem realizadas na ALU.

O módulo **BranchUnit** recebe o sinal Branch, o funct3 da instrução e o resultado da ULA, determinando se o desvio deve ser tomado. Quando a condição é satisfeita, o módulo seleciona **PC = PC + Imm**, direcionando o fluxo para o bloco de código alvo; caso contrário, o fluxo segue sequencialmente para **PC + 4**.

```
always_comb begin
    Branch_Sel = 1'b0; // inicializando variável
    if (Branch) begin
        case(Funct3)
            // o bit menos significativo do AluResult indica o resultado do equal e slt (0 ou 1)

            3'b000: // beq
                Branch_Sel = AluResult[0]; // assume o valor do Equal
            3'b001: // bne
                Branch_Sel = ~AluResult[0]; // assume o valor negado do Equal
            3'b100: // blt
                Branch_Sel = AluResult[0]; // assume o valor do SLT
            3'b101: // bge
                Branch_Sel = ~AluResult[0]; // assume o valor contrário ao SLT

            default: // caso não seja nenhum dos funct3s acima
                Branch_Sel = 1'b0; // branch não é tomado!!
        endcase
    end

    else // caso o branch tenha sinal = 0
        Branch_Sel = 1'b0; //else do if(branch)
end
```

Imagem 14: Sinal de Branch_Sel recebe o valor de AluResult ou ~AluResult para decidir se o Branch deve ser tomado ou não.

## 2.5. Instruções J-Type

### 2.5.1. Instruções JAL, JALR

As instruções J-Type são tratadas pelo processador por meio da lógica combinacional presente no módulo BranchUnit, que identifica o tipo de salto a partir do campo *opcode* e calcula o endereço de destino correspondente.

```
2'b11: begin // jal e jalr
    Operation = 4'b0010; // realiza a soma
end
```

Imagem 15: Definição do Operation para as instruções JAL e JALR.

No caso de **JAL**, o endereço de desvio é obtido somando o valor atual do PC ao imediato estendido (inicialmente a instrução formene 20 bits de imediato, que se tornam 32 após tratamento no módulo imm_Gen), enquanto o registrador rd recebe o valor de PC + 4 no estágio WB, permitindo o retorno da subrotina.

```
7'b1101111: // instruções do tipo JAL (desembaralhar 21 bits de immediato)
  Imm_out = {{11{inst_code[31]}}, inst_code[31], inst_code[19:12], inst_code[20], inst_code[30:21], 1'b0};

7'b1100111: // instruções do tipo JALR (mesma estrutura das instruções do tipo I)
  Imm_out = {inst_code[31] ? 20'hFFFFF : 20'b0, inst_code[31:20]};
```

Imagem 16: Extensão de Sinal do Imediato da instrução JAL e JALR (módulo imm_Gen).

Já em **JALR**, o destino do salto é calculado como (rs1 + imediato), cujo resultado passa pela ULA e tem seu bit menos significativo forçado a zero para garantir alinhamento, conforme especificação do padrão RISC-V.

```
if (Jump) begin// se jump tiver sido tomado:
    case(Opcode)
        7'b1101111: begin// JAL
            // o endereço é pc + offset (vem do imm_Gen) = pc_imm
            Jump_Address = PC_Imm;
            Jump_Sel = 1'b1; // indica que jump foi tomado
        end

        7'b1100111: begin// JALR
            // o endereço é (reg1 + deslocamento) & ~1 (forçar alinhamento com endereço par!)
            // alu_result = reg1 + deslocamento
            Jump_Address = {AluResult[31:1], 1'b0}; // forçar o último bit a ser zero.
            Jump_Sel = 1'b1; //indicar que o jump foi tomado
        end

        default: begin // caso o opcode não seja jal nem jalr
            Jump_Address = PC_Full; // mantém o cur_pc
            Jump_Sel = 1'b0;
```

Imagem 17: Definição do sinal Jump_Address para execução das instruções JAL e JALR.

O BranchUnit seleciona entre esses endereços através do sinal Jump_Sel, priorizando JALR ou JAL caso um salto seja detectado, e repassa o destino ao multiplexador de atualização do PC no estágio IF. No caminho de escrita, o sinal Jump propaga-se até o estágio WB, ativando um multiplexador que substitui o resultado “normal” (ALU ou memória) pelo valor de PC + 4. Dessa forma, **JAL** e **JALR** integram-se corretamente ao pipeline, preservando o fluxo de execução e garantindo a compatibilidade com chamadas de função e desvios indiretos.

## 3. CONCLUSÃO

Com o uso da plataforma ModelSim e com o suporte do professor, todas as instruções implementadas e apresentadas ao longo deste relatório foram devidamente testadas. Dessa forma, o objetivo inicial de desenvolver um processador RISC-V (RV32I) funcional foi alcançado para as instruções JAL, JALR, BNE, BLT, BGE, LB, LH, LBU, SB, SH, SLTI, ADDI, SLLI, SRLI, SRAI, SUB, SLT, XOR e OR. Com exceção da instrução HALT, todas as operações previstas no escopo original foram implementadas, testadas e estão funcionando. Além do resultado prático, o desenvolvimento do projeto possibilitou uma compreensão significativamente mais profunda sobre princípios de execução paralela, controle de fluxo e a organização interna de processadores modernos, consolidando o aprendizado teórico por meio da aplicação prática.