Relatório detalhado, elaborado por Deoclécio Ivo, que apresenta a sequência cronológica das implementações das instruções, destacando o desenvolvimento progressivo, os ajustes realizados em cada etapa e a integração gradual das funcionalidades no pipeline, com ênfase na lógica, controle e manipulação dos sinais específicos para cada tipo de instrução.

IMPLEMENTAÇÃO DO R-TYPE E I-TYPE

instruções:
R-TYPE: instruções que realizam operações aritméticas e lógicas entre dois registradores, retornando o resultado em um registrador destino. Exemplo: `add`, `sub`, `and`, `or`, `sll`, `slt`.

I-TYPE (ARITH): instruções similares às R-TYPE, mas que usam um valor imediato (constante) como segundo operando em vez de um registrador. Exemplo: `addi`, `andi`, `ori`, `slli`, `slti`.

na primeira parte do projeto foram adicionadas as instruções R-TYPE e -TYPE (ARITH);

No controller.sv: adicionei os opcodes do R_TYPE e do I_TYPE (ARITH), e ativei os sinais de controle correspondentes no pipeline: RegWrite, ALUSrc e para o **I-TYPE_ARITH:** ALUOp, e WBSel.

```
23  // R-TYPE: Instruções aritméticas e lógicas entre registradores.
24  // Exemplos: add, sub, and, or, xor, sll, srl, sra, slt, sltu.
25  localparam R_TYPE      = 7'b0110011;
26
27  // I-TYPE (ARITH): Instruções aritméticas/lógicas com imediato.
28  // Exemplos: addi, andi, ori, xori, slli, srli, srai, slti, sltiu.
29  localparam I_TYPE_ARITH= 7'b0010011;
```

alucontroller.sv:
Após isso,Implementei no alucontroller.sv uma lógica que combina sinais auxiliares, `funct7` e `funct3` para definir o código da operação da ALU. Cada padrão dessa combinação é mapeado para um código de 4 bits que indica qual operação a ALU deve executar, como ADD para load/store, códigos específicos para branch, e operações R-type/I-type como ADD, SUB, AND, OR, entre outras.

```
// Operações R-type e I-type
{2'b10, 7'b0100000, 3'b000}: Operation = 4'b0011; // SUB
{2'b10, 7'b0000000, 3'b000}: Operation = 4'b0010; // ADD
{2'b10, 7'b???????, 3'b000}: Operation = 4'b0010; // ADDI (genérico)

{2'b10, 7'b0000000, 3'b111}: Operation = 4'b0000; // AND/ANDI
{2'b10, 7'b0000000, 3'b110}: Operation = 4'b0001; // OR/ORI
{2'b10, 7'b0000000, 3'b100}: Operation = 4'b0110; // XOR/XORI
{2'b10, 7'b???????, 3'b011}: Operation = 4'b1100; // SLTU/SLTIU
{2'b10, 7'b0000000, 3'b001}: Operation = 4'b0100; // SLL/SLLI
{2'b10, 7'b0000000, 3'b101}: Operation = 4'b0111; // SRL/SRLI
{2'b10, 7'b0100000, 3'b101}: Operation = 4'b1000; // SRA/SRAI
```

alu.sv:
após isso, Implementei no alu.sv um `case` que, com base no código de operação de 4 bits, executa a operação correspondente entre os operandos `SrcA` e `SrcB`. São tratadas operações lógicas (AND, OR, XOR, NOR), aritméticas (ADD, SUB) e shifts (SLL, SRL, SRA), aplicando a operação correta conforme o código recebido do controlador da ALU

```
case(Operation)
    // Operações Lógicas
    4'b0000: ALUResult = SrcA & SrcB;        // AND/ANDI
    4'b0001: ALUResult = SrcA | SrcB;        // OR/ORI
    4'b0110: ALUResult = SrcA ^ SrcB;        // XOR/XORI
    4'b1001: ALUResult = ~(SrcA | SrcB);     // NOR

    // Operações Aritméticas
    4'b0010: ALUResult = $signed(SrcA) + $signed(SrcB);  // ADD/ADDI
    4'b0011: ALUResult = $signed(SrcA) - $signed(SrcB);  // SUB

    // Shifts
    4'b0100: ALUResult = SrcA << (SrcB[4:0] & 5'b11111);  // SLL/SLLI
    4'b0111: ALUResult = SrcA >> (SrcB[4:0] & 5'b11111);  // SRL/SRLI
    4'b1000: ALUResult = $signed(SrcA) >>> (SrcB[4:0] & 5'b11111); // SRA/SRAI
```

IMPLEMENTAÇÃO DA BRANCH

instruções:

As instruções B-TYPE são usadas para operações de desvio condicional (branch), como `beq`, `bne`, `blt`, `bge`, e bgu. Elas comparam dois registradores e decidem se o fluxo do programa deve saltar para um endereço diferente.

No controller.sv, implementei o opcode do B_TYPE como 7'b1100011 e ativei os sinais de controle específicos para identificar e tratar as instruções de desvio condicional no pipeline, como Branch, ALUOp, e PcSel. Esses sinais permitem avaliar a condição de salto e decidir se o fluxo de execução deve ser desviado.

```
// B-TYPE: Instruções de desvio condicional (branch).
// Exemplos: beq, bne, blt, bge, bltu, bgeu.
localparam B_TYPE      = 7'b1100011;
```

No alucontroller.sv, implementei a lógica para mapear os códigos `funct3` das instruções B-TYPE, ignorando o campo `funct7`, para códigos de operação específicos da ALU que realizam comparações usadas para o branch, como BEQ, BNE, BLT e BGE.

```
// Operações R-type e I-type
{2'b10, 7'b0100000, 3'b000}: Operation = 4'b0011; // SUB
{2'b10, 7'b0000000, 3'b000}: Operation = 4'b0010; // ADD
{2'b10, 7'b???????, 3'b000}: Operation = 4'b0010; // ADDI (genérico)
```

No **alu.sv**,implementei as operações de comparação baseadas nesse código, que influenciam o controle do fluxo, permitindo determinar se o salto deve ser tomado ou não.

```
// Comparações
4'b0101: ALUResult = ($signed(SrcA) < $signed(SrcB)) ? 1 : 0; // SLT/SLTI
4'b1100: ALUResult = (SrcA < SrcB) ? 1 : 0;  // SLTU/SLTIU
4'b1010: ALUResult = (SrcA == SrcB) ? 1 : 0; // BEQ
4'b1011: ALUResult = (SrcA != SrcB) ? 1 : 0; // BNE
4'b1101: ALUResult = ($signed(SrcA) < $signed(SrcB)) ? 1 : 0; // BLT
4'b1110: ALUResult = ($signed(SrcA) >= $signed(SrcB)) ? 1 : 0; // BGE
```

na BranchUnit.sv, implementei a lógica específica para as instruções do tipo B (branch), calculando corretamente o endereço efetivo do salto com o imediato deslocado e gerando o sinal de seleção do PC (PcSel) para controlar o desvio condicional do fluxo de execução.

```
// Expande PC curto para 32 bits (com zeros à esquerda)
logic [31:0] PC_Full;
assign PC_Full = { {(32-PC_W){1'b0}}, Cur_PC };

// PC + 4 (próximo endereço sequencial)
assign PC_Four = PC_Full + 32'd4;

// Immediate deslocado à esquerda por 1 (multiplica por 2 para byte offset)
logic signed [31:0] Imm_shifted;
assign Imm_shifted = Imm <<< 1;

// Cálculo do endereço do salto para branch e jal
assign PC_Imm = PC_Full + Imm_shifted;

// Condição para branch: deve estar ativo o sinal Branch e ALU sinalizar condição verdadeira (flag == 1)
logic Branch_Taken;
assign Branch_Taken = Branch && (AluResult == 32'd1);
```

IMPLEMENTAÇÃO DO LOAD E STORE

As instruções I-TYPE (LOAD) são usadas para carregar dados da memória para um registrador. Exemplos incluem lb, lh, lw, lbu, lhu.

As instruções S-TYPE (STORE) são usadas para armazenar dados de um registrador na memória. Exemplos incluem sb, sh, sw.

No controller.sv, defini o opcode do I-TYPE LOAD como 7'b0000011 e do S-TYPE STORE como 7'b0100011, ativando os sinais de controle específicos para essas instruções no pipeline, tais como MemRead para LOAD, MemWrite para STORE, além de RegWrite, ALUSrc, MemtoReg e ALUOp para controle adequado da execução.

```
// I-TYPE (LOAD): Instruções de carregamento de memória para registradores.
// Exemplos: lb, lh, lw, lbu, lhu.
localparam I_TYPE_LOAD = 7'b0000011;

// S-TYPE: Instruções de armazenamento de registradores na memória.
// Exemplos: sb, sh, sw.
localparam S_TYPE      = 7'b0100011;
```

No alucontroller.sv, para instruções de LOAD e STORE (I-TYPE LOAD e S-TYPE), a operação da ALU é sempre uma soma, pois o endereço efetivo é calculado somando o registrador base com o imediato de deslocamento. Por isso, mapeei o padrão correspondente para a operação ADD da ALU.

```
// Operações de Load/Store (soma de endereço)
{2'b00, 7'b???????, 3'b???}: Operation = 4'b0010; // ADD para LW/SW
```

No módulo datamemory.sv, implementei suporte às instruções LOAD com extensão de sinal para LB e LH, e extensão zero para LBU e LHU, usando os bits `a[1:0]` para selecionar o deslocamento dentro da palavra.

```
// --- LEITURA ---
if (MemRead) begin
  case (Funct3)
    3'b010: rd = Dataout; // LW

    3'b000: begin // LB
      case (byte_offset)
        2'b00: rd = $signed(Dataout[7:0]);
        2'b01: rd = $signed(Dataout[15:8]);
        2'b10: rd = $signed(Dataout[23:16]);
        2'b11: rd = $signed(Dataout[31:24]);
      endcase
    end

    3'b001: begin // LH
      case (byte_offset[1])
        1'b0: rd =  $signed(Dataout[15:0]);
        1'b1: rd = $signed(Dataout[31:16]);
      endcase
    end

    3'b100: begin // LBU
      case (byte_offset)
        2'b00: rd = {24'b0, Dataout[7:0]};
        2'b01: rd = {24'b0, Dataout[15:8]};
        2'b10: rd = {24'b0, Dataout[23:16]};
        2'b11: rd = {24'b0, Dataout[31:24]};
      endcase
    end

    default: rd = Dataout; // fallback
  endcase

end
```

Para STORE, as instruções SB, SH e SW têm controle preciso dos bytes escritos, ativando os bits de escrita (`Wr`) conforme o alinhamento, e replicando os dados conforme o tamanho. O endereço de leitura usa o valor completo, enquanto o de escrita é alinhado a 4 bytes para garantir acesso correto na memória de 32 bits.

```
// --- ESCRITA ---
else if (MemWrite) begin
  case (Funct3)
    3'b010: begin // SW
      Wr     = 4'b1111;
      Datain = wd;
    end

    3'b000: begin // SB
      case (byte_offset)
        2'b00: Wr = 4'b0001;
        2'b01: Wr = 4'b0010;
        2'b10: Wr = 4'b0100;
        2'b11: Wr = 4'b1000;
      endcase
      Datain = {4{wd[7:0]}}; // Replica o byte para toda a palavra (evita lixo)
    end

    3'b001: begin // SH
      case (byte_offset[1])
        1'b0: Wr = 4'b0011;
        1'b1: Wr = 4'b1100;
      endcase
      Datain = {2{wd[15:0]}};
    end

    default: begin
      Wr     = 4'b1111;
      Datain = wd;
    end
  endcase
end
```

imm_gem.sv:

No módulo `imm_Gen`, o imediato para instruções I-TYPE LOAD (opcode 7'b0000011) e S-TYPE STORE (opcode 7'b0100011) é extraído de forma distinta, conforme o formato RISC-V. Para I-TYPE LOAD, o imediato de 12 bits está em `inst_code[31:20]` e é sinal-estendido para 32 bits para preservar o sinal, usando

`Imm_out = {{20{inst_code[31]}}, inst_code[31:20]}`.

Para S-TYPE STORE, o imediato de 12 bits é formado pela concatenação dos bits altos `inst_code[31:25]` e baixos `inst_code[11:7]`, também sinal-estendido:

`Imm_out = {{20{inst_code[31]}}, inst_code[31:25], inst_code[11:7]}`.

Esse imediato é usado no estágio EX do pipeline para calcular o endereço efetivo de memória, somando-o ao registrador base, com a ALU realizando sempre uma operação ADD.

IMPLEMENTAÇÃO DO JAL E JALR

As instruções JAL e JALR realizam saltos e armazenam o endereço de retorno (PC + 4) no registrador de destino.

JAL usa um imediato somado ao PC para o salto, enquanto JALR soma um imediato a um registrador base para calcular o destino.

No controller.sv, implementei o opcode do J_TYPE como 7'b1101111 e do JALR_TYPE como 7'b1100111, e ativei os sinais de controle específicos no pipeline: jal, jalr, RegWrite, WBSel e PcSel. Além disso, adicionei saídas Jump e Jalr para indicar quando uma instrução jal ou jalr está ativa, permitindo que esses sinais sejam propagados pelos estágios do pipeline(no datapath) e utilizados nos multiplexadores e na lógica final de controle de salto.

```
// J-TYPE: Instrução de salto incondicional com armazenamento de retorno (jal).
// Exemplo: jal (jump and link).
localparam J_TYPE      = 7'b1101111;

// I-TYPE (JALR): Salto incondicional indireto via registrador com armazenamento de retorno.
// Exemplo: jalr (jump and link register).
localparam JALR_TYPE   = 7'b1100111;
```

No alucontroller.sv, para as instruções JAL e JALR (J-TYPE e I-TYPE), a operação da ALU também é uma soma, pois o endereço de salto é calculado somando o PC (ou um registrador base, no caso de JALR) com o imediato. Por isso, mapeei o padrão correspondente para a operação ADD da ALU.

Na BranchUnit.sv, implementei a lógica para calcular o endereço efetivo de salto (BrPC). Para instruções jalr, o endereço vem do resultado da ALU com o bit menos significativo forçado a zero para garantir alinhamento. Já para jal e instruções de branch, o endereço de salto é calculado como PC + imediato (PC_Imm). Essa lógica é feita com a atribuição condicional:

assign BrPC = (jalr) ? {AluResult[31:1], 1'b0} : PC_Imm;

No Datapath.sv, os sinais Jump e Jalr são propagados ao longo do pipeline para controlar o desvio do fluxo normal do programa. Com base nesses sinais, a BranchUnit calcula o novo endereço de salto — seja PC + imediato para jal ou registrador + imediato para jalr —, e o mux do PC seleciona entre PC + 4 e o endereço de salto (BrPC) por meio do sinal PcSel. Além disso, no estágio de Write Back (WB), o valor a ser escrito no registrador destino é selecionado por dois multiplexadores (mux_A e mux_B), controlados pelo sinal WBSel. O primeiro mux escolhe entre o resultado da ALU e o dado lido da memória, enquanto o segundo escolhe entre esse valor e PC + 4, que representa o endereço de retorno para as instruções jal e jalr. Assim, o PC + 4 é corretamente propagado e escrito no registrador destino quando a instrução é de salto com link, garantindo a integração dos sinais ao longo do pipeline.

```
// Primeiro mux: escolhe entre ALUResult e MemReadData
mux2 #(32) mux_A (
    .A(D.Alu_Result),
    .B(D.MemReadData),
    .Sel(D.WBSel[0]),
    .Out(muxA_out)
);

// Segundo mux: escolhe entre resultado anterior e PC+4
mux2 #(32) mux_B (
    .A(muxA_out),
    .B(D.Pc_Four),
    .Sel(D.WBSel[1]),
    .Out(WrmuxSrc)
);

// Atribuição final ao dado que será escrito no banco de registradores
assign WB_Data = WrmuxSrc;
```

IMPLEMENTAÇÃO DO HALT

Instrução `halt`: instrução especial que sinaliza o término da execução do programa, causando a parada controlada do processador, evitando que novas instruções sejam buscadas e garantindo que o pipeline finalize corretamente.

No controller.sv, criei um opcode específico para a instrução halt e implementei um sinal de saída para indicar quando essa instrução é detectada, permitindo a identificação clara do evento de parada no pipeline.

```
// HALT-TYPE: Instrução personalizada para parar a execução do processador.
// Não faz parte do padrão RISC-V; usada para simulação ou controle.
localparam HALT_TYPE   = 7'b1111111;
```

No módulo Datapath.sv, implementei a propagação do sinal de halt através dos estágios do pipeline. O halt é detectado nos estágios ID/EX ou EX/MEM e, ao chegar no estágio MEM/WB, o processador executa a parada definitiva. A partir desse momento, o estágio IF (Instruction Fetch) deixa de atualizar o PC, congelando-o para que o processador não busque mais instruções, garantindo assim o término ordenado da execução.

```
    // Detecta HALT no pipeline a partir do estágio ID/EX ou EX/MEM
    if ((B.Halt == 2'b01 || C.Halt == 2'b01) && !halt_detected)
      halt_detected <= 1'b1;

    // Finaliza só após HALT no estágio MEM/WB e sinal de finalização ainda não acionado
    if (D.Halt == 2'b01 && halt_detected && !halt_finalized) begin
      halt_finalized <= 1'b1;
      $display("[Datapath] Time=%0t | HALT finalizado no MEM/WB | PC=%0d", $time, PC);
      $finish;
    end
  end
end
```

CRIAÇÃO E DEFINIÇÃO DE SINAIS

A cada sinal novo implementado como entrada ou saída — como halt, jal e jump — é preciso:

- Declarar o sinal nos arquivos **risc_v.sv** (top-level do processador) e **regpack.sv** (estrutura que agrupa sinais entre estágios).
- Definir corretamente se o sinal é entrada, saída ou wire, além de garantir sua **propagação adequada entre os módulos**, respeitando o fluxo de dados e controle do pipeline.
- Atualizar os registradores de pipeline (IF_ID, ID_EX, EX_MEM, MEM_WB) para transportar esses sinais de controle pelos estágios.
- Assegurar que os sinais sejam utilizados corretamente em módulos como `BranchUnit`, `HazardDetection`, `ForwardingUnit`, e `WriteBack`.

No caso da instrução `halt`, o **sinal `Halt`** foi criado e propagado até o estágio `MEM/WB`, onde é verificado para realizar a parada definitiva (`$finish` na simulação). Além disso, o PC é congelado utilizando um controle adicional (`Reg_Stall || halt_detected`) para que **nenhuma nova instrução entre no pipeline após o halt**.