

# Implementação do Processador RISC-V em Verilog

Lucas Amaral Leme<sup>1</sup>, Matheus Martins Nunes<sup>1</sup>

<sup>1</sup> Universidade Federal de Ouro Preto (UFOP)  
Instituto de Ciências Exatas e Aplicadas (ICEA)  
Departamento de Computação e Sistemas (DECSI)  
João Monlevade – MG – Brazil

**Abstract.** *This document details the implementation of a 32-bit processor with the RISC-V architecture, structured in a five-stage pipeline (IF, ID, EX, MEM, WB). The project covers the creation of all datapath components, including instruction and data memories, register file, and the Arithmetic Logic Unit (ALU). A central focus of the work was handling pipeline hazards, with the implementation of a forwarding unit to resolve data hazards and a dedicated hazard detection unit.*

**Resumo.** *Este documento detalha a implementação de um processador de 32 bits com a arquitetura RISC-V, estruturado em um pipeline de cinco estágios (IF, ID, EX, MEM, WB). O projeto abrange a criação de todos os componentes do caminho de dados, incluindo memórias de instrução e dados, banco de registradores e a Unidade Lógica e Aritmética (ULA). Um foco central do trabalho foi o tratamento de conflitos (hazards) do pipeline, com a implementação de uma unidade de adiantamento (forwarding) para resolver hazards de dados e uma unidade de detecção de hazards dedicada.*

## 1. Introdução

Este trabalho apresenta o desenvolvimento de um processador RISC-V, projetado em Verilog com uma arquitetura pipeline de cinco estágios. O projeto foi realizado para a disciplina de Organização e Arquitetura de Computadores II, com o objetivo de aplicar na prática os conceitos de arquiteturas modernas.

Para o desenvolvimento, utilizamos o Visual Studio Code como editor de código. A compilação foi realizada com o Icarus Verilog (iverilog) e a visualização das formas de onda foi feita com o GTKWave.

Durante a implementação, focamos em construir as cinco etapas do pipeline (busca, decodificação, execução, memória e escrita) e em criar lógicas para resolver conflitos de dados e de controle, conhecidos como *hazards*. Este relatório detalha a metodologia usada, a estrutura do projeto e os resultados obtidos na simulação.

O código fonte completo do projeto, incluindo todos os módulos Verilog e arquivos de simulação, está disponível no seguinte repositório do GitHub: [Acessar Repositório do Projeto](#).

## 2. Metodologia

Nesta seção, detalhamos o processo de implementação dos módulos que compõem o processador RISC-V. Adotamos uma abordagem modular, inspirada na metodologia apresentada no livro, onde cada componente funcional do processador foi desenvolvido e testado individualmente em arquivos separados.

Inicialmente, realizamos o estudo da arquitetura RISC-V e do pipeline de cinco estágios, identificando os principais blocos necessários: Unidade de Controle, ALU, Banco de Registradores, Memória de Dados, Unidade de Busca de Instruções (IFU), MUXes e unidades de pipeline. Cada módulo foi implementado em Verilog, respeitando as interfaces e sinais definidos no projeto.

Após a implementação de cada módulo, desenvolvemos testbenches específicos para validar o funcionamento isolado de cada componente, garantindo que todos os sinais e operações estivessem corretos antes da integração. Só então os módulos foram conectados para formar o processador completo, permitindo a simulação do pipeline e a verificação do funcionamento conjunto.

Essa metodologia incremental facilitou a identificação e correção de erros, além de proporcionar maior clareza na organização do projeto e na documentação dos resultados.

### 2.1. Unidade de Controle

A unidade de controle, implementada no arquivo `control.v`, funciona como o cérebro do processador. Sua principal função é ler a instrução para entender qual operação deve ser executada (como uma soma, uma carga de dados ou um desvio). Para fazer isso, ela analisa o campo `opcode` da instrução e, com base nele, gera todos os sinais de controle que comandam as outras partes do processador, como a ULA, a memória e o banco de registradores.

A Tabela 1 resume os principais sinais gerados pela unidade de controle.

Tabela 1. Sinais gerados pela unidade de controle	
Sinal	Função
<code>alu_src_a, alu_src_b</code>	Seleção dos operandos da ALU (registrador, imediato, PC)
<code>mem_to_reg</code>	Seleção do dado para escrita no registrador (ALU ou memória)
<code>alu_control</code>	Operação a ser realizada pela ALU
<code>regwrite</code>	Habilita escrita no banco de registradores
<code>mem_read</code>	Habilita leitura da memória de dados
<code>mem_write</code>	Habilita escrita na memória de dados
<code>branch</code>	Indica instrução de desvio condicional

Essa abordagem garante flexibilidade e clareza na decodificação das instruções, facilitando a manutenção e expansão do processador.

### 2.2. Unidade Lógica e Aritmética (ALU)

A Unidade Lógica e Aritmética (ALU), implementada no arquivo `alu.v`, é o componente responsável por realizar as operações de cálculo do processador. Trata-se de um circuito puramente combinacional, implementado com um bloco `always @(*)`, que garante que as saídas sejam atualizadas sempre que qualquer uma das entradas mudar.

A ALU recebe duas entradas de 32 bits (`in1` e `in2`) e um sinal de controle de 4 bits (`alu_control`) que determina qual operação será executada. Suas saídas são

o resultado da operação (`alu_result`) e uma flag (`zero_flag`), que é ativada se o resultado for zero, sendo essencial para a lógica de desvios condicionais.

A Tabela 2 detalha as operações que a ALU pode realizar, com base no sinal de controle.

**Tabela 2. Operações da Unidade Lógica e Aritmética (ALU)**

<b>alu_control</b>	<b>Operação Realizada</b>
4'b0000	AND lógico ( $in1 \& in2$ )
4'b0001	OR lógico ( $in1   in2$ )
4'b0010	Soma ( $in1 + in2$ )
4'b0110	Subtração ( $in1 - in2$ )
4'b0111	XOR ( $in1 \wedge in2$ )
4'b1000	Shift Right Logical (SRL)
4'b1001	Shift Left Logical (SLL)
4'b1010	Shift Right Arithmetic (SRA)
4'b1100	Set Less Than (SLT, com sinal)
4'b1101	Set Less Than Unsigned (SLTU, sem sinal)

A Figura 1 mostra a implementação completa do módulo da ALU em Verilog.

```
1 module ALU (  
2     input [31:0] in1, in2,  
3     input [3:0] alu_control,  
4     output reg [31:0] alu_result,  
5     output reg zero_flag  
6 );  
7     always @(*) begin  
8         case(alu_control)  
9             // Operacoes logicas  
10            4'b0000: alu_result = in1 & in2;           // AND  
11            4'b0001: alu_result = in1 | in2;           // OR  
12            4'b0010: alu_result = in1 + in2;           // ADD  
13            4'b0110: alu_result = in1 - in2;           // SUB  
14            4'b0111: alu_result = in1 ^ in2;           // XOR  
15  
16            // Operacoes de deslocamento (shift)  
17            4'b1000: alu_result = in1 >> in2[4:0]; // SRL  
18            4'b1001: alu_result = in1 << in2[4:0]; // SLL  
19            4'b1010: alu_result = $signed(in1) >>> in2[4:0]; // SRA  
20  
21            // Operacoes de comparacao  
22            4'b1100: alu_result = ($signed(in1) < $signed(in2)) ? 32'b1  
23            : 32'b0; // SLT  
24            4'b1101: alu_result = (in1 < in2) ? 32'b1 : 32'b0; // SLTU  
25  
26            default: alu_result = 32'b0;  
27        endcase  
28  
29        // Flag zero e ativada quando o resultado e zero  
30        zero_flag = (alu_result == 32'b0);  
31    end  
32 endmodule
```

**Listing 1. Implementação da Unidade Lógica e Aritmética (ALU)**

## 2.3. Banco de Registradores

O banco de registradores foi implementado no módulo `reg_file.v` e é composto por 32 registradores de 32 bits. Ele permite a leitura simultânea de dois registradores e a escrita síncrona em um registrador por ciclo de clock.

A escrita ocorre na borda de subida do clock, desde que o sinal de controle `regwrite` esteja ativo e o registrador de destino não seja o registrador zero (`x0`), que por hardware deve permanecer sempre com valor zero. Na inicialização da simulação, todos os 32 registradores são zerados para garantir um estado inicial previsível.

### Entradas e Saídas do Módulo:

- **Entradas:**

- `clock, reset`: Sinais de clock e reset.
- `regwrite`: Habilita a operação de escrita.
- `read_reg_num1[4:0], read_reg_num2[4:0]`: Endereços dos dois registradores a serem lidos.
- `write_reg[4:0]`: Endereço do registrador a ser escrito.
- `write_data[31:0]`: Dado de 32 bits a ser escrito.

- **Saídas:**

- `read_data1[31:0], read_data2[31:0]`: Dados de 32 bits lidos dos registradores.

**Destaque do Código - Forwarding Interno:** Uma característica importante deste módulo é a implementação de um "forwarding interno" para evitar hazards de dados dentro do mesmo ciclo. A lógica de leitura, implementada com o comando `assign`, verifica continuamente se o endereço de um registrador sendo lido (`read_reg_num1` ou `read_reg_num2`) é o mesmo que está sendo escrito (`write_reg`) e se a escrita está habilitada (`regwrite`). Se essa condição for verdadeira, o módulo não lê o valor antigo da memória de registradores, mas sim "adianta" o novo valor (`write_data`) diretamente para a saída. Isso garante que a instrução seguinte receba o dado mais atualizado sem a necessidade de um stall.

## 2.4. Memórias de Instrução e Dados

O processador utiliza duas memórias distintas: a Memória de Instruções, que armazena o programa a ser executado, e a Memória de Dados, que guarda os valores manipulados durante a execução. Ambas foram implementadas como arrays de 1024 posições de 32 bits, totalizando 4KB cada.

### 2.4.1. Memória de Instrução

A Memória de Instrução, implementada no módulo `inst_mem.v`, funciona como uma memória ROM (Read-Only Memory), sendo apenas para leitura. Sua principal função é receber um endereço de 32 bits do Program Counter (PC) e retornar a instrução de 32 bits correspondente de forma assíncrona.

Um bloco `initial` é utilizado para pré-carregar a memória com um programa de teste no início da simulação. O módulo também possui uma lógica de proteção que retorna

uma instrução NOP caso o endereço do PC não esteja alinhado em 4 bytes ou esteja fora dos limites da memória. As entradas e saídas do módulo, bem como o programa de teste carregado, são detalhados a seguir.

#### **Entradas e Saídas do Módulo:**

- **Entradas:** PC[31:0], reset
- **Saídas:** Instruction\_Code[31:0]

#### **Programa de Teste Carregado na Memória:**

```
1. addi x12, x0, 50
2. addi x13, x0, 15
3. sub x14, x12, x13
4. or x15, x14, x12
5. andi x16, x13, 31
6. sh x15, 0(x0)
7. lh x17, 0(x0)
8. srli x18, x16, 2
9. beq x12, x12, +8
10. addi x19, x0, 255 (Instrução a ser descartada pelo flush)
11. nop (Alvo do desvio)
12. jal x0, 0 (Fim do programa)
```

### **2.4.2. Memória de Dados**

Diferente da anterior, a Memória de Dados (`data_memory.v`) é uma RAM, suportando tanto leitura quanto escrita. A leitura dos dados é assíncrona, ocorrendo sempre que o sinal `mem_read` está ativo. A escrita, por sua vez, é síncrona e acontece na borda de subida do clock, controlada pelo sinal `mem_write`.

O módulo foi projetado para manipular dados de diferentes tamanhos, com suporte explícito para *halfword* (16 bits), necessário para as instruções `lh` e `sh`. Para a leitura, a lógica de extensão de sinal é aplicada corretamente com base no bit mais significativo do dado lido. As características do módulo são resumidas abaixo.

#### **Entradas e Saídas do Módulo:**

- **Entradas:**
  - `clk, reset`: Sinais de clock e reset.
  - `address[31:0]`: Endereço para leitura ou escrita.
  - `write_data[31:0]`: Dado a ser escrito na memória.
  - `mem_read, mem_write`: Sinais de controle para habilitar leitura/escrita.
  - `size[1:0]`: Define o tamanho do dado (byte, halfword, word).
  - `unsigned_load`: Define o tipo de extensão de sinal na leitura.
- **Saídas:** `read_data[31:0]` (Dado lido da memória).

#### **Estado Inicial da Memória:**

- A memória é inicializada com valores de teste para a simulação. A posição de endereço 0 começa com o valor 32' h001D0000, e as demais posições são preenchidas com zero.

## 2.5. Unidade de Busca de Instrução (IFU)

A Unidade de Busca de Instrução, implementada no módulo `ifu.v`, é o primeiro estágio do pipeline e sua principal responsabilidade é buscar a instrução correta da memória a cada ciclo de clock. Para isso, ela gerencia o Program Counter (PC), o registrador que armazena o endereço da instrução a ser executada.

A lógica da unidade determina o endereço da próxima instrução (`next_pc`) com base na situação atual do processador. Em uma execução normal, o próximo endereço é simplesmente  $PC + 4$ . No entanto, se um desvio condicional for tomado (`branch_taken`) ou uma instrução de salto for executada (`jump_taken`), o `next_pc` recebe o endereço de destino correspondente.

O PC só é atualizado na borda de subida do clock se não houver um sinal de `stall` ativo, vindo da unidade de detecção de hazards. Isso é crucial para paralisar o pipeline quando necessário. O módulo também instancia a memória de instruções (`inst_mem`) para efetivamente ler a instrução no endereço apontado pelo PC.

### Entradas e Saídas do Módulo:

- **Entradas:**

- `clk, reset`: Sinais de clock e reset.
- `stall`: Sinal que paralisa a atualização do PC.
- `branch_taken, jump_taken`: Sinais que indicam a ocorrência de desvios ou saltos.
- `branch_target, jump_target`: Endereços de destino para desvios e saltos.

- **Saídas:**

- `Instruction_Code`: A instrução de 32 bits lida da memória.
- `PC`: O endereço da instrução atual.
- `PC_plus_4`: O valor de  $PC+4$ , disponibilizado para uso em instruções como JAL.

## 2.6. Gerador de Imediatos

O Gerador de Imediatos, implementado no módulo `immediate_generator.v`, é um componente combinacional crucial para o estágio de Decodificação. Sua função é extrair o valor imediato codificado dentro da palavra de instrução de 32 bits e formatá-lo corretamente para ser usado em operações posteriores, principalmente na ALU.

Como o formato e a posição dos bits do imediato variam entre os diferentes tipos de instrução do RISC-V, o módulo utiliza o `opcode` da instrução para determinar como o valor deve ser extraído e estendido. O módulo dá suporte aos seguintes formatos:

- **Tipo-I (Load, Op-Imm)**: Extrai o imediato de 12 bits e realiza a extensão de sinal para 32 bits.
- **Tipo-S (Store)**: Reconstrói o imediato de 12 bits a partir de dois campos separados na instrução e realiza a extensão de sinal.
- **Tipo-B (Branch)**: Reconstrói o imediato de 13 bits (com o bit menos significativo sempre zero) a partir de múltiplos campos da instrução e realiza a extensão de sinal.

A extensão de sinal é um passo fundamental para garantir que o valor do imediato, que pode ser negativo, seja representado corretamente em 32 bits antes de ser enviado para a ALU.

#### Entradas e Saídas do Módulo:

- **Entradas:** `instruction[31:0]` (A palavra de instrução completa).
- **Saídas:** `imm_out[31:0]` (O valor imediato de 32 bits com sinal estendido).

## 2.7. Multiplexadores (MUXes)

Os multiplexadores são componentes combinacionais fundamentais que direcionam o fluxo de dados através do processador. No projeto, foram implementados três MUXes principais para selecionar as entradas corretas para a ALU e para o estágio de Write Back.

- **mux\_alu\_a.v:** Seleciona a primeira entrada para a ALU. As opções são o dado vindo do banco de registradores (para operações normais), o valor do PC (para o cálculo de endereço de desvio) ou o valor zero, dependendo do sinal de controle `alu_src_a`.
- **mux\_alu\_src.v:** Responsável por selecionar a segunda entrada da ALU. Ele escolhe entre o dado do segundo registrador (para instruções do Tipo-R) ou o valor imediato estendido (para instruções do Tipo-I e S), com base no sinal `alu_src_b`.
- **mux\_writeback.v:** Localizado no final do pipeline, este MUX determina qual valor será escrito de volta no banco de registradores. Ele seleciona entre o resultado da ALU ou o dado lido da Memória de Dados, controlado pelo sinal `mem_to_reg`.

## 2.8. Registradores de Pipeline

Os registradores de pipeline são a espinha dorsal da arquitetura, atuando como barreiras síncronas que isolam os cinco estágios e permitem a sobreposição da execução das instruções. A cada ciclo de clock, eles capturam uma fotografia do estado de uma instrução e a propagam para o estágio seguinte. Essa separação é o que viabiliza o paralelismo e aumenta a vazão (throughput) do processador. Todos os registradores possuem lógica para `reset` e `flush`, zerando os sinais de controle para neutralizar uma instrução e evitar operações incorretas.

### 2.8.1. Registrador IF/ID (`if_id_register.v`)

Este registrador conecta os estágios de Busca e Decodificação. Sua função é armazenar a instrução e o PC para o próximo estágio. Ele também pode ser paralisado ou limpo para controle de hazards.

- **Entradas:**
  - `clk`, `reset`: Sinais de clock e reset.
  - `write_enable`: Habilita a escrita, permitindo a implementação de *stalls*.
  - `flush`: Limpa o registrador, inserindo um NOP.
  - `instruction_in[31:0]`: A instrução vinda da memória de instruções.



- pc\_in[31:0]: O endereço da instrução.
- **Saídas:**
  - instruction\_out[31:0]: A instrução passada para o estágio ID.
  - pc\_out[31:0]: O PC passado para o estágio ID.

### 2.8.2. Registrador ID/EX (id\_ex\_register.v)

Este registrador transporta a instrução decodificada e todos os seus dados e sinais de controle associados para o estágio de Execução. É o maior dos registradores, pois carrega o "plano de execução" completo da instrução.

- **Entradas:**
  - clk, reset, flush: Sinais de clock, reset e limpeza.
  - Sinais de Controle: regwrite\_in, mem\_read\_in, mem\_write\_in, branch\_in, alu\_src\_a\_in[1:0], alu\_src\_b\_in[1:0], mem\_to\_reg\_in[1:0], alu\_control\_in[3:0].
  - Dados: read\_data1\_in[31:0], read\_data2\_in[31:0], pc\_in[31:0], immediate\_in[31:0].
  - Endereços de Registradores: rs1\_in[4:0], rs2\_in[4:0], rd\_in[4:0].
- **Saídas:** Todas as entradas correspondentes com o sufixo \_out.

### 2.8.3. Registrador EX/MEM (ex\_mem\_register.v)

Responsável por armazenar os resultados da fase de cálculo (EX) para serem utilizados na fase de acesso à memória (MEM). Ele transporta o endereço de memória, os dados para escrita e os sinais de controle necessários.

- **Entradas:**
  - clk, reset, flush: Sinais de clock, reset e limpeza.
  - Sinais de Controle: regwrite\_in, mem\_read\_in, mem\_write\_in, mem\_to\_reg\_in[1:0], branch\_in.
  - Dados e Flags: alu\_result\_in[31:0], branch\_target\_in[31:0], write\_data\_in[31:0], rd\_in[4:0], zero\_flag\_in.
- **Saídas:** Todas as entradas correspondentes com o sufixo \_out.

### 2.8.4. Registrador MEM/WB (mem\_wb\_register.v)

Último registrador do pipeline, ele transporta o resultado final de uma instrução (seja da ALU ou da memória) para o estágio de Write Back, onde o banco de registradores será finalmente atualizado.

- **Entradas:**
  - clk, reset, flush: Sinais de clock, reset e limpeza.
  - Sinais de Controle: regwrite\_in, mem\_to\_reg\_in[1:0].
  - Dados: mem\_read\_data\_in[31:0], alu\_result\_in[31:0], pc\_plus\_4\_in[31:0], rd\_in[4:0].
- **Saídas:** Todas as entradas correspondentes com o sufixo \_out.

## 2.9. Unidade de Forwarding

A Unidade de Forwarding (adiantamento), implementada no módulo `forwarding_unit.v`, é uma otimização crucial para o desempenho do pipeline. Sua função é resolver hazards de dados do tipo RAW (Read-After-Write), que ocorrem quando uma instrução tenta ler um registrador antes que uma instrução anterior tenha terminado de escrever nele.

Em vez de paralisar o pipeline (stall), a unidade de forwarding detecta essa dependência e "adianta" o resultado correto diretamente da saída dos estágios EX ou MEM para a entrada da ALU, evitando a espera. A lógica compara os registradores fonte da instrução no estágio EX (`rs1_ex`, `rs2_ex`) com os registradores de destino das instruções nos estágios MEM (`rd_mem`) e WB (`rd_wb`). A unidade prioriza o dado mais recente, adiando do estágio MEM em vez do WB se houver conflito.

- **Entradas:**

- `rs1_ex[4:0]`, `rs2_ex[4:0]`: Endereços dos registradores fonte no estágio EX.
- `rd_mem[4:0]`, `regwrite_mem`: Endereço de destino e sinal de escrita do estágio MEM.
- `rd_wb[4:0]`, `regwrite_wb`: Endereço de destino e sinal de escrita do estágio WB.

- **Saídas:**

- `forward_a[1:0]`, `forward_b[1:0]`: Sinais de controle para os MUXes da ALU, indicando a origem do dado.

## 2.10. Unidade de Detecção de Hazards

A Unidade de Detecção de Hazards (`hazard_detection_unit.v`) lida com situações que o forwarding não consegue resolver. Ela é responsável por paralisar (stall) ou limpar (flush) o pipeline para garantir a execução correta. Por padrão, ela mantém os sinais de controle em um estado que permite o fluxo normal do pipeline.

- **Hazard de Dependência de Carga (Load-Use):** Ocorre quando uma instrução tenta usar o resultado de uma instrução de *load* que está no estágio EX. Como o dado só estará disponível após o estágio MEM, o forwarding não é suficiente. Nesse caso, a unidade detecta a dependência (`mem_read_ex` ativo e `rd_ex` igual a `rs1_id` ou `rs2_id`) e insere uma bolha no pipeline. Isso é feito paralisando o PC e o registrador IF/ID por um ciclo (`pc_write_enable`  $\leq 0$ , `if_id_write_enable`  $\leq 0$ ), enquanto o registrador ID/EX é limpo (`id_ex_flush`  $\leq 1$ ).
- **Hazard de Controle:** Ocorre quando uma instrução de desvio (*branch*) é tomada. As instruções que foram buscadas sequencialmente estão incorretas e precisam ser descartadas. Ao receber o sinal `branch_taken`, a unidade gera sinais de flush para limpar os registradores IF/ID, ID/EX e EX/MEM, anulando as instruções indesejadas.
- **Entradas:**
  - `rs1_id[4:0]`, `rs2_id[4:0]`: Endereços dos registradores fonte no estágio ID.

- `mem_read_ex, rd_ex[4:0]`: Sinal de leitura de memória e endereço de destino no estágio EX.
- `branch_taken`: Sinaliza que um desvio foi tomado no estágio MEM.
- **Saídas:**
  - `pc_write_enable, if_id_write_enable`: Sinais para paralisar os estágios iniciais. [?]
  - `if_id_flush, id_ex_flush, ex_mem_flush`: Sinais para limpar os respectivos registradores de pipeline.

## 2.11. Implementação da CPU

A implementação final da CPU, contida no módulo de topo `riscv_processor.v`, representa a integração de todos os componentes descritos anteriormente em uma arquitetura pipeline de cinco estágios funcional. Este módulo é responsável por instanciar cada unidade e registrador de pipeline, e principalmente, por conectar os fios (wires) que transportam os dados e os sinais de controle através dos estágios.

A beleza da arquitetura pipeline reside na forma como os dados fluem para frente através dos registradores de pipeline, enquanto os sinais de controle de hazard (como o resultado de um desvio ou a necessidade de adiantamento) fluem para trás, alterando o comportamento dos estágios iniciais. A seguir, detalhamos a integração em cada um dos cinco estágios.

### 2.11.1. Estágio 1: Busca de Instrução (IF)

O estágio IF é centrado na IFU, que calcula o próximo PC. O ponto mais crítico aqui é a realimentação (feedback) do sinal `branch_taken_mem`, que vem do final do estágio MEM. Se um desvio é tomado, a IFU descarta o PC sequencial e carrega o endereço de desvio. O fluxo é controlado pela Unidade de Detecção de Hazards: o sinal `stall` congela a IFU durante um hazard de load-use, e o sinal `if_id_flush` limpa o registrador `if_id_register` após um desvio.

### 2.11.2. Estágio 2: Decodificação (ID)

No estágio ID, a instrução vinda do registrador IF/ID é decomposta. Seus campos são enviados para a control unit, para o `reg_file` (para leitura dos operandos `rs1` e `rs2`) e para o `immediate_generator`. É neste estágio que a `hazard_detection_unit` atua, comparando os registradores lidos (`rs1_id`, `rs2_id`) com o registrador de destino da instrução no estágio EX (`rd_ex`) para detectar a necessidade de um stall. Um ponto fundamental da arquitetura é visível aqui: o `reg_file` é lido neste estágio, mas a escrita nele acontece apenas no estágio WB, utilizando dados do final do pipeline (`rd_wb`, `write_data_wb`).

### 2.11.3. Estágio 3: Execução (EX)

O estágio EX é o coração computacional. A `forwarding_unit` opera aqui, recebendo os endereços dos registradores dos estágios EX, MEM e WB para decidir se os dados

para a ALU devem vir do registrador ID/EX ou ser adiantados dos estágios posteriores. Os sinais `forward_a` e `forward_b` controlam um conjunto de MUXes (implementados com lógica ‘assign’) que selecionam a fonte correta para os operandos. Após a seleção, os dados passam pelos `mux_alu_a` e `mux_alu_src` para finalmente chegarem à ALU, que realiza a operação e calcula o endereço de desvio.

#### 2.11.4. Estágio 4: Acesso à Memória (MEM)

No estágio MEM, o resultado da ALU (`alu_result_mem`) é usado como endereço para a `data.memory`. Os sinais de controle `mem_read_mem` e `mem_write_mem`, que viajam pelo pipeline, determinam se a operação é de leitura ou escrita. A lógica mais crítica deste estágio é a decisão final do desvio: a linha `assign branch_taken_mem = branch_mem & zero_flag_mem;` combina o sinal de controle (que indica que a instrução é um branch) com o resultado da comparação da ALU (a `zero_flag`). O resultado, `branch_taken_mem`, é o sinal que retorna ao estágio IF para controlar o fluxo do programa.

#### 2.11.5. Estágio 5: Escrita (WB)

O último estágio, Write Back, finaliza a execução da instrução. O `mux_writeback` seleciona o dado final a ser escrito, escolhendo entre o resultado da ALU (`alu_result_wb`) e o dado lido da memória (`mem_read_data_wb`) com base no sinal `mem_to_reg_wb`. O valor selecionado, `write_data_wb`, juntamente com o endereço do registrador de destino, `rd_wb`, e o sinal de habilitação `regwrite_wb`, são enviados de volta ao `reg_file` no estágio ID, fechando o ciclo e atualizando o estado do processador.

### 2.12. Validação Modular Pré-Integração

Antes da integração final do processador, cada módulo foi rigorosamente testado de forma isolada para garantir seu funcionamento correto. Essa abordagem modular permitiu identificar e corrigir erros de forma localizada, reduzindo significativamente a complexidade da depuração no sistema completo. Para cada componente, desenvolvemos testbenches específicos que simulavam cenários críticos, incluindo casos extremos e condições de borda.

## 3. Resultados e Discussão

Nesta seção, são apresentados e analisados os resultados obtidos a partir da simulação do processador RISC-V implementado. O objetivo é validar o funcionamento do caminho de dados e da unidade de controle por meio da execução de um programa de teste específico, que abrange todo o conjunto de instruções designado para o grupo. A análise foca na verificação do estado final dos registradores e na correta execução de operações de memória e desvios condicionais.

### 3.1. Programa de Teste e Configuração do Testbench

Para a validação do processador, foi elaborado um programa de teste contendo exclusivamente as sete instruções atribuídas ao grupo: `lh`, `sh`, `sub`, `or`, `andi`, `srl` e `beq`.

As instruções foram organizadas de forma a exercitar diferentes caminhos de dados, operações lógicas e aritméticas, bem como acessos à memória e desvios condicionais.

O carregamento dessas instruções na memória de instruções foi feito por meio de inicialização direta no bloco `initial`, assegurando valores distintos nos registradores de origem para facilitar a verificação posterior. Além disso, o código foi estruturado de forma a incluir um desvio condicional (`beq`) seguido de instruções que não deveriam ser executadas, permitindo observar o mecanismo de *flush*.

Ao término, o estado dos registradores e da memória de dados é exibido, juntamente com verificações específicas para instruções de acesso à memória, confirmando a correta execução de `sh` e `lh`.

### 3.2. Resultados Obtidos

A simulação do processador RISC-V com o programa de teste proposto confirmou a execução correta das sete instruções atribuídas ao grupo (`lh`, `sh`, `sub`, `or`, `andi`, `srl` e `beq`).

Ao término da execução, o estado final dos registradores e da memória de dados indicou que todas as operações lógicas, aritméticas e de acesso à memória foram realizadas conforme esperado. Destaca-se a correta escrita e leitura de `halfwords` na memória, bem como o funcionamento do desvio condicional `beq`, que resultou no salto previsto e na não execução das instruções subsequentes.

Os registradores finais comprovam a execução correta:

#### Carga Inicial

- `x12 = 0x32 (50)` e `x13 = 0x0F (15)`: Resultado das instruções `addi` que carregam valores imediatos

#### Operações Aritméticas

- `x14 = 0x23 (35)`: Saída da subtração `50 - 15 (sub x14, x12, x13)`

#### Operações de Deslocamento

- `x18 = 0x03 (3)`: Resultado de `srl x18, x16, 2`  
(15 em hexadecimal: `0x0F → 15 >> 2 = 3`)

#### Lógica e Memória

- `x15 = 0x33 (51)`: Resultado do OR entre `x14 (35)` e `x12 (50)`
- `x17 = 0x33 (51)`: Mostra que o valor de `x15` foi corretamente armazenado (`sh`) e lido (`lh`) da memória

#### Controle de Fluxo

- `x19`: Mantém valor inicial (não alterado), provando que o `addi x19` foi descartado após o `beq` (desvio)

A Figura 1 apresenta a captura de tela do *testbench* com o estado final dos registradores e da memória de dados após a simulação, evidenciando o funcionamento correto do caminho de dados e da unidade de controle.

```
=====
ESTADO FINAL DA MEMORIA DE DADOS
=====
memoria[ 0]: 0x00000033
memoria[ 1]: 0x00000000
memoria[ 2]: 0x00000000
memoria[ 3]: 0x00000000
memoria[ 4]: 0x00000000
memoria[ 5]: 0x00000000
memoria[ 6]: 0x00000000
memoria[ 7]: 0x00000000

=====
ESTADO FINAL DE TODOS OS REGISTRADORES
=====
register[ 0]:      0
register[ 1]:      0
register[ 2]:      0
register[ 3]:      0
register[ 4]:      0
register[ 5]:      0
register[ 6]:      0
register[ 7]:      0
register[ 8]:      0
register[ 9]:      0
register[10]:      0
register[11]:      0
register[12]:     50
register[13]:     15
register[14]:     35
register[15]:     51
register[16]:     15
register[17]:     51
register[18]:      3
register[19]:      0
register[20]:      0
register[21]:      0
register[22]:      0
register[23]:      0
register[24]:      0
register[25]:      0
register[26]:      0
register[27]:      0
register[28]:      0
register[29]:      0
register[30]:      0
register[31]:      0
```

**Figura 1. Estado final dos registradores e da memória de dados após execução do programa de teste.**

### 3.3. Análise da Execução de Instrução Aritmética (sub)

O primeiro cenário de teste verifica a execução de uma instrução aritmética básica do Tipo-R, a `sub x14, x12, x13`. O objetivo é observar o caminho de dados desde a decodificação até a escrita do resultado no banco de registradores. A Figura 2 exibe a forma de onda capturada durante a execução desta instrução.

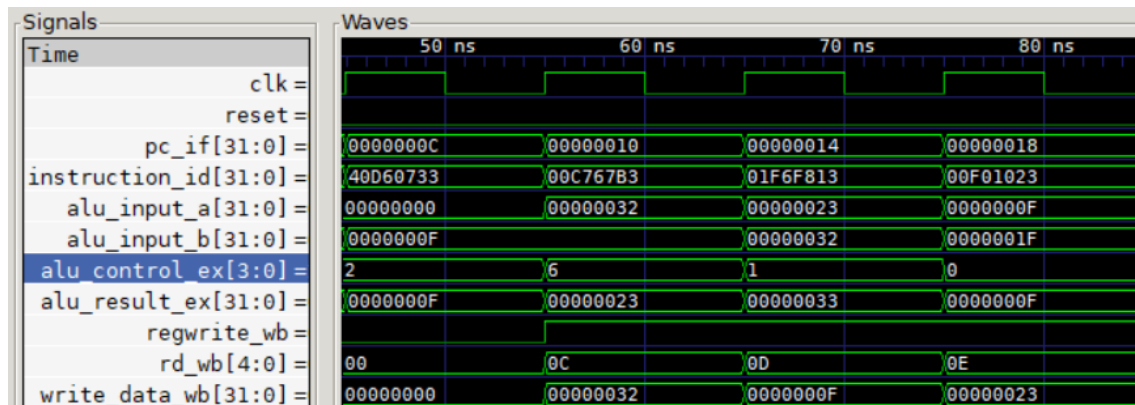


Figura 2. Forma de onda da execução da instrução `sub x14, x12, x13`.

A análise da Figura 2 revela o seguinte comportamento, ciclo a ciclo:

- **Em 50ns:** A instrução `sub` (código 0x40D60733) está no estágio de Decodificação (ID), como visto no sinal `instruction_id`.
- **Em 60ns:** A instrução avança para o estágio de Execução (EX). Os sinais confirmam que a operação está correta:
  - `alu_input_a` recebe o valor de `x12`, que é 0x32 (50).
  - `alu_input_b` recebe o valor de `x13`, que é 0x0F (15).
  - `alu_control_ex` mostra o valor 6 (binário 0110), que é o código correto para a operação de subtração.
  - Como resultado, `alu_result_ex` exibe 0x23 (35), confirmando que a subtração (50 - 15) foi executada com sucesso.
- **Em 80ns:** Após passar pelo estágio de Memória, a instrução chega ao estágio de Write Back (WB). Os sinais demonstram a conclusão da instrução:
  - `regwrite_wb` está em nível alto ('1'), habilitando a escrita no banco de registradores.
  - `rd_wb` indica o registrador de destino, que é 0E (`x14`).
  - `write_data_wb` contém o resultado final, 0x23 (35), que será escrito em `x14`.

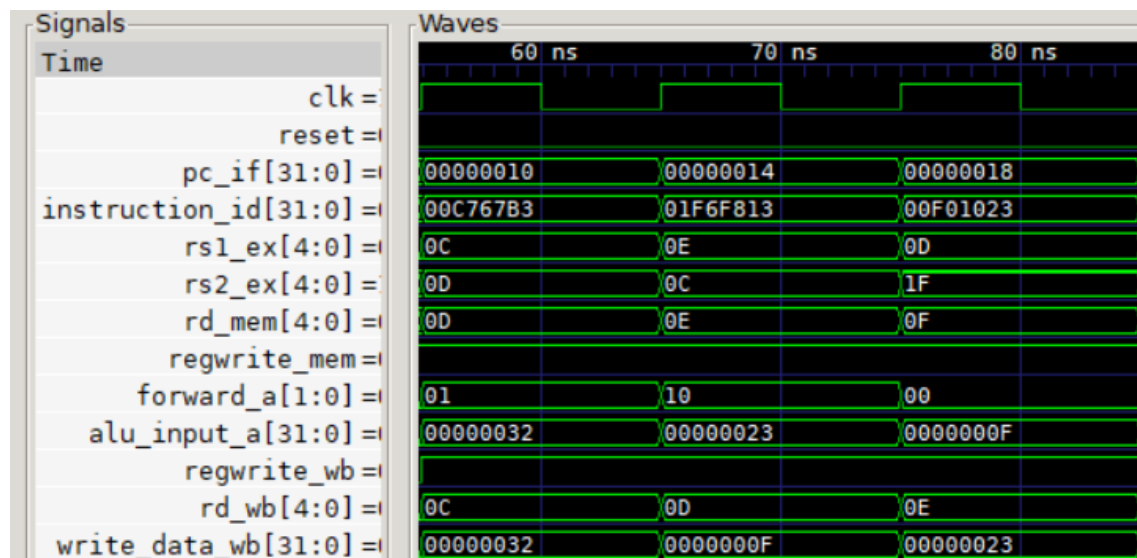
A análise da forma de onda valida com sucesso a execução de uma instrução do Tipo-R, confirmando que os estágios do pipeline, a unidade de controle e a ALU estão operando corretamente em conjunto.

### 3.4. Análise de Hazard de Dados com Forwarding

Este cenário testa uma das otimizações mais importantes do pipeline: a capacidade da Unidade de Forwarding de resolver um hazard de dados do tipo RAW (Read-After-Write) sem a necessidade de paralisar o processador. Analisamos a sequência de instruções `sub`

x14, x12, x13, seguida imediatamente por or x15, x14, x12. A instrução or depende do resultado da sub, que ainda não foi escrito no banco de registradores.

A Figura 3 captura o momento exato em que a Unidade de Forwarding entra em ação.



**Figura 3.** Forma de onda demonstrando o forwarding do resultado de sub para or.

A análise da forma de onda, com foco no ciclo de clock entre 60ns e 70ns, revela:

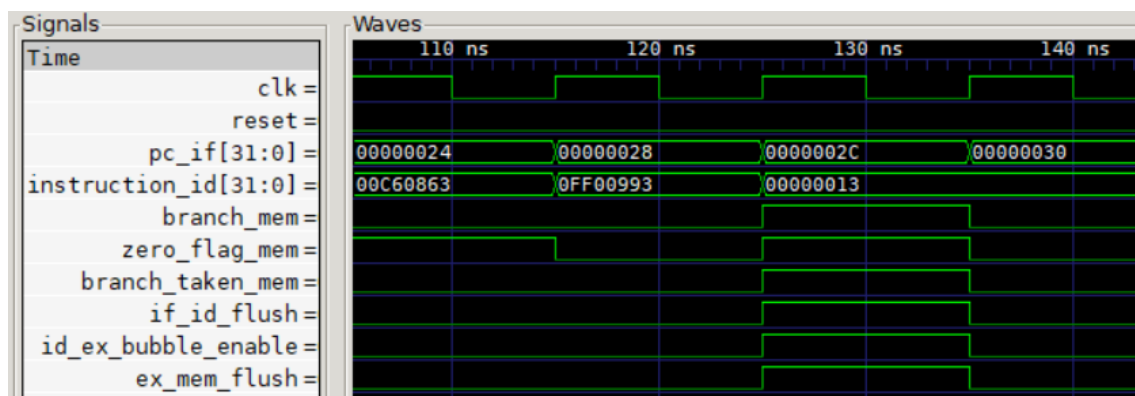
- **Condição de Hazard:** A instrução or está no estágio de Execução (EX) e precisa ler o registrador x14 (o sinal rs1\_ex é 0E). Ao mesmo tempo, a instrução anterior, sub, está no estágio de Memória (MEM) e seu registrador de destino é também o x14 (rd\_mem é 0E e regwrite\_mem está ativo).
- **Ação do Forwarding:** A Unidade de Forwarding detecta essa dependência e ativa o sinal de controle forward\_a, mudando seu valor para 10. Este sinal instrui o MUX de entrada da ALU a ignorar o dado vindo do banco de registradores e, em vez disso, capturar o resultado que está saindo do estágio MEM.
- **Resultado:** O sinal alu\_input\_a mostra o valor 0x23 (35), que é o resultado da instrução sub. Isso confirma que o dado foi "adiantado" (forwarded) com sucesso do estágio MEM para o EX, permitindo que a instrução or execute sem atrasos e com o operando correto.

Este resultado valida o funcionamento da Unidade de Forwarding, uma otimização essencial para a eficiência de processadores com arquitetura pipeline.

### 3.5. Análise de Hazard de Controle com Flush

O último e mais complexo cenário de teste valida a resposta do processador a um hazard de controle. Analisamos a execução da instrução beq x12, x12, +8, um desvio que, por comparar um registrador com ele mesmo, será sempre tomado. Isso exige que o processador descarte as instruções que foram buscadas especulativamente, assumindo um fluxo sequencial.





**Figura 4. Forma de onda do flush no pipeline após um desvio condicional (beq) ser tomado.**

A Figura 4 captura o momento crítico em que a decisão do desvio é tomada no estágio MEM e o mecanismo de flush é acionado para corrigir o pipeline.

A análise da forma de onda, com foco no ciclo entre 110ns e 120ns, demonstra a sequência de eventos:

- **O Gatilho (Estágio MEM):** A instrução `beq` está no estágio MEM. A sua condição é validada quando os sinais `branch_mem` (indicando que é um desvio) e `zero_flag_mem` (indicando que o resultado da comparação foi zero) vão para nível alto. Como resultado direto, o sinal **`branch_taken_mem`** é ativado.
- **A Ação (Hazard Unit):** O sinal `branch_taken_mem` serve como entrada para a Unidade de Detecção de Hazards. Ela responde imediatamente ativando os três sinais de limpeza: `if_id_flush`, `id_ex_bubble_enable` e `ex_mem_flush`.
- **A Consequência (Estágios IF e ID):** O efeito do flush é visível nos ciclos seguintes:
  - No ciclo de 120ns, o `pc_if` salta de seu valor sequencial (0x24) para o endereço de destino do desvio (0x28), corrigindo o fluxo do programa.
  - No ciclo de 130ns, a instrução no estágio ID (`instruction_id`) torna-se um NOP (0x13). Isso prova que a instrução incorreta (`addi x19, . . .`) foi com sucesso anulada e substituída por uma bolha.

Esta simulação valida com sucesso o mecanismo de flush do pipeline, demonstrando que o processador consegue se recuperar de desvios tomados, garantindo a integridade da execução do programa.

## Referências

- [1] Patterson, D. A. and Hennessy, J. L. (2017). *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, Cambridge.