”Desenvolvimento de um processador em arquitetura MIPS com suporte a pipeline”

1st Elian R. Pinheiro
Engenharia Elétrica - Telecomunicações.
Universidade Federal do Amazonas
Manaus, Brasil
elian.pinheiro@ufam.edu.br

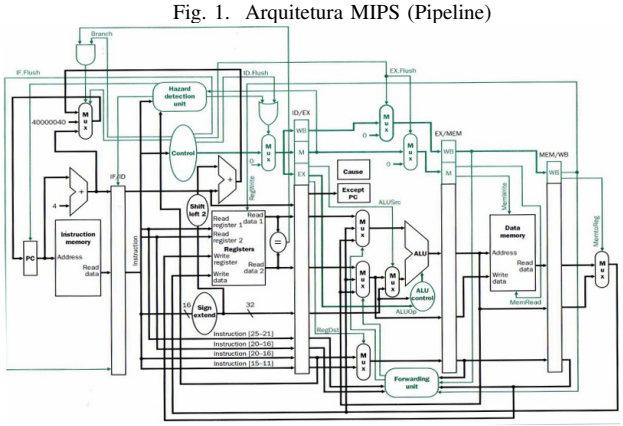
2nd Daniel Ramos Maia
Engenharia Elétrica - Eletrônica.
Universidade Federal do Amazonas
Manaus, Brasil
daniel.maia@ufam.edu.br

2nd Rodrigo Martins Henrique
Engenharia Elétrica - Eletrônica.
Universidade Federal do Amazonas
Manaus, Brasil
rodrigo.henrique@ufam.edu.br

I. Introdução

Este relatório técnico apresenta o desenvolvimento de um processador MIPS com suporte a pipeline, implementado utilizando a linguagem de descrição de hardware Verilog. O projeto foi realizado no contexto da disciplina de Arquitetura de Sistemas Digitais e Laboratório, com o objetivo de consolidar o conhecimento teórico-prático sobre arquitetura de processadores, controle de fluxo de dados e paralelismo em nível de instrução.

Fig. 1. Arquitetura MIPS (Pipeline)



Fonte: Produção própria. 14 de julho de 2025.

A arquitetura MIPS foi escolhida por sua simplicidade e ampla utilização acadêmica, sendo ideal para o estudo de conceitos fundamentais como segmentação de instruções (pipeline), identificação e resolução de perigos (hazards) de dados, controle e estrutura, além da modularização de componentes. O projeto foi desenvolvido de forma estruturada, dividindo o processador em blocos funcionais independentes, parametrizáveis e testáveis individualmente.

A etapa final do projeto consiste na integração dos módulos e na validação do sistema por meio da execução de um algoritmo de ordenação, escrito em linguagem C e traduzido para assembly MIPS. Para garantir a fidelidade e correção da implementação, foram realizadas simulações e testes no ambiente ModelSim, bem como a posterior síntese do projeto para embarque em um FPGA.

Este relatório segue o modelo de publicação técnica da IEEE e está estruturado para apresentar os detalhes do projeto, incluindo a descrição dos módulos, os códigos implementados, os testes realizados e a análise dos resultados obtidos.

II. Objetivo geral

Desenvolver um processador MIPS com suporte a pipeline, utilizando a linguagem de descrição de hardware Verilog, além de embarcar em um FPGA e executar um algoritimo de ordenação

III. Materiais e Ferramentas Utilizadas

A implementação do processador MIPS foi realizada utilizando a placa de desenvolvimento **FPGA DE10-Lite**, em conjunto com a linguagem de descrição de hardware **Verilog**.

Para embarcar os módulos no FPGA DE10-Lite, foi utilizada a ferramenta **Intel Quartus Prime**, versão 24.1std. Essa ferramenta foi empregada tanto para a síntese quanto para a simulação dos módulos desenvolvidos.

Por fim, utilizou-se o **ModelSim Starter Edition**, versão 10.5b, para a criação de testbenches e formas de onda, possibilitando uma análise prévia do funcionamento dos módulos. E usamos 100kHz como clock.

IV. Visão Geral dos Estágios e seus Módulos

Um processador MIPS com um pipeline clássico de 5 estágios (IF, ID, EX, MEM, WB) possui os seguintes módulos principais:

A. Estágio de Busca da Instrução (IF - Instruction Fetch)

O objetivo deste estágio é buscar a próxima instrução da memória.

Módulos:

1) **PC (Program Counter / Contador de Programa)**: Um registrador que armazena o endereço da instrução a ser buscada.
2) **Memória de Instruções (Instruction Memory)**: Um bloco de memória (geralmente uma cache de instruções) que armazena o código do programa. Ele recebe um endereço do PC e retorna a instrução correspondente.
3) **Somador (Adder)**: Um circuito simples que calcula PC + 4 para apontar para a próxima instrução sequencial (já que cada instrução MIPS tem 4 bytes).

B. Estágio de Decodificação da Instrução e Leitura de Registradores (ID - Instruction Decode)

O objetivo é decodificar a instrução buscada e ler os valores dos registradores necessários.

Módulos:

1) **Banco de Registradores (Register File)**: Contém os 32 registradores de propósito geral do MIPS. Possui duas portas de leitura (para ler os valores dos registradores `rs` e `rt`) e uma porta de escrita (usada no estágio WB).
2) **Unidade de Controle (Control Unit)**: O "cérebro" do processador. É um circuito combinacional que analisa o *opcode* (e *funct* para instruções do tipo R) da instrução e gera os sinais de controle para os outros estágios (ex: `RegWrite`, `MemRead`, `MemWrite`, `ALUOp`, etc.).
3) **Unidade de Extensão de Sinal (Sign-Extend Unit)**: Pega o valor imediato de 16 bits da instrução (usado em instruções como `addi`, `lw`, `sw`) e o estende para 32 bits, preservando o sinal.

C. Estágio de Execução ou Cálculo de Endereço (EX - Execute)

O objetivo é executar a operação matemática/lógica ou calcular o endereço para acesso à memória.

Módulos:

1) **ULA (Unidade Lógica e Aritmética) / ALU (Arithmetic Logic Unit)**: Realiza operações como soma, subtração, AND, OR, etc. Seus operandos vêm do estágio ID (seja de registradores ou do valor imediato).
2) **Multiplexadores (MUXes)**: Usados para selecionar a entrada correta para a ULA. Por exemplo, um MUX decide se o segundo operando da ULA é o valor de um registrador (instrução tipo R) ou o valor imediato estendido (instrução tipo I).

D. Estágio de Acesso à Memória (MEM - Memory Access)

O objetivo é ler ou escrever dados na memória principal. Apenas instruções de `load` (como `lw`) e `store` (como `sw`) usam este estágio ativamente.

Módulos:

1) **Memória de Dados (Data Memory)**: Um bloco de memória (geralmente uma cache de dados) onde os dados do programa são armazenados.
   a) Para `lw`, ela lê o dado do endereço calculado pela ULA.
   b) Para `sw`, ela escreve um dado (vindo do estágio ID) no endereço calculado pela ULA.

E. Estágio de Escrita de Volta (WB - Write Back)

O objetivo é escrever o resultado da operação (seja da ULA ou da memória) de volta no banco de registradores.

Módulos:

1) **Multiplexador (MUX)**: Seleciona o que será escrito de volta no banco de registradores. A escolha é entre:
   a) O resultado da ULA (para instruções aritméticas/lógicas).
   b) O dado lido da memória (para instruções de `load`).

V. Contador de Programa (PC)

O contador de programa (Program Counter - PC) é um componente essencial em qualquer arquitetura de processador. Sua função é armazenar o endereço da próxima instrução a ser buscada da memória. O código-fonte do módulo implementado em Verilog pode ser acessado na íntegra no repositório do projeto[1].

A seguir, realiza-se uma análise detalhada do funcionamento do módulo.

A. Declaração do Módulo

O trecho inicial do código define as interfaces de entrada e saída do módulo, como mostra a Listagem 1.

Listing 1. Declaração do módulo PC

```
module PC (
    input wire clk,
    input wire rst,
    input wire [1:0] PCSrc,
    input wire [31:0] branch_target_i,
    input wire [31:0] jump_target_i,
    input wire [31:0] jr_target_i,
    output wire [31:0] pc_o
);
```

Os sinais do módulo possuem as seguintes funções:

- **clk**: sinal de clock que sincroniza o avanço do PC.
- **rst**: sinal de reset (ativo em nível alto) que reinicializa o PC.
- **PCSrc**: sinal de controle que seleciona a origem do próximo valor do PC.
- **branch_target_i**: endereço de destino para instruções de desvio condicional.
- **jump_target_i**: endereço de destino para instruções de salto incondicional.
- **jr_target_i**: endereço de destino armazenado em registrador (instruções tipo `jr`).
- **pc_o**: saída que representa o valor atual do PC.

B. Sinais Internos

Os sinais internos utilizados para armazenar e processar o valor do PC são declarados conforme a Listagem 2.

Listing 2. Declaração de sinais internos

```
reg  [31:0] pc_reg;
wire [31:0] pc_plus_4_w;
reg  [31:0] pc_next_w;
```

Esses sinais têm os seguintes propósitos:

- **pc_reg**: registrador que armazena o valor atual do PC.
- **pc_plus_4_w**: valor calculado de `PC + 4`, usado em instruções sequenciais.
- **pc_next_w**: valor a ser carregado como próximo PC, selecionado pela lógica de controle.

[1] https://github.com/ElianPinheiro12/Processador-MIPS-Pipeline/blob/6fc9946b307aa67a98e3e460705fe1dbe49e3a19/Verilog/PC.v

C. Definição dos Tipos de Acesso ao PC

O módulo define constantes locais para facilitar a leitura e manutenção do código, como mostra a Listagem 3.

Listing 3. Definição de parâmetros de controle

```
localparam S_PC_PLUS_4 = 2'b00;
localparam S_BRANCH    = 2'b01;
localparam S_JUMP      = 2'b10;
localparam S_JR        = 2'b11;
```

O sinal `PCSrc` controla a seleção do próximo valor do PC conforme os seguintes valores:

- **00** (`S_PC_PLUS_4`): execução sequencial padrão (`PC + 4`).
- **01** (`S_BRANCH`): salto condicional.
- **10** (`S_JUMP`): salto incondicional.
- **11** (`S_JR`): salto baseado no conteúdo de um registrador.

D. Lógica Combinacional de Seleção do Próximo PC

A lógica combinacional apresentada na Listagem 4 é responsável por selecionar, com base no valor de `PCSrc`, qual será o próximo endereço de instrução.

Listing 4. Seleção do próximo valor do PC

```
always @(*) begin
    case (PCSrc)
        S_PC_PLUS_4: pc_next_w = pc_plus_4_w;
        S_BRANCH:    pc_next_w =
            branch_target_i;
        S_JUMP:      pc_next_w = jump_target_i
            ;
        S_JR:        pc_next_w = jr_target_i;
        default:     pc_next_w = pc_plus_4_w;
    endcase
end
```

Essa lógica garante que o PC avance corretamente, de acordo com o tipo de instrução que está sendo executada.

E. Lógica Sequencial de Atualização do PC

A atualização do registrador do PC ocorre em resposta à borda de subida do sinal de clock ou ao sinal de reset, como descrito na Listagem 5.

Listing 5. Atualização sequencial do PC

```
always @(posedge clk or posedge rst) begin
    if (rst) begin
        pc_reg <= 32'h00000000;
    end else begin
        pc_reg <= pc_next_w;
    end
end
```

Se o reset estiver ativado, o PC é reiniciado para o endereço 0. Caso contrário, recebe o valor selecionado pela lógica combinacional.

F. Saída do Valor Atual do PC

Por fim, o valor atual armazenado no registrador é disponibilizado na saída do módulo, como mostra a Listagem 6.

Listing 6. Atribuição da saída do PC

```
assign pc_o = pc_reg;
```

Esse valor será utilizado nos estágios posteriores do pipeline para buscar a próxima instrução a ser executada.

G. Forma de Onda do PC

Nas figuras a seguir, são apresentados os resultados do testbench do módulo *Program Counter* (PC), responsável por determinar o endereço da próxima instrução a ser executada no processador. Os testes foram realizados variando o valor de entrada do sinal de controle **PCSrc**, responsável por selecionar a origem do novo valor do PC, conforme descrito a seguir.

Incremento Sequencial (`PC + 4`): Inicialmente, o sinal **PCSrc** é configurado com o valor `00`, o que indica o modo de execução sequencial padrão (`PC + 4`). Nesse modo, o registrador de saída **pc_o** deve apresentar a soma do valor atual de PC acrescido de 4.

- **PCSrc** = `00`

Como resultado, a saída **pc_o** apresenta corretamente o valor `0x00000100`, confirmando o comportamento esperado do módulo para essa configuração.

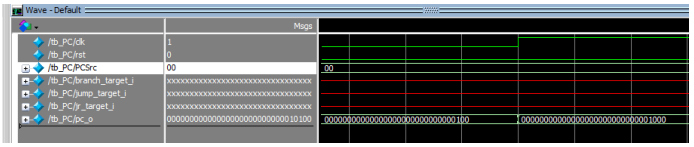


Fig. 2. Forma de onda — Incremento Sequencial

Fonte: Produção própria. 13 de junho de 2023.

Salto Condicional (Branch): Na sequência, o sinal **PCSrc** é alterado para o valor `01`, ativando o modo de salto condicional (*Branch*). Nesse caso, espera-se que a saída **pc_o** assuma diretamente o valor fornecido pela entrada **branch_target_i**.

- **PCSrc** = `01`

A forma de onda resultante demonstra que **pc_o** é corretamente atualizado com o valor de **branch_target_i**, validando o comportamento do módulo para essa configuração de controle.

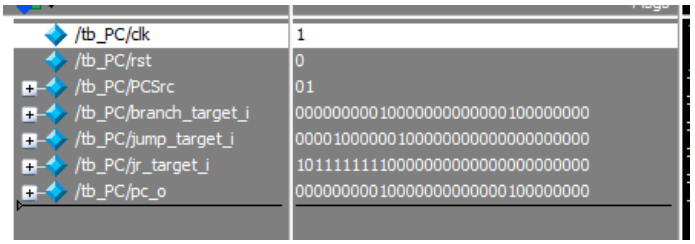


Fig. 3. Forma de onda — Salto condicional (Branch)

Fonte: Produção própria. 13 de junho de 2023.

Salto Incondicional (Jump): Posteriormente, o valor de **PCSrc** é definido como `10`, selecionando a operação de salto incondicional (*Jump*). Com isso, espera-se que **pc_o** adote o valor da entrada **jump_target_i**.

- **PCSrc** = `10`

O valor observado na saída **pc_o** corresponde corretamente a **jump_target_i**, validando a funcionalidade do salto incondicional.

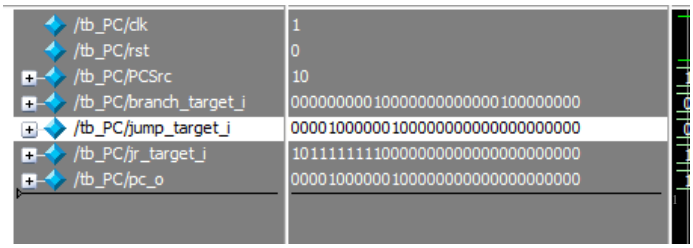


Fig. 4. Forma de onda — Salto incondicional (jump)

Fonte: Produção própria. 13 de junho de 2025.

Salto por Registrador (JR): Por fim, a entrada **PCSrc** é configurada como `11`, selecionando a operação de salto baseado em registrador (*Jump Register - JR*). Nesse modo, o valor de saída **pc_o** deve ser igual ao conteúdo da entrada **jr_target_i**.

- **PCSrc** = 11

A forma de onda confirma que **pc_o** adota corretamente o valor de **jr_target_i**, demonstrando o comportamento esperado para essa operação.

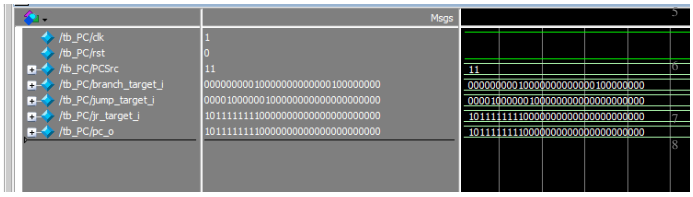


Fig. 5. Forma de onda - Salto por Registrador

Fonte: Produção própria. 13 de junho de 2023.

VI. Memória de Instruções

A memória de instruções tem como principal função armazenar o conjunto de instruções do programa que será executado pelo processador. O módulo analisado foi implementado em Verilog e pode ser acessado no repositório do projeto[2].

Este módulo representa uma memória de instruções simplificada, adequada para fins de simulação ou implementação de um processador MIPS básico.

A. Interfaces do Módulo

O módulo apresenta as seguintes entradas e saídas, conforme ilustrado na Listagem 7.

[2]https://github.com/ElianPinheiro12/Processador-MIPS-Pipeline/blob/8430018388e888d28e3b9522d23d1f47f3ce4aae/Verilog/instruction_memory.v

Listing 7. Declaração das interfaces do módulo de memória de instruções

```
input wire [31:0] address,          //
    Endere o da instru  o (vindo do PC)
output wire [31:0] instruction      //
    Instru  o lida da mem ria
);
```

A funcionalidade de cada sinal é descrita a seguir:

- **address**: endereço da instrução fornecido pelo contador de programa (PC).
- **instruction**: instrução de 32 bits correspondente ao endereço especificado.

B. Estrutura da Memória Interna

A memória interna é implementada como um vetor de registradores, conforme apresentado na Listagem 8.

Listing 8. Declaração da memória interna

```
reg [31:0] memory [0:255]; // Mem ria com 256
    posi  es (32 bits cada)
```

Este vetor representa uma memória com capacidade de 256 palavras de 32 bits, totalizando 1 KB (256 × 4 bytes). Cada posição armazena uma instrução do conjunto MIPS, codificada em formato binário.

C. Inicialização da Memória

Durante a simulação, as instruções são carregadas por meio de um bloco `initial`, conforme a Listagem 9.

Listing 9. Inicialização da memória com instruções MIPS

```
initial begin
    memory[0] = 32'h20080001; // addi $t0,
        $zero, 1
    memory[1] = 32'h20090002; // addi $t1,
        $zero, 2
    memory[2] = 32'h01095020; // add  $t2, $t0
        , $t1
    memory[3] = 32'hAC0A0000; // sw   $t2, 0(
        $zero)
    memory[4] = 32'h8C0B0000; // lw   $t3, 0(
        $zero)
    memory[5] = 32'h00000000; // nop
end
```

As instruções presentes neste bloco correspondem a operações básicas do conjunto MIPS:

- `addi $t0, $zero, 1`: carrega o valor 1 em $t0.
- `addi $t1, $zero, 2`: carrega o valor 2 em $t1.
- `add $t2, $t0, $t1`: soma $t0 e $t1, armazenando o resultado em $t2.
- `sw $t2, 0($zero)`: armazena o conteúdo de $t2 no endereço 0 da memória.
- `lw $t3, 0($zero)`: carrega em $t3 o conteúdo da memória do endereço 0.
- `nop`: instrução de ”não operação”, usada para preencher ciclos do pipeline.

D. Leitura da Instrução

A leitura da instrução é realizada conforme a Listagem 10.

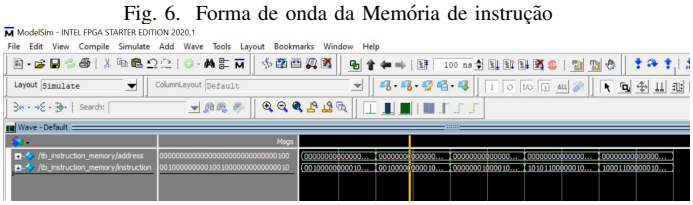
Listing 10. Leitura da instrução da memória

```
assign instruction = memory[address[9:2]];
```

A indexação `address[9:2]` é utilizada para garantir alinhamento com instruções de 4 bytes, uma vez que a memória é palavra-endereçável. Assim, os 8 bits utilizados (bits 2 a 9) permitem endereçar corretamente as 256 posições disponíveis na memória.

E. Forma de onda da memória de instruções

Fig. 6. Forma de onda da Memória de instrução



Fonte: Produção própria. 13 de junho de 2023.

VII. Banco de Registradores

O banco de registradores é uma das estruturas fundamentais em um processador MIPS, responsável por fornecer armazenamento temporário de dados durante a execução das instruções. O módulo analisado encontra-se disponível no repositório do projeto[3].

Este componente implementa 32 registradores de 32 bits, com suporte para duas leituras simultâneas e uma escrita por ciclo de clock.

A. Interfaces de Entrada

As entradas do módulo estão listadas na Listagem 11.

Listing 11. Declaração das entradas do módulo de banco de registradores

```
input wire clk, rst, RegWrite;
input wire [4:0] ReadAddress1, ReadAddress2,
    WriteAddress;
input wire [31:0] WriteData;
```

A função de cada sinal de entrada é descrita a seguir:

- **clk**: sinal de clock, utilizado para sincronizar a escrita nos registradores.
- **rst**: sinal de reset assíncrono, utilizado para zerar todos os registradores.
- **RegWrite**: sinal de controle que habilita a escrita no registrador de destino.
- **ReadAddress1** e **ReadAddress2**: endereços dos registradores que serão lidos.
- **WriteAddress**: endereço do registrador que será escrito.
- **WriteData**: valor a ser escrito no registrador especificado.

B. Interfaces de Saída

As saídas do módulo são responsáveis por fornecer os dados lidos dos registradores, como mostrado na Listagem 12.

Listing 12. Declaração das saídas do banco de registradores

```
output wire [31:0] ReadData1, ReadData2;
```

As saídas possuem os seguintes significados:

- **ReadData1**: valor contido no registrador especificado por `ReadAddress1`.
- **ReadData2**: valor contido no registrador especificado por `ReadAddress2`.

[3]https://github.com/ElianPinheiro12/Processador-MIPS-Pipeline/blob/fb4a6fc99b0e0d834c458b0196bd8ba53570b32d/Verilog/Register_file.v

C. Memória Interna

A memória interna do banco é composta por 32 registradores de 32 bits, conforme definido na Listagem 13.

Listing 13. Declaração dos registradores internos

```
reg [31:0] registers [31:0];
```

Cada registrador é indexado por um número de 0 a 31, sendo que o registrador de índice 0 ($zero) é sempre igual a zero, por definição da arquitetura MIPS.

D. Lógica de Escrita (Síncrona)

A escrita nos registradores ocorre de forma síncrona com o sinal de clock, conforme apresentado na Listagem 14. Há dois comportamentos possíveis:

- **Reset**: quando `rst` está ativo, todos os registradores são inicializados com o valor zero.
- **Escrita**: se `RegWrite` estiver ativo e o registrador de destino for diferente de $zero, o dado é escrito na posição especificada.

Listing 14. Lógica de reset e escrita nos registradores

```
always @(posedge clk or posedge rst) begin
    if (rst) begin
        for (i = 0; i < 32; i = i + 1) begin
            registers[i] <= 32'b0;
        end
    end else if (RegWrite) begin
        if (WriteAddress != 5'b0) begin
            registers[WriteAddress] <=
                WriteData;
        end
    end
end
```

E. Lógica de Leitura (Assíncrona)

As leituras são realizadas de forma assíncrona, isto é, os valores dos registradores são refletidos nas saídas imediatamente após a mudança dos sinais de endereço. Se o endereço de leitura for igual a 0, o valor retornado será sempre zero, conforme mostrado na Listagem 15.

Listing 15. Leitura assíncrona dos registradores

```
assign ReadData1 = (ReadAddress1 == 5'b0) ?
    32'b0 : registers[ReadAddress1];
assign ReadData2 = (ReadAddress2 == 5'b0) ?
    32'b0 : registers[ReadAddress2];
```

Essa abordagem garante conformidade com a especificação da arquitetura MIPS, em que o registrador $zero é uma constante de valor nulo.

F. Forma de Onda do banco registrador

Nas figuras a seguir, são apresentados os resultados do testbench do módulo *register file*, responsável por fornecer armazenamento temporario de dados durante a execucao das instrucoes. nesse teste o input RegWite esta ativo, logo os dados armazenados em WriteData podem ser escrito no endereço fornecido por WriteAddress ou seja, o dado vai ser ligo pelo readData1 que corresponde ao endereço 01010

- **WriteAddress** = 01010

A forma de onda confirma que **ReadData1** adota corretamente o valor de **WriteData**, demonstrando o comportamento esperado para essa operação.

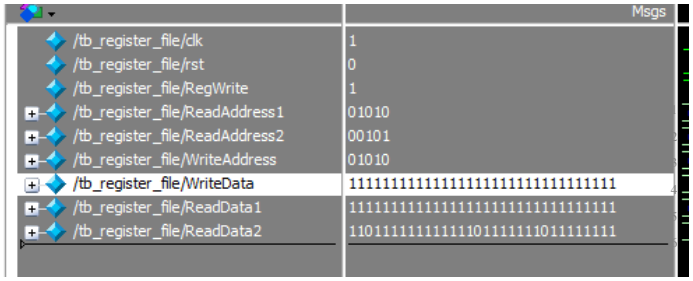


Fig. 7. Forma de onda banco registrador

VIII. Extensor de Sinal (Sign-Extend)

O código analisado se encontra Aqui. Esse módulo implementa um extensor de sinal (sign extender), utilizado em processadores MIPS para converter um valor imediato de 16 bits para 32 bits, preservando seu sinal (positivo ou negativo).

Quando se usa um valor imediato de 16 bits em uma instrução como em addi, lw, sw, etc. Ele precisa ser convertido para 32 bits para ser compatível com os registradores de 32 bits. A extensão de sinal faz isso mantendo o valor numérico correto, se o bit mais significativo (in[15]) for 0, o número é positivo, e os bits extras são preenchidos com 0. Se in[15] for 1, o número é negativo, e os bits extras são preenchidos com 1.

Assim temos:

```
module sign_extender (
    input wire [15:0] in,           //
        Imediato de 16 bits (entrada)
    output wire [31:0] out          // Sa da
        estendida para 32 bits
);
```

Onde:

- **in**: valor imediato de 16 bits, vindo de uma instrução tipo I (formato imediato).
- **out**: saída de 32 bits com o sinal estendido corretamente.

IX. Unidade Lógica e Aritmética (ALU)

O código descrito Aqui. É um código descreve uma Unidade Lógica e Aritmética (ALU) simples, usada comumente em processadores para realizar operações como soma e subtração.

A. Declaração do Módulo

O observemos o seguimento:

```
    module alu(
    input  [31:0] a,
    input  [31:0] b,
    input  [3:0]  alu_control, // Sinal que
        define a opera  o
    output reg [31:0] result,
    output zero
);
```

Onde:

- **a** e **b**: Entradas de 32 bits, operandos da operação.
- **alu_control**: Entrada de 4 bits que define qual operação será realizada.
- **result**: Saída de 32 bits que guarda o resultado da operação.
- **zero**: Saída que indica se o resultado foi igual a zero.

B. Lógica da ALU

Mas o principal seguimento, onde se encontra a lógica da ULA é:

```
always @(*) begin

case (alu_control)
        OP_ADD: result = a + b;
        OP_SUB: result = a - b;
        default: result = 32'hxxxxxxxx; // 'x'
            = valor indefinido
    endcase
end
```

Onde:

- Bloco always @(*) indica que essa é uma lógica combinacional, ou seja, result muda sempre que a, b ou alu_control mudarem.
- O **case** compara alu_control com os parâmetros definidos:
  - Se for **OP_ADD**, faz a soma.
  - Se for **OP_SUB**, faz a subtração.
  - **default**: coloca xxxxxxxx para indicar que o resultado é indefinido se a operação não for reconhecida.

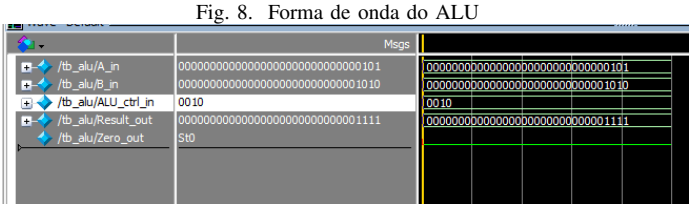
C. Forma de Onda da ALU

A Figura 3 apresenta a forma de onda resultante da simulação de uma operação aritmética realizada pela Unidade Lógica e Aritmética (ALU). Nesse teste, foram fornecidos os seguintes valores de entrada:

- **A_in** = 5;
- **B_in** = 10;
- **ALU_ctrl_in** = 2, representando a operação de adição.

Como resultado, a saída **result_out** apresenta corretamente o valor **15**, confirmando o funcionamento esperado da ALU para a operação de soma.

Fig. 8. Forma de onda do ALU



Fonte: Produção própria. 13 de junho de 2023.

X. Memória de Registrador

O código analisado se encontra Aqui. Esse módulo representa um registrador de instrução (Instruction Register), um componente comum em processadores. Ele serve para armazenar a instrução atual que está sendo executada, vinda da memória de instruções.

A. Entradas e Saídas

O segmento que introduz as entradas e saídas é:

```
    input wire clk,
    input wire reset,
    input wire load,
    input wire [31:0] instruction_in,
    output reg [31:0] instruction_out
);
```

Onde:

- **clk**: sinal de clock. O registrador é atualizado na borda de subida.
- **reset**: zera o registrador quando ativado.
- **load**: habilita o carregamento da nova instrução.
- **instruction_in**: instrução de entrada (vinda da memória de instruções).
- **instruction_out**: instrução atualmente armazenada (usada pelas etapas seguintes da CPU).

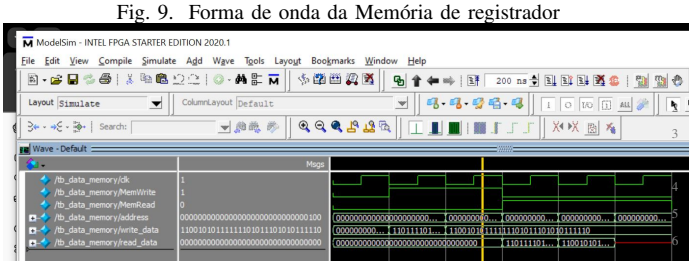
B. Lógica Sequencial

```
        always @(posedge clk) begin
        if (reset)
            instruction_out <= 32'b0;
        else if (load)
            instruction_out <= instruction_in;
    end
```

A cada borda de subida do clk. Se reset estiver ativo, zera o registrador. Se não, se load estiver ativo, armazena a nova instrução. Caso contrário, mantém o valor atual.

C. Forma de onda da memória de registrador

Fig. 9. Forma de onda da Memória de registrador



Fonte: Produção própria. 13 de junho de 2023.

XI. Memória de Dados

O código em questão se encontra Aqui.

Esse módulo representa uma memória de dados simples, geralmente usada em arquiteturas como a do MIPS para armazenar dados durante a execução de programas.

A. Entradas e Saídas

O seguinte seguimento define as entradas e saídas do código:

```
    input wire clk,                    // Clock
        para escrita
    input wire MemWrite,               //
        Habilita escrita
    input wire MemRead,                //
        Habilita leitura
    input wire [31:0] address,         //
        Endere o da mem ria
    input wire [31:0] write_data,      // Dado a
        ser escrito
    output reg [31:0] read_data        // Dado
        lido
```

Onde:

- **clk**: sinal de clock, necessário para a escrita síncrona.
- **MemWrite**: ativa a escrita na memória.
- **address**: endereço de memória de 32 bits.
- **write_data**: dado que será escrito na memória.
- **read_data**: valor lido da memória, retornado como saída.

B. Memória Interna

Definindo a memória como um vetor de 256 palavras de 32 bits. Isso dá 1 KB de memória total (256 × 4 bytes). Temos:

```
reg [31:0] memory [0:255];        // Mem ria
    com 256 posi  es de 32 bits
```

C. Escrita Síncrona

Observemos:

```
always @(posedge clk) begin
    if (MemWrite)
        memory[address[9:2]] <= write_data; //
            Escrita sincronizada
end
```

A escrita só ocorre na borda de subida do clk (sincronizada). O endereço address[9:2], considera apenas bits 2 a 9 do endereço.

Isso ocorre porque a memória é palavra-endereçável (acesso por palavra de 4 bytes). Como cada palavra tem 4 bytes (32 bits), os dois bits menos significativos ([1:0]) são ignorados (pois indicam deslocamentos dentro de uma palavra). Com 8 bits ([9:2]), acessamos 256 posições.

D. Leitura Combinacional

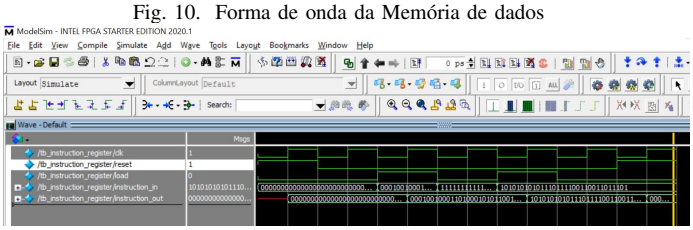
Observemos:

```
always @(*) begin
    if (MemRead)
        read_data = memory[address[9:2]];    //
            Leitura combinacional
    else
        read_data = 32'b0;
end
```

A leitura é combinacional, ocorre imediatamente quando MemRead está ativo. Se MemRead == 1, lê o conteúdo da memória no endereço address[9:2]. Se MemRead == 0, o valor de read_data é zerado (32'b0).

E. Forma de onda da memória de dados

Fig. 10. Forma de onda da Memória de dados



Fonte: Produção própria. 13 de junho de 2023.

F. Registrador de Instrução (IR)

O código analisado se encontra Aqui. Esse módulo implementa um deslocador lógico à esquerda por 2 posições (shift left by 2), um componente comum em arquiteturas como o MIPS, especialmente na construção de endereços de salto e branches.

Em processadores como o MIPS, o deslocamento à esquerda por 2 é usado para calcular endereços-alvo de instruções beq, bne (branch) e j, jal (jump). Isso ocorre porque os endereços de instruções são palavra-alinhados (4 bytes), e os deslocamentos nos campos de instrução são fornecidos em palavras, não bytes.

G. Entradas e Saídas

O segmento de código que introduz as entradas e saídas é:

```
    input wire [31:0] in,      // Entrada de 32 bits
    output wire [31:0] out     // Sa da deslocada
);
```

Onde:

- **in**: valor de entrada de 32 bits.
- **out**: resultado da operação de deslocamento lógico à esquerda por 2 bits.

Esse módulo realiza um deslocamento lógico fixo de 2 bits à esquerda. Usado principalmente em cálculos de endereços de salto em CPUs RISC como MIPS. Simples, eficiente e frequentemente integrado ao caminho de dados.

XII. Pipe Line

A estrutura do pipeline é, essencialmente, um conjunto de registradores intermediários que tem como função aumentar a eficiência geral de um processador.

Na arquitetura MIPS, por exemplo, um processador possui um total de 5 etapas de execução. O pipeline permite que essas etapas sejam realizadas de forma sobreposta, ou seja, em série com múltiplas instruções ao mesmo tempo.

Em outras palavras, uma única instrução precisa passar por 5 estágios. O pipeline garante que, enquanto a primeira instrução estiver sendo executada para o segundo estágio, a próxima instrução já possa iniciar o primeiro estágio, e assim por diante.

É uma forma de paralelismo interno que aumenta o desempenho geral do processador ao melhorar a taxa de instruções processadas por ciclo de clock.

No final, o que chamamos de pipeline é um conjunto de registradores intermediários posicionados entre os blocos funcionais de operação de um processador. Esses registradores servem para armazenar os dados e sinais de controle que serão passados de uma etapa para outra, permitindo a execução sobreposta de várias instruções.

Um pipeline completo é composto por um total de 4 blocos (ou estágios) independentes, que operam de forma sequencial, mas simultânea em instruções diferentes. Esses estágios exercem, em essência, a mesma função: permitir a execução contínua e eficiente das instruções. Os estágios clássicos são, respectivamente:

A. IF/ID

O código a ser analisado pode ser encontrado aqui. Esse código tem como função receber as operações do PC, e em seguida as transmiti-la para o bloco EX. Onde podemos observar que:

```
input clk,
input rst,
input en,
input flush,
input [31:0] pc_plus_4_if,
input [31:0] instruction_if,
output reg [31:0] pc_plus_4_id,
output reg [31:0] instruction_id
```

Essas sãs as entradas e saídas do registrador, onde:

- **clk**: Clock do sistema.
- **rst**: Reset sincronizado, zera as saídas.
- **en**: Permite a atualização, usado para stall.
- **flush**: Força a invalidação da instrução.
- **pc_plus_4_if**: PC + 4, valor para o próximo endereço.
- **instruction_if**: Instrução vinda da memória.
- **pc_plus_4_id**: Saída para o próximo estágio (ID).
- **instruction_id**: Instrução a ser decodificada.

O que logo em seguida nos leva a lógica principal, dentre as os pipe lines, o IF/ID é o mais simples e direto dentre eles, portanto mais curto.

```
if (rst) begin
    ...
end
else if (flush) begin
    ...
end
else if (en) begin
    ...
end
```

Onde se rst for ativo limpa o registrador. Se en é ativo sem um flush, os valores dos registradores são atualizados, e se en == 0, o valor dos registradores é mantido.

B. ID/EX

O código a ser analisado pode ser encontrado aqui. Esse código tem como função receber as operações de IF, e em seguida as transmiti-la para o bloco EX. Onde podemos observar que:

```
module id_ex_register(
    input clk,
    input rst,
    input en,
    input flush,
```

São as entradas iniciais que não servem exatamente para recepcionar em especifico nada. Mas tem como objetivo introduzir elementos importantes para a operação do pipe line. Onde:

- **clk**: clock do sistema.
- **reset**: sinal de reset, zera o registrador de pipeline.
- **en**: sinal de enable, usado para trava o pipeline.
- **flush**: limpa os valores do estágio em caso de desvio errado.

O que nos leva às entradas dos dados de instrução, onde se encontra a maior importância do sistema.

```
input [31:0] rdA_id, rdB_id,
input [4:0] rs_id, rt_id, rd_id,
input [5:0] funct_id,
input [31:0] imm_ext_id,
```

Onde:

- **rdA_id** e **e rdB_id**: valores lidos dos registradores A e B.
- **rs_id**, **rt_id**, **rd_id**: registradores de origem/destino da instrução.
- **funct_id**: campo funct das instruções do tipo R.
- **imm_ext_id**: imediato estendido para instruções tipo I, como lw, addi, etc.

As entradas de controle por sua vez são:

```
input RegDst_id,
input ALUSrc_id,
input [1:0] ALUOp_id,
```

Onde:

- **RegDst**: define se o registrador de destino será rd, tipo R. Ou rt, tipo I.
- **ALUSrc**: seleciona entre registrador ou imediato como operando da ALU.
- **ALUOp**: indica qual operação a ALU deve executar.

O que leva as saídas para o estágio EX.

```
output reg [31:0] rdA_ex, rdB_ex,
output reg [4:0] rs_ex, rt_ex, rd_ex,
output reg [5:0] funct_ex,
output reg [31:0] imm_ext_ex,

output reg ctrl_RegDst_ex,
output reg ctrl_ALUSrc_ex,
output reg [1:0] ctrl_ALUOp_ex
```

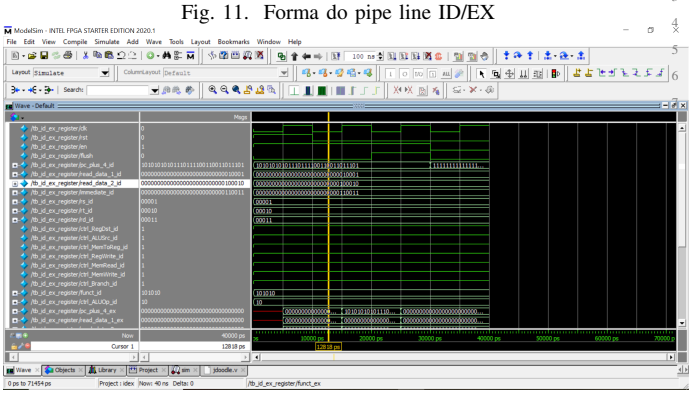
Estas são as versões registradas dos sinais que foram capturados no estágio ID e enviados ao estágio EX. Mas ainda há o bloco de atualização:

```
always @(posedge clk) begin
    if (rst || flush) begin
        ...
    end else if (en) begin
        ...
    end
end
```

Onde se rst ou flush estiveram ativos, zera todos os dados e sinais de controle. Isso equivale a inserir uma instrução NOP no pipeline. Se en estiver ativo, atualiza todos os registradores com os dados e sinais do estágio ID.

C. Forma de onda

Fig. 11. Forma do pipe line ID/EX



Fonte: Produção própria. 14 de julho de 2025.

D. EX/MEM

O código a ser analisado pode ser encontrado aqui. Esse código tem como função receber as operações de EX, e em seguida as transmiti-las para o bloco MEM. Onde podemos observar que:

```
input  [31:0]   alu_result_ex,
input  [31:0]   write_data_ex,
input  [4:0]    write_reg_addr_ex,
input  ctrl_MemToReg_ex,
input  ctrl_RegWrite_ex,
input  ctrl_MemRead_ex,
input  ctrl_MemWrite_ex,
```

Essas são as entradas do estagio EX, onde esses são os sinais que são resultados da execução da instrução. Onde:

- **alu_result_ex**: saída da ALU.
- **write_data_ex**: valor que pode ser armazenado na memória (sw).
- **write_reg_addr_ex**: registrador onde o resultado será escrito no estágio WB.

O que nos leva aos sinais de controle:

Esses sinais de controle determinam o comportamento da instrução no estágio MEM. Onde:

- **MemToReg**: decide se o valor vem da ALU ou da memória, usado no WB.
- **RegWrite**: ativa a escrita no banco de registradores.
- **MemRead**: ativa a leitura de memória (lw).
- **MemWrite**: ativa a escrita de memória (sw).

E por fim as entradas que serão enviadas para o segmento MEM:

```
output reg [31:0]   alu_result_mem,
output reg [31:0]   write_data_mem,
output reg [4:0]    write_reg_addr_mem,

output reg          ctrl_MemToReg_mem,
output reg          ctrl_RegWrite_mem,
output reg          ctrl_MemRead_mem,
output reg          ctrl_MemWrite_mem
```

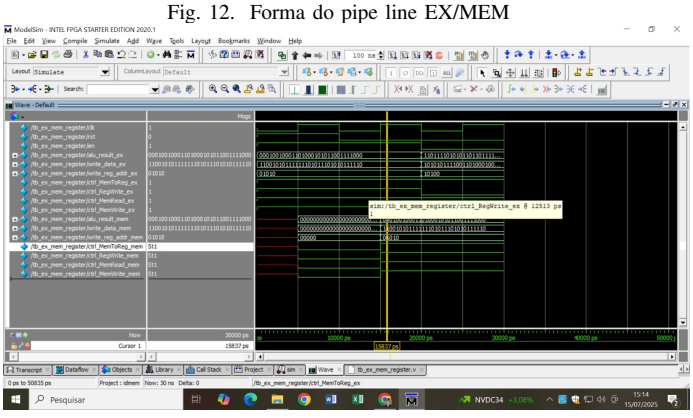
Estas são as versões registradas dos sinais, prontas para uso no próximo estágio MEM. O que levamos ao seguimento:

```
always @(posedge clk) begin
    if (rst) begin
        ...
    end else if (en) begin
        ...
    end
end
```

Onde se rst for ativo, zera os valores do registrador. Se en for ativo os dados de EX, passam para MEM. E por fim se en é ativo, mantêm os valores.

E. Forma de onda

Fig. 12. Forma do pipe line EX/MEM



Fonte: Produção própria. 14 de julho de 2025.

F. MEM/WB

O código a ser analisado pode ser encontrado aqui. Esse código tem como função receber as operações de MEM, e em seguida as transmiti-las para o bloco WB. Onde podemos observar que:

```
input  [31:0] read_data_mem,
input  [31:0] alu_result_mem,
input  [4:0]  write_reg_addr_mem,
```

São as entradas do estágio MEM, onde:

- **read_data_mem**: valor lido da memória para instruções como lw.
- **alu_result_mem**: resultado da operação da ALU.
- **write_reg_addr_mem**: registrador de destino para escrita no estágio WB.

Os sinais de controle por sua vez são:

```
input ctrl_MemToReg_mem,
input ctrl_RegWrite_mem,
```

Onde:

- **MemToReg**: define se o valor a ser escrito no registrador vem da memória ou da ALU.
- **RegWrite**: sinal que autoriza a escrita no banco de registradores.

E que levam as saídas que levam para o estágio WB, onde:

```
output reg [31:0] read_data_wb,
output reg [31:0] alu_result_wb,
output reg [4:0]  write_reg_addr_wb,
output reg        ctrl_MemToReg_wb,
output reg        ctrl_RegWrite_wb
```

Essas saídas são as versões armazenadas (registradas) dos dados e sinais de controle. São usados no estágio WB para finalizar a instrução, escrevendo o resultado no banco de registradores.

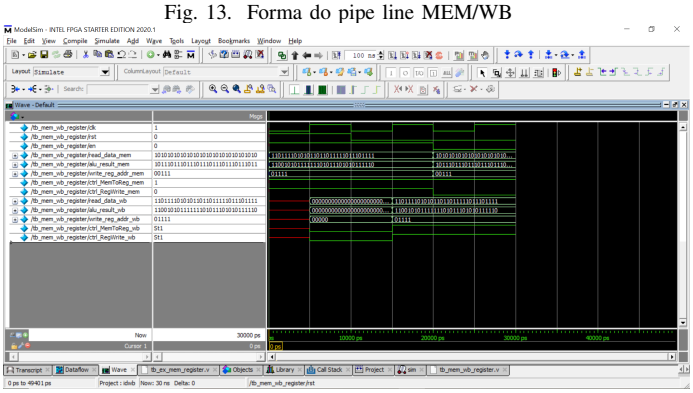
Finalizando o bloco com a lógica principal, como observamos no segmento:

```
always @(posedge clk) begin
    if (rst) begin
        ...
    end
    else if (en) begin
        ...
    end
end
```

Onde se rst for ativo, limpa todos os registradores. Se for en transfere os dados do estágio MEM para WB. Se en == 0, os valores ficam retidos.

G. Forma de onda

Fig. 13. Forma do pipe line MEM/WB



Fonte: Produção própria. 14 de julho de 2025.

XIII. Top_level

O Top level tem como função principal controlar o sistema como um todo. O seu código na íntegra se encontra aqui.

Onde podemos observar que:

```
pc pc_inst(
    .clk(clk),
    .rst(rst),
    .d(pc_plus_4),
    .en(en),
    .q(pc_current)
);
```

Esse segmento guarda o endereço da próxima instrução, podendo ser alterado por um Branch. Já o seguimento:

```
somador som_inst(...)          // pc + 4
somador_ab somAB_inst(...)     // Branch Target
     = PC+4 + (offset << 2)
```

Calculam o endereço da próxima instrução. E soma_result é usado no caso de Branch. Por sua vez:

```
assign pc_plus_4 = Branch ? soma_result :
    soma4_result;
```

Determina Decide entre seguir o fluxo sequencial (PC + 4) ou saltar para o endereço do branch. Enquanto:

```
memoria_instrucao mi_inst(...);
assign op_code_w = instruction_w[31:26];
assign instr_rs = instruction_w[25:21]; ...
```

Extrai os campos da instrução.

```
register_file reg_file(...);
```

Lê dois registradores (rdA_w, rdB_w). E escreve em um destino (rW_w) com valor vindo de mux_mem. O que nos leva naturalmente ao segmento:

```
assign rW_w = RegDst ? instr_rd : instr_rt;
```

Onde se Se RegDst == 1, usa rd, instruções tipo R. Se RegDst == 0, usa rt, instruções tipo I. Enquanto:

```
alu alu_inst(
  .a(rdA_w),
  .b(alu_input_b),
  .op(alu_op),
  .c(alu_result)
);
```

Opera com dois operandos, rdA_w e alu_input_b. Onde a operação definida por alu_op.

```
assign alu_input_b = alu_scr ? sin_ex_w :
    rdB_w;
```

Decide entre registrador (rdB_w) ou imediato estendido (sin_ex_w). O exetensor de sinal por sua vez:

```
sinal_ex sinal_ex_inst(...);
shift_left_2 shift_left_inst(...);
```

Expande O immediate de 16 para 32 bits. E multiplica por 4 para cálculos de desvio. Chegando ao fim:

```
data_memory data_mem_inst(
  ...
```

```
  .address(alu_result),
  .write_data(rdB_w),
  .read_data(r_data)
);
```

A memória de dados, usada para lw e sw, onde o endereço de memória vem da ULA. Onde a leitura ou escrita depende de MemRead e MemWrite.

```
assign mux_mem = MemToReg ? r_data :
    alu_result;
```

Por sua vez define se o valor escrito no registrador será o resultado da ALU ou o valor da memória. O que finaliza o código é o seguimento:

```
assign fim = alu_result;
```

Que Exibe o resultado da ALU.

XIV. Planejamento e Organização do Projeto

A. Objetivos

Essa relatório é o somatório de um esforço conjunto, cujo o resultado final se encontra na construção de um processador MIBS plenamente funcional. Nessa entrega se finaliza com a implementação do pipe line, finalizando assim o projeto como um todo.

Referências

[1] TERASIC TECHNOLOGIES, “DE2-115 User Manual” [Online]. Disponível em: https://www.terasic.com.tw/attachment/archive/502/DE2_115_User_manual.pdf. [Acessado em: 05-jun-2025].

[2] Neal S. Widmer, Gregory L. Moss, Ronald J. Tocci, ”Digital Systems-principles and applications.”. Pearson Education Limited, 2017