Implementação e Verificação de um Processador RISC-V *Pipeline* de Cinco Estágios para um Subconjunto de Instruções em Verilog

Victor Cabral, Otávio Guimarães

[1]Universidade Federal de Ouro Preto (UFOP) – ICEA
João Monlevade, MG – Brasil

{victor.cabral, otavio.agf}@ufop.edu.br

Abstract. *This work presents the design, implementation, and verification of a five-stage pipelined RISC-V processor in Verilog, supporting the instruction subset* `lh`*,* `sh`*,* `sub`*,* `or`*,* `andi`*,* `srl` *and* `beq`*. Core modules such as the ALU, register file, data/instruction memories, forwarding and hazard detection units, and pipeline registers were developed and integrated. Validation was performed through a dedicated test program with simulation in Icarus Verilog and waveform inspection in GTKWave, confirming correct functionality of each instruction and the effectiveness of the pipeline control logic in resolving data and control hazards.*

Resumo. *Este trabalho apresenta o projeto, implementação e verificação de um processador RISC-V pipeline de cinco estágios em Verilog, com suporte para o conjunto de instruções* `lh`*,* `sh`*,* `sub`*,* `or`*,* `andi`*,* `srl` *e* `beq`*. Foram desenvolvidos e integrados os módulos essenciais, incluindo ULA, banco de registradores, memórias de instrução e dados, unidade de forwarding, unidade de detecção de hazards e registradores de pipeline. A validação foi realizada com um programa de teste específico, em simulação no Icarus Verilog e visualização no GTKWave, confirmando a correta funcionalidade das instruções e a eficácia dos mecanismos de controle para a resolução de conflitos de dados e de controle, essenciais em arquiteturas pipeline modernas.*

Repositório do projeto:
`https://github.com/Vitor0302/Trabalho-RISC-V`

Sumário

1. Introdução

O conjunto de instruções RISC-V é uma arquitetura aberta que vem ganhando espaço na indústria e na academia pela simplicidade e flexibilidade. O objetivo deste trabalho foi projetar e implementar um processador RISC-V com arquitetura *pipeline* de cinco estágios capaz de executar um subconjunto de instruções, com ênfase no entendimento do caminho de dados, na interação entre seus componentes e, crucialmente, na resolução de conflitos inerentes a essa arquitetura.

A técnica de *pipeline* permite que múltiplas instruções sejam processadas simultaneamente em diferentes estágios, aumentando significativamente a vazão (throughput) do processador. No entanto, essa sobreposição introduz desafios conhecidos como conflitos ou *hazards*. Os **conflitos de dados** ocorrem quando uma instrução depende do resultado de uma instrução anterior que ainda não foi concluído. Os **conflitos de controle** surgem com instruções de desvio, que podem alterar o fluxo sequencial de execução, tornando as instruções já buscadas inválidas. Este trabalho aborda diretamente esses desafios através de unidades de adiantamento (*forwarding*) e de detecção de conflitos, que implementam lógicas de *stall* e *flush*.

O projeto foi realizado em Verilog e validado por meio de simulações, permitindo observar a propagação dos sinais e o comportamento do processador durante a execução de diferentes instruções, com suporte a `lh`, `sh`, `sub`, `or`, `andi`, `srl` e `beq`.

2. Metodologia

O processador foi projetado seguindo a arquitetura clássica de *pipeline* de cinco estágios do RISC-V: Busca de Instrução (IF), Decodificação (ID), Execução (EX), Acesso à Memória (MEM) e Escrita (Write-Back, WB). A implementação foi realizada em Verilog HDL, com uma abordagem estritamente modular, onde cada componente funcional (`ula.v`, `main_control.v`) e estágio do pipeline (`stage_if.v`, `stage_id.v`, etc.) foi desenvolvido em um módulo separado para garantir clareza, reusabilidade e facilidade de verificação. A integração final foi feita de forma estrutural no módulo de topo, `processador_pipeline_modular.v`.

2.1. Unidade de Controle

A unidade de controle é composta por dois módulos principais: a unidade de controle principal (`main_control.v`) e a unidade de controle da ULA (`alu_control.v`). A unidade principal analisa o campo *opcode* da instrução para gerar os sinais de controle primários. A unidade de controle da ULA refina a operação a ser executada pela ULA, utilizando os campos *funct3* e *funct7* para instruções do Tipo-R. A Tabela 1 resume os sinais de controle.

Tabela 1. Sinais de Controle Gerados por Instrução

Instrução	RegWrite	ALUSrc	MemToReg	MemRead	MemWrite	Branch	ALUOp
Tipo-R	1	0	0	0	0	0	10
lh (Load)	1	1	1	1	0	0	00
sh (Store)	0	1	X	0	1	0	00
andi	1	1	0	0	0	0	11
beq (Branch)	0	0	X	0	0	1	01

Tabela 2. Função dos Sinais de Controle

Sinal	Função
`RegWrite`	Habilita a escrita no banco de registradores.
`ALUSrc`	Seleciona a fonte do segundo operando da ULA (registrador ou imediato).
`MemToReg`	Seleciona o dado a ser escrito no registrador (resultado da ULA ou dado da memória).
`MemRead`	Habilita a leitura da memória de dados.
`MemWrite`	Habilita a escrita na memória de dados.
`Branch`	Indica uma instrução de desvio condicional.
`ALUOp`	Define a operação a ser realizada pela ULA (diretamente ou via *functs*).

2.2. Unidade Lógica e Aritmética (ULA)

A ULA (`ula.v`) é um módulo combinacional que recebe dois operandos de 32 bits e um código de controle de 4 bits, retornando o resultado da operação e uma *flag* que indica se o resultado é zero. A Listagem 1 mostra sua implementação.

```
module ULA (
    input  [31:0] in1, in2,
    input  [3:0]  ula_control,
    output reg [31:0] ula_result,
    output reg        zero_flag
);

    always @(*)
    begin
        // Bloco case para selecionar a operacao
        case(ula_control)
            4'b0000: ula_result = in1 & in2;    // AND
            4'b0001: ula_result = in1 | in2;    // OR
            4'b0010: ula_result = in1 + in2;    // ADD
            4'b0100: ula_result = in1 - in2;    // SUB
            4'b0101: ula_result = in1 >> in2;   // SRL
            4'b0011: ula_result = in1 << in2;   // SLL (extra)
            4'b0110: ula_result = in1 * in2;    // Multiply (extra)
            4'b0111: ula_result = in1 ^ in2;    // XOR (extra)
            4'b1000: begin                      // SLT (extra)
                if(in1 < in2)
                    ula_result = 32'd1;
```

```
                else
                    ula_result = 32'd0;
            end
            default: ula_result = 32'hxxxxxxxx; // Saida indefinida
        endcase

        // Logica para a flag 'zero', fora do case
        if (ula_result == 32'd0)
            zero_flag = 1'b1;
        else
            zero_flag = 1'b0;
    end

endmodule
```

Listagem 1. Implementação da ULA

2.3. Banco de Registradores

O módulo `banco_registradores.v` implementa os 32 registradores de 32 bits da arquitetura. Possui duas portas de leitura combinacionais (`rs1`, `rs2`) e uma porta de escrita síncrona (`rd`), que atua na borda de subida do *clock*. O registrador `x0` é fixo em zero, conforme a especificação.

2.4. Memórias de Instrução e de Dados

O processador utiliza duas memórias separadas.

2.4.1. Memória de Instrução

A memória de instrução (`memoria_instrucao.v`) é uma ROM que armazena as instruções do programa. A leitura é combinacional, e o endereço em bytes do PC é convertido para um endereço de palavra.

```
module Memoria_Instrucao (
    input  [31:0] address,
    output [31:0] instruction
);

    reg [31:0] rom_memory [0:255];

    // Carregando o programa diretamente na mem ria
    initial begin
        // Programa de teste
        rom_memory[0] = 32'h00001503; // lh x10, 0(x0)
        rom_memory[1] = 32'h00401583; // lh x11, 4(x0)
        rom_memory[2] = 32'h40b50633; // sub x12, x10, x11
        // ... (restante do programa)
    end

    assign instruction = rom_memory[address[9:2]];

endmodule
```

Listagem 2. Implementação da Memória de Instrução

2.4.2. Memória de Dados

A memória de dados (`memoria_dados.v`) é uma RAM para armazenar os dados manipulados. A leitura é combinacional, enquanto a escrita é síncrona, controlada pelo sinal `MemWrite` e pelo *clock*.

```
module Memoria_Dados (
    input         clock,
    input         MemWrite,
    input         MemRead,
    input  [31:0] address,
    input  [31:0] write_data,
    output [31:0] read_data
);

    // Declara  o da mem ria: 1024 posi  es de 32 bits.
    reg [31:0] data_memory [1023:0];

    // Inicializa  o da memoria com zeros
    initial begin
        // ... (c digo de inicializa  o)
    end

    // Logica de Leitura: Combinacional.
    assign read_data = data_memory[address[11:2]];

    // Logica de Escrita: Sincrona.
    always @(posedge clock) begin
        if (MemWrite) begin
            data_memory[address[11:2]] <= write_data;
        end
    end

endmodule
```

Listagem 3. Implementação da Memória de Dados

2.5. Componentes Estruturais do Pipeline

2.5.1. Multiplexadores

Multiplexadores (`mux2_1.v`, `mux3_1.v`) são seletores de dados combinacionais usados para guiar o fluxo de informações, como na escolha dos operandos da ULA ou do dado a ser escrito de volta no banco de registradores.

2.5.2. Registradores de Pipeline

Os registradores de *pipeline* (`reg_if_id.v`, `reg_id_ex.v`, `reg_ex_mem.v`, `reg_mem_wb.v`) são responsáveis por armazenar e propagar dados e sinais de controle entre os estágios a cada ciclo de *clock*, isolando-os e permitindo o paralelismo.

2.6. Unidades de Forwarding e Hazard Detection

A unidade de *forwarding* (`forwarding_unit.v`) implementa a lógica de adiantamento, que resolve a maioria dos conflitos de dados do tipo RAW (Read-After-Write). Ela compara os endereços dos registradores de origem (`rs1`, `rs2`) da instrução no estágio ID/EX com os endereços de destino (`rd`) das instruções nos estágios EX/MEM e MEM/WB. Se uma dependência é detectada, um multiplexador seleciona o resultado recém-calculado (da ULA ou da memória) em vez do valor obsoleto do banco de registradores.

A unidade de detecção de *hazards* (`hazard_detection_unit.v`) trata situações que o *forwarding* não resolve, como o conflito *load-use* (dependência de uma instrução de carga). Se uma instrução no estágio ID/EX depende do resultado de uma instrução `lh` no estágio EX/MEM, a unidade força um *stall* de um ciclo (uma bolha) no pipeline, travando o PC e o registrador IF/ID para que a instrução dependente espere o dado ser lido da memória. Esta unidade também gerencia o *flush* das instruções nos estágios iniciais quando um desvio condicional (`beq`) é tomado.

2.7. Ambiente de Simulação e Verificação

As simulações foram feitas com Icarus Verilog e a visualização das formas de onda no GTKWave. A verificação do projeto foi centralizada no módulo de *testbench* (`testbench_pipeline.v`). Este módulo é responsável por instanciar o processador, gerar o sinal de *clock*, controlar o *reset* inicial e pré-carregar os programas de teste na memória de instrução. Além disso, o *testbench* monitora sinais chave do processador a cada ciclo para imprimir um *log* detalhado do fluxo de execução, facilitando a depuração e a validação funcional.

3. Resultados e Discussão

3.1. Log de Execução do Testbench

A Listagem 4 apresenta o *output* íntegro da execução, que serve como guia para a análise das formas de onda.

```
Iniciando simulacao do processador RISC-V com PIPELINE - Grupo 12
Instrucoes suportadas: lh, sh, sub, or, andi, srl, beq

--- INICIALIZANDO MEMORIA DE DADOS ---
Mem[0] = 10, Mem[4] = 3


--- INICIO DA EXECUCAO DO PROGRAMA ---
```

```
--- Ciclo 1 ---
  IF : Buscando instrucao em 0x00 (lh x10, 0(x0))
  ID/EX/MEM/WB: <vazio>

--- Ciclo 2 ---
  IF : Buscando instrucao em 0x04 (lh x11, 4(x0))
  ID : Decodificando lh x10
  EX/MEM/WB: <vazio>

--- Ciclo 3 ---
  IF : Buscando instrucao em 0x08 (sub x12, x10, x11)
  ID : Decodificando lh x11
  EX : Executando lh x10 (Calculando endereco 0+x0)
  MEM/WB: <vazio>

--- Ciclo 4 ---
  IF : Buscando instrucao em 0x0C (or x13, x10, x11)
  ID : Decodificando sub x12, x10, x11
  EX : Executando lh x11 (Calculando endereco 4+x0)
  MEM: Acessando Mem[0] para lh x10
  WB : <vazio>

--- Ciclo 5 ---
  IF : Buscando instrucao em 0x10 (andi x14, x10, 8)
  ID : Decodificando or x13, x10, x11
  EX : Executando sub x12 (Operandos adiantados de MEM e WB!)
  MEM: Acessando Mem[4] para lh x11
  WB : Escrevendo resultado de lh x10 em x10. Resultado: x10 =
               10

--- Ciclo 6 ---
  IF : Buscando instrucao em 0x14 (srl x15, x10, 1)
  ID : Decodificando andi x14, x10, 8
  EX : Executando or x13
  MEM: <sem acesso>
  WB : Escrevendo resultado de lh x11 em x11. Resultado: x11 =
                3

--- Ciclo 7 ---
  IF : Buscando instrucao em 0x18 (beq x14, x10, 4)
  ID : Decodificando srl x15, x10, 1
  EX : Executando andi x14
  MEM: <sem acesso>
  WB : Escrevendo resultado de sub em x12. Resultado: x12 =
     7

--- Ciclo 8 ---
  IF : Buscando instrucao em 0x1C (sh x12, 8(x0))
  ID : Decodificando beq x14, x10, 4
  EX : Executando srl x15
  MEM: <sem acesso>
  WB : Escrevendo resultado de or em x13. Resultado: x13 =          11

--- Ciclo 9 ---
  IF : Buscando instrucao em 0x20 (beq x0, x0, 0)
```

```
  ID : Decodificando sh x12, 8(x0)
  EX : Executando beq x14, x10 (Comparando          14 e          10)
     . Desvio NAO sera tomado.
  MEM: <sem acesso>
  WB : Escrevendo resultado de andi em x14. Resultado: x14 =
               8

--- Ciclo 10 ---
  IF : Buscando proxima instrucao... (PC continua em +4)
  ID : Decodificando beq x0, x0, 0
  EX : Executando sh x12 (Calculando endereco 8+x0)
  MEM: <sem acesso>
  WB : Escrevendo resultado de srl em x15. Resultado: x15 =
     5

--- Ciclo 11 ---
  ID : <Instrucao apos o beq sendo anulada (flush)>
  EX : Executando beq x0, x0. Desvio SERA tomado! PC sera 0x20.
  MEM: Escrevendo valor de x12 (           7) em Mem[8] para sh
  WB : <sem escrita>

--- Ciclo 12 ---
  IF : Buscando instrucao em 0x20 (beq x0, x0, 0) - Loop iniciado.
  ID : <bolha>
  EX : <bolha>
  MEM: <sem acesso>
  WB : <sem escrita>


================ ESTADO FINAL ================

--- Registradores ---
x00: 0x00000000    x01: 0x00000001    x02: 0x00000002    x03: 0
    x00000003
x04: 0x00000004    x05: 0x00000005    x06: 0x00000006    x07: 0
    x00000007
x08: 0x00000008    x09: 0x00000009    x10: 0x0000000a    x11: 0
    x00000003
x12: 0x00000007    x13: 0x0000000b    x14: 0x00000008    x15: 0
    x00000005
x16: 0x00000010    x17: 0x00000011    x18: 0x00000012    x19: 0
    x00000013
x20: 0x00000014    x21: 0x00000015    x22: 0x00000016    x23: 0
    x00000017
x24: 0x00000018    x25: 0x00000019    x26: 0x0000001a    x27: 0
    x0000001b
x28: 0x0000001c    x29: 0x0000001d    x30: 0x0000001e    x31: 0
    x0000001f

--- Memoria de Dados (enderecos 0-36) ---
Mem[           0]: 0x0000000a (         10)
Mem[           4]: 0x00000003 (          3)
Mem[           8]: 0x00000007 (          7)
Mem[          12]: 0x00000000 (          0)
Mem[          16]: 0x00000000 (          0)
Mem[          20]: 0x00000000 (          0)
```

```
Mem[          24]: 0x00000000 (          0)
Mem[          28]: 0x00000000 (          0)
Mem[          32]: 0x00000000 (          0)
Mem[          36]: 0x00000000 (          0)

========================================
tb/testbench_pipeline.v:159: $finish called at 295000 (1ps)
```

Listagem 4. Log de execução do testbench (vvp)

3.2. Análise dos Resultados

A validação do processador foi realizada pela análise detalhada das formas de onda em conjunto com o log de execução. Os resultados confirmam a correta implementação do pipeline de cinco estágios e de seus mecanismos de controle de conflitos.

A execução de instruções de acesso à memória foi verificada com sucesso. A **Figura 1** detalha o percurso da instrução `lh x10, 0(x0)`, mostrando o cálculo do endereço `0x0` no estágio EX, o acesso à memória que retorna o dado `0xA` no estágio MEM, e finalmente a escrita deste valor no registrador `x10` durante o estágio WB. Da mesma forma, a **Figura 3** demonstra a operação `sh x12, 8(x0)`, onde o sinal `memwrite` é ativado no estágio MEM para armazenar corretamente o valor `0x7` (de `x12`) no endereço de memória `0x8`.

O mecanismo de *forwarding* foi essencial para o desempenho e sua funcionalidade é comprovada na **Figura 2**. Durante a execução da instrução `sub x12, x10, x11`, que depende dos resultados das duas instruções `lh` anteriores, os sinais `w_forward_a=2` e `w_forward_b=1` indicam que os operandos `x10` e `x11` foram adiantados dos estágios WB e MEM, respectivamente. Isso evita um *stall*, como também é mencionado no log do "Ciclo 5", garantindo que a ULA receba os valores corretos (`0xA` e `0x3`) para a subtração.

Finalmente, o tratamento de conflitos de controle foi validado pela instrução `beq`, como visto na **Figura 4**. No primeiro `beq` (`beq x14, x10`), o desvio não é tomado, e o fluxo continua sequencialmente. No segundo `beq` (`beq x0, x0`), o desvio é tomado. O sinal `branch_taken` é ativado, o PC é atualizado para o endereço de destino (`0x20`), e as instruções que estavam nos estágios IF e ID são invalidadas (*flush*), garantindo a correção do fluxo de execução, como descrito nos ciclos 11 e 12 do log.

3.3. Formas de Onda

As figuras a seguir ilustram momentos chave da execução que validam os mecanismos implementados.

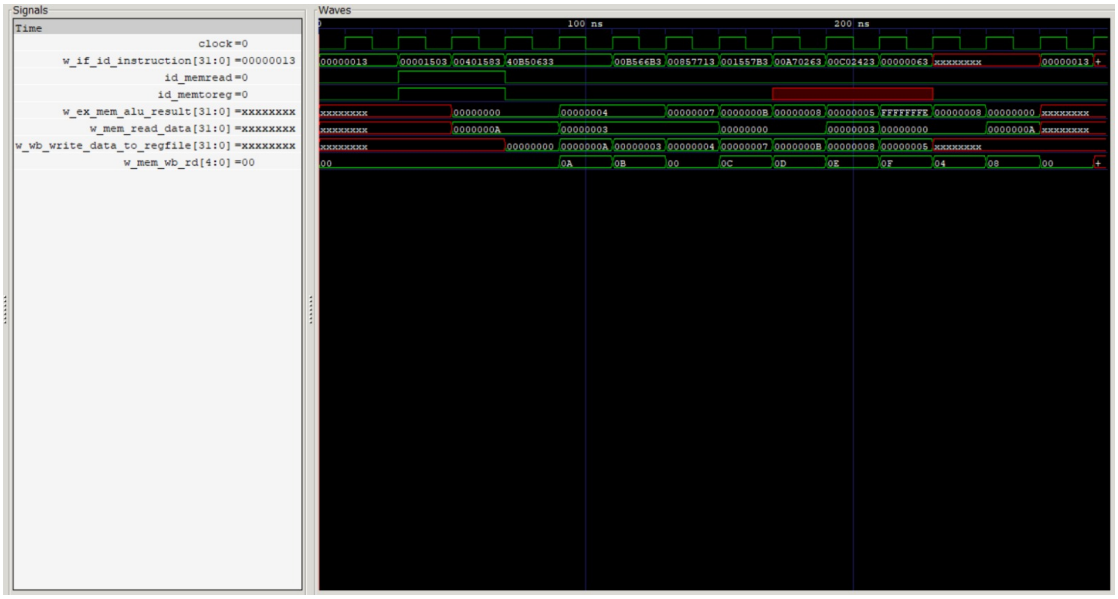

Figura 1. Execução da instrução `lh x10, 0(x0)` ao longo do pipeline. Em t=25ns, a instrução é decodificada com `MemRead=1` e `MemToReg=1`. No estágio EX, o endereço '0x0' é calculado. A leitura ocorre em t=65-85ns, retornando '0x0000000A'. Por fim, no estágio WB, este valor é enviado para escrita em `x10`.

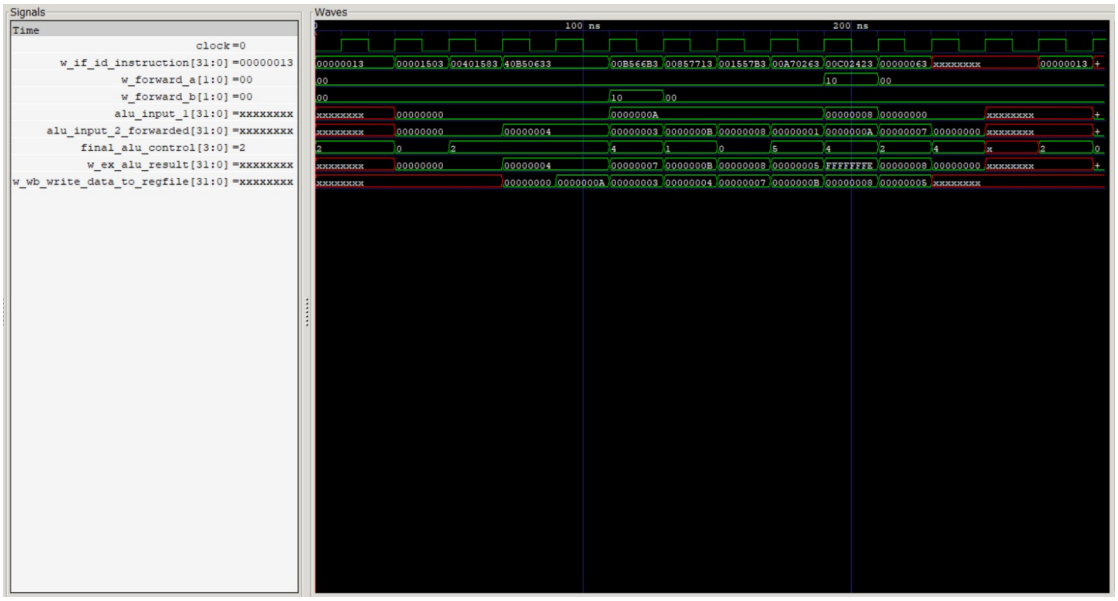

Figura 2. Forwarding de dados na instrução `sub x12, x10, x11`. Os sinais `w_forward_a=2` e `w_forward_b=1` demonstram o encaminhamento dos operandos dos estágios WB e MEM. Os valores corretos `0xA` (x10) e `0x3` (x11) são entregues à ULA, que executa a subtração resultando em `0x7`.

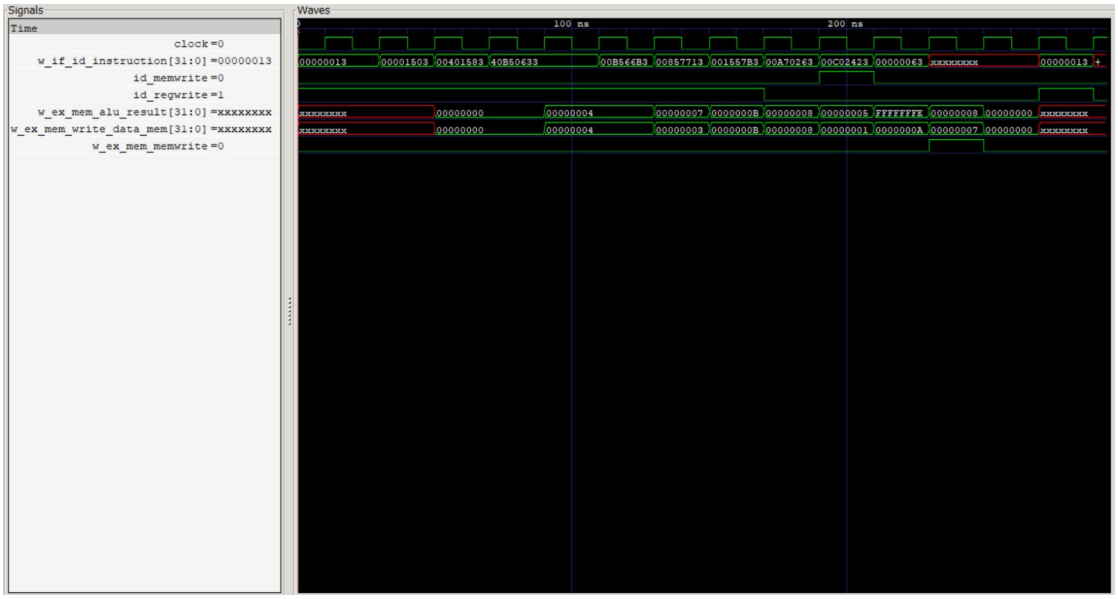

Figura 3. Execução da instrução `sh x12, 8(x0)`. O sinal `id_memwrite=1` é ativado na decodificação. No estágio EX, o endereço '0x8' é calculado. No estágio MEM, com `memwrite=1` ativo, o dado '0x7' (de 'x12') é escrito corretamente na memória.

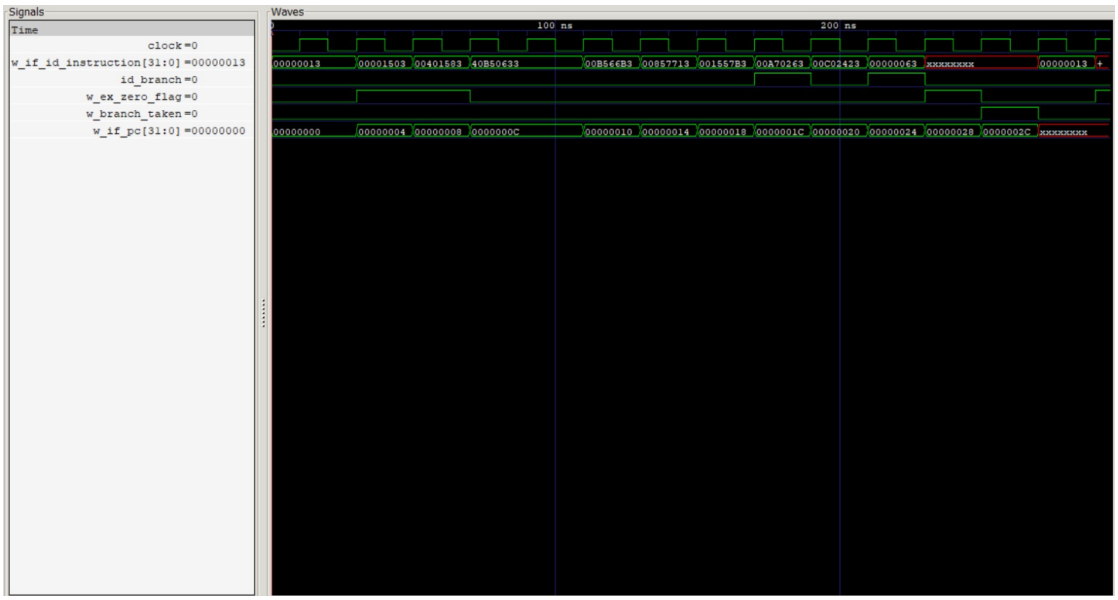

Figura 4. Tratamento de desvio condicional `beq`. No primeiro `beq` (t=145ns), o desvio não é tomado (`branch_taken=0`). No segundo (t=185ns), a condição é verdadeira (`zero_flag=1`), o desvio é tomado (`branch_taken=1`) e o PC é atualizado para `0x20`, realizando o `flush` do pipeline.

4. Conclusão

Este trabalho detalhou com sucesso o projeto, a implementação e a verificação de um processador RISC-V com pipeline de cinco estágios. A análise das simulações, combinando o log de execução e as formas de onda, evidenciou o correto funcionamento

do caminho de dados para instruções aritméticas, de acesso à memória e de desvio condicional. Mais importante, foi validada a eficácia dos mecanismos de controle implementados: a unidade de *forwarding* demonstrou ser capaz de resolver conflitos de dados do tipo RAW, evitando paradas desnecessárias no pipeline, enquanto a unidade de detecção de conflitos tratou corretamente o caso de *load-use* e gerenciou o *flush* do pipeline após desvios tomados. Os resultados comprovam que as unidades de controle, memória e adiantamento estão corretamente implementadas, atendendo aos requisitos funcionais e de desempenho esperados.

Como trabalhos futuros, diversas expansões podem ser exploradas para ampliar a capacidade do processador. Entre elas, destacam-se: a implementação de um conjunto maior de instruções da base RV32I, como as lógicas com imediatos e os saltos incondicionais; o desenvolvimento de um sistema de tratamento de exceções e interrupções, essencial para a execução de sistemas operacionais; e a síntese do projeto para uma plataforma de hardware reconfigurável (FPGA), permitindo a verificação em um ambiente real e a análise de métricas de desempenho como frequência de operação e consumo de área.

Referências

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