UFRJ
UNIVERSIDADE FEDERAL
DO RIO DE JANEIRO

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

TRABALHO DE ARQUITETURA DE COMPUTADORES
ATIVIDADES PRÁTICAS EM VHDL:
SOMADOR/SUBTRATOR VETORIAL
CPU COM RISC-V

DHAYSE DE LIMA TITO DRE: 120019062
EDUARDO DOS SANTOS GUIMARÃES JUNIOR DRE: 12013895
JOÃO PEDRO DUARTE BAPTISTA DRE: 121066907
LÍGIA CALINA BUENO BONIFÁCIO DRE: 122046065
LUIZA LISSANDRA RODRIGUES ROSA DRE: 119046349

Rio de Janeiro-RJ
2024

SUMÁRIO

1 INTRODUÇÃO

O presente trabalho tem como intuito englobar e apresentar três projetos propostos pela disciplina de Arquitetura de Computadores, sendo eles um somador vetorial, um projeto com Reduced Instruction Set Computer 5ª geração (RISC-V) e um projeto com RISC-V atrelado a Single Instruction, Multiple Data (SIMD).

Para fins de contextualização teórica, o RISC-V é uma abordagem responsável por trazer maior simplicidade a um conjunto de instruções e executá-las em uma menor quantidade de ciclos de clock. Já o SIMD está relacionado ao objetivo de fazer com que uma única instrução consiga controlar várias unidades de processamento com o intuito de executar uma mesma operação, com diferentes dados, de forma paralela, ou seja, simultânea.

O primeiro projeto proposto consiste em desenvolver um módulo VHDL capaz de realizar instruções de soma e subtração. O módulo tem a capacidade de lidar com dois vetores de 32 bits. As entradas do módulo são os vetores A e B, a operação a ser realizada (soma ou subtração) e o tamanho do vetores.

Já o segundo e o terceiro têm o intuito de criar uma Unidade de Processamento Central (CPU) de 32 bits capaz de suportar a arquitetura RISC-V, além da implementação do pipeline de inteiros com 5 estágios necessário para melhorar o desempenho na execução das instruções. No contexto do projeto, pipeline consiste em uma abordagem que permite que uma instrução seja dividida em etapas menores consecutivas e que serão processadas de maneira paralela.

Para desenvolver o trabalho foi utilizado o simulador LogiSIM-Evolution e implementado com blocos feitos em Linguagem de Descrição de Hardware VHSIC (VHDL), no qual VHSIC significa Circuito Integrado de Altíssima Velocidade. Esta linguagem permite modelar sistemas eletrônicos digitais através da definição, simulações e testagens das estruturas e comportamentos de circuitos digitais.

2 PROJETO DE UM SOMADOR/SUBTRATOR DE 32 BITS EM VHDL

Neste projeto, foi desenvolvido um módulo VHDL para um circuito combinacional de somador/subtrator vetorial de 32 bits. O circuito opera sobre números inteiros de 4, 8, 16 e 32 bits, conforme especificado por um sinal de controle. O objetivo principal foi projetar um sistema orientado ao desempenho, minimizando a latência no cálculo das somas.

2.1 DESCRIÇÃO DO MÓDULO

O módulo `SomadorVetorial` é responsável por realizar operações de soma e subtração em vetores de diferentes tamanhos. Ele possui as seguintes entradas e saídas:

- **Entradas**:
  - `A_i, B_i`: Operandos de 32 bits (`std_logic_vector(31 DOWNTO 0)`).
  - `mode_i`: Modo Somador ou Subtrator (1 bit `std_logic`).
  - `vecSize_i`: Tamanho do vetor (2 bits `std_logic_vector(1 DOWNTO 0)`).
    - `00`: 4 bits
    - `01`: 8 bits
    - `10`: 16 bits
    - `11`: 32 bits
- **Saída**:
  - `S_o`: Resultado da operação (`std_logic_vector(31 DOWNTO 0)`).

2.2 ARQUITETURA DO MÓDULO

A arquitetura `TypeArchitecture` define os sinais e funções auxiliares necessários para a operação do módulo:

2.2.1 Definição da Entidade e Arquitetura

```
4  ENTITY SomadorVetorial IS
5    PORT (
6      A_i, B_i    : IN  std_logic_vector (31 DOWNTO 0);
7      vecSize_i   : IN  std_logic_vector (1 DOWNTO 0);
8      mode_i      : IN  std_logic;
9      S_o         : OUT std_logic_vector (31 DOWNTO 0)
10   );
```

A entidade `SomadorVetorial` define as entradas e saídas do módulo.

```
13 ARCHITECTURE TypeArchitecture OF SomadorVetorial IS
14   SIGNAL C_cla : std_logic_vector (31 DOWNTO 0) := (others =&gt; '0'); --
     ↪  Vetor de carry
15   SIGNAL B_u   : std_logic_vector (31 DOWNTO 0) := (others =&gt; '0'); --
     ↪  Valor de B após a operação lógica com o modo
16   SIGNAL A_adj, B_adj : std_logic_vector (31 DOWNTO 0); -- Ajustados pelo
     ↪  tamanho do vetor
17
18   -- Função para ajustar o vetor de entrada de acordo com o tamanho
     ↪  especificado
```

A arquitetura `TypeArchitecture` declara os sinais internos necessários para a operação.

2.2.2 Função de Ajuste de Vetor

```
18   -- Função para ajustar o vetor de entrada de acordo com o tamanho
     ↪  especificado
19   FUNCTION adjust_vector(input : std_logic_vector(31 DOWNTO 0); size :
     ↪  std_logic_vector(1 DOWNTO 0)) RETURN std_logic_vector IS
20     VARIABLE result : std_logic_vector(31 DOWNTO 0) := (others =&gt;
       ↪  '0');
21   BEGIN
22     CASE size IS
23       WHEN "00" =&gt; result(3 DOWNTO 0) := input(3 DOWNTO 0); -- 4 bits
```

```
24        WHEN "01" =&gt; result(7 DOWNTO 0) := input(7 DOWNTO 0); -- 8 bits
25        WHEN "10" =&gt; result(15 DOWNTO 0) := input(15 DOWNTO 0); -- 16
          ↪ bits
26        WHEN OTHERS =&gt; result := input; -- 32 bits
```

A função `adjust_vector` ajusta o vetor de entrada conforme o tamanho especificado por `vecSize_i`.

2.2.3 Ajuste dos Vetores `A` e `B`

```
31  BEGIN
32    -- Ajustando vetores A e B pelo tamanho do vetor
33    A_adj &lt;= adjust_vector(A_i, vecSize_i);
34    B_adj &lt;= adjust_vector(B_i, vecSize_i);
35
36    -- Definição de B_u com base no modo de operação (somador ou subtrator)
```

Os vetores `A_adj` e `B_adj` são ajustados conforme o tamanho do vetor.

2.2.4 Definição de `B_u` Baseada no Modo de Operação

```
38    BEGIN
39      FOR i IN 0 TO 31 LOOP
40        B_u(i) &lt;= (B_adj(i) AND (NOT mode_i)) OR ((NOT B_adj(i)) AND
          ↪ mode_i);
41      END LOOP;
42    END PROCESS;
43
```

O vetor `B_u` é definido com base no modo de operação (somador ou subtrator).

2.2.5 Cálculo do Vetor de Carry

```
44    -- Cálculo do vetor de carry usando a técnica de Carry Lookahead
45    C_cla(0) &lt;= mode_i; -- Carry-in inicial é o modo de operação (0 para
      ↪ soma, 1 para subtração)
```

O vetor de carry `C_cla` é calculado utilizando a técnica de Carry Lookahead.

2.2.6 Soma Vetorizada

```
46    PROCESS (A_adj, B_u)
47    BEGIN
```

```
48      FOR i IN 1 TO 31 LOOP
49        C_cla(i) &lt;= (A_adj(i-1) AND B_u(i-1)) OR ((A_adj(i-1) OR
          ↪  B_u(i-1)) AND C_cla(i-1));
50      END LOOP;
```

A soma vetorizada é realizada bit a bit entre `A_adj`, `B_u` e `C_cla`.

2.3 SIMULAÇÃO

Foi utilizado o simulador Quartus Lite para testar o componente conforme mostra a imagem a seguir.

3 CPU RISC-V DE 32 BITS: DESIGN E IMPLEMENTAÇÃO COM PIPELINE DE 5 ESTÁGIOS

3.1 FUNDAMENTAÇÃO TEÓRICA

RISC-V é uma *Instruction Set Architectur (ISA)* baseada em poucas instruções, ela foi pensada para ser aberta, escalável e simples. RV32I refere-se ao conjunto de instruções base de 32 bits da arquitetura RISC-V, que representa a mínima arquitetura necessária para seu correto funcionamento em um computador.

As instruções base apresentam 6 formatos: Tipo-R para operações de registradores, Tipo-I para valores imediatos short e loads, Tipo-S para stores, Tipo-B para desvios condicionais, Tipo-U para valores imediatos longos, e tipo-J para saltos incondicionais. Para esse trabalho, utilizaremos especificamente as instruções a seguir.

- **Aritméticas**: add, addi e sub;
- **Lógicas**: and, andi, or, ori, xor e xori;
- **Deslocamento**: sll, slli, srl e srli;
- **Controle de fluxo**: jal, jalr, beq e bne;
- **Carregamento**: lw, lui;
- **Armazenamento**: sw;
- **Gerenciamento de ponteiros**: auipc.

Para aumentar a eficiência e maximizar o *throughput* do processador podemos utilizar técnicas como o *pipeline*. Essa técnica consiste em dividir as execuções das instruções em vários estágios, possibilitando assim que as instruções possam ser executadas simultaneamente. Os 5 estágios do pipeline que trabalharemos são Fetch (Busca), Decode (Decodificação), Execute (Execução), Memory (Memória) e Write Back (Escrita de Resultado).

É importante lembrar que, ao projetar uma CPU, as memórias de instrução e de dados são diferentes. Dentro do simulador LogiSIM-Evolution, utilizaremos a memória ROM para as intruções e a RAM para memória de dados. Além disso, essas memórias serão transferidas de modo assíncrono de modo a não dependerem de sincronia do clock para o correto funcionamento.

3.2 PLANEJAMENTO DO PROJETO

Na figura 1 podemos observar o diagrama de blocos da CPU com cada estágio destacado.

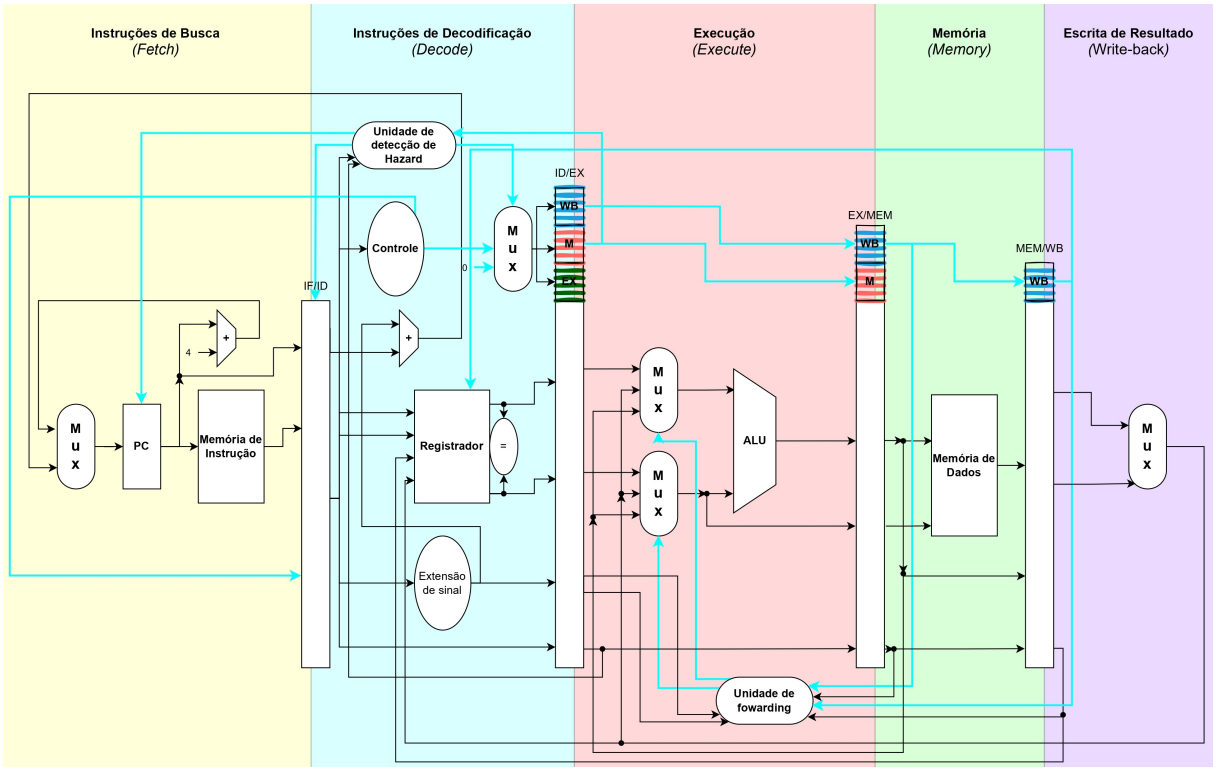

Figura 1 – Diagrama de blocos da CPU com pipeline de 5 estágio. Controle da ALU e o MUX ALUScr foi omitido para fins de simplificação.*Autoria própria.*

De modo geral as instruções seguem um fluxo linear de estágios da esquerda para a direita. No entanto, existem duas exceções principais a esse fluxo linear: a atualização do contador de programa (PC) e a etapa de escrita do resultado. A seleção do próximo valor do PC, pode ser o PC incrementado ou o endereço de desvio do estágio EX. Por sua vez, no estágio de escrita, os resultados da ALU ou dados da memória são enviados "para trás"para serem escritos no banco de registradores (estágio de decodificação).

Para organização da CPU, criamos os componentes em VHDL a partir de cada estágio do pipeline.

3.2.1 IF - Instruções de busca

Neste estágio, a próxima instrução a ser executada é buscada da memória de instruções. O endereço da próxima instrução é obtido a partir do *Program Counter* (PC), atualizada a cada ciclo de clock e a instrução é carregada no registrador.

Abaixo o detalhamento de cada componente desse estágio:

- **PC (Program Counter)**: Mantém o endereço da próxima instrução a ser buscada;
- **MUX**: Seleciona que valor substitui o **PC** (PC + 4 ou o endereço de destino do desvio), sendo controlada pela **Unidade de detecção de Hazard**;

- **Memória de Instrução**: Armazena as instruções do programa na memória ROM;
- **Somador (+4)**: Incrementa o valor do PC em 4 para apontar para a próxima instrução;
- **Registrador de pipeline IF/ID**: Armazena a instrução buscada e o endereço do PC para uso no próximo estágio. Possui 96 bits de largura (32 bits de instruções de busca e 64 bits do endereço do PC).

ID - Instrução de Decodificação

A instrução buscada é decodificada para entender o que precisa ser feito. Os campos da instrução são separados e os operandos necessários são identificados e lidos dos registradores. A seguir os componentes desse estágio, no qual a MUX AluScr encontra-se omitida na figura 1.

- **Controle**: Gera sinais de controle com base na instrução decodificada;
- **Registradores**: Armazenam os dados a serem utilizados nas operações;
- **Extensão de Sinal**: Estende o imediato para o tamanho correto;
- **Unidade de Detecção de Hazard**: Detecta conflitos de dados e controla o pipeline para evitar riscos;
- **MUX**: Seleciona os operandos corretos a partir dos registradores ou resultados anteriores.
- **Somador**: Utilizado para calcular endereços para instruções de desvio;
- **MUX**: Seleciona o segundo operando da ALU entre o registrador ou um valor imediato;
- **Registrador de pipeline ID/EX**: Armazena informações decodificadas e sinais de controle para posteriormente serem executadas. Possui 256 bits de largura.

3.2.2 EX - Execução

Estágio onde a operação especificada pela instrução é realizada. As unidades de execução do processador realizam a operação aritmética ou lógica nos operandos. Neste estágio também ocorre o *forwarding*, onde o registrador de pipeline ID/EX passa os números dos registradores operando do estágio ID para determinar se os valores devem sofrer *Forwarding*. A seguir, os componentes deste estágio, no qual o Controle da ALU está implícito.

- **ALU (Unidade Lógica Aritmética)**: Realiza operações aritméticas e lógicas nos operandos;

- **MUX**: Selecionam entre diferentes origens dos dados para a ALU;
- **Unidade de Forwarding**: Resolve riscos de dados, encaminhando resultados recentes diretamente para a ALU;
- **Controle da ALU**: Define qual operação será feita na ALU;
- **Registrador de pipeline EX/MEM**: Armazena resultados da ALU, endereços de memória e sinais de controle para o próximo estágio. Possui 193 bits de largura.

The Zero status information from the ALU is used to decide which adder result to store in the PC.

Na tabela 1, podemos visualizar os valores de controle para a Unidade de *Forwarding* que são selecionados pelos multiplexadores.

Controle do Mux	Origem	Explicação
ForwardA = 00	ID/EX	O primeiro operando ALU vem do arquivo de registradores.
ForwardA = 10	EX/MEM	O primeiro operando ALU é encaminhado do resultado ALU anterior.
ForwardA = 01	MEM/WB	O primeiro operando ALU é encaminhado da memória de dados ou de um resultado ALU anterior.
ForwardB = 00	ID/EX	O segundo operando ALU vem do arquivo de registradores.
ForwardB = 10	EX/MEM	O segundo operando ALU é encaminhado do resultado ALU anterior.
ForwardB = 01	MEM/WB	O segundo operando ALU é encaminhado da memória de dados ou de um resultado ALU anterior.

Tabela 1 – Controles de Encaminhamento para Operandos ALU.

3.2.3 MEM - Memória

Se a instrução envolver acesso à memória (leitura ou escrita), este acesso ocorre neste estágio. Dados são lidos da memória ou escritos nela, dependendo da instrução. A seguir, os componentes desse estágio.

- **Memória de Dados**: Lê ou escreve dados na memória RAM;
- **Registrador de pipeline MEM/WB**: Armazena dados lidos da memória ou resultados da ALU para posterior escrita. Possui 128 bits de largura.

3.2.4 WB - Escrita de Resultados

Os resultados da execução são escritos de volta aos registradores. Os resultados são armazenados nos registradores de destino para uso em futuras instruções. A seguir, os componentes desse estágio.

- **MUX**: Seleciona entre os dados da memória ou o resultado da ALU para escrever de volta nos registradores;
- **Registradores**: Atualizados com os novos dados.

4 IMPLEMENTAÇÃO DOS BLOCOS VHDL

Para iniciar o projeto utilizamos da tabela 2, que mostra o código de instrução referentes à todas as operações pedidas.

R-Type					
funct7	funct3	opcode	Instruction	ALUOp	ALUControl
0000000	000	0110011	ADD	10	0000
0100000	000	0110011	SUB	10	0001
0000000	111	0110011	AND	10	0010
0000000	110	0110011	OR	10	0011
0000000	100	0110011	XOR	10	0100
0000000	001	0110011	SLL	10	0101
0000000	101	0110011	SRL	10	0110
I-Type					
imm[11:0]	funct3	opcode	Instruction	ALUOp	ALUControl
imm[11:0]	000	0010011	ADDI	11	0000
imm[11:0]	111	0010011	ANDI	11	0010
imm[11:0]	110	0010011	ORI	11	0011
imm[11:0]	100	0010011	XORI	11	0100
imm[11:0]	001	0010011	SLLI	11	0101
imm[11:0]	101	0010011	SRLI	11	0110
imm[31:12]	-	0110111	LUI	-	-
imm[31:12]	-	0010111	AUIPC	-	-
Load					
imm[11:0]	funct3	opcode	Instruction	ALUOp	ALUControl
imm[11:0]	010	0000011	LW	-	-
Store					
imm[11:5]	funct3	opcode	Instruction	ALUOp	ALUControl
imm[11:5]	010	0100011	SW	-	-
Jump					
imm[20|10:1|11|19:12]	-	opcode	Instruction	ALUOp	ALUControl
imm[20|10:1|11|19:12]	-	1101111	JAL	-	-
imm[11:0]	000	1100111	JALR	-	-
Branch					
imm[12|10:5]	funct3	opcode	Instruction	ALUOp	ALUControl
imm[12|10:5]	000	1100011	BEQ	01	0001
imm[12|10:5]	001	1100011	BNE	01	0001

Tabela 2 – RV32I Base Instruction Set com ALUOp e ALUControl.

A tabela 3 é utilizadas como base para o controle de sinais no estágio de decodificação.

Instrução	**ALUOp**	**ALUSrc**	**Branch**	**Mem-Read**	**Mem-Write**	**Reg-Write**	**Memto-Reg**
R	10	0	0	0	0	1	0
I (Aritmética)	11	1	0	0	0	1	0
UJ (JAL)	00	0	0	0	0	1	0
I (JALR)	00	1	0	0	0	1	0
SB (BEQ, BNE)	01	0	1	0	0	0	0
I (LW)	00	1	0	1	0	1	1
U (LUI)	00	1	0	0	0	1	0
S (SW)	00	1	0	0	1	0	0
U (AUIPC)	00	1	0	0	0	1	0

Tabela 3 – Sinais de Controle combinados para Instruções RISC-V

Para o controle da ULA utilizamos a tabela 4 como padronização.

Operação	**Código de Controle**
ADD	0000
SUB	0001
AND	0010
OR	0011
XOR	0100
SLL	0101
SRL	0110

Tabela 4 – Tabela de Controle da ALU

Para fazer fazer o projeto com SIMD efetuamos apenas algumas mudanças para adicionar dois vetores a mais `vecSize` e `mode`, na tabela 5 estão as explicações sobre ambos os vetores. Na seção 5.2 pode-se encontrar os códigos que sofreram mudanças. Além dos 3 anexados, é necessário também adicionar esses dois vetores novos aos registradores de pipeline (IF/ID, ID/EX, EX/MEM, MEM/WB).

mode	Define se a operação será vetorial ou não (0 $\rightarrow$ operação não vetorial; 1 $\rightarrow$ operação vetorial)
vecSize	Define o tamanho das seções internas aos operandos (00 $\rightarrow$ 4 bits; 01 $\rightarrow$ 8 bits; 10 $\rightarrow$ 16 bits; 32 bits foi deixado de fora pois para ser realizado basta que mode = 0)

Tabela 5 – Explicações sobre os novos sinais de controle.

Na figura 2 podemos encontrar a arquitetura RISC-V (RV32I) implementada e na figura 3 o módulo criado para os registradores.

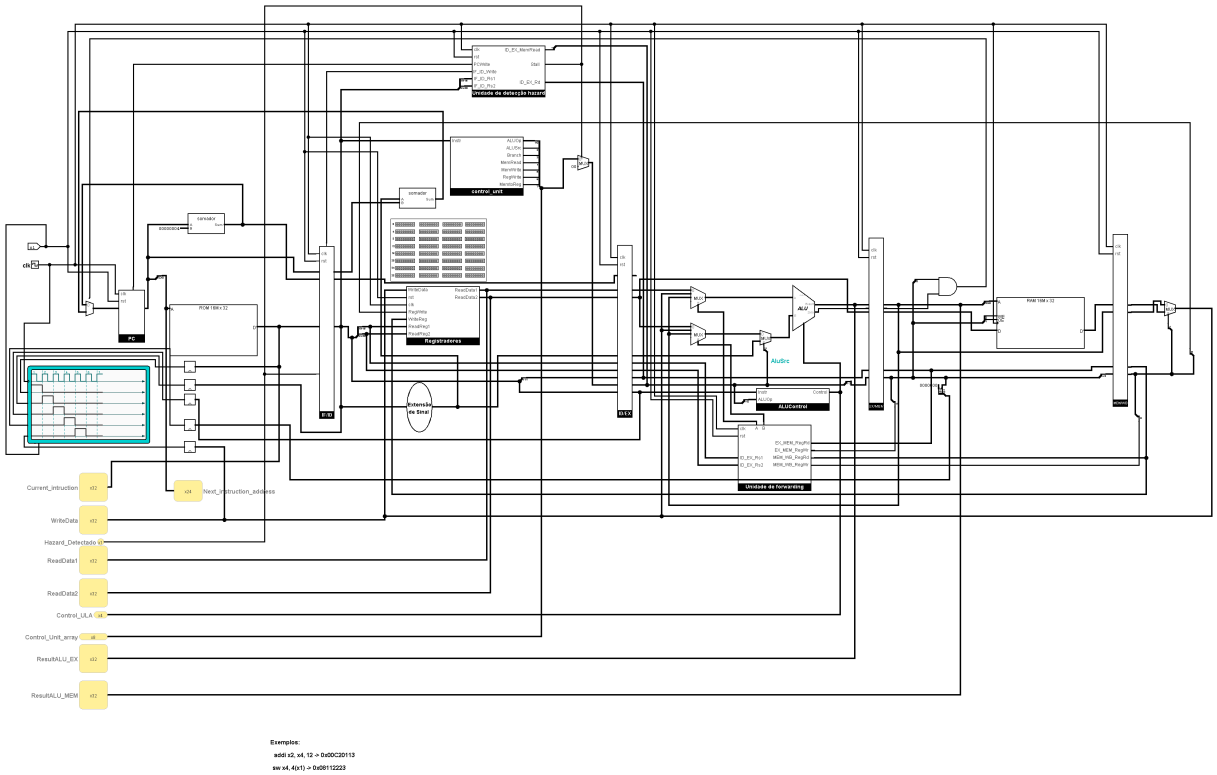

Figura 2 – CPU RISC-V (RV32I)

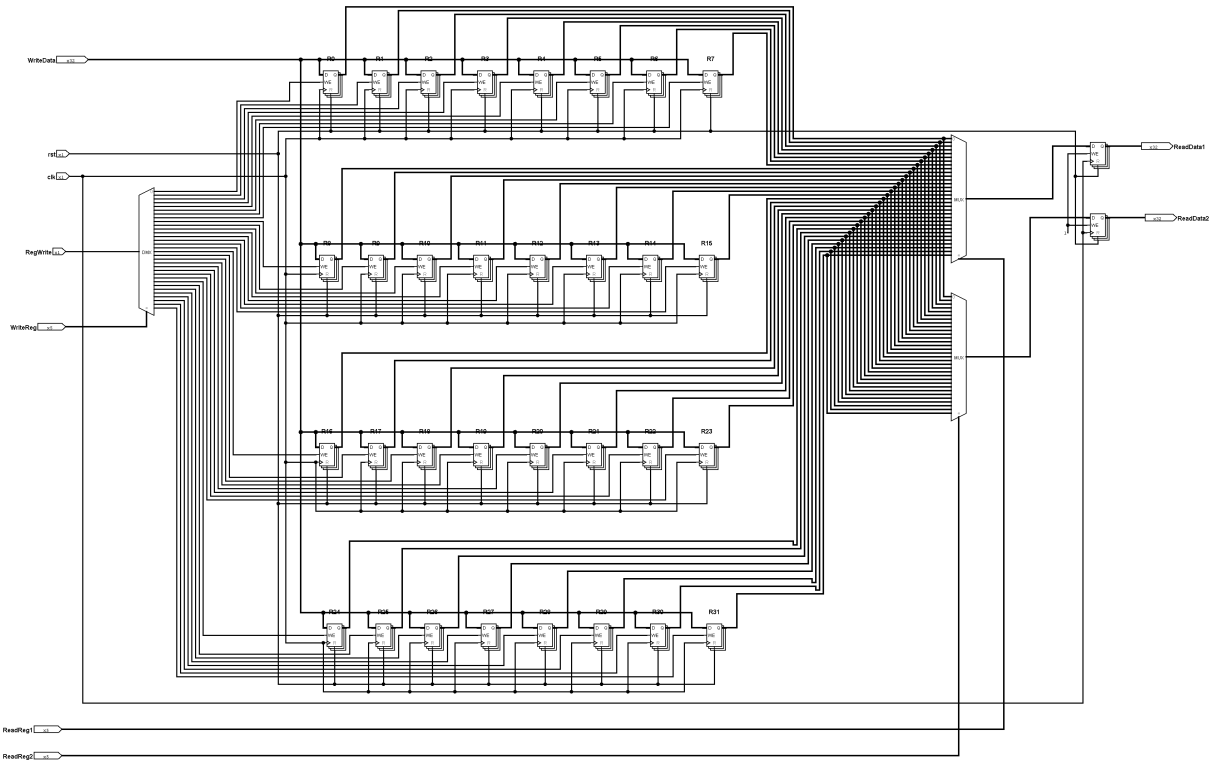

Figura 3 – Registradores.

5 CÓDIGOS E TESTES

Exemplo de cabeçalho feito em todos os códigos.

```
1  ----------------------------------------------------------------------------------
2  -- Projeto: CPU RISC-V DE 32 BITS: DESIGN E IMPLEMENTACAO COM PIPELINE DE
   ↪  5 ESTAGIOS
3  -- Arquivo: ULA
4  -- Autores: Dhayse Tito,
5  --                    Eduardo Guimaraes Fr.,
6  --                    Joao Pedro Baptista,
7  --                    Ligia Calina Bonifacio e
8  --                    Luiza Lissandra Rosa
9  ----------------------------------------------------------------------------------
10 -- Descricao : Unidade Logica e Aritmetica - ULA
11 ----------------------------------------------------------------------------------
12
```

5.1 CÓDIGOS PARA RV32I

5.1.1 ALU

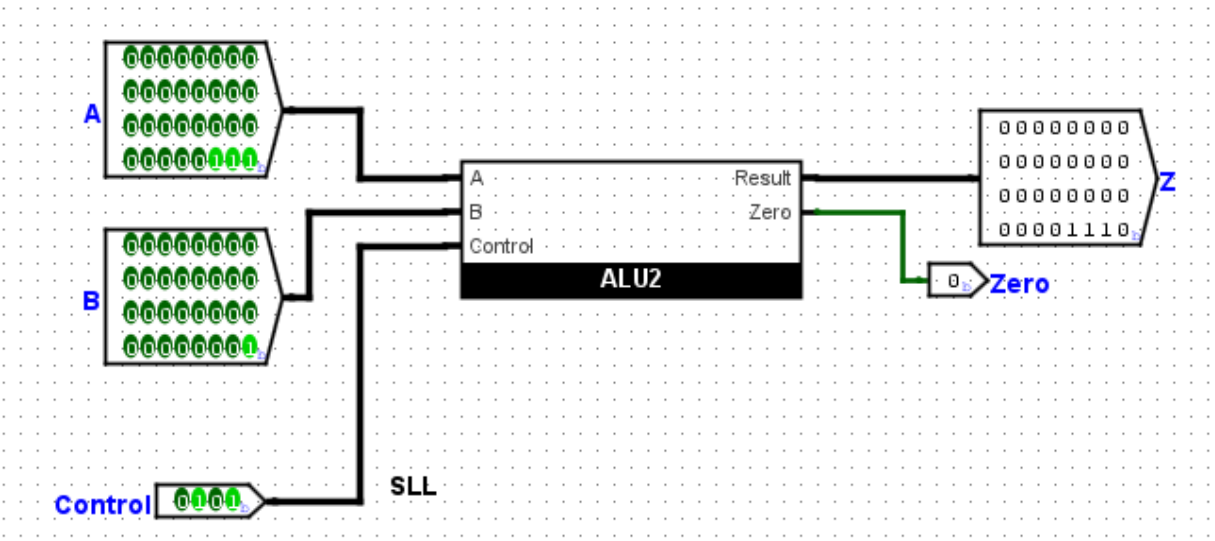

Figura 4 – ALU.

```
12
13 -- Importacoes necessarias
14 library IEEE;
15 use IEEE.STD_LOGIC_1164.ALL;
16 use ieee.numeric_std.ALL;
```

```
17 USE IEEE.std_logic_unsigned.ALL;
18
19 -- Definicao da entidade ALU
20 entity ALU2 is
21     Port ( A, B       : in  STD_LOGIC_VECTOR (31 downto 0);
22            ALUControl : in  STD_LOGIC_VECTOR (3 downto 0);
23            Result     : out STD_LOGIC_VECTOR (31 downto 0);
24            Zero       : out STD_LOGIC);
25 end ALU2;
26
27 -- Definicao da arquitetura da ALU
28 architecture Behavioral of ALU2 is
29
30 signal Result_aux : std_logic_vector(31 downto 0);
31 signal Zero_aux   : std_logic_vector(32 downto 0);
32
33 -- Implementacao da logica da ALU
34 begin
35     process(A, B, ALUControl)
36     begin
37         case ALUControl is
38             when "0000" =>  -- ADD
39                 Result_aux <= std_logic_vector(unsigned(A) +
                    ↪ unsigned(B));
40             when "0001" =>  -- SUB
41                 Result_aux <= std_logic_vector(unsigned(A) -
                    ↪ unsigned(B));
42             when "0010" =>  -- AND
43                 Result_aux <= A and B;
44             when "0011" =>  -- OR
45                 Result_aux <= A or B;
46             when "0100" =>  -- XOR
47                 Result_aux <= A xor B;
48             when "0101" =>  -- SLL
49                 Result_aux <= std_logic_vector(shift_left(unsigned(A),
                    ↪ to_integer(unsigned(B(4 downto 0)))));
50             when "0110" =>  -- SRL
51                 Result_aux <= std_logic_vector(shift_right(unsigned(A),
                    ↪ to_integer(unsigned(B(4 downto 0)))));
```

```
52              when others =>
53                  Result_aux <= (others => '0');
54          end case;
55
56      end process;
57
58      -- Atribuir valores aos sinais de saida
59      Result <= Result_aux;
60
61          -- Inicializacao do sinal Zero_aux
62          Zero_aux(0) <= '0';
63
64          -- Geracao dos valores Zero_aux
65          G2: for I in 1 to 32 generate
66              Zero_aux(I) <= Zero_aux(I - 1) or Result_aux(I - 1);
67          end generate;
68
69          -- Definicao final do sinal zero
70          Zero <= not Zero_aux(32);
71
72  end Behavioral;
```

5.1.2 Controle ALU

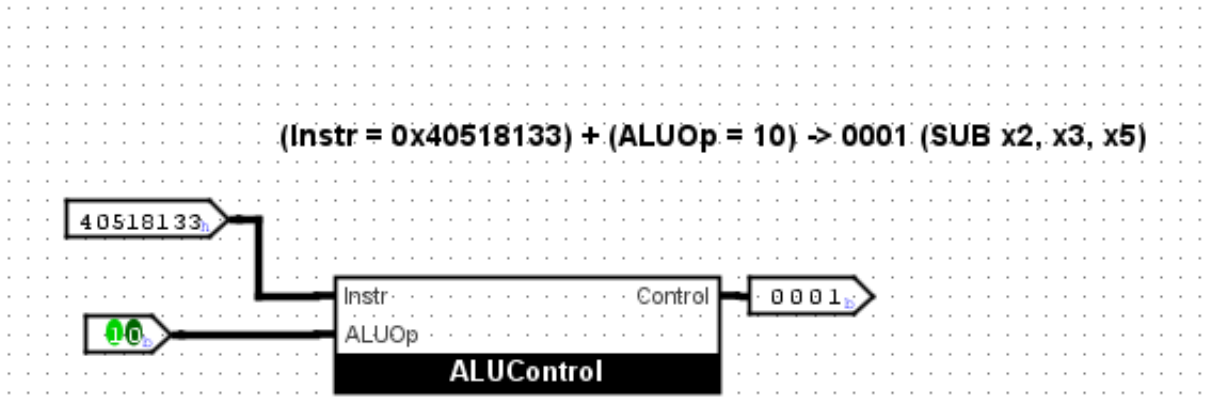

Figura 5 – ALUControl.

```
12
13  -- Importacoes necessarias
14  library IEEE;
15  use IEEE.STD_LOGIC_1164.ALL;
16  use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
17
18 -- Definicao da entidade Controle da ALU
19 entity ALUControl is
20     Port ( Instr      : in STD_LOGIC_VECTOR (31 downto 0);
21            ALUOp      : in  STD_LOGIC_VECTOR (1 downto 0);
22            Control    : out STD_LOGIC_VECTOR (3 downto 0)
23          );
24 end ALUControl;
25
26 -- Definicao da arquitetura da ALU
27 architecture Behavioral of ALUControl is
28
29 -- Definicao de sinal auxiliar
30 signal Control_aux  : STD_LOGIC_VECTOR (3 downto 0);
31 signal aux1         : STD_LOGIC;  -- Sinal para armazenar Instr[30]
32 signal aux3         : STD_LOGIC_VECTOR(2 downto 0);  -- Sinal para
   ↪  armazenar Instr[14-12]
33
34 -- Implementacao da logica do Controle da ALU
35 begin
36
37     aux1  <= Instr(30); -- Extrai func7 da instrução
38     aux3  <= Instr(14 downto 12); -- Extrai func3 da instrução
39     process(aux1, aux3, ALUOp)
40     begin
41         case ALUOp is
42             when "00" =>  -- LW, SW
43                 Control_aux <= "0000"; -- igual ADD
44             when "01" =>  -- BEQ, BNE
45                 Control_aux <= "0001"; -- igual SUB
46             when "10" =>  -- R-Type
47                 case aux3 is
48                     when "000" =>  -- ADD, SUB
49                         if aux1 = '0' then
50                             Control_aux <= "0000";  -- ADD
51                         elsif aux1 = '1' then
52                             Control_aux <= "0001";  -- SUB
53                         end if;
54                     when "111" =>  -- AND
```

```
55                          Control_aux <= "0010";
56                      when "110" =>  -- OR
57                          Control_aux <= "0011";
58                      when "100" =>  -- XOR
59                          Control_aux <= "0100";
60                      when "001" =>  -- SLL
61                          Control_aux <= "0101";
62                      when "101" =>  -- SRL
63                          Control_aux <= "0110";
64                      when others =>
65                          Control_aux <= (others => '0');
66                  end case;
67              when "11" =>  -- I-Type
68                  case aux3 is
69                      when "000" =>  -- ADDI
70                          Control_aux <= "0000";
71                      when "111" =>  -- ANDI
72                          Control_aux <= "0010";
73                      when "110" =>  -- ORI
74                          Control_aux <= "0011";
75                      when "100" =>  -- XORI
76                          Control_aux <= "0100";
77                      when "001" =>  -- SLLI
78                          Control_aux <= "0101";
79                      when "101" =>  -- SRLI
80                          Control_aux <= "0110";
81                      when others =>
82                          Control_aux <= (others => '0');
83                  end case;
84              when others =>
85                  Control_aux <= (others => '0');
86          end case;
87      end process;
88
89          Control <= Control_aux;
90
91  end Behavioral;
```

5.1.3 Controle

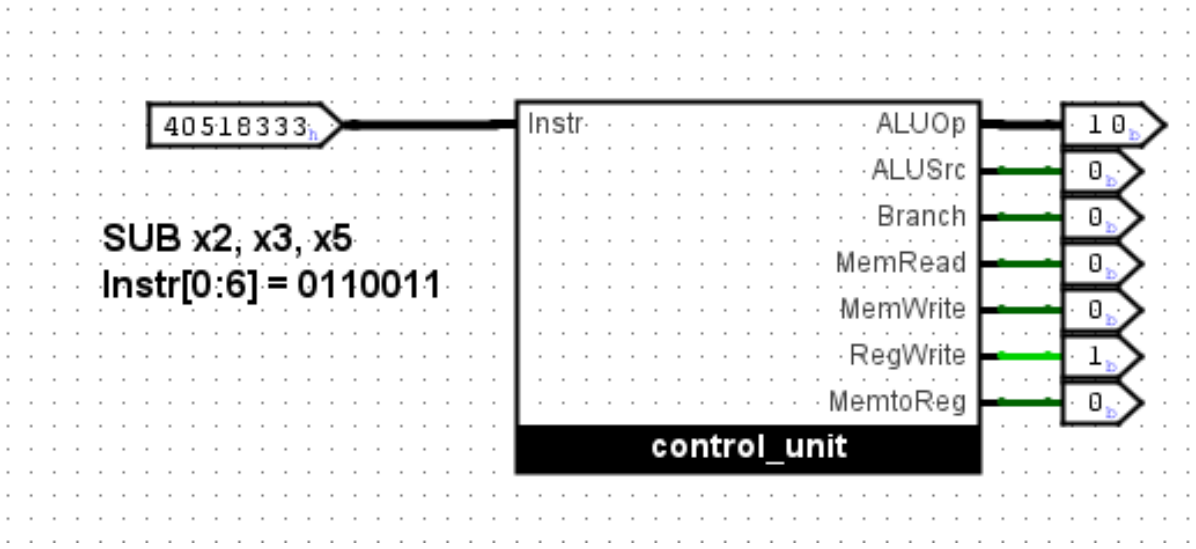

Figura 6 – Unidade de controle.

```
12
13  -- Importacoes necessarias
14  library IEEE;
15  use IEEE.STD_LOGIC_1164.ALL;
16  use IEEE.NUMERIC_STD.ALL;
17
18  -- Definição da entidade da unidade de controle
19  entity control_unit is
20      Port(
21          Instr     : in std_logic_vector(31 downto 0);   -- Entrada:
            ↪  instrução de 32 bits
22          ALUOp     : out std_logic_vector(1 downto 0);   -- EX
23          ALUSrc    : out std_logic;
24          Branch    : out std_logic;                      -- MEM
25          MemRead   : out std_logic;                      -- MEM
26          MemWrite  : out std_logic;                      -- MEM
27          RegWrite  : out std_logic;                      -- WB
28          MemtoReg  : out std_logic                       -- WB
29      );
30  end control_unit;
31
32  -- Definição da arquitetura da unidade de controle
33  architecture Behavioral of control_unit is
34
```

```
35      -- Sinais auxiliares de controle
36      signal opcode       : std_logic_vector(6 downto 0);
37      signal ALUOp_aux    : std_logic_vector(1 downto 0); -- EX
38      signal ALUSrc_aux   : std_logic; -- EX
39      signal Branch_aux   : std_logic; -- MEM
40      signal MemRead_aux  : std_logic; -- MEM
41      signal MemWrite_aux : std_logic; -- MEM
42      signal RegWrite_aux : std_logic; -- WB
43      signal MemtoReg_aux : std_logic; -- WB
44
45  begin
46      opcode <= Instr(6 downto 0);  -- Extrai o opcode da instrução
47
48      process(opcode)
49      begin
50          -- Valores padrão para sinais de controle
51          ALUOp_aux    <= "00";
52          ALUSrc_aux   <= '0';
53          Branch_aux   <= '0';
54          MemRead_aux  <= '0';
55          MemWrite_aux <= '0';
56          RegWrite_aux <= '0';
57          MemtoReg_aux <= '0';
58
59          case opcode is
60              when "0110011"  => -- R-Type
61                  ALUOp_aux    <= "10";
62                  ALUSrc_aux   <= '0';
63                  RegWrite_aux <= '1';
64
65              when "0010011" => -- I-Type Arith
66                      ALUSrc_aux      <= '1';
67                      RegWrite_aux    <= '1';
68                      ALUOp_aux       <= "11";
69
70              when "0000011" => -- I-Type (LW)
71                  ALUOp_aux    <= "00";
72                  ALUSrc_aux   <= '1';
73                  MemRead_aux  <= '1';
```

```
74                  MemtoReg_aux <= '1';
75                  RegWrite_aux <= '1';
76
77              when "0100011" => -- S-Type (SW)
78                  ALUOp_aux    <= "00";
79                  ALUSrc_aux   <= '1';
80                  MemWrite_aux <= '1';
81
82              when "1100011" => -- SB-Type (BEQ/BNE)
83                  ALUOp_aux    <= "01";
84                  ALUSrc_aux   <= '0';
85                  Branch_aux   <= '1';
86
87              when "1101111" => -- JAL
88                  ALUOp_aux    <= "00";  -- Não importa para JAL
89                  ALUSrc_aux   <= '1';   -- A origem do dado para a ALU é
                    ↪ um imediato
90                  RegWrite_aux <= '1';   -- Escreve no registrador
91
92              when "1100111" => -- JALR
93                  ALUOp_aux    <= "00";  -- Não importa para JALR
94                  ALUSrc_aux   <= '1';   -- A origem do dado para a ALU é
                    ↪ um imediato
95                  RegWrite_aux <= '1';   -- Escreve no registrador
96
97              when "0110111" => -- LUI
98                  ALUOp_aux    <= "00";  -- Não importa para LUI
99                  ALUSrc_aux   <= '1';   -- A origem do dado para a ALU é
                    ↪ um imediato
100                 RegWrite_aux <= '1';   -- Escreve no registrador
```

5.1.4 Extensão de Sinal

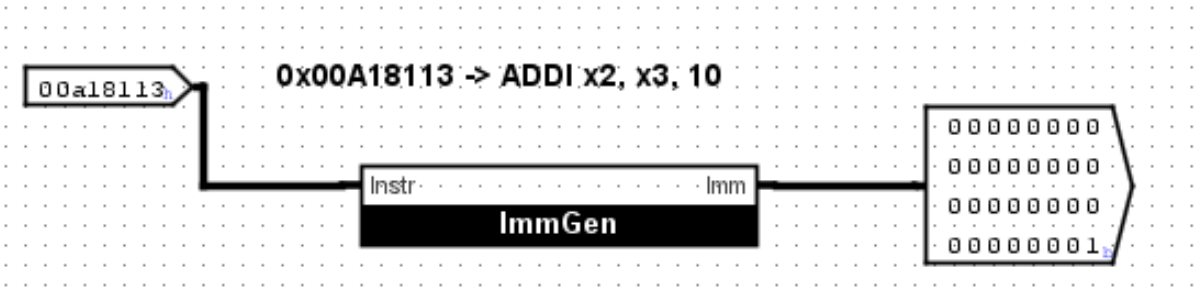

Figura 7 – Extensão de sinal.

```
12
13 library IEEE;
14 use IEEE.STD_LOGIC_1164.ALL;
15 USE IEEE.NUMERIC_STD.ALL;
16
17 entity ImmGen is
18     Port(
19         Instr: in std_logic_vector(31 downto 0);  -- Entrada: instrução
           ↪  de 32 bits
20         Imm: out std_logic_vector(31 downto 0)    -- Saída: imediato de 32
           ↪  bits
21     );
22 end ImmGen;
23
24 architecture Behavioral of ImmGen is
25     signal opcode       : std_logic_vector(6 downto 0);  -- Sinal para
       ↪  armazenar o opcode da instrução
26     signal Imm_aux      : std_logic_vector(31 downto 0) := (others =>
       ↪  '0'); -- Inicializa com '0'
27
28 begin
29
30     opcode <= Instr(6 downto 0);  -- Extrai o opcode da instrução
31
32     -- Processo para determinar o imediato baseado no opcode
33     process(opcode)
34     begin
35         case opcode is
```

```
36              when "0010011" | "0000011" => -- I-type (ADDI, ANDI, ORI,
                ↪ XORI, SLLI, SRLI) | (LW)
37                  Imm_aux(11 downto 0)  <= Instr(31 downto 20);  -- Campo
                    ↪ imediato
38                  Imm_aux(31 downto 12) <= (others => Instr(31)); --
                    ↪ Extensão de sinal
39
40              when "1100111" => -- I-type (JALR)
41                  Imm_aux(11 downto 0)  <= Instr(31 downto 20);  -- Campo
                    ↪ imediato
42                  Imm_aux(31 downto 12) <= (others => Instr(31)); --
                    ↪ Extensão de sinal
43
44              when "0100011" => -- S-type (SW)
45                  Imm_aux(11 downto 5)  <= Instr(31 downto 25);  -- Parte
                    ↪ alta do campo imediato
46                  Imm_aux(4 downto 0)   <= Instr(11 downto 7);   -- Parte
                    ↪ baixa do campo imediato
47                  Imm_aux(31 downto 12) <= (others => Instr(31)); --
                    ↪ Extensão de sinal
48
49              when "1100011" => -- SB-type (BEQ, BNE)
50                  Imm_aux(12)           <= Instr(31);           -- Bit de
                    ↪ sinal para o campo imediato
51                  Imm_aux(10 downto 5)  <= Instr(30 downto 25); -- Bits
                    ↪ 10-5 para o campo imediato
52                  Imm_aux(4 downto 1)   <= Instr(11 downto 8);  -- Bits
                    ↪ 4-1 para o campo imediato
53                  Imm_aux(11)           <= Instr(7);            -- Bit 11
                    ↪ para o campo imediato
54                  Imm_aux(31 downto 13) <= (others => Instr(31)); --
                    ↪ Extensão de sinal
55
56              when "1101111" => -- UJ-type (JAL)
57                  Imm_aux(20)           <= Instr(31);           -- Bit 20
                    ↪ para o campo imediato
58                  Imm_aux(10 downto 1)  <= Instr(30 downto 21); -- Bits
                    ↪ 10-1 para o campo imediato
```

```
59                  Imm_aux(11)            <= Instr(20);           -- Bit 11
                    ↪ para o campo imediato
60                  Imm_aux(19 downto 12)  <= Instr(19 downto 12); -- Bits
                    ↪ 19-12 para o campo imediato
61                  Imm_aux(31 downto 21)  <= (others => Instr(31)); --
                    ↪ Extensão de sinal
62
63              when "0110111" | "0010111" => -- U-Type (LUI) | (AUIPC)
64                  Imm_aux(31 downto 12) <= Instr(31 downto 12); -- Campo
                    ↪ imediato
65                  Imm_aux(11 downto 0)  <= (others => '0');      -- Zero
                    ↪ padding
66
67              when others => -- Default para opcode não reconhecido
68                  Imm_aux <= (others => '0'); -- Zera o imediato se o
                    ↪ opcode não for reconhecido
69          end case;
70      end process;
71
72      -- Saída do imediato
73      Imm <= Imm_aux;
74
75  end Behavioral;
76
```

5.1.5 PC

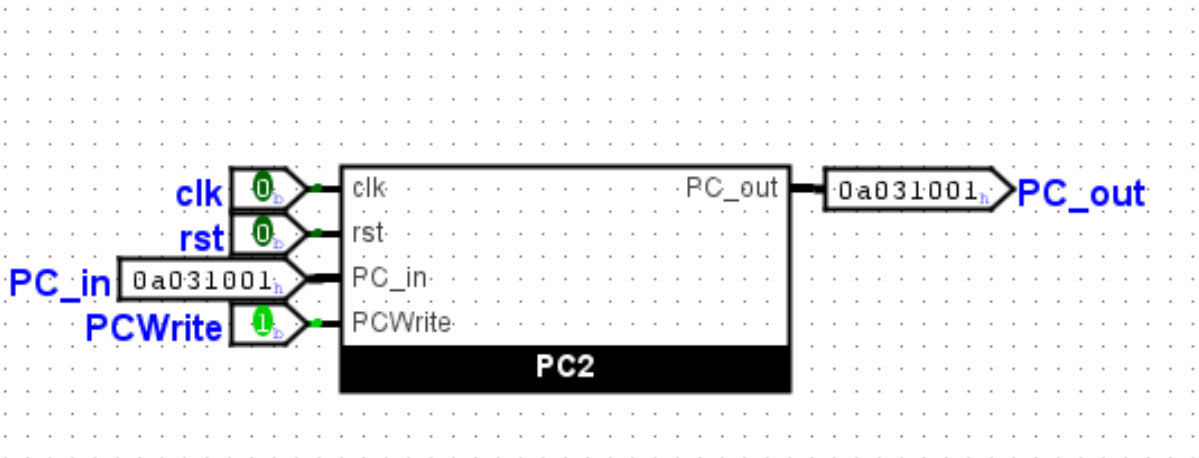

Figura 8 – Contador.

```
12
13 -- Importacoes necessarias
14 library IEEE;
15 use IEEE.STD_LOGIC_1164.ALL;
16 use IEEE.STD_LOGIC_ARITH.ALL;
17 use IEEE.STD_LOGIC_UNSIGNED.ALL;
18
19 -- Definicao da entidade PC
20 entity PC2 is
21     Port ( clk    : in  STD_LOGIC;
22                  rst    : in  STD_LOGIC;
23                  PC_in  : in  STD_LOGIC_VECTOR (31 downto 0);
24                  PCWrite: in STD_LOGIC;
25                  PC_out : out STD_LOGIC_VECTOR (31 downto 0)
26                 );
27 end PC2;
28
29 -- Definicao da arquitetura do PC
30 architecture Behavioral of PC2 is
31
32 -- Definicao de sinal
33 signal PC_reg : STD_LOGIC_VECTOR (31 downto 0);
34
35 -- Implementacao da logica do PC
36 begin
37     process(clk, rst)
38
39     begin
40         if rst = '1' then
41             PC_reg <= (others => '0');
42         elsif rising_edge(clk) then
43                   if PCWrite = '1' then
44                     PC_reg <= PC_in;
45             end if;
46         end if;
47
48     end process;
49
50     PC_out <= PC_reg;
```

```
51
52 end Behavioral;
```

5.1.6 Somador

```
12
13 -- Importacoes necessarias
14 library IEEE;
15 use IEEE.STD_LOGIC_1164.ALL;
16 use IEEE.STD_LOGIC_ARITH.ALL;
17 use IEEE.STD_LOGIC_UNSIGNED.ALL;
18
19 -- Definicao da entidade somador
20 entity somador is
21     Port ( A   : in  STD_LOGIC_VECTOR (31 downto 0);
22            B   : in  STD_LOGIC_VECTOR (31 downto 0);
23            Sum : out STD_LOGIC_VECTOR (31 downto 0) );
24 end somador;
25
26 -- Definicao da arquitetura do somador
27 architecture Behavioral of somador is
28
29 -- Implementacao da logica do somador
30 begin
31
32     Sum <= A + B;
33
34 end Behavioral;
```

5.1.7 IF/ID

```
12
13 library IEEE;
14 use IEEE.STD_LOGIC_1164.ALL;
15 use IEEE.STD_LOGIC_ARITH.ALL;
16 use IEEE.STD_LOGIC_UNSIGNED.ALL;
17
18 entity IF_ID_Pipe is
19     Port (
20         clk             : in  STD_LOGIC;
21         rst             : in  STD_LOGIC;
```

```
22          IF_ID_Write     : in  STD_LOGIC; -- Forward
23          IF_ID_Flush     : in  STD_LOGIC; -- Hazard
24          PC_in           : in  STD_LOGIC_VECTOR(31 downto 0);
25          PC_4_in         : in  STD_LOGIC_VECTOR(31 downto 0);
26          Instr_in        : in  STD_LOGIC_VECTOR(31 downto 0);
27          PC_out          : out STD_LOGIC_VECTOR(31 downto 0);
28          PC_4_out        : out STD_LOGIC_VECTOR(31 downto 0);
29          Instr_out       : out STD_LOGIC_VECTOR(31 downto 0)
30      );
31  end IF_ID_Pipe;
32
33  architecture Behavioral of IF_ID_Pipe is
34      signal PC_reg       : STD_LOGIC_VECTOR(31 downto 0);
35      signal PC_4_reg     : STD_LOGIC_VECTOR(31 downto 0);
36      signal Instr_reg    : STD_LOGIC_VECTOR(31 downto 0);
37  begin
38      process(clk, rst)
39      begin
40          if rst = '1' then
41              PC_reg <= (others => '0');
42              PC_4_reg <= (others => '0');
43              Instr_reg <= (others => '0');
44          elsif rising_edge(clk) then
45              if IF_ID_Flush = '1' then
46                  PC_reg <= (others => '0');
47                  PC_4_reg <= (others => '0');
48                  Instr_reg <= (others => '0');
49              elsif IF_ID_Write = '1' then
50                  PC_reg <= PC_in;
51                  PC_4_reg <= PC_4_in;
52                  Instr_reg <= Instr_in;
53              end if;
54          end if;
55      end process;
56
57      PC_out <= PC_reg;
58      PC_4_out <= PC_4_reg;
59      Instr_out <= Instr_reg;
60  end Behavioral;
```

5.1.8 ID/EX

```
61
12
13 library IEEE;
14 use IEEE.STD_LOGIC_1164.ALL;
15 use IEEE.STD_LOGIC_ARITH.ALL;
16 use IEEE.STD_LOGIC_UNSIGNED.ALL;
17
18 entity ID_EX_Pipe is
19     Port (
20         clk             : in  STD_LOGIC;
21         rst             : in  STD_LOGIC;
22         -- sinais de entrada
23         PC_in           : in  STD_LOGIC_VECTOR(31 downto 0);
24         ReadData1_in    : in  STD_LOGIC_VECTOR(31 downto 0);
25         ReadData2_in    : in  STD_LOGIC_VECTOR(31 downto 0);
26         SignExtend_in   : in  STD_LOGIC_VECTOR(31 downto 0);
27         Rs1_in          : in  STD_LOGIC_VECTOR(4 downto 0);
28         Rs2_in          : in  STD_LOGIC_VECTOR(4 downto 0);
29         Rd_in           : in  STD_LOGIC_VECTOR(4 downto 0);
30         Control_in      : in  STD_LOGIC_VECTOR(7 downto 0);
31         Instr_in            : in  STD_LOGIC_VECTOR(31 downto 0);
32         -- sinais de saida
33         PC_out          : out STD_LOGIC_VECTOR(31 downto 0);
34         ReadData1_out   : out STD_LOGIC_VECTOR(31 downto 0);
35         ReadData2_out   : out STD_LOGIC_VECTOR(31 downto 0);
36         SignExtend_out  : out STD_LOGIC_VECTOR(31 downto 0);
37         Rs1_out         : out STD_LOGIC_VECTOR(4 downto 0);
38         Rs2_out         : out STD_LOGIC_VECTOR(4 downto 0);
39         Rd_out          : out STD_LOGIC_VECTOR(4 downto 0);
40         Control_out     : out STD_LOGIC_VECTOR(7 downto 0);
41         Instr_out           : out  STD_LOGIC_VECTOR(31 downto 0)
42     );
43 end ID_EX_Pipe;
44
45 architecture Behavioral of ID_EX_Pipe is
46     signal PC_reg           : STD_LOGIC_VECTOR(31 downto 0);
47     signal ReadData1_reg    : STD_LOGIC_VECTOR(31 downto 0);
```

```
48      signal ReadData2_reg    : STD_LOGIC_VECTOR(31 downto 0);
49      signal SignExtend_reg   : STD_LOGIC_VECTOR(31 downto 0);
50      signal Rs1_reg          : STD_LOGIC_VECTOR(4 downto 0);
51      signal Rs2_reg          : STD_LOGIC_VECTOR(4 downto 0);
52      signal Rd_reg           : STD_LOGIC_VECTOR(4 downto 0);
53      signal Control_reg      : STD_LOGIC_VECTOR(7 downto 0);
54      signal Instr_reg          : STD_LOGIC_VECTOR(31 downto 0);
55  begin
56      process(clk, rst)
57      begin
58          if rst = '1' then
59              PC_reg <= (others => '0');
60              ReadData1_reg <= (others => '0');
61              ReadData2_reg <= (others => '0');
62              SignExtend_reg <= (others => '0');
63              Rs1_reg <= (others => '0');
64              Rs2_reg <= (others => '0');
65              Rd_reg <= (others => '0');
66              Control_reg <= (others => '0');
67              Instr_reg <= (others => '0');
68          elsif rising_edge(clk) then
69              PC_reg <= PC_in;
70              ReadData1_reg <= ReadData1_in;
71              ReadData2_reg <= ReadData2_in;
72              SignExtend_reg <= SignExtend_in;
73              Rs1_reg <= Rs1_in;
74              Rs2_reg <= Rs2_in;
75              Rd_reg <= Rd_in;
76              Control_reg <= Control_in;
77              Instr_reg <= Instr_in;
78          end if;
79      end process;
80
81      PC_out <= PC_reg;
82      ReadData1_out <= ReadData1_reg;
83      ReadData2_out <= ReadData2_reg;
84      SignExtend_out <= SignExtend_reg;
85      Rs1_out <= Rs1_reg;
86      Rs2_out <= Rs2_reg;
```

```
87      Rd_out <= Rd_reg;
88      Control_out <= Control_reg;
89      Instr_out <= Instr_reg;
90  end Behavioral;
```

5.1.9 EX/MEM

```
12
13  library IEEE;
14  use IEEE.STD_LOGIC_1164.ALL;
15  use IEEE.STD_LOGIC_ARITH.ALL;
16  use IEEE.STD_LOGIC_UNSIGNED.ALL;
17
18  entity EX_MEM_Pipe is
19      Port (
20          clk             : in  STD_LOGIC;
21          rst             : in  STD_LOGIC;
22          ALUResult_in    : in  STD_LOGIC_VECTOR(31 downto 0);
23          zero_in         : in  STD_LOGIC;
24          WriteData_in    : in  STD_LOGIC_VECTOR(31 downto 0);
25          Rd_in           : in  STD_LOGIC_VECTOR(4 downto 0);
26          Control_in      : in  STD_LOGIC_VECTOR(4 downto 0);
27          ALUResult_out   : out STD_LOGIC_VECTOR(31 downto 0);
28          zero_out        : out STD_LOGIC;
29          WriteData_out   : out STD_LOGIC_VECTOR(31 downto 0);
30          Rd_out          : out STD_LOGIC_VECTOR(4 downto 0);
31          Control_out     : out STD_LOGIC_VECTOR(4 downto 0)
32      );
33  end EX_MEM_Pipe;
34
35  architecture Behavioral of EX_MEM_Pipe is
36      signal ALUResult_reg   : STD_LOGIC_VECTOR(31 downto 0);
37      signal zero_reg        : STD_LOGIC;
38      signal WriteData_reg   : STD_LOGIC_VECTOR(31 downto 0);
39      signal Rd_reg          : STD_LOGIC_VECTOR(4 downto 0);
40      signal Control_reg     : STD_LOGIC_VECTOR(4 downto 0);
41  begin
42      process(clk, rst)
43      begin
44          if rst = '1' then
```

```
45              ALUResult_reg <= (others => '0');
46              WriteData_reg <= (others => '0');
47              Rd_reg <= (others => '0');
48              Control_reg <= (others => '0');
49              zero_reg <= '0';
50          elsif rising_edge(clk) then
51              ALUResult_reg <= ALUResult_in;
52              WriteData_reg <= WriteData_in;
53              Rd_reg <= Rd_in;
54              Control_reg <= Control_in;
55              zero_reg <= zero_in;
56          end if;
57      end process;
58
59      ALUResult_out <= ALUResult_reg;
60      WriteData_out <= WriteData_reg;
61      Rd_out <= Rd_reg;
62      Control_out <= Control_reg;
63      zero_out <= zero_reg;
64  end Behavioral;
65
```

5.1.10 MEM/WB

```
12
13
14  library IEEE;
15  use IEEE.STD_LOGIC_1164.ALL;
16  use IEEE.STD_LOGIC_ARITH.ALL;
17  use IEEE.STD_LOGIC_UNSIGNED.ALL;
18
19  entity MEM_WB_Pipe is
20      Port (
21          clk             : in  STD_LOGIC;
22          rst             : in  STD_LOGIC;
23          ReadData_in     : in  STD_LOGIC_VECTOR(31 downto 0);
24          ALUResult_in    : in  STD_LOGIC_VECTOR(31 downto 0);
25          Rd_in           : in  STD_LOGIC_VECTOR(4 downto 0);
26          Control_in      : in  STD_LOGIC_VECTOR(1 downto 0);
27          ReadData_out    : out STD_LOGIC_VECTOR(31 downto 0);
```

```
28          ALUResult_out    : out STD_LOGIC_VECTOR(31 downto 0);
29          Rd_out           : out STD_LOGIC_VECTOR(4 downto 0);
30          Control_out      : out STD_LOGIC_VECTOR(1 downto 0)
31      );
32  end MEM_WB_Pipe;
33
34  architecture Behavioral of MEM_WB_Pipe is
35      signal ReadData_reg    : STD_LOGIC_VECTOR(31 downto 0);
36      signal ALUResult_reg   : STD_LOGIC_VECTOR(31 downto 0);
37      signal Rd_reg          : STD_LOGIC_VECTOR(4 downto 0);
38      signal Control_reg     : STD_LOGIC_VECTOR(1 downto 0);
39  begin
40      process(clk, rst)
41      begin
42          if rst = '1' then
43              ReadData_reg <= (others => '0');
44              ALUResult_reg <= (others => '0');
45              Rd_reg <= (others => '0');
46              Control_reg <= (others => '0');
47          elsif rising_edge(clk) then
48              ReadData_reg <= ReadData_in;
49              ALUResult_reg <= ALUResult_in;
50              Rd_reg <= Rd_in;
51              Control_reg <= Control_in;
52          end if;
53      end process;
54
55      ReadData_out <= ReadData_reg;
56      ALUResult_out <= ALUResult_reg;
57      Rd_out <= Rd_reg;
58      Control_out <= Control_reg;
59  end Behavioral;
60
```

5.2 CÓDIGOS PARA SIMD

5.2.1 ALU

```
12          vecSize     : in  STD_LOGIC_VECTOR (1 downto 0);  -- 2 bits,
            ↪ tamanho da seção interna
13          Result      : out STD_LOGIC_VECTOR (31 downto 0);
14          Zero        : out STD_LOGIC
15      );
16  end ALU2;
17
18  -- Definicao da arquitetura da ALU
19  architecture Behavioral of ALU2 is
20
21  signal Result_aux : std_logic_vector(31 downto 0);
22  signal Zero_aux   : std_logic_vector(32 downto 0);
23
24  -- Função auxiliar para operações vetoriais
25  function vec_op (
26      A, B : std_logic_vector;
27      op   : string;
28      size : integer
29  ) return std_logic_vector is
30      variable Res : std_logic_vector(A'range);
31  begin
32      for i in 0 to A'length/size - 1 loop
33          case op is
34              when "ADD" =>
35                  Res((i+1)*size-1 downto i*size) :=
                    ↪ std_logic_vector(unsigned(A((i+1)*size-1 downto
                    ↪ i*size)) + unsigned(B((i+1)*size-1 downto i*size)));
36              when "SUB" =>
37                  Res((i+1)*size-1 downto i*size) :=
                    ↪ std_logic_vector(unsigned(A((i+1)*size-1 downto
                    ↪ i*size)) - unsigned(B((i+1)*size-1 downto i*size)));
38              when "AND" =>
39                  Res((i+1)*size-1 downto i*size) := A((i+1)*size-1 downto
                    ↪ i*size) and B((i+1)*size-1 downto i*size);
40              when "OR" =>
```

```
41                  Res((i+1)*size-1 downto i*size) := A((i+1)*size-1 downto
                    ↪  i*size) or B((i+1)*size-1 downto i*size);
42              when "XOR" =>
43                  Res((i+1)*size-1 downto i*size) := A((i+1)*size-1 downto
                    ↪  i*size) xor B((i+1)*size-1 downto i*size);
44              when "SLL" =>
45                  Res((i+1)*size-1 downto i*size) :=
                    ↪  std_logic_vector(shift_left(unsigned(A((i+1)*size-1
                    ↪  downto i*size)), to_integer(unsigned(B(4 downto
                    ↪  0)))));
46              when "SRL" =>
47                  Res((i+1)*size-1 downto i*size) :=
                    ↪  std_logic_vector(shift_right(unsigned(A((i+1)*size-1
                    ↪  downto i*size)), to_integer(unsigned(B(4 downto
                    ↪  0)))));
48              when others =>
49                  Res((i+1)*size-1 downto i*size) := (others => '0');
50          end case;
51      end loop;
52      return Res;
53  end function;
54
55  -- Implementacao da logica da ALU
56  begin
57      process(A, B, Control, mode, vecSize)
58          variable size : integer;
59      begin
60          -- Determinar o tamanho da seção interna baseado em vecSize
61          case vecSize is
62              when "00" => size := 4;
63              when "01" => size := 8;
64              when "10" => size := 16;
65              when others => size := 32;
66          end case;
67
68          if mode = '1' then  -- Operação vetorial
69              case Control is
70                  when "0000" =>  -- ADD
71                      Result_aux <= vec_op(A, B, "ADD", size);
```

```
72                  when "0001" =>  -- SUB
73                      Result_aux <= vec_op(A, B, "SUB", size);
74                  when "0010" =>  -- AND
75                      Result_aux <= vec_op(A, B, "AND", size);
76                  when "0011" =>  -- OR
77                      Result_aux <= vec_op(A, B, "OR", size);
78                  when "0100" =>  -- XOR
79                      Result_aux <= vec_op(A, B, "XOR", size);
80                  when "0101" =>  -- SLL
81                      Result_aux <= vec_op(A, B, "SLL", size);
82                  when "0110" =>  -- SRL
83                      Result_aux <= vec_op(A, B, "SRL", size);
84                  when others =>
85                      Result_aux <= (others => '0');
86              end case;
87          else  -- Operação escalar
88              case Control is
89                  when "0000" =>  -- ADD
90                      Result_aux <= std_logic_vector(unsigned(A) +
                        ↪  unsigned(B));
91                  when "0001" =>  -- SUB
92                      Result_aux <= std_logic_vector(unsigned(A) -
                        ↪  unsigned(B));
93                  when "0010" =>  -- AND
94                      Result_aux <= A and B;
95                  when "0011" =>  -- OR
96                      Result_aux <= A or B;
97                  when "0100" =>  -- XOR
98                      Result_aux <= A xor B;
99                  when "0105" =>  -- SLL
100                     Result_aux <= std_logic_vector(shift_left(unsigned(A),
                        ↪  to_integer(unsigned(B(4 downto 0)))));
```

5.2.2 Unidade de controle

```
12          ALUOp     : out std_logic_vector(1 downto 0);  -- EX
13          ALUSrc    : out std_logic;
14          Branch    : out std_logic;                     -- MEM
15          MemRead   : out std_logic;                     -- MEM
16          MemWrite  : out std_logic;                     -- MEM
```

```
17          RegWrite   : out std_logic;                    -- WB
18          MemtoReg   : out std_logic                     -- WB
19      );
20  end control_unit;
21
22  -- Definição da arquitetura da unidade de controle
23  architecture Behavioral of control_unit is
24
25      -- Sinais auxiliares de controle
26      signal opcode       : std_logic_vector(6 downto 0);
27      signal ALUOp_aux    : std_logic_vector(1 downto 0); -- EX
28      signal ALUSrc_aux   : std_logic; -- EX
29      signal Branch_aux   : std_logic; -- MEM
30      signal MemRead_aux  : std_logic; -- MEM
31      signal MemWrite_aux : std_logic; -- MEM
32      signal RegWrite_aux : std_logic; -- WB
33      signal MemtoReg_aux : std_logic; -- WB
34      signal mode_aux     : std_logic; -- Novo sinal de controle para
        ↪  operação vetorial
35      signal vecSize_aux  : std_logic_vector(1 downto 0); -- Novo sinal de
        ↪  controle para tamanho da seção interna
36
37  begin
38      opcode <= Instr(6 downto 0);  -- Extrai o opcode da instrução
39
40      process(opcode)
41      begin
42          -- Valores padrão para sinais de controle
43          ALUOp_aux    <= "00";
44          ALUSrc_aux   <= '0';
45          Branch_aux   <= '0';
46          MemRead_aux  <= '0';
47          MemWrite_aux <= '0';
48          RegWrite_aux <= '0';
49          MemtoReg_aux <= '0';
50          mode_aux     <= '0';
51          vecSize_aux  <= "00";
52
53          case opcode is
```

```
54              when "0110011"  => -- R-Type
55                  ALUOp_aux    <= "10";
56                  ALUSrc_aux   <= '0';
57                  RegWrite_aux <= '1';
58                  mode_aux     <= '1';  -- Exemplo de operação vetorial
59                  vecSize_aux  <= "00"; -- Exemplo de tamanho de 4 bits
60
61              when "0010011" => -- I-Type Arith
62                  ALUOp_aux    <= "11";
63                  ALUSrc_aux   <= '1';
64                  RegWrite_aux <= '1';
65                  mode_aux     <= '1';  -- Exemplo de operação vetorial
66                  vecSize_aux  <= "01"; -- Exemplo de tamanho de 8 bits
67
68              when "0000011" => -- I-Type (LW)
69                  ALUOp_aux    <= "00";
70                  ALUSrc_aux   <= '1';
71                  MemRead_aux  <= '1';
72                  MemtoReg_aux <= '1';
73                  RegWrite_aux <= '1';
74                  mode_aux     <= '1';  -- Exemplo de operação vetorial
75                  vecSize_aux  <= "10"; -- Exemplo de tamanho de 16 bits
76
77              when "0100011" => -- S-Type (SW)
78                  ALUOp_aux    <= "00";
79                  ALUSrc_aux   <= '1';
80                  MemWrite_aux <= '1';
81                  mode_aux     <= '0';  -- Exemplo de operação escalar
82
83              when "1100011" => -- SB-Type (BEQ/BNE)
84                  ALUOp_aux    <= "01";
85                  ALUSrc_aux   <= '0';
86                  Branch_aux   <= '1';
87                  mode_aux     <= '0';  -- Exemplo de operação escalar
88
89              when "1101111" => -- JAL
90                  ALUOp_aux    <= "00";  -- Não importa para JAL
91                  ALUSrc_aux   <= '1';   -- A origem do dado para a ALU é
                    ↪ um imediato
```

```
92                  RegWrite_aux <= '1';    -- Escreve no registrador
93                  mode_aux     <= '0';    -- Exemplo de operação escalar
94
95              when "1100111" => -- JALR
96                  ALUOp_aux    <= "00";   -- Não importa para JALR
97                  ALUSrc_aux   <= '1';    -- A origem do dado para a ALU é
                    ↪  um imediato
98                  RegWrite_aux <= '1';    -- Escreve no registrador
99                  mode_aux     <= '0';    -- Exemplo de operação escalar
100
```

5.2.3 Extensão de Sinal

```
12          ImmOut    : out std_logic_vector(31 downto 0) -- Saída: imediato
            ↪  de 32 bits
13      );
14  end ImmGen;
15
16  -- Definição da arquitetura da unidade de geração de imediato
17  architecture Behavioral of ImmGen is
18      signal imm_4bit   : std_logic_vector(3 downto 0);
19      signal imm_8bit   : std_logic_vector(7 downto 0);
20      signal imm_16bit  : std_logic_vector(15 downto 0);
21      signal imm_32bit  : std_logic_vector(31 downto 0);
22  begin
23      process(Instr, mode, vecSize)
24      begin
25          if mode = '1' then
26              -- Operação vetorial
27              case vecSize is
28                  when "00" =>
29                      imm_4bit <= Instr(3 downto 0);
30                      ImmOut <= (others => '0');
31                      ImmOut(3 downto 0) <= imm_4bit;
32                  when "01" =>
33                      imm_8bit <= Instr(7 downto 0);
34                      ImmOut <= (others => '0');
35                      ImmOut(7 downto 0) <= imm_8bit;
36                  when "10" =>
37                      imm_16bit <= Instr(15 downto 0);
```

```
38                      ImmOut <= (others => '0');
39                      ImmOut(15 downto 0) <= imm_16bit;
40                  when others =>
41                      ImmOut <= (others => '0');
42              end case;
43          else
44              -- Operação escalar (32 bits)
45              imm_32bit <= Instr(31 downto 0);
46              ImmOut <= imm_32bit;
47          end if;
48      end process;
49  end Behavioral;
```

6 CONCLUSÃO

Após os testes realizados, na CPU RISC-V (RV32I) é possível concluir que o trabalho foi finalizado com êxito e cumpriu os objetivos almejados pelos estudantes. Todas as funcionalidades desejadas foram implementadas, proporcionando uma boa utilização do sistema.

Todos os códigos e o projeto pode ser encontrado clicando aqui.

BIBLIOGRAFIA

[1] Patterson, D. A., Hennessy, J. L. (2021). *Computer Organization and Design RISC-V Edition: The Hardware Software Interface, 2nd*. Morgan Kaufman. ISBN: 978-0-12-820331-6.

[2] Waterman, A., Asanović, K. (2017). *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2*. SiFive Inc. and EECS Department, University of California, Berkeley.

[3] RISC-V Simulador Online. *RISC-V Web Simulator*. Disponível em: `https://riscv.vercel.app/`.