Atividade Prática - CPU RISC-V

Luís Rafael Sena, Pedro Videira Rubistein, Eduardo de Mendonça Freire,
João Pedro Peçanha Cotrim, Carlos Augusto Bertão Rodrigues

July 22, 2025

Contents

1 Introdução

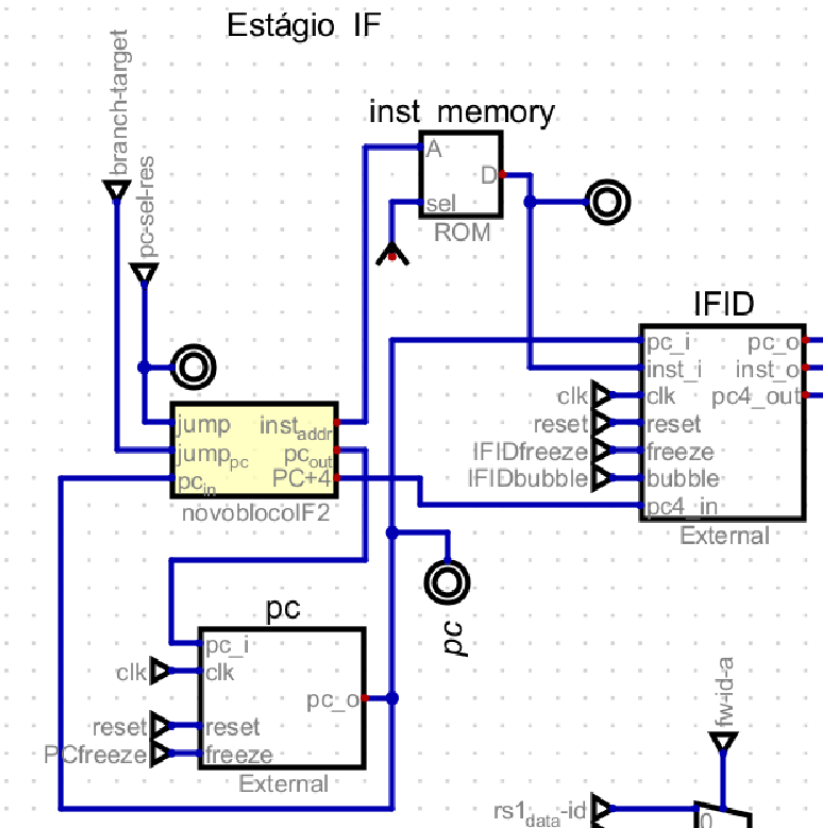
Este relatório apresenta a síntese do trabalho desenvolvido ao longo da realização de três projetos baseados na arquitetura RISC-V: o somador vetorial, a CPU RISC-V e a implementação de instruções vetoriais. Todos os projetos foram desenvolvidos em VHDL, utilizando o simulador Digital e integrados, ao final do processo, em um único arquivo .dig.

O texto está organizado em seções dedicadas a cada um dos projetos, com subseções que detalham o desenvolvimento individual de cada parte. Em especial, destaca-se a seção referente à adição de instruções SIMD, que aborda a integração dos módulos anteriores em um sistema unificado.

2 CPU

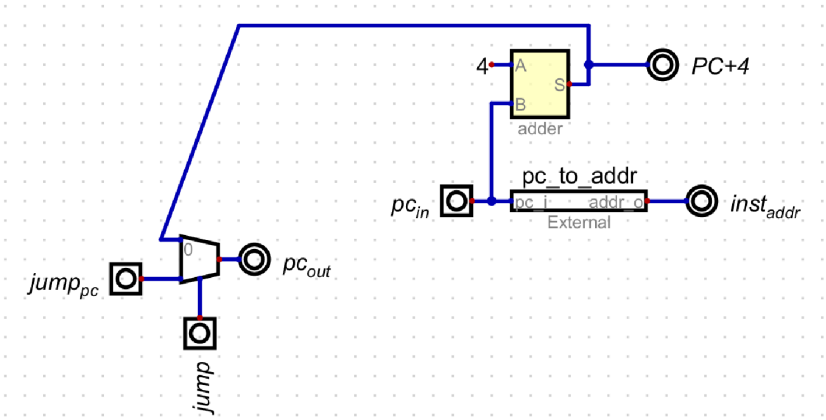
A CPU desenvolvida é uma arquitetura RISC-V com pipeline de cinco estágios: Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM) e Write-Back (WB). Além dos estágios principais, a CPU conta com uma unidade de detecção de dependências (Hazard Detection Unit), uma unidade de encaminhamento (Forwarding Unit) e realiza o cálculo de saltos (jumps e branches) diretamente no estágio de decodificação (ID), com o objetivo de reduzir o impacto de branches no pipeline.

2.1 IF



No estágio IF, ocorre o acesso ao Program Counter (PC) e à memória de instrução. Para isso, há um bloco combinacional, representado em amarelo na imagem.

Esse bloco recebe um booleano indicando a ocorrência de jump neste ciclo e o endereço do jump em questão, ambos calculados no estágio ID. Além disso, o bloco recebe o valor atual de PC. Com esses parâmetros, ele gera o novo valor de PC, além do endereço a ser lido na memória de instrução e do valor de PC+4 (para instruções como jal). A seguinte imagem mostra o interior do bloco:



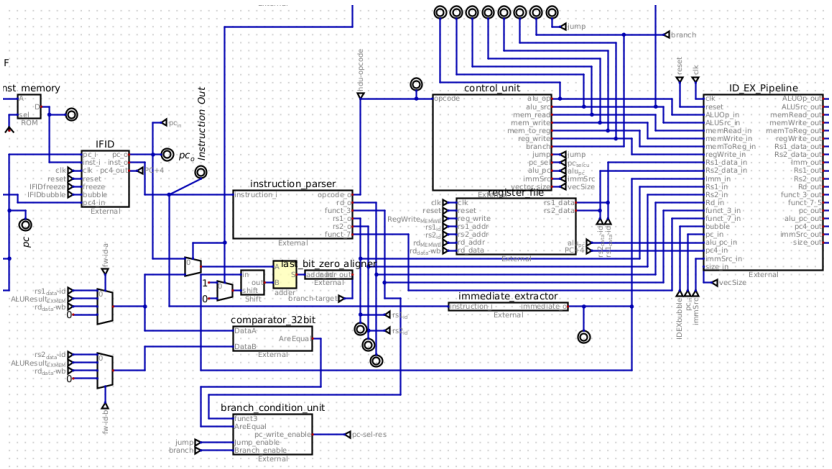
Um multiplexador seleciona o novo valor de PC entre PC+4 e o endereço de jump/branch. Além disso, um módulo combinacional em VHDL determina o endereço a ser lido com base no valor de PC.

A necessidade desse módulo vem de uma restrição de escopo associada à memória de instrução. A memória ROM do Digital só aceita endereços de até 24 bits, menores que o valor de PC, o qual tem 32 bits. Assim, optou-se por utilizar a memória ROM com 2^{24} palavras de 32 bits cada. Com isso, o endereço addr_o a ser lido é calculado pelo módulo VHDL como pc(25 downto 2). Isso preserva a propriedade de que PC+4 refere-se à próxima instrução. A desvantagem dessa abordagem é que a quantidade total de instruções que cabe na memória é de apenas 2^{24}, valor que, embora seja menor do que a quantidade que PC poderia teoricamente endereçar, é mais que suficiente para programas inteiros.

Ao final do estágio IF, os valores de PC, da instrução lida e de PC+4 são enviados ao registrador IF/ID.

2.2 ID

O estágio ID é responsável por decodificar a instrução a ser executada, gerar os sinais de controle, calcular os branches/jumps, além de realizar o acesso aos dados contidos no register file e também o cálculo dos imediatos. Cada uma dessas funções é implementada por um ou mais módulos em VHDL.



O Instruction Parser é responsável por subdividir a instrução em diferentes faixas de bits, que são encaminhadas para os demais módulos encarregados das etapas seguintes da decodificação.

A Control Unit utiliza o opcode para gerar os sinais de controle que serão propagados ao longo da pipeline, coordenando operações como acesso à memória, operação da ALU, entre outras.

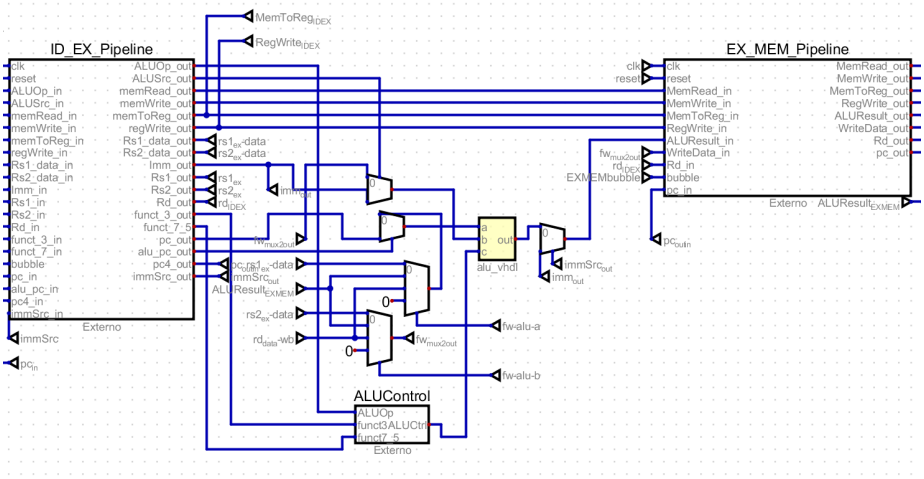
Neste estágio também está presente o Register File, que conta com lógica interna de forwarding para permitir escrita na primeira metade do ciclo de clock e leitura na segunda.

O Immediate Extractor é responsável por gerar, a partir da instrução, o valor imediato a ser utilizado nas operações subsequentes.

Por fim, um comparador em conjunto com a Branch Condition Unit aciona o multiplexador responsável por selecionar entre o endereço de salto (jump/branch) ou o próximo endereço sequencial de instrução. O endereço de desvio é calculado por um circuito de adder e align.

2.3 EX

O estágio EX é responsável por realizar as operações lógicas e aritméticas, cálculo de endereço de memória (lw e sw), e operações imediatas como a lui (load Upper Immediate).



Todas as operações aritméticas e lógicas são realizadas pela ALU (Arithmetic Logic Unit), a qual obedece ao seguinte código VHDL:

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity alu_combinational is
    Port (
        a_in            : in  std_logic_vector(31 downto 0);
        b_in            : in  std_logic_vector(31 downto 0);
        c_in            : in  std_logic_vector(3 downto 0);
        result_out      : out std_logic_vector(31 downto 0)

    );
end alu_combinational;

architecture Behavioral of alu_combinational is

begin

    process(a_in, b_in, c_in)
        variable temp_result : std_logic_vector(31 downto 0);
    begin
        -- Initialize temp_result for safety (though not strictly necessary for full case co
        temp_result := (others => '0');

        case c_in is
            when "0000" =>
```

```
                temp_result := a_in and b_in;
            when "0001" =>
                temp_result := a_in or b_in;
            when "0011" =>
                temp_result := a_in xor b_in;
            when "0010" =>
                temp_result := std_logic_vector(unsigned(a_in) + unsigned(b_in));
            when "0110" =>
                -- For subtraction, ensure both operands are treated as unsigned for consis
                -- If signed subtraction is intended, use signed types for both a_in and b_
                temp_result := std_logic_vector(unsigned(a_in) - unsigned(b_in));
            when "0100" =>
                -- For shift operations, the shift amount should be an integer,
                -- and using 'unsigned' directly on the slice ensures proper conversion.
                temp_result := std_logic_vector(shift_left(unsigned(a_in), to_integer(unsign
            when "0101" =>
                temp_result := std_logic_vector(shift_right(unsigned(a_in), to_integer(unsig
            when others =>
                temp_result := (others => '0'); -- Default or undefined operation
        end case;

        -- Assign the calculated temporary result to the output signal
        result_out <= temp_result;

    end process;

end Behavioral;
```

A operação específica realizada pela ALU é determinada pelo sinal de controle ALUCtrl, que, por sua vez, é gerado pela unidade de controle ALUControl a partir de três sinais de entrada: o ALUOp (uma classificação ampla do tipo de operação, gerado no estado ID), o vetor funct3 e o bit funct7_5 (5^{o} bit de funct7). Os dois últimos são extraídos da instrução no estágio ID e usados para melhor precisar a operação desejada. O código em VHDL da ALUControl é o seguinte:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ALUControl is
    port (
        ALUOp  : in  std_logic_vector(1 downto 0);
```

```
        funct3 : in  std_logic_vector(2 downto 0);
        funct7_5: in std_logic;  -- bit 30 da instrução (funct7[5])
        ALUCtrl : out std_logic_vector(3 downto 0)
    );
end entity;

architecture Behavioral of ALUControl is
begin
    process(ALUOp, funct3, funct7_5)
    begin
        case ALUOp is
            when "00" =>  -- Loads, Stores, AUIPC (ADD)
                ALUCtrl <= "0010";  -- ADD

            when "01" =>  -- Branches (SUB)
                ALUCtrl <= "0110";  -- SUB

            when "10" =>  -- R-Type instructions
                case funct3 is
                    when "000" =>  -- ADD or SUB
                        if funct7_5 = '0' then
                            ALUCtrl <= "0010"; -- ADD
                        else
                            ALUCtrl <= "0110"; -- SUB
                        end if;
                    when "111" => ALUCtrl <= "0000"; -- AND
                    when "110" => ALUCtrl <= "0001"; -- OR
                    when "100" => ALUCtrl <= "0011"; -- XOR
                    when "001" => ALUCtrl <= "0100"; -- SLL
                    when "101" => ALUCtrl <= "0101"; -- SRL
                    when others =>
                        ALUCtrl <= "1111"; -- Default / não definido
                end case;

            when "11" =>  -- I-Type aritméticas e shifts imediatas
                case funct3 is
                    when "000" => ALUCtrl <= "0010"; -- ADDI
                    when "111" => ALUCtrl <= "0000"; -- ANDI
                    when "110" => ALUCtrl <= "0001"; -- ORI
                    when "100" => ALUCtrl <= "0011"; -- XORI
                    when "001" => ALUCtrl <= "0100"; -- SLLI
                    when "101" => ALUCtrl <= "0101"; -- SRLI
                    when others =>
                        ALUCtrl <= "1111"; -- Default / não definido
                end case;
```

```
            when others =>
                ALUCtrl <= "1111"; -- Default / não definido
        end case;
    end process;
end architecture;
```

Além disso, há outros sinais de controle importantes:

O ALUSrc decide se a entrada B da ALU irá receber o valor de um registrador ou de um imediato — decisão necessária para diferenciar operações envolvendo valores imediatos.

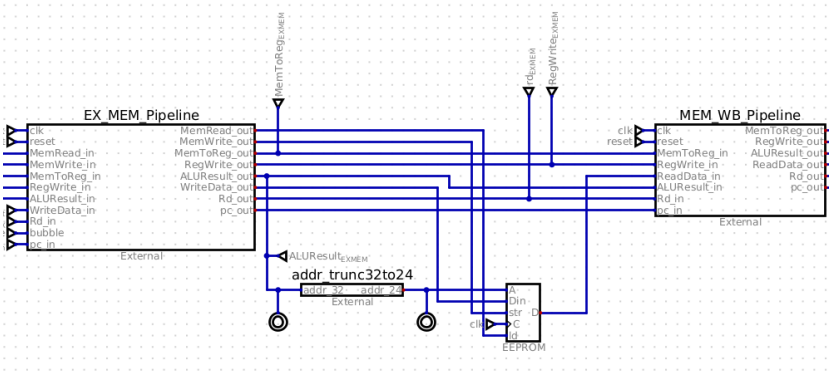
O ALU_{pc} decide quando que o valor de PC será utilizado para a operação auipc.

O ImmSrc decide se será utilizado o valor calculado pela ALU ou o próprio imediato para o estágio MEM, tendo como papel permitir a operação LUI (load Upper Immediate).

Por fim, para que sejam utilizados nos estágios seguintes, os sinais de controle MemtoReg, MemRead, MemWrite são propagados para o registrador EXMEM. O valor de PC+4, para as operações de Jal e Jalr, também é propagado.

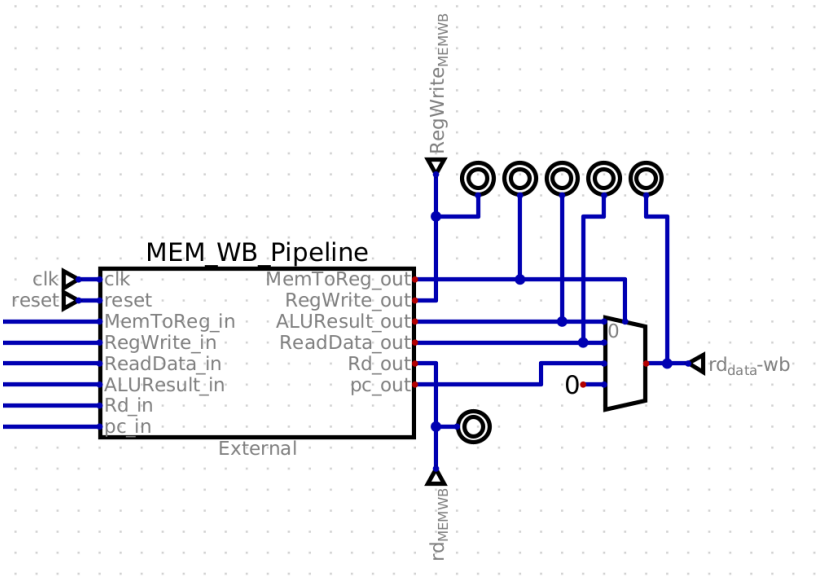
2.4 MEM

O estágio MEM é responsável por realizar o acesso à memória de dados. Optamos por utilizar a EEPROM do Digital como elemento de memória, dada a possibilidade de carregamento de dados assíncrono. Por limitação do Digital, não podemos fazer o endereçamento desse módulo com 32 bits, dada sua limitação de máximo de 24 bits. Por isso, optamos por truncar o endereço fornecido pelas instruções, utilizando somente os 24 bits mais baixos.



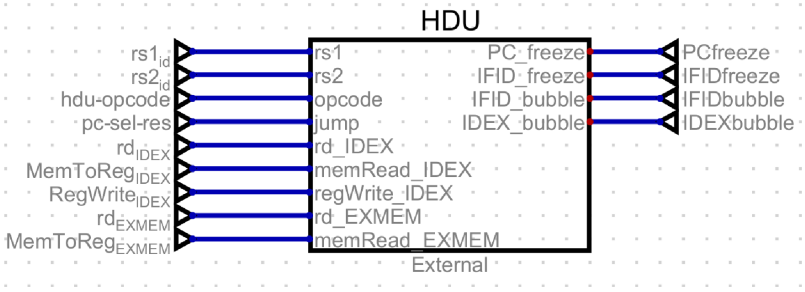
2.5 WB

O Writeback consiste no estágio mais simples. Basicamente, é composto pelo registrador MEM/WB e um MUX, que determina a entrada a ser escrita no registrador Rd a partir do sinal MemToReg.



2.6 HDU e Forwarding

2.6.1 HDU



Para garantir a corretude da execução de qualquer programa, o processador conta com Hazard Detection Unit (HDU). Com base na instrução que está no estágio ID, ela insere bolhas na pipeline caso essa instrução vá depender de

dados que ainda não foram computados. Além disso, caso a instrução em ID vá causar jump, a HDU garante que a instrução no endereço seguinte não seja executada. Para isso, a HDU tem os seguintes pinos:

- Entradas rs1, rs2 e opcode provenientes da instrução do estágio ID permitem que se identifique o tipo da instrução, e de quais registradores ela depende.
- Entrada jump indica se está ocorrendo salto neste ciclo.
- Entradas rd, MemToReg e RegWrite provenientes da instrução em EX e entradas rd e MemToReg da instrução em MEM permitem a identificação de hazards de dados.
- Saídas PCfreeze e IFIDfreeze congelam os registradores em questão. Quando são '1', o registrador mantém seu valor atual no próximo ciclo.
- Saídas IFIDbubble e IDEXbubble inserem bolhas nos registradores em questão. Quando são '1', o registrador recebe um NOP no próximo ciclo (nesse caso, seu valor atual pode continuar se propagando ao longo da pipeline, pois o NOP é inserido de forma síncrona).

A lógica da HDU consiste em identificar três tipos de data hazard e um tipo de control hazard:

- Data hazard de load-use — uma instrução em EX irá ler um dado na memória e escrever em um registrador do qual precisamos no estágio de execução. Isso demanda um stall. Note que, caso a instrução em EX fosse escrever no registrador sem ler na memória, a unidade de forwarding resolveria a dependência, sem a necessidade de stall.
- Data hazard de load-branch — uma instrução em EX ou MEM irá ler um dado na memória e escrever em um registrador do qual precisamos em ID para o cálculo de um salto. Isso demanda um ou dois stalls.
- Data hazard de write-branch — uma instrução em EX irá calcular (mesmo sem acesso à memória) um dado e escrever em um registrador do qual precisamos em ID para o cálculo de um salto. Como o dado é necessário de forma antecipada, essa situação não é resolvida puramente pela unidade de forwarding, demandando um stall.

  Os três tipos de data hazard são resolvidos pelas mesmas saídas. PCfreeze = IFIDfreeze = '1' congelam a instrução com dependência de dados. IFIDbubble recebe '0', pois o registrador IF/ID é congelado, e não recebe NOP. Por fim, IDEXbubble = '1' garante que a instrução com o valor errado não se propaga pela pipeline.

- Control hazard — caso não haja data hazard, ainda é checada mais uma condição. Se vamos executar um salto este ciclo, é necessário impedir que a instrução apontada por PC+4 seja executada.

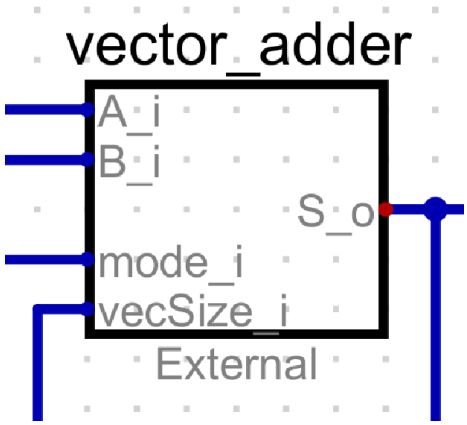
  Para a resolução de control hazard, as saídas são as seguintes: PCfreeze = '0', para permitir o salto. IFIDbubble = '1', para que a instrução apontada por PC+4 não avance pela pipeline, e IFIDfreeze = '0', pois IFID já está recebendo NOP. Por fim, IDEXbubble = '0', para que a instrução de salto continue se propagando normalmente.

2.6.2 Forwarding

O mecanismo de forwarding tem como objetivo principal reduzir a necessidade de stalls na pipeline, resolvendo a maior parte dos data hazards do tipo Read After Write (RAW) por meio do encaminhamento adiantado de valores recém-computados para os estágios que deveriam esperar.

Ela é responsável por controlar os sinais dos MUXs que selecionam as entradas dos diferentes sinais intermediários, tanto no estágio ID (Jumps/branches), quanto para a entrada da ALU. Dessa forma, o forwarding cobre tanto as dependências aritméticas quanto as dependências envolvidas em controle de fluxo, minimizando a inserção de bolhas na pipeline e aumentando o desempenho do processador.

3 Somador Vetorial



Para o somador vetorial, a abordagem utilizada foi a seguinte: definiu-se uma entity VHDL adder4, a qual implementa um somador de 4 bits com entradas

A, B, Cin e saídas S, Cout. adder4 tem carry lookahead; por isso, no somador vetorial, o carry se propaga de 4 em 4 bits, em vez de bit a bit, visando a baixa latência no cálculo das somas. A lógica de carry lookahead é conhecida e está no seguinte trecho de código VHDL:

```
signals_generate: for i in 0 to 3 generate
    prop(i) <= A(i) xor B(i);
    gen(i) <= A(i) and B(i);
end generate;

-- Carry lookahead
carry(0) <= Cin;
carry(1) <= gen(0) or (prop(0) and Cin);

carry(2) <= gen(1) or (prop(1) and gen(0)) or (prop(1) and prop(0) and Cin);

carry(3) <= gen(2) or (prop(2) and gen(1)) or (prop(2) and prop(1) and gen(0))
or (prop(2) and prop(1) and prop(0) and Cin);

Cout <= gen(3) or (prop(3) and gen(2)) or (prop(3) and prop(2) and gen(1))
or (prop(3) and prop(2) and prop(1) and gen(0)) or (prop(3) and prop(2) and
prop(1) and prop(0) and Cin);

sum_generate: for i in 0 to 3 generate
    S(i) <= prop(i) xor carry(i);
end generate;
```

A entity VHDL vector_adder agrega 8 componentes adder4, de índices 0 a 7. Assim, o somador 0 recebe os bits 3 a 0 de A, e assim por diante, até o somador 7, que recebe os bits 31 a 28 de A. A única circunstância que se altera entre os diferentes valores de vecSize_i e mode_i é a entrada Cin de cada adder4. Isso permite que os mesmos somadores sejam utilizados para os diferentes tamanhos de vetor, o que leva a uma economia de circuito e espaço. Para determinar-se as entradas de cada adder4:

Primeiro, determina-se a quantidade n de somadores que estarão juntos em cada bloco. Por exemplo, para vecSize_i = "11", temos n = 4 (cada soma de 16 bits é feita por um grupo de 4 somadores). Assim, para todos os somadores a partir do segundo de cada bloco (ou seja, para os somadores tais que i mod n é diferente de 0), Cin vem diretamente do Cout do somador anterior. Porém, para o primeiro somador de cada bloco (tal que i mod n = 0), Cin é igual a mode_i, que é '0' para soma. Ou seja, no início de cada bloco, o carry é forçadamente '0', pois não deve ser propagado do bloco anterior.

No caso de subtração, aplica-se bloco a bloco a lógica de inverter todos os

bits de B e estabelecer Cin = '1' (complemento de 2). As entradas B de todos os somadores passam a vir de not(B). A propagação de carry a partir do segundo somador de cada bloco é idêntica. Porém, a entrada Cin do primeiro somador de cada bloco é igual a mode_i, que agora é '1'.

Essa lógica é implementada pelo seguinte trecho de VHDL:

```
-- Instanciar os 8 somadores
adder_generate: for i in 0 to 7 generate

    signal prop: std_logic;
    signal actual_carry: std_logic;
    signal B: std_logic_vector(3 downto 0);

begin
    -- Propagar carry apenas no meio dos blocos
    prop <=
        '1' when
            ((vecSize_i = "01") and (i mod 2 /= 0)) or -- Blocos de 8, os adders se juntam
            -- de 2 em 2
            ((vecSize_i = "10") and (i mod 4 /= 0)) or -- Blocos de 16, os adders se juntam
            -- de 4 em 4
            ((vecSize_i = "11") and (i /= 0)) -- Bloco de 32, todos os 8 adders se juntam

            -- com vecSize_i = "00", não há propagação
        else '0';

    -- No meio dos blocos, propagamos carry. No início dos blocos, colocamos carry '0'
    -- para soma e carry '1' para subtração (complemento de 2)
    actual_carry <=
        carry(i) when prop = '1'
        else mode_i;

    B <=
        B_i(4*i+3 downto 4*i) when mode_i = '0'
        else not_B(4*i+3 downto 4*i);

    adder: adder4
    port map(
        A => A_i(4*i+3 downto 4*i),

        B => B,

        Cin => actual_carry,
```

```
        S => S_o(4*i+3 downto 4*i),

        Cout => carry(i+1)
    );
end generate;
```

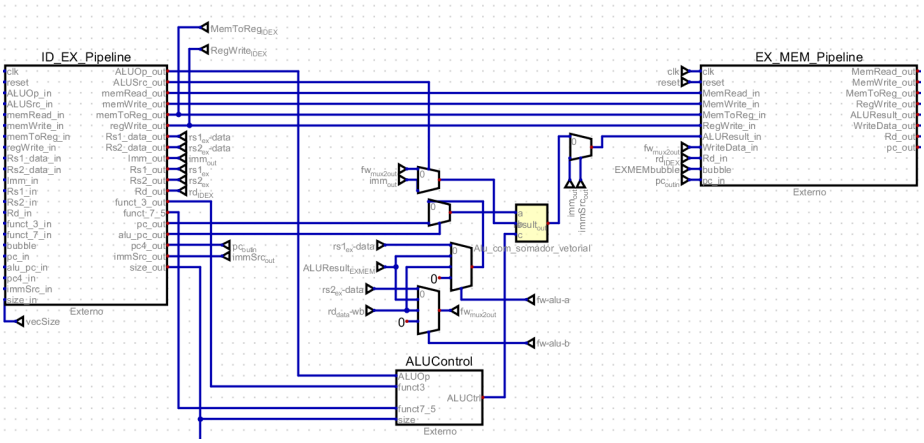
4 Adição de SIMD

Para viabilizar as instruções vetoriais, foi necessário adicionar novos opcodes que identificam os diferentes tipos de instruções e tamanhos de vetores, mantendo a utilização dos registradores já existentes no register-file. Para isso, foram definidos os opcodes listados abaixo e a unidade de controle foi modificada para incluir uma nova saída: o tamanho do vetor a ser utilizado.

```
    constant VECTOR_R_OP4B   : std_logic_vector(6 downto 0) := "1010000";
    constant VECTOR_R_OP8B   : std_logic_vector(6 downto 0) := "1010001";
    constant VECTOR_R_OP16B  : std_logic_vector(6 downto 0) := "1010010";
    constant VECTOR_I_OP4B   : std_logic_vector(6 downto 0) := "1010100";
    constant VECTOR_I_OP8B   : std_logic_vector(6 downto 0) := "1010101";
    constant VECTOR_I_OP16B  : std_logic_vector(6 downto 0) := "1010110";
    constant VECTOR_U_OP4B  : std_logic_vector(6 downto 0) := "1011000";
    constant VECTOR_U_OP8B  : std_logic_vector(6 downto 0) := "1011001";
    constant VECTOR_U_OP16B : std_logic_vector(6 downto 0) := "1011010";
```

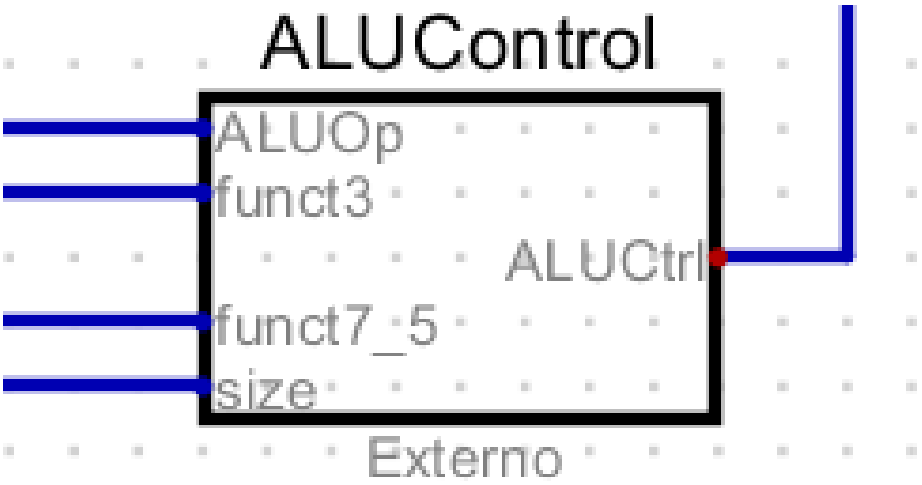
Desses novos opcodes é gerado um novo sinal de controle, size, que representa o tamanho do bloco de cada vetor.

4.1 Novo estágio Ex



Para suportar as operações vetoriais, o estágio Ex sofreu mudanças cruciais, entre elas no ALUControl devido ao aumento do número de instruções realizadas pela CPU. Além disso, na própria ALU houveram mudanças para realizar esses novos cálculos.

4.2 ALUControl com as operações vetoriais



No sistema de controle da ALU teve que ser adicionado uma nova entrada, size, que para as operações vetoriais representaria o tamanho do bloco a ser trabalhado, além disso o valor da saída sofreu mudanças, tendo um aumento na quantidade de bits, devido ao aumento do número de instruções.

Assim, o novo ALUControl é gerado por esse novo código vhdl:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ALUControl is
    port (
        ALUOp  : in  std_logic_vector(1 downto 0);
        funct3 : in  std_logic_vector(2 downto 0);
        funct7_5: in std_logic;  -- bit 30 da instrução (funct7[5])
        size : in std_logic_vector(1 downto 0); -- controle externo para operações vetoriais
        ALUCtrl : out std_logic_vector(4 downto 0)
    );
end entity;

architecture Behavioral of ALUControl is
begin
    process(ALUOp, funct3, funct7_5, size)
    begin
        case ALUOp is
            when "00" =>  -- Loads, Stores, AUIPC (ADD)
                case size is
                        when "00" =>
                                ALUCtrl <= "01000";  -- 04
                        when "01" =>
                                ALUCtrl <= "01001";  -- 08
                        when "10" =>
                                ALUCtrl <= "01010";  -- 16
                        when "11" =>
                                ALUCtrl <= "00010";  -- 32
                        when others =>
                                        ALUCtrl <= "11111";
                end case;

            when "01" =>  -- Branches (SUB)
                case size is
                        when "00" =>
                                ALUCtrl <= "01100";  -- 04
                        when "01" =>
                                ALUCtrl <= "01101";  -- 08
                        when "10" =>
                                ALUCtrl <= "01110";  -- 16
                        when "11" =>
```

```
                    ALUCtrl <= "00110";  -- 32
                when others =>
                                ALUCtrl <= "11111";
        end case;

    when "10" =>  -- R-Type instructions
        case funct3 is
            when "000" =>  -- ADD or SUB
                if funct7_5 = '0' then
                        case size is
                                when "00" =>
                                        ALUCtrl <= "01000";  -- 04
                                when "01" =>
                                        ALUCtrl <= "01001";  -- 08
                                when "10" =>
                                        ALUCtrl <= "01010";  -- 16
                                when "11" =>
                                        ALUCtrl <= "00010";  -- 32
                                when others =>
                                                ALUCtrl <= "11111";
                        end case;

                else
                    case size is
                        when "00" =>
                                ALUCtrl <= "01100";  -- 04
                        when "01" =>
                                ALUCtrl <= "01101";  -- 08
                        when "10" =>
                                ALUCtrl <= "01110";  -- 16
                        when "11" =>
                                ALUCtrl <= "00110";  -- 32
                        when others =>
                                        ALUCtrl <= "11111";
                end case;
                end if;
            when "111" => ALUCtrl <= "00000"; -- AND
            when "110" => ALUCtrl <= "00001"; -- OR
            when "100" => ALUCtrl <= "00011"; -- XOR
            when "001" => -- SLL
                case size is
                        when "00" =>
                                ALUCtrl <= "10000";  -- 04
                        when "01" =>
                                ALUCtrl <= "10001";  -- 08
                        when "10" =>
```

```
                        ALUCtrl <= "10010";  -- 16
                    when "11" =>
                        ALUCtrl <= "00100";  -- 32
                    when others =>
                                ALUCtrl <= "11111";
            end case;

        when "101" => -- SRL
            case size is
                    when "00" =>
                        ALUCtrl <= "11000";  -- 04
                    when "01" =>
                        ALUCtrl <= "11001";  -- 08
                    when "10" =>
                        ALUCtrl <= "11010";  -- 16
                    when "11" =>
                        ALUCtrl <= "00101";  -- 32
                    when others =>
                                ALUCtrl <= "11111";
            end case;

        when others =>
            ALUCtrl <= "11111"; -- Default / não definido
    end case;

when "11" =>  -- I-Type aritméticas e shifts imediatas
    case funct3 is
        when "000" =>
            case size is
                    when "00" =>
                        ALUCtrl <= "01000";  -- 04
                    when "01" =>
                        ALUCtrl <= "01001";  -- 08
                    when "10" =>
                        ALUCtrl <= "01010";  -- 16
                    when "11" =>
                        ALUCtrl <= "00010";  -- 32
                    when others =>
                                ALUCtrl <= "11111";
            end case;

        when "111" => ALUCtrl <= "00000"; -- ANDI
        when "110" => ALUCtrl <= "00001"; -- ORI
        when "100" => ALUCtrl <= "00011"; -- XORI
        when "001" => -- SLLI
            case size is
```

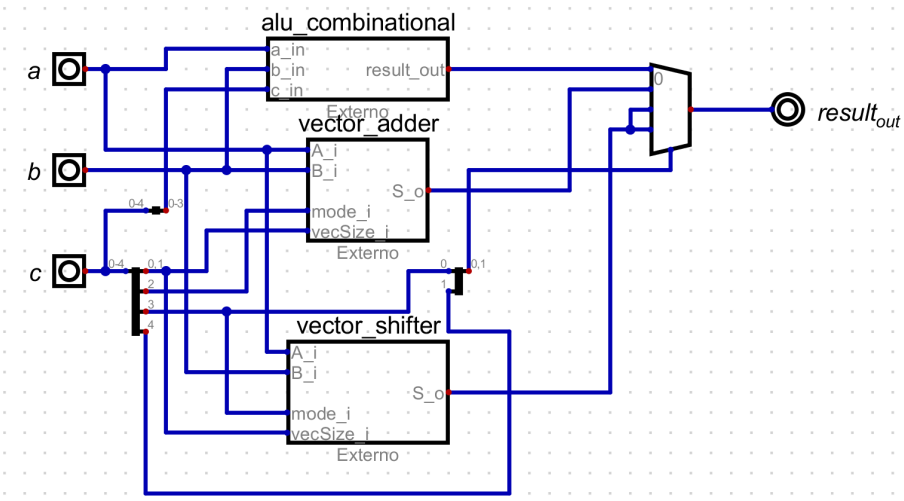
```
                                when "00" =>
                                        ALUCtrl <= "10000";  -- 04
                                when "01" =>
                                        ALUCtrl <= "10001";  -- 08
                                when "10" =>
                                        ALUCtrl <= "10010";  -- 16
                                when "11" =>
                                        ALUCtrl <= "00100";  -- 32
                                when others =>
                                                ALUCtrl <= "11111";
                            end case;

                        when "101" => -- SRLI
                            case size is
                                when "00" =>
                                        ALUCtrl <= "11000";  -- 04
                                when "01" =>
                                        ALUCtrl <= "11001";  -- 08
                                when "10" =>
                                        ALUCtrl <= "11010";  -- 16
                                when "11" =>
                                        ALUCtrl <= "00101";  -- 32
                                when others =>
                                                ALUCtrl <= "11111";
                            end case;

                        when others =>
                            ALUCtrl <= "11111"; -- Default / não definido
                    end case;

                when others =>
                    ALUCtrl <= "11111"; -- Default / não definido
            end case;
        end process;
    end architecture;
```

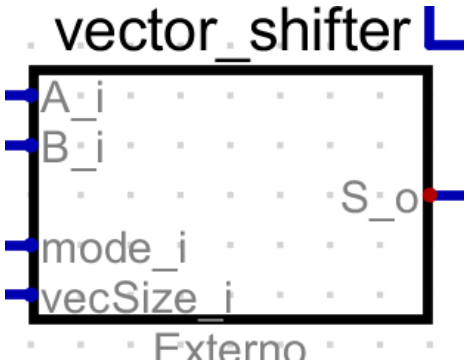
4.3 ALU com somador vetorial



Para que a CPU possa suportar as seguintes operações vetoriais: add, addi, sub, sll, srl, slli e srli, precisamos adicionar o somador vetorial e um shifter vetorial, além disso seu sinal de controle precisou aumentar de 4 para 5 bits devido ao aumento do número de operações que seria realizado.

A saída selecionada é controlada pelos dois bits mais significativos do Alu-Control.

4.4 Shifter Vetorial



Além das operações de soma e subtração vetorial, precisou-se implementar as operações de deslocamento de bit vetorial, para que a arquitetura pudesse suportar as seguintes operações vetoriais: sll, slli, srl e srli.

O shifter vetorial trabalha com 32 bits, no entanto, por causa da abordagem em vetor, ele pode dividir as entradas em grupos de 4,8 e 16 bits ou tratar com os números inteiros.

Essa lógica é implementada pelo seguinte código em vhdl:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity vector_shifter is
    Port(
        A_i       : in  std_logic_vector(31 downto 0);
        B_i       : in  std_logic_vector(31 downto 0);  -- rs2 ou imediato
        mode_i    : in  std_logic;                      -- '0' = shift left, '1' = shift ri
        vecSize_i : in  std_logic_vector(1 downto 0);   -- 00:4b, 01:8b, 10:16b, 11:32b
        S_o       : out std_logic_vector(31 downto 0)
    );
end vector_shifter;

architecture Behavioral of vector_shifter is

    signal result : std_logic_vector(31 downto 0);
    signal shamt  : std_logic_vector(4 downto 0);
    signal shamt_int : integer;

begin

    -- Pega os 5 LSBs do B_i como shift amount
    shamt <= B_i(4 downto 0);
    shamt_int <= to_integer(unsigned(shamt));

    -- Processo único com geradores internos para diferentes tamanhos de vetores
    process(A_i, shamt_int, mode_i, vecSize_i)
        variable temp_result : std_logic_vector(31 downto 0);
        variable segment4 : std_logic_vector(3 downto 0);
        variable segment8 : std_logic_vector(7 downto 0);
        variable segment16 : std_logic_vector(15 downto 0);
    begin
      temp_result := (others => '0');

      case vecSize_i is

      when "00" =>  -- 4-bit vetores (8 grupos)
        gen4: for i in 0 to 7 loop
```

```
  if shamt_int >= 4 then
                      segment4 := (others => '0');
  else
      if mode_i = '0' then
          segment4 := std_logic_vector(shift_left(unsigned(A_i(4*i+3 downto 4*i)), sha
      else
          segment4 := std_logic_vector(shift_right(unsigned(A_i(4*i+3 downto 4*i)), sh
      end if;
          temp_result(4*i+3 downto 4*i) := segment4;
  end if;
  end loop;

when "01" =>  -- 8-bit vetores (4 grupos)
  gen8: for i in 0 to 3 loop
  if shamt_int >= 8 then
      segment8 := (others => '0');
  else
      if mode_i = '0' then
          segment8 := std_logic_vector(shift_left(unsigned(A_i(8*i+7 downto 8*i)), sha
      else
          segment8 := std_logic_vector(shift_right(unsigned(A_i(8*i+7 downto 8*i)), sh
      end if;
      temp_result(8*i+7 downto 8*i) := segment8;
  end if;
  end loop;

when "10" =>  -- 16-bit vetores (2 grupos)
  gen16: for i in 0 to 1 loop
  if shamt_int >= 16 then
      segment16 := (others => '0');
  else
      if mode_i = '0' then
          segment16 := std_logic_vector(shift_left(unsigned(A_i(16*i+15 downto 16*i))
      else
          segment16 := std_logic_vector(shift_right(unsigned(A_i(16*i+15 downto 16*i)
      end if;
      temp_result(16*i+15 downto 16*i) := segment16;
  end if;
  end loop;

when others =>  -- 32-bit vetor (1 grupo)
  if mode_i = '0' then
      temp_result := std_logic_vector(shift_left(unsigned(A_i), shamt_int mod 32));
  else
      temp_result := std_logic_vector(shift_right(unsigned(A_i), shamt_int mod 32));
  end if;
```

```
      end case;

        S_o <= temp_result;

    end process;

end Behavioral;
```

O componente $vector_{shifter}$ aplica deslocamentos (shifts) lógicos à esquerda ou à direita em um vetor de 32 bits (A_i). A direção do shift é controlada por $mode_i$, e a quantidade de deslocamento é extraída dos 5 bits menos significativos de B_i.

A operação é feita de forma vetorial, ou seja, o vetor de 32 bits é dividido em blocos menores (de 4, 8, 16 ou 32 bits) conforme o sinal de controle $vecSize_i$. Cada bloco é deslocado separadamente pela mesma quantidade ($shamt_{int}$). Caso o deslocamento seja maior ou igual ao tamanho do bloco, o resultado daquele bloco é zerado.

Os deslocamentos em cada bloco são feitos por loops, na qual o tamanho do loop é decidido pela entrada $vecSize_i$, e caso o valor dentro do sinal $shamt_{int}$ for maior que a faixa de bits, o bloco será automaticamente zerado.

Os blocos processados são então reunidos para formar o vetor de saída S_o.

5 Conclusão

Para facilitar os testes da CPU, nos primeiros 24 endereços da memória de instrução está carregado um programa de ordenação. Para seu funcionamento, o primeiro endereço da memória de dados contém o tamanho n do vetor a ser ordenado, e os n endereços seguintes contém o vetor em questão. O programa segue uma lógica simples de selection sort, encontrando o mínimo e trocando com a primeira posição:

```
# main:
    # Carregar o tamanho n do vetor e calcular n+1, usado como limite de repetições
    lw x18, 0(x0)
    addi x18, x18, 1
    # Inicializar o loop "for"
    addi x19, x0, 1
        # Escrever o parâmetro e chamar min_idx
        addi x10, x19, 0
```

```
        jalr x1, 48(x0)
        # Fazer o swap
        lw x10, 0(x19)
        lw x11, 0(x5)
        sw x10, 0(x5)
        sw x11, 0(x19)
        # Checar se já terminamos de ordenar
        addi x19, x19, 1
        bne x19, x18, -14
    # Ao fim da execução, a CPU fica presa nesta instrução jal
    jal x0, 0
# min_idx:
    # Recebe em x10 o endereço a partir do qual percorrer
    # Retorna em x5 o índice do valor mínimo do vetor a partir desse endereço

    # x11 é o mínimo até agora
    # x5 é seu índice
    # x12 é o iterador do "for"
    lw x11, 0(x10)
    addi x5, x10, 0
    addi x12, x10, 0
        lw x13, 0(x12)
        # Checar se o valor em questão é menor que o mínimo até agora
        sub x14, x13, x11
        srli x14, x14, 31
        beq x14, x0, 6
            addi x11, x13, 0
            addi x5, x12, 0
        # Checar se já terminamos de percorrer o vetor
        addi x12, x12, 1
        bne x12, x18, -14
    # Retornar
    jalr x0, 0(x1)
```

Na memória de dados, está carregado um vetor de 3 elementos: 7, 5 e 6.

Esse exemplo ilustra como as instruções da ISA RISC-V, corretamente implementadas pelo projeto, podem se unir para resolver problemas maiores, assim levando a uma CPU completa.

6 Referências

Geeks for geeks. *Carry Look-Ahead Adder*. `https://www.geeksforgeeks.org/digital-logic/carry-look-ahead-adder/`. Acessada em 05 de julho de 2025. 2023.

hneeman. *Digital*. `https://github.com/hneemann/Digital`. Acessada em 05 de julho de 2025. 2025.

LupLab @ University of California, Davis. *RISC-V Instruction Encoder/Decoder*. `https://luplab.gitlab.io/rvcodecjs/`. Acessada em 05 de julho de 2025. 2021.

msyksphinz. *RV32I, RV64I Instructions*. `msyksphinz-self.github.io/riscv-isadoc/html/rvi.html`. Acessada em 05 de julho de 2025. 2019.

PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. 2.ª ed. Morgan Kaufmann. 2021.