Arquitetura de Computadores - RISC-V com SIMD

Carolina Santiago
Gustavo Roxo
Zuilho Segundo

Engenharia de Computação e Informação
Universidade Federal do Rio de Janeiro - UFRJ
Brasil
Julho 2023

Sumário

1 Introdução

O presente relatório tem por objetivo relatar o desenvolvimento de uma CPU que tenha suporte para a arquitetura RISC-V. Aliado a isso, também será desenvolvido um pipeline de execução, para que haja um ganho de performance na utilização dessa CPU e arquitetura. Os comando que a CPU irá suportar são: add, addi, auipc, sub, and, andi, or, ori, xor, xori, sll, slli, srl, srli, lw, lui, sw, jal, jalr, beq e bne. Além disso, a CPU não opera em modo supervisor e não opera durante a carga de dados, possuindo um sinal de reset.

Por fim, caso haja interesse, é possível encontrar os circuitos utilizados no seguinte link do GitHub: `https://github.com/carol-santiago/CPU-RISCV-EEL580`.

2 Especificações

Seguindo as especificações do pipeline, nossa CPU possui cinco etapas: Instruction Fetch (IF), Instruction Decode and Register File Read (ID), Execution and Address Calculation (EX), Data Memory Access (MEM) e Write Back (WB).

2.1 Instruction Fetch (IF)

Durante esse estágio, a instrução é obtida da memória de instruções com a ajuda de um endereçador. Nesse estágio o próximo endereço é calculado, porém não é imediatamente registrado, pois antes ainda é preciso verificar a possibilidade de ocorrência de um branch do endereço. No que tange os componente, serão necessários três: um PC (Program Counter), que será o endereçador, uma ROM, que representa a memória de instruções e um somador de 32 bits, para sempre adicionar 1 ao endereço atual da memória.

2.2 Instruction Decode and Register File Read (ID)

No segundo estágio, os conteúdos capturados anteriormente serão decodificados. Essas informações são codificadas da seguinte forma: registrador de destino (RD), onde será escrito o resultado da operação, registrador de origem (RS1 e RS2), de onde vem os operadores, imediato, que é uma constante de 12 a 20 bits que precisa ser estendida até possuir 32 bits, opcode, que define o tipo da instrução, e funct3/funct7, que auxiliam o opcode quando necessário.

Para os registrador RD, RS1 e RS2, vamos precisar de um Register File, uma matriz de 32 registradores. Para lidar com o imediato, vamos utilizar um Immediate Generator, que decodifica o imediato com base na instrução e completa ele até que possua 32 bits. Por fim, também é necessário um componente de controle central, que utiliza o opcode para gerar sinais de controle.

2.3 Executionand Address Calculation (EX)

Aqui, os registradores de origem e o imediato entram em uma ALU para serem utilizados de acordo com a operação definida com a ajuda do sinal de CONTROL, funct3 e funct7. Além disso, a ALU possui uma flag de 0, para ajudar com as instruções de branch. Para estas e outras Instruções que desviam o rumo do PC, também existe um somador nesta etapa que soma o valor de um imediato ao valor atual de PC, para definir o destino do branching.

2.4 Data Memory Access (MEM)

Durante esse estágio, a memória de dados, uma RAM, será acessada, tendo seus conteúdos lidos ou escritos com base nas entradas de controle, e endereçados conforme o resultado da ALU. Ainda nesse momento é tomada a decisão do próximo endereço de memória, se será PC+1 ou o endereço de branch calculado na ALU, baseado nas entradas de controle.

2.5 Write Back (WB)

Por fim, um MUX decide, com base em dois bits do sinal CONTROL, uma de quatro entradas (PC+1, PC+Imediato, resultado, dados) será registrada no RD na Register File. Por fim, um bit do sinal CONTROL que define o Write Enable da Register File é transmitido de volta, junto com os dados, caso seja para serem escritos ou não.

2.6 Pipelining e Hazard Detection

Para completar o pipeline, colocamos registradores entre cada uma dessas etapas. Eles armazenam as saídas da etapa anterior na borda de subida de clock e passam elas pra próxima etapa na borda de descida. Isso permite que o datapath realize 5 vezes mais instruções por ciclo de clock, uma para etapa descrita.

Também foi necessário criar dois módulos que lidam com "erros" no pipeline. Como, por exemplo, se fizermos um load no registrador x1 e quisermos usar o novo valor de x1 na próxima instrução. Nesse caso, dada a natureza do pipeline, x1 só assumiria seu novo valor alguns ciclos de clock depois. Os dois módulos desenvolvidos foram o de Harzard Detection e Forwarding Unit, que lidam com a com com esse problemas.

3 Desenvolvimento do Projeto

Após termos concluído o último projeto, adquirimos o conhecimento necessário para iniciar o novo projeto. Inicialmente, procedemos à definição de três novos opcodes, designados para cada categoria de instruções, compreendendo campos de 4 em 4 bits, 8 em 8 bits e 16 em 16 bits, respectivamente. Em sequência, introduzimos dois novos sinais de controle: "mode", representado por um único bit, responsável por determinar se a instrução é vetorial; e "vecSize", encarregado de especificar o tamanho da instrução.

Em relação aos opcodes associados ao trabalho anterior, foram atualizados com os seguintes valores: "mode"igual a 0 e "vecSize"igual a "XX".

	R-Type			I-Type			auipc		
	4 bits	8 bits	16 bits	4 bits	8 bits	16 bits	4 bits	8 bits	16 bits
Opcode	00000000	10000001	00000010	0000100	0000101	0000001	0001000	0001001	0001010
mode	1	1	1	1	1	1	1	1	1
vecSize	00	01	10	00	01	10	00	01	10

Tabela 1 - Os novos sinais de comando para cada opcode. Os sinais de comando especificados no último trabalho permanecem iguais entre opcodes que representam a mesma instrução.

mode	Define se a operação será vetorial ou não.
	0 = operação não vetorial
	1 = operação vetorial
vecSize	Define o tamanho das seções internas aos operandos
	00 = 4 bits; 01 = 8 bits; 10 = 16 bits
	32 bits não foi definido, pois para ser realizado basta que mode = 0

Tabela 2 - Descrição dos novos sinais de controle.

4 Explicação do código

4.1 ALU

O código VHDL apresenta uma entidade chamada ALU (Unidade Lógico-Aritmética), que representa a unidade de cálculo do processador responsável por realizar operações lógicas e aritméticas.

A ALU implementada suporta operações lógicas e aritméticas básicas, bem como operações vetoriais. As operações vetoriais são controladas pelo sinal *vecSize*, que define o tamanho do vetor e direciona as operações de acordo. O sinal de controle *control* determina a operação a ser realizada pela ALU. O resultado é selecionado de acordo com o modo (modo único ou modo vetorial) e é fornecido nas saídas *result* e *zero*, que indicam se o resultado é zero ou não.

4.1.1 Declaração da entidade ALU:

```
        entity ALU is
    port (
        A, B       : in std_logic_vector(31 downto 0);
        control    : in std_logic_vector(3 downto 0);
        mode       : in std_logic;
        vecSize    : in std_logic_vector(1 downto 0);
        result     : out std_logic_vector(31 downto 0);
        zero       : out std_logic
    );
end ALU;
```

4.1.2 Declaração dos sinais internos da ALU:

```
signal zero_out   : std_logic_vector(32 downto 0);
signal zero_vec   : std_logic_vector(32 downto 0);
signal result_out : std_logic_vector(31 downto 0);
signal result_vec : std_logic_vector(31 downto 0);
signal add_vec    : std_logic_vector(31 downto 0);
signal sub_vec    : std_logic_vector(31 downto 0);
signal shiftl_vec : std_logic_vector(31 downto 0);
signal shiftr_vec : std_logic_vector(31 downto 0);

signal A3116            : std_logic_vector(15 downto 0);
signal A1500            : std_logic_vector(15 downto 0);

...
```

Esses sinais internos são utilizados para realizar os cálculos internos da ALU. Eles são usados para armazenar resultados intermediários das operações aritméticas e lógicas. Além disso, existem sinais auxiliares para armazenar partes específicas do sinal de entrada A, que são usadas nos cálculos vetoriais.

4.1.3 Cálculos das operações vetoriais:

Em seguida, são realizadas as operações em modo vetorial. A operação é selecionada com base no tamanho do vetor (vecSize). Dependendo do tamanho do vetor, os operandos são divididos em partes menores e a adição é realizada em cada uma dessas partes. Depois, o resultado final da operação vetorial é selecionado com base no sinal de controle (control). Dependendo do valor do sinal de controle, um dos resultados vetoriais calculados anteriormente é selecionado.

```
result_vec <= add_vec when control = "0010" else
              sub_vec when control = "0110" else
              shiftl_vec when control = "0011" else
              shiftr_vec when control = "0111" else
              (others => '0');
```

4.2 ALU_Control

Representa um bloco responsável por gerar o sinal de controle da ALU (Unidade Lógico-Aritmética) com base nos sinais *Functs* e *AluOp*. Os sinais *Functs* e *AluOp* são utilizados para selecionar a operação desejada na ALU. O bloco ALU_Control recebe as entradas *Functs* e *AluOp*, que representam respectivamente o código da função e o código da operação a ser realizada na ALU. Com base nos valores desses sinais, o bloco gera o sinal de controle *Control*, que será utilizado para selecionar a operação correta na ALU. As condições definidas no código mapeiam as combinações possíveis de *Functs* e *AluOp* para suas respectivas operações. Caso nenhuma das condições seja satisfeita, o valor padrão será "0000", o que representa uma operação de AND.

```
        control <= "0010" when AluOp = "00" else
                        "0110" when AluOp = "01" else
                        "0010" when Functs = "0000" else
                        "0110" when Functs = "1000" else
                        "0011" when Functs = "0001" else
                        "0101" when Functs = "0100" else
                        "0111" when Functs = "0101" else
                        "0001" when Functs = "0110" else
                        "0000" when Functs = "0111";
```

Essa parte do código define a lógica de controle da ALU, que seleciona a operação apropriada com base nos sinais de entrada "AluOp"e "Functs". Cada combinação de valores para esses sinais corresponde a uma operação específica da ALU. A saída é representada pelo sinal "control", que indica a operação que será realizada pela ALU de acordo com os valores dos sinais de entrada.

4.3 Control_Unit

Representa um bloco responsável por gerar os sinais de controle para a unidade de controle do processador. Os sinais de controle determinam as operações que devem ser executadas pelos diversos componentes do processador com base no opcode da instrução. O bloco Control_Unit recebe como entrada o opcode, que representa o código da instrução que deve ser executada pelo processador. Com base nesse opcode, a unidade de controle gera os sinais de controle corretos para cada componente do processador (ALU, memória, desvios, etc.). Os sinais de controle são então utilizados para controlar a execução das operações apropriadas durante o ciclo de busca e execução das instruções. Cada bloco IF/THEN/ELSE no processo mapeia um opcode específico para seus respectivos sinais de controle. O último bloco de IF é um exemplo para uma instrução do tipo R-Type com operação de deslocamento à esquerda (shift left).

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

entity Control_Unit is
    port(
        opcode                      : IN  std_logic_vector(6 downto 0);
        AluSrc, blockA, RegWrite          : OUT std_logic;
        MemRead, MemWrite, Branch   : OUT std_logic;
        BranchNotEq, BrIncond                 : OUT std_logic;
        regToPC, mode                             : OUT std_logic;
        AluOp,regSrc, vecSize                 : OUT std_logic_vector(1 downto 0)
        );
end Control_Unit;

architecture TypeArchitecture of Control_Unit is

begin
        process(opcode)
```

```
        begin
            IF (opcode = "0110011") THEN
                AluSrc      <= '0';
                blockA      <= '0';
                RegWrite    <= '1';
                MemRead     <= '0';
                MemWrite    <= '0';
                Branch      <= '0';
                AluOp       <= "10";
                regSrc      <= "00";
                BranchNotEq <= '0';
                BrIncond    <= '0';
                regToPC     <= '0';
                mode        <= '0';
                vecSize     <= "00";
            END IF;
        end process;
end TypeArchitecture;
```

4.4 ImmGen

É responsável por gerar o imediato (*immediate value*) de uma instrução, que é a parte da instrução que contém dados ou constantes a serem utilizados na operação. O bloco ImmGen recebe como entrada *inst*, que é a instrução completa de 32 bits, e a partir do opcode desta instrução, gera o imediato apropriado e o fornece na saída *imm*. O bloco é utilizado para a extração do imediato de instruções de diferentes tipos: I-type (*immediate-type*), U-type (*upper immediate-type*), S-type (*store-type*), SB-type (*branch-type*) e UJ-type (*jump-type*). Cada bloco IF/THEN/ELSE no processo mapeia um opcode específico para a extração do imediato correto, conforme o tipo da instrução. Os bits do imediato são copiados ou concatenados de acordo com a instrução específica para formar o valor final do imediato *imm_interno*, que é então fornecido como a saída *imm*.

```
-- Bloco de seleção para gerar o imediato com base no opcode da instrução
        if (opcode = "0010011" or opcode = "1100110" or opcode = "0000100"
            or opcode = "0000101" or opcode = "0000001") then --I type
            imm_interno(11 downto 0)  <= inst(31 downto 20);
            imm_interno(31 downto 12) <= (others => inst(31));
        elsif (opcode = "1110111" or opcode = "0110111" or opcode = "0001000"
            or opcode = "0001001" or opcode = "0001010") then -- U type
            imm_interno(31 downto 12) <= inst(31 downto 12);
        elsif (opcode = "0100011") then -- S type
            imm_interno(11 downto 5) <= inst(31 downto 25);
            imm_interno(4 downto 0)  <= inst(11 downto 7);
            imm_interno(31 downto 12) <= (others => inst(31));
        elsif (opcode = "1100111" or opcode = "1100011") then
            imm_interno(12) <= inst(31);
            imm_interno(11) <= inst(7);
            imm_interno(10 downto 5) <= inst(30 downto 25);
            imm_interno(4 downto 1) <= inst(11 downto 8);
            imm_interno(31 downto 13) <= (others => inst(31));
        elsif (opcode = "1101111") then -- UJ type
            imm_interno(20) <= inst(31);
            imm_interno(19 downto 12) <= inst(19 downto 12);
            imm_interno(11) <= inst(20);
            imm_interno(10 downto 1) <= inst(30 downto 21);
        end if;
    end process;
```

```
    -- Saída do imediato gerado.
    imm <= imm_interno;
```

4.5 PC2

Representa um bloco responsável por implementar um registrador de 32 bits com clock e sinal de reset. A entidade PC2 representa um bloco de registro síncrono de 32 bits. A entrada consiste em um sinal de clock *clk*, um sinal de reset *reset* e um vetor de dados de entrada *pc_in* de 32 bits. A saída é um vetor de dados de 32 bits *pc_out*. A arquitetura TypeArchitecture contém um processo que é sensível aos sinais *clk* e *reset*. No evento de borda de subida do sinal de clock (*rising_edge(clk)*), o bloco verifica se o valor de *pc_in* é desconhecido (U). Se todos os bits forem desconhecidos, a saída *pc_out* é zerada (todos os bits definidos para '0'). Caso contrário, o valor de *pc_in* é copiado para a saída *pc_out*. Além disso, o bloco também possui uma verificação para o sinal de reset. Se o sinal de reset for '1', o dado de saída *pc_out* é zerado (todos os bits definidos para '0'). Isso garante que o registrador seja resetado para um valor conhecido sempre que o sinal de reset for ativado. Portanto, esse bloco representa um registrador de 32 bits com capacidade de reset e tratamento de valores desconhecidos (U).

```
BEGIN
PROCESS (clk, reset)
BEGIN
        IF (rising_edge(clk)) THEN
                if (pc_in = "UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU") then
                -- Se a entrada for desconhecida (U), o dado de saída é zerado.
                pc_out <= (others => '0');
                else
                -- Caso contrário, copia o dado de entrada para a saída.
                pc_out <= pc_in;
            end if;
        END IF;
        IF (reset = '1') THEN
                -- Se o sinal de reset for '1', o dado de saída é zerado.
                pc_out <= (others => '0');
        END IF;
END PROCESS;
```

4.6 Somador

A entidade "somador"representa um bloco combinacional que realiza a soma de dois números de 32 bits. A entrada consiste em dois vetores de dados de 32 bits (A e B), e a saída é outro vetor de 32 bits (Z) que representa o resultado da soma. A arquitetura TypeArchitecture contém uma única atribuição (*Z ¡= A + B;*), onde o valor de Z é definido como a soma dos valores presentes em A e B. Essa operação de soma é realizada usando a biblioteca *IEEE.std_logic_unsigned.ALL*, que fornece funções e operações aritméticas para vetores de sinal (*std_logic_vector*) sem a necessidade de conversões explícitas. Portanto, o bloco "somador"é responsável por realizar a adição dos valores presentes em A e B, fornecendo o resultado da soma na saída Z. Vale destacar que essa soma é combinacional, o que significa que ela é calculada instantaneamente, assim que os valores de A e B forem definidos.

```
-- O sinal de saída Z recebe a soma dos valores de A e B.
  Z <= A + B;
```

4.7 Register_file

A entidade "register_file"representa um banco de registradores com 32 registros de 32 bits cada. A entidade possui diversas portas de entrada e saída que permitem operar e interagir com o banco de registradores:

Portas de Saída:

- *outA*: Saída do registrador A.
- *outB*: Saída do registrador B.
- *outRegManual*: Saída de um registrador selecionado manualmente.

Portas de Entrada:

- *input*: Dado de entrada para escrever em um registrador.
- *regSelManual*: Seleção manual de um registrador para a saída *outRegManual*.
- *writeEnable*: Sinal de habilitação de escrita em um registrador.
- *regASel*: Seleção do registrador A para leitura.
- *regBSel*: Seleção do registrador B para leitura.
- *writeRegSel*: Seleção do registrador para escrita.
- *clk*: Sinal de clock.

A arquitetura TypeArchitecture possui uma declaração do tipo *registerFile*, que é um array de 32 elementos, cada elemento sendo um vetor de 32 bits. Esse array representa os registradores do banco. O processo *regFile* é sensível ao sinal de clock (*clk*) e é responsável por realizar as operações de leitura e escrita nos registradores. Ele também implementa a lógica de bypass, permitindo que os dados escritos em um registrador possam ser lidos instantaneamente nos registradores A e B, caso eles sejam selecionados. Além disso, o processo também permite a seleção manual de um registrador e a saída do seu valor por meio da porta *outRegManual*.

```
architecture TypeArchitecture of register_file is
  -- Declaração de um tipo array que representa o banco de registradores com 32 registros de 32 bi
  type registerFile is array(0 to 31) of std_logic_vector(31 downto 0);
  -- Declaração de um sinal (variável interna) chamado "registers" do tipo registerFile
  signal registers : registerFile :=
   ("00000000000000000000000000011111","00000000000000000000000000000001",
   ...[REDUZIDO PARA FORMATAÇÃP - CHECAR CÓDIGO FONTE],"00000000000000000000000000000000");
begin
  regFile : process (clk) is
  begin
    if rising_edge(clk) then
    if (registers(to_integer(unsigned(regASel))) = "UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU") THEN
    outA <= (others => '0');  -- Se o registro for inválido (contém "U"), a saída é zerada
    else
    outA <= registers(to_integer(unsigned(regASel)));  -- Caso contrário, a saída é o valor do reg
    end if;
    IF (registers(to_integer(unsigned(regBSel))) = "UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU") THEN
    outB <= (others => '0');  -- Se o registro for inválido (contém "U"), a saída é zerada
    else
    outB <= registers(to_integer(unsigned(regBSel)));  -- Caso contrário, a saída é o valor do reg
    end if;
    if writeEnable = '1' then
    -- Se o sinal de escrita estiver habilitado (writeEnable = '1'), então o dado de entrada "input
    registers(to_integer(unsigned(writeRegSel))) <= input;  -- Write
    -- Além disso, é feito o bypass para leitura dos registradores A e B
    if regASel = writeRegSel then  -- Bypass para leitura do registrador A
        outA <= input;  -- O valor de "input" é direcionado para a saída do registrador A
    end if;
    if regBSel = writeRegSel then  -- Bypass para leitura do registrador B
```

```
        outB <= input;  -- O valor de "input" é direcionado para a saída do registrador B
    end if;
    end if;
end if;
end process;
  -- seleção manual
  -- Saída de um registrador selecionado manualmente com base no valor de "regSelManual"
  outRegManual <= registers(to_integer(unsigned(regSelManual)));
end TypeArchitecture;
```

4.8 IFID

Apresenta a descrição de uma entidade chamada "IFID"(Instruction Fetch / Instruction Decode Pipeline Register). Essa entidade representa o estágio IF/ID de um pipeline, que é responsável por armazenar as informações relevantes da instrução buscada na memória e fornecer essas informações para o próximo estágio do pipeline (ID/EX - Instruction Decode / Execute Pipeline Register).

A entidade "IFID"possui as seguintes portas:

- *clk*: Sinal de clock usado para sincronizar as operações.
- *pcIn*: Entrada de 32 bits representando o valor do Program Counter (PC) no estágio IF/ID.
- *pcPl4In*: Entrada de 32 bits representando o valor do Program Counter mais 4 (PC + 4) no estágio IF/ID.
- *instIn*: Entrada de 32 bits representando a instrução buscada na memória no estágio IF/ID.
- *pcOut*: Saída de 32 bits representando o valor do Program Counter (PC) que é enviado para o próximo estágio do pipeline (ID/EX).
- *pcPl4Out*: Saída de 32 bits representando o valor do Program Counter mais 4 (PC + 4) que é enviado para o próximo estágio do pipeline (ID/EX).
- *instOut*: Saída de 32 bits representando a instrução que é enviada para o próximo estágio do pipeline (ID/EX).

O registrador interno *IDIF* é de 96 bits e é usado para armazenar as informações relevantes da instrução e dos PCs durante a borda de subida do clock (*rising edge*) e fornecer essas informações nas saídas durante a borda de descida do clock (*falling edge*). Essa implementação representa um estágio de pipeline IF/ID simples, que retém a instrução buscada e os PCs para fornecer aos próximos estágios do pipeline. Note que, em implementações reais de processadores, esses registradores de pipeline podem ter uma estrutura mais complexa, como a inclusão de sinais de controle e lógica de controle adicionais para lidar com atrasos e outros detalhes de projeto.

```
IF (rising_edge(clk)) THEN
-- No pulso de subida do clock, os dados de entrada são armazenados no registrador IDIF
    IDIF(31 DOWNTO 0) <= instIn;
    IDIF(63 DOWNTO 32) <= pcIn;
    IDIF(95 DOWNTO 64) <= pcPl4In;
END IF;
IF (falling_edge(clk)) THEN
    pcOut <= IDIF(63 DOWNTO 32);
    instOut <= IDIF(31 DOWNTO 0);
    pcPl4Out <= IDIF(95 DOWNTO 64);
END IF;
```

4.9 IDEX

A entidade chamada "IDEX"(Instruction Decode / Execute Pipeline Register) representa o estágio IDEX de um pipeline, que é responsável por armazenar as informações relevantes necessárias para a etapa de execução das instruções.

A entidade "IDEX"possui as seguintes portas:

- *clk*: Sinal de clock usado para sincronizar as operações.
- *pcIn*: Entrada de 32 bits representando o valor do Program Counter (PC) no estágio IDEX.
- *read1In*: Entrada de 32 bits representando o valor do registrador de leitura 1 no estágio IDEX.
- *read2In*: Entrada de 32 bits representando o valor do registrador de leitura 2 no estágio IDEX.
- *immGenIn*: Entrada de 32 bits representando o valor gerado pelo bloco "ImmGen"(gerador de imediatos) no estágio IDEX.
- *aluControlin*: Entrada de 4 bits representando o controle da Unidade Lógica e Aritmética (ALU) no estágio IDEX.
- *wbAddIn*: Entrada de 5 bits representando o endereço do registrador de escrita de volta no estágio IDEX.
- *WBin*: Entrada de 3 bits representando o controle do estágio de escrita de volta (WB) no estágio IDEX.
- *Min*: Entrada de 6 bits representando o controle do estágio de memória (M) no estágio IDEX.
- *EXin*: Entrada de 7 bits representando o controle do estágio de execução (EX) no estágio IDEX.
- *pcPl4In*: Entrada de 32 bits representando o valor do Program Counter mais 4 (PC + 4) no estágio IDEX.
- *pcOut*: Saída de 32 bits representando o valor do Program Counter (PC) que é enviado para o próximo estágio do pipeline (EX/M).
- *read1Out*: Saída de 32 bits representando o valor do registrador de leitura 1 que é enviado para o próximo estágio do pipeline (EX/M).
- *read2Out*: Saída de 32 bits representando o valor do registrador de leitura 2 que é enviado para o próximo estágio do pipeline (EX/M).
- *immGenOut*: Saída de 32 bits representando o valor gerado pelo bloco "ImmGen"(gerador de imediatos) que é enviado para o próximo estágio do pipeline (EX/M).
- *aluControlout*: Saída de 4 bits representando o controle da Unidade Lógica e Aritmética (ALU) que é enviado para o próximo estágio do pipeline (EX/M).
- *wbAddOut*: Saída de 5 bits representando o endereço do registrador de escrita de volta que é enviado para o próximo estágio do pipeline (EX/M).
- *WBout*: Saída de 3 bits representando o controle do estágio de escrita de volta (WB) que é enviado para o próximo estágio do pipeline (EX/M).
- *Mout*: Saída de 6 bits representando o controle do estágio de memória (M) que é enviado para o próximo estágio do pipeline (EX/M).
- *EXout*: Saída de 7 bits representando o controle do estágio de execução (EX) que é enviado para o próximo estágio do pipeline (EX/M).

- *pcPl4Out*: Saída de 32 bits representando o valor do Program Counter mais 4 (PC + 4) que é enviado para o próximo estágio do pipeline (EX/M).

O registrador interno *idex_s* é de 185 bits e é usado para armazenar as informações relevantes para o estágio de execução (EX/M) das instruções durante a borda de subida do clock (*rising edge*) e fornecer essas informações nas saídas durante a borda de descida do clock (*falling edge*). Essa implementação representa um estágio de pipeline IDEX simples, que retém as informações necessárias para a etapa de execução das instruções e as fornece para o próximo estágio do pipeline (EX/M). Note que, em implementações reais de processadores, esses registradores de pipeline podem ter uma estrutura mais complexa, como a inclusão de sinais de controle e lógica de controle adicionais para lidar com atrasos e outros detalhes de projeto.

```
IF (rising_edge(clk)) THEN
    -- No pulso de subida do clock, os dados de entrada são armazenados no registrador idex_s
    idex_s(31 DOWNTO 0)    <= pcIn;
    idex_s(63 DOWNTO 32)   <= read1In;
    idex_s(95 DOWNTO 64)   <= read2In;
    idex_s(127 DOWNTO 96)  <= immGenIn;
    idex_s(131 DOWNTO 128) <= aluControlin;
    idex_s(136 DOWNTO 132) <= wbAddIn;
    idex_s(139 downto 137) <= WBin;
    idex_s(145 downto 140) <= Min;
    idex_s(152 downto 146) <= EXin;
    idex_s(184 downto 153) <= pcPl4In;
END IF;
IF (falling_edge(clk)) THEN
    -- No pulso de descida do clock, os dados armazenados no registrador idex_s são disponibiliza
    pcOut    <= idex_s(31 DOWNTO 0);
    read1Out <= idex_s(63 DOWNTO 32);
    read2Out <= idex_s(95 DOWNTO 64);
    immGenOut <= idex_s(127 DOWNTO 96);
    aluControlout <= idex_s(131 DOWNTO 128);
    wbAddOut <= idex_s(136 DOWNTO 132);
    WBout <= idex_s(139 downto 137);
    Mout <= idex_s(145 downto 140);
    EXout <= idex_s(152 downto 146);
    pcPl4Out <= idex_s(184 downto 153);
END IF;
```

4.10 EXMEM

A entidade "EXMEM"(Execute / Memory Pipeline Register) representa o estágio EXMEM de um pipeline, que é responsável por armazenar as informações relevantes necessárias para a etapa de acesso à memória das instruções.

A entidade "EXMEM"possui as seguintes portas:

- *clk*: Sinal de clock usado para sincronizar as operações.
- *sumIn*: Entrada de 32 bits representando o resultado da Unidade Lógica e Aritmética (ALU) no estágio EXMEM.
- *zeroIn*: Entrada de um bit que indica se a saída da ALU é zero no estágio EXMEM.
- *aluIn*: Entrada de 32 bits representando o resultado da ALU no estágio EXMEM.
- *read2In*: Entrada de 32 bits representando o valor do registrador de leitura 2 no estágio EXMEM.
- *wbAddIn*: Entrada de 5 bits representando o endereço do registrador de escrita de volta no estágio EXMEM.

- *WBin*: Entrada de 3 bits representando o controle do estágio de escrita de volta (WB) no estágio EXMEM.
- *Min*: Entrada de 6 bits representando o controle do estágio de memória (M) no estágio EXMEM.
- *pcPl4In*: Entrada de 32 bits representando o valor do Program Counter mais 4 (PC + 4) no estágio EXMEM.
- *pcOut*: Saída de 32 bits representando o valor do Program Counter (PC) que é enviado para o próximo estágio do pipeline (M/WB).
- *zeroOut*: Saída de um bit que indica se a saída da ALU é zero e é enviado para o próximo estágio do pipeline (M/WB).
- *aluOut*: Saída de 32 bits representando o valor da ALU que é enviado para o próximo estágio do pipeline (M/WB).
- *read2Out*: Saída de 32 bits representando o valor do registrador de leitura 2 que é enviado para o próximo estágio do pipeline (M/WB).
- *wbAddOut*: Saída de 5 bits representando o endereço do registrador de escrita de volta que é enviado para o próximo estágio do pipeline (M/WB).
- *WBout*: Saída de 3 bits representando o controle do estágio de escrita de volta (WB) que é enviado para o próximo estágio do pipeline (M/WB).
- *Mout*: Saída de 6 bits representando o controle do estágio de memória (M) que é enviado para o próximo estágio do pipeline (M/WB).
- *pcPl4Out*: Saída de 32 bits representando o valor do Program Counter mais 4 (PC + 4) que é enviado para o próximo estágio do pipeline (M/WB).

Essa entidade "EXMEM"atua como um registrador de pipeline no estágio EXMEM, armazenando os dados resultantes do estágio de execução (EX) e fornecendo-os para o próximo estágio do pipeline (M/WB). Note que, como nas outras etapas do pipeline, a implementação real do registrador de pipeline pode ser mais complexa, considerando sinais de controle e lógica de controle adicionais para melhorar o desempenho e a eficiência do processador.

```
PROCESS (clk)
BEGIN
    IF (rising_edge(clk)) THEN
-- No pulso de subida do clock, os dados de entrada são armazenados no
-- registrador exmem_s
        exmem_s(31 DOWNTO 0)   <= sumIn;
        exmem_s(32)  <= zeroIn;
        exmem_s(64 DOWNTO 33)  <= aluIn;
        exmem_s(96 DOWNTO 65) <= read2In;
        exmem_s(101 DOWNTO 97) <= wbAddIn;
        exmem_s(104 DOWNTO 102) <= WBin;
        exmem_s(110 DOWNTO 105) <= Min;
        exmem_s(142 downto 111) <= pcPl4In;
    END IF;
    IF (falling_edge(clk)) THEN
-- No pulso de descida do clock, os dados armazenados no registrador
-- exmem_s são disponibilizados nas saídas
        pcOut    <= exmem_s(31 DOWNTO 0);

        if (exmem_s(32) = 'U') then
            zeroOut <= '0';
```

```
        else
            zeroOut <= exmem_s(32);
        end if;

        if (exmem_s(64 DOWNTO 33) = "UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU") then
            aluOut <= "00000000000000000000000000000000";
        else
            aluOut <= exmem_s(64 DOWNTO 33);
        end if;

        read2Out <= exmem_s(96 DOWNTO 65);
        wbAddOut <= exmem_s(101 DOWNTO 97);

        if (exmem_s(110 DOWNTO 105) = "UUUUUU") then
            Mout <= "000000";
        else
            Mout <= exmem_s(110 DOWNTO 105);
        end if;

        WBout <= exmem_s(104 DOWNTO 102);

        pcPl4Out <= exmem_s(142 downto 111);


    END IF;
END PROCESS;
```

4.11 MEMWB

A entidade "MEMWB"(Memory / Write-Back Pipeline Register) representa o estágio MEMWB de um pipeline, que é responsável por armazenar as informações relevantes necessárias para a etapa de escrita de volta das instruções.

A entidade "MEMWB"possui as seguintes portas:

- *clk*: Sinal de clock usado para sincronizar as operações.
- *aluIn*: Entrada de 32 bits representando o valor da ALU no estágio MEMWB.
- *read2In*: Entrada de 32 bits representando o valor do registrador de leitura 2 no estágio MEMWB.
- *wbAddIn*: Entrada de 5 bits representando o endereço do registrador de escrita de volta no estágio MEMWB.
- *WBin*: Entrada de 3 bits representando o controle do estágio de escrita de volta (WB) no estágio MEMWB.
- *pcPl4In*: Entrada de 32 bits representando o valor do Program Counter mais 4 (PC + 4) no estágio MEMWB.
- *pcOut*: Saída de 32 bits representando o valor do Program Counter (PC) que é enviado para o próximo estágio do pipeline (WB).
- *aluOut*: Saída de 32 bits representando o valor da ALU que é enviado para o próximo estágio do pipeline (WB).
- *read2Out*: Saída de 32 bits representando o valor do registrador de leitura 2 que é enviado para o próximo estágio do pipeline (WB).

- *wbAddOut*: Saída de 5 bits representando o endereço do registrador de escrita de volta que é enviado para o próximo estágio do pipeline (WB).
- *WBout*: Saída de 3 bits representando o controle do estágio de escrita de volta (WB) que é enviado para o próximo estágio do pipeline (WB).
- *pcPl4Out*: Saída de 32 bits representando o valor do Program Counter mais 4 (PC + 4) que é enviado para o próximo estágio do pipeline (WB).

Essa entidade "MEMWB"atua como um registrador de pipeline no estágio MEMWB, armazenando os dados resultantes do estágio de acesso à memória (M) e fornecendo-os para a etapa de escrita de volta (WB) do pipeline. Note que, como nas outras etapas do pipeline, a implementação real do registrador de pipeline pode ser mais complexa, considerando sinais de controle e lógica de controle adicionais para melhorar o desempenho e a eficiência do processador.

```
PROCESS (clk)
BEGIN
    IF (rising_edge(clk)) THEN
        memwb_s(31 DOWNTO 0)   <= readIn;
        memwb_s(63 DOWNTO 32)  <= aluIn;
        memwb_s(68 DOWNTO 64)  <= wbAddIn;
        memwb_s(71 DOWNTO 69)  <= WBin;
        memwb_s(103 downto 72) <= pcPl4In;
        memwb_s(135 downto 104) <= pcPlIIn;

    END IF;
    IF (falling_edge(clk)) THEN
        readOut    <=  memwb_s(31 DOWNTO 0);
        aluOut  <=  memwb_s(63 DOWNTO 32);
        wbAddOut <= memwb_s(68 DOWNTO 64);
        WBout <= memwb_s(71 DOWNTO 69);
        pcPl4Out <= memwb_s(103 downto 72);
        pcPlIOut <= memwb_s(135 downto 104);
    END IF;
END PROCESS;
```

5 Resultados e Conclusões

Durante o desenvolvimento do trabalho fomos capazes de compreender as partes fundamentais do funcionamento de um processador capaz de suportar a arquitetura RISC-V. Como falado na introdução, o esquemático e o circuito desenvolvido no logisim podem ser encontrados no link disposto, no GitHub. Infelizmente nossa simulação utilizando o logisim não foi capaz de funcionar como o esperado, mas ainda assim estamos enviando, mostrando o que foi desenvolvido.