Arquitetura de Computadores - RISC-V

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 Desenvolvimento

Iniciamos desenvolvendo os elementos que não possuíam dependências em relação a lógica externa. Isso inclui os registradores de pipeline, somadores, Unidade Lógica e Aritmética (ALU), Banco de Registradores, Contador de Programa (PC) e Gerador de Imediato. Os códigos desses componentes podem ser encontrados na seção "Explicação do Código".

2.1 Definindo as Instruções

	R-Type	I-Type	sw	beq	bnq	jalr	jal	lw	lui	auipc
opcode	0110011	0010011	0100011	1100111	1100011	1100110	1101111	0000011	0110111	1110111
ALUSrc	0	1	1	0	0	1	X	1	X	X
BlockA	0	0	0	0	0	0	0	0	1	0
ALUOp	10	11	00	01	01	00	00	00	00	00
RegSrc	00	00	XX	XX	XX	10	10	01	00	11
RegWrite	1	1	0	0	0	1	1	1	1	1
MemWrite	0	0	1	0	0	0	0	0	0	0
MemRead	0	0	0	0	0	0	0	1	0	0
Branch	0	0	0	1	0	1	X	0	0	X
BrNotEq	0	0	0	0	1	0	X	0	0	X
BrIncond	0	0	0	0	0	0	1	0	0	1
RegToPC	0	0	0	0	0	1	0	0	0	0

Tabela 1: relaciona o tipo de instrução aos seus sinais de controle. Para as operações de tipo I que não estão listadas separadamente, elas seguem a coluna "I-Type".

opcode	O código de operação da instrução
ALUSrc	Se o segundo operando da ALU deve ser o segundo operando lido na *Register File* (ALUSrc = 0) ou o imediato (ALUSrc = 1)
BlockA	Se o primeiro operando da ALU deve ser o primeiro operando lido na *Register File* (BlockA = 0) ou 0 (BlockA = 1); utilizado somente para lui, para pouparmos fios no WB
ALUOp	Será passado para ALUControl para definir a operação a ser realizada na ALU (00 → Soma; 01 → Subtração; 1X → Definido por func3/func7)
RegSrc	Define o que será a entrada de input da *Register File* (00 → Resultado da ALU; 01 → Dados lidos na memória de dados; 10 → PC+1; 11 → PC+Imediato)
RegWrite	*Write Enable* para a *Register File*
MemWrite	*Write Enable* para a memória de dados
MemRead	*Read Enable* para a memória de dados
Branch	Se estiver ativo e a flag de zero na ALU também for ativa, o PC irá para o destino do branch
BrNotEq	Se estiver ativo e a flag de zero na ALU **não** estiver ativa, o PC irá para o destino do branch
BrIncond	O PC irá incondicionalmente para o destino do branch
RegToPC	O PC irá incondicionalmente para o resultado da operação na ALU

Tabela 2: Explica cada sinal de controle.

3 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).

3.1 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.

3.2 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.

3.3 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.

3.4 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.

3.5 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.

3.6 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.

4 Explicação do Código

4.1 CPU Base - Sem unidades de Hazard ou Fowarding

4.1.1 ALU

A ALU (Unidade Lógica e Aritmética) é um módulo responsável por realizar operações lógicas e aritméticas em um processador. O módulo 'ALU' possui as seguintes portas de entrada e saída:

- Portas de Entrada:
  - 'A', 'B': Entradas de 32 bits representando os operandos para as operações da ALU.
  - 'control': Entrada de 4 bits representando o sinal de controle para selecionar a operação a ser realizada.
- Portas de Saída:
  - 'result': Saída de 32 bits representando o resultado da operação.
  - 'zero': Saída de 1 bit que é '1' se o resultado da operação for zero, caso contrário, é '0'.

```
result_out <= (A + B) when control = "0010" else
    (A - B) when control = "0110" else
    (A XOR B) when control = "0101" else
    (A OR B) when control = "0001" else
```

```
    (A AND B) when control = "0000" else
    (std_logic_vector(shift_left(unsigned(A), to_integer(unsigned(B)))))
    when control = "0011" else
    (std_logic_vector(shift_right(unsigned(A), to_integer(unsigned(B)))))
    when control = "0111";

result <= result_out;
zero_out(0) <= '0';
G2: for I in 1 to 32 generate
    zero_out(I) <= zero_out(I - 1) or result_out(I - 1);
end generate;
zero <= not zero_out(32);
```

A arquitetura do módulo 'ALU' implementa a operação definida pelo sinal de controle 'control'. Por exemplo, se 'control = "0010"', a ALU realiza uma soma (A + B), se 'control = "0110"', realiza uma subtração (A - B), e assim por diante. A saída 'result_out' armazena o resultado da operação e é atribuído à saída 'result'. O sinal 'zero_out' é usado para verificar se o resultado é zero, e o bit mais significativo desse sinal é atribuído à saída 'zero'.

4.1.2 ALU_Control

O módulo 'ALU_Control' é responsável por gerar o sinal de controle para a ALU com base nos campos da instrução recebida. Ele possui as seguintes portas:

- Portas de Entrada:
  - 'Functs': Entrada de 4 bits representando os códigos de função da ALU.
  - 'AluOp': Entrada de 2 bits representando os códigos de operação da ALU.
- Portas de Saída:
  - 'Control': Saída de 4 bits representando o sinal de controle para a ALU.

```
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";
```

A arquitetura do módulo 'ALU_Control' utiliza combinações condicionais para definir o valor do sinal de controle 'Control' com base nos códigos de função 'Functs' e nos códigos de operação 'AluOp'. Cada combinação condicional corresponde a uma operação específica da ALU.

4.1.3 Control_Unit

A 'Control_Unit' (Unidade de Controle) é responsável por gerar os sinais de controle para várias unidades dentro do processador com base no opcode da instrução. Ela possui as seguintes portas de entrada e saída:

- Portas de Entrada:
  - 'opcode': Entrada de 7 bits representando o código de operação da instrução.
- Portas de Saída:

- ‘AluSrc‘, ‘blockA‘, ‘RegWrite‘: Saídas de 1 bit representando os sinais de controle individuais.
- ‘MemRead‘, ‘MemWrite‘, ‘Branch‘, ‘BranchNotEq‘, ‘BrIncond‘, ‘regToPC‘: Saídas de 1 bit representando outros sinais de controle individuais.
- ‘AluOp‘, ‘regSrc‘: Saídas de 2 bits representando códigos de operação da ALU e seleção de registradores.

```
IF (opcode = "0110011") THEN
-- R-Type sem imediato (add, sub, and, or, xor, slt)
    AluSrc      <= '0';
    -- Seleção de fonte para a ALU (0: A_i, 1: Immediato)
    blockA      <= '0';
    -- Seleção de origem do registrador A (0: rs1, 1: zero)
    RegWrite    <= '1';
    -- Habilita escrita no registrador destino
    MemRead     <= '0';
    -- Habilita leitura de memória
    MemWrite    <= '0';
    -- Habilita escrita na memória
    Branch      <= '0';
    -- Habilita desvio condicional
    AluOp       <= "10";
    -- Código de operação da ALU para R-Type (00: and, 01: or, 10: add, 11: sub)
    regSrc      <= "00";
    -- Seleção de registradores para R-Type (00: rs1, 01: rs2, 10: rd, 11: zero)
    BranchNotEq <= '0';
    -- Habilita desvio condicional se as entradas forem diferentes
    BrIncond    <= '0';
    -- Habilita desvio incondicional
    regToPC     <= '0';
    -- Seleção de registrador para atualizar o PC (0: PC+4, 1: rd)
END IF;
...
```

A arquitetura do módulo ‘Control_Unit‘ usa processamento condicional (com base no ‘opcode‘) para definir os valores dos sinais de controle, determinando quais operações e unidades dentro do processador devem ser habilitadas para uma instrução específica.

4.1.4 ImmGen

O módulo ‘ImmGen‘ (Unidade Geradora de Imediatos) é responsável por extrair o valor imediato de uma instrução e estendê-lo para o tamanho adequado. Ele possui as seguintes portas:

- Portas de Entrada:
  - ‘inst‘: Entrada de 32 bits representando a instrução.
- Portas de Saída:
  - ‘imm‘: Saída de 32 bits representando o valor imediato gerado.

```
if (opcode = "0010011" or opcode = "1100110") then
-- Tipo I (load-immediate, jump-register)
    imm_interno(11 downto 0)  <= inst(31 downto 20);
    -- Extrai o imediato de 11 bits
    imm_interno(31 downto 12) <= (others => inst(31));
    -- Sinal extendido para os bits mais significativos
```

```
elsif (opcode = "1110111" or opcode = "0110111") then
-- Tipo U (lui, auipc)
    imm_interno(31 downto 12) <= inst(31 downto 12);
    -- Extrai o imediato de 20 bits
elsif (opcode = "0100011") then  -- Tipo S (store)
    imm_interno(11 downto 5)  <= inst(31 downto 25);
    -- Extrai o imediato de 7 bits (5-11)
    imm_interno(4 downto 0)   <= inst(11 downto 7);
    -- Extrai o imediato de 5 bits (0-4)
    imm_interno(31 downto 12) <= (others => inst(31));
    -- Sinal extendido para os bits mais significativos
    .
    .
    .
```

A arquitetura do módulo 'ImmGen' utiliza processamento condicional com base no opcode da instrução para determinar o tipo de instrução e extrair o valor imediato adequado, estendendo-o para o tamanho correto (32 bits) conforme necessário.

4.1.5 PC2

O módulo `PC2` é responsável por manter o valor do Program Counter (PC) atualizado. O PC é um registrador que armazena o endereço da próxima instrução a ser buscada e executada no processador. Ele possui as seguintes portas:

- **Portas de Entrada:**
  - `clk`: Entrada do sinal de clock para sincronizar as operações.
  - `reset`: Entrada do sinal de reset para reiniciar o PC.
  - `pc_in`: Entrada de 32 bits representando o valor atual do PC.
- **Portas de Saída:**
  - `pc_out`: Saída de 32 bits representando o valor atualizado do PC.

```
IF (rising_edge(clk)) THEN
-- Detecta a borda de subida do sinal de clock
    if (pc_in = "UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU") then
    -- Verifica se o valor atual do PC é desconhecido (não inicializado)
        pc_out <= (others => '0');
        -- Se for desconhecido, zera o valor do PC
    elsif (writeEnableL = '0') then
    -- Verifica se o sinal de habilitação de escrita está desabilitado
        pc_out <= pc_in;
        -- Se estiver desabilitado, o PC mantém o valor atual (não ocorre atualização)
    end if;
END IF;
IF (reset = '1') THEN
-- Detecta a ocorrência de um reset (sinal de reset em nível alto)
    pc_out <= (others => '0');
    -- Nesse caso, zera o valor do PC
END IF;
```

A arquitetura do módulo `PC2` utiliza a borda de subida do clock para atualizar o valor do PC. Durante o reset (sinal `reset = '1'`), o PC é zerado. Caso contrário, o valor do PC é copiado da entrada `pc_in` para a saída `pc_out` na borda de subida do clock.

4.1.6 somador

O módulo `somador` é um somador simples que realiza a soma de dois valores de entrada `A` e `B`. Ele possui as seguintes portas:

- **Portas de Entrada:**
  - `A`, `B`: Entradas de 32 bits representando os valores a serem somados.
- **Portas de Saída:**
  - `Z`: Saída de 32 bits representando o resultado da soma.

```
Z <= A + B; -- Saída Z é a soma dos vetores de entrada A e B
```

A arquitetura do módulo `somador` simplesmente atribui à saída `Z` o resultado da soma das entradas `A` e `B`.

4.1.7 register_file

A entidade `register_file` representa um banco de registradores, que é um componente fundamental em um processador. O banco de registradores armazena os valores dos registradores do processador e permite a leitura e escrita nesses valores. Algumas das portas presentes nessa entidade são:

- **Saídas:**
  - `outA`, `outB`: Saídas dos valores armazenados nos registradores A e B, respectivamente.
  - `outRegManual`: Saída do registrador selecionado manualmente.
- **Entradas:**
  - `input`: Entrada de dados para escrita em um registrador específico.
  - `regSelManual`: Entrada para seleção manual de um registrador de saída.
  - `writeEnable`: Sinal de habilitação de escrita no banco de registradores.
  - `regASel`, `regBSel`: Entradas para seleção dos registradores A e B para leitura.
  - `writeRegSel`: Entrada para seleção do registrador para escrita.
  - `clk`: Sinal de clock.

```
if rising_edge(clk) then
  -- Read A and B before bypass
  -- Leitura dos registradores A e B antes do bypass
  if (registers(to_integer(unsigned(regASel))) = "UUUUUUU...UUUUUUUUUUU") THEN
    outA <= (others => '0');
    -- Se o valor for "UUUU..." (indeterminado), a saída A será "0000..."
  ELSE
    outA <= registers(to_integer(unsigned(regASel)));
    -- Caso contrário, a saída A será o valor do registrador selecionado
  END IF;
  IF (registers(to_integer(unsigned(regBSel))) = "UUUUUUUU...UUUUUU") THEN
    outB <= (others => '0');
    -- Se o valor for "UUUU..." (indeterminado), a saída B será "0000..."
  ELSE
    outB <= registers(to_integer(unsigned(regBSel)));
    -- Caso contrário, a saída B será o valor do registrador selecionado
  END IF;
  -- Write and bypass
  -- Escrita e bypass
```

```
  if writeEnable = '1' then
    registers(to_integer(unsigned(writeRegSel))) <= input;
    -- Escreve o valor da entrada no registrador selecionado
    if regASel = writeRegSel then  -- Bypass para leitura A
      outA <= input;
    end if;
    if regBSel = writeRegSel then
    -- Bypass para leitura B
      outB <= input;
    end if;
  end if;
end if;
```

O processo principal (`regFile`) realiza as operações de leitura e escrita no banco de registradores. Na borda de subida do sinal de clock, a leitura dos registradores A e B é realizada, e na borda de descida, a escrita é realizada se o sinal `writeEnable` estiver ativado. Além disso, esse processo também inclui um mecanismo de "bypass"para evitar conflitos de leitura e escrita simultâneas.

4.1.8 IFID (Instruction Fetch/Decode)

Essa entidade representa o estágio do pipeline responsável pela busca e decodificação da instrução. O pipeline é dividido em estágios para permitir que várias instruções sejam processadas simultaneamente. As principais portas são:

- **Saídas:**
  - `pcOut`: Saída do PC (Contador de Programa).
  - `pcPl4Out`: Saída do valor do PC + 4 (próxima instrução no pipeline).
  - `instOut`: Saída da instrução.
- **Entradas:**
  - `clk`: Sinal de clock.
  - `pcIn`: Entrada do PC.
  - `pcPl4In`: Entrada do valor do PC + 4 (próxima instrução no pipeline).
  - `instIn`: Entrada da instrução.

```
  IF (rising_edge(clk)) THEN
  -- Detecta a borda de subida do sinal de clock
                      if (writeEnableL = '0') then
 -- Verifica se o sinal de habilitação de escrita está desabilitado
                              IDIF(31 DOWNTO 0) <= instIn;
  -- Se estiver desabilitado, armazena a instrução atual no IFID
                              IDIF(63 DOWNTO 32) <= pcIn;
  -- Armazena o valor atual do PC
                              IDIF(95 downto 64) <= pcPl4In;
  -- Armazena o valor do PC mais 4 (próxima instrução)
                      end if;
              END IF;
              IF (falling_edge(clk)) THEN
-- Detecta a borda de descida do sinal de clock
                      pcOut <= IDIF(63 DOWNTO 32);
 -- Na borda de descida, coloca na saída o valor do PC atual
                      instOut <= IDIF(31 DOWNTO 0);
 -- Coloca na saída a instrução armazenada no IFID
                      pcPl4Out <= IDIF(95 downto 64);
```

```
    -- Coloca na saída o valor do PC mais 4
                END IF;
```

A função dessa entidade é armazenar a instrução e os valores do PC para uso nos estágios subsequentes do pipeline.

4.1.9 IDEX (Instruction Decode/Execute)

Essa entidade representa o estágio do pipeline responsável pela decodificação da instrução e preparação para a execução. As principais portas são:

- **Saídas:**
  - `EXin`: Saída do sinal de extensão do sinal de escrita.
- **Entradas:**
  - `clk`: Sinal de clock.
  - `pcIn`: Entrada do PC (Contador de Programa).
  - `read1In`: Entrada do valor lido do registrador 1.
  - `read2In`: Entrada do valor lido do registrador 2.
  - `immGenIn`: Entrada da unidade de geração de imediatos.
  - `aluControlin`: Entrada do controle da ALU (Unidade Lógica Aritmética).
  - `wbAddIn`: Entrada do endereço de escrita no banco de registradores.
  - `WBin`: Entrada do sinal de escrita no banco de registradores.
  - `Min`: Entrada do sinal de escrita no banco de memória.
  - `pcPl4In`: Entrada do PC + 4 (próxima instrução no pipeline).

```
IF (rising_edge(clk)) THEN
    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(149 downto 146) <= EXin;
    idex_s(181 downto 150) <= pcPl4In;
    idex_s(186 downto 182) <= rs1In;
    idex_s(191 downto 187) <= rs2In;
END IF;
IF (falling_edge(clk)) THEN
    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(149 downto 146);
    pcPl4Out <= idex_s(181 downto 150);
END IF;
```

A função dessa entidade é preparar os dados para serem utilizados no próximo estágio, incluindo a seleção dos registradores de leitura e a geração de imediatos para operações aritméticas.

4.1.10 EXMEM (Execute/Memory)

Essa entidade representa o estágio do pipeline responsável pela execução das instruções e acesso à memória (se necessário). As principais portas são:

- **Saídas:**
  - `pcPl4In`: Saída do PC + 4 (próxima instrução no pipeline).
- **Entradas:**
  - `clk`: Sinal de clock.
  - `sumIn`: Entrada do resultado da ALU.
  - `zeroIn`: Entrada do sinal de zero da ALU.
  - `aluIn`: Entrada do valor da ALU.
  - `read2In`: Entrada do valor lido do registrador 2.
  - `wbAddIn`: Entrada do endereço de escrita no banco de registradores.
  - `WBin`: Entrada do sinal de escrita no banco de registradores.
  - `Min`: Entrada do sinal de escrita no banco de memória.

```
IF (rising_edge(clk)) THEN
    exmem_s(31 DOWNTO 0)   <= sumIn;
    -- Armazena o valor do resultado da ALU no registrador exmem_s
    exmem_s(32)            <= zeroIn;
    -- Armazena o valor do sinal de zero da ALU no registrador exmem_s
    exmem_s(64 DOWNTO 33)  <= aluIn;
    -- Armazena o valor da ALU no registrador exmem_s
    exmem_s(96 DOWNTO 65)  <= read2In;
    -- Armazena o valor lido do registrador 2 no registrador exmem_s
    exmem_s(101 DOWNTO 97) <= wbAddIn;
    exmem_s(104 DOWNTO 102) <= WBin;
    exmem_s(110 DOWNTO 105) <= Min;
    -- Armazena o valor do sinal de escrita no banco de memória no registrador exmem_s
    exmem_s(142 downto 111) <= pcPl4In;
    -- Armazena o valor do PC + 4 de entrada no registrador exmem_s
END IF;
```

A função dessa entidade é realizar operações aritméticas na ALU, verificar se o resultado é zero e preparar os dados para escrita no banco de registradores ou na memória (dependendo da instrução executada).

4.1.11 MEMWB (Memory/Write Back)

Essa entidade representa o estágio do pipeline responsável pela busca na memória e escrita de dados no banco de registradores. As principais portas são:

- **Entradas:**
  - `clk`: Sinal de clock.
  - `readIn`: Entrada do valor lido da memória.
  - `aluIn`: Entrada do valor da ALU.
  - `wbAddIn`: Entrada do endereço de escrita no banco de registradores.
  - `WBin`: Entrada do sinal de escrita no banco de registradores.
  - `pcPl4In`: Entrada do PC + 4 (próxima instrução no pipeline).
  - `pcPlIIn`: Entrada do PC + I (próxima instrução no pipeline).

A função dessa entidade é buscar os dados na memória, se necessário, e escrever os resultados no banco de registradores para que estejam disponíveis para a próxima instrução.

```
IF (rising_edge(clk)) THEN
    memwb_s(31 DOWNTO 0)   <= readIn;
    -- Armazena o valor lido da memória no registrador memwb_s
    memwb_s(63 DOWNTO 32)  <= aluIn;
    -- Armazena o valor da ALU no registrador memwb_s
    memwb_s(68 DOWNTO 64)  <= wbAddIn;
    -- Armazena o endereço de escrita no banco de registradores no memwb_s
    memwb_s(71 DOWNTO 69)  <= WBin;
    -- Armazena o sinal de escrita no banco de registradores no memwb_s
    memwb_s(103 downto 72) <= pcPl4In;
    -- Armazena o valor do PC + 4 vindo do estágio MEM no memwb_s
    memwb_s(135 downto 104) <= pcPlIIn;
    -- Armazena o valor do PC + I vindo do estágio MEM no memwb_s

END IF;
IF (falling_edge(clk)) THEN
    readOut    <=  memwb_s(31 DOWNTO 0);
    -- Saída do valor lido da memória armazenado no registrador memwb_s
    aluOut  <=  memwb_s(63 DOWNTO 32);
    -- Saída do valor da ALU armazenado no registrador memwb_s
    wbAddOut <= memwb_s(68 DOWNTO 64);
    -- Saída do endereço de escrita no banco de registradores armazenado no memwb_s
    WBout <= memwb_s(71 DOWNTO 69);
    -- Saída do sinal de escrita no banco de registradores armazenado no memwb_s
    pcPl4Out <= memwb_s(103 downto 72);
    -- Saída do valor do PC + 4 armazenado no registrador memwb_s
    pcPlIOut <= memwb_s(135 downto 104);
    -- Saída do valor do PC + I armazenado no registrador memwb_s
END IF;
```

Essas entidades formam os principais estágios de um pipeline de processador e permitem o processamento de múltiplas instruções em paralelo para melhorar o desempenho do processador. Cada estágio é responsável por tarefas específicas, e os dados fluem sequencialmente através dos estágios durante a execução das instruções.

4.2 Unidades de detecção de Hazard e Fowarding

Para resolver os problemas causados pelos hazards em pipelines simples, as CPUs modernas implementam mecanismos de detecção e resolução de hazards, como o forwarding (encaminhamento) e a bolha (stall).

- **Forwarding (Encaminhamento):** Esse mecanismo permite que o resultado de uma instrução ainda não concluída seja enviado diretamente para o próximo estágio, evitando que a instrução dependente precise esperar pelo resultado. Dessa forma, os dados são encaminhados (forwarded) para a instrução que os necessita, evitando os atrasos causados por hazards de dados.
- **Bolha (Stall):** Quando um hazard de controle é detectado, a CPU pode inserir bolhas (ou stalls) no pipeline, pausando temporariamente o avanço das instruções. Isso permite que a instrução condicional seja resolvida antes que instruções incorretas sejam executadas. Durante esse período de pausa, nenhuma instrução é processada (bolha de instrução).

Esses mecanismos são essenciais para melhorar o desempenho e garantir a correta execução das instruções em CPUs modernas com pipelines mais sofisticados. Eles permitem a execução paralela e o tratamento adequado de dependências entre as instruções, resultando em um aumento significativo no desempenho geral do processador.

4.2.1 Hazard

Explicação do Código "Harzard":

```
process(clk)
begin
    if ((MemReadEX = '1') and ((RDEX = RSrc1) or (RDEX = RSrc2))) then
        stall_interno <='1';
        -- Se houver uma leitura na memória (MemReadEX = '1')
        -- e o resultado da EX estiver em RSrc1 ou RSrc2, aplica um stall
    end if;
end process;
stall <= stall_interno;
```

Esse código descreve um módulo denominado "Harzard", que é responsável por detectar a ocorrência de hazards em um pipeline de processador e gerar um sinal de "stall"(atraso) quando necessário. O "stall"é um mecanismo que pausa temporariamente o avanço das instruções no pipeline para evitar problemas de dependências entre as instruções. O "Harzard"verifica conflitos de leitura (load-use hazards) que ocorrem quando uma instrução depende do resultado de outra que ainda não concluiu o estágio de escrita no banco de registradores.

Portas de Entrada:

- **Portas de Entrada:**
  - `clk`: Entrada do sinal de clock usado para sincronização das operações.
  - `MemReadEX`: Sinal que indica se há leitura na memória no estágio EX (Execute) do pipeline.
  - `RDEX, RSrc1, RSrc2`: Registradores de destino e fontes da instrução provenientes do estágio EX do pipeline.
- **Portas de Saída:**
  - `stall`: Saída que indica se um "stall"deve ser aplicado (sinal '1') ou não (sinal '0').

Funcionamento: O módulo "Harzard"é implementado como uma combinação de processos síncronos e sinais internos. Ele verifica se há um conflito de leitura (load-use hazard) comparando o valor dos registradores de destino RDEX (que representa o resultado da instrução atual no estágio EX) com os registradores de fonte RSrc1 e RSrc2 (que são as fontes da instrução atual no estágio EX). Se houver uma leitura na memória (MemReadEX = '1') e o resultado da instrução atual estiver em uma das fontes, significa que a instrução depende de um resultado que ainda não foi escrito no banco de registradores.

Caso o hazard seja detectado, o sinal interno stall_interno é atribuído com o valor '1', indicando que um "stall"deve ser aplicado. Caso contrário, o stall_interno recebe o valor '0'.

O processo é sensível à borda de subida do sinal de clock (clk). Na borda de subida, o stall_interno é atualizado de acordo com a detecção de hazard. Em seguida, o valor de stall_interno é atribuído à saída stall, que indica se um "stall"deve ser aplicado ou não.

4.2.2 Foward

Explicação do Código "Forward":

```
architecture TypeArchitecture of Forward is
signal forwardA_interno, forwardB_interno : std_logic_vector(1 downto 0) := "00";

begin
process(clk)
begin
    if ((regWriteMEM = '1') and (not(RDMEM = "00000")) and (RDMEM = RSrc1)) then
```

```
        forwardA_interno <= "10";
        -- Encaminha o valor do registrador de MEM/WB para a fonte A
    end if;

    if ((regWriteMEM = '1') and (not(RDMEM = "00000")) and (RDMEM = RSrc2)) then
        forwardB_interno <= "10";
        -- Encaminha o valor do registrador de MEM/WB para a fonte B
    end if;

    if ((regWriteWB = '1') and (not(RDWB = "00000")) and (RDWB = RSrc1)) then
        forwardA_interno <= "01";
        -- Encaminha o valor do registrador de WB para a fonte A
    end if;

    if ((regWriteWB = '1') and (not(RDWB = "00000")) and (RDWB = RSrc2)) then
        forwardB_interno <= "01";
        -- Encaminha o valor do registrador de WB para a fonte B
    end if;

end process;

forwardA <= forwardA_interno;
-- Atribui o valor do sinal interno forwardA_interno à saída forwardA
forwardB <= forwardB_interno;
-- Atribui o valor do sinal interno forwardB_interno à saída forwardB
end TypeArchitecture;
```

Esse código descreve um módulo denominado "Forward", que é responsável por implementar o mecanismo de "forwarding"(encaminhamento) em um pipeline de processador. O "forwarding"permite que os resultados de instruções ainda não concluídas sejam encaminhados diretamente para as instruções dependentes, evitando a necessidade de esperar pelos resultados e melhorando o desempenho do pipeline.

- **Portas de Entrada:**
  - `clk`: Entrada do sinal de clock usado para sincronização das operações.
  - `regWriteWB, regWriteMEM`: Sinais de escrita no registrador provenientes dos estágios MEM/WB (Memory/Write Back) e WB (Write Back) do pipeline.
  - `RSrc1, RSrc2`: Registradores de fonte da instrução provenientes do estágio ID/EX (Instruction Decode/Execute) do pipeline.
  - `RDWB, RDMEM`: Registradores de destino provenientes dos estágios MEM/WB e EX/MEM (Execute/Memory) do pipeline.
- **Portas de Saída:**
  - `forwardA, forwardB`: Saídas que indicam os encaminhamentos para as fontes A e B, respectivamente.

Funcionamento: O módulo "Forward"é implementado como uma combinação de processos síncronos e sinais internos. Ele verifica se há instruções no pipeline que possam fornecer os valores necessários para as instruções atuais, mesmo antes de terem sido escritas nos registradores.

O "forwarding"ocorre em duas fases principais:

Forwarding de MEM/WB para ID/EX: Nesta fase, o módulo verifica se alguma instrução no estágio MEM/WB (Memory/Write Back) escreveu no registrador (regWriteMEM = '1') e, se o fez, verifica se o registrador de destino (RDMEM) é igual a uma das fontes da instrução atual (RSrc1 ou RSrc2). Se essas condições forem atendidas, o módulo define os sinais internos forwardA_interno e

forwardB_interno para '10', indicando que o valor do registrador MEM/WB deve ser encaminhado para a fonte A ou B da instrução atual.

Forwarding de WB para ID/EX: Nesta fase, o módulo realiza o mesmo processo, mas verificando as informações do estágio WB (Write Back). Se uma instrução no estágio WB escreveu no registrador (regWriteWB = '1') e o registrador de destino (RDWB) é igual a uma das fontes da instrução atual (RSrc1 ou RSrc2), então o valor do registrador WB deve ser encaminhado para a fonte A ou B da instrução atual. Nesse caso, o módulo define os sinais internos forwardA_interno e forwardB_interno para '01'.

O processo é sensível à borda de subida do sinal de clock (clk). Na borda de subida, os sinais internos forwardA_interno e forwardB_interno são atualizados de acordo com as detecções de "forwarding". Em seguida, esses sinais internos são atribuídos às saídas forwardA e forwardB, que indicam os encaminhamentos para as fontes A e B da instrução atual no estágio ID/EX do pipeline.

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.