

Universidade: Universidade do Vale do Itajaí (Univali)

Curso: Ciência da Computação

Disciplina: Arquitetura e Organização de Processadores

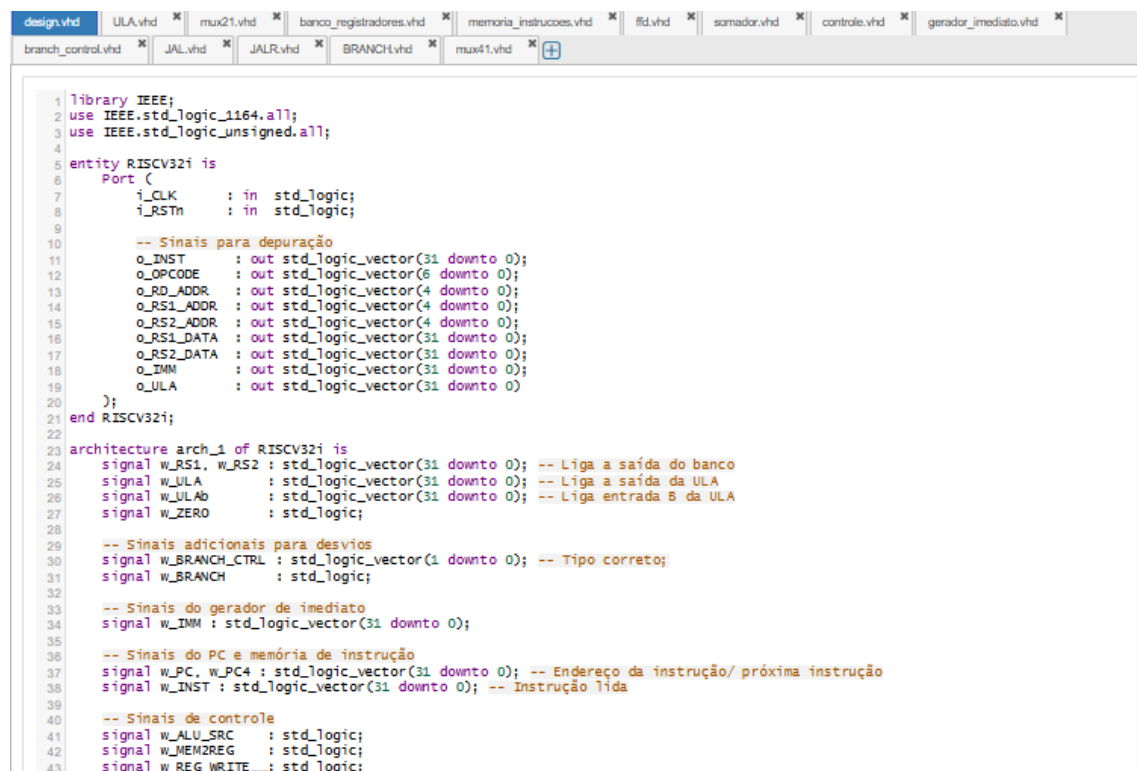
Professor: Thiago Felski Pereira

Título do Trabalho: Organização do RISC-V32i Monociclo no EDAPlayground

Nome das Alunas: Emily Mendes dos Santos e Marcela Furtado Leffer

Relatório Processador RISC-V32i

Desing



```
1 library IEEE;
2 use IEEE.std_logic_1164.all;
3 use IEEE.std_logic_unsigned.all;
4
5 entity RISC_V32i is
6   Port (
7     i_CLK      : in  std_logic;
8     i_RSTn     : in  std_logic;
9
10    -- Sinais para depuração
11    o_INST      : out std_logic_vector(31 downto 0);
12    o_OPCODE    : out std_logic_vector(6  downto 0);
13    o_RD_ADDR   : out std_logic_vector(4  downto 0);
14    o_RS1_ADDR  : out std_logic_vector(4  downto 0);
15    o_RS2_ADDR  : out std_logic_vector(4  downto 0);
16    o_RS1_DATA  : out std_logic_vector(31 downto 0);
17    o_RS2_DATA  : out std_logic_vector(31 downto 0);
18    o_IMM_DATA  : out std_logic_vector(31 downto 0);
19    o_ULA       : out std_logic_vector(31 downto 0);
20  );
21 end RISC_V32i;
22
23 architecture arch_1 of RISC_V32i is
24   signal w_RS1, w_RS2 : std_logic_vector(31 downto 0); -- Liga a saída do banco
25   signal w_ULA        : std_logic_vector(31 downto 0); -- Liga a saída da ULA
26   signal w_ULAb       : std_logic_vector(31 downto 0); -- Liga entrada B da ULA
27   signal w_ZERO       : std_logic;
28
29   -- Sinais adicionais para desvios
30   signal w_BRANCH_CTRL : std_logic_vector(1 downto 0); -- Tipo correto;
31   signal w_BRANCH      : std_logic;
32
33   -- Sinais do gerador de imediato
34   signal w_IMM : std_logic_vector(31 downto 0);
35
36   -- Sinais do PC e memória de instrução
37   signal w_PC, w_PC4 : std_logic_vector(31 downto 0); -- Endereço da instrução/ próxima instrução
38   signal w_INST : std_logic_vector(31 downto 0); -- Instrução lida
39
40   -- Sinais de controle
41   signal w_ALU_SRC : std_logic;
42   signal w_MEM2REG : std_logic;
43   signal w_REG_WRITE : std_logic;
```

Componentes principais:

1. Componente de controle de desvios:

- O módulo `branch_control` determina se a instrução atual deve sofrer um desvio (branch) com base no sinal `w_ZERO`, que indica se a operação anterior (geralmente uma comparação) resultou em zero.

2. Somas de PC:

- O módulo somador é utilizado para calcular os endereços de desvio para diferentes tipos de instruções (branch, JAL, JALR), somando o valor do PC com o valor do imediato (`w_IMM`).

3. Multiplexadores:

- O multiplexador `mux41` seleciona o próximo valor para o PC com base no controle do tipo de desvio (`w_BRANCH_CTRL`). Isso pode ser:
 - O valor do PC somado com o imediato para um branch (`w_PC_NEXT_BRANCH`),
 - O valor do PC somado com o imediato para JAL (`w_PC_NEXT_JAL`),
 - O valor de `RS1` somado com o imediato para JALR (`w_PC_NEXT_JALR`).

4. Componente de controle principal:

- O módulo controle gera os sinais de controle para a ULA (ALU), como `ALU_SRC` (seleção da fonte para a ULA), `MEM2REG` (controle de onde os dados a serem escritos nos registradores vêm: da ULA ou da memória), `REG_WRITE` (controle de escrita no banco de registradores), `MEM_READ` e `MEM_WRITE` (controle de acesso à memória), e `ALUOP` (código da operação da ULA).

5. Banco de registradores:

- O banco de registradores (`banco_registradores`) armazena e fornece os valores dos registradores. Ele também permite que a ULA escreva um valor de volta aos registradores.

6. **ULA (Unidade Lógica e Aritmética):**

- A ULA executa operações aritméticas e lógicas. Ela recebe os valores de RS1 e RS2 (ou o imediato w_IMM, dependendo de ALU_SRC) e executa a operação indicada por ALUOP.

7. **Gerador de imediato:**

- O gerador_imediato é responsável por extrair o valor do imediato da instrução (localizada em w_INST) e passá-lo para outros componentes, como a ULA.

8. **Memória de instruções:**

- A memoria_instrucoes armazena e fornece a instrução baseada no valor do PC.

9. **Sinalização para depuração:**

- Vários sinais, como o_INST, o_OPCODE, o_RD_ADDR, entre outros, são gerados para fins de depuração e observação do funcionamento do processador, ajudando na verificação do comportamento da máquina durante a simulação ou execução do testbench.

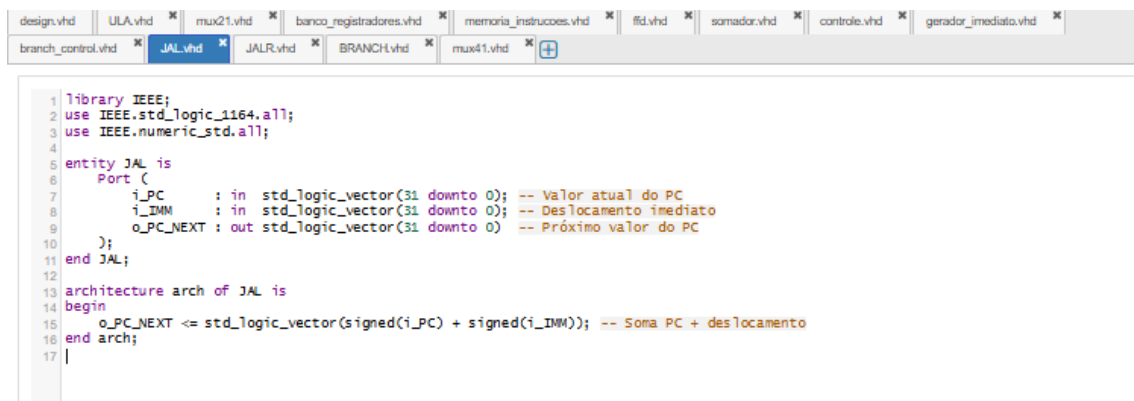
Fluxo de execução:

- O processador começa com o valor do PC e a instrução correspondente é lida da memória de instruções.
- Dependendo da instrução e dos sinais de controle, o processador decide o próximo valor de PC (se é um desvio, JAL ou JALR) e executa a operação correspondente usando a ULA.
- O valor da ULA é escrito no banco de registradores ou na memória, dependendo da instrução.
- O ciclo continua, movendo-se para a próxima instrução ou saltando, caso uma instrução de desvio tenha sido executada.

Resumo dos principais sinais:

- **w_PC**: Armazena o endereço da instrução atual.
- **w_IMM**: Imediato extraído da instrução.
- **w_ALU_SRC**: Controle para escolher entre o valor de RS2 ou o imediato como entrada para a ULA.
- **w_MEM2REG**: Controle que define se o valor a ser escrito no registrador vem da ULA ou da memória.
- **w_BRANCH_CTRL**: Controle que determina o tipo de desvio (branch, JAL, JALR).
- **w_REG_WRITE**: Permite ou não a escrita nos registradores.
- **w_PC_NEXT**: Calcula os próximos valores de PC para os desvios (branch, JAL, JALR).

Entidade JAL



```
1 library IEEE;
2 use IEEE.std_logic_1164.all;
3 use IEEE.numeric_std.all;
4
5 entity JAL is
6   Port (
7     i_PC      : in  std_logic_vector(31 downto 0); -- Valor atual do PC
8     i_IMM     : in  std_logic_vector(31 downto 0); -- Deslocamento imediato
9     o_PC_NEXT : out std_logic_vector(31 downto 0) -- Próximo valor do PC
10  );
11 end JAL;
12
13 architecture arch of JAL is
14 begin
15   o_PC_NEXT <= std_logic_vector(signed(i_PC) + signed(i_IMM)); -- Soma PC + deslocamento
16 end arch;
```

A entidade JAL tem três portas:

- **Entradas:**
 - **i_PC**: Recebe o valor atual do PC, que é um vetor de 32 bits representando o endereço da instrução que está sendo executada.
 - **i_IMM**: Recebe um valor imediato de 32 bits, que é o deslocamento que será somado ao valor atual do PC. Esse deslocamento é calculado e fornecido pela unidade de controle ou por alguma lógica da arquitetura.

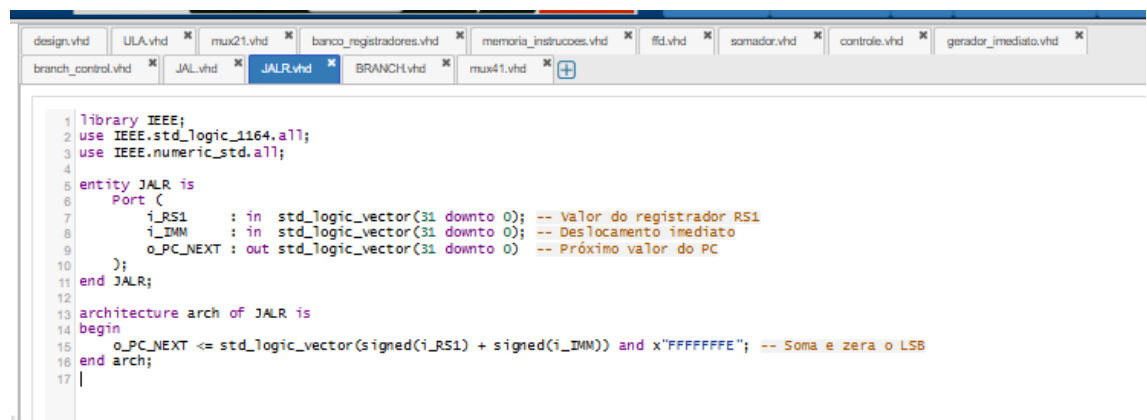
- **Saída:**
 - o_PC_NEXT: É o próximo valor do PC, após a soma do valor atual do PC com o deslocamento imediato. Esse valor será o novo endereço da próxima instrução a ser executada após o desvio (jump).

Arquitetura arch

Dentro da arquitetura arch, o cálculo do próximo valor do PC é feito da seguinte forma:

- A soma do valor de i_PC e i_IMM é realizada. Como i_PC e i_IMM são do tipo std_logic_vector (vetores de bits), é necessário convertê-los para o tipo signed (número com sinal) usando a função signed(). Após a conversão, a soma é feita e o resultado é novamente convertido para std_logic_vector para que a saída o_PC_NEXT possa ser fornecida na forma de um vetor de 32 bits.

Entidade JALR



```

1 library IEEE;
2 use IEEE.std_logic_1164.all;
3 use IEEE.numeric_std.all;
4
5 entity JALR is
6     Port (
7         i_RS1      : in std_logic_vector(31 downto 0); -- Valor do registrador RS1
8         i_IMM       : in std_logic_vector(31 downto 0); -- Deslocamento imediato
9         o_PC_NEXT   : out std_logic_vector(31 downto 0) -- Próximo valor do PC
10    );
11 end JALR;
12
13 architecture arch of JALR is
14 begin
15     o_PC_NEXT <= std_logic_vector(signed(i_RS1) + signed(i_IMM)) and x"FFFFFFFE"; -- Soma e zera o LSB
16 end arch;
17

```

A entidade JALR tem três portas:

- **Entradas:**
 - i_RS1: Recebe o valor do registrador RS1, que contém um endereço de memória base.
 - i_IMM: Recebe o deslocamento imediato de 32 bits, que será somado ao valor de RS1.

- **Saída:**
 - `o_PC_NEXT`: É o próximo valor do Program Counter (PC), que será calculado pela soma de `i_RS1` com `i_IMM`.

Arquitetura arch

Dentro da arquitetura arch, o cálculo do próximo valor do PC é feito da seguinte forma:

- A soma de `i_RS1` e `i_IMM` é realizada. Como `i_RS1` e `i_IMM` são do tipo `std_logic_vector` (vetores de bits), eles são convertidos para o tipo `signed` (número com sinal) usando a função `signed()`. Após a soma, o resultado é convertido novamente para o tipo `std_logic_vector`.
- **Ajuste do bit menos significativo (LSB):**
 - O valor resultante da soma de `i_RS1` e `i_IMM` é então "mascarado" com o valor `x"FFFFFFFE"`, que é o valor hexadecimal para um número onde todos os bits são 1, exceto o bit menos significativo (LSB), que é 0. Isso garante que o próximo valor do PC tenha o LSB zerado.
 - O operador `and` é utilizado para garantir que o bit LSB do valor de `o_PC_NEXT` seja 0, independentemente do valor da soma.

Explicação do comportamento

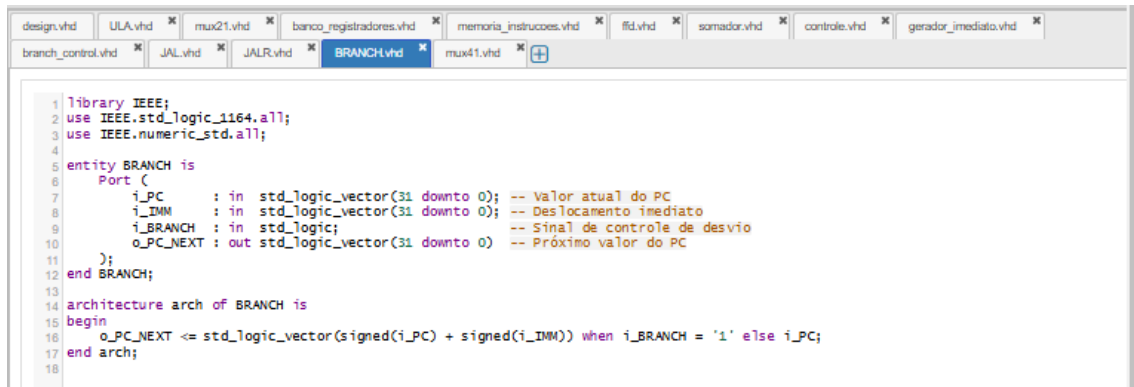
1. **Somando RS1 e IMM:** A operação de soma é realizada entre o valor do registrador RS1 e o deslocamento imediato IMM, que juntos definem o endereço do próximo PC.
2. **Zerando o LSB:** Para instruções JALR, é necessário garantir que o LSB do endereço de destino seja sempre 0, pois as instruções JALR requerem que o PC seja alinhado em múltiplos de 2. Portanto, a operação `and x"FFFFFFFE"` zera o LSB, fazendo com que o valor de `o_PC_NEXT` seja um endereço válido para o PC.

Exemplo de como funciona:

- Se `i_RS1` for `x"00000004"` (um valor hipotético) e `i_IMM` for `x"00000008"`, a soma resultará em `x"0000000C"`. A operação `and x"FFFFFFFE"` garantirá que o

bit menos significativo seja 0, e o resultado final de o_PC_NEXT será x"0000000C" (caso o valor não seja múltiplo de 2, ele será ajustado).

Entidade BRANCH



```
1 library IEEE;
2 use IEEE.std_logic_1164.all;
3 use IEEE.numeric_std.all;
4
5 entity BRANCH is
6     Port (
7         i_PC      : in  std_logic_vector(31 downto 0); -- Valor atual do PC
8         i_IMM     : in  std_logic_vector(31 downto 0); -- Deslocamento imediato
9         i_BRANCH  : in  std_logic;                    -- Sinal de controle de desvio
10        o_PC_NEXT : out std_logic_vector(31 downto 0); -- Próximo valor do PC
11    );
12 end BRANCH;
13
14 architecture arch of BRANCH is
15 begin
16     o_PC_NEXT <= std_logic_vector(signed(i_PC) + signed(i_IMM)) when i_BRANCH = '1' else i_PC;
17 end arch;
```

A entidade BRANCH tem quatro portas:

- **Entradas:**
 - i_PC: O valor atual do Program Counter (PC), que é um endereço de memória indicando a próxima instrução a ser executada.
 - i_IMM: O deslocamento imediato de 32 bits, que será somado ao valor de i_PC se o desvio for ativado.
 - i_BRANCH: O sinal de controle de desvio (branch). Ele determina se o desvio ocorrerá ou se o PC permanecerá inalterado.
- **Saída:**
 - o_PC_NEXT: O próximo valor do PC, que pode ser o valor atual (i_PC) ou o valor resultante da soma do PC com o deslocamento imediato (i_IMM), dependendo do valor de i_BRANCH.

Arquitetura arch

Na arquitetura arch, o cálculo do próximo valor do PC é feito da seguinte maneira:

- A expressão $o_PC_NEXT \leq std_logic_vector(signed(i_PC) + signed(i_IMM))$ when $i_BRANCH = '1'$ else i_PC ; é uma estrutura condicional que verifica o valor de i_BRANCH . O comportamento é o seguinte:
 - **Se i_BRANCH for igual a '1':** O próximo valor do PC será o resultado da soma entre i_PC e i_IMM . Ambas as entradas são convertidas de std_logic_vector para $signed$ para realizar a soma aritmética. O resultado é então convertido de volta para std_logic_vector para ser atribuído a o_PC_NEXT .
 - **Se i_BRANCH for igual a '0':** O próximo valor do PC será o valor atual de i_PC . Ou seja, não há desvio e o PC mantém seu valor atual.

Explicação do comportamento

1. Verificação do sinal de controle i_BRANCH :

- Se o sinal i_BRANCH for '1', a instrução de branch será executada, e o PC será atualizado com o valor de $i_PC + i_IMM$, ou seja, o próximo endereço será calculado com base no deslocamento.
- Se i_BRANCH for '0', o PC não será alterado, mantendo o fluxo normal de execução.

2. Cálculo do próximo PC:

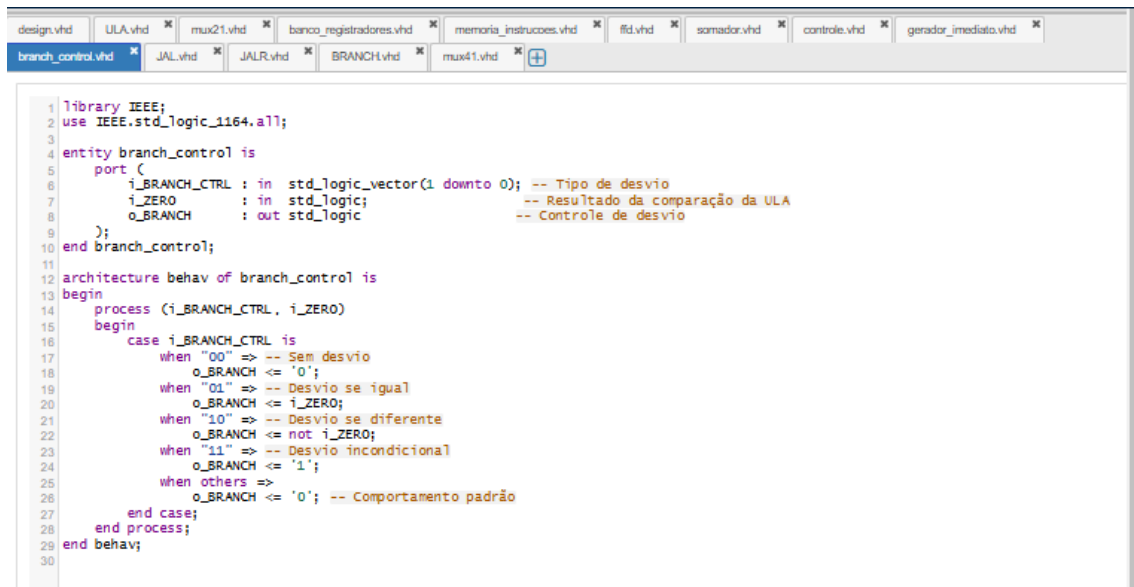
- A soma de i_PC e i_IMM é realizada se o desvio for ativo ($i_BRANCH = '1'$). Isso ajusta o PC para o endereço de destino do desvio.
- A operação de soma usa os tipos $signed$ para garantir que a operação leve em conta o sinal do deslocamento imediato (caso o deslocamento seja negativo).

Exemplo de funcionamento:

- Se o valor atual do PC (i_PC) for $x"00000004"$, e o deslocamento imediato (i_IMM) for $x"00000008"$, o valor do próximo PC seria $x"0000000C"$, se o sinal i_BRANCH for '1'.

- Se o sinal i_BRANCH for '0', o próximo PC será igual ao valor de i_PC, ou seja, x"00000004".

Entidade branch_control



```

1 library IEEE;
2 use IEEE.std_logic_1164.all;
3
4 entity branch_control is
5     port (
6         i_BRANCH_CTRL : in  std_logic_vector(1 downto 0); -- Tipo de desvio
7         i_ZERO         : in  std_logic;                    -- Resultado da comparação da ULA
8         o_BRANCH       : out std_logic;                    -- Controle de desvio
9     );
10 end branch_control;
11
12 architecture behav of branch_control is
13 begin
14     process (i_BRANCH_CTRL, i_ZERO)
15     begin
16         case i_BRANCH_CTRL is
17             when "00" => -- Sem desvio
18                 o_BRANCH <= '0';
19             when "01" => -- Desvio se igual
20                 o_BRANCH <= i_ZERO;
21             when "10" => -- Desvio se diferente
22                 o_BRANCH <= not i_ZERO;
23             when "11" => -- Desvio incondicional
24                 o_BRANCH <= '1';
25             when others =>
26                 o_BRANCH <= '0'; -- Comportamento padrão
27         end case;
28     end process;
29 end behav;
30

```

A entidade branch_control tem as seguintes portas:

- **Entradas:**
 - i_BRANCH_CTRL: Um vetor de 2 bits que especifica o tipo de desvio.
Cada valor de i_BRANCH_CTRL indica uma condição de desvio diferente:
 - "00": Sem desvio.
 - "01": Desvio se igual (quando o resultado da comparação da ULA é zero).
 - "10": Desvio se diferente (quando o resultado da comparação da ULA não é zero).
 - "11": Desvio incondicional (sempre desvia).
 - i_ZERO: Um sinal de 1 bit que representa o resultado da comparação da ULA. Se i_ZERO for '1', isso indica que a comparação resultou em zero, e se for '0', a comparação não resultou em zero.

- **Saída:**
 - o_BRANCH: O sinal de controle de desvio. Ele determina se o desvio deve ocorrer. Se o_BRANCH for '1', o desvio ocorre, e se for '0', não ocorre.

Arquitetura behav

A arquitetura behav descreve o comportamento do controlador de desvio. Ela utiliza um processo sensível às entradas i_BRANCH_CTRL e i_ZERO, que determina o valor de o_BRANCH com base nos valores dessas entradas.

A estrutura condicional case analisa o valor de i_BRANCH_CTRL e define o valor de o_BRANCH com base no seguinte:

- **Caso i_BRANCH_CTRL = "00" (sem desvio):**
 - O desvio não ocorre, então o_BRANCH é atribuído a '0'.
- **Caso i_BRANCH_CTRL = "01" (desvio se igual):**
 - Se o resultado da comparação da ULA (i_ZERO) for '1' (indicando que a comparação foi igual), o desvio ocorre, e o_BRANCH é atribuído a '1'. Caso contrário, o desvio não ocorre, e o_BRANCH é atribuído a '0'.
- **Caso i_BRANCH_CTRL = "10" (desvio se diferente):**
 - Se o resultado da comparação da ULA (i_ZERO) for '0' (indicando que a comparação foi diferente), o desvio ocorre, e o_BRANCH é atribuído a '1'. Caso contrário, o desvio não ocorre, e o_BRANCH é atribuído a '0'.
- **Caso i_BRANCH_CTRL = "11" (desvio incondicional):**
 - O desvio ocorre sempre, então o_BRANCH é atribuído a '1'.
- **Caso others:**
 - Se o valor de i_BRANCH_CTRL for diferente de qualquer um dos casos anteriores, o_BRANCH é atribuído a '0', o que é considerado um comportamento padrão.

Explicação do comportamento

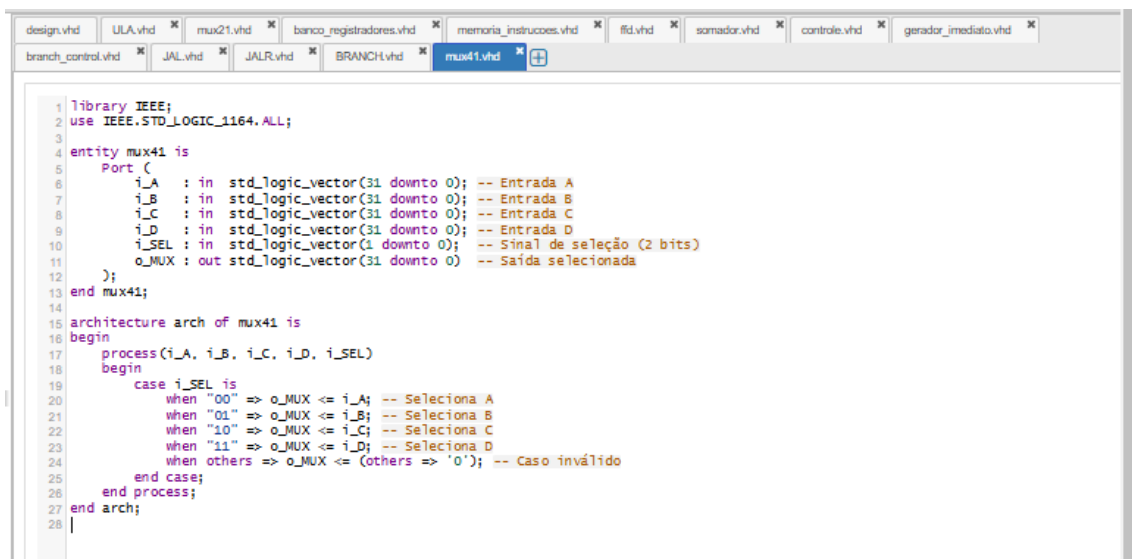
Este bloco de controle de desvio serve para implementar os diferentes tipos de desvios condicional e incondicional, dependendo do valor de `i_BRANCH_CTRL`:

- **Sem desvio (`i_BRANCH_CTRL = "00"`):**
 - `o_BRANCH` será '0', indicando que o desvio não ocorre.
- **Desvio se igual (`i_BRANCH_CTRL = "01"`):**
 - Se o resultado da comparação (`i_ZERO`) for '1', significa que a comparação foi igual, e o desvio ocorrerá (`o_BRANCH = '1'`).
- **Desvio se diferente (`i_BRANCH_CTRL = "10"`):**
 - Se o resultado da comparação (`i_ZERO`) for '0', significa que a comparação foi diferente, e o desvio ocorrerá (`o_BRANCH = '1'`).
- **Desvio incondicional (`i_BRANCH_CTRL = "11"`):**
 - O desvio ocorrerá independentemente do resultado da comparação, ou seja, `o_BRANCH` será sempre '1'.

Exemplo de funcionamento:

- Se `i_BRANCH_CTRL = "01"` e `i_ZERO = '1'`, o desvio ocorrerá (porque a comparação foi igual), e `o_BRANCH = '1'`.
- Se `i_BRANCH_CTRL = "10"` e `i_ZERO = '0'`, o desvio ocorrerá (porque a comparação foi diferente), e `o_BRANCH = '1'`.
- Se `i_BRANCH_CTRL = "00"`, o desvio não ocorrerá, independentemente de `i_ZERO`, e `o_BRANCH = '0'`.

Entidade mux41



```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity mux41 is
5     Port (
6         i_A : in std_logic_vector(31 downto 0); -- Entrada A
7         i_B : in std_logic_vector(31 downto 0); -- Entrada B
8         i_C : in std_logic_vector(31 downto 0); -- Entrada C
9         i_D : in std_logic_vector(31 downto 0); -- Entrada D
10        i_SEL : in std_logic_vector(1 downto 0); -- Sinal de seleção (2 bits)
11        o_MUX : out std_logic_vector(31 downto 0) -- Saída selecionada
12    );
13 end mux41;
14
15 architecture arch of mux41 is
16 begin
17     process(i_A, i_B, i_C, i_D, i_SEL)
18     begin
19         case i_SEL is
20             when "00" => o_MUX <= i_A; -- Seleciona A
21             when "01" => o_MUX <= i_B; -- Seleciona B
22             when "10" => o_MUX <= i_C; -- Seleciona C
23             when "11" => o_MUX <= i_D; -- Seleciona D
24             when others => o_MUX <= (others => '0'); -- Caso inválido
25         end case;
26     end process;
27 end arch;
```

A entidade mux41 define as portas do multiplexador:

- **Entradas:**
 - i_A, i_B, i_C, i_D: São as quatro entradas de 32 bits. Cada uma dessas entradas pode ser selecionada para a saída, dependendo do valor do sinal de seleção (i_SEL).
 - i_SEL: Um vetor de 2 bits usado para selecionar qual das entradas (A, B, C ou D) será direcionada para a saída. O sinal i_SEL pode ter os seguintes valores:
 - "00": Seleciona a entrada A.
 - "01": Seleciona a entrada B.
 - "10": Seleciona a entrada C.
 - "11": Seleciona a entrada D.
- **Saída:**
 - o_MUX: A saída de 32 bits que será igual à entrada selecionada de acordo com o valor de i_SEL.

Arquitetura arch

A arquitetura arch descreve o comportamento do multiplexador. O processo sensível a `i_A`, `i_B`, `i_C`, `i_D` e `i_SEL` verifica o valor de `i_SEL` e seleciona a entrada correspondente para atribuir à saída `o_MUX`.

- A estrutura condicional case verifica o valor de `i_SEL` e seleciona a entrada apropriada:
 - **Caso `i_SEL = "00"`:** A saída `o_MUX` recebe o valor de `i_A` (entrada A).
 - **Caso `i_SEL = "01"`:** A saída `o_MUX` recebe o valor de `i_B` (entrada B).
 - **Caso `i_SEL = "10"`:** A saída `o_MUX` recebe o valor de `i_C` (entrada C).
 - **Caso `i_SEL = "11"`:** A saída `o_MUX` recebe o valor de `i_D` (entrada D).
 - **Caso `others`:** Se o valor de `i_SEL` não corresponder a nenhum dos casos acima (como um valor inválido), a saída `o_MUX` é atribuída a zero (`((others => '0'))`).

Explicação do comportamento

O multiplexador 4x1 seleciona uma das entradas (`i_A`, `i_B`, `i_C`, `i_D`) com base no valor do sinal de seleção (`i_SEL`). Dependendo de qual valor `i_SEL` tem, a saída `o_MUX` receberá uma das entradas.

Exemplo de funcionamento:

- Se `i_SEL = "00"`, então `o_MUX` será igual a `i_A`.
- Se `i_SEL = "01"`, então `o_MUX` será igual a `i_B`.
- Se `i_SEL = "10"`, então `o_MUX` será igual a `i_C`.
- Se `i_SEL = "11"`, então `o_MUX` será igual a `i_D`.
- Se `i_SEL` for um valor diferente dos que são esperados, `o_MUX` será igual a `'0'` (comportamento padrão).

Considerações Finais

Os componentes VHDL descritos nos códigos apresentados representam blocos fundamentais de uma arquitetura de processador, sendo essenciais para a execução de operações de controle e manipulação de dados em sistemas digitais. A seguir, discutimos os principais pontos abordados:

1. **JAL (Jump and Link):** O bloco JAL foi projetado para atualizar o contador de programa (PC) com um valor que é a soma do valor atual do PC e um deslocamento imediato. Ele é fundamental para a implementação de saltos incondicionais, como ocorre em sub-rotinas e funções dentro de um processador. A utilização de operações aritméticas com números binários e a manipulação de deslocamentos imediatos são abordagens essenciais para a execução correta desses saltos.
2. **JALR (Jump and Link Register):** O módulo JALR permite a atualização do PC com base no valor armazenado no registrador RS1 somado ao deslocamento imediato, garantindo a flexibilidade de realizar saltos a partir de endereços registrados. Além disso, a máscara x"FFFFFFFE" é utilizada para garantir que o endereço resultante esteja alinhado a um múltiplo de 4, o que é um requisito comum em arquiteturas de processadores que operam com endereços de palavras alinhadas.
3. **BRANCH:** O módulo de desvio condicional (BRANCH) realiza a atualização do PC com base em um sinal de controle de desvio (i_BRANCH) e no deslocamento imediato. Este módulo é utilizado para implementar saltos condicionais em estruturas de controle, como if e while, em um processador. Quando o desvio é permitido, o PC é ajustado, caso contrário, o PC mantém o valor anterior.
4. **Branch Control:** O controle de desvios condicional é implementado através do módulo branch_control, que interpreta um sinal de controle de 2 bits (i_BRANCH_CTRL) e decide qual condição de desvio aplicar com base no resultado de uma comparação (i_ZERO). Este bloco é crucial para a correta execução de instruções de controle de fluxo, como saltos em caso de igualdade, diferença ou saltos incondicionais.

5. **Mux 4x1:** O multiplexador 4x1 (mux41) implementa uma função de seleção entre quatro entradas com base em um sinal de controle de 2 bits (i_SEL). Ele é fundamental para a escolha dinâmica de dados em sistemas digitais, permitindo que diferentes fontes de dados sejam selecionadas conforme a necessidade do processamento.