UNIVERSIDADE da MADEIRA

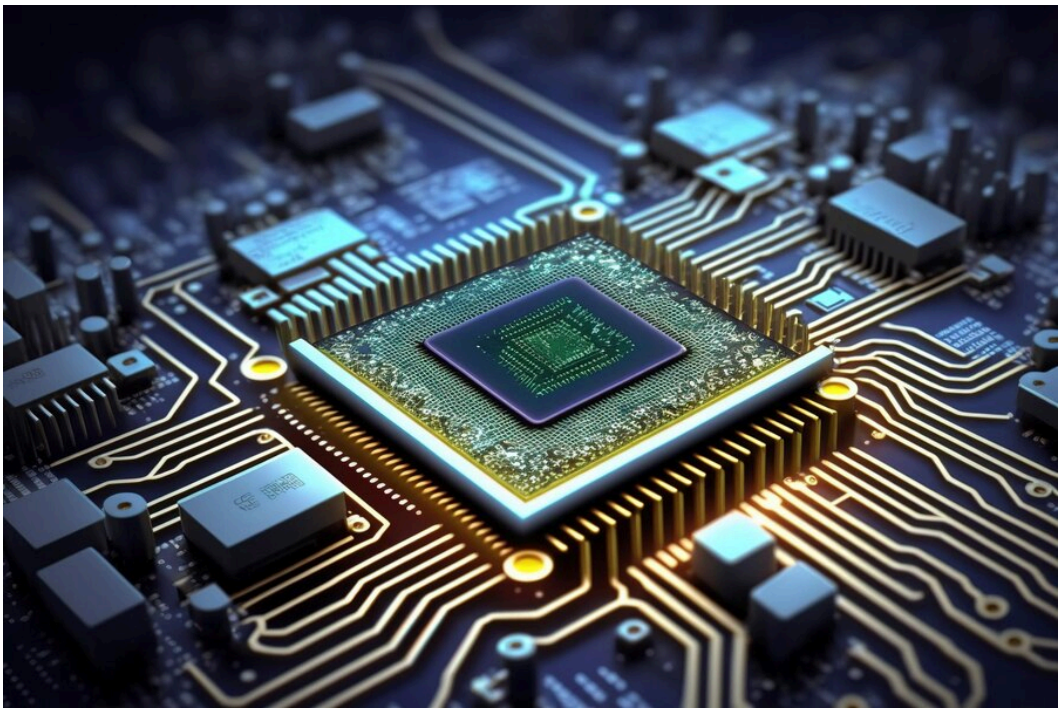
FCEE - LEI

ARQUITETURA DE COMPUTADORES

Sofia Inácio | Dionísio Barros | Pedro Camacho | Dino Vasconcelos

2023/2024

TRABALHO PRÁTICO 1



Trabalho realizado por:

Ana Leonor Freitas - nº2081821

Diogo Miguel Paixão - nº2079921

6 de março de 2024

Índice

Introdução

No primeiro trabalho prático de Arquitetura de computadores, foi-nos proposto a realização de um processador básico, com um conjunto mínimo de instruções,, capaz de manipular diretamente dados de 8 bits, em linguagem de descrição de hardware (VHDL), utilizando o software ISE Design Suite da Xilinx.

O processador é constituído por diversos blocos que, juntamente com a memória de instruções e a memória de dados, dão origem à placa-mãe.

Ao explorar a arquitetura e instruções do processador, este trabalho visa fornecer uma compreensão abrangente da conceção de hardware. A realização bem-sucedida deste projeto não só solidificará conhecimentos em VHDL, mas também proporcionará uma valiosa experiência prática na implementação de processadores em dispositivos FPGA, representando um avanço significativo na compreensão da arquitetura de computadores.

Objetivos

Neste trabalho pretendemos a implementação de uma placa-mãe constituída pelo processador, onde o mesmo é constituído por um conjunto de blocos que teremos de implementar separadamente, uma memória de dados e memória de instruções.

Como principal objetivo, como tal referido na introdução, é pretendido o desenvolvimento do processador básico, capaz de manipular dados de 8 bits, na linguagem VHDL.

Desenvolvimento

Começamos por desenvolver o código de cada bloco do processador e seguidamente efetuamos os respectivos testes.

1. Periférico de Entrada

O Periférico de Entrada facilita a comunicação entre o processador e o ambiente externo, permitindo que o utilizador introduza dados para posterior processamento.

Na nossa implementação, começamos por definir duas entradas, *ESCR_P* (indicando a operação de escrita), *PIN* (representando os dados de entrada) e uma saída *Dados_IN* (indicando os dados de entrada lidos). Utilizando uma arquitetura "Behavioral", implementamos um processo que monitora as mudanças em *ESCR_P* e *PIN*. Quando *ESCR_P* está em '0', atribuímos os dados presentes em *PIN* à saída *Dados_IN*, simulando assim o comportamento de leitura dos dados de entrada pelo periférico.

2. Periférico de Saída

O Periférico de Saída permite que o utilizador visualize os resultados dos programas executados pelo processador.

Na nossa implementação, definimos três entradas, *Operando1* (representando os dados de entrada), *ESCR_P* , clk (sinal de relógio) e uma saída, *POUT* (indicando os dados de saída). Utilizando uma arquitetura "Behavioral", criamos um processo que é sensível às

mudanças nas três entradas. Se *ESCR_P* estiver em '1' e houver uma transição ascendente no sinal de relógio, o valor presente em *Operando1* é atribuído à saída *POUT*. Isso simula o comportamento de escrita dos dados de saída pelo periférico, quando necessário.

3. Multiplexer do Banco de Registos (Mux R)

O Multiplexer do Banco de Registos tem a função de direcionar um dos quatro sinais disponíveis à sua entrada, para a sua saída.

Na nossa implementação, definimos cinco entradas, *SEL_Dados* (indicando a seleção do sinal a ser direcionado), *Constante*, *Dados_M*, *Dados_IN*, *Resultado* (os quatro representando os sinais disponíveis), e *Dados_R* como saída (representando o sinal direcionado).

Utilizando uma arquitetura "Behavioral", implementamos um processo que monitora as mudanças nas cinco entradas. Utilizamos uma estrutura de seleção de casos (case) para determinar qual sinal deve ser direcionado para *Dados_R*, dependendo do valor presente em *SEL_Dados*. Se o valor for "00", direcionamos *Resultado* para *Dados_R*; se for "01", direcionamos *Dados_IN*; se for "10", direcionamos *Dados_M*; e se for "11", direcionamos *Constante*. Para quaisquer outros valores de *SEL_Dados*, atribuímos 'X' a *Dados_R*.

4. Banco de Registos

O Banco de Registos consiste em uma coleção de oito registros (oito variáveis) que permite operações de escrita e leitura desses registros.

Na nossa implementação, definimos quatro entradas: *ESCR_R*, *SEL_R* (indicando a seleção do registro a ser escrito e quais destes são direcionados às saídas), *Dados_R* e *clk* (sinal de relógio). Além disso, as saídas *Operando1* e *Operando2*.

Utilizando uma arquitetura "Behavioral", implementamos um processo que monitora as mudanças nas quatro entradas. Quando ocorre uma transição ascendente do sinal de relógio (*clk*) e o bit menos significativo de *ESCR_R* está em '1', o valor presente no sinal de entrada *Dados_R* é armazenado no registro especificado pelos três bits menos significativos do sinal *SEL_R*. Além disso, o banco de registros está continuamente realizando leituras: a saída *Operando1* apresenta o valor do registro especificado pelos três bits menos significativos do sinal *SEL_R*, enquanto a saída *Operando2* apresenta o valor do registro especificado pelos três bits mais significativos do sinal *SEL_R*.

5. Unidade Lógica e Aritmética (ALU)

A Unidade Lógica e Aritmética (ALU) desempenha um papel crucial em processadores, sendo responsável por realizar uma variedade de operações aritméticas e lógicas.

No âmbito deste projeto, desenvolvemos uma ALU capaz de executar operações de adição, subtração, AND, NAND, OR, NOR, XOR e XNOR, bem como realizar comparações como menor que (<), maior que (>), menor ou igual a (<=), maior ou igual a (>=) e igual a (=).

Na nossa implementação, definimos três entradas: *SEL_ALU* (seleção da operação a ser realizada), *Operando1* (primeiro operando) e *Operando2* (segundo operando). Além

disso, temos duas saídas: *E_FLAG* (sinal de bandeira de estado) e *Resultado* (indica o resultado da operação realizada pela ALU).

Utilizando uma arquitetura "Behavioral", implementamos um processo que monitora as mudanças nas três entradas. Dependendo do valor de *SEL_ALU*, realizamos a operação correspondente entre *Operando1* e *Operando2*. Para as operações de comparação, atualizamos o sinal *E_FLAG* de acordo com o resultado da comparação. Por fim, o resultado final é atribuído à saída *Resultado*.

6. Registo de Flags

O Registo de Flags foi desenvolvido para armazenar o valor do sinal de entrada *E_FLAG* e encaminhar um dos bits guardados para o sinal de saída *S_FLAG*.

Na nossa implementação, definimos quatro entradas, *clk* (relógio), *E_FLAG*, *ESCR_R* (sinal de escrita no registo) e *SEL_FLAG* (seleção do bit a ser enviado para *S_FLAG*), e uma saída *S_FLAG* (recebe um dos bits armazenados no *registo*).

Utilizando uma arquitetura "Behavioral", implementamos um processo sensível à transição ascendente do relógio (*clk*), que verifica se o bit mais significativo de *ESCR_R* está em estado '1'. Em caso afirmativo, o valor do sinal *E_FLAG* é armazenado na variável *registo*. Em seguida, utilizamos uma estrutura de seleção de casos para atribuir o valor de um dos bits da variável *registo* à saída *S_FLAG*, dependendo do valor de *SEL_FLAG*. Se o valor de *SEL_FLAG* não corresponder a nenhum dos casos especificados, atribuímos 'X' a *S_FLAG*, indicando um estado indefinido.

7. Contador de Programa (PC)

O Contador de Programa desempenha um papel crucial na execução de um programa, indicando a posição atual na sequência de instruções.

Na nossa implementação, implementamos um sinal de relógio (*clk*), um sinal de reset (*reset*), um sinal de entrada para a constante (*Constante*), um sinal de controlo (*ESCR_PC*), e uma única saída *Endereco*. Temos também uma variável *contagem* que contém o valor do Contador de Programa.

Utilizando uma arquitetura "Behavioral", definimos um processo que monitora as mudanças nas quatro entradas. Na transição ascendente do sinal de relógio, o valor de *contagem* é atualizado de acordo com as seguintes condições:

- Se o sinal de *reset* estiver a '0', a *contagem* é incrementada se *ESCR_PC* for '0', caso contrário, é atribuído o valor da entrada *Constante* à *contagem*.
- Se o sinal de *reset* estiver a '1', a *contagem* é reiniciada a zero.

O valor final de contagem é atribuído à saída *Endereco*, indicando assim a próxima posição na sequência de instruções a ser executada.

8. Multiplexer do Contador de Programa (Mux_PC)

O Multiplexer do Contador de Programa desempenha um papel crucial na instrução do PC caso deva executar um salto ou simplesmente incrementar o contador. Este módulo recebe várias entradas de controlo, incluindo seleções para determinar qual sinal deve ser encaminhado para o PC.

Na nossa implementação, definimos um total de seis entradas, *zero*, *um*, *S_FLAG*, *Operando1*, *NOR_Operando1* e *SEL_PC*, e uma saída *ESCR_PC*.

Através de uma estrutura de seleção de casos, dependendo do valor de *SEL_PC*, um valor de uma das entradas é atribuído a *ESCR_PC*. Por exemplo, se *SEL_PC* for "000", o sinal *zero* é atribuído, se for "001", o sinal um é atribuído, e assim por diante. Se o valor de *SEL_PC* não corresponder a nenhum dos casos especificados, a saída *ESCR_PC* é definida como 'X', indicando um estado indeterminado.

9. ROM de Descodificação (ROM)

A Unidade de ROM de Descodificação desempenha um papel crucial ao fornecer os sinais de controlo necessários aos outros blocos do sistema.

Na nossa implementação, definimos como entrada o *opcode*, e como saídas os sinais de controlo *WR*, *ESCR_P*, *SEL_Dados*, *ESCR_R*, *SEL_ALU*, *SEL_FLAG* e *SEL_PC*.

Utilizando uma arquitetura "Behavioral", implementamos um processo sensível a mudanças no opcode. Utilizando uma estrutura de seleção "case", associamos a cada valor de opcode uma configuração específica dos sinais de controlo.

10. Memória de Instruções

A Memória de Instruções desempenha um papel fundamental no processamento de um programa, sendo responsável por armazenar as instruções a serem executadas.

Na nossa implementação, definimos como entrada o *Endereco*, *opcode*, e como saídas *Constante* e *SEL_R*.

Utilizando uma arquitetura "Behavioral", criamos um processo sensível a mudanças no sinal *Endereco*. Utilizamos um case para mapear cada endereço para as instruções correspondentes, atribuindo os valores apropriados às saídas *opcode*, *Constante* e *SEL_R*. Esta estrutura simula o comportamento de uma memória de instruções, onde cada endereço contém uma instrução específica a ser executada pelo processador.

11. Memória de Dados (RAM)

A Memória de Dados desempenha um papel fundamental na conservação dos dados.

Na nossa implementação, definimos cinco entradas, *Constante* (indica o endereço na memória onde os dados devem ser escritos ou lidos), *WR* (controla a operação de escrita (quando '1') ou leitura (quando '0')), *clk*, *Operando1* e *Dados_M*, e uma saída *Dados_M*. Os dados a serem escritos são fornecidos através do sinal *Operando1*, e os dados lidos são disponibilizados na saída *Dados_M*.

Utilizando uma arquitetura "Behavioral", implementamos um processo sensível às mudanças nas entradas. Quando *WR* está em estado '1' e ocorre uma transição ascendente do sinal de relógio, os dados presentes em *Operando1* são escritos na posição de memória indicada pelo sinal de entrada *Constante*. Por outro lado, quando *WR* está em estado '0', é realizada uma leitura na posição de memória indicada pelo sinal de entrada *Constante*, e os dados correspondentes são atribuídos à saída *Dados_M*.

12. Processador e Placa-Mãe

O Processador e a Placa-Mãe são componentes essenciais de um sistema computacional, trabalhando em conjunto para executar operações e gerenciar recursos.

Tanto o Processador quanto a Placa-Mãe são desenvolvidos de forma estruturada, utilizando port maps e sinais para interligar os diversos módulos e garantir o funcionamento eficiente e coordenado do sistema computacional.

Discussão de resultados

Após a execução da simulação do teste da placa-mãe e a análise dos dados obtidos, pudemos observar que os valores de *POUT* variam de acordo com os valores de *PIN*. O intervalo de variação de *PIN* vai de -128 a 127.

Observações dos Resultados:

- Para *PIN* menor que -21, inclusive, o valor de *POUT* é 1 se o número for ímpar e 0 se for par (*IMG.2*).
- No intervalo de -20 a -17, inclusive, ocorre um overflow, onde os valores continuam a ser múltiplos de 8 (como no intervalo de -16 a -1), mas os números exibidos são positivos, ultrapassando o valor mínimo de -128 de *POUT* (*IMG.3* no *PIN*=-17).
- Para *PIN* no intervalo de -16 a -1, inclusive, o valor de *POUT* é o resultado de *PIN* multiplicado por 8 (*IMG.3*).
- Quando *PIN* é igual a 0, o valor de *POUT* é único e é -1 (*IMG.4*).
- Para *PIN* no intervalo de 1 a 15, inclusive, o valor de *POUT* é o resultado de *PIN* multiplicado por 8 (*IMG.5*).
- No intervalo de 16 a 19, inclusive, ocorre um overflow, onde os valores continuam a ser múltiplos de 8 (como no intervalo de 1 a 15), mas os números exibidos são negativos, ultrapassando o valor máximo de 127 de *POUT* (*IMG.5* no *PIN*=16).
- Para *PIN* maior que 20, inclusive, o valor de *POUT* é o resultado de *PIN* subtraído por 15(*IMG.6*).

Conclusão

O desenvolvimento do processador básico em VHDL permitiu uma imersão prática nos conceitos fundamentais de arquitetura de computadores. Ao integrar os diversos componentes, consolidamos habilidades em VHDL e compreendemos a importância da sua aplicação em sistemas embarcados.

A análise dos resultados da simulação evidenciou o desempenho da placa-mãe, corroborando a eficiência do processador na manipulação de dados de entrada. Além disso, destacamos observações relevantes, como tratamento de overflow e valores negativos, contribuindo para uma compreensão mais ampla da arquitetura de computadores e sua implementação em FPGA.

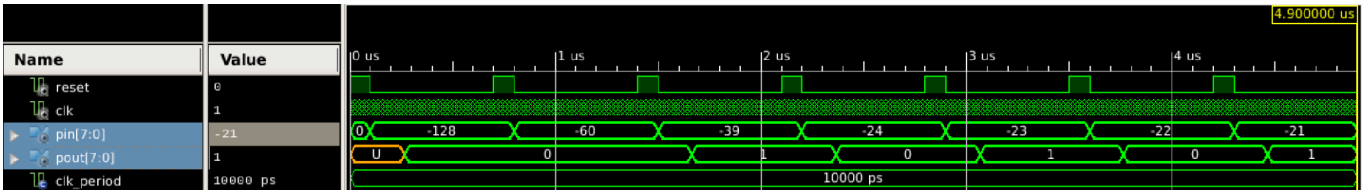
Bibliografia

Materiais de referência utilizados incluem as práticas laboratoriais, seus exercícios, e slides fornecidos durante o seu curso.

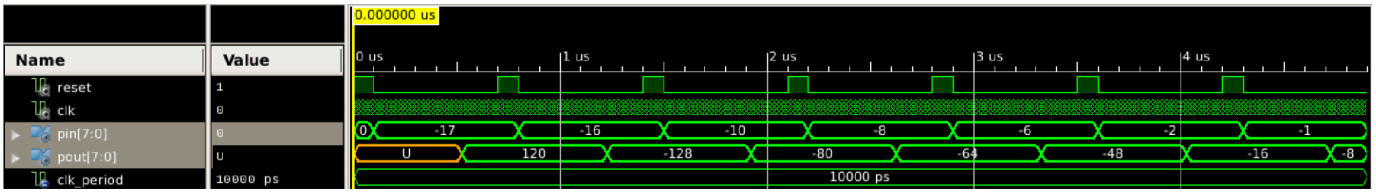
Anexo A

Endereço	Instrução (*assembly*)	Instrução (código máquina)		
		opcode	**SEL_R**	**Constante**
00000000	LDP R0	00000	XXX000	XXXXXXXX
00000001	JZ R0, 24	10100	XXX000	00011000
00000010	ST [5], R0	00100	XXX000	00000101
00000011	LD R0, 1	00010	XXX000	00000001
00000100	LD R1, -1	00010	XXX001	11111111
00000101	LD R2, 20	00010	XXX010	00010100
00000110	LD R3, 6	00010	XXX011	00000110
00000111	LD R7, [5]	00011	XXX111	00000101
00001000	CMP R7, R2	01101	010111	XXXXXXXX
00001001	JGE 19	10001	XXXXXX	00010011
00001010	XOR R2, R1	01011	001010	XXXXXXXX
00001011	ADD R2, R0	00101	000010	XXXXXXXX
00001100	CMP R7, R2	01101	010111	XXXXXXXX
00001101	JL 22	01110	XXXXXX	00010110
00001110	LD R6, [5]	00011	XXX110	00000101
00001111	ADD R7, R6	00101	110111	XXXXXXXX
00010000	SUB R3, R0	00110	000011	XXXXXXXX
00010001	JN R3, 25	10101	XXX011	00011001
00010010	JMP 15	10011	XXXXXX	00001111
00010011	LD R4, -15	00010	XXX100	11110001
00010100	ADD R7, R4	00101	100111	XXXXXXXX
00010101	JMP 25	10011	XXXXXX	00011001
00010110	AND R7, R0	00111	000111	XXXXXXXX
00010111	JMP 25	10011	XXXXXX	00011001
00011000	LD R7, -1	00010	XXX111	11111111
00011001	STP R7	00001	XXX111	XXXXXXXX
00011010	JMP 26	10011	XXXXXX	00011010

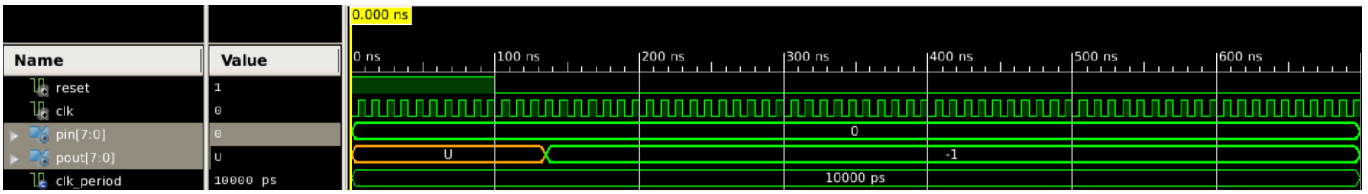
Img 1 - Instruções de teste do projeto



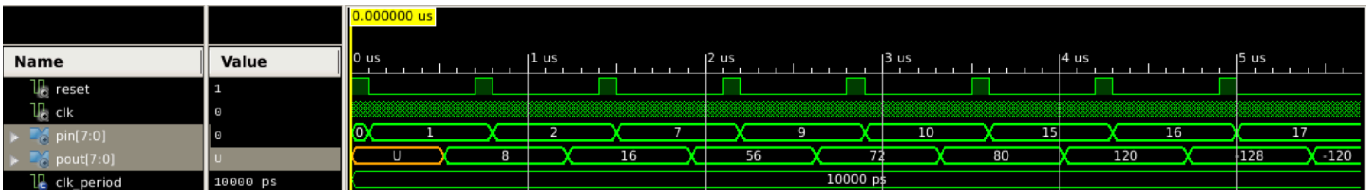
Img 2 - Teste quando PIN é menor ou igual a -21



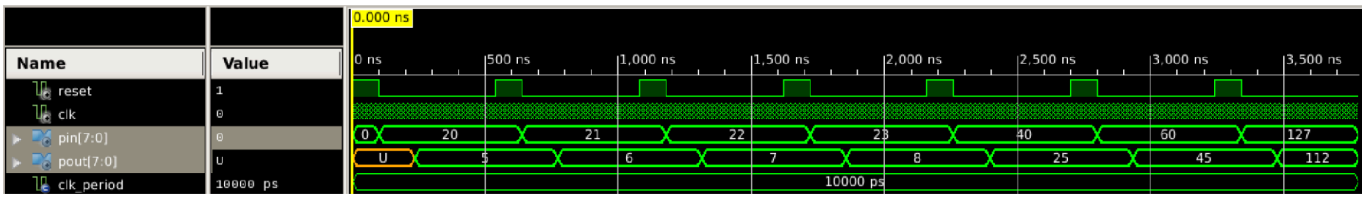
Img 3 - Teste quando PIN está entre -1 e -16, depois de -16 ocorre overflow



Img 4 - Teste quando PIN está a zero



Img 5 - Teste quando PIN está entre 1 e 15, depois de 15 ocorre overflow



Img 6 - Teste quando PIN é maior ou igual a 20

Anexo B

Placa-mãe

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

entity Placa_Mae is
   Port ( reset : in  STD_LOGIC;
          clk : in STD_LOGIC;
          PIN : in  STD_LOGIC_VECTOR (7 downto 0);
          POUT : out  STD_LOGIC_VECTOR (7 downto 0));
end Placa_Mae;

architecture Struct of Placa_Mae is

Component Processador is
   Port ( reset : in  STD_LOGIC;
          opcode : in  STD_LOGIC_VECTOR (4 downto 0);
          SEL_R : in  STD_LOGIC_VECTOR (5 downto 0);
          Constante_IN : in  STD_LOGIC_VECTOR (7 downto 0);
          Dados_M : in  STD_LOGIC_VECTOR (7 downto 0);
          PIN : in  STD_LOGIC_VECTOR (7 downto 0);
                        clk : STD_LOGIC;
                        Constante_OUT : out  STD_LOGIC_VECTOR (7 downto 0);
          Endereco : out  STD_LOGIC_VECTOR (7 downto 0);
          WR : out  STD_LOGIC;
          Operando1 : out  STD_LOGIC_VECTOR (7 downto 0);
          POUT : out  STD_LOGIC_VECTOR (7 downto 0));
end Component;

Component Memoria_de_Dados is
   Port ( Constante : in  STD_LOGIC_VECTOR (7 downto 0);
          WR : in  STD_LOGIC;
          clk : in  STD_LOGIC;
          Operando1 : in  STD_LOGIC_VECTOR (7 downto 0);
          Dados_M : out  STD_LOGIC_VECTOR (7 downto 0));
end Component;

Component Memoria_de_Instrucoes is
   Port ( Endereco : in  STD_LOGIC_VECTOR (7 downto 0);
          opcode : out  STD_LOGIC_VECTOR (4 downto 0);
          Constante : out  STD_LOGIC_VECTOR (7 downto 0);
          SEL_R : out  STD_LOGIC_VECTOR (5 downto 0));
end Component;
```

```
signal WR : STD_LOGIC;
signal opcode : STD_LOGIC_VECTOR(4 downto 0);
signal SEL_R : STD_LOGIC_VECTOR(5 downto 0);
signal Endereco, Constante_IN, Constante_OUT, Dados_M, Operando1 :
STD_LOGIC_VECTOR(7 downto 0);

begin

        Processador_PM : Processador port map (reset, opcode, SEL_R, Constante_IN,
Dados_M, PIN, clk, Constante_OUT, Endereco, WR, Operando1, POUT);
        Memoria_de_Dados_PM : Memoria_de_Dados port map (Constante_OUT, WR, clk,
Operando1, Dados_M);
        Memoria_de_Instrucoes_PM : Memoria_de_Instrucoes port map (Endereco, opcode,
Constante_IN, SEL_R);

end Struct;
```

Processador

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

entity Processador is
   Port ( reset : in  STD_LOGIC;
          opcode : in  STD_LOGIC_VECTOR (4 downto 0);
          SEL_R : in  STD_LOGIC_VECTOR (5 downto 0);
          Constante_IN : in  STD_LOGIC_VECTOR (7 downto 0);
          Dados_M : in  STD_LOGIC_VECTOR (7 downto 0);
          PIN : in  STD_LOGIC_VECTOR (7 downto 0);
                        clk : STD_LOGIC;
                        Constante_OUT : out  STD_LOGIC_VECTOR (7 downto 0);
          Endereco : out  STD_LOGIC_VECTOR (7 downto 0);
          WR : out  STD_LOGIC;
          Operando1 : out  STD_LOGIC_VECTOR (7 downto 0);
          POUT : out  STD_LOGIC_VECTOR (7 downto 0));
end Processador;

architecture Struct of Processador is

Component PC is
   Port ( clk : in  STD_LOGIC;
          reset : in  STD_LOGIC;
          Constante : in  STD_LOGIC_VECTOR (7 downto 0);
```

```
        ESCR_PC : in  STD_LOGIC;
        Endereco : out  STD_LOGIC_VECTOR (7 downto 0));
end Component;

Component MUX_PC is
   Port ( zero : in  STD_LOGIC;
        um : in  STD_LOGIC;
        S_FLAG : in  STD_LOGIC;
        O7 : in  STD_LOGIC;
        SEL_PC : in  STD_LOGIC_VECTOR (2 downto 0);
                        NOR_Operando1 : in STD_LOGIC;
        ESCR_PC : out  STD_LOGIC);
end Component;

Component Registo_Flags is
   Port ( clk : in  STD_LOGIC;
        E_FLAG : in  STD_LOGIC_VECTOR (4 downto 0);
        ESCR_R : in  STD_LOGIC_VECTOR (1 downto 0);
        SEL_FLAG : in  STD_LOGIC_VECTOR (2 downto 0);
        S_FLAG : out  STD_LOGIC);
end Component;

Component ROM_de_Descodificacao is
   Port ( opcode : in  STD_LOGIC_VECTOR (4 downto 0);
        WR : out  STD_LOGIC;
        ESCR_P : out  STD_LOGIC;
        SEL_Dados : out  STD_LOGIC_VECTOR (1 downto 0);
        ESCR_R : out  STD_LOGIC_VECTOR (1 downto 0);
        SEL_ALU : out  STD_LOGIC_VECTOR (3 downto 0);
        SEL_FLAG : out  STD_LOGIC_VECTOR (2 downto 0);
        SEL_PC : out  STD_LOGIC_VECTOR (2 downto 0));
end Component;

Component ALU is
   Port ( SEL_ALU : in  STD_LOGIC_VECTOR (3 downto 0);
        Operando1 : in  STD_LOGIC_VECTOR (7 downto 0);
        Operando2 : in  STD_LOGIC_VECTOR (7 downto 0);
        E_FLAG : out  STD_LOGIC_VECTOR (4 downto 0);
        Resultado : out  STD_LOGIC_VECTOR (7 downto 0));
end Component;

Component Banco_de_Registos is
   Port ( ESCR_R : in  STD_LOGIC_VECTOR (1 downto 0);
        clk : in  STD_LOGIC;
```

```
        SEL_R : in  STD_LOGIC_VECTOR (5 downto 0);
        Dados_R : in  STD_LOGIC_VECTOR (7 downto 0);
        Operando1 : out  STD_LOGIC_VECTOR (7 downto 0);
        Operando2 : out  STD_LOGIC_VECTOR (7 downto 0));
end Component;

Component MUX_R is
   Port ( SEL_Dados : in  STD_LOGIC_VECTOR (1 downto 0);
        Constante : in  STD_LOGIC_VECTOR (7 downto 0);
        Dados_M : in  STD_LOGIC_VECTOR (7 downto 0);
        Dados_IN : in  STD_LOGIC_VECTOR (7 downto 0);
        Resultado : in  STD_LOGIC_VECTOR (7 downto 0);
        Dados_R : out  STD_LOGIC_VECTOR (7 downto 0));
end Component;

Component Periferico_de_Entrada is
   Port ( ESCR_P : in  STD_LOGIC;
        PIN : in  STD_LOGIC_VECTOR (7 downto 0);
        Dados_IN : out  STD_LOGIC_VECTOR (7 downto 0));
end Component;

Component Periferico_de_Saida is
   Port ( Operando1 : in  STD_LOGIC_VECTOR (7 downto 0);
        ESCR_P : in  STD_LOGIC;
        clk : in  STD_LOGIC;
        POUT : out  STD_LOGIC_VECTOR (7 downto 0));
end Component;

Component NOR_Operando is
   Port ( O0 : in  STD_LOGIC;
                        O1 : in  STD_LOGIC;
                        O2 : in  STD_LOGIC;
                        O3 : in  STD_LOGIC;
                        O4 : in  STD_LOGIC;
                        O5 : in  STD_LOGIC;
                        O6 : in  STD_LOGIC;
                        O7 : in  STD_LOGIC;
        NOR_Operando1 : out  STD_LOGIC);
end Component;

signal ESCR_PC, S_FLAG, ESCR_P, NOR_Operando1 : STD_LOGIC;
signal ESCR_R, SEL_Dados : STD_LOGIC_VECTOR (1 downto 0);
signal SEL_FLAG, SEL_PC : STD_LOGIC_VECTOR (2 downto 0);
signal SEL_ALU : STD_LOGIC_VECTOR (3 downto 0);
```

```
signal E_FLAG : STD_LOGIC_VECTOR (4 downto 0);
signal Sinal_Operando1, Operando2, Resultado, Dados_R, Dados_IN :
STD_LOGIC_VECTOR (7 downto 0);

begin

        PC_P : PC port map (clk, reset, Constante_IN, ESCR_PC, Endereco);
        MUX_PC_P : MUX_PC port map ('0', '1', S_FLAG, Sinal_Operando1(7), SEL_PC,
NOR_Operando1, ESCR_PC);
        Registo_Flags_P : Registo_Flags port map (clk, E_FLAG, ESCR_R, SEL_FLAG,
S_FLAG);
        ROM_de_Descodificacao_P : ROM_de_Descodificacao port map (opcode, WR,
ESCR_P, SEL_Dados, ESCR_R, SEL_ALU, SEL_FLAG, SEL_PC);
        MUX_R_P : MUX_R port map (SEL_Dados, Constante_IN, Dados_M, Dados_IN,
Resultado, Dados_R);
        Banco_de_Registos_P : Banco_de_Registos port map (ESCR_R, clk, SEL_R,
Dados_R, Sinal_Operando1, Operando2);
        ALU_P : ALU port map (SEL_ALU, Sinal_Operando1, Operando2, E_FLAG,
Resultado);
        Periferico_de_Entrada_P : Periferico_de_Entrada port map (ESCR_P, PIN,
Dados_IN);
        Periferico_de_Saida_P : Periferico_de_Saida port map (Sinal_Operando1, ESCR_P,
clk, POUT);
        NOR_Operando_P : NOR_Operando port map (Sinal_Operando1(0),
Sinal_Operando1(1), Sinal_Operando1(2), Sinal_Operando1(3), Sinal_Operando1(4),
Sinal_Operando1(5), Sinal_Operando1(6), Sinal_Operando1(7), NOR_Operando1);

        Constante_OUT <= Constante_IN;
        Operando1 <= Sinal_Operando1;

end Struct;
```

Periférico de Entrada

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

entity Periferico_de_Entrada is
   Port ( ESCR_P : in  STD_LOGIC;
          PIN : in  STD_LOGIC_VECTOR (7 downto 0);
          Dados_IN : out  STD_LOGIC_VECTOR (7 downto 0));
end Periferico_de_Entrada;

architecture Behavioral of Periferico_de_Entrada is
```

```
begin

        process(ESCR_P, PIN)
        begin

                -- Se o sinal ESCR_P está a 0
                if ESCR_P = '0' then

                        -- Atribui o valor da entrada PIN á saída Dados_IN
                        Dados_IN <= PIN;

                end if;

        end process;

end Behavioral;
```

Periférico de Saída

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

entity Periferico_de_Saida is
   Port ( Operando1 : in  STD_LOGIC_VECTOR (7 downto 0);
          ESCR_P : in  STD_LOGIC;
          clk : in  STD_LOGIC;
          POUT : out  STD_LOGIC_VECTOR (7 downto 0));
end Periferico_de_Saida;

architecture Behavioral of Periferico_de_Saida is

begin

        process(Operando1, ESCR_P, clk)
        begin

                -- Se o sinal ESCR_P está a 1
                if ESCR_P = '1' then

                        -- Se está na transição ascendente do relógio
                        if rising_edge(clk) then

                                -- Atribui o valor da entrada Operando1 á saída POUT
```

```
                    POUT <= Operando1;

                end if;

            end if;

        end process;

end Behavioral;
```

Multiplexer do Banco de Registos (Mux R)

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

entity MUX_R is
    Port ( SEL_Dados : in  STD_LOGIC_VECTOR (1 downto 0);
           Constante : in  STD_LOGIC_VECTOR (7 downto 0);
           Dados_M : in  STD_LOGIC_VECTOR (7 downto 0);
           Dados_IN : in  STD_LOGIC_VECTOR (7 downto 0);
           Resultado : in  STD_LOGIC_VECTOR (7 downto 0);
           Dados_R : out  STD_LOGIC_VECTOR (7 downto 0));
end MUX_R;

architecture Behavioral of MUX_R is
begin

        process(SEL_Dados, Constante, Dados_M, Dados_IN, Resultado)
        begin

            -- Início da estrutura de seleção de casos dependendo do valor de SEL_Dados
            case SEL_Dados is

                -- Atribui um valor á saída Dados_R dependendo do valor de
SEL_Dados
                when "00" => Dados_R <= Resultado;
                when "01" => Dados_R <= Dados_IN;
                when "10" => Dados_R <= Dados_M;
                when "11" => Dados_R <= Constante;
                -- Para qualquer outro valor de SEL_Dados, atribui 'X' a Dados_R
                when others => Dados_R <= (others => 'X');

            end case;
```

```
        end process;

end Behavioral;
```

Banco de Registos

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

entity Banco_de_Registos is
   Port ( ESCR_R : in  STD_LOGIC_VECTOR (1 downto 0);
          clk : in  STD_LOGIC;
          SEL_R : in  STD_LOGIC_VECTOR (5 downto 0);
          Dados_R : in  STD_LOGIC_VECTOR (7 downto 0);
          Operando1 : out  STD_LOGIC_VECTOR (7 downto 0);
          Operando2 : out  STD_LOGIC_VECTOR (7 downto 0));
end Banco_de_Registos;

architecture Behavioral of Banco_de_Registos is

begin

        process(ESCR_R, clk, SEL_R, Dados_R)

        -- Array que contem o 8 registos de 8 bits
        type array_r is array (0 to 7) of STD_LOGIC_VECTOR (7 downto 0); -- Guardar os
dados
        variable registos : array_r;

        begin

                -- Se o sinal do bit menos significativo de ESCR_P está a 1
                if ESCR_R(0) = '1' then

                        -- Se está na transição ascendente do relógio
                        if rising_edge(clk) then

                                -- Atribui os valores aos registos dependendo dos três bits
menos significativos de SEL_R
                                registos(to_integer(unsigned(SEL_R (2 downto 0)))) :=
Dados_R;

                        end if;
```

```
            end if;

            -- Atribui o valor do registo correspondente a cada operando
            Operando1 <= registos(to_integer(unsigned(SEL_R (2 downto 0))));
            Operando2 <= registos(to_integer(unsigned(SEL_R (5 downto 3))));

        end process;

end Behavioral;
```

Unidade Lógica e Aritmética (ALU)

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

entity ALU is
   Port ( SEL_ALU : in  STD_LOGIC_VECTOR (3 downto 0);
          Operando1 : in  STD_LOGIC_VECTOR (7 downto 0);
          Operando2 : in  STD_LOGIC_VECTOR (7 downto 0);
          E_FLAG : out  STD_LOGIC_VECTOR (4 downto 0);
          Resultado : out  STD_LOGIC_VECTOR (7 downto 0));
end ALU;

architecture Behavioral of ALU is

begin

        process(SEL_ALU, Operando1, Operando2)
        begin

            -- Início da estrutura de seleção de casos dependendo do valor de SEL_ALU
            case SEL_ALU is

                -- Atribui o valor da operação á variável operacao dependendo do valor
de SEL_ALU
                when "0000" => Resultado <= Operando1 + Operando2;
                when "0001" => Resultado <= Operando1 - Operando2;
                when "0010" => Resultado <= Operando1 and Operando2;
                when "0011" => Resultado <= not(Operando1 and Operando2);
                when "0100" => Resultado <= Operando1 or Operando2;
                when "0101" => Resultado <= not(Operando1 or Operando2);
```

```
                        when "0110" => Resultado <= Operando1 xor Operando2;
                        when "0111" => Resultado <= not(Operando1 xor Operando2);
                        -- Atribui o valor 1 ao bit de E_FLAG dependendo da comparação
                        when "1000" =>
                                -- Todos os bits de E_FLAG ficam a 0 no início
                                E_FLAG <= (others => '0');

        if (Operando1 < Operando2) then
           E_FLAG(0) <= '1';
        end if;
        if (Operando1 <= Operando2) then
           E_FLAG(1) <= '1';
        end if;
        if (Operando1 = Operando2) then
           E_FLAG(2) <= '1';
        end if;
        if (Operando1 >= Operando2) then
           E_FLAG(3) <= '1';
        end if;
        if (Operando1 > Operando2) then
           E_FLAG(4) <= '1';
        end if;
                        -- Para qualquer outro valor de SEL_ALU, atribui 'X' a E_FLAG e a
Resultado
                        when others => E_FLAG <= (others => 'X'); Resultado <= (others =>
'X');

                end case;

        end process;

end Behavioral;
```

Registo de Flags

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

entity Registo_Flags is
   Port ( clk : in  STD_LOGIC;
          E_FLAG : in  STD_LOGIC_VECTOR (4 downto 0);
          ESCR_R : in  STD_LOGIC_VECTOR (1 downto 0);
          SEL_FLAG : in  STD_LOGIC_VECTOR (2 downto 0);
          S_FLAG : out  STD_LOGIC);
```

```
end Registo_Flags;

architecture Behavioral of Registo_Flags is

begin

	process(clk, E_FLAG, ESCR_R, SEL_FLAG)

	-- Declaração da variavel registo
	variable registo: STD_LOGIC_VECTOR (4 downto 0);

	begin

		-- Se está na transição ascendente do relógio
		if rising_edge(clk) then

			-- Se o bit mais significativo de ESCR_R está a 1
			if ESCR_R(1) = '1' then

				-- Guarda o valor da entrada E_FLAG na variável registo
				registo := E_FLAG;

			end if;

		end if;

		-- Início da estrutura de seleção de casos dependendo do valor de SEL_FLAG
		case SEL_FLAG is

			-- Atribui o valor de um bit da variavel registo á saída S_FLAG
dependendo do valor de SEL_FLAG
			when "000" => S_FLAG <= registo(0);
			when "001" => S_FLAG <= registo(1);
			when "010" => S_FLAG <= registo(2);
			when "011" => S_FLAG <= registo(3);
			when "100" => S_FLAG <= registo(4);
			-- Para qualquer outro valor de SEL_FLAG, atribui 'X' a S_FLAG
			when others => S_FLAG <= 'X';

		end case;

	end process;

end Behavioral;
```

Contador de Programa (PC)

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

entity PC is
   Port ( clk : in  STD_LOGIC;
          reset : in  STD_LOGIC;
          Constante : in  STD_LOGIC_VECTOR (7 downto 0);
          ESCR_PC : in  STD_LOGIC;
          Endereco : out  STD_LOGIC_VECTOR (7 downto 0));
end PC;

architecture Behavioral of PC is

begin

          process(clk, reset, ESCR_PC, Constante)
          -- Declaração da variavel contagem responsável por guardar o valor da posição atual
do programa
          variable contagem : STD_LOGIC_VECTOR (7 downto 0) := "00000000";
  begin

                    -- Se está na transição ascendente do relógio
                    if rising_edge(clk) then

                              -- Se o sinal do reset for 0
                              if reset = '0' then

                                        -- Se o sinal do ESCR_PC for 0 aumenta em um o valor da
variável contagem
                                        if ESCR_PC = '0' then

                                                  contagem := contagem + "00000001";

                                        -- Se o sinal do ESCR_PC for 1 atribui o valor da entrada
Constante á variável contagem
                                        else

                                                  contagem := Constante;

                                        end if;
```

```
                    -- Se o sinal do reset for 1 o valor da variável contagem passa a zero
                    else

                     contagem := "00000000";

                    end if;

                    -- Atribui o valor da variável contagem á saída Endereco
                    Endereco <= contagem;

                end if;

    end process;

end Behavioral;
```

Multiplexer do Program Counter (PC)

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

entity MUX_PC is
    Port ( zero : in  STD_LOGIC;
           um : in  STD_LOGIC;
           S_FLAG : in  STD_LOGIC;
           O7 : in  STD_LOGIC;
           SEL_PC : in  STD_LOGIC_VECTOR (2 downto 0);
                        NOR_Operando1 : in STD_LOGIC;
           ESCR_PC : out  STD_LOGIC);
end MUX_PC;

architecture Behavioral of MUX_PC is

begin

          process(zero, um, S_FLAG, O7, SEL_PC, NOR_Operando1)
          begin

                -- Início da estrutura de seleção de casos dependendo do valor de SEL_PC
                case SEL_PC is

                    -- Atribui o valor de uma das entradas á saída ESCR_PC dependendo
do valor de SEL_PC
                    when "000" => ESCR_PC <= zero;
```

```
                    when "001" => ESCR_PC <= um;
                    when "010" => ESCR_PC <= S_FLAG;
                    when "011" => ESCR_PC <= O7;
                    when "100" => ESCR_PC <= NOR_Operando1;
                    -- Para qualquer outro valor de SEL_PC, atribui 'X' a ESCR_PC
                    when others => ESCR_PC <= 'X';

            end case;

    end process;

end Behavioral;
```

ROM de descodificação (ROM)

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

entity ROM_de_Descodificacao is
   Port ( opcode : in  STD_LOGIC_VECTOR (4 downto 0);
          WR : out  STD_LOGIC;
          ESCR_P : out  STD_LOGIC;
          SEL_Dados : out  STD_LOGIC_VECTOR (1 downto 0);
          ESCR_R : out  STD_LOGIC_VECTOR (1 downto 0);
          SEL_ALU : out  STD_LOGIC_VECTOR (3 downto 0);
          SEL_FLAG : out  STD_LOGIC_VECTOR (2 downto 0);
          SEL_PC : out  STD_LOGIC_VECTOR (2 downto 0));
end ROM_de_Descodificacao;

architecture Behavioral of ROM_de_Descodificacao is

begin

        process(opcode)
        begin

                -- Início da estrutura de seleção de casos dependendo do valor de opcode
                case opcode is

                --Periféricos
                        -- LDP Ri
                        when "00000" => SEL_ALU <= "XXXX"; ESCR_P <= '0';
SEL_Dados <= "01"; ESCR_R <= "01"; WR <= '0'; SEL_PC <= "000"; SEL_FLAG <=
"XXX";
```

```
                        -- STP Ri
                        when "00001" => SEL_ALU <= "XXXX"; ESCR_P <= '1';
SEL_Dados <= "XX"; ESCR_R <= "00"; WR <= '0'; SEL_PC <= "000"; SEL_FLAG <=
"XXX";

                --Leitura e Escrita
                        -- LD Ri, constante
                        when "00010" => SEL_ALU <= "XXXX"; ESCR_P <= '0';
SEL_Dados <= "11"; ESCR_R <= "01"; WR <= '0'; SEL_PC <= "000"; SEL_FLAG <=
"XXX";
                        -- LD Ri, [constante]
                        when "00011" => SEL_ALU <= "XXXX"; ESCR_P <= '0';
SEL_Dados <= "10"; ESCR_R <= "01"; WR <= '0'; SEL_PC <= "000"; SEL_FLAG <=
"XXX";
                        -- ST [constante], Ri
                        when "00100" => SEL_ALU <= "XXXX"; ESCR_P <= '0';
SEL_Dados <= "XX"; ESCR_R <= "00"; WR <= '1'; SEL_PC <= "000"; SEL_FLAG <=
"XXX";

                --Lógica e Aritmética
                        -- ADD Ri, Rj
                        when "00101" => SEL_ALU <= "0000"; ESCR_P <= '0'; SEL_Dados
<= "00"; ESCR_R <= "01"; WR <= '0'; SEL_PC <= "000"; SEL_FLAG <= "XXX";
                        -- SUB Ri, Rj
                        when "00110" => SEL_ALU <= "0001"; ESCR_P <= '0'; SEL_Dados
<= "00"; ESCR_R <= "01"; WR <= '0'; SEL_PC <= "000"; SEL_FLAG <= "XXX";
                        -- AND Ri, Rj
                        when "00111" => SEL_ALU <= "0010"; ESCR_P <= '0'; SEL_Dados
<= "00"; ESCR_R <= "01"; WR <= '0'; SEL_PC <= "000"; SEL_FLAG <= "XXX";
                        -- NAND Ri, Rj
                        when "01000" => SEL_ALU <= "0011"; ESCR_P <= '0'; SEL_Dados
<= "00"; ESCR_R <= "01"; WR <= '0'; SEL_PC <= "000"; SEL_FLAG <= "XXX";
                        -- OR Ri, Rj
                        when "01001" => SEL_ALU <= "0100"; ESCR_P <= '0'; SEL_Dados
<= "00"; ESCR_R <= "01"; WR <= '0'; SEL_PC <= "000"; SEL_FLAG <= "XXX";
                        -- NOR Ri, Rj
                        when "01010" => SEL_ALU <= "0101"; ESCR_P <= '0'; SEL_Dados
<= "00"; ESCR_R <= "01"; WR <= '0'; SEL_PC <= "000"; SEL_FLAG <= "XXX";
                        -- XOR Ri, Rj
                        when "01011" => SEL_ALU <= "0110"; ESCR_P <= '0'; SEL_Dados
<= "00"; ESCR_R <= "01"; WR <= '0'; SEL_PC <= "000"; SEL_FLAG <= "XXX";
                        -- XNOR Ri, Rj
                        when "01100" => SEL_ALU <= "0111"; ESCR_P <= '0'; SEL_Dados
<= "00"; ESCR_R <= "01"; WR <= '0'; SEL_PC <= "000"; SEL_FLAG <= "XXX";
```

```
                        -- CMP Ri, Rj
                        when "01101" => SEL_ALU <= "1000"; ESCR_P <= '0'; SEL_Dados
<= "XX"; ESCR_R <= "10"; WR <= '0'; SEL_PC <= "000"; SEL_FLAG <= "XXX";

                --Salto
                        -- JL constante
                        when "01110" => SEL_ALU <= "XXXX"; ESCR_P <= '0';
SEL_Dados <= "XX"; ESCR_R <= "00"; WR <= '0'; SEL_PC <= "010"; SEL_FLAG <=
"000";
                        -- JLE constante
                        when "01111" => SEL_ALU <= "XXXX"; ESCR_P <= '0';
SEL_Dados <= "XX"; ESCR_R <= "00"; WR <= '0'; SEL_PC <= "010"; SEL_FLAG <=
"001";
                        -- JE constante
                        when "10000" => SEL_ALU <= "XXXX"; ESCR_P <= '0';
SEL_Dados <= "XX"; ESCR_R <= "00"; WR <= '0'; SEL_PC <= "010"; SEL_FLAG <=
"010";
                        -- JGE constante
                        when "10001" => SEL_ALU <= "XXXX"; ESCR_P <= '0';
SEL_Dados <= "XX"; ESCR_R <= "00"; WR <= '0'; SEL_PC <= "010"; SEL_FLAG <=
"011";
                        -- JG constante
                        when "10010" => SEL_ALU <= "XXXX"; ESCR_P <= '0';
SEL_Dados <= "XX"; ESCR_R <= "00"; WR <= '0'; SEL_PC <= "010"; SEL_FLAG <=
"100";
                        -- JMP constante
                        when "10011" => SEL_ALU <= "XXXX"; ESCR_P <= '0';
SEL_Dados <= "XX"; ESCR_R <= "00"; WR <= '0'; SEL_PC <= "001"; SEL_FLAG <=
"XXX";
                        -- JZ Ri, constante
                        when "10100" => SEL_ALU <= "XXXX"; ESCR_P <= '0';
SEL_Dados <= "XX"; ESCR_R <= "00"; WR <= '0'; SEL_PC <= "100"; SEL_FLAG <=
"XXX";
                        -- JN, Ri, constante
                        when "10101" => SEL_ALU <= "XXXX"; ESCR_P <= '0';
SEL_Dados <= "XX"; ESCR_R <= "00"; WR <= '0'; SEL_PC <= "011"; SEL_FLAG <=
"XXX";

                --Outros
                        when others => SEL_ALU <= "XXXX"; ESCR_P <= '0'; SEL_Dados
<= "XX"; ESCR_R <= "00"; WR <= '0'; SEL_PC <= "000"; SEL_FLAG <= "XXX";

                end case;
```

```
        end process;

end Behavioral;
```

Memória de instruções

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

entity Memoria_de_Instrucoes is
    Port ( Endereco : in  STD_LOGIC_VECTOR (7 downto 0);
           opcode : out  STD_LOGIC_VECTOR (4 downto 0);
           Constante : out  STD_LOGIC_VECTOR (7 downto 0);
           SEL_R : out  STD_LOGIC_VECTOR (5 downto 0));
end Memoria_de_Instrucoes;

architecture Behavioral of Memoria_de_Instrucoes is

begin

        process(Endereco)
        begin

                -- Início da estrutura de seleção de casos dependendo do valor de Endereco
                case Endereco is

                        -- LDP R0
                        when "00000000" => opcode <= "00000"; SEL_R <= "XXX000";
Constante <= "XXXXXXXX";
                        -- JZ R0, 24
                        when "00000001" => opcode <= "10100"; SEL_R <= "XXX000";
Constante <= "00011000";
                        -- ST [5], R0
                        when "00000010" => opcode <= "00100"; SEL_R <= "XXX000";
Constante <= "00000101";
                        -- LD R0, 1
                        when "00000011" => opcode <= "00010"; SEL_R <= "XXX000";
Constante <= "00000001";
                        -- LD R1, -1
                        when "00000100" => opcode <= "00010"; SEL_R <= "XXX001";
Constante <= "11111111";
                        -- LD R2, 20
                        when "00000101" => opcode <= "00010"; SEL_R <= "XXX010";
Constante <= "00010100";
```

```
                    -- LD R3, 6
                    when "00000110" => opcode <= "00010"; SEL_R <= "XXX011";
Constante <= "00000110";
                    -- LD R7, [5]
                    when "00000111" => opcode <= "00011"; SEL_R <= "XXX111";
Constante <= "00000101";
                    -- CMP R7, R2
                    when "00001000" => opcode <= "01101"; SEL_R <= "010111";
Constante <= "XXXXXXXX";
                    -- JGE 19
                    when "00001001" => opcode <= "10001"; SEL_R <= "XXXXXX";
Constante <= "00010011";
                    -- XOR R2, R1
                    when "00001010" => opcode <= "01011"; SEL_R <= "001010";
Constante <= "XXXXXXXX";
                    -- ADD R2, R0
                    when "00001011" => opcode <= "00101"; SEL_R <= "000010";
Constante <= "XXXXXXXX";
                    -- CMP R7, R2
                    when "00001100" => opcode <= "01101"; SEL_R <= "010111";
Constante <= "XXXXXXXX";
                    -- JL 22
                    when "00001101" => opcode <= "01110"; SEL_R <= "XXXXXX";
Constante <= "00010110";
                    -- LD R6, [5]
                    when "00001110" => opcode <= "00011"; SEL_R <= "XXX110";
Constante <= "00000101";
                    -- ADD R7, R6
                    when "00001111" => opcode <= "00101"; SEL_R <= "110111";
Constante <= "XXXXXXXX";
                    -- SUB R3, R0
                    when "00010000" => opcode <= "00110"; SEL_R <= "000011";
Constante <= "XXXXXXXX";
                    -- JN R3, 25
                    when "00010001" => opcode <= "10101"; SEL_R <= "XXX011";
Constante <= "00011001";
                    -- JMP 15
                    when "00010010" => opcode <= "10011"; SEL_R <= "XXXXXX";
Constante <= "00001111";
                    -- LD R4, -15
                    when "00010011" => opcode <= "00010"; SEL_R <= "XXX100";
Constante <= "11110001";
                    -- ADD R7, R4
```

```
                        when "00010100" => opcode <= "00101"; SEL_R <= "100111";
Constante <= "XXXXXXXX";
                        -- JMP 25
                        when "00010101" => opcode <= "10011"; SEL_R <= "XXXXXX";
Constante <= "00011001";
                        -- AND R7, R0
                        when "00010110" => opcode <= "00111"; SEL_R <= "000111";
Constante <= "XXXXXXXX";
                        -- JMP 25
                        when "00010111" => opcode <= "10011"; SEL_R <= "XXXXXX";
Constante <= "00011001";
                        -- LD R7, -1
                        when "00011000" => opcode <= "00010"; SEL_R <= "XXX111";
Constante <= "11111111";
                        -- STP R7
                        when "00011001" => opcode <= "00001"; SEL_R <= "XXX111";
Constante <= "XXXXXXXX";
                        -- JMP 26
                        when "00011010" => opcode <= "10011"; SEL_R <= "XXXXXX";
Constante <= "00011010";
                        --Outros
                        when others => opcode <= "XXXXX"; SEL_R <= "XXXXXX";
Constante <= "XXXXXXXX";

                end case;

        end process;

end Behavioral;
```

Memória de Dados (RAM)

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

entity Memoria_de_Dados is
   Port ( Constante : in  STD_LOGIC_VECTOR (7 downto 0);
          WR : in  STD_LOGIC;
          clk : in  STD_LOGIC;
          Operando1 : in  STD_LOGIC_VECTOR (7 downto 0);
          Dados_M : out  STD_LOGIC_VECTOR (7 downto 0));
end Memoria_de_Dados;
```

```
architecture Behavioral of Memoria_de_Dados is

-- Array de memória com 256 posições, que sºao todos os valores possíveis da Constante
type memoria is array (0 to 255) of STD_LOGIC_VECTOR (7 downto 0);
signal Mem : memoria;

begin

	process(Constante, WR, clk, Operando1)
	begin

		if WR = '1' then

			-- Se está na transição ascendente do relógio
			if rising_edge(clk) then

				-- Guarda os dados do Operando1 na posição de memória indicada pelo sinal de entrada Constante
				Mem(to_integer(unsigned(Constante))) <= Operando1;

			end if;

		else -- WR = '0'

			-- Atribui o valor da posição de memória indicada pelo Sinal Constante à saída Dados_M
			Dados_M <= Mem(to_integer(unsigned(Constante)));

		end if;

	end process;

end Behavioral;
```