Projeto de Circuitos Integrados Digitais - Projeto final

Felipe R. Sobrinho (17/0141764), Luiz Henrique N. de Oliveira (16/0013712) e Kaio Vinicius P. de Carvalho (18/0065416)

I. Introdução

O processador de seis instruções é um dos modelos mais básicos de processador a ser implementado. De fato, processadores comerciais possuem dezenas ou centenas de instruções para a realização de diversas operações, mas esse processador sintetiza bem as instruções fundamentais de um processador, donde as demais são derivadas.

Para implementação de um processador seis instruções, é necessário a implementação de seus blocos fundamentais, que a princípio são dois: bloco(ou unidade) de controle e bloco de operacional. Cada bloco possuí seus componentes com os quais é possível a execução de todos os comandos programados. A figura 1 mostra o esquema básico de um processador de seis instruções.

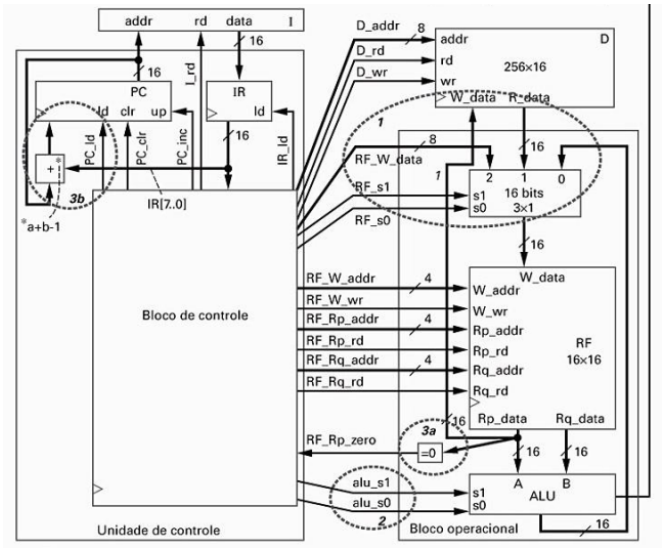

Fig. 1: Processador de seis instruções dividido em dois blocos fundamentais: bloco de controle e bloco operacional.

Uma forma de se descrever a implementação desse processador fisicamente é através da linguagem VHDL ("*VHSIC hardware Description Language*"). O VHDL é uma linguagem utilizada mundialmente para descrição de *hardware*, em conjunto com outras como o Verilog. Após realizada a descrição do *hardware*, é necessário sintetizá-lo, para verificação dos parâmetros reais do circuito, uma vez que para a implementação desse tipo de circuito é utilizado transistores de tecnologia CMOS, que possuí limitações físicas. A sintetização deve ser feita já considerando a tecnologia de fabricação que deverá ser utilizada. Após feita a sintetização do circuito, é possível então realizar a construção do layout no formato de um circuito integrado; processo feito também automaticamente (por conta da alta quantidade de transistores envolvida), mas com o controle de alguns parâmetros.

Tendo todos esses passos em mente, esse projeto visa realizar a implementação de um processador de seis instruções passando por todos os passos para a construção de um circuito integrado. Como software principal para a construção de tal processador, será utilizado o Cadence para a simulação do circuito (Nclaunch) e para a sintetização do circuito (Genos).

II. Descrição do funcionamento dos blocos internos

1) Unidade de controle: A unidade de controle é composta por 4 sub-blocos, são estes: Bloco de controle, somador, PC e IR. Esse formato de blocos funcionais de circuitos permite que seja realizado um *debug* passo-a-passo durante o processo de descrição das estruturas, adicionando confiabilidade e melhor correção das falhas encontradas no processo de teste. Em nível hierárquico, podemos dizer que o bloco de controle se coloca em posição central neste processador de 6 instruções, sendo responsável por comandar todas as variáveis de controle internas que determinam tando os caminhos de dados via MUX quanto as operações lógicas e aritméticas a serem realizadas pela ALU. Não menos importante, os blocos PC e IR componentes dessa unidade de controle são responsáveis pela comunicação correta com a memória de instruções, que ao carregar um arquivo externo .dat deve criar comandos no processador.

- Bloco de controle: Com construção interna simples baseada em uma maquina de estado finita, segue uma ordem pré-definida de estados padrão conforme sugerido na 11, além de interpretar e obedecer a instrução lida na memória, também é responsável por carregar os sinais de controle essenciais para que aconteça o funcionamento esperado do processador.
- Somador: Realiza um deslocamento do contador de programa quando identificado um comando de "saltar se zero". O deslocamento é feito realizando a soma do próprio valor do contador de programa, com o Offset destinado nos últimos 8 bits do programa, e subtraindo uma unidade, para descontar o comando de busca já executado.
- PC: o contador de programa realiza a contabilidade do estado do programa. Ele é responsável por gerenciar os passos dados pelo programa, fazendo saltar as

linhas do código de máquina presente na memória de programa/instrução.
- IR: é um registrador intermediário que guarda o valor do dado da memória de instrução para um determinado valor no contador de programa quando solicitado um comando de busca.

2) Bloco operacional: O bloco operacional é responsável por realizar todas as transformações de dados. Ele é comandado diretamente pelo bloco de controle, e realiza as operações utilizando os dados presentes na memória de dados e no banco de registradores. A memória de dados é utilizada apenas quando solicitado um comando de carregar ou armazenar, enquanto o banco de registradores é utilizado em todos os tipos de instruções. Para realizar as operações aritméticas entre os registradores, é utilizado a unidade lógica aritmética (ULA).

- ALU: Unidade lógica aritmética, responsável por realizar as operações matemáticas solicitadas na instrução, soma, subtração e concatenação.
- Multiplexador 3x1: O bloco multiplexador serve para guiar a informação desejada para o próximo bloco de processamento, neste caso temos um mux 3 entradas que de acordo com as portas de seleção transmite uma das opções de entrada na saída.
- Memória de Dados: a memória de dados funciona como uma memória dinâmica e volátil, ou seja, só guarda os dados quando o processador está em pleno funcionamento. Muito semelhante a memória RAM, possuí uma quantidade razoável de endereços que são utilizados para armazenar o valor presente nos registradores do bloco de registradores.
- Bloco de registradores: também conhecidos como ”Register Files (RF)”, são um conjunto de registradores utilizados pelo processador de maneira direta, diferentemente dos dados armazenados na memória de dados. Eles são indexados e operados diretamente pelo bloco de controle, num formato ditado pelo arquivo de programa.
- Comparador de Igualdade: responsável por levantar uma *flag* 0 caso o ultimo dado processado tenha todos os bits iguais a zero gerando um indicador útil para alguns processos dentro da maquina de estados do controlador, determinando diferentes caminhos possíveis para a mudança de estado.

III. Metodologia

Para construção das descrições de hardware pela linguagem VHDL, será utilizado o Vivado, para que seja feita uma checagem prévia da sintaxe utilizada, poupando tempo de desenvolvimento.

Como já dito antes, será utilizado o NcLaunch para compilar e realizar a simulação do código e verificar o funcionamento de cada módulo, antes de integrá-los em um bloco único. Quando confirmado que todos os blocos estão em pleno funcionamento, será feito a integração deles utilizando a instanciação, e feita uma nova rodada de simulação utilizando o NClaunch.

Concluindo as etapas anteriores, será feito a síntese das descrições de hardware utilizando o Genos. Com isso, teremos uma boa estimativa da área ocupada, dos atrasos e da potência consumida pelo hardware.

Por último, a etapa de implementação será feita, onde será desenhado todo o layout do circuito integrado resultante das analises anteriores. Feita a implementação, será feita uma nova rodada de simulações para observar o comportamento do circuito. O intuito de se realizar tantas verificações é para que se possa realizar um comparativo entre as simulações de cada uma das etapas.

IV. Resultados

A. Simulação

1) Unidade de Controle: A simulação do bloco de controle junto com a memória de instrução foi feito utilizando um testbench simples que realiza apenas o estímulo do *clock*. O programa responsável pelo teste está no arquivo *hexFile.mem*, que contém uma instrução de cada tipo, para observar a máquina de estados. A captura de tela referente a unidade de controle pode ser vista na figura 2.

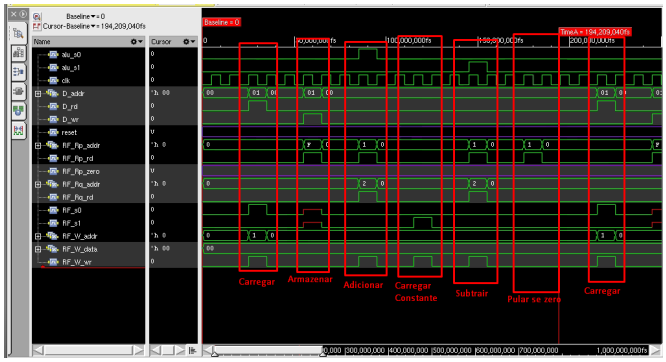

Fig. 2: Teste da unidade de controle.

O código utilizado para realizar cada uma das operações pode ser visto na tabela I.

TABLE I: Tabela do programa utilizado para realizar o teste.

Código de máquina (HEX)	Valor em binário	Operação	Descrição
0x0101	0b0000000 100000001	**Ra[0] = D[1]**	**Carrega o valor de endereço 1 da memória D no registrador Ra[0]**
0x1F01	0b00011111 00000001	D[F] = Ra[1]	Armazena o valor de Ra na memória D
0x2012	0b00100000 00010010	Ra[0] = Rb[1] + Rc[2]	Realiza a soma de Rb[1] e Rb[2] e armazena em Ra[0]
0x3012	0b00110000 00010010	Ra[0] = 0b10010	Carrega a constante 0b10010 no registrador Ra[0]
0x4012	0b01000000 00010010	Ra[0] = Rb[1] - Rc[2]	Subtraí Rb[1] e Rc[2] e armazena em Ra[0]
0x5105	0b01010001 00000101	PC=PC+5 se Ra[1] == 0	Realiza o offset do contador de programa se Ra[1] == 0

Com isso, foi possível verificar que para cada comando, a unidade de controle consegue enviar corretamente os sinais para o bloco operacional, e também consegue realizar o acesso à memória de instrução.

2) Bloco operacional: Para simular o bloco operacional, junto com memoria D, foi criando um tesbench (Apêndice V-H), que simula as operações possíveis no do registrado. Primeiro é executado o carregamento de uma constante no registrador 2 do banco de registradores, depois é armazenada

essa constante na memoria D na posição 17 e em seguida carregada, essa mesma posição da memória, em no registrador 2. Podemos ver o resultados da saída desse bloco na figura 3.

Fig. 3: Formas de onda da simulação do bloco operacional e da memoria D

Como descrito anteriormente, o primeira instrução simulada é o carregamento de uma constante no registrador, podemos na figura 4 os valores da saída, onde vale destacar as seguintes entradas RF_W_addr que identifica o endereço do registrador que vai ser armazenado o valor da constante, RF_W_data que o valor que vai ser armazenado no registrador de endereço RF_W_addr, RF_W_Wr que é o sinal que aciona o modo de escrita do banco de registradores e as entrada RF_s1 e RF_s0 que selecionam a saída 3 do multiplexador de 3 entradas.

Fig. 4: Simulação da instrução CONSTANTE (MOV Ra,C)

A segunda instrução simulada é o armazenamento de um registrador na memoria D, podemos ver este passo na figura 5, observe que as entradas D_wr é acionada e a entrada D_addr tem valor 0x11 que o endereço da memoria D.

Fig. 5: Simulação da instrução ARMAZENAR (MOV d, Ra)

A terceira instrução simulada é carregar o valor armazenado na memoria D em um registrador, mostrado na figura 6.

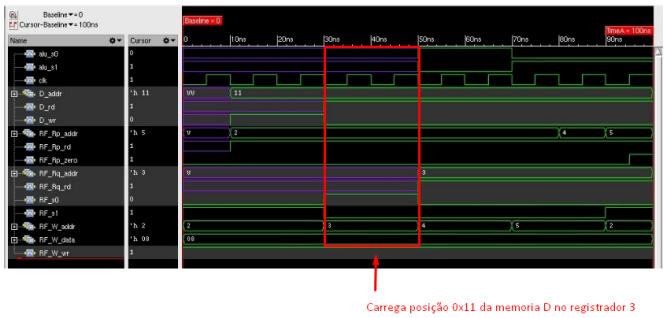

Fig. 6: Simulação da instrução CARREGAR (MOV Ra, d)

Feito os carregamentos vamos a operação de soma e subtração, mostrados na figura 7 e figura 8, respectivamente. Como o registrador 2 tem valor 8 e o registrador 3 também tem valor 8, esses são os registradores que foram carregados nas instruções anteriores, a soma vai ser igual a 16 e a subtração vai ser igual a 0, logo no registrador 4 que é armazenado o resultado da soma tera valor 16 e o registrador 5 que tem o resultado da subtração tera valor 0.

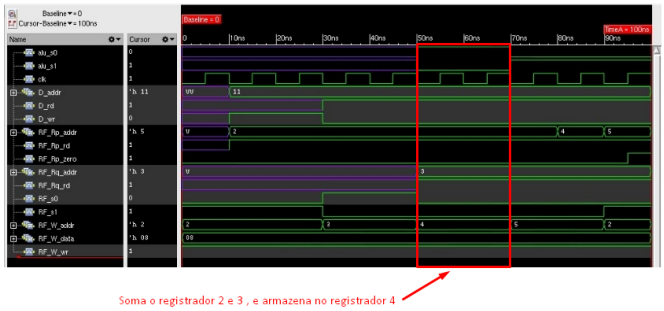

Fig. 7: Simulação da instrução SOMAR (ADD Ra, Rb, Rc)

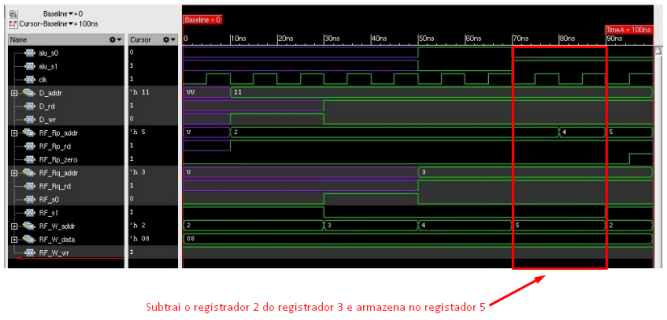

Fig. 8: Simulação da instrução SUBTRAIR (SUB Ra, Rb, Rc)

As ultimas instruções são a de ”saltar-se”, que pula para próxima instrução se o valor do registrador for igual a 0, podemos ver então que na figura 9 como o registrador 4 tem valor igual 16 o sinal de saída RF_Rp_zero tem sinal lógico

igual 0, entretanto na figura 10 como estamos comparando o valor do registrador 5 que tem valor 0 a saída RF_Rp_zero é igual 1, ou seja, neste caso a próxima instrução seria pulada.

Fig. 9: Simulação da instrução ”SALTAR-SE” para o registrador 4 que tem valor 16 (JUMPZ Ra,Offset)

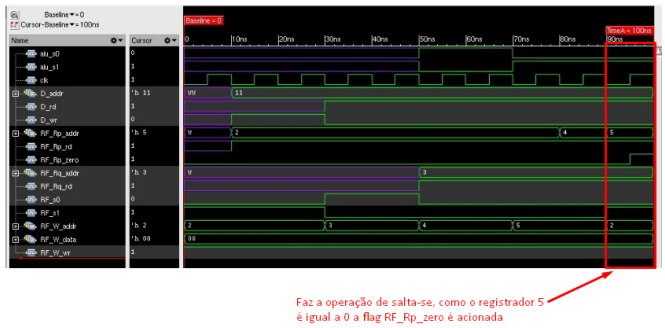

Fig. 10: Simulação da instrução ”SALTAR-SE” para o registrador 5 que tem valor 0 (JUMPZ Ra,Offset)

V. Apêndice

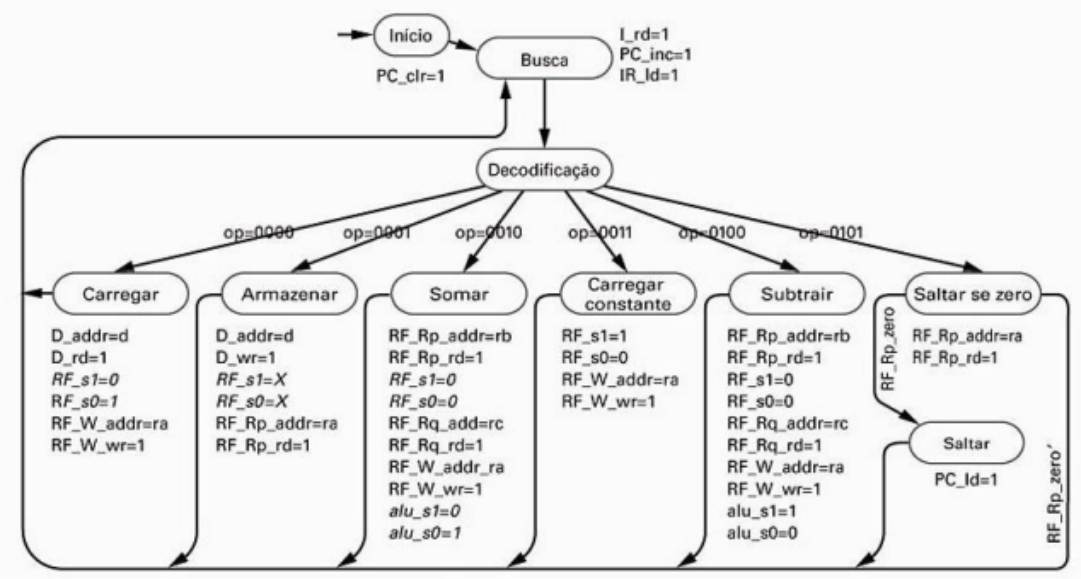

Fig. 11: Diagrama de estados da máquina

A. Memoria D

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

entity memD is
    Port ( clk : in STD_LOGIC;
           addr : in STD_LOGIC_VECTOR (7 downto 0);
           D_rd : in STD_LOGIC;
           D_wr : in STD_LOGIC;
           W_data : in STD_LOGIC_VECTOR (15 downto 0);
           R_data : out STD_LOGIC_VECTOR (15 downto 0));
end memD;



architecture behavorial of memD is

    type array_mem is array ((2**8 - 1) downto 0) of STD_LOGIC_VECTOR (15 DOWNTO 0);
    signal memory: array_mem;


begin

    process(clk, D_wr, D_rd)
    begin
        if rising_edge(clk) then
            if D_wr = '1' then
                memory(CONV_INTEGER(addr)) <= W_data;
            end if;
            if D_rd = '1' then
                R_data<=memory(CONV_INTEGER(addr));
            end if;
        end if;
```

```
    end process;

end behavorial;
```

B. Multiplexador de 3 entradas

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

entity MUX3x1 is
    Port ( in_0 : in STD_LOGIC_VECTOR (15 downto 0);
           in_1 : in STD_LOGIC_VECTOR (15 downto 0);
           in_2 : in STD_LOGIC_VECTOR (7 downto 0);
           s0 : in STD_LOGIC;
           s1 : in STD_LOGIC;
           mux3x1_out : out STD_LOGIC_VECTOR (15 downto 0));
end MUX3x1;

architecture Behavioral of MUX3x1 is

    signal sel: std_logic_vector(1 downto 0);
    signal in_2_8b: std_logic_vector(7 downto 0);
    signal in_2_16b: std_logic_vector(15 downto 0);

begin

    -- extrai bit mais significativo da entrada 2
    with in_2(7) select
    in_2_8b <= (others =>'1') when '1',
               (others =>'0') when others;

    -- concatena bit mais significativo da entrada 2
    in_2_16b <= in_2_8b & in_2;
    sel <= s1 & s0;

    -- seleciona a saida do mux
    with sel select
    mux3x1_out <=  in_0 when "00",
                   in_1 when "01",
                   in_2_16b when others;

end Behavioral;
```

C. Banco de registradores

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

entity RegisterFile is
    Port ( clk : in STD_LOGIC;
           W_RF_data : in STD_LOGIC_VECTOR (15 downto 0);
           W_addr : in STD_LOGIC_VECTOR (3 downto 0);
           W_wr : in STD_LOGIC;
           Rp_addr : in STD_LOGIC_VECTOR (3 downto 0);
           Rp_rd : in STD_LOGIC;
```

```
            Rq_addr : in STD_LOGIC_VECTOR (3 downto 0);
            Rq_rd : in STD_LOGIC;
            Rp_data : out STD_LOGIC_VECTOR (15 downto 0);
            Rq_data : out STD_LOGIC_VECTOR (15 downto 0));
end RegisterFile;

architecture behavorial of RegisterFile is

    type registerFile is array(15 downto 0) of STD_LOGIC_VECTOR(15 downto 0);
    signal RF : registerFile; -- Memoria do banco de registradores


begin
    -- processo para escrita no banco de resgistradores
    mem_wr: process(clk, W_wr)
    begin
        if rising_edge(clk) then
        if(W_wr = '1') then
            RF(CONV_INTEGER(W_addr)) <= W_RF_data;
        end if;
        end if;
    end process;

    -- processo para leitura no banco de registradores
    mem_up: process(clk, Rp_rd)
    begin
        if rising_edge(clk) then
        if(Rp_rd = '1') then
                Rp_data <= RF(CONV_INTEGER(Rp_addr));
        end if;
        end if;
    end process;
    process(clk,Rq_rd)
    begin
        if rising_edge(clk) then
        if(Rq_rd = '1') then
            Rq_data <= RF(CONV_INTEGER(Rq_addr));
        end if;
        end if;
    end process;


end behavorial;
```

D. Unidade logica e aritmética

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

entity ALU is
    Port ( A : in STD_LOGIC_VECTOR (15 downto 0);
           B : in STD_LOGIC_VECTOR (15 downto 0);
           alu_s1 : in STD_LOGIC;
           alu_s0 : in STD_LOGIC;
           alu_out : out STD_LOGIC_VECTOR (15 downto 0));
end ALU;
```

```
architecture Behavioral of ALU is

    signal alu_s : std_logic_vector(1 downto 0);
    signal sum , sub : std_logic_vector(15 downto 0);

begin

    sum <= std_logic_vector(signed(A) + signed(B)); -- Faz operacao de soma
    sub <= std_logic_vector(signed(A) - signed(B)); -- Faz operacao de subtracao
    alu_s <= alu_s1 & alu_s0; -- Concatena bits de selecao

    -- Seleciona saida dependendo da operacao
    with alu_s select
        alu_out <=  A when "00",
                    sum when "01",
                    sub when others;

end Behavioral;
```

E. Flag zero

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity flag0 is
        port(   a: in STD_LOGIC_VECTOR(15 downto 0);
                        y: out STD_LOGIC);
end flag0;

architecture synth of flag0 is
        signal i: integer;
        signal x: STD_LOGIC_VECTOR(15 downto 1);
begin -- N-bit AND of inverted inputs

        -- Verifica se o valor de a eh igual a 0x0000
        with a select
        y <= '1' when "0000000000000000",
             '0' when others;
end synth;
```

F. Bloco operacional - Top level

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

entity bloco_operacional is
    Port ( clk: in STD_LOGIC;
           R_DR_data: in STD_LOGIC_VECTOR(15 downto 0);
           RF_W_data : in STD_LOGIC_VECTOR (7 downto 0);
           RF_s1 : in STD_LOGIC;
           RF_s0 : in STD_LOGIC;
           RF_W_addr : in STD_LOGIC_VECTOR (3 downto 0);
           RF_W_wr : in STD_LOGIC;
           RF_Rp_addr : in STD_LOGIC_VECTOR (3 downto 0);
           RF_Rp_rd : in STD_LOGIC;
```

```
            RF_Rq_addr : in STD_LOGIC_VECTOR (3 downto 0);
            RF_Rq_rd : in STD_LOGIC;
            alu_s1 : in STD_LOGIC;
            alu_s0 : in STD_LOGIC;
            W_DR_data: out STD_LOGIC_VECTOR (15 downto 0);
            RF_Rp_zero : out STD_LOGIC);
end bloco_operacional;

architecture Behavioral of bloco_operacional is

    component MUX3x1 is
    Port ( in_0 : in STD_LOGIC_VECTOR (15 downto 0);
           in_1 : in STD_LOGIC_VECTOR (15 downto 0);
           in_2 : in STD_LOGIC_VECTOR (7 downto 0);
           s0 : in STD_LOGIC;
           s1 : in STD_LOGIC;
           mux3x1_out : out STD_LOGIC_VECTOR (15 downto 0));
    end component;

    component RegisterFile is
    Port ( clk : in STD_LOGIC;
           W_RF_data : in STD_LOGIC_VECTOR (15 downto 0);
           W_addr : in STD_LOGIC_VECTOR (3 downto 0);
           W_wr : in STD_LOGIC;
           Rp_addr : in STD_LOGIC_VECTOR (3 downto 0);
           Rp_rd : in STD_LOGIC;
           Rq_addr : in STD_LOGIC_VECTOR (3 downto 0);
           Rq_rd : in STD_LOGIC;
           Rp_data : out STD_LOGIC_VECTOR (15 downto 0);
           Rq_data : out STD_LOGIC_VECTOR (15 downto 0));
    end component;

    component ALU is
    Port ( A : in STD_LOGIC_VECTOR (15 downto 0);
           B : in STD_LOGIC_VECTOR (15 downto 0);
           alu_s1 : in STD_LOGIC;
           alu_s0 : in STD_LOGIC;
           alu_out : out STD_LOGIC_VECTOR (15 downto 0));
    end component;

    component flag0 is
        port(   a: in STD_LOGIC_VECTOR(15 downto 0);
                    y: out STD_LOGIC);
    end component;

    signal s_alu_out, s_mux3x1, s_Rp_data, s_Rq_data : STD_LOGIC_VECTOR (15 downto 0);
begin

    -- Multiplexador de 3 entradas
    MUX_3x1: MUX3x1 port map
         ( in_0 => s_alu_out,
           in_1 => R_DR_data,
           in_2 => RF_W_data,
           s0 => RF_s0,
           s1 => RF_s1,
           mux3x1_out => s_mux3x1);

    -- Banco de registradores
```

```
    RF: RegisterFile port map
                ( clk => clk,
                  W_RF_data => s_mux3x1,
                  W_addr => RF_W_addr,
                  W_wr => RF_W_wr,
                  Rp_addr => RF_Rp_addr,
                  Rp_rd => RF_Rp_rd,
                  Rq_addr => RF_Rq_addr,
                  Rq_rd => RF_Rq_rd,
                  Rp_data => s_Rp_data,
                  Rq_data => s_Rq_data);


    -- Unidade de operacao logica e aritmetica
     myalu: ALU port map
          ( A => s_Rp_data,
            B => s_Rq_data,
            alu_s1 => alu_s1,
            alu_s0 => alu_s0,
            alu_out => s_alu_out);

    -- Identifica se a entrada eh igual a 0
    zerp_flag: flag0 port map
            ( a => s_Rp_data,
                        y => RF_Rp_zero);

    -- Entrada da memoria D, na operacao de escrita de constante
    W_DR_data <= s_Rp_data;
end Behavioral;
```

G. Bloco operacional e Memoria D

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

entity top_level_operacional is
  Port (D_addr : in STD_LOGIC_VECTOR (7 downto 0);
        D_rd : in STD_LOGIC;
        D_wr : in STD_LOGIC;
        clk: in STD_LOGIC;
        RF_W_data : in STD_LOGIC_VECTOR (7 downto 0);
        RF_s1 : in STD_LOGIC;
        RF_s0 : in STD_LOGIC;
        RF_W_addr : in STD_LOGIC_VECTOR (3 downto 0);
        RF_W_wr : in STD_LOGIC;
        RF_Rp_addr : in STD_LOGIC_VECTOR (3 downto 0);
        RF_Rp_rd : in STD_LOGIC;
        RF_Rq_addr : in STD_LOGIC_VECTOR (3 downto 0);
        RF_Rq_rd : in STD_LOGIC;
        alu_s1 : in STD_LOGIC;
        alu_s0 : in STD_LOGIC;
        RF_Rp_zero : out STD_LOGIC);
end top_level_operacional;

architecture Behavioral of top_level_operacional is

    component memD is
```

```
    Port ( clk : in STD_LOGIC;
           addr : in STD_LOGIC_VECTOR (7 downto 0);
           D_rd : in STD_LOGIC;
           D_wr : in STD_LOGIC;
           W_data : in STD_LOGIC_VECTOR (15 downto 0);
           R_data : out STD_LOGIC_VECTOR (15 downto 0));
    end component;

    component bloco_operacional is
    Port ( clk: in STD_LOGIC;
           R_DR_data: in STD_LOGIC_VECTOR(15 downto 0);
           RF_W_data : in STD_LOGIC_VECTOR (7 downto 0);
           RF_s1 : in STD_LOGIC;
           RF_s0 : in STD_LOGIC;
           RF_W_addr : in STD_LOGIC_VECTOR (3 downto 0);
           RF_W_wr : in STD_LOGIC;
           RF_Rp_addr : in STD_LOGIC_VECTOR (3 downto 0);
           RF_Rp_rd : in STD_LOGIC;
           RF_Rq_addr : in STD_LOGIC_VECTOR (3 downto 0);
           RF_Rq_rd : in STD_LOGIC;
           alu_s1 : in STD_LOGIC;
           alu_s0 : in STD_LOGIC;
           W_DR_data: out STD_LOGIC_VECTOR (15 downto 0);
           RF_Rp_zero : out STD_LOGIC);
    end component;

    signal s_W_data, s_R_data: std_logic_vector(15 downto 0);

begin
    -- Memoria de dados
    memoryD: memD port map
          ( clk => clk,
            addr => D_addr,
            D_rd => D_rd,
            D_wr => D_wr,
            W_data => s_W_data,
            R_data => s_R_data);

    -- Bloco operacional com a ALU, RF, flag0 e o MUX3x1
    operating_block: bloco_operacional port map
          ( clk => clk,
            R_DR_data => s_R_data,
            RF_W_data => RF_W_data,
            RF_s1 => RF_s1,
            RF_s0 => RF_s0,
            RF_W_addr => RF_W_addr,
            RF_W_wr => RF_W_wr,
            RF_Rp_addr => RF_Rp_addr,
            RF_Rp_rd => RF_Rp_rd,
            RF_Rq_addr => RF_Rq_addr,
            RF_Rq_rd => RF_Rq_rd,
            alu_s1 => alu_s1,
            alu_s0 => alu_s0,
            W_DR_data => s_W_data,
            RF_Rp_zero => RF_Rp_zero);

end Behavioral;
```

H. Bloco operacional e Memoria D

```
-- Testebench do bloco operacional:
-- Simula as 6 operacoes do processador:
-- 1. Carrega uma constante no registrador 2
-- 2. Armazenar na posicao 17 da memoria D
-- 3. Carrega da no registrador 3 a posicao 17 da memoria D
-- 4. Faz operacao de soma com o registrador 2 e 3 e armazena no registrador 4
-- 5. Faz operacao de subtracao com o registrador 2 e 3 e
-- armazena no registrador 5
-- 7. Salta se registrador 4 for 0, o que eh esperado que nao salte
-- 8. Salta se registrador 5 for 0, e o sinal RF_Rp_zero vai para nivel logico
-- alto

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity tb_top_level_operacional is
--  Port ( );
end tb_top_level_operacional;

architecture Behavioral of tb_top_level_operacional is

  component top_level_operacional is
  Port (D_addr : in STD_LOGIC_VECTOR (7 downto 0);
        D_rd : in STD_LOGIC;
        D_wr : in STD_LOGIC;
        clk: in STD_LOGIC;
        RF_W_data : in STD_LOGIC_VECTOR (7 downto 0);
        RF_s1 : in STD_LOGIC;
        RF_s0 : in STD_LOGIC;
        RF_W_addr : in STD_LOGIC_VECTOR (3 downto 0);
        RF_W_wr : in STD_LOGIC;
        RF_Rp_addr : in STD_LOGIC_VECTOR (3 downto 0);
        RF_Rp_rd : in STD_LOGIC;
        RF_Rq_addr : in STD_LOGIC_VECTOR (3 downto 0);
        RF_Rq_rd : in STD_LOGIC;
        alu_s1 : in STD_LOGIC;
        alu_s0 : in STD_LOGIC;
        RF_Rp_zero : out STD_LOGIC);
  end component;

  signal D_addr :  STD_LOGIC_VECTOR (7 downto 0);
  signal D_rd :  STD_LOGIC;
  signal D_wr :  STD_LOGIC;
  signal clk:  STD_LOGIC := '0';
  signal RF_W_data :  STD_LOGIC_VECTOR (7 downto 0);
  signal RF_s1 :  STD_LOGIC;
  signal RF_s0 :  STD_LOGIC;
  signal RF_W_addr :  STD_LOGIC_VECTOR (3 downto 0);
  signal RF_W_wr :  STD_LOGIC;
  signal RF_Rp_addr :  STD_LOGIC_VECTOR (3 downto 0);
  signal RF_Rp_rd :  STD_LOGIC;
  signal RF_Rq_addr :  STD_LOGIC_VECTOR (3 downto 0);
  signal RF_Rq_rd :  STD_LOGIC;
  signal alu_s1 :  STD_LOGIC;
  signal alu_s0 :  STD_LOGIC;
  signal RF_Rp_zero :  STD_LOGIC;
```

```
  constant period: time := 10 ns;

begin

    utt:top_level_operacional port map
       (D_addr => D_addr,
        D_rd => D_rd,
        D_wr => D_wr,
        clk => clk,
        RF_W_data => RF_W_data,
        RF_s1 => RF_s1,
        RF_s0 => RF_s0,
        RF_W_addr => RF_W_addr,
        RF_W_wr => RF_W_wr,
        RF_Rp_addr => RF_Rp_addr,
        RF_Rp_rd => RF_Rp_rd,
        RF_Rq_addr => RF_Rq_addr,
        RF_Rq_rd => RF_Rq_rd,
        alu_s1 => alu_s1,
        alu_s0 => alu_s0,
        RF_Rp_zero => RF_Rp_zero);

   clk <= not clk after period/2;
   stimulus: process
               begin
               -- Carregar constante
               RF_W_data <= "00001000";
               RF_s1 <= '1';
               RF_s0 <= '0';
               RF_W_addr <= "0010";
               RF_W_wr <= '1';
               wait for period;
               -- Armazenar
               D_addr <= "00010001";
               D_wr <= '1';
               RF_Rp_addr <= "0010";
               RF_Rp_rd <= '1';
               wait for 2*period;
               -- Carregar
               D_wr <= '0';
               D_addr <= "00010001";
               D_rd <= '1';
               RF_s1 <= '0';
               RF_s0 <= '1';
               RF_W_addr <= "0011";
               RF_W_wr <= '1';
               wait for  2*period;
               -- Somar
               RF_W_wr <= '1';
               RF_s1 <= '0';
               RF_s0 <='0';
               RF_W_addr <= "0100";
               RF_Rp_rd <='1';
               RF_Rq_rd <='1';
               RF_Rp_addr <= "0010";
               RF_Rq_addr <= "0011";
               alu_s1 <= '0';
               alu_s0 <= '1';
```

```
                wait for 2*period;
                -- Subtrair
                RF_W_wr <= '1';
                RF_s1 <= '0';
                RF_s0 <='0';
                RF_W_addr <= "0101";
                RF_Rp_rd <='1';
                RF_Rq_rd <='1';
                RF_Rp_addr <= "0010";
                RF_Rq_addr <= "0011";
                alu_s1 <= '1';
                alu_s0 <= '0';
                wait for period;
                -- Saltar se zero
                RF_Rp_addr <= "0100";
                RF_Rp_rd <= '1';
                wait for period;
                RF_Rp_addr <= "0101";
                RF_Rp_rd <= '1';
            end process;
end Behavioral;
```

I. TestBench da memória de instruções e da unidade de controle

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity topUnidadeControle_tb is
end topUnidadeControle_tb;

architecture Behavioral of topUnidadeControle_tb is

component top_level_controle is
  Port (clk, reset: in STD_LOGIC;
            RF_Rp_zero: in STD_LOGIC;
            D_addr, RF_W_data: out STD_LOGIC_VECTOR(7 downto 0);
            RF_W_addr, RF_Rp_addr, RF_Rq_addr: out STD_LOGIC_VECTOR(3 downto 0);
            D_rd, D_wr, RF_s1, RF_s0, RF_W_wr, RF_Rp_rd, RF_Rq_rd,

            alu_s1, alu_s0 : out STD_LOGIC);
end component;

    signal clk, reset: STD_LOGIC;
    signal RF_Rp_zero: STD_LOGIC;
    signal D_addr, RF_W_data: STD_LOGIC_VECTOR(7 downto 0);
    signal RF_W_addr, RF_Rp_addr, RF_Rq_addr: STD_LOGIC_VECTOR(3 downto 0);
    signal D_rd, D_wr, RF_s1, RF_s0, RF_W_wr, RF_Rp_rd, RF_Rq_rd,

    alu_s1, alu_s0 : STD_LOGIC;

begin

    control_unit: top_level_controle port map (
        clk, reset,
            RF_Rp_zero,
            D_addr, RF_W_data,
            RF_W_addr, RF_Rp_addr, RF_Rq_addr,
            D_rd, D_wr, RF_s1, RF_s0, RF_W_wr, RF_Rp_rd, RF_Rq_rd, alu_s1, alu_s0
```

```
    );

    -- test bench apenas para o estimulo do clock.
    clk_stim: process
    begin
        clk <= '0';
        wait for 5 ns;
        clk <= '1';
        wait for 5 ns;
    end process;

end Behavioral;
```

J. Top Level da unidade de controle e memória I

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

entity top_level_controle is
  Port (clk, reset: in STD_LOGIC;
            RF_Rp_zero: in STD_LOGIC;
            D_addr, RF_W_data: out STD_LOGIC_VECTOR(7 downto 0);
            RF_W_addr, RF_Rp_addr, RF_Rq_addr: out STD_LOGIC_VECTOR(3 downto 0);
            D_rd, D_wr, RF_s1, RF_s0, RF_W_wr, RF_Rp_rd, RF_Rq_rd,

            alu_s1, alu_s0 : out STD_LOGIC);
end top_level_controle;

architecture Behavioral of top_level_controle is

        -- Unidade de controle completa excluindo a memória I.
    component unidade_de_controle is -- top-level design for testing
        port(   clk, reset: in STD_LOGIC;
                        data: in STD_LOGIC_VECTOR(15 downto 0);
                        RF_Rp_zero: in STD_LOGIC;
                        PC: out STD_LOGIC_VECTOR(15 downto 0);
                        I_rd: out STD_LOGIC;
                        D_addr, RF_W_data: out STD_LOGIC_VECTOR(7 downto 0);
                        RF_W_addr, RF_Rp_addr,
                        RF_Rq_addr: out STD_LOGIC_VECTOR(3 downto 0);
                        D_rd, D_wr, RF_s1, RF_s0, RF_W_wr, RF_Rp_rd, RF_Rq_rd,
                        alu_s1, alu_s0 : out STD_LOGIC);
    end component;

    --Memória de Instruçao.
    component memoryI is
    Port ( addr : in STD_LOGIC_VECTOR (15 downto 0);
           rd : in STD_LOGIC;
           data : out STD_LOGIC_VECTOR (15 downto 0));
    end component;

    signal s_data_I, s_PC: std_logic_vector(15 downto 0);
    signal s_I_rd: std_logic;
begin
        -- Mapeamento da unidade de controle.
    unit_control:unidade_de_controle port map
        (       clk => clk,
```

```
            reset => reset,
                        data => s_data_I,
                        RF_Rp_zero => RF_Rp_zero,
                        PC => s_PC,
                        I_rd => s_I_rd,
                        D_addr => D_addr,
                        RF_W_data => RF_W_data,
                        RF_W_addr => RF_W_addr,
                        RF_Rp_addr => RF_Rp_addr,
                        RF_Rq_addr => RF_Rq_addr,
                        D_rd => D_rd,
                        D_wr => D_wr,
                        RF_s1 => RF_s1,
                        RF_s0 => RF_s0,
                        RF_W_wr => RF_W_wr,
                        RF_Rp_rd => RF_Rp_rd,
                        RF_Rq_rd => RF_Rq_rd,
                        alu_s1 => alu_s1,
                        alu_s0 => alu_s0);

        -- Mapeamento da memória I.
    memoria_I: memoryI port map
         ( addr => s_PC,
           rd => s_I_rd,
           data => s_data_I);


end Behavioral;
```

K. Top Level da unidade de controle

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

entity unidade_de_controle is
        port(   clk, reset: in STD_LOGIC;
                        data: in STD_LOGIC_VECTOR(15 downto 0);
                        RF_Rp_zero: in STD_LOGIC;
                        PC: out STD_LOGIC_VECTOR(15 downto 0);
                        I_rd: out STD_LOGIC;
                        D_addr, RF_W_data: out STD_LOGIC_VECTOR(7 downto 0);
                        RF_W_addr, RF_Rp_addr,
                        RF_Rq_addr: out STD_LOGIC_VECTOR(3 downto 0);
                        D_rd, D_wr, RF_s1, RF_s0, RF_W_wr,
                        RF_Rp_rd, RF_Rq_rd, alu_s1, alu_s0 : out STD_LOGIC);
end;

architecture UC of unidade_de_controle is

component bloco_controle is -- entradas e saídas do bloco de controle
        port(           clk, reset: in STD_LOGIC;
                                IR: in STD_LOGIC_VECTOR(15 downto 0);
                                RF_Rp_zero: in STD_LOGIC;
                                D_rd, D_wr, RF_s1, RF_s0: out STD_LOGIC;
                                D_addr, RF_W_data: out STD_LOGIC_VECTOR(7 downto 0);
                                PC_id, PC_clr, PC_inc, I_rd, IR_id: out STD_LOGIC;
```

```
                                    RF_W_wr, RF_Rp_rd, RF_Rq_rd: out STD_LOGIC;
                                    alu_s1, alu_s0: out STD_LOGIC;
                                    RF_W_addr, RF_Rp_addr,
                                    RF_Rq_addr: out STD_LOGIC_VECTOR(3 downto 0));
end component;

--Instanciaçao do contador de programa.
component programCounter is
    Port ( clk : in STD_LOGIC;
           PC_Id : in STD_LOGIC;
           PC_clr : in STD_LOGIC;
           PC_inc : in STD_LOGIC;
           PC_in : in STD_LOGIC_VECTOR (15 downto 0);
           PC : out STD_LOGIC_VECTOR (15 downto 0));
end component;

--Instanciaçao do Registrador de instruç es .
component InstructionRegister is
    Port ( clk : in STD_LOGIC;
           IR_Id : in STD_LOGIC;
           data_in : in STD_LOGIC_VECTOR (15 downto 0);
           data_out : out STD_LOGIC_VECTOR (15 downto 0));
end component;

-- Instanciaçao do somador.
component somador is
        port(   a: in STD_LOGIC_VECTOR(15 downto 0);
                    b: in STD_LOGIC_VECTOR(7 downto 0);
                    s: out STD_LOGIC_VECTOR(15 downto 0));
end component;

signal s_PC, s_IR, ss : STD_LOGIC_VECTOR(15 downto 0);
signal sPC_Id, sPC_clr, sPC_inc, sIR_id: STD_LOGIC;
begin
        --Mapeamento do bloco de controle.
 U0: bloco_controle port map (clk => clk,
          reset => reset,
          IR => s_IR,
          RF_Rp_zero => RF_Rp_zero,
          D_rd => D_rd,
          D_wr => D_wr,
          RF_s1 => RF_s1,
          RF_s0 => RF_s0,
          D_addr => D_addr,
          RF_W_data => RF_W_data,
          PC_id => sPC_Id,
          PC_clr => sPC_clr,
          PC_inc => sPC_inc,
          I_rd => I_rd,
          IR_id => sIR_id,
          RF_W_wr => RF_W_wr,
          RF_Rp_rd => RF_Rp_rd,
          RF_Rq_rd => RF_Rq_rd,
          alu_s1 => alu_s1,
          alu_s0 => alu_s0,
          RF_W_addr => RF_W_addr,
          RF_Rp_addr => RF_Rp_addr,
          RF_Rq_addr => RF_Rq_addr);
```

```
 --Mapeamento do contador de programa.
 U1: programCounter port map (  clk => clk,
            PC_Id => sPC_Id,
            PC_clr => sPC_clr,
            PC_inc => sPC_inc,
            PC_in => ss,
            PC => s_PC);
 --Mapeamento do Registrador de Instruçoes
 U2: InstructionRegister port map(clk => clk,
          IR_Id => sIR_id,
      data_in => data,
      data_out => s_IR);
 --Mapeamento do somador.
 U3: somador port map(a => s_PC,
          b => s_IR(7 DOWNTO 0),
          s => ss);

PC <= s_PC;

end;
```

L. Somador para offset do contador de programa

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

entity somador is
        port(   a: in STD_LOGIC_VECTOR(15 downto 0);
                b: in STD_LOGIC_VECTOR(7 downto 0);
                s: out STD_LOGIC_VECTOR(15 downto 0));
end somador;

architecture asynth of somador is
    signal sum : unsigned(15 downto 0);
begin
        --Realiza a soma utilizando dois unsigned.
        sum<=unsigned(a)+unsigned(b);
        --Atribuí como sendo um standard logic vector.
        s <= STD_LOGIC_VECTOR(sum - X"0001");
end;
```

M. Contador de programa

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

entity programCounter is
    Port ( clk : in STD_LOGIC;
           PC_Id : in STD_LOGIC;
           PC_clr : in STD_LOGIC;
           PC_inc : in STD_LOGIC;
           PC_in : in STD_LOGIC_VECTOR (15 downto 0);
           PC : out STD_LOGIC_VECTOR (15 downto 0));
```

```
end programCounter;

architecture behavorial of programCounter is

    signal PC_val: STD_LOGIC_VECTOR (15 downto 0) := X"0000";

begin

    process(clk, PC_clr)
    begin
        -- Reseta o valor do clock se solicitado
        if rising_edge(clk) then
            if PC_clr = '1' then
                PC_val <= X"0000";
            else
                --Se PC_Id, ent o o PC é carregado com o valor do somador.
                if PC_Id = '1' then
                    PC_val <= PC_in;
                    -- Se PC_inc, o valor é incrementado.
                elsif PC_inc = '1' then
                    PC_val <= PC_val + X"0001";
                end if;
            end if;
        end if;
    end process;

    --Saída atribuída assincronamente.
    PC <= PC_val;

end behavorial;
```

N. Memória de Instruções

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;
use STD.textio.ALL; --Biblioteca para manipulacao de arquivos de texto.

entity memoryI is
    Port ( addr : in STD_LOGIC_VECTOR (15 downto 0);
           rd : in STD_LOGIC;
           data : out STD_LOGIC_VECTOR (15 downto 0));
end memoryI;

architecture Behavioral of memoryI is

        --Declaraçao do formato de memória que será utilizado.
    type memI_type is array (2**16 - 1 downto 0) of STD_LOGIC_VECTOR (15 downto 0);
    signal memI : memI_type;


begin

    mem_init:process
        file hex_File: text open read_mode is "hexFile.mem";
        variable index: integer;
```

```
        variable data_line : line;
        variable hex_val: integer;
        variable char: character;

        begin
        --Memória é inicializada com todos os valores em 0
        for i in 0 to 65535 loop
            memI(i) <= X"0000";
        end loop;

        index:=0;
        -- Carrega todos os valores do arquivo na memória I.
        while not endfile(hex_file) loop
            readline(hex_file, data_line);
            hex_val:=0;
            -- A cada loop,uma localizaç o  da memória é preenchida com uma linha
            for k in 0 to 3 loop
            --Realiza a convers o utilizando a tabela ASCII.
                read(data_line, char);
                if char <= '9' and char >= '0' then
                    hex_val := hex_val*(16) + character'pos(char) - character'pos('0');
                elsif char >= 'a' and char <= 'f' then
                    hex_val := hex_val*(16) + character'pos(char) - character'pos('a')+10;
                elsif char>='A' and char <= 'F' then
                    hex_val := hex_val*(16) + character'pos(char) - character'pos('A')+10;
                end if;
        end loop;
            --Armazena o valor na memória como tipo STD_LOGIC_VECTOR.
            memI(index) <= STD_LOGIC_VECTOR(TO_UNSIGNED(hex_val, data'length));

            index := index + 1;
        end loop;
        wait;
        end process;
       --Memória assíncrona, entao só irá mudar o dado quando rd='1' e quando
       -- mudar o valor do endereço.
     data_out:process(rd,addr)
     begin
        if rd='1' then
        data <= memI(CONV_INTEGER(addr));
        end if;
     end process;

end Behavioral;
```

O. Registrador de Instruções

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

entity InstructionRegister is
    Port ( clk : in STD_LOGIC;
           IR_Id : in STD_LOGIC;
           data_in : in STD_LOGIC_VECTOR (15 downto 0);
           data_out : out STD_LOGIC_VECTOR (15 downto 0));
end InstructionRegister;
```

```
architecture behavorial of InstructionRegister is

begin

    -- Processo sensível ao clock para saídas síncronas.
    process(clk)
    begin
        if rising_edge(clk) then
                --Apenas quando IR_Id está em alto que é possível carregar
                --o valor do dado.
            if (IR_Id = '1') then
                data_out <= data_in;
            end if;
        end if;
    end process;

end behavorial;
```

P. Bloco de controle

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


entity bloco_controle is -- entradas e saídas do bloco de controle
        port(           clk, reset: in STD_LOGIC;
                                IR: in STD_LOGIC_VECTOR(15 downto 0);
                                RF_Rp_zero: in STD_LOGIC;
                                D_rd, D_wr, RF_s1, RF_s0: out STD_LOGIC;
                                D_addr, RF_W_data: out STD_LOGIC_VECTOR(7 downto 0);
                                PC_id, PC_clr, PC_inc, I_rd, IR_id: out STD_LOGIC;
                                RF_W_wr, RF_Rp_rd, RF_Rq_rd: out STD_LOGIC;
                                alu_s1, alu_s0: out STD_LOGIC;
                                RF_W_addr, RF_Rp_addr,
                                RF_Rq_addr: out STD_LOGIC_VECTOR(3 downto 0));
end bloco_controle;

architecture synth of bloco_controle is -- declaraç o  da variavel estados e sinais internos

        type statetype is (inicio, busca, decodifica, carrega, armazena, soma,
        constante, subtrair, saltar_se, saltar);
        signal state, nextstate: statetype;
        signal op, r, ra, rb, rc:STD_LOGIC_VECTOR(3 downto 0);
        signal d, c, o: STD_LOGIC_VECTOR(7 downto 0);


begin
        -- sinais que ser o utilizados pela maquina de estados para transferir os dados espera
        -- op deve comandar a qual operaç o  a ser realizada pelo bloco operacional
        -- r, ra, rb, rc s o registradores utilizados para carregar os valores nas operaç es
        -- d, o e c s o posiç es  de memória.
    op <= IR(15 downto 12);
        r <= IR(11 downto 8);
        ra <= IR(11 downto 8);
        rb <= IR(7 downto 4);
        rc <= IR(3 downto 0);
        d <= IR(7 downto 0);
```

```
        c <= IR(7 downto 0);
        o <= IR(7 downto 0);

process (clk) begin -- registrador de estados que promove a mudança de estado sincrona ao sina
                if clk'event and clk = '1' then
                        if reset = '1' then state <= inicio;
                        else state <= nextstate;
                        end if;
                end if;
        end process;

        process (state, op) begin -- logica de mudança de estados
                case state is
                        when inicio => nextstate <= busca;
                        when busca => nextstate <= decodifica;
                        when decodifica => case op is -- novo case determina qual operaç o  co
                                                when "0000" => nextstate <= carrega;
                                                when "0001" => nextstate <= armazena;
                                                when "0010" => nextstate <= soma;
                                                when "0011" => nextstate <= constante;
                                                when "0100" => nextstate <= subtrair;
                                                when "0101" => nextstate <= saltar_se;
                                                when others => nextstate <= inicio; -- caso n
                                        end case;

                        when saltar_se => case RF_Rp_zero is --identificando se a flag0 está a
                                                when '1' => nextstate <= saltar;
                                                when '0' => nextstate <= busca;
                                                when others => nextstate <= inicio; -- caso n
                                        end case;

                        when carrega => nextstate <= busca;
                        when armazena => nextstate <= busca;
                        when soma => nextstate <= busca;
                        when constante => nextstate <= busca;
                        when subtrair => nextstate <= busca;
                        when saltar => nextstate <= busca;
                        when others => nextstate <= inicio; -- caso n o esperado
                end case;

        end process;

        process (state) begin --atualizaç oo  das saídas para cada estado
        -- reset de todas as saídas para manipular apenas as desejadas
                        PC_clr <= '0'; PC_id <= '0'; PC_inc <= '0';
                        I_rd <= '0';IR_id <= '0';
                        D_addr <= "00000000"; D_rd <= '0'; D_wr <= '0';
                        RF_Rp_addr <= "0000"; RF_Rp_rd <= '0'; RF_Rq_addr <= "0000"; RF_Rq_rd <
                        RF_W_addr <= "0000"; RF_W_data <= "00000000"; RF_W_wr <= '0'; RF_s0 <=
                        RF_s1 <= '0'; alu_s0 <= '0'; alu_s1 <= '0';

        case state is -- case de definiç o , onde a partir do estado as variaveis de controle
                        when inicio => PC_clr <= '1';
                        when busca => I_rd <= '1';
                                PC_inc <= '1';
                                IR_id <= '1';
                        when carrega => D_addr <= d;
                                D_rd <= '1';
```

```
                RF_s1 <= '0';
                RF_s0 <= '1';
                RF_W_addr <= ra;
                RF_W_wr <='1';
            when armazena => D_addr <= d;
                D_wr <= '1';
                RF_s1 <= 'X';
                RF_s0 <= 'X';
                RF_Rp_addr <= ra;
                RF_Rp_rd <='1';
            when soma => RF_Rp_addr <= rb;
                RF_Rp_rd <= '1';
                RF_s1 <= '0';
                RF_s0 <= '0';
                RF_Rq_addr <= rc;
                RF_Rq_rd <= '1';
                RF_W_addr <= ra;
                RF_W_wr <='1';
                alu_s0 <= '1';
                alu_s1 <= '0';
            when constante => RF_s1 <= '1';
                RF_s0 <= '0';
                RF_W_addr <= ra;
                RF_W_wr <= '1';
            when subtrair => RF_Rp_addr <= rb;
                RF_Rp_rd <= '1';
                RF_s1 <= '0';
                RF_s0 <= '0';
                RF_Rq_addr <= rc;
                RF_Rq_rd <= '1';
                RF_W_addr <= ra;
                RF_W_wr <='1';
                alu_s0 <= '0';
                alu_s1 <= '1';
            when saltar_se => RF_Rp_addr <= ra;
                RF_Rp_rd <= '1';
            when saltar => PC_id <= '1';
            when others => null;
end case;
end process;
```