Instituto Superior Técnico

MEEC

2º Semestre 2014/2015

Arquitecturas Avançadas de Computadores

1º Projecto

Simulacao processador μRISC com funcionamento multi-ciclo

João Baúto **Nº 72856**
João Severino **Nº 73608**

Docente: Prof.Leonel Sousa

29 de Março de 2015

Conteúdo

1. Introdução

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam interdum libero a eros posuere porta. Praesent a risus id enim consectetur facilisis. Nulla ut euismod tortor. Cras in ipsum tempus, vestibulum lorem nec, posuere odio. Proin mollis, lectus non lacinia cursus, odio lacus gravida enim, non rutrum turpis dolor eu nunc. Praesent cursus semper lorem, commodo elementum nisi ullamcorper et. Donec eu ligula diam. Ut nunc ante, viverra eget posuere ut, accumsan in diam. Aliquam erat volutpat. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse a augue est. Quisque ut neque lorem.

Fusce id neque at urna viverra faucibus a ac enim. Aenean porttitor ex et vehicula fermentum. Praesent sit amet vehicula ex, vitae mattis quam. Praesent sit amet magna a sem suscipit cursus. Ut blandit finibus elit sed interdum. Praesent dignissim nulla ut lacus dictum congue. Quisque maximus nibh nunc, vitae consequat quam eleifend ut. Etiam vel ultrices lectus, sit amet imperdiet eros. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Proin consequat justo ut massa imperdiet fermentum. Integer eget velit felis. Interdum et malesuada fames ac ante ipsum primis in faucibus. Nullam imperdiet leo.

2. Arquitectura do μRISC

2.1 Unidade de descodificação - Decoder

Por decisão própria, na unidade de descodificação foi efectuado o máximo possível de descodificação de operações, selectores de *multiplexers* e unidades funcionais. Desta forma é nos possível generalizar as restantes unidades funcionais centralizando toda a descodificação numa só unidade. Uma consequência desta metodologia é o aumento da complexidade da unidade e o número de sinais de *output*.

O mesmo método foi aplicado para as instruções de *jump*. Uma descodificação parcial é feita no *decoder* permitindo diminuir a lógica aplicada à unidade funcional de saltos.

2.2 Unidade de Armazenamento - Memória RAM/ROM partilhada

De forma a facilitar o endereçamento da memória optou-se por uma unidade de armazenamento partilhado. No início da simulação esta é inicializada a partir de um ficheiro .txt graças à utilização de uma *impure function* que introduz as instruções a primeira posição de memória incrementando o endereço para a seguinte instrução. Esta unidade apresenta três entradas, Din para o armazenamento de dados através para instrução *store*, Addr_Instr que indica o endereço da próxima instrução a ser enviada para o *Decoder* e Addr_Dados que endereça a posição para onde será feito uma instrução de *load*. Como saídas tem-se Dout_Dados proveniente da instrução *load* e Instr que indica a próxima instrução.

As vantagens deste tipo de memória é a facilidade de endereçamento uma vez que não é necessário fornecer um offset para o caso em que é necessário aceder a um array *por exemplo*. Como desvantagem tem se o facto de o programador necessitar de uma extra atenção às posições de memória onde guarda dos dados podendo substituir futuras instruções.

Com duas memórias independentes seria possível evitar este problema caso ambas as memórias fossem inicializadas com os mesmos dados (instruções e *arrays*).

2.3 Unidade lógico-aritmética - ALU

Desenhou-se a ALU com três unidades a funcionarem em paralelo, abaixo descritas com maior detalhe. O resultado produzido por estas unidades é introduzido num *multiplexer* que selecciona de acordo com sinais provenientes da unidade de descodificação qual o resultado e as *Flags* a colocar à saída da ALU.

2.3.1 Unidade Aritmética

A unidade Aritmética é responsável pelas operações apresentadas na tabela 2.1.

OP	Operação	Mnemónica	Flags actualizadas
00000	$C = A + B$	add c, a, b	S,C,Z,V
00001	$C = A + B + 1$	addinc c, a, b	S,C,Z,V
00011	$C = A + 1$	inca c, a	S,C,Z,V
00100	$C = A - B - 1$	subdec c, a, b	S,C,Z,V
00101	$C = A - B$	sub c, a, b	S,C,Z,V
00110	$C = A - 1$	deca c, a	S,C,Z,V

Tabela 2.1: Operações Aritméticas

A unidade aritmética começa por analisar qual a operação a executar de acordo com os dados vindos da unidade de descodificação e em seguida começa por calcular o segundo membro da operação $C = A + operB$ em que

$$operB = \begin{cases} B & : OP = 00000 \\ B+1 & : OP = 00001 \\ 1 & : OP = 00011 \\ -B-1 & : OP = 00100 \\ -B & : OP = 00101 \\ -1 & : OP = 00110 \end{cases}$$

De seguida calcula $C = A + operB$ e as *Flags* correspondentes com base na análise do resultado e dos operandos.

2.3.2 Unidade Lógica

A unidade Lógica é responsável pelas operações apresentadas na tabela 2.2.

OP	Operação	Mnemónica	Flags actualizadas
10000	$C = 0$	zeros c	Nenhuma
10001	$C = A\&B$	and c, a, b	S,Z
10010	$C =!A\&B$	andnota c, a, b	S,Z
10011	$C = B$	passb c, b	Nenhuma
10100	$C = A\&!B$	andnotb c, a, b	S,Z
10101	$C = A$	passa c, a	S,Z
10110	$C = A \oplus B$	xor c, a, b	S,Z
10111	$C = A\|B$	or c, a, b	S,Z
11000	$C =!A\&!B$	nor c, a, b	S,Z
11001	$C =!(A \oplus B)$	xnor c, a, b	S,Z
11010	$C =!A$	passnota c, a	S,Z
11011	$C =!A\|B$	ornota c, a, b	S,Z
11100	$C =!B$	passnotb c, b	S,Z
11101	$C =!A\|!B$	nand c, a, b	S,Z
11111	$C = 1$	ones c	Nenhuma

Tabela 2.2: Operações de Deslocamento

2.3.3 Unidade de Deslocamentos

A unidade de Deslocamentos é responsável pelas operações apresentadas na tabela 2.3.

OP	Operação	Mnemónica	Flags actualizadas
01000	$C = ShiftLgicoEsq.(A)$	lsl c, a	S,C,Z
01001	$C = ShiftAritmticoDir.(A)$	asr c, a	S,C,Z

Tabela 2.3: Operações de Deslocamento

No caso do *shift* lógico a saída resulta do deslocamento do sinal de entrada uma posição e preenchimento do *bit0* com 0. No caso do *shift* aritmético a saída resulta do deslocamento do sinal de entrada uma posição e preenchimento do *bit15* com o *bit15* da entrada.

2.4 Unidade de Constantes

A unidade de Constantes é responsável pelas operações apresentadas na tabela 2.4.

Optámos por separar estas operações das restantes da ALU de modo a facilitar a descodificação das instruções por parte do *Decoder* e uma vez que o caminho crítico é devido à elevada complexidade da ALU a separação desta unidade funcional da ALU não tem qualquer tipo de influência na frequência de relógio.

Formato	Operação	Mnemónica
I	$C = Constante$	loadlit c, Const
II	$C = Const8\|(C\&0xff00)$	lcl c, Const8
II	$C = (Const8 << 8)\|(C\&0x00ff)$	lch c, Const8

Tabela 2.4: Operações com Constantes

2.5 Unidade de Controlo de saltos e *flags*

Desenhou-se a unidade de modo a controlar o próximo endereço a enviar ao *Program Counter* (PC). Esta unidade guarda os valores das *flags* provenientes da ALU em registos e depois usa esses registos para calcular as condições de salto.

Juntámos os registos das *flags* com a unidade de controlo de saltos de modo a que consoante a condição de salto vinda do *Decoder* se pudesse calcular se se deveria executar um salto ou se permitíamos que o valor do *PC* fosse incrementado normalmente.

O cálculo do próximo endereço do *PC* é feito em 4 fases.

1. Cálculo da condição de salto
2. Cálculo do *offset* para o caso de saltos no Formato I ou do Formato II
3. Determinar se o salto é para um *offset* ou para um registo
4. Determinar o próximo endereço do *PC* com base no tipo de *jump* (condicional ou incondicional) e a condição

$$\text{Condição} = \begin{cases} 1 & : \quad COND = 0000 \\ flagV & : \quad COND = 0011 \\ flagS & : \quad COND = 0100 \\ flagZ & : \quad COND = 0101 \\ flagC & : \quad COND = 0110 \\ flagS + flagZ & : \quad COND = 0111 \\ 0 & : \quad others \end{cases}$$

$$\text{Offset} = \begin{cases} \begin{array}{r} Destino(11)\&Destino(11)\&Destino(11)\&Destino(11) \\ \&Destino(11\ downto\ 0) \end{array} & : \quad OP = 10 \\ \begin{array}{r} Destino(7)\&Destino(7)\&Destino(7)\&Destino(7)\&Destino(7) \\ \&Destino(7)\&Destino(7)\&Destino(7)\&Destino(7\ downto\ 0) \end{array} & : \quad others \end{cases}$$

$$
\text{Jump Address} = \begin{cases} RB & : \quad OP = 11 \\ PC + 1 + Offset & : \quad others \end{cases}
$$

$$
\text{Próximo PC} = \begin{cases} Jump\ Address & : \quad enable\_jump = 1 \ \cdot \ (\text{Condição} \oplus OP(0)) = 1 \\ Jump\ Address & : \quad enable\_jump = 1 \ \cdot \ OP(1) = 1 \\ PC + 1 & : \quad others \end{cases}
$$

2.6 Esquema da Arquitectura

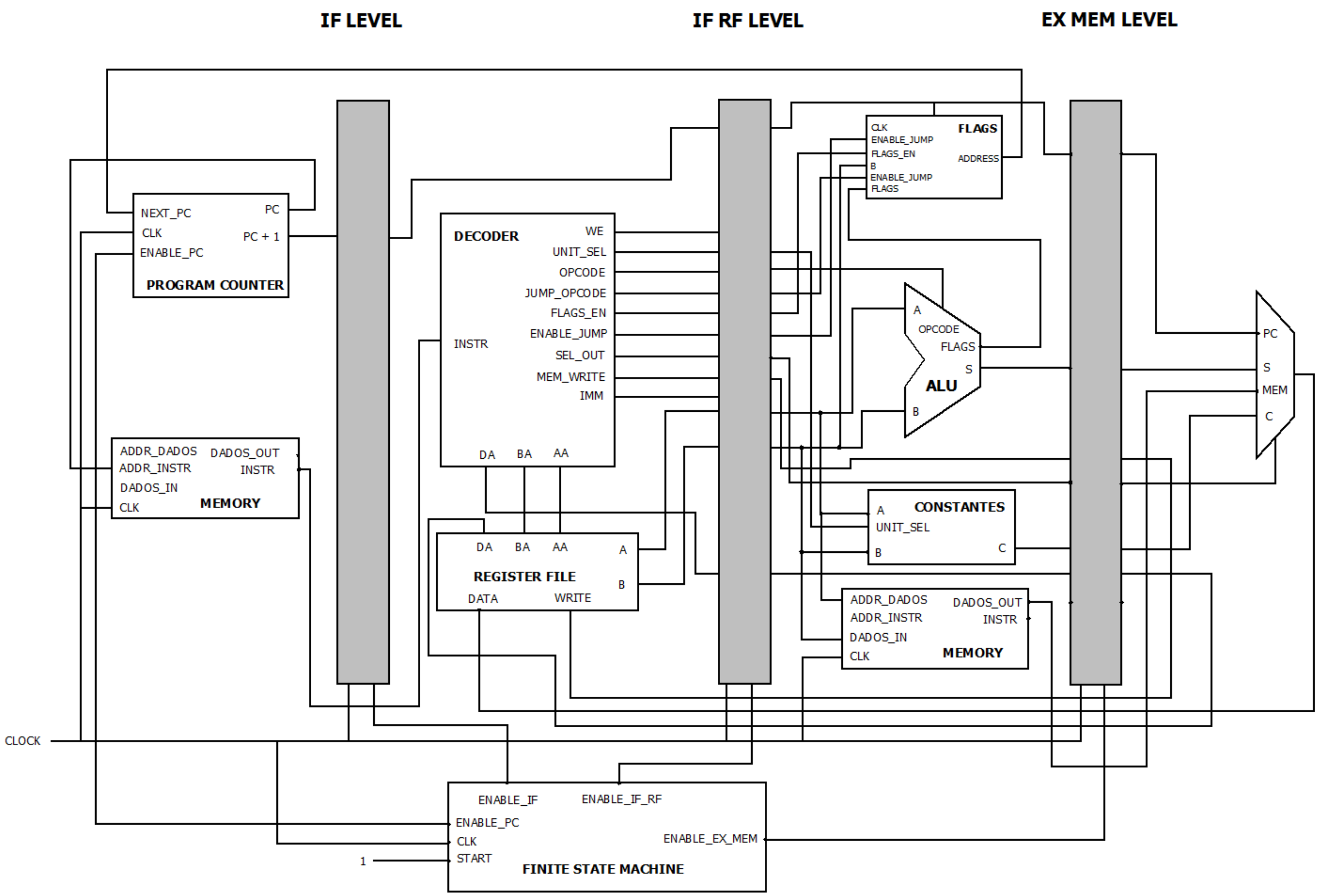

Figura 2.1: Arquitectura μRISC.

3. Anexos

3.1 Unidade de Descodificação

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

entity decoder is
    Port ( INSTRUCTION : in STD_LOGIC_VECTOR (15 downto 0);
           IMM  : out STD_LOGIC_VECTOR (15 downto 0); -- 10-0 quando 01XXX; 7-0 quando 11XXX
           OPCODE : out STD_LOGIC_VECTOR (4 downto 0);
           UNIT_SEL : out STD_LOGIC_VECTOR(1 downto 0);
           DA : out STD_LOGIC_VECTOR(2 downto 0);
           AA : out STD_LOGIC_VECTOR(2 downto 0);
           BA : out STD_LOGIC_VECTOR(2 downto 0);
           WE : out STD_LOGIC;
           SEL_OUT : out STD_LOGIC_VECTOR(1 downto 0);
           MEM_WRITE : out STD_LOGIC;
           jump_opcode : out STD_LOGIC_VECTOR(13 downto 0);
           flags_en : out STD_LOGIC_VECTOR(3 downto 0);
           enable_jump : out STD_LOGIC
           );
end decoder;

architecture Behavioral of decoder is
   signal instruct : STD_LOGIC_VECTOR(15 downto 0);
   signal format : STD_LOGIC_VECTOR(1 downto 0);
   signal imm_temp : STD_LOGIC_VECTOR(15 downto 0);
begin
   instruct <= INSTRUCTION;
   format <= instruct(15) & instruct(14);
   jump_opcode <= instruct(13 downto 0);

   --select unit inside alu
```

```
UNIT_SEL <= instruct(10 downto 9) when format = "10" else
           instruct(15) & instruct(10) when format = "11" or format = "01" else
           (others =>'0');

DA <= (others => '1') when instruct(15 downto 11)="00110" else
      instruct(13 downto 11);

AA <= instruct(13 downto 11) when format = "01" or format = "11" else
      instruct(5 downto 3);

BA <= instruct(2 downto 0);

OPCODE <= instruct(10 downto 6) when format = "10" or format = "00" else
         (others =>'0');

imm_temp(10 downto 0) <= instruct(10 downto 0) when format = "01" else
                       instruct(7) & instruct(7) & instruct(7) & instruct(7 downto 0) when format =
                           "11" else
                       (others =>'0');

imm_temp(15 downto 11) <= (others => imm_temp(10));
IMM <= imm_temp;

--write enable for register file
WE <= '1' when instruct(15 downto 11)="00110" else -- Jump and Link
      '1' when instruct(15 downto 14)="10" and instruct(10 downto 6)="01010" else -- load c, a
      '1' when instruct(15 downto 14)="01" or instruct(15 downto 14)="11" else -- Constantes
      '1' when instruct(15 downto 14)="10" and instruct(10 downto 7)/="0101" else -- ALU
      '0';

--write back mux select
SEL_OUT <= "00" when instruct(15 downto 14)="10" and  instruct(10 downto 7) /= "0101" else --ALU
          "01" when instruct(15 downto 14)="10" else                                      --MEM
          "10" when instruct(14)='1' else                                                 --Const
          "11";                                                                           --PC

--flags_enable
--Z
flags_en(3) <='0' when instruct(15 downto 14)/="10" else
             '0' when instruct(10 downto 7)="0101" else
             '0' when instruct(10 downto 6)="10000" else
             '0' when instruct(10 downto 6)="10011" else
```

```
            '0' when instruct(10 downto 6)="11111" else
            '1';
  --S
  flags_en(2) <='0' when instruct(15 downto 14)/="10" else
            '0' when instruct(10 downto 7)="0101" else
            '0' when instruct(10 downto 6)="10000" else
            '0' when instruct(10 downto 6)="10011" else
            '0' when instruct(10 downto 6)="11111" else
            '1';
  --C
  flags_en(1) <='0' when instruct(15 downto 14)/="10" else
            '0' when instruct(10 downto 7)="0101" else
            '0' when instruct(10)='1' else
            '1';
  --V
  flags_en(0) <='1' when instruct(15 downto 14)="10" and instruct(10 downto 9)="00" else
            '0';

  enable_jump <= '1' when instruct(15 downto 14)="00" else
             '0';

  MEM_WRITE <= '1' when format = "10" and instruct(10 downto 6) = "01011" else
           '0';

end Behavioral;
```

3.2 Unidade lógico-aritmética

3.2.1 Unidade Aritmética

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


entity arithmetic_unit is
  Port (A : in STD_LOGIC_VECTOR (15 downto 0);
```

```vhdl
        B : in STD_LOGIC_VECTOR (15 downto 0);
        C : out STD_LOGIC_VECTOR (15 downto 0);
        OP : in STD_LOGIC_VECTOR (2 downto 0);
        Flags : out STD_LOGIC_VECTOR (3 downto 0)); --ZSCV
end arithmetic_unit;

architecture Behavioral of arithmetic_unit is
   signal C_extra : STD_LOGIC_VECTOR (16 downto 0);
   signal operB, operB2, Cin : STD_LOGIC_VECTOR (15 downto 0);
begin
   -- C_extra <= A+B when OP="000" else
           -- A+B+1 when OP="001" else
           -- A+1 when OP="011" else
           -- A-B-1 when OP="100" else
           -- A-B when OP="101" else
           -- A-1;

   operB <= B when OP(2 downto 1)="00" else
           -B when OP(2 downto 1)="10" else
           (others=>'0');
   Cin <= X"0001" when OP(2)='0' and OP(0)='1' else
         X"FFFF" when OP(2)='1' and OP(0)='0' else
         (others=>'0');
   operB2 <= operB+Cin;
   C_extra <= ('0'& A)+('0' & operB2);


   C <= C_extra(15 downto 0);

   --Zero
   Flags(3) <= not(C_extra(15) or C_extra(14) or C_extra(13) or C_extra(12) or C_extra(11) or C_extra(10)
        or C_extra(9) or C_extra(8) or C_extra(7) or C_extra(6) or C_extra(5) or C_extra(4) or C_extra(3)
        or C_extra(2) or C_extra(1) or C_extra(0));
   --Sign
   Flags(2) <= C_extra(15);
   --Carry
   Flags(1) <= C_extra(16);
   --oVerflow
   Flags(0) <= ((A(15) xnor operB2(15)) and (A(15) xor C_extra(15))) or ((operB(15) xnor Cin(15)) and
        (operB(15) xor operB2(15)));

end Behavioral;
```

3.2.2 Unidade Lógica

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


entity logic_unit is
   Port (A : in STD_LOGIC_VECTOR (15 downto 0);
         B : in STD_LOGIC_VECTOR (15 downto 0);
         C : out STD_LOGIC_VECTOR (15 downto 0);
         OP : in STD_LOGIC_VECTOR (3 downto 0);
         Flags : out STD_LOGIC_VECTOR (3 downto 0)); --ZSCV
end logic_unit;

architecture Behavioral of logic_unit is
   signal C_intern : STD_LOGIC_VECTOR (15 downto 0);
begin
   C_intern <= (others => '0') when OP="0000" else
            A and B when OP="0001" else
            not(A) and B when OP="0010" else
            B when OP="0011" else
            A and not(B) when OP="0100" else
            A when OP="0101" else
            A xor B when OP="0110" else
            A or B when OP="0111" else
            not(A) and not(B) when OP="1000" else
            not(A xor B) when OP="1001" else
            not(A) when OP="1010" else
            not(A) or B when OP="1011" else
            not(B) when OP="1100" else
            A or not(B) when OP="1101" else
            not(A) or not(B) when OP="1110" else
            (others => '1');

   C <= C_intern;

   --Zero
   Flags(3) <= not(C_intern(15) or C_intern(14) or C_intern(13) or C_intern(12) or C_intern(11) or
```

```
      C_intern(10) or C_intern(9) or C_intern(8) or C_intern(7) or C_intern(6) or C_intern(5) or
      C_intern(4) or C_intern(3) or C_intern(2) or C_intern(1) or C_intern(0));
   --Sign
   Flags(2) <= C_intern(15);
   --Carry
   Flags(1) <= '0';
   --oVerflow
   Flags(0) <= '0';

end Behavioral;
```

3.2.3 Unidade Deslocamento

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

entity arith_logic_shift is
   Port (A : in STD_LOGIC_VECTOR (15 downto 0);
        C : out STD_LOGIC_VECTOR (15 downto 0);
        OP : in STD_LOGIC;
        Flags : out STD_LOGIC_VECTOR (3 downto 0)); --ZSCV
end arith_logic_shift;

architecture Behavioral of arith_logic_shift is
   signal C_intern : STD_LOGIC_VECTOR (15 downto 0);
begin
   C_intern <= A(14 downto 0) & '0' when OP='0' else
            A(15) & A(15 downto 1);

   C <= C_intern;

   --Zero
   Flags(3) <= not(C_intern(15) or C_intern(14) or C_intern(13) or C_intern(12) or C_intern(11) or
      C_intern(10) or C_intern(9) or C_intern(8) or C_intern(7) or C_intern(6) or C_intern(5) or
      C_intern(4) or C_intern(3) or C_intern(2) or C_intern(1) or C_intern(0));
   --Sign
   Flags(2) <= C_intern(15);
   --Carry
```

```vhdl
   Flags(1) <= A(15) and not(OP);
   --oVerflow
   Flags(0) <= '0';

end Behavioral;
```

3.3 Unidade de Constantes

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

entity constantes is
    Port ( Const : in STD_LOGIC_VECTOR (15 downto 0);
          C_in : in STD_LOGIC_VECTOR (15 downto 0);
          C : out STD_LOGIC_VECTOR (15 downto 0);
          OP_SEL : in STD_LOGIC_VECTOR (1 downto 0));
end constantes;

architecture Behavioral of constantes is
   signal Cand : STD_LOGIC_VECTOR (15 downto 0);
begin
   Cand <= C_in and X"FF00" when OP_SEL(0)='0' else
          C_in and X"00FF";
   C <= Const when OP_SEL(1)='0' else
        X"00" & Const(7 downto 0) or Cand when OP_SEL(0)='0' else
        Const(7 downto 0) & X"00" or Cand;

end Behavioral;
```

3.4 Unidade de Controlo de saltos e *flags*

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

```vhdl
use IEEE.STD_LOGIC_SIGNED.ALL;

entity Flags is
    Port ( clk : in STD_LOGIC;
           Flags : in STD_LOGIC_VECTOR (3 downto 0);
           enable_flags : in STD_LOGIC_VECTOR (3 downto 0);
           enable_jump : in STD_LOGIC;
           jump_info : in STD_LOGIC_VECTOR (13 downto 0);
           PCm1 : in STD_LOGIC_VECTOR (15 downto 0);
           RB : in STD_LOGIC_VECTOR (15 downto 0);
           output_address : out STD_LOGIC_VECTOR (15 downto 0));
end Flags;

architecture Behavioral of Flags is
   signal Z,S,C,V : STD_LOGIC := '0';
   signal Cond_Status : STD_LOGIC := '0';
   signal OP : STD_LOGIC_VECTOR(1 downto 0):= (others=>'0');
   signal COND : STD_LOGIC_VECTOR(3 downto 0):= (others=>'0');
   signal Destino : STD_LOGIC_VECTOR(15 downto 0):= (others=>'0');
   signal address : STD_LOGIC_VECTOR(15 downto 0) := (others=>'0');
   signal offset : STD_LOGIC_VECTOR(15 downto 0):= (others=>'0');
begin

   -- Flag Registers
   Z <= Flags(3) when enable_flags(3)='1' and rising_edge(clk);
   S <= Flags(2) when enable_flags(2)='1' and rising_edge(clk);
   C <= Flags(1) when enable_flags(1)='1' and rising_edge(clk);
   V <= Flags(0) when enable_flags(0)='1' and rising_edge(clk);

   OP <= jump_info(13 downto 12);
   COND <= jump_info(11 downto 8);

   Cond_Status <= '1' when COND="0000" else
                V when COND="0011" else
                S when COND="0100" else
                Z when COND="0101" else
                C when COND="0110" else
                S or Z when COND="0111" else
                '0';

   offset <= jump_info(11) & jump_info(11) & jump_info(11) & jump_info(11) & jump_info(11 downto 0) when
       OP="10" else
```

```
        jump_info(7) & jump_info(7) & jump_info(7) & jump_info(7) & jump_info(7) & jump_info(7) &
            jump_info(7) & jump_info(7) & jump_info(7 downto 0);

   Destino <= RB when OP="11" else
            PCm1 + offset;

   address <= Destino when enable_jump='1' and (Cond_Status xnor OP(0))='1' else
            Destino when enable_jump='1' and OP(1)='1' else
            PCm1;

   output_address <= address;

end Behavioral;
```

3.5 *Program Counter*

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

entity ProgramCounter is
    Port ( next_PC : in STD_LOGIC_VECTOR (15 downto 0);
          enable_pc : in STD_LOGIC;
          PC : out STD_LOGIC_VECTOR (15 downto 0);
          PCm1 : out STD_LOGIC_VECTOR (15 downto 0);
          clk : in STD_LOGIC);
end ProgramCounter;

architecture Behavioral of ProgramCounter is
   signal currentPC : STD_LOGIC_VECTOR (15 downto 0) := (others =>'0');
begin
   process(clk,enable_pc)
   begin
      if enable_pc='1' and rising_edge(clk) then
         currentPC <= next_PC;
      end if;
   end process;

   PC <= currentPC;
```

```
  PCm1 <= currentPC+1;

end Behavioral;
```

3.6 Unidade de Constantes

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

entity constantes is
    Port ( Const : in STD_LOGIC_VECTOR (15 downto 0);
          C_in : in STD_LOGIC_VECTOR (15 downto 0);
          C : out STD_LOGIC_VECTOR (15 downto 0);
          OP_SEL : in STD_LOGIC_VECTOR (1 downto 0));
end constantes;

architecture Behavioral of constantes is
  signal Cand : STD_LOGIC_VECTOR (15 downto 0);
begin
  Cand <= C_in and X"FF00" when OP_SEL(0)='0' else
          C_in and X"00FF";
  C <= Const when OP_SEL(1)='0' else
       X"00" & Const(7 downto 0) or Cand when OP_SEL(0)='0' else
       Const(7 downto 0) & X"00" or Cand;

end Behavioral;
```

3.7 Unidade de Armazenamento Partilhada

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use STD.TEXTIO.all;
use STD.TEXTIO;
```

```vhdl
use IEEE.STD_LOGIC_TEXTIO.all;


entity rom_instrc is
   port(clk : in std_logic;

        we : in std_logic;
        addr_instr : in std_logic_vector(15 downto 0);
        addr_dados : in std_logic_vector(15 downto 0);
        din : in std_logic_vector(15 downto 0);
        dout_instr : out std_logic_vector(15 downto 0);
        dout_dados : out std_logic_vector(15 downto 0);
        print : out std_logic);
   end rom_instrc;

architecture Behavioral of rom_instrc is
  type RamType is array(0 to 65535) of STD_LOGIC_VECTOR(15 downto 0);
   impure function InitRamFromFile (RamFileName : in string) return RamType is
      file INFILE : TEXT is in "demo_lab1_3.txt";
      variable DATA_TEMP : STD_LOGIC_VECTOR(15 downto 0);
      variable IN_LINE: LINE;
      variable RAM : RamType;
      variable index :integer;

      begin
           index := 0;
           while NOT(endfile(INFILE)) loop
              readline(INFILE,IN_LINE);
              hread(IN_LINE, DATA_TEMP);
              RAM(index) := DATA_TEMP;
              index := index + 1;
           end loop;
           for index in index to 65535 loop
              RAM(index) := X"0000";
           end loop;
   return RAM;
   end function;


signal RAM : RamType := InitRamFromFile("demo_lab1_3.txt");
--signal teste : integer := InitRamToFile(RAM);
signal dados : STD_LOGIC_VECTOR(15 downto 0);
```

```
signal instr: STD_LOGIC_VECTOR(15 downto 0);
begin
  process (clk)
  begin
    if clk'event and clk = '1' then
      if we ='1' then
        RAM(conv_integer(dados)) <= din;
      end if;
      dout_instr <= RAM(conv_integer(instr)); -- REVER
      dout_dados <= RAM(conv_integer(dados)); -- REVER
    end if;
  end process;

  dados <= addr_dados(15 downto 0);
  instr <= addr_instr(15 downto 0);
  print <= '1' when RAM(conv_integer(instr)) = X"2FFF" else
          '0';

end Behavioral;
```

3.8 Multiplexer de Selecção de saída

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

entity writeback_mux is
  Port (ALU : in STD_LOGIC_VECTOR (15 downto 0);
      MEM : in STD_LOGIC_VECTOR (15 downto 0);
      Consts : in STD_LOGIC_VECTOR (15 downto 0);
      PC : in STD_LOGIC_VECTOR (15 downto 0);
      Sel_WB : in STD_LOGIC_VECTOR (1 downto 0);
      C : out STD_LOGIC_VECTOR (15 downto 0)
      );
end writeback_mux;

architecture Behavioral of writeback_mux is
```

```
begin

   C <= ALU when Sel_WB="00" else
        MEM when Sel_WB="01" else
        Consts when Sel_WB="10" else
        PC;

end Behavioral;
```

3.9 Registos do Nível ID

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



entity IF_Regs is
   Port (Next_PC_in : in STD_LOGIC_VECTOR (15 downto 0);
         Next_PC_out : out STD_LOGIC_VECTOR (15 downto 0);
         clk : in STD_LOGIC;
         enable : in STD_LOGIC
         );
end IF_Regs;

architecture Behavioral of IF_Regs is
begin
   process (clk)
   begin
   if clk'event and clk='1' then
      if enable = '1' then
         Next_PC_out <= Next_PC_in;
      end if;
   end if;
end process;

end Behavioral;
```

3.10 Registos do Nível ID RD

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


entity ID_RF_Regs is
   Port (Current_PC_in : in STD_LOGIC_VECTOR (15 downto 0);
         Current_PC_out : out STD_LOGIC_VECTOR (15 downto 0);
         Next_PC_in : in STD_LOGIC_VECTOR (15 downto 0);
         Next_PC_out : out STD_LOGIC_VECTOR (15 downto 0);
         IMM_in : in STD_LOGIC_VECTOR (15 downto 0);
         IMM_out : out STD_LOGIC_VECTOR (15 downto 0);
         OPCODE_in : in STD_LOGIC_VECTOR (4 downto 0);
         OPCODE_out : out STD_LOGIC_VECTOR (4 downto 0);
         UNIT_SEL_in : in STD_LOGIC_VECTOR (1 downto 0);
         UNIT_SEL_out : out STD_LOGIC_VECTOR (1 downto 0);
         DA_in : in STD_LOGIC_VECTOR (2 downto 0);
         DA_out : out STD_LOGIC_VECTOR (2 downto 0);
         A_in : in STD_LOGIC_VECTOR (15 downto 0);
         A_out : out STD_LOGIC_VECTOR (15 downto 0);
         B_in : in STD_LOGIC_VECTOR (15 downto 0);
         B_out : out STD_LOGIC_VECTOR (15 downto 0);
         WE_in : in STD_LOGIC;
         WE_out : out STD_LOGIC;
         SEL_OUT_in : in STD_LOGIC_VECTOR (1 downto 0);
         SEL_OUT_out : out STD_LOGIC_VECTOR (1 downto 0);
         MEM_WRITE_in : in STD_LOGIC;
         MEM_WRITE_out : out STD_LOGIC;
         jump_opcode_in : in STD_LOGIC_VECTOR (13 downto 0);
         jump_opcode_out : out STD_LOGIC_VECTOR (13 downto 0);
         flags_en_in : in STD_LOGIC_VECTOR (3 downto 0);
         flags_en_out : out STD_LOGIC_VECTOR (3 downto 0);
         enable_jump_in : in STD_LOGIC;
         enable_jump_out : out STD_LOGIC;
         clk : in STD_LOGIC;
         enable : in STD_LOGIC);
end ID_RF_Regs;

architecture Behavioral of ID_RF_Regs is
begin
   process (clk)
   begin
```

```
  if clk'event and clk='1' then
    if enable = '1' then
      Current_PC_out <= Current_PC_in;
      Next_PC_out <= Next_PC_in;
      OPCODE_out <= OPCODE_in;
      IMM_out <= IMM_in;
      UNIT_SEL_out <= UNIT_SEL_in;
      A_out <= A_in;
      B_out <= B_in;
      WE_out <= WE_in;
      SEL_OUT_out <= SEL_OUT_in;
      MEM_WRITE_out <= MEM_WRITE_in;
      jump_opcode_out <= jump_opcode_in;
      flags_en_out <= flags_en_in;
      enable_jump_out <= enable_jump_in;
      DA_out <= DA_in;
    end if;
  end if;
end process;
end Behavioral;
```

3.11 Registos do Nível EX MEM

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

entity EX_MEM_Regs is
  Port (Next_PC_in : in STD_LOGIC_VECTOR (15 downto 0); --alterado
      Next_PC_out : out STD_LOGIC_VECTOR (15 downto 0); --alterado
      C_in : in STD_LOGIC_VECTOR (15 downto 0);
      C_out : out STD_LOGIC_VECTOR (15 downto 0);
      WE_in : in STD_LOGIC;
      WE_out : out STD_LOGIC;
      S_in : in STD_LOGIC_VECTOR (15 downto 0);
      S_out : out STD_LOGIC_VECTOR (15 downto 0);
      DA_in : in STD_LOGIC_VECTOR (2 downto 0);
      DA_out : out STD_LOGIC_VECTOR (2 downto 0);
      MUX_WB_in : in STD_LOGIC_VECTOR(1 downto 0);
      MUX_WB_out : out STD_LOGIC_VECTOR(1 downto 0);
```

```
        clk : in STD_LOGIC;
        enable : in STD_LOGIC
        );
end EX_MEM_Regs;

architecture Behavioral of EX_MEM_Regs is

begin
process (clk)
   begin
   if clk'event and clk='1' then
      if enable = '1' then
         Next_PC_out <= Next_PC_in; --alterado
         C_out <= C_in;
         WE_out <= WE_in;
         S_out <= S_in;
         DA_out <= DA_in;
         MUX_WB_out <= MUX_WB_in;
      end if;
   end if;
end process;

end Behavioral;
```

3.12 Finite State Machine - FSM

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use STD.TEXTIO.all;
use STD.TEXTIO;
use IEEE.STD_LOGIC_TEXTIO.all;


entity FSM_Regs is
   Port( START : in STD_LOGIC;
         PRINT : in STD_LOGIC;
         ENABLE_IF : out STD_LOGIC;
         ENABLE_IF_RF : out STD_LOGIC;
         ENABLE_EX_MEM : out STD_LOGIC;
```

```
        ENABLE_PC : out STD_LOGIC;
        CLK : in STD_LOGIC
   );
end FSM_Regs;

architecture Behavioral of FSM_Regs is
   type fsm_states is (INIT, IF_level, IF_RF_level, EX_MEM_level, WB_level, IF2_level, WR_FILE);
   signal curr_state, next_state : fsm_states;
   signal counter : integer := 0;
   signal start2 : std_logic := '1';
   signal stop_print : std_logic := '0';
begin

   process (clk)
      begin
         if clk'event and clk = '1' then
            if START = '0' then
               curr_state <= INIT;
            else
               curr_state <= next_state;
            end if;
         end if;
   end process ;

   process(curr_state, START,start2,print, stop_print)
   file output: TEXT open write_mode is "ram_out.txt";
   variable my_line : LINE;
   begin
      next_state <= curr_state;

      case curr_state is
         when INIT =>
            ENABLE_IF <= '0';
            ENABLE_PC <= '0';
            ENABLE_IF_RF <= '0';
            ENABLE_EX_MEM <= '0';

            if START = '1' and start2 = '1' then
               next_state <= IF_level;
            end if;

            if print = '1' and stop_print = '0' then
```

```
      next_state <= WR_FILE;
    end if;

  when IF_level =>
    ENABLE_IF <= '0';
    ENABLE_PC <= '0';
    ENABLE_IF_RF <= '0';
    ENABLE_EX_MEM <= '0';

    if print = '1' and stop_print = '0' then
      next_state <= WR_FILE;
    else
      next_state <= IF_RF_level;
    end if;

  when IF2_level =>
    ENABLE_IF <= '0';
    ENABLE_PC <= '1';
    ENABLE_IF_RF <= '0';
    ENABLE_EX_MEM <= '0';

    if print = '1' and stop_print = '0' then
      next_state <= WR_FILE;
    else
      next_state <= IF_RF_level;
    end if;

  when IF_RF_level =>
    ENABLE_IF <= '1';
    ENABLE_PC <= '0';
    ENABLE_IF_RF <= '0';
    ENABLE_EX_MEM <= '0';

    if print = '1' and stop_print = '0' then
      next_state <= WR_FILE;
    else
      next_state <= EX_MEM_level;
    end if;

  when EX_MEM_level =>

    ENABLE_IF <= '0';
```

```
        ENABLE_PC <= '0';
        ENABLE_IF_RF <= '1';
        ENABLE_EX_MEM <= '0';

        if print = '1' and stop_print = '0' then
          next_state <= WR_FILE;
        else
          next_state <= WB_level;
        end if;

      when WB_level =>
        ENABLE_IF <= '0';
        ENABLE_PC <= '0';
        ENABLE_IF_RF <= '0';
        ENABLE_EX_MEM <= '1';

        if print = '1' and stop_print = '0' then
          next_state <= WR_FILE;
        else
          next_state <= IF2_level;
        end if;

      when WR_FILE =>
        ENABLE_IF <= '0';
        ENABLE_PC <= '0';
        ENABLE_IF_RF <= '0';
        ENABLE_EX_MEM <= '0';

        start2 <= '0';
        stop_print <= '1';
        next_state <= INIT;

    end case;
  end process;


end Behavioral;
```

3.13 Arquitectura μRISC

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
--use work.full_memory.all;

entity cpu is
   Port( CLK, TEST1 : in STD_LOGIC;
         MEMORY : out STD_LOGIC_VECTOR(15 downto 0);
         INST_NB : out std_LOGIC_VECTOR(15 downto 0);
         PRINT : out STD_LOGIC;
         TEST: out STD_LOGIC_VECTOR(15 downto 0));
end cpu;

architecture Behavioral of cpu is
  component alu
      port(OP_SEL : in STD_LOGIC_VECTOR (4 downto 0);
         A : in STD_LOGIC_VECTOR (15 downto 0);
         B : in STD_LOGIC_VECTOR (15 downto 0);
         S : out STD_LOGIC_VECTOR (15 downto 0);
         Flags : out STD_LOGIC_VECTOR (3 downto 0));
   end component alu;

   component decoder
      port (INSTRUCTION : in STD_LOGIC_VECTOR (15 downto 0);
           IMM  : out STD_LOGIC_VECTOR (15 downto 0); -- 10-0 quando 01XXX; 7-0 quando 11XXX
           OPCODE : out STD_LOGIC_VECTOR (4 downto 0);
           UNIT_SEL : out STD_LOGIC_VECTOR(1 downto 0);
           DA : out STD_LOGIC_VECTOR(2 downto 0);
           AA : out STD_LOGIC_VECTOR(2 downto 0);
           BA : out STD_LOGIC_VECTOR(2 downto 0);
           WE : out STD_LOGIC;
           SEL_OUT : out STD_LOGIC_VECTOR(1 downto 0);
           MEM_WRITE : out STD_LOGIC;
           jump_opcode : out STD_LOGIC_VECTOR(13 downto 0);
           flags_en : out STD_LOGIC_VECTOR(3 downto 0);
           enable_jump : out STD_LOGIC
       );
   end component decoder;

   component registerfile
      port(AA : in STD_LOGIC_VECTOR (2 downto 0);
          A : out STD_LOGIC_VECTOR (15 downto 0);
```

```
        BA : in STD_LOGIC_VECTOR (2 downto 0);
        B : out STD_LOGIC_VECTOR (15 downto 0);
        DA : in STD_LOGIC_VECTOR (2 downto 0);
        WE : in STD_LOGIC;
        DATA : in STD_LOGIC_VECTOR (15 downto 0);
        clk : in STD_LOGIC);
end component registerfile;

component constantes
   port(Const : in STD_LOGIC_VECTOR (15 downto 0);
        C_in : in STD_LOGIC_VECTOR (15 downto 0);
        C : out STD_LOGIC_VECTOR (15 downto 0);
        OP_SEL : in STD_LOGIC_VECTOR (1 downto 0));
end component constantes;

component writeback_mux
   port(ALU : in STD_LOGIC_VECTOR (15 downto 0);
        MEM : in STD_LOGIC_VECTOR (15 downto 0);
        PC : in STD_LOGIC_VECTOR (15 downto 0);
        Consts : in STD_LOGIC_VECTOR (15 downto 0);
        Sel_WB : in STD_LOGIC_VECTOR (1 downto 0);
        C : out STD_LOGIC_VECTOR (15 downto 0));
end component writeback_mux;

component sync_ram
   port (clock  : in  std_logic;
         we     : in  std_logic;
         address : in std_logic_vector;
         datain  : in  std_logic_vector;
         dataout : out std_logic_vector);
end component sync_ram;

component rom_instrc
   port(clk : in std_logic;
        we : in std_logic;
        --MEM_array : out RamType;
        addr_instr : in std_logic_vector(15 downto 0);
        addr_dados : in std_logic_vector(15 downto 0);
        din : in std_logic_vector(15 downto 0);
        print : out std_logic;
        dout_instr : out std_logic_vector(15 downto 0);
        dout_dados : out std_logic_vector(15 downto 0));
```

```
end component;

component Flags
   port(clk : in STD_LOGIC;
        Flags : in STD_LOGIC_VECTOR (3 downto 0);
        enable_flags : in STD_LOGIC_VECTOR (3 downto 0);
        enable_jump : in STD_LOGIC;
        jump_info : in STD_LOGIC_VECTOR (13 downto 0);
        PCm1 : in STD_LOGIC_VECTOR (15 downto 0);
        RB : in STD_LOGIC_VECTOR (15 downto 0);
        output_address : out STD_LOGIC_VECTOR (15 downto 0));
   end component Flags;

component ProgramCounter is
   port(next_PC : in STD_LOGIC_VECTOR (15 downto 0);
        enable_pc : in STD_LOGIC;
        PC : out STD_LOGIC_VECTOR (15 downto 0);
        PCm1 : out STD_LOGIC_VECTOR (15 downto 0);
        clk : in STD_LOGIC);
end component ProgramCounter;

component IF_Regs is
   Port (Next_PC_in : in STD_LOGIC_VECTOR (15 downto 0);
         Next_PC_out : out STD_LOGIC_VECTOR (15 downto 0);
         clk : in STD_LOGIC;
         enable : in STD_LOGIC);
end component IF_Regs;

component ID_RF_Regs is
   Port (Current_PC_in : in STD_LOGIC_VECTOR (15 downto 0);
         Current_PC_out : out STD_LOGIC_VECTOR (15 downto 0);
         Next_PC_in : in STD_LOGIC_VECTOR (15 downto 0);
         Next_PC_out : out STD_LOGIC_VECTOR (15 downto 0);
         IMM_in : in STD_LOGIC_VECTOR (15 downto 0);
         IMM_out : out STD_LOGIC_VECTOR (15 downto 0);
         OPCODE_in : in STD_LOGIC_VECTOR (4 downto 0);
         OPCODE_out : out STD_LOGIC_VECTOR (4 downto 0);
         UNIT_SEL_in : in STD_LOGIC_VECTOR (1 downto 0);
         UNIT_SEL_out : out STD_LOGIC_VECTOR (1 downto 0);
         DA_in : in STD_LOGIC_VECTOR (2 downto 0);
         DA_out : out STD_LOGIC_VECTOR (2 downto 0);
         A_in : in STD_LOGIC_VECTOR (15 downto 0);
```

```
        A_out : out STD_LOGIC_VECTOR (15 downto 0);
        B_in : in STD_LOGIC_VECTOR (15 downto 0);
        B_out : out STD_LOGIC_VECTOR (15 downto 0);
        WE_in : in STD_LOGIC;
        WE_out : out STD_LOGIC;
        SEL_OUT_in : in STD_LOGIC_VECTOR (1 downto 0);
        SEL_OUT_out : out STD_LOGIC_VECTOR (1 downto 0);
        MEM_WRITE_in : in STD_LOGIC;
        MEM_WRITE_out : out STD_LOGIC;
        jump_opcode_in : in STD_LOGIC_VECTOR (13 downto 0);
        jump_opcode_out : out STD_LOGIC_VECTOR (13 downto 0);
        flags_en_in : in STD_LOGIC_VECTOR (3 downto 0);
        flags_en_out : out STD_LOGIC_VECTOR (3 downto 0);
        enable_jump_in : in STD_LOGIC;
        enable_jump_out : out STD_LOGIC;
        clk : in STD_LOGIC;
        enable : in STD_LOGIC);
end component ID_RF_Regs;

component EX_MEM_Regs is
   Port (Next_PC_in : in STD_LOGIC_VECTOR (15 downto 0);
      Next_PC_out : out STD_LOGIC_VECTOR (15 downto 0);
      C_in : in STD_LOGIC_VECTOR (15 downto 0);
      C_out : out STD_LOGIC_VECTOR (15 downto 0);
      WE_in : in STD_LOGIC;
      WE_out : out STD_LOGIC;
      S_in : in STD_LOGIC_VECTOR (15 downto 0);
      S_out : out STD_LOGIC_VECTOR (15 downto 0);
      DA_in : in STD_LOGIC_VECTOR (2 downto 0);
      DA_out : out STD_LOGIC_VECTOR (2 downto 0);
      MUX_WB_in : in STD_LOGIC_VECTOR(1 downto 0);
      MUX_WB_out : out STD_LOGIC_VECTOR(1 downto 0);
      clk : in STD_LOGIC;
      enable : in STD_LOGIC
      );
end component EX_MEM_regs;

component FSM_Regs is
   Port( START : in STD_LOGIC;
      ENABLE_IF : out STD_LOGIC;
      ENABLE_IF_RF : out STD_LOGIC;
      ENABLE_EX_MEM : out STD_LOGIC;
```

```
    ENABLE_PC : out STD_LOGIC;
    PRINT : in STD_LOGIC;
    CLK : in STD_LOGIC
);
end component FSM_Regs;

signal opcde : STD_LOGIC_VECTOR(4 downto 0);
signal opcde_2 : STD_LOGIC_VECTOR(4 downto 0);
signal instr : STD_LOGIC_VECTOR (15 downto 0);
signal instr_2 : STD_LOGIC_VECTOR (15 downto 0);
signal imediato : STD_LOGIC_VECTOR(15 downto 0);
signal imediato_2 : STD_LOGIC_VECTOR(15 downto 0);
signal sel_unid : STD_LOGIC_VECTOR(1 downto 0);
signal sel_unid_2 : STD_LOGIC_VECTOR(1 downto 0);
signal da1,aa1,ba1 : STD_LOGIC_VECTOR(2 downto 0);
signal da1_2 : STD_LOGIC_VECTOR(2 downto 0);
signal da1_3 : STD_LOGIC_VECTOR(2 downto 0);
signal wenable : STD_LOGIC;
signal wenable_2 : STD_LOGIC;
signal wenable_3 : STD_LOGIC;
signal a_v,b_v : STD_LOGIC_VECTOR(15 downto 0);
signal a_v_2,b_v_2 : STD_LOGIC_VECTOR(15 downto 0);
signal writedata : STD_LOGIC_VECTOR(15 downto 0);
signal Flags_alu : STD_LOGIC_VECTOR(3 downto 0);
signal consts : STD_LOGIC_VECTOR(15 downto 0);
signal consts_2 : STD_LOGIC_VECTOR(15 downto 0);
signal Alu_S : STD_LOGIC_VECTOR(15 downto 0);
signal Alu_S_2 : STD_LOGIC_VECTOR(15 downto 0);
signal MUXWB : STD_LOGIC_VECTOR(1 downto 0);
signal MUXWB_2 : STD_LOGIC_VECTOR(1 downto 0);
signal MUXWB_3 : STD_LOGIC_VECTOR(1 downto 0);
signal mem_dados : STD_LOGIC_VECTOR(15 downto 0);
signal mem_dados_2 : STD_LOGIC_VECTOR(15 downto 0);
signal MEM_EN : STD_LOGIC;
signal MEM_EN_2 : STD_LOGIC;
signal jump_opc : STD_LOGIC_VECTOR(13 downto 0);
signal jump_opc_2 : STD_LOGIC_VECTOR(13 downto 0);
signal PCm1 : STD_LOGIC_VECTOR(15 downto 0);
signal PCm1_2 : STD_LOGIC_VECTOR(15 downto 0);
signal PCm1_3 : STD_LOGIC_VECTOR(15 downto 0);
signal PCm1_4 : STD_LOGIC_VECTOR(15 downto 0); --alterado
signal addr : STD_LOGIC_VECTOR(15 downto 0);
```

```
signal fl_en : STD_LOGIC_VECTOR(3 downto 0);
signal fl_en_2 : STD_LOGIC_VECTOR(3 downto 0);
signal jpen : STD_LOGIC;
signal jpen_2 : STD_LOGIC;
signal IFaddr : STD_LOGIC_VECTOR(15 downto 0);
signal IFaddr_2 : STD_LOGIC_VECTOR(15 downto 0);
signal IFaddr_3 : STD_LOGIC_VECTOR(15 downto 0);
--alterado
signal en_lvl1 : STD_LOGIC;
signal en_lvl2 : STD_LOGIC;
signal en_lvl3 : STD_LOGIC;
signal en_pc : STD_LOGIC;
signal ram_print : STD_LOGIC;

begin

Decoder_Inst: decoder port map(
   INSTRUCTION => instr,
   OPCODE => opcde,
   IMM => imediato,
   UNIT_SEL => sel_unid,
   DA => da1,
   AA => aa1,
   BA => ba1,
   WE => wenable,
   SEL_OUT => MUXWB,
   MEM_WRITE => MEM_EN,
   jump_opcode => jump_opc,
   flags_en => fl_en,
   enable_jump => jpen
);

Constante : constantes port map(
   Const => imediato_2,
   C_in => a_v,
   C => consts,
   OP_SEL => sel_unid_2
);

RegFile : registerfile port map(
   AA => aa1,
   A => a_v,
```

```
  BA => ba1,
  B => b_v,
  DA => da1_3,
  WE => wenable_3,
  DATA => writedata,
  clk => CLK
);

ALU_OP : alu port map(
  OP_SEL => opcde_2,
  A => a_v_2,
  B => b_v_2,
  S => Alu_S,
  Flags => Flags_alu
);

WB_Mux : writeback_mux port map(
  ALU => Alu_S_2,
  MEM => mem_dados,
  Consts => consts_2,
  PC => PCm1_4,   -- alterado
  Sel_WB => MUXWB_3, -- change  -->  MUXWB
  C => writedata
);
RAM : rom_instrc port map(
  clk => CLK,
  we =>MEM_EN_2,
  addr_instr => IFaddr,
  addr_dados => a_v_2,
  din => b_v_2,
  dout_instr => instr,
  dout_dados => mem_dados,
  print => ram_print
);

Flags_Jumps : Flags port map(
  clk => CLK,
  Flags => Flags_alu,
  enable_flags => fl_en_2,
  enable_jump => jpen_2,
  jump_info => jump_opc_2,
  PCm1 => PCm1_3,
```

```
    RB => b_v,
    output_address => addr
);

PC : ProgramCounter port map(
    next_PC => addr,
    enable_pc => en_pc,
    PC => IFaddr,
    PCm1 => PCm1,
    clk => CLK
);

IF_Registers : IF_Regs port map(
    Next_PC_in => PCm1,
    Next_PC_out => PCm1_2,
    clk => CLK,
    enable => en_lvl1
);

ID_RF_Registers : ID_RF_Regs port map(
    Current_PC_in => IFaddr_2,
    Current_PC_out => IFaddr_3,
    Next_PC_in => PCm1_2,
    Next_PC_out => PCm1_3,
    IMM_in => imediato,
    IMM_out => imediato_2,
    OPCODE_in =>opcde,
    OPCODE_out =>opcde_2,
    UNIT_SEL_in => sel_unid,
    UNIT_SEL_out => sel_unid_2,
    DA_in => da1,
    DA_out => da1_2,
    A_in => a_v,
    A_out => a_v_2,
    B_in => b_v,
    B_out => b_v_2,
    WE_in => wenable,
    WE_out => wenable_2,
    SEL_OUT_in => MUXWB,
    SEL_OUT_out => MUXWB_2,
    MEM_WRITE_in => MEM_EN,
    MEM_WRITE_out => MEM_EN_2,
```

```
      jump_opcode_in => jump_opc,
      jump_opcode_out => jump_opc_2,
      flags_en_in => fl_en,
      flags_en_out => fl_en_2,
      enable_jump_in => jpen,
      enable_jump_out => jpen_2,
      clk => CLK,
      enable => en_lvl2
   );

   EX_MEM_Registers : EX_MEM_Regs port map(
      WE_in => wenable_2,
      WE_out => wenable_3,
      Next_PC_in => PCm1_3, --alterado
      Next_PC_out => PCm1_4, --alterado
      C_in => consts,
      C_out => consts_2,
      S_in => Alu_S,
      S_out => Alu_S_2,
      DA_in => da1_2,
      DA_out => da1_3,
      MUX_WB_in => MUXWB_2,
      MUX_WB_out => MUXWB_3,
      clk => CLK,
      enable => en_lvl3
   );

   FSM : FSM_Regs port map(
      START => TEST1,
      PRINT => ram_print,
      ENABLE_IF => en_lvl1,
      ENABLE_IF_RF => en_lvl2,
      ENABLE_EX_MEM => en_lvl3,
      ENABLE_PC => en_pc,
      CLK => CLK
   );

   TEST <= writedata;
   MEMORY <= instr;
   INST_NB <= IFaddr;
   PRINT <= ram_print;
end Behavioral;
```