



INSTITUTO SUPERIOR TÉCNICO

MESTRADO INTEGRADO EM ENGENHARIA ELECTROTÉCNICA E DE
COMPUTADORES

ARQUITECTURAS AVANÇADAS DE COMPUTADORES

**Descrição do processador μ Risc a funcionar em
*pipeline***

| | |
|-------------------------------|-----------|
| Guilherme Branco Teixeira | n.º 70214 |
| Maria Margarida Dias dos Reis | n.º 73099 |
| Nuno Miguel Rodrigues Machado | n.º 74236 |

Lisboa, 10 de Maio 2015

Índice

| | | |
|----------|--|----------|
| 1 | Introdução | 1 |
| 2 | Conflitos associados a uma arquitectura <i>pipeline</i> | 1 |
| 2.1 | Conflitos estruturais | 1 |
| 2.2 | Conflitos de dados | 1 |
| 2.3 | Conflitos de controlo | 1 |
| 3 | Métodos de resolução de conflitos | 1 |
| 3.1 | Conflitos de dados | 1 |
| 3.2 | Conflitos de controlo | 1 |
| 4 | Estrutura do processador | 2 |
| 5 | Testes de <i>performance</i> | 2 |
| 6 | Conclusões | 3 |
| 7 | Anexos | 4 |
| 7.1 | Código do andar de IF | 4 |
| 7.2 | Código do andar de ID e OF | 8 |
| 7.3 | Código do andar de EX e MEM | 17 |
| 7.4 | Código do andar de WB | 25 |
| 7.5 | Código da memória ROM | 27 |
| 7.6 | Código da memória RAM | 28 |
| 7.7 | Código da BTB | 29 |
| 7.8 | Código da máquina de estados | 30 |

1 Introdução

Com este trabalho laboratorial pretende-se projectar um processador μ Risc com funcionamento em *pipeline*. O processador possui 4 andares de *pipelining*: no primeiro andar é feito o *instruction fetch* (IF), no segundo andar é feito o *instruction decode* (ID) e o *operand fetch* (OF), no terceiro andar são executadas operações da ALU (EX) e de acesso à memória de dados (MEM) e, por fim, no quarto é feita a escrita no banco de registos, o *write back* (WB). Com o funcionamento em *pipelining* podem ocorrer dois tipos de conflitos - de dados (*data hazards*) e de controlo (*control hazards*).

2 Conflitos associados a uma arquitectura *pipeline*

2.1 Conflitos estruturais

2.2 Conflitos de dados

2.3 Conflitos de controlo

3 Métodos de resolução de conflitos

Em primeiro lugar apresentam-se os métodos e técnicas de resolução dos conflitos de controlo e dados. Em segundo lugar quais as técnicas que foram de facto utilizadas.

3.1 Conflitos de dados

Conflito que surge quando uma instrução depende dos resultados de uma instrução anterior, de forma a afectar o resultado obtido pela linha de processamento.

- **Solução 1:** Bloqueio dos andares do *pipeline*, *stall*, até que os dados correctos estejam disponíveis;
- **Solução 2:** Se o dado correcto existir algures no *pipeline*, estabelece-se um *bypass* para o andar correcto, aplicando a técnica de *forwarding*;
- **Solução 3:** Escalonar/reordenar as instruções, se a ordenação for feita pelo compilador, tem-se um escalonamento estático, se for feita pelo *hardware*, escalonamento dinâmico;

3.2 Conflitos de controlo

Um conflito de controlo surge quando uma instrução de controlo condicional depende dos resultados de uma instrução anterior, de uma forma a impedir uma predição correcta. Analisando as soluções

estrutura ideal do relatório:
1: referir no geral que conflitos há. 2: referir quais os conflitos que temos neste processador. 3: referir como os resolvemos. 4: comparar as arquitecturas que testámos. 5: entregar antes da meia-noite LooLoL

referir nesta secção explicações teóricas dos conflitos que há, e referir que conflitos há de facto no nosso processador

aqui por exemplo nos so vamos ter RAW, mas temos de explicar todos os que há

nao impede a prediccao

apresentadas verificou-se que o escalonamento estático e dinâmico não seria a solução desejada devido a complexidade para um processador de 4 andares comparativamente às outras anteriores. Ponderou-se inicialmente a utilização de *stalls* devido à facilidade de implementação mas devido ao inconveniente de reduzir o número médio de instruções por ciclo, optou-se pela segunda solução de forma a aumentar o número médio de instruções por ciclo como também a complexidade de implementação.

- **Solução 1:** BTB, uma tabela que contém informação sobre os saltos de forma a prever se estes são *taken* ou *not-taken*, previsão esta que é realizada por uma BPB (composta por 1 ou 2 bits), diminuindo assim o número de ciclos desperdiçados em instruções do tipo controlo;
- **Solução 2:** *forwarding* de flags, após se obter o resultado necessário para a predição, estabelece-se um bypass para verificar a condição do salto;

explicar o que é uma BTB, explicar BPB

4 Estrutura do processador

5 Testes de *performance*

Tabela 1: *Performance* obtida para os diversos testes com o processador demonstrado na aula.

| Processador Demonstrado | Frequência [MHz] | Número de Ciclos de Execução | Número de Predições Falhadas | Tempo de Excução [μs] |
|-------------------------|------------------|------------------------------|------------------------------|-----------------------|
| Teste #1 | 183,169 | 57 | 0 | 0,3112 |
| Teste #2 | 183,169 | 2358 | 108 | 12,8734 |
| Teste #3 | 183,169 | 1058 | 122 | 5,7761 |

Tabela 2: Diversas topologias do processador que foram testadas.

| | |
|----------------|--|
| Processador #1 | com <i>forwarding</i> de dados e BTB |
| Processador #2 | sem <i>forwarding</i> de dados, com NOPS e com BTB |
| Processador #3 | com <i>forwarding</i> de dados e sem BTB |
| Processador #4 | sem <i>forwarding</i> de dados e sem BTB |

Tabela 3: Resultados obtidos para o teste#1.

| Teste #1 | Frequência [MHz] | Número de Ciclos de Execução | Número de Predições Falhadas | Tempo de Excução [μs] |
|----------------|------------------|------------------------------|------------------------------|-----------------------|
| Processador #1 | 211,338 | 57 | 0 | 0,2697 |
| Processador #2 | 211,338 | 114 | 0 | 0,5394 |
| Processador #3 | 285,608 | 57 | 0 | 0,1996 |
| Processador #4 | 285,608 | 114 | 0 | 0,3991 |

Tabela 4: Resultados obtidos para o teste#2.

| Teste #2 | Frequência [MHz] | Número de Ciclos de Execução | Número de Predições Falhadas | Tempo de Excução [μs] |
|----------------|------------------|------------------------------|------------------------------|-----------------------|
| Processador #1 | 211,338 | 2358 | 108 | 11,1575 |
| Processador #2 | 211,338 | 3044 | 108 | 14,4035 |
| Processador #3 | 285,608 | 2705 | 463 | 9,4710 |
| Processador #4 | 285,608 | 3323 | 463 | 11,6348 |

Tabela 5: Resultados obtidos para o teste#3.

| Teste #3 | Frequência [MHz] | Número de Ciclos de Execução | Número de Predições Falhadas | Tempo de Excução [μs] |
|----------------|------------------|------------------------------|------------------------------|-----------------------|
| Processador #1 | 211,338 | 1058 | 122 | 5,0062 |
| Processador #2 | 211,338 | 1582 | 122 | 7,4856 |
| Processador #3 | 285,608 | 1123 | 187 | 3,9320 |
| Processador #4 | 285,608 | 1647 | 187 | 5,7666 |

6 Conclusões

7 Anexos

7.1 Código do andar de IF

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.STD_LOGIC_ARITH.ALL;
4 use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6 entity InF is
7 port(
8     -- input
9     clk, rst           : in std_logic;
10    en_if               : in std_logic;
11    reg_pc_IN           : in std_logic_vector(11 downto 0);    -- PC realimentado
12
13    pc_branch           : in std_logic_vector(11 downto 0);    -- acrescentar
14    Jump_BTBT           : in std_logic_vector(11 downto 0);
15    OUT_BTBT            : in std_logic_vector(16 downto 0);
16    act_BTBT            : in std_logic;
17    Conflit_DADOS       : in std_logic;
18    en_Jump_FII         : in std_logic;
19    en_Jump_Incond       : in std_logic;
20    count_IN            : in std_logic_vector(15 downto 0);
21    Jump_Incon          : in std_logic_vector(11 downto 0);
22    out_ROM             : in std_logic_vector(15 downto 0);
23
24    -- output
25    addr_BTBT_A         : out std_logic_vector(8 downto 0);
26    addr_BTBT_B         : out std_logic_vector(8 downto 0);
27    IN_BTBT             : out std_logic_vector(16 downto 0);
28    we_BTBT             : out std_logic;
29    reg_PCMEM_OUT       : out std_logic_vector(11 downto 0);    -- PC + 1
30    reg_pc_OUT          : out std_logic_vector(11 downto 0);    -- PC realimentado
31    addr               : out std_logic_vector(11 downto 0);    -- PC + 1
32    reg_IfOut_ROM       : out std_logic_vector(15 downto 0);
33    reg_IfOut_MUX_BrPred : out std_logic;
34    count_OUT           : out std_logic_vector(15 downto 0);
35    reg_IfOut_PC        : out std_logic_vector(11 downto 0)    -- registo entre
                        andares
36 );
37 end InF;
38
39 architecture Behavioral of InF is
40
41     -----
42     ----- Aux Signals -----
```

```

43 -----
44 signal aux_pc_add_1      : std_logic_vector(11 downto 0) := (others => '0');
45 signal aux_count        : std_logic_vector(15 downto 0) := (others => '0');
46 signal signal_count     : std_logic := '0';
47 signal aux_saida_mux    : std_logic_vector(11 downto 0) := (others => '0');
48 signal aux_reg_pc       : std_logic_vector(11 downto 0) := (others => '0');
49 signal aux_reg_pc_backup : std_logic_vector(11 downto 0) := (others => '0');
50 signal aux_out_ROM      : std_logic_vector(15 downto 0) := (others => '0');
51 signal Aux_IN_BTBT      : std_logic_vector(16 downto 0) := (others => '0');
52 signal aux_pc_OUT       : std_logic_vector(11 downto 0) := (others => '0');
53 ----Sinais para o BTB
54 signal MSB_PC_TAG       : std_logic := '0';
55 signal MUX_BrPred       : std_logic := '0';
56 signal aux_we_BTBT      : std_logic := '0';
57 signal Addr_BTBT_Act    : std_logic_vector( 8 downto 0) := (others => '0');
58 signal MSB_BTBT         : std_logic_vector( 2 downto 0) := (others => '0');
59 signal PC_BTBT          : std_logic_vector( 11 downto 0) := (others => '0');
60 signal Jump_From_BTBT   : std_logic_vector( 11 downto 0) := (others => '0');
61 signal Prediction_Bit   : std_logic := '0';
62 signal Validate_Bit     : std_logic := '0';
63 signal Clean_BTBT       : std_logic := '0';
64 signal Insert_BTBT      : std_logic := '0';
65 signal jump_equal_pc    : std_logic := '0';
66 signal MUX_NEXTPC       : std_logic_vector( 1 downto 0) := (others => '0');
67 signal aux_erros        : std_logic_vector(11 downto 0) := (others => '0');
68
69 -----
70 ----- Constantes -----
71 -----
72 constant one            : std_logic_vector(11 downto 0) := "000000000001";
73 constant zeros_16       : std_logic_vector(15 downto 0) := (others => '0');
74 constant zeros_12       : std_logic_vector(11 downto 0) := (others => '0');
75
76 begin
77
78 -----
79 ----- BTB -----
80 -----
81
82 --Leitura da BTB
83 addr_BTBT_B      <= aux_pc_add_1(8 downto 0);--confirmar
84 MSB_BTBT         <= OUT_BTBT(16 downto 14);
85 Jump_From_BTBT   <= OUT_BTBT(13 downto 2);
86 Prediction_Bit   <= OUT_BTBT(1);
87 Validate_Bit     <= OUT_BTBT(0);
88
89

```

```

90  --Atualizacao da BTB
91  addr_BTBA_A    <=  pc_branch(8 downto 0);
92  IN_BTBA        <=  pc_branch(11 downto 9)&Jump_BTBA&act_BTBA&'1';
93  we_BTBA        <=  Insert_BTBA or Clean_BTBA;
94
95  --Mux Not Taken='0'/ Taken='1'
96  MSB_PC_TAG     <=  ((aux_pc_add_1(11) xnor MSB_BTBA(2)) and (aux_pc_add_1(10) xnor
97                    MSB_BTBA(1)) and (aux_pc_add_1(9) xnor MSB_BTBA(0))) and Validate_Bit;
98
99  MUX_BrPred     <=  Prediction_Bit when MSB_PC_TAG = '1' else '0';
100
101  PC_BTBA        <=  Jump_From_BTBA when MUX_BrPred = '1' else
102                    reg_pc_IN + one;
103
104  MUX_NEXTPC(1) <=  Conflit_DADOS AND ( (NOT(en_Jump_Incond) AND NOT(jump_equal_pc)
105                    AND ((Act_BTBA AND NOT(en_Jump_FII)) OR en_Jump_FII)) OR (jump_equal_pc AND NOT(
106                    Act_BTBA) AND NOT(en_Jump_FII) AND NOT(en_Jump_Incond)) );
107
108  MUX_NEXTPC(0) <=  Conflit_DADOS AND (en_Jump_Incond OR (jump_equal_pc AND NOT(Act_BTBA
109                    ) AND NOT(en_Jump_FII) AND NOT(en_Jump_Incond)));
110
111  aux_saida_mux <=  PC_BTBA                when MUX_NEXTPC = "00" else
112                    Jump_Incon             when MUX_NEXTPC = "01" else
113                    Jump_BTBA              when MUX_NEXTPC = "10" else
114                    aux_reg_pc_backup;
115
116  --aux_saida_mux <= reg_pc_IN ;
117
118  -----
119  ----- Registo PC -----
120  -----
121
122  process (clk, rst, MUX_BrPred)
123  begin
124      if clk'event and clk = '1' then
125          if rst = '1' then
126              aux_reg_pc <= zeros_12;
127          elsif MUX_BrPred = '1' then
128              aux_reg_pc_backup <= reg_pc_IN + one;
129              aux_reg_pc <= aux_saida_mux;
130          else
131              aux_reg_pc <= aux_saida_mux;
132          end if;
133      end if;
134  end process;
135
136  process (clk, rst, count_IN)
137  begin
138      if clk'event and clk = '1' then

```



```

133     if rst = '1' then
134         aux_count <= zeros_16;
135     elsif signal_count = '0' then
136         aux_count <= count_IN + one;
137     end if;
138 end if;
139 end process;
140
141 aux_pc_add_1 <= aux_reg_pc;
142
143 count_OUT <= aux_count;
144
145 --Controlo de saltos em relacao a BTB
146
147 jump_equal_pc <= Conflit_DADOS when Jump_BTBT = aux_pc_add_1 else '0';
148
149 -- CASOS POSSIVEIS
150 -- jump_equal_pc = 0 e act_BTBT = 1 tem que carregar o salto e actualizar btb
151 -- jump_equal_pc = 1 e act_BTBT = 0 tem que carregar o pc antigo e limpar btb
152
153 Clean_BTBT <= jump_equal_pc AND Not(act_BTBT) AND NOT(en_Jump_FII) AND NOT(
    en_Jump_Incond);
154 Insert_BTBT <= (NOT(jump_equal_pc) AND act_BTBT AND NOT(en_Jump_FII) AND NOT(
    en_Jump_Incond)) OR (NOT(jump_equal_pc) AND en_Jump_FII AND NOT(en_Jump_Incond));
155
156
157 signal_count <= '1' when out_ROM = x"2fff" else '0' ;
158 -----
159 ----- Registo de Instrucoes (IR) -----
160 -----
161 process (clk, rst)
162 begin
163     if clk'event and clk = '1' then
164         if rst = '1' then
165             aux_out_ROM <= zeros_16;
166             aux_pc_OUT <= zeros_12;
167         elsif en_if = '1' then
168             if (Conflit_DADOS = '1') and (Clean_BTBT = '1' or Insert_BTBT = '1' or
    en_Jump_Incond = '1') then
169                 aux_out_ROM <= x"0000";
170                 aux_erros <= aux_erros + one;
171                 aux_pc_OUT <= aux_pc_add_1;
172             else
173                 aux_out_ROM <= out_ROM;
174                 aux_pc_OUT <= aux_pc_add_1 + one; --Modificado em relacao ao
    apresentado no Laboratorio, foi acrescentado + one;
175             end if;

```

```

176         end if;
177     end if;
178 end process;
179
180 -----
181 ----- Exit -----
182 -----
183 reg_PCMEM_OUT    <= aux_reg_pc;
184 addr             <= aux_reg_pc;
185 reg_pc_OUT       <= aux_pc_add_1;
186 reg_IfOut_ROM    <= aux_out_ROM;
187 reg_IfOut_PC     <= aux_pc_OUT;
188
189 end Behavioral;

```

7.2 Código do andar de ID e OF

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.STD_LOGIC_ARITH.ALL;
4 use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6 entity IDeOF is
7     port(
8         -- input
9         clk, rst          : in std_logic;
10        reg_IfOut_PC       : in std_logic_vector(11 downto 0);    -- registo entre andares
11        reg_IfOut_ROM      : in std_logic_vector(15 downto 0);
12        FLAGS_IN           : in std_logic_vector(3 downto 0);
13        Forw_FLAGS_IN      : in std_logic_vector(3 downto 0);
14        ADD_RWC_EXMEM      : in std_logic_vector(2 downto 0);
15        ADD_RWC_WB         : in std_logic_vector(2 downto 0);
16        ovWE_EXMEM         : in std_logic;
17        ovWE_WB            : in std_logic;
18        en_idOF            : in std_logic;
19        R0                 : in std_logic_vector(15 downto 0);
20        R1                 : in std_logic_vector(15 downto 0);
21        R2                 : in std_logic_vector(15 downto 0);
22        R3                 : in std_logic_vector(15 downto 0);
23        R4                 : in std_logic_vector(15 downto 0);
24        R5                 : in std_logic_vector(15 downto 0);
25        R6                 : in std_logic_vector(15 downto 0);
26        R7                 : in std_logic_vector(15 downto 0);
27        --Forwarding
28        Forw_EXMEN         : in std_logic_vector(15 downto 0);
29        Forw_WB            : in std_logic_vector(15 downto 0);
30
31        -- output

```

```

32     en_Jump_FII           : out std_logic;
33     en_Jump_Incond        : out std_logic;
34     Jump_Incon            : out std_logic_vector(11 downto 0);
35     pc_branch             : out std_logic_vector(11 downto 0);    -- acrescentar
36     Jump_BTBT             : out std_logic_vector(11 downto 0);
37     act_BTBT              : out std_logic;
38     Conflit_ON            : out std_logic;
39     Conflit_DADOS         : out std_logic;
40     pc_add_jump           : out std_logic_vector(11 downto 0);
41     reg_IDOF_OUT_WERAM     : out std_logic;
42     reg_IDOF_OUT_ALUOPER   : out std_logic_vector(4 downto 0);
43     reg_IDOF_OUT_bit15     : out std_logic;
44     reg_IDOF_OUT_bit14     : out std_logic;
45     reg_IDOF_OUT_OperA     : out std_logic_vector(15 downto 0);
46     reg_IDOF_OUT_OperB     : out std_logic_vector(15 downto 0);
47     reg_IDOF_OUT_ALUvsMEM  : out std_logic;
48     reg_IDOF_OUT_ovWE      : out std_logic;
49     reg_IDOF_OUT_AddRWC    : out std_logic_vector(2 downto 0);
50     reg_IDOF_OUT_PCadd1    : out std_logic_vector(11 downto 0);
51     reg_IDOF_OUT_SelMuxWB  : out std_logic_vector(1 downto 0);
52     reg_IDOF_OUT_MuxConst  : out std_logic_vector(15 downto 0)
53 );
54 end IDeOF;
55
56 architecture Behavioral of IDeOF is
57
58 -----
59 ----- Aux Signals -----
60 -----
61 signal inst_IN           : std_logic_vector(15 downto 0) := (others => '0');
62 signal aux_ADD_RWC       : std_logic_vector(2 downto 0) := (others => '0');
63 signal aux_ADD_RA        : std_logic_vector(2 downto 0) := (others => '0');
64 signal aux_ADD_RB        : std_logic_vector(2 downto 0) := (others => '0');
65 signal aux_ADD_RA_C      : std_logic_vector(2 downto 0) := (others => '0');
66 signal aux_ALU_OPER      : std_logic_vector(4 downto 0) := (others => '0');
67 signal aux_CONS_FI_11B   : std_logic_vector(10 downto 0) := (others => '0');
68 signal aux_CONS_FII_8B   : std_logic_vector(7 downto 0) := (others => '0');
69 signal aux_active_FLAGTEST : std_logic := '0';
70 signal aux_FLAGMUX       : std_logic := '0';
71 signal aux_FLAGMUX_C_For : std_logic := '0';
72 signal aux_FLAGMUX_S_For : std_logic := '0';
73 signal aux_TRANS_OP      : std_logic_vector(1 downto 0) := (others => '0');
74 signal aux_TRANS_FI_DES  : std_logic_vector(11 downto 0) := (others => '0');
75 signal aux_TRANS_FII_DES : std_logic_vector(11 downto 0) := (others => '0');
76 signal aux_TRANS_DES     : std_logic_vector(11 downto 0) := (others => '0');
77 signal aux_TRANS_FIII_R  : std_logic := '0';
78 signal aux_JUMPS_active  : std_logic := '0';

```

```

79 signal aux_JUMPS_MUX_WB      : std_logic := '0';
80 signal aux_pc_add_one       : std_logic_vector(11 downto 0) := (others => '0');
81 signal aux_TEST_NOP         : std_logic := '0';
82 signal aux_FLAGTEST         : std_logic := '0';
83 signal aux_FLAGTEST_MUXPC   : std_logic := '0';
84 signal JUMP_MUXWB_OUT       : std_logic := '0';
85 signal ALU_vs_MEM           : std_logic := '0';
86 signal Aux_Conflit_DADOS    : std_logic := '0';
87 signal RA_C                 : std_logic_vector(15 downto 0) := (others => '0');
88 signal RB                   : std_logic_vector(15 downto 0) := (others => '0');
89 signal oper_A               : std_logic_vector(15 downto 0) := (others => '0'); -- operando
    A para a ALU
90 signal oper_B               : std_logic_vector(15 downto 0) := (others => '0'); -- operando
    B para a ALU
91 signal const11              : std_logic_vector(15 downto 0) := (others => '0'); --
    operando B para a ALU
92 signal lcl                  : std_logic_vector(15 downto 0) := (others => '0'); -- operando
    B para a ALU
93 signal lch                  : std_logic_vector(15 downto 0) := (others => '0'); -- operando
    B para a ALU
94 signal select_mux_constantes : std_logic_vector(1 downto 0) := (others => '0');
95 signal out_mux_constantes    : std_logic_vector(15 downto 0) := (others => '0'); --
    operando para carregamento de constantes
96 signal bit14                : std_logic := '0'; -- sinal de selecao para MUX
    entre operacao da ALU e operacao de carregamento de constantes
97 signal bit15                : std_logic := '0';
98 signal WE_RAM               : std_logic := '0';
99 signal TRANS_FI_COND_IN     : std_logic_vector(3 downto 0) := (others => '0');
100 signal pc_add1              : std_logic_vector(11 downto 0) := (others => '0');
101 signal mux_jump_cond         : std_logic_vector(1 downto 0) := (others => '0');
102 signal aux_saida_mux        : std_logic_vector(11 downto 0) := (others => '0');
103 signal ALU_e_MEM            : std_logic := '0';
104 signal soMEM                 : std_logic := '0';
105 signal soALU                : std_logic := '0';
106 signal loadMEM              : std_logic := '0';
107 signal controlo              : std_logic := '0';
108 signal controloJump         : std_logic := '0';
109 signal ovWE                  : std_logic := '0';
110 signal aux_sel_bit1          : std_logic := '0';
111 signal aux_sel_bit0          : std_logic := '0';
112 signal sel_mux_WB           : std_logic_vector(1 downto 0) := (others => '0');
113 signal aux_reg_IDOF_OUT_WERAM : std_logic := '0';
114 signal aux_reg_IDOF_OUT_ALUOPER : std_logic_vector(4 downto 0) := (others => '0');
115 signal aux_reg_IDOF_OUT_bit15 : std_logic := '0';
116 signal aux_reg_IDOF_OUT_bit14 : std_logic := '0';
117 signal aux_reg_IDOF_OUT_OperA : std_logic_vector(15 downto 0) := (others => '0');
118 signal aux_reg_IDOF_OUT_OperB : std_logic_vector(15 downto 0) := (others => '0');

```

```

119 signal aux_reg_IDOF_OUT_ALUvsMEM : std_logic := '0';
120 signal aux_reg_IDOF_OUT_ovWE : std_logic := '0';
121 signal aux_reg_IDOF_OUT_AddRWC : std_logic_vector(2 downto 0) := (others => '0');
122 signal aux_reg_IDOF_OUT_PCadd1 : std_logic_vector(11 downto 0) := (others => '0');
123 signal aux_reg_IDOF_OUT_SelMuxWB : std_logic_vector(1 downto 0) := (others => '0');
124 signal aux_reg_IDOF_OUT_MuxConst : std_logic_vector(15 downto 0) := (others => '0');
125 --Conflito
126 signal RA_C_EXMEM_CONFLITO : std_logic := '0';
127 signal RA_C_WB_CONFLITO : std_logic := '0';
128 signal RB_EXMEM_CONFLITO : std_logic := '0';
129 signal RB_WB_CONFLITO : std_logic := '0';
130 signal aux_Conflit_ON : std_logic := '0';
131 signal Conflit_EXMEN_RA_C_ON : std_logic := '0';
132 signal Conflit_WB_RA_C_ON : std_logic := '0';
133 signal Conflit_EXMEN_RB_ON : std_logic := '0';
134 signal Conflit_WB_RB_ON : std_logic := '0';
135 signal mux_RA : std_logic_vector(1 downto 0) := (others => '0');
136 signal mux_RB : std_logic_vector(1 downto 0) := (others => '0');
137 signal aux_oper_A : std_logic_vector(15 downto 0) := (others => '0');
138
139
140
141 -----
142 ----- Constantes -----
143 -----
144 constant one : std_logic_vector(11 downto 0) := "000000000001" ;
145 constant zero_12 : std_logic_vector(11 downto 0) := (others => '0');
146 constant zero_16 : std_logic_vector(15 downto 0) := (others => '0');
147
148 begin
149
150 inst_IN <= reg_IfOut_ROM;
151 pc_add1 <= reg_IfOut_PC; --Modificado em relacao ao apresentado no Laboratorio, foir
    retirado + one;
152
153 aux_ADD_RWC <= inst_IN(13 downto 11);
154 aux_ADD_RA <= inst_IN(5 downto 3);
155 aux_ADD_RB <= inst_IN(2 downto 0);
156
157 aux_ADD_RA_C <= aux_ADD_RWC when (inst_IN(15) and inst_IN(14)) = '1' else
    aux_ADD_RA;
158
159
160 -----
161 ----- Id conflito -----
162 -----
163

```

```

164 RA_C_EXMEM_CONFLITO    <= ((ADD_RWC_EXMEM(2) XNOR aux_ADD_RA_C(2)) and (ADD_RWC_EXMEM
    (1) XNOR aux_ADD_RA_C(1)) and (ADD_RWC_EXMEM(0) XNOR aux_ADD_RA_C(0))) and
    ovWE_EXMEM;
165 RA_C_WB_CONFLITO      <= ((ADD_RWC_WB(2) XNOR aux_ADD_RA_C(2)) and (ADD_RWC_WB(1)
    XNOR aux_ADD_RA_C(1)) and (ADD_RWC_WB(0) XNOR aux_ADD_RA_C(0))) and ovWE_WB;
166
167 RB_EXMEM_CONFLITO     <= ((ADD_RWC_EXMEM(2) XNOR aux_ADD_RB(2)) and (ADD_RWC_EXMEM(1)
    XNOR aux_ADD_RB(1)) and (ADD_RWC_EXMEM(0) XNOR aux_ADD_RB(0))) and ovWE_EXMEM;
168 RB_WB_CONFLITO        <= ((ADD_RWC_WB(2) XNOR aux_ADD_RB(2)) and (ADD_RWC_WB(1) XNOR
    aux_ADD_RB(1)) and (ADD_RWC_WB(0) XNOR aux_ADD_RB(0))) and ovWE_WB;
169
170
171 Conflit_EXMEN_RA_C_ON  <= RA_C_EXMEM_CONFLITO and inst_IN(15);
172 Conflit_WB_RA_C_ON    <= RA_C_WB_CONFLITO and inst_IN(15);
173 Conflit_EXMEN_RB_ON   <= (RB_EXMEM_CONFLITO and inst_IN(15)) OR (RB_EXMEM_CONFLITO
    AND NOT(inst_IN(15)) AND NOT(inst_IN(14)) AND inst_IN(13) AND inst_IN(12));
174 Conflit_WB_RB_ON      <= (RB_WB_CONFLITO and inst_IN(15)) OR (RB_EXMEM_CONFLITO AND
    NOT(inst_IN(15)) AND NOT(inst_IN(14)) AND inst_IN(13) AND inst_IN(12));
175
176 -----
177 ----- Conjuntos de instrucoes -----
178 ----- Inst_IN(15:14) -----
179 ----- 0 0 => Transferencia de Controllo -----
180 ----- 0 1 => Constantes Formato I -----
181 ----- 1 0 => Instrucoes para ALU/Memoria -----
182 ----- 1 1 => Constante Formato II -----
183 -----
184
185 aux_active_FLAGTEST <= inst_IN(15) or inst_IN(14); --- Activa a FLAGTESTE
186 aux_TEST_NOP <= inst_IN(15) or inst_IN(14) or inst_IN(13) or inst_IN(12) or
187     inst_IN(11) or inst_IN(10) or inst_IN(9) or inst_IN(8) or
188     inst_IN(7) or inst_IN(6) or inst_IN(5) or inst_IN(4) or
189     inst_IN(3) or inst_IN(2) or inst_IN(1) or inst_IN(0);
190 -----
191 ----- 0 0 -> Transferencia de Controllo -----
192 ----- exitsem 3 formatos -----
193 -----
194 aux_TRANS_OP <= inst_IN(13 downto 12);
195
196 ----- 0 0/ 0 1 -> Formato I condicional -----
197
198 aux_TRANS_FI_DES <= (11 downto 8 => inst_IN(7)) & inst_IN(7 downto 0);
199 TRANS_FI_COND_IN <= inst_IN(11 downto 8 );
200
201 ----- 1 0 -> Formato II incondicional -----
202 aux_TRANS_FII_DES <= inst_IN(11 downto 0);
203

```

```

204 ----- 1 1 -> Formato III jumps -----
205
206 aux_JUMPS_active  <= not(aux_active_FLAGTEST) and inst_IN(13) and inst_IN(12); -----
    TESTE de activacao do FIII, escolhe o mux2:1
207 aux_JUMPS_MUX_WB  <= not(inst_IN(11)) and aux_JUMPS_active;
208
209 -----
210 ----- 0 1 -> Constantes  Formato I -----
211 -----
212
213 aux_CONS_FI_11B    <=  inst_IN(10 downto 0 );
214
215
216 -----
217 ----- 1 0 -> Instrucoes para a ALU/Memoria -----
218 -----
219
220 aux_ALU_OPER       <=  inst_IN(10 downto 6);
221
222 -----
223 ----- 1 1 -> Constantes  Formato II -----
224 -----
225 aux_CONS_FII_8B    <=  inst_IN(7  downto 0);
226
227
228 --
    -----
229 ----- TESTE FLAGS -----
    -----
230 --
    -----

231 aux_FLAGMUX_S_For  <=FLAGS_IN(3)                when TRANS_FI_COND_IN = "0101" else
232                FLAGS_IN(2)                when TRANS_FI_COND_IN = "0100" else
233                FLAGS_IN(1)                when TRANS_FI_COND_IN = "0110" else
234                FLAGS_IN(0)                when TRANS_FI_COND_IN = "0011" else
235                '1',                        when TRANS_FI_COND_IN = "0000" else
236                FLAGS_IN(3) or FLAGS_IN(2) when TRANS_FI_COND_IN = "0111" else
237                '0';
238
239 aux_FLAGMUX_C_For  <=Forw_FLAGS_IN(3)                when TRANS_FI_COND_IN = "0101"
    else
240                Forw_FLAGS_IN(2)                when TRANS_FI_COND_IN = "0100" else
241                Forw_FLAGS_IN(1)                when TRANS_FI_COND_IN = "0110" else
242                Forw_FLAGS_IN(0)                when TRANS_FI_COND_IN = "0011" else
243                '1',                        when TRANS_FI_COND_IN = "0000" else

```

```

244         Forw_FLAGS_IN(3) or Forw_FLAGS_IN(2) when TRANS_FI_COND_IN = "0111" else
245         '0';
246
247 aux_FLAGMUX <= aux_FLAGMUX_C_For when not(aux_active_FLAGTEST) = '1' else
248         aux_FLAGMUX_S_For;
249
250 aux_FLAGTEST <= (not(aux_TRANS_OP(1)) and aux_TRANS_OP(0) and aux_FLAGMUX) or (not(
251         aux_TRANS_OP(1)) and not(aux_TRANS_OP(0)) and not(aux_FLAGMUX));
252
253 aux_FLAGTEST_MUXPC <= not(aux_active_FLAGTEST) and aux_FLAGTEST;
254
255 -----
256 ----- Operand Fetch -----
257 -----
258 RA_C <= R0 when aux_ADD_RA_C = "000" else
259         R1 when aux_ADD_RA_C = "001" else
260         R2 when aux_ADD_RA_C = "010" else
261         R3 when aux_ADD_RA_C = "011" else
262         R4 when aux_ADD_RA_C = "100" else
263         R5 when aux_ADD_RA_C = "101" else
264         R6 when aux_ADD_RA_C = "110" else
265         R7;
266
267
268 RB <= R0 when aux_ADD_RB = "000" else
269         R1 when aux_ADD_RB = "001" else
270         R2 when aux_ADD_RB = "010" else
271         R3 when aux_ADD_RB = "011" else
272         R4 when aux_ADD_RB = "100" else
273         R5 when aux_ADD_RB = "101" else
274         R6 when aux_ADD_RB = "110" else
275         R7;
276 -- operando A da ALU
277 mux_RA <= (Conflit_WB_RA_C_ON&Conflit_EXMEN_RA_C_ON);
278 aux_oper_A <= Forw_EXMEN when mux_RA = "01" else
279         Forw_EXMEN when mux_RA = "11" else
280         Forw_WB when mux_RA = "10" else
281         RA_C;
282 oper_A <= aux_oper_A;
283 -- operando B da ALU
284 mux_RB <= Conflit_WB_RB_ON&Conflit_EXMEN_RB_ON;
285 oper_B <= Forw_EXMEN when mux_RB = "01" else
286         Forw_EXMEN when mux_RB = "11" else
287         Forw_WB when mux_RB = "10" else
288         RB ;

```



```

289
290
291 const11 <= (15 downto 11 => aux_CONS_FI_11B(10)) & aux_CONS_FI_11B; -- loadlit c
292 lcl <= aux_oper_A(15 downto 8) & aux_CONS_FII_8B; -- lcl c
293 lch <= aux_CONS_FII_8B & aux_oper_A(7 downto 0); -- lch c
294
295 select_mux_constantes <= inst_IN(15) & inst_IN(10);
296
297 out_mux_constantes <= const11 when select_mux_constantes = "00" else
298 const11 when select_mux_constantes = "01" else
299 lcl when select_mux_constantes = "10" else
300 lch;
301
302 ALU_vs_MEM <= (aux_ALU_OPER(1) and not(aux_ALU_OPER(2))) and (aux_ALU_OPER(3) and not(
aux_ALU_OPER(4)));
303
304 WE_RAM <= (inst_IN(15) and not(inst_IN(14))) and (ALU_vs_MEM and inst_IN(6));
305
306
307
308 bit15 <= inst_IN(15);
309 bit14 <= inst_IN(14);
310
311 -----
312 -----Saltos Condicionais e incondicionais-----
313 -----
314 Aux_Conflit_DADOS <= not(aux_active_FLAGTEST) AND aux_TEST_NOP;
315
316 act_BTBTB <= aux_FLAGTEST_MUXPC;
317 en_Jump_FII <= aux_TRANS_OP(1) and not(aux_TRANS_OP(0));
318 en_Jump_Incond <= aux_JUMPS_active; --acrescentar e fazer logica
319
320 pc_branch <= reg_IfOut_PC - one; --Modificado em relacao ao apresentado no
Laboratorio, foi acrescentado -one;
321 JUMP_MUXWB_OUT <= aux_JUMPS_MUX_WB;
322
323
324 -----Escolha da constante do destino-----
325 Jump_Incon <= oper_B(11 downto 0);
326
327 Jump_BTBTB <= aux_TRANS_FII_DES + pc_add1 when aux_TRANS_OP = "10" else
aux_TRANS_FI_DES + pc_add1;
328
329
330 -----
331 -----Andar WB-- Wirte enable do registo de saida-----
332 -----
333

```

```

334 ALU_e_MEM <= (inst_IN(15) and (not(inst_IN(14))));
335
336 soMEM <= ALU_e_MEM and ALU_vs_MEM;
337
338 soALU <= ALU_e_MEM and not(ALU_vs_MEM);
339
340 loadMEM <= soMEM and not(inst_IN(6));
341
342 controlo <= inst_IN(14) nor inst_IN(15);
343
344 controloJump <= controlo and JUMP_MUXWB_OUT;
345
346 ovWE <= (soALU or loadMEM) or (controloJump or inst_IN(14));
347
348
349 aux_sel_bit1 <= JUMP_MUXWB_OUT or inst_IN(14);
350     -- JUMP_MUXWB_OUT or inst_IN(14)
351
352 aux_sel_bit0 <= JUMP_MUXWB_OUT or (not(inst_IN(14)) and ALU_vs_MEM) ;
353     -- JUMP_MUXWB_OUT or (not(inst_IN(14)) and ALU_vs_MEM)
354
355 sel_mux_WB <= aux_sel_bit1&aux_sel_bit0;
356 -----
357 ----- Exit -----
358 -----
359
360 ----- registro de saida do segundo andar: ID e OF -----
361 process (clk, rst)
362 begin
363     if clk'event and clk = '1' then
364         if rst = '1' then
365             aux_reg_IDOF_OUT_bit15 <= '0'; --need
366             aux_reg_IDOF_OUT_bit14 <= '0';
367             aux_reg_IDOF_OUT_WERAM <= '0';-- need
368             aux_reg_IDOF_OUT_ALUOPER <= "00000"; --need
369             aux_reg_IDOF_OUT_ALUvsMEM <= '0';--need
370             aux_reg_IDOF_OUT_PCadd1 <= zero_12;--need
371             aux_reg_IDOF_OUT_SelMuxWB <= "00"; --add
372             aux_reg_IDOF_OUT_AddRWC <= "000"; -- need
373             aux_reg_IDOF_OUT_OperA <= zero_16; --need
374             aux_reg_IDOF_OUT_OperB <= zero_16; --need
375             aux_reg_IDOF_OUT_MuxConst <= zero_16;
376             aux_reg_IDOF_OUT_ovWE <= '0';
377         elsif en_idOF = '1' then
378             aux_reg_IDOF_OUT_bit15 <= bit15; --need
379             aux_reg_IDOF_OUT_bit14 <= bit14;
380             aux_reg_IDOF_OUT_WERAM <= WE_RAM;-- need

```

```

381     aux_reg_IDOF_OUT_ALUOPER    <= aux_ALU_OPER; --need
382     aux_reg_IDOF_OUT_ALUvsMEM  <= ALU_vs_MEM;--need
383     aux_reg_IDOF_OUT_PCadd1    <= pc_add1;--need
384     aux_reg_IDOF_OUT_SelMuxWB  <= sel_mux_WB; --add
385     aux_reg_IDOF_OUT_AddRWC    <= aux_ADD_RWC; -- need
386     aux_reg_IDOF_OUT_OperA     <= oper_A; --need
387     aux_reg_IDOF_OUT_OperB     <= oper_B; --need
388     aux_reg_IDOF_OUT_MuxConst  <= out_mux_constantes;
389     aux_reg_IDOF_OUT_ovWE      <= ovWE;
390     end if;
391     end if;
392 end process;
393
394 reg_IDOF_OUT_bit15    <= aux_reg_IDOF_OUT_bit15; --need
395 reg_IDOF_OUT_bit14    <= aux_reg_IDOF_OUT_bit14;
396 reg_IDOF_OUT_WERAM    <= aux_reg_IDOF_OUT_WERAM;-- need
397 reg_IDOF_OUT_ALUOPER  <= aux_reg_IDOF_OUT_ALUOPER; --need
398 reg_IDOF_OUT_ALUvsMEM <= aux_reg_IDOF_OUT_ALUvsMEM;--need
399 reg_IDOF_OUT_PCadd1   <= aux_reg_IDOF_OUT_PCadd1;--need
400 reg_IDOF_OUT_SelMuxWB <= aux_reg_IDOF_OUT_SelMuxWB; --add
401 reg_IDOF_OUT_AddRWC   <= aux_reg_IDOF_OUT_AddRWC; -- need
402 reg_IDOF_OUT_OperA    <= aux_reg_IDOF_OUT_OperA; --need
403 reg_IDOF_OUT_OperB    <= aux_reg_IDOF_OUT_OperB; --need
404 reg_IDOF_OUT_MuxConst <= aux_reg_IDOF_OUT_MuxConst;
405 reg_IDOF_OUT_ovWE     <= aux_reg_IDOF_OUT_ovWE;
406 Conflit_DADOS         <= Aux_Conflit_DADOS;
407
408 end Behavioral;

```

7.3 Código do andar de EX e MEM

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_ARITH.ALL;
4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6  entity EXeMEM is
7      port(
8          -- input
9          clk, rst          : in std_logic;
10         reg_IDOF_OUT_WERAM : in std_logic;
11         reg_IDOF_OUT_ALUOPER : in std_logic_vector(4 downto 0);
12         reg_IDOF_OUT_bit15   : in std_logic;
13         reg_IDOF_OUT_bit14   : in std_logic;
14         reg_IDOF_OUT_OperA    : in std_logic_vector(15 downto 0);
15         reg_IDOF_OUT_OperB    : in std_logic_vector(15 downto 0);
16         reg_IDOF_OUT_ALUvsMEM : in std_logic;
17         reg_IDOF_OUT_ovWE     : in std_logic;

```

```

18  reg_IDOF_OUT_AddRWC      : in std_logic_vector(2 downto 0);
19  reg_IDOF_OUT_PCadd1     : in std_logic_vector(11 downto 0);
20  reg_IDOF_OUT_SelMuxWB   : in std_logic_vector(1 downto 0);
21  reg_IDOF_OUT_MuxConst   : in std_logic_vector(15 downto 0);
22  FLAGS_IN                : in std_logic_vector(3 downto 0);
23  out_RAM                 : in std_logic_vector(15 downto 0);
24  en_EX                   : in std_logic;
25
26  -- output
27  reg_EXMEM_OUT_PCadd1    : out std_logic_vector(11 downto 0);
28  reg_EXMEM_OUT_AddRWC    : out std_logic_vector(2 downto 0);
29  reg_EXMEM_OUT_OutALU    : out std_logic_vector(15 downto 0);
30  reg_EXMEM_OUT_OutMEM    : out std_logic_vector(15 downto 0);
31  reg_EXMEM_OUT_MuxConst  : out std_logic_vector(15 downto 0);
32  reg_EXMEM_OUT_ovWE      : out std_logic;
33  reg_EXMEM_OUT_SelMuxWB  : out std_logic_vector(1 downto 0);
34
35  ADD_RWC_EXMEM           : out std_logic_vector(2 downto 0);
36  ovWE_EXMEM              : out std_logic;
37
38  out_ADD_MEM             : out std_logic_vector(11 downto 0);    -- para enderecar a
RAM
39  out_WE_MEM              : out std_logic;                        -- para controlar o WE da RAM
40  FLAGS_OUT               : out std_logic_vector(3 downto 0);
41  Forw_FLAGSTEST_OUT      : out std_logic_vector(3 downto 0);
42  Forw_EXMEN              : out std_logic_vector(15 downto 0);
43  FLAGSTEST_OUT           : out std_logic_vector(3 downto 0);
44  in_RAM                  : out std_logic_vector(15 downto 0)
45
46 );
47 end EXeMEM;
48
49 architecture Behavioral of EXeMEM is
50
51 -----
52 ----- Aux Signals -----
53 -----
54 signal out_ALU            : std_logic_vector(15 downto 0) := (others => '0'); -- saida
da ALU
55 signal out_MEM            : std_logic_vector(15 downto 0) := (others => '0'); -- saida
da memoria
56 signal aux_FLAGS_ARI      : std_logic_vector(3 downto 0) := (others => '0');  -- Z,N,C
,O
57 signal aux_FLAGS_SHIFT    : std_logic_vector(2 downto 0) := (others => '0');  -- Z,N
,C
58 signal aux_FLAGS_LOG      : std_logic_vector(1 downto 0) := (others => '0');  -- Z,N
59 signal aux_FLAGS          : std_logic_vector(3 downto 0) := (others => '0');  -- Z,N,C,O

```

```

60 signal aux_MSR_FLAGS      : std_logic_vector(3 downto 0) := (others => '0');
61 signal operando_A        : std_logic_vector(15 downto 0) := (others => '0');
62 signal operando_B        : std_logic_vector(15 downto 0) := (others => '0');
63 signal oper_ALU          : std_logic_vector(4 downto 0) := (others => '0');
64 signal p_ALU             : std_logic_vector(15 downto 0) := (others => '0');
65 signal sel_mux_q         : std_logic_vector(1 downto 0) := (others => '0');
66 signal q_ALU             : std_logic_vector(15 downto 0) := (others => '0');
67 signal cIN_ALU          : std_logic := '0';
68 signal out_ARI           : std_logic_vector(16 downto 0) := (others => '0');
69 signal out_LOG           : std_logic_vector(15 downto 0) := (others => '0');
70 signal out_SHIFT         : std_logic_vector(16 downto 0) := (others => '0');
71 signal sel_mux_LOG       : std_logic_vector(2 downto 0) := (others => '0');
72 signal sel_mux_ALU       : std_logic_vector(1 downto 0) := (others => '0');
73 signal aux_sel_mux_ALU_bit0 : std_logic := '0';
74 signal aux_FLAGMUX       : std_logic := '0';
75 signal aux_flagtest_rel  : std_logic := '0';
76 signal TRANS_OP          : std_logic_vector(1 downto 0) := (others => '0');
77 signal TRANS_FI_COND_IN  : std_logic_vector(3 downto 0) := (others => '0');
78 signal FLAGTEST_active_IN : std_logic := '0';
79 signal aux_EXMEM_bit15   : std_logic := '0';
80 signal Sign_FLAG         : std_logic_vector(1 downto 0) := (others => '0');
81 signal aux_Sign_FLAG     : std_logic_vector(1 downto 0) := (others => '0');
82 signal out_ADD_MEM_aux   : std_logic_vector(11 downto 0) := (others => '0');
83 signal out_WE_MEM_aux    : std_logic := '0';
84
85
86 signal aux_reg_EXMEM_OUT_PCadd1 : std_logic_vector(11 downto 0) := (others => '0');
87 signal aux_reg_EXMEM_OUT_AddRWC : std_logic_vector(2 downto 0) := (others => '0');
88 signal aux_reg_EXMEM_OUT_OutALU : std_logic_vector(15 downto 0) := (others => '0');
89 signal aux_reg_EXMEM_OUT_OutMEM : std_logic_vector(15 downto 0) := (others => '0');
90 signal aux_reg_EXMEM_OUT_MuxConst : std_logic_vector(15 downto 0) := (others => '0');
91 signal aux_reg_EXMEM_OUT_ovWE : std_logic := '0';
92 signal aux_reg_EXMEM_OUT_SelMuxWB : std_logic_vector(1 downto 0) := (others => '0');
93
94 -----
95 ----- Constantes -----
96 -----
97 constant zero_12         : std_logic_vector(11 downto 0) := (others => '0');
98 constant zero_16         : std_logic_vector(15 downto 0) := (others => '0');
99 constant zeros_4         : std_logic_vector(3 downto 0) := (others => '0');
100 constant menusum         : std_logic_vector(15 downto 0) := (others => '1');
101 constant zeros_ALU       : std_logic_vector(15 downto 0) := (others => '0');
102
103 begin
104
105 operando_A      <= reg_IDOF_OUT_OperA;
106 operando_B      <= reg_IDOF_OUT_OperB;

```

```

107 oper_ALU      <= reg_IDOF_OUT_ALUOPER;
108 aux_EXMEM_bit15 <= reg_IDOF_OUT_bit15;
109
110 --
111 -----
112 ----- MEMORIA -----
113 -----
114 out_ADD_MEM_aux <= operando_A(11 downto 0); -- para enderecar leitura e escrita da RAM
115
116 out_ADD_MEM <= out_ADD_MEM_aux;
117
118 out_WE_MEM <= reg_IDOF_OUT_WERAM;
119 out_MEM <= out_RAM; -- armazenar depois em RC o valor contido na posicao de memoria
    enderecada por A
120
121 in_RAM <= operando_B; -- armazenar na posicao de memoria enderecada por A o valor
    contido em B
122
123 --
124 -----
125 ----- ALU -----
126 -----
127 ----- aritmeticas -----
128 -----
129 p_ALU <= operando_A;
130
131 sel_mux_q <= oper_ALU(2) & oper_ALU(1);
132
133 q_ALU <= operando_B      when sel_mux_q = "00" else
134      zeros_ALU          when sel_mux_q = "01" else
135      not(operando_B)    when sel_mux_q = "10" else
136      minusum;
137
138 cIN_ALU <= oper_ALU(0);
139
140 out_ARI <= ('0' & p_ALU) + ('0' & q_ALU) + cIN_ALU;

```

```

141
142 ----- logicas
143 -----
144 sel_mux_LOG <= (oper_ALU(3) xor oper_ALU(2)) & (oper_ALU(3) xor oper_ALU(1)) & (
145     oper_ALU(3) xor oper_ALU(0));
146
147 out_LOG <=  zeros_ALU                when sel_mux_LOG = "000" else
148     operando_A and operando_B        when sel_mux_LOG = "001" else
149     not(operando_A) and operando_B    when sel_mux_LOG = "010" else
150     operando_B                        when sel_mux_LOG = "011" else
151     operando_A and not(operando_B)    when sel_mux_LOG = "100" else
152     operando_A                        when sel_mux_LOG = "101" else
153     operando_A xor operando_B         when sel_mux_LOG = "110" else
154     operando_A or operando_B;
155 ----- shifts
156 -----
157 out_SHIFT <= (operando_A(15 downto 0) & '0') when oper_ALU(0) = '0' else -- SLL
158     (operando_A(15) & operando_A(15) & operando_A(15 downto 1)); -- SRA
159
160 ----- resultado final da ALU
161 -----
162 sel_mux_ALU <= oper_ALU(4) & oper_ALU(3);
163
164 out_ALU <=  out_ARI(15 downto 0)      when sel_mux_ALU = "00"   else
165     out_SHIFT(15 downto 0)            when sel_mux_ALU = "01"   else
166     out_LOG                           when sel_mux_ALU = "10"   else
167     not(out_LOG);
168
169 --
170 ----- FLAGS
171 -----
172
173 -----QUAIS FLAGS ATUALIZAM
174     ??-----
175 aux_Sign_FLAG(1) <=  not(oper_ALU(4));
176

```

```

177 aux_Sign_FLAG(0) <= (oper_ALU(2) and (not(oper_ALU(1)) or oper_ALU(2))) or (oper_ALU
    (4) and (not(oper_ALU(2)) and ((not(oper_ALU(1)) and oper_ALU(0)) or oper_ALU(3)))
    ) or ((oper_ALU(4)) nor oper_ALU(3)) or (not(oper_ALU(0)) and oper_ALU(1));
178
179 Sign_FLAG <= aux_Sign_FLAG when (aux_EXMEM_bit15 and NOT(reg_IDOF_OUT_bit14) and NOT
    (reg_IDOF_OUT_ALUvsMEM)) = '1' else "00";
180 --Actualizar FLAGS
181
182 -----FLAGS DA ARI-----
183
184
185 --OVERFLOW
186 aux_FLAGS_ARI(0) <= (q_ALU(15) xnor p_ALU(15)) and (q_ALU(15) xor out_ALU(15));
187
188 --CARRY
189 aux_FLAGS_ARI(1) <= out_ARI(16);
190
191 --NEGATIVE
192 aux_FLAGS_ARI(2) <= out_ARI(15);
193
194 --ZERO
195 aux_FLAGS_ARI(3) <= not(out_ARI(15) or out_ARI(14) or out_ARI(13) or out_ARI(12) or
    out_ARI(11)
196     or out_ARI(10) or out_ARI(9) or out_ARI(8) or out_ARI(7) or out_ARI(6)
197     or out_ARI(5) or out_ARI(4) or out_ARI(3) or out_ARI(2) or out_ARI(1)
198     or out_ARI(0));
199
200 -----FLAGS LOGICA-----
201 --NEGATIVE
202 aux_FLAGS_LOG(0) <= out_LOG(15);
203
204 --ZERO
205 aux_FLAGS_LOG(1) <= not(out_LOG(15) or out_LOG(14) or out_LOG(13) or out_LOG(12) or
    out_LOG(11)
206     or out_LOG(10) or out_LOG(9) or out_LOG(8) or out_LOG(7) or out_LOG(6)
207     or out_LOG(5) or out_LOG(4) or out_LOG(3) or out_LOG(2) or out_LOG(1)
208     or out_LOG(0));
209
210
211 -----FLAGS SHIFT-----
212 --CARRY
213 aux_FLAGS_SHIFT(0) <= out_SHIFT(16);
214
215 --NEGATIVE
216 aux_FLAGS_SHIFT(1) <= out_SHIFT(15);
217
218 --ZERO

```



```

219 aux_FLAGS_SHIFT(2)  <= not(out_SHIFT(15) or out_SHIFT(14) or out_SHIFT(13) or
    out_SHIFT(12) or out_SHIFT(11)
220         or out_SHIFT(10) or out_SHIFT(9) or out_SHIFT(8) or out_SHIFT(7) or
    out_SHIFT(6)
221         or out_SHIFT(5) or out_SHIFT(4) or out_SHIFT(3) or out_SHIFT(2) or out_SHIFT
    (1)
222         or out_SHIFT(0));
223 -----
224
225 aux_FLAGS <= FLAGS_IN                when Sign_FLAG = "00" else
226     aux_FLAGS_LOG & FLAGS_IN(1 downto 0)  when Sign_FLAG = "01" else
227     aux_FLAGS_SHIFT & FLAGS_IN(0)         when Sign_FLAG = "10" else
228     aux_FLAGS_ARI;
229
230
231
232
233 --
    -----
234 ----- REGISTO FLAGS -----
    -----
235 --
    -----
236
237 process (clk, rst)
238 begin
239     if clk'event and clk = '1' then
240         if rst = '1' then
241             aux_MSR_FLAGS <= zeros_4;
242         else
243             aux_MSR_FLAGS <= aux_FLAGS;
244         end if;
245     end if;
246 end process;
247
248 Forw_FLAGSTEST_OUT <= aux_FLAGS;
249 FLAGS_OUT          <= aux_MSR_FLAGS;
250 FLAGSTEST_OUT <= aux_MSR_FLAGS;
251
252
253
254 -----
255 ----- Exit -----
256 -----
257

```

```

258 ----- registro de saida do terceiro andar: EX e MEM -----
259 process (clk, rst)
260 begin
261     if clk'event and clk = '1' then
262         if rst = '1' then
263             aux_reg_EXMEM_OUT_ovWE      <= '0';
264             aux_reg_EXMEM_OUT_SelMuxWB  <= "00";
265             aux_reg_EXMEM_OUT_PCadd1    <= zero_12;
266             aux_reg_EXMEM_OUT_AddRWC    <= "000";
267             aux_reg_EXMEM_OUT_OutALU    <= zero_16;
268             aux_reg_EXMEM_OUT_OutMEM    <= zero_16;
269             aux_reg_EXMEM_OUT_MuxConst  <= zero_16;
270         elsif en_EX = '1' then
271             aux_reg_EXMEM_OUT_ovWE      <= reg_IDOF_OUT_ovWE;
272             aux_reg_EXMEM_OUT_SelMuxWB  <= reg_IDOF_OUT_SelMuxWB;
273             aux_reg_EXMEM_OUT_PCadd1    <= reg_IDOF_OUT_PCadd1;
274             aux_reg_EXMEM_OUT_AddRWC    <= reg_IDOF_OUT_AddRWC;
275             aux_reg_EXMEM_OUT_OutALU    <= out_ALU;
276             aux_reg_EXMEM_OUT_OutMEM    <= out_MEM;
277             aux_reg_EXMEM_OUT_MuxConst  <= reg_IDOF_OUT_MuxConst;
278             -- reg_EXMEM_OUT <= aux_EXMEM_bit6 & aux_EXMEM_bit15 & out_MEM & ALU_vs_MEM &
save_pc_add_1 & JUMP_MUXWB_OUT & aux_ADD_RWC &
279             -- out_ALU & out_mux_constantes & ALU_CONS_SEL;
280         end if;
281     end if;
282 end process;
283
284 reg_EXMEM_OUT_ovWE      <= aux_reg_EXMEM_OUT_ovWE;
285 reg_EXMEM_OUT_SelMuxWB  <= aux_reg_EXMEM_OUT_SelMuxWB;
286 reg_EXMEM_OUT_PCadd1    <= aux_reg_EXMEM_OUT_PCadd1;
287 reg_EXMEM_OUT_AddRWC    <= aux_reg_EXMEM_OUT_AddRWC;
288 reg_EXMEM_OUT_OutALU    <= aux_reg_EXMEM_OUT_OutALU;
289 reg_EXMEM_OUT_OutMEM    <= aux_reg_EXMEM_OUT_OutMEM;
290 reg_EXMEM_OUT_MuxConst  <= aux_reg_EXMEM_OUT_MuxConst;
291 -----
292 ---Conflito---
293 -----
294 ADD_RWC_EXMEM <= reg_IDOF_OUT_AddRWC;
295 ovWE_EXMEM    <= reg_IDOF_OUT_ovWE;
296
297
298 Forw_EXMEN    <= out_ALU when reg_IDOF_OUT_SelMuxWB = "00" else
299 out_MEM when reg_IDOF_OUT_SelMuxWB = "01" else
300 reg_IDOF_OUT_MuxConst when reg_IDOF_OUT_SelMuxWB = "10" else
301 X"0"&reg_IDOF_OUT_PCadd1;
302
303

```

```
304 end Behavioral;
```

7.4 Código do andar de WB

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  -- Uncomment the following library declaration if using
5  -- arithmetic functions with Signed or Unsigned values
6  --use IEEE.NUMERIC_STD.ALL;
7
8  -- Uncomment the following library declaration if instantiating
9  -- any Xilinx primitives in this code.
10 --library UNISIM;
11 --use UNISIM.VComponents.all;
12
13 entity WB is
14   port(
15     -- input
16     clk, rst      : in std_logic;
17     reg_EXMEM_OUT_PCadd1      : in std_logic_vector(11 downto 0);
18     reg_EXMEM_OUT_AddRWC      : in std_logic_vector(2 downto 0);
19     reg_EXMEM_OUT_OutALU      : in std_logic_vector(15 downto 0);
20     reg_EXMEM_OUT_OutMEM      : in std_logic_vector(15 downto 0);
21     reg_EXMEM_OUT_MuxConst    : in std_logic_vector(15 downto 0);
22     reg_EXMEM_OUT_ovWE        : in std_logic;
23     reg_EXMEM_OUT_SelMuxWB     : in std_logic_vector(1 downto 0);
24
25     -- output
26     ADD_RWC_WB                : out std_logic_vector(2 downto 0);
27     ovWE_WB                   : out std_logic;
28     Forw_WB                   : out std_logic_vector(15 downto 0);
29     out_mux_WB                : out std_logic_vector(15 downto 0);
30     out_saida                 : out std_logic_vector(15 downto 0);
31     en_regs                   : out std_logic_vector(7 downto 0)
32   );
33 end WB;
34
35
36 architecture Behavioral of WB is
37
38   -----
39   ----- Aux Signals -----
40   -----
41   signal sel_mux_WB           : std_logic_vector(1 downto 0) := (others => '0');
42   signal aux_out_alu          : std_logic_vector(15 downto 0) := (others => '0');
43   signal aux_out_const        : std_logic_vector(15 downto 0) := (others => '0');
44   signal aux_mux_WB           : std_logic_vector(15 downto 0) := (others => '0');
```

```

45 signal aux_out_pcadd1      : std_logic_vector(15 downto 0) := (others => '0');
46 signal aux_en_WC          : std_logic_vector(2 downto 0) := (others => '0');
47
48
49 constant zeros            : std_logic_vector(15 downto 0) := (others => '0');
50 begin
51
52
53 sel_mux_WB <= reg_EXMEM_OUT_SelMuxWB;
54
55 aux_out_alu    <= reg_EXMEM_OUT_OutALU;
56 aux_out_const  <= reg_EXMEM_OUT_MuxConst;
57 aux_out_pcadd1 <= X"0" & reg_EXMEM_OUT_PCadd1;
58
59 aux_mux_WB <= aux_out_alu          when sel_mux_WB = "00" else -- escrever
a saida da ALU      (out_ALU)
60 reg_EXMEM_OUT_OutMEM          when sel_mux_WB = "01" else -- escrever
saida MEM          (out_MEM)
61 aux_out_const          when sel_mux_WB = "10" else -- fazer load de
uma constante      (out_mux_constantes)
62 aux_out_pcadd1; -- guardar em R7 o valor
de PC+1 (save_pc_add_1)
63
64 out_mux_WB <= aux_mux_WB;
65 aux_en_WC <= "111" when (sel_mux_WB(1) and sel_mux_WB(0)) = '1' else
reg_EXMEM_OUT_AddRWC;
66
67
68
69 with aux_en_WC select
70 en_regs <= "0000000" & reg_EXMEM_OUT_ovWE          when "000",
71 "000000" & reg_EXMEM_OUT_ovWE & '0'          when "001",
72 "00000" & reg_EXMEM_OUT_ovWE & "00"          when "010",
73 "0000" & reg_EXMEM_OUT_ovWE & "000"          when "011",
74 "000" & reg_EXMEM_OUT_ovWE & "0000"          when "100",
75 "00" & reg_EXMEM_OUT_ovWE & "00000"          when "101",
76 '0' & reg_EXMEM_OUT_ovWE & "000000"          when "110",
77 reg_EXMEM_OUT_ovWE & "0000000"          when "111",
78 "00000000"          when others;
79
80
81 -----
82 -----EXIT-----
83
84 ADD_RWC_WB <= reg_EXMEM_OUT_AddRWC;
85 ovWE_WB <= reg_EXMEM_OUT_ovWE;
86

```

```

87 Forw_WB <= aux_mux_WB;
88
89 end Behavioral;

```

7.5 Código da memória ROM

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_UNSIGNED.ALL;
4  use STD.TEXTIO.all;
5  use STD.TEXTIO;
6  use IEEE.STD_LOGIC_TEXTIO.all;
7
8  entity memoria_ROM is
9      Generic(
10         ADDR_SIZE : positive := 12
11     );
12     Port(
13         Addr_ROM : in  STD_LOGIC_VECTOR(ADDR_SIZE-1 downto 0);
14         DO_ROM   : out STD_LOGIC_VECTOR(15 downto 0)
15     );
16 end memoria_ROM;
17
18 architecture Behavioral of memoria_ROM is
19
20 type MEM_TYPE is array(0 to (2**ADDR_SIZE)-1) of STD_LOGIC_VECTOR(15 downto 0);
21
22 impure function InitRamFromFile (RamFileName : in string) return MEM_TYPE is
23     file INFILE : TEXT is in "rom_inst.txt";
24     variable DATA_TEMP : STD_LOGIC_VECTOR(15 downto 0);
25     variable IN_LINE: LINE;
26     variable ROM : MEM_TYPE;
27     variable index : integer;
28
29     begin
30         index := 0;
31         while NOT(endfile(INFILE)) loop
32             readline(INFILE, IN_LINE);
33             hread(IN_LINE, DATA_TEMP);
34             ROM(index) := DATA_TEMP;
35             index := index + 1;
36         end loop;
37         for index in index to 4095 loop
38             ROM(index) := X"0000";
39         end loop;
40     return ROM;
41 end function;
42

```

```

43 signal ROM : MEM_TYPE := InitRamFromFile("rom_inst.txt");
44
45
46 begin
47
48 DO_ROM <= ROM(conv_integer(Addr_ROM)); -- leitura assincrona
49
50 end Behavioral;

```

7.6 Código da memória RAM

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_UNSIGNED.ALL;
4  use STD.TEXTIO.all;
5  use STD.TEXTIO;
6  use IEEE.STD_LOGIC_TEXTIO.all;
7
8
9  entity memoria_RAM is
10     Generic(
11         ADDR_SIZE : positive := 12
12     );
13     Port(
14         CLK_A      : in  STD_LOGIC;
15         WE_A       : in  STD_LOGIC;
16         Addr_A     : in  STD_LOGIC_VECTOR(ADDR_SIZE-1 downto 0);
17         DI_A       : in  STD_LOGIC_VECTOR(15 downto 0);
18         DO_A       : out STD_LOGIC_VECTOR(15 downto 0)
19     );
20 end memoria_RAM;
21
22 architecture Behavioral of memoria_RAM is
23
24 type MEM_TYPE is array(0 to (2**ADDR_SIZE)-1) of STD_LOGIC_VECTOR(15 downto 0);
25
26 impure function InitRamFromFile (RamFileName : in string) return MEM_TYPE is
27     file INFILE : TEXT is in "ram_inst.txt";
28     variable DATA_TEMP : STD_LOGIC_VECTOR(15 downto 0);
29     variable IN_LINE: LINE;
30     variable RAM : MEM_TYPE;
31     variable index : integer;
32     variable i : integer;
33
34     begin
35         index := 0;
36         i:=0;
37         readline(INFILE, IN_LINE);

```

```

38     hread(IN_LINE, DATA_TEMP);
39     index := CONV_INTEGER(DATA_TEMP);
40
41     for i in i to index loop
42         RAM(i) := X"0000";
43     end loop;
44     while NOT(endfile(INFILE)) loop
45         readline(INFILE, IN_LINE);
46         hread(IN_LINE, DATA_TEMP);
47         RAM(index) := DATA_TEMP;
48         index := index + 1;
49     end loop;
50     for index in index to 4095 loop
51         RAM(index) := X"0000";
52     end loop;
53     return RAM;
54 end function;
55
56 shared variable RAM : MEM_TYPE := InitRamFromFile("ram_inst.txt");
57
58 begin
59 process (CLK_A)
60     begin
61         if rising_edge(CLK_A) then
62             if WE_A='1' then
63                 RAM(conv_integer(Addr_A)) := DI_A;
64             end if;
65         end if;
66 end process;
67
68 DO_A <= RAM(conv_integer(Addr_A)); -- leitura assincrona
69
70 end Behavioral;

```

7.7 Código da BTB

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.STD_LOGIC_UNSIGNED.ALL;
4 use STD.TEXTIO.all;
5 use STD.TEXTIO;
6 use IEEE.STD_LOGIC_TEXTIO.all;
7
8
9 entity BTB_bram is
10     Generic(
11         ADDR_SIZE : positive := 9
12     );

```

```

13   Port(
14       CLK_A    : in  STD_LOGIC;
15       WE_A     : in  STD_LOGIC;
16       Addr_A   : in  STD_LOGIC_VECTOR(ADDR_SIZE-1 downto 0);
17       Addr_B   : in  STD_LOGIC_VECTOR(ADDR_SIZE-1 downto 0);
18       DI_A     : in  STD_LOGIC_VECTOR(16 downto 0);
19       DO_B     : out STD_LOGIC_VECTOR(16 downto 0)
20   );
21 end BTB_bram;
22
23 architecture Behavioral of BTB_bram is
24
25 type MEM_TYPE is array(0 to (2**ADDR_SIZE)-1) of STD_LOGIC_VECTOR(16 downto 0);
26
27 constant InitValue : MEM_TYPE := ( others => "000000000000000000");
28
29 shared variable myRAM : MEM_TYPE := InitValue;
30
31 begin
32 process (CLK_A)
33     begin
34         if rising_edge(CLK_A) then
35             if WE_A='1' then
36                 myRAM(conv_integer(Addr_A)) := DI_A;
37             end if;
38         end if;
39     end process;
40
41 DO_B <= myRAM(conv_integer(Addr_B)); -- leitura assincrona
42
43 end Behavioral;

```

7.8 Código da máquina de estados

```

1
2 library IEEE;
3 use IEEE.STD_LOGIC_1164.ALL;
4 use IEEE.STD_LOGIC_ARITH.ALL;
5 use IEEE.STD_LOGIC_SIGNED.ALL;
6
7
8 entity Controlo is
9     port (
10         clk, rst : in std_logic;
11         en_if     : out std_logic;
12         en_idOF   : out std_logic;
13         en_EX     : out std_logic;
14     );

```



```

15 end Controlo;
16
17
18 architecture Behavioral of Controlo is
19     type fsm_states is (s_inicial,s_inicial2,s_inicial3,s_cont);
20     signal currstate, nextstate: fsm_states;
21 begin
22
23 state_reg: process (clk,rst)
24     begin
25         if rst = '1' then
26             currstate <= s_inicial ;
27         elsif clk'event and clk = '1' then
28             currstate <= nextstate ;
29         end if ;
30 end process;
31
32 state_comb: process (currstate )
33     begin -- process
34         nextstate <= currstate;
35
36
37         case currstate is
38         when s_inicial =>
39             nextstate <= s_inicial2;
40             en_if      <= '1';
41             en_idOF    <= '1';
42             en_EX      <= '1';
43         when s_inicial2 =>
44             nextstate <= s_inicial3;
45             en_if      <= '1';
46             en_idOF    <= '1';
47             en_EX      <= '1';
48         when s_inicial3 =>
49             nextstate <= s_cont;
50             en_if      <= '1';
51             en_idOF    <= '1';
52             en_EX      <= '1';
53
54         when s_cont =>
55             nextstate <= s_cont ;
56             en_if      <= '1';
57             en_idOF    <= '1';
58             en_EX      <= '1';
59
60         end case;
61 end process;

```

62
63
64

```
end Behavioral;
```

Todo list

| | |
|---|---|
| ■ estrutura ideal do relatorio: 1:referir no geral que conflitos ha. 2:referir quais os conflitos que temos neste processador. 3:referir como os resolvemos. 4:comparar as arquiteturas que testamos. 5:entregar antes da meia-noite LooLoL | 1 |
| ■ referir nesta secção explicações teóricas dos conflitos que ha. e referir que conflitos ha de facto no nosso processador | 1 |
| ■ aqui por exemplo nos só vamos ter RAW. mas temos de explicar todos os que ha | 1 |
| ■ não impede a predição | 1 |
| ■ explicar o que é uma BTB, explicar BPB | 2 |