Politecnico di Torino
Dipartimento di Elettronica e Telecomunicazioni

Laboratory #3
Integrated Systems Architecture
MIPS-lite Processor

Master Degree in Electronic Engineering

Authors: Group 1

Alberto Aimaro 253196
Beatrice Bussolino 251190
Alessio Colucci 251197
Fabio Zanoni 232113

January 31, 2019

Contents

CHAPTER 1

MIPS-lite processor

1.1 Developed codes

In table 1.1 are reported the assembly commands supported by the developed MIPS processor. To each command is associated an OPCODE and a FUNC on 6 bits. The used values are found on the MIPS reference data.

ASM code	**Operation**	**OPCODE**$_{hex}$	**FUNC**$_{hex}$
andi	R[rt]=R[rs] & ZeroExtImm	C	
ori	R[rt]=R[rs] | ZeroExtImm	D	
xor	R[rd]=R[rs] xor R[rt]	0	26
sra	R[rd]=R[rt]>>shamt	0	3
add	R[rd]=R[rt]+R[rs]	0	20
addi	R[rt]=R[rs]+SignExtImm	8	
lui	R[rt]=Imm<<16	F	
slt	if (R[rs]<R[rt]) then R[rd]=1 else R[rd]=0	0	2A
sw	M[R[rs]+SignExtImm]=R[rt]	2B	
lw	R[rt]=M[R[rs]=+SignExtImm]	23	
beq	if (R[rs]=R[rt]) then PC=PC+4+BranchAddr BranchAddr=SignExtImm<<2	4	
jump	PC=JumpAddr JumpAddr=(PC+4)[31-28] & address & "00"	2	
nop		0	0

SignExtImm = {16{Immediate[15]}, Immediate}
ZeroExtImm = {16{1b'0}, Immediate}

Table 1.1: Subset of ASM commands implemented and relative codes

The OPCODEs and FUNCs are used by the control unit of the system that coherently sets the control signals for the ALU and the memories.

1.2 MIPS-lite processor structure

The MIPS processor is implemented with a 5 stages pipeline; an operation is divided in: instruction fetch, instruction decode, execution (arithmetic operations and addresses computation), data memory writing and write back in register file.

The branch calculation can be executed half in the EXecution stage and half in the MEMory stage, to have the final result in the WRiteback stage. This could be useful to increase the maximum frequency, since the critical path would be split. Since the longest path of the execution stage passes through the ALU, splitting the branch target calculation would only add more latency for the jump itself, leading to longer waiting times and more difficult handling of branches and jumps.

Hazards are managed by inserting `nop`s in the ASM code; this choice simplifies the hardware, avoiding the insertion of bypassing hardware.

In Figure 1.1, instruction memory and data memory appear; in VHDL code, these memories are not inserted in the code of the MIPS since they are not synthesizable.

1.3 Behavioral simulation

To test the processor, an assembly code is written. In particular, the code computes the absolute value of the minimum between a group of numbers stored in data memory.

The ASM code for a non-pipelined structure is the following:

./ASMcode_no_pipe.txt

```
Nm1=7
main:
    li $s0,Nm1        # load $s0 with Nm1
                      # Nm1 is the number of data to compare
    la $a0,v          # put in $a0 the address of v
                      # v is the address of the first data
    li $a1,0x1001001c # put in $a1 the address of m
                      # m is the address where to store the abs of the minimum
    li $t5,0x3fffffff # init $t5 with max pos

    # STORAGE OF DATA IN DATA MEMORY
    addi $8,$0,10
    sw $8,0($4)       #load 10 in memory
    addi $8,$0,-47
    sw $8,4($4)       #load -47 in memory
    addi $8,$0,22
    sw $8,8($4)       # load 22 in memory
    addi $8,$0,-3
    sw $8,12($4)      # load -3 in memory
    addi $8,$0,15
    sw $8,16($4)      # load 15 in memory
    addi $8,$0,27
    sw $8,20($4)      # load 27 in memory
    addi $8,$0,-4
    sw $8,24($4)      # load -4 in memory
    addi $8,$0,0
    sw $8,28($4)      # load 0 in m position

loop:   beq $s0,$0,done # check all elements have been tested
    lw $t0,0($a0)     # load new element in $t0
    sra $t1,$t0,31    # apply shift to get sign mask in $t1
    xor $t2,$t0,$t1 # $t2 = sign($t0)^$t0
```

```
    andi $t1,$t1,0x1 # $t1 &= 0x1 (carry in)
    add $t2,$t2,$t1 # $t2 += $t1 (add the carry in)
    add $a0,$a0,4   # point to next element
    sub $s0,$s0,1   # decrease $s0 by 1
    slt $t3,$t2,$t5 # $t3 = ($t2 < $t5) ? 1 : 0
    beq $t3,$0,loop # next element
    add $t5,$t2,$0  # update min
    j loop          # next element
done:   sw $t5,0($a1)   # store the result
endc:   j endc          # infinite loop
    nop
    nop
```

To avoid data hazards in the pipelined implementation, `nop` are inserted where necessary. To reduce the number of `nop`s, some operations have been reordered if possible.

The final code for a pipelined structure is the following:

./ASMcode_pipe.txt

```
Nm1=7
main:
    ori $16,$0,Nm1  # load $s0 with Nm1
            # Nm1 is the number of data to compare
    lui $4, 4097    # put in $a0 the address of v
    lui $6, 4097

    # DATA STORAGE IN DATA MEMORY
    addi $8,$0,10
    addi $9,$0,-47
    addi $10,$0,22
    ori $5, $6, 28  # put in $a1 the address of m
    lui $6, 16383   # init $t5 with max pos
    addi $11,$0,-3
    sw $8,0($4)     #load 10 in memory
    addi $8,$0,15
    ori $13, $6, 0xffff
    sw $9,4($4)     #load -47 in memory
    addi $9,$0,27
    sw $10,8($4)    # load 22 in memory
    addi $10,$0,-4
    sw $11,12($4)   # load -3 in memory
    sw $8,16($4)    # load 15 in memory
    sw $9,20($4)    # load 27 in memory
    sw $10,24($4)   # load -4 in memory

loop:   beq $s0,$0,done # check all elements have been tested
    lw $t0,0($a0)   # load new element in $t0
    nop
    nop
    nop
    sra $t1,$t0,31  # apply shift to get sign mask in $t1
    nop
    nop
    nop
    xor $t2,$t0,$t1 # $t2 = sign($t0)^$t0
    andi $t1,$t1,0x1 # $t1 &= 0x1 (carry in)
    nop
    add $a0,$a0,4   # point to next element
    addi $16,$16,-1 # decrease $s0 by 1
    add $t2,$t2,$t1 # $t2 += $t1 (add the carry in)
    nop
    nop
```

```
44      nop
45      slt  $t3,$t2,$t5 # $t3 = ($t2 < $t5) ? 1 : 0
46      nop
47      nop
48      nop
49      beq  $t3,$0,loop # next element
50      nop
51      nop
52      nop
53      add  $t5,$t2,$0  # update min
54      j loop           # next element
55      nop
56      nop
57      nop
58  done:   sw $t5,0($a1)   # store the result
59  endc:   j endc          # infinite loop
60      nop
61      nop
62      nop
63      nop
```

With the aid of PCSpim, the machine code is derived from the ASM code reported above. The codes are expressed in their binary representation: at the beginning of the simulation, the testbench loads these binary instructions in instruction memory.

After the execution of all the instructions, the content of data memory is analyzed to check the result.

Address (HEX)	Data (DEC)
0x10010000	10
0x10010004	-47
0x10010008	22
0x1001000c	-3
0x10010010	15
0x10010014	27
0x10010018	-4
0x1001001c	3

Table 1.2: Content of data memory when all the instructions have been executed. Location 0x1001001c contains the result of the code.

In Table 1.2 it is possible to see that the last data stored in memory is the absolute value of the minimum between all the other numbers in data memory.

The number of clock cycles needed to execute this code (except for the infinite loop at the end) is 288.

1.4 Synthesis

The MIPS processor is synthesized without instruction and data memories. The results are reported in Table 1.3.

The synthesized MIPS is tested with reference ASM code and it is verified that the content of data memory is the same as the one reported in Table 1.2.

Minimum period Maximum frequency	3.05 ns 327.87 MHz
$4 \cdot T$ $f/4$	12.20 ns 81.98 MHz
Area	13836.521484 μm^2
Total dynamic power Cell leakage power	953.0098 μW 313.1748 μW

Table 1.3: Synthesis results for MIPS-lite processor without ABS module

1.5 Place and Route

Place and Routing is applied to the synthesized design. The estimates of area and power are reported in Table 1.4.

Area	12757.6 μm^2
Total internal power	290.30292 μW
Total switching power	89.92408 μW
Total leakage power	234.43290 μW
Total power	614.6599 μW

Table 1.4: Place and Route results for MIPS-lite processor without ABS module

The MIPS is tested also after place and route operation.

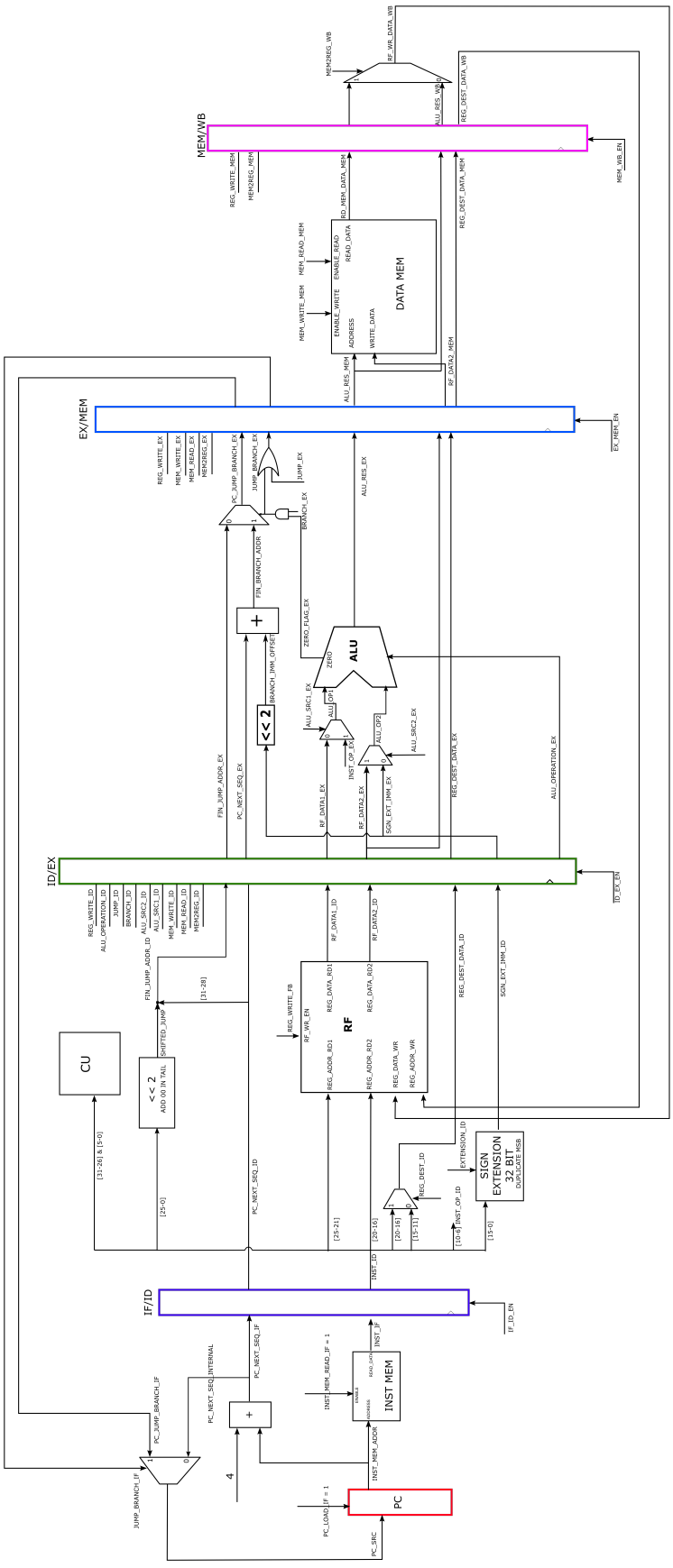

Figure 1.1: Block diagram of MIPS lite processor

CHAPTER 2

Absolute value module

A module that computes the absolute value of a number is inserted in the ALU. Therefore, a new operation is added in the assembly commands set, with an associated OPCODE.

ASM code	**Operation**	**OPCODE**$_{hex}$	**FUNC**$_{hex}$
`abs`	`R[rt]=R[rs] if R[rs]>=0 else R[rt]=-R[rs]`	14	

2.1 Functional simulation

The ASM code is modified to exploit the new available operation:

./ASMcode_pipe_abs.txt

```
Nm1=7
main:
    ori $16,$0,Nm1  # load $s0 with Nm1
    lui $4, 4097    # put in $a0 the address of v
    lui $6, 4097    # put in $a1 the address of m
    addi $8,$0,10
    addi $9,$0,-47
    addi $10,$0,22
    ori $5, $6, 28
    lui $6, 16383   # init $t5 with max pos
    addi $11,$0,-3
    sw $8,0($4)     #load 10 in memory
    addi $8,$0,15
    ori $13, $6, 0xffff
    sw $9,4($4)     #load -47 in memory
    addi $9,$0,27
    sw $10,8($4)    # load 22 in memory
    addi $10,$0,-4
    sw $11,12($4)   # load -3 in memory
    sw $8,16($4)    # load 15 in memory
    sw $9,20($4)    # load 27 in memory
    sw $10,24($4)   # load -4 in memory
loop:   beq $s0,$0,done # check all elements have been tested
    lw $t0,0($a0)   # load new element in $t0
    nop
    nop
    nop
```

```
28      addi $t2,$t0,0  # abs $t2, $t0  [0x510a0000], addi used to run on PCSpim
29      nop
30      add $a0,$a0,4   # point to next element
31      addi $16,$16,-1 # decrease $s0 by 1
32      slt $t3,$t2,$t5 # $t3 = ($t2 < $t5) ? 1 : 0
33      nop
34      nop
35      nop
36      beq $t3,$0,loop # next element
37      nop
38      nop
39      nop
40      add $t5,$t2,$0  # update min
41      j loop          # next element
42      nop
43      nop
44      nop
45  done:   sw $t5,0($a1)   # store the result
46  endc:   j endc          # infinite loop
47      nop
48      nop
49      nop
50      nop
```

Again, machine code is generated with the aid of PCSpim and the MIPS is tested. It is verified that the content of data memory is the same reported in Table 1.2. However the total time needed to execute the code (except the final infinite loop) is of 166 clock cycles, almost half of the noABS time, as it is expected since the longer operation (the absolute value) is optimized through a custom instruction.

2.2 Synthesis

The MIPS-lite processor with ABS module is synthesized and the results are reported in Table 2.1.

Minimum period Maximum frequency	3.05 ns 327.87 MHz
$4 \cdot T$ $f/4$	12.2 ns 81.98 MHz
Area	13921.375977 μm^2
Total dynamic power Cell leakage power	934.0355 μW 314.5584 μW

Table 2.1: Synthesis results for MIPS-lite processor with ABS module

The insertion of the ABS module slightly increases the total area. The power consumption is almost the same.

2.3 Place and Route

Place an routing of MIPS-lite processor with ABS module produces the results reported in Table 2.2.

Area	12846.5 μm^2
Total internal power	290.30292 μW
Total switching power	89.92408 μW
Total leakage power	234.43290 μW
Total power	614.65990 μW

Table 2.2: Place and Route results for MIPS-lite processor with ABS module

Appendices

APPENDIX A

Common MIPS-lite code

Here is all the common code for the two MIPS implementations, with all the testbenches.

./Code/MIPS_lite_common/data_memory.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library work;
use work.util.all;

entity data_memory is
port (clk : in std_logic;
    RSn : in std_logic;
    enable_write, enable_read : in std_logic;
    address : in std_logic_vector (Nbit_address-1 downto 0);
    write_data : in std_logic_vector (Nbit-1 downto 0);
    read_data : out std_logic_vector (Nbit-1 downto 0));
end entity data_memory;

architecture behavior of data_memory is

subtype word is std_logic_vector(word_length - 1 downto 0);
type mem is array(0 to (2 **(Nbit_address+PC_N_BIT_OFFSET)-1)) of word;

signal memory : mem;
begin

process (clk, RSn)
--  variable memory : mem;
begin
--  read_data <= (others=>'0');

    if (RSn='1') then
        if (clk'event and clk=clock_data_mem) then
            if (enable_write='1' and enable_read='0') then
                for i in 0 to (PC_OFFSET_INT - 1) loop
                    -- inverted for big endian
                    memory(to_integer(unsigned(address)) + PC_OFFSET_INT - 1 - i) <=
    write_data((i + 1) * word_length - 1 downto i * word_length);
                end loop;
            elsif (enable_write='0' and enable_read='1') then
                for i in 0 to (PC_OFFSET_INT - 1) loop
```

```
                    read_data((i + 1) * word_length - 1 downto i * word_length) <=
    memory(to_integer(unsigned(address)) + PC_OFFSET_INT - 1 - i);
                end loop;
            end if;
        end if;
    else
        memory <= (others => (others => '0'));
        read_data <= (others => '0');
    end if;
end process;

end architecture behavior;
```

./Code/MIPS_lite_common/datapath.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library work;
use work.util.all;

entity datapath is
    port (opcode : out std_logic_vector(OPCODE_LENGTH - 1 downto 0);
          func : out std_logic_vector(FUNCT_LENGTH - 1 downto 0);
          reg_dest : in std_logic;
          reg_write : in std_logic;
          ALU_src1 : in std_logic;
          ALU_src2 : in std_logic;
          extension : in std_logic;
          branch : in std_logic;
          jump : in std_logic;
          mem_write : in std_logic;
          mem_read : in std_logic;
          mem2reg : in std_logic;
          ALU_operation: in alu_opcode_states;

          inst_mem_rd_en : out std_logic; -- output for inst mem enable

          inst_mem_addr : out std_logic_vector(Nbit_address - 1 downto 0);
          inst_mem_data : in std_logic_vector(Nbit - 1 downto 0); -- output
    instruction memory data

          address : out std_logic_vector(Nbit_address - 1 downto 0);
          write_data : out std_logic_vector(Nbit - 1 downto 0);
          read_data : in std_logic_vector(Nbit - 1 downto 0);
          enable_write, enable_read : out std_logic;

          inst_mem_rd : in std_logic; -- always to 1
          pc_load : in std_logic; -- always to 1

          if_id_reg_en : in std_logic;
          id_ex_reg_en : in std_logic;
          ex_mem_reg_en : in std_logic;
          mem_wb_reg_en : in std_logic;

          clk : in std_logic;
          rstn : in std_logic
          );
end datapath;

architecture behav of datapath is
```

```
47      component dff
48          port (D : in std_logic;
49                clock, resetN, en : in std_logic;
50                Q : out std_logic);
51      end component;
52
53      component dff_alu_opcode
54          port (D : in alu_opcode_states;
55                clock, resetN, en : in std_logic;
56                Q : out alu_opcode_states);
57      end component;
58
59      component regn
60          generic (N : integer := 32);
61          port (D : in signed(N - 1 downto 0);
62                clock, resetN, en : in std_logic;
63                Q : out signed(N - 1 downto 0));
64      end component;
65
66      component regn_std_logic_vector
67          generic (N : integer := 32);
68          port (D : in std_logic_vector(N - 1 downto 0);
69                clock, resetN, en : in std_logic;
70                Q : out std_logic_vector(N - 1 downto 0));
71      end component;
72
73      component if_stage
74          port (inst_mem_addr : out std_logic_vector(Nbit_address - 1 downto 0);
75                inst_mem_data : in std_logic_vector(Nbit - 1 downto 0); -- output
    instruction memory data
76                inst_if : out std_logic_vector(Nbit - 1 downto 0);
77                pc_next_seq_if : out std_logic_vector(Nbit - 1 downto 0);
78                pc_jump_branch_if : in std_logic_vector(Nbit - 1 downto 0);
79
80                inst_mem_rd_if : in std_logic; -- always to 1
81                inst_mem_rd_en : out std_logic; -- output for inst mem enable
82                pc_load_if : in std_logic; -- load signal for PC, always to 1
83                jump_branch_if : in std_logic;
84
85                clk : in std_logic;
86                rstn : in std_logic
87                );
88      end component;
89
90      component id_stage
91          port (clk : in std_logic;
92              rstn : in std_logic;
93
94              cu_opcode_id : out std_logic_vector(OPCODE_LENGTH - 1 downto 0);
95              cu_funct_id : out std_logic_vector(FUNCT_LENGTH - 1 downto 0);
96
97              pc_next_seq_id : in std_logic_vector(Nbit - 1 downto 0);
98              inst_id : in std_logic_vector(Nbit - 1 downto 0);
99
100             --    reg_addr_rd1 : out std_logic_vector(bitNreg - 1 downto 0);
101             --    reg_addr_rd2 : out std_logic_vector(bitNreg - 1 downto 0);
102             --    reg_data_wr : out std_logic_vector(Nbit - 1 downto 0);
103             --    reg_addr_wr : out std_logic_vector(bitNreg - 1 downto 0);
104             --    rf_wr_en_id : out std_logic;
105             --    reg_data_rd1 : in std_logic_vector(Nbit - 1 downto 0);
106             --    reg_data_rd2 : in std_logic_vector(Nbit - 1 downto 0);
107             reg_write_id : in std_logic;
```

```
108             reg_data_wr_id : in std_logic_vector(Nbit - 1 downto 0);
109             reg_addr_wr_id : in std_logic_vector(bitNreg - 1 downto 0);
110 
111             reg_dest_id : in std_logic;
112             reg_dest_data_id : out std_logic_vector(bitNreg - 1 downto 0);
113 
114             rf_data1_id : out std_logic_vector(Nbit - 1 downto 0);
115             rf_data2_id : out std_logic_vector(Nbit - 1 downto 0);
116 
117             inst_op_id : out std_logic_vector(INST_OP_LENGTH - 1 downto 0);
118 
119             extension_id : in std_logic;
120             sgn_ext_imm_id : out std_logic_vector(Nbit - 1 downto 0);
121 
122             fin_jump_addr_id : out std_logic_vector(Nbit - 1 downto 0);
123             pc_next_seq_id_ex : out std_logic_vector(Nbit -1  downto 0)
124             );
125     end component;
126 
127     component ex_stage
128         port (clk : in std_logic;
129             rstn : in std_logic;
130 
131             sgn_ext_imm_ex : in std_logic_vector(Nbit - 1 downto 0);
132             rf_data1_ex : in std_logic_vector(Nbit - 1 downto 0);
133             rf_data2_ex : in std_logic_vector(Nbit - 1 downto 0);
134             pc_next_seq_ex : in std_logic_vector(Nbit - 1 downto 0);
135             fin_jump_addr_ex : in std_logic_vector(Nbit - 1 downto 0);
136             inst_op_ex : in std_logic_vector(INST_OP_LENGTH - 1 downto 0);
137 
138             alu_operation_ex : in alu_opcode_states;
139             alu_src1_ex : in std_logic;
140             alu_src2_ex : in std_logic;
141 
142             pc_jump_branch_ex : out std_logic_vector(Nbit - 1 downto 0);
143             alu_res_ex : out std_logic_vector(Nbit - 1 downto 0);
144 
145             branch_ex : in std_logic;
146             jump_ex : in std_logic;
147 
148             jump_branch_ex : out std_logic;
149 
150             reg_dest_data_ex_mem : out std_logic_vector(bitNreg - 1 downto 0);
151             reg_dest_data_ex : in std_logic_vector(bitNreg - 1 downto 0);
152             rf_data2_ex_mem : out std_logic_vector(Nbit - 1 downto 0)
153             );
154     end component;
155 
156     component mem_stage
157         port (clk : in std_logic;
158             rstn : in std_logic;
159 
160             mem_write_mem : in std_logic;
161             mem_read_mem : in std_logic;
162             alu_res_mem : in std_logic_vector(Nbit - 1 downto 0);
163             read_data : in std_logic_vector(Nbit - 1 downto 0);
164             rf_data2_mem : in std_logic_vector(Nbit - 1 downto 0);
165 
166             rd_mem_data_mem : out std_logic_vector(Nbit - 1 downto 0);
167             address : out std_logic_vector(Nbit_address - 1 downto 0);
168             write_data : out std_logic_vector(Nbit - 1 downto 0);
169             enable_write, enable_read : out std_logic;
```

```
170
171             alu_res_mem_wb : out std_logic_vector(Nbit - 1 downto 0);
172             reg_dest_data_mem_wb : out std_logic_vector(bitNreg - 1 downto 0);
173             reg_dest_data_mem : in std_logic_vector(bitNreg - 1 downto 0);
174             pc_jump_branch_mem : in std_logic_vector(Nbit - 1 downto 0);
175             pc_jump_branch_mem_wb : out std_logic_vector(Nbit - 1 downto 0);
176             jump_branch_mem : in std_logic;
177             jump_branch_mem_wb : out std_logic
178             );
179     end component;
180
181     component wb_stage
182         port (clk : in std_logic;
183             rstn : in std_logic;
184
185             rd_mem_data_wb : in std_logic_vector(Nbit - 1 downto 0);
186             alu_res_wb : in std_logic_vector(Nbit - 1 downto 0);
187             mem2reg_wb : in std_logic;
188
189             rf_wr_data_wb : out std_logic_vector(Nbit - 1 downto 0);
190
191             reg_dest_data_wb : in std_logic_vector(bitNreg - 1 downto 0);
192             reg_dest_data_fb : out std_logic_vector(bitNreg - 1 downto 0)
193             );
194     end component;
195
196     signal pipe_if_out : std_logic_vector(2 * Nbit - 1 downto 0);
197     signal pipe_id_in : std_logic_vector(2 * Nbit - 1 downto 0);
198     signal pipe_id_out : std_logic_vector(INST_OP_LENGTH + bitNreg + 5 * Nbit + 7
    downto 0);
199     signal pipe_ex_in : std_logic_vector(INST_OP_LENGTH + bitNreg + 5 * Nbit + 7
    downto 0);
200     signal pipe_ex_out : std_logic_vector(3 * Nbit + bitNreg + 4 downto 0);
201     signal pipe_mem_in : std_logic_vector(3 * Nbit + bitNreg + 4 downto 0);
202     signal pipe_mem_out : std_logic_vector(3 * Nbit + bitNreg + 2 downto 0);
203     signal pipe_wb_in : std_logic_vector(2 * Nbit + bitNreg + 1 downto 0);
204     signal pipe_wb_out : std_logic_vector(bitNreg + Nbit downto 0);
205
206     signal pipe_id_alu_op_out : alu_opcode_states;
207     signal pipe_ex_alu_op_in : alu_opcode_states;
208
209     -- *********
210     signal jmp_brch : std_logic;
211
212     begin
213         if_comp : if_stage port map
214                   (inst_mem_addr => inst_mem_addr,
215                    inst_mem_data => inst_mem_data,
216                    inst_if => pipe_if_out(Nbit - 1 downto 0),
217                    pc_next_seq_if  => pipe_if_out(2 * Nbit - 1 downto Nbit),
218
219                    inst_mem_rd_if => inst_mem_rd,
220                    inst_mem_rd_en => inst_mem_rd_en,
221                    pc_load_if => pc_load,
222                    pc_jump_branch_if => pipe_mem_out(3 * Nbit + bitNreg - 1 + 2 downto
     2 * Nbit + bitNreg + 2),
223                    jump_branch_if => pipe_mem_out(3 * Nbit + bitNreg - 1 + 3),
224
225                    clk => clk,
226                    rstn => rstn);
227
228         id_comp : id_stage port map
```

```
229                 (clk => clk,
230                 rstn => rstn,
231 
232                 cu_opcode_id => opcode,
233                 cu_funct_id  => func,
234 
235                 pc_next_seq_id => pipe_id_in(2 * Nbit - 1 downto Nbit),
236                 inst_id => pipe_id_in(Nbit - 1 downto 0),
237 
238                 reg_write_id => pipe_wb_out(bitNreg + Nbit),
239                 reg_data_wr_id => pipe_wb_out(Nbit - 1 downto 0),
240                 reg_addr_wr_id => pipe_wb_out(bitNreg - 1 + Nbit downto Nbit),
241 
242                 reg_dest_id => reg_dest,
243                 reg_dest_data_id => pipe_id_out(bitNreg - 1 + 5 * Nbit downto 5 *
    Nbit),
244 
245                 rf_data1_id => pipe_id_out(Nbit - 1 downto 0),
246                 rf_data2_id => pipe_id_out(2 * Nbit - 1 downto Nbit),
247 
248                 inst_op_id => pipe_id_out(INST_OP_LENGTH - 1 + bitNreg + 5 * Nbit
    downto 5 * Nbit + bitNreg),
249 
250                 extension_id => extension,
251                 sgn_ext_imm_id => pipe_id_out(3 * Nbit - 1 downto 2 * Nbit),
252 
253                 fin_jump_addr_id => pipe_id_out(4 * Nbit - 1 downto 3 * Nbit),
254                 pc_next_seq_id_ex => pipe_id_out(5 * Nbit - 1 downto 4 * Nbit)
255             );
256 
257         pipe_id_out(INST_OP_LENGTH + bitNreg + 5 * Nbit) <= mem2reg;
258         pipe_id_out(INST_OP_LENGTH + bitNreg + 5 * Nbit + 1) <= reg_write;
259         pipe_id_out(INST_OP_LENGTH + bitNreg + 5 * Nbit + 2) <= ALU_src1;
260         pipe_id_out(INST_OP_LENGTH + bitNreg + 5 * Nbit + 3) <= ALU_src2;
261         pipe_id_out(INST_OP_LENGTH + bitNreg + 5 * Nbit + 4) <= branch;
262         pipe_id_out(INST_OP_LENGTH + bitNreg + 5 * Nbit + 5) <= jump;
263         pipe_id_out(INST_OP_LENGTH + bitNreg + 5 * Nbit + 6) <= mem_write;
264         pipe_id_out(INST_OP_LENGTH + bitNreg + 5 * Nbit + 7) <= mem_read;
265         pipe_id_alu_op_out <= ALU_operation;
266 
267         ex_comp : ex_stage port map
268                  (clk => clk,
269                  rstn => rstn,
270 
271                  sgn_ext_imm_ex => pipe_ex_in(3 * Nbit - 1 downto 2 * Nbit),
272                  rf_data1_ex => pipe_ex_in(Nbit - 1 downto 0),
273                  rf_data2_ex => pipe_ex_in(2 * Nbit - 1 downto Nbit),
274 
275                  pc_next_seq_ex => pipe_ex_in(5 * Nbit - 1 downto 4 * Nbit),
276                  fin_jump_addr_ex => pipe_ex_in(4 * Nbit - 1 downto 3 * Nbit),
277                  inst_op_ex => pipe_ex_in(INST_OP_LENGTH - 1 + bitNreg + 5 * Nbit
    downto 5 * Nbit + bitNreg),
278 
279                  alu_operation_ex => pipe_ex_alu_op_in,
280                  alu_src1_ex => pipe_ex_in(INST_OP_LENGTH + bitNreg + 5 * Nbit + 2),
281                  alu_src2_ex => pipe_ex_in(INST_OP_LENGTH + bitNreg + 5 * Nbit + 3),
282 
283                  pc_jump_branch_ex => pipe_ex_out(Nbit - 1 downto 0),
284                  alu_res_ex => pipe_ex_out(2 * Nbit - 1 downto Nbit),
285 
286                  branch_ex => pipe_ex_in(INST_OP_LENGTH + bitNreg + 5 * Nbit + 4),
287                  jump_ex => pipe_ex_in(INST_OP_LENGTH + bitNreg + 5 * Nbit + 5),
```

```
288
289                 jump_branch_ex => pipe_ex_out(3 * Nbit + bitNreg + 4),
290
291                 reg_dest_data_ex_mem => pipe_ex_out(3 * Nbit + bitNreg - 1 downto 3
    * Nbit),
292                 reg_dest_data_ex => pipe_ex_in(bitNreg - 1 + 5 * Nbit downto 5 *
    Nbit),
293                 rf_data2_ex_mem => pipe_ex_out(3 * Nbit - 1 downto 2 * Nbit)
294         );
295     pipe_ex_out(3 * Nbit + bitNreg) <= pipe_ex_in(INST_OP_LENGTH + bitNreg + 5 *
    Nbit); -- mem2reg
296     pipe_ex_out(3 * Nbit + bitNreg + 1) <= pipe_ex_in(INST_OP_LENGTH + bitNreg + 5
     * Nbit + 1); -- reg write
297     pipe_ex_out(3 * Nbit + bitNreg + 2) <= pipe_ex_in(INST_OP_LENGTH + bitNreg + 5
     * Nbit + 6); -- mem write
298     pipe_ex_out(3 * Nbit + bitNreg + 3) <= pipe_ex_in(INST_OP_LENGTH + bitNreg + 5
     * Nbit + 7); -- mem read
299
300     mem_comp : mem_stage port map
301                 (clk => clk,
302                  rstn => rstn,
303
304                  mem_write_mem => pipe_mem_in(3 * Nbit + bitNreg + 2),
305                  mem_read_mem => pipe_mem_in(3 * Nbit + bitNreg + 3),
306                  alu_res_mem => pipe_mem_in(2 * Nbit - 1 downto Nbit),
307                  read_data => read_data,
308                  rf_data2_mem => pipe_mem_in(3 * Nbit - 1 downto 2 * Nbit),
309
310                  rd_mem_data_mem => pipe_mem_out(Nbit - 1 downto 0),
311                  address => address,
312                  write_data => write_data,
313                  enable_write => enable_write,
314                  enable_read => enable_read,
315
316                  alu_res_mem_wb => pipe_mem_out(2 * Nbit - 1 downto Nbit),
317                  reg_dest_data_mem_wb => pipe_mem_out(bitNreg - 1 + 2 * Nbit  downto
     2 * Nbit),
318                  reg_dest_data_mem => pipe_mem_in(3 * Nbit + bitNreg - 1 downto 3 *
    Nbit),
319                  pc_jump_branch_mem => pipe_mem_in(Nbit - 1 downto 0),
320                  pc_jump_branch_mem_wb => pipe_mem_out(3 * Nbit + bitNreg - 1 + 2
    downto 2 * Nbit + bitNreg + 2), -- not in pipe
321                  jump_branch_mem => pipe_mem_in(3 * Nbit + bitNreg + 4),
322                  jump_branch_mem_wb => pipe_mem_out(3 * Nbit + bitNreg - 1 + 3) --
    not in pipe
323         );
324
325     pipe_mem_out(2 * Nbit + bitNreg) <= pipe_mem_in(3 * Nbit + bitNreg); --
    mem2reg
326     pipe_mem_out(2 * Nbit + bitNreg + 1) <= pipe_mem_in(3 * Nbit + bitNreg + 1);
    -- reg write
327
328     wb_comp : wb_stage port map
329                 (clk => clk,
330                  rstn => rstn,
331
332                  rd_mem_data_wb => pipe_wb_in(Nbit - 1 downto 0),
333                  alu_res_wb => pipe_wb_in(2 * Nbit - 1 downto Nbit),
334                  mem2reg_wb => pipe_wb_in(2 * Nbit + bitNreg),
335
336                  rf_wr_data_wb => pipe_wb_out(Nbit - 1 downto 0),
337
```

```
338                     reg_dest_data_wb => pipe_wb_in(bitNreg - 1 + 2 * Nbit  downto 2 *
    Nbit),
339                     reg_dest_data_fb => pipe_wb_out(bitNreg - 1 + Nbit downto Nbit)
340             );
341
342         pipe_wb_out(bitNreg + Nbit) <= pipe_wb_in(2 * Nbit + bitNreg + 1); -- reg
    write
343
344         pipe_gen : if pipe_flag = true generate
345             pipe_if_id_reg : regn_std_logic_vector
346                                 generic map (N => pipe_if_out'length)
347                                 port map (clock => clk,
348                                           resetN => rstn,
349                                           en => if_id_reg_en,
350                                           Q => pipe_id_in,
351                                           D => pipe_if_out);
352             pipe_id_ex_reg : regn_std_logic_vector
353                                 generic map (N => pipe_id_out'length)
354                                 port map (clock => clk,
355                                           resetN => rstn,
356                                           en => id_ex_reg_en,
357                                           Q => pipe_ex_in,
358                                           D => pipe_id_out);
359             pipe_id_ex_dff_alu_op : dff_alu_opcode port map (
360                                     D => pipe_id_alu_op_out,
361                                     clock => clk,
362                                     resetN => rstn,
363                                     en => id_ex_reg_en,
364                                     Q => pipe_ex_alu_op_in);
365             pipe_ex_mem_reg : regn_std_logic_vector
366                                 generic map (N => pipe_ex_out'length)
367                                 port map (clock => clk,
368                                           resetN => rstn,
369                                           en => ex_mem_reg_en,
370                                           Q => pipe_mem_in,
371                                           D => pipe_ex_out);
372             pipe_mem_wb_reg : regn_std_logic_vector
373                                 generic map (N => pipe_wb_in'length)
374                                 port map (clock => clk,
375                                           resetN => rstn,
376                                           en => mem_wb_reg_en,
377                                           Q => pipe_wb_in,
378                                           D => pipe_mem_out(2 * Nbit + bitNreg + 1
    downto 0));
379         end generate;
380         no_pipe_gen : if pipe_flag = false generate
381             pipe_id_in <= pipe_if_out;
382             pipe_ex_in <= pipe_id_out;
383             pipe_ex_alu_op_in <= pipe_id_alu_op_out;
384             pipe_mem_in <= pipe_ex_out;
385             pipe_wb_in <= pipe_mem_out(2 * Nbit + bitNreg + 1 downto 0);
386         end generate;
387     end behav;
```

./Code/MIPS_lite_common/dff.vhd

```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.all;
3 USE ieee.numeric_std.all;
4
5 -- Flip Flop di tipo D, con parallelismo N e reset asincrono
6
```

```
ENTITY dff IS
PORT (D : IN STD_LOGIC;      -- ingresso
     Clock, Resetn, EN : IN STD_LOGIC;           -- clock, reset, enable
     Q : OUT STD_LOGIC);      -- uscita
END dff;

ARCHITECTURE Behavior OF dff IS
BEGIN
PROCESS (Clock)
BEGIN

     IF (Clock'EVENT AND Clock = '1') THEN  -- se c'è il fronte
     IF (resetn = '0') then
          q <= '0';
     elsIF (EN='1') THEN  -- e enable è attivo
          Q <= D;
          END IF;
END IF;
END PROCESS;
END Behavior;
```

./Code/MIPS_lite_common/dff_alu_opcode.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

-- Flip Flop di tipo D, con parallelismo N e reset asincrono

library work;
use work.util.all;

ENTITY dff_alu_opcode IS
PORT (D : IN alu_opcode_states;      -- ingresso
     Clock, Resetn, EN : IN STD_LOGIC;           -- clock, reset, enable
     Q : OUT alu_opcode_states);      -- uscita
END dff_alu_opcode;

ARCHITECTURE Behavior OF dff_alu_opcode IS
BEGIN
PROCESS (Clock)
BEGIN

     IF (Clock'EVENT AND Clock = '1') THEN  -- se c'è il fronte
     IF (resetn = '0') then
          q <= NOP;
     elsIF (EN='1') THEN  -- e enable è attivo
          Q <= D;
          END IF;
END IF;
END PROCESS;
END Behavior;
```

./Code/MIPS_lite_common/ex_stage.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library work;
use work.util.all;

```

```
8  entity ex_stage is
9      port (clk : in std_logic;
10           rstn : in std_logic;
11
12           sgn_ext_imm_ex : in std_logic_vector(Nbit - 1 downto 0);
13           rf_data1_ex : in std_logic_vector(Nbit - 1 downto 0);
14           rf_data2_ex : in std_logic_vector(Nbit - 1 downto 0);
15           pc_next_seq_ex : in std_logic_vector(Nbit - 1 downto 0);
16           fin_jump_addr_ex : in std_logic_vector(Nbit - 1 downto 0);
17           inst_op_ex : in std_logic_vector(INST_OP_LENGTH - 1 downto 0);
18
19           alu_operation_ex : in alu_opcode_states;
20           alu_src1_ex : in std_logic;
21           alu_src2_ex : in std_logic;
22
23           pc_jump_branch_ex : out std_logic_vector(Nbit - 1 downto 0);
24           alu_res_ex : out std_logic_vector(Nbit - 1 downto 0);
25
26           branch_ex : in std_logic;
27           jump_ex : in std_logic;
28
29           jump_branch_ex : out std_logic;
30
31           reg_dest_data_ex_mem : out std_logic_vector(bitNreg - 1 downto 0);
32           reg_dest_data_ex : in std_logic_vector(bitNreg - 1 downto 0);
33           rf_data2_ex_mem : out std_logic_vector(Nbit - 1 downto 0)
34
35
36
37           );
38 end ex_stage;
39
40 architecture behav of ex_stage is
41     component alu
42         generic (Nbit : integer := 32);
43         port (operand1, operand2 : in std_logic_vector(Nbit - 1 downto 0);
44             result : out std_logic_vector(Nbit - 1 downto 0);
45             zero : out std_logic;
46             alu_opcode : in alu_opcode_states);
47     end component;
48
49     signal alu_op1, alu_op2 : std_logic_vector(Nbit - 1 downto 0);
50     signal zero_flag_ex : std_logic;
51     signal branch_imm_offset : std_logic_vector(Nbit - 1 downto 0);
52     signal fin_branch_flag_ex : std_logic;
53     signal fin_seq_branch_addr : std_logic_vector(Nbit - 1 downto 0);
54     signal fin_branch_addr : std_logic_vector(Nbit - 1 downto 0);
55
56     begin
57         alu_comp : alu generic map (Nbit => Nbit)
58                        port map (operand1 => alu_op1,
59                                  operand2 => alu_op2,
60                                  result => alu_res_ex,
61                                  alu_opcode => alu_operation_ex,
62                                  zero => zero_flag_ex);
63
64         alu_op2 <= rf_data2_ex when alu_src2_ex = C_registers2 else
65                    sgn_ext_imm_ex;
66         process(rf_data1_ex, alu_src1_ex, inst_op_ex)
67         begin
68             if (alu_src1_ex = C_registers1) then
69                 alu_op1 <= rf_data1_ex;
```

```
70              else
71                  alu_op1(INST_OP_LENGTH - 1 downto 0) <= inst_op_ex;
72                  alu_op1(Nbit - 1 downto INST_OP_LENGTH) <= (others => '0');
73              end if;
74          end process;
75
76          branch_imm_offset(Nbit - 1 downto Nbit - POS_LEFT_SHIFT_LENGTH) <=
    sgn_ext_imm_ex(POS_LEFT_SHIFT_TOP downto POS_LEFT_SHIFT_BOTTOM);
77          branch_imm_offset(Nbit - POS_LEFT_SHIFT_LENGTH - 1 downto 0) <= (others =>
    '0');
78
79          -- signed sum for offset
80          fin_branch_addr <= std_logic_vector(signed(branch_imm_offset) + signed(
    pc_next_seq_ex));
81
82          fin_branch_flag_ex <= branch_ex and zero_flag_ex;
83
84          pc_jump_branch_ex <= fin_branch_addr
85                               when fin_branch_flag_ex = C_branch_yes else
86                               fin_jump_addr_ex;
87
88          jump_branch_ex <= fin_branch_flag_ex or jump_ex;
89
90          reg_dest_data_ex_mem <= reg_dest_data_ex;
91          rf_data2_ex_mem <= rf_data2_ex;
92
93      end behav;
```

./Code/MIPS_lite_common/id_stage.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  library work;
6  use work.util.all;
7
8  entity id_stage is
9      port (clk : in std_logic;
10           rstn : in std_logic;
11
12           cu_opcode_id : out std_logic_vector(OPCODE_LENGTH - 1 downto 0);
13           cu_funct_id : out std_logic_vector(FUNCT_LENGTH - 1 downto 0);
14
15           pc_next_seq_id : in std_logic_vector(Nbit - 1 downto 0);
16           inst_id : in std_logic_vector(Nbit - 1 downto 0);
17
18         --   reg_addr_rd1 : out std_logic_vector(bitNreg - 1 downto 0);
19         --   reg_addr_rd2 : out std_logic_vector(bitNreg - 1 downto 0);
20         --   reg_data_wr : out std_logic_vector(Nbit - 1 downto 0);
21         --   reg_addr_wr : out std_logic_vector(bitNreg - 1 downto 0);
22         --   rf_wr_en_id : out std_logic;
23         --   reg_data_rd1 : in std_logic_vector(Nbit - 1 downto 0);
24         --   reg_data_rd2 : in std_logic_vector(Nbit - 1 downto 0);
25           reg_write_id : in std_logic;
26           reg_data_wr_id : in std_logic_vector(Nbit - 1 downto 0);
27           reg_addr_wr_id : in std_logic_vector(bitNreg - 1 downto 0);
28
29           reg_dest_id : in std_logic;
30           reg_dest_data_id : out std_logic_vector(bitNreg - 1 downto 0);
31
32           rf_data1_id : out std_logic_vector(Nbit - 1 downto 0);
```

```
33          rf_data2_id : out std_logic_vector(Nbit - 1 downto 0);
34
35          inst_op_id : out std_logic_vector(INST_OP_LENGTH - 1 downto 0);
36
37          extension_id : in std_logic;
38          sgn_ext_imm_id : out std_logic_vector(Nbit - 1 downto 0);
39
40          fin_jump_addr_id : out std_logic_vector(Nbit - 1 downto 0);
41          pc_next_seq_id_ex : out std_logic_vector(Nbit -1  downto 0)
42          );
43  end id_stage;
44
45  architecture behav of id_stage is
46      component regfile
47          generic (bitNregaddr : integer := 5; -- 2 ** bitNregaddr is total number of
    regs
48                   Nbitdata : integer := 32);
49          port (addr_rd_reg1 : in std_logic_vector(bitNregaddr - 1 downto 0);
50              addr_rd_reg2 : in std_logic_vector(bitNregaddr - 1 downto 0);
51              addr_wr_reg : in std_logic_vector(bitNregaddr - 1 downto 0);
52              data_wr_reg : in std_logic_vector(Nbitdata - 1 downto 0);
53              data_rd_reg1 : out std_logic_vector(Nbitdata - 1 downto 0);
54              data_rd_reg2 : out std_logic_vector(Nbitdata - 1 downto 0);
55              write_en : in std_logic;
56              clk : in std_logic;
57              rstn : in std_logic);
58      end component;
59
60      component sign_extension
61          generic (IN_WIDTH : integer := 16;
62                   OUT_WIDTH : integer := 32);
63          port (data_in : in std_logic_vector (IN_WIDTH-1 downto 0);
64              data_out : out std_logic_vector (OUT_WIDTH-1 downto 0);
65              extension: in std_logic);
66      end component;
67
68      begin
69          rf : regfile generic map (bitNregaddr => bitNreg, Nbitdata => Nbit)
70                       port map (addr_rd_reg1 => inst_id(POS_REG1_ADDR_TOP downto
    POS_REG1_ADDR_BOTTOM),
71                                 addr_rd_reg2 => inst_id(POS_REG2_ADDR_TOP downto
    POS_REG2_ADDR_BOTTOM),
72                                 addr_wr_reg => reg_addr_wr_id,
73                                 data_wr_reg => reg_data_wr_id,
74                                 data_rd_reg1 => rf_data1_id,
75                                 data_rd_reg2 => rf_data2_id,
76                                 write_en => reg_write_id,
77                                 clk => clk,
78                                 rstn => rstn);
79
80          se : sign_extension generic map (IN_WIDTH => IMMEDIATE_LENGTH,
81                                           OUT_WIDTH => Nbit)
82                              port map (data_in => inst_id(POS_IMMEDIATE_TOP downto
    POS_IMMEDIATE_BOTTOM),
83                                        data_out => sgn_ext_imm_id,
84                                        extension => extension_id);
85
86
87          reg_dest_data_id <= inst_id(POS_REG_DEST_ADDR1_TOP downto
    POS_REG_DEST_ADDR1_BOTTOM)
88                              when reg_dest_id = C_i_20_16 else
89                              inst_id(POS_REG_DEST_ADDR2_TOP downto
```

```
    POS_REG_DEST_ADDR2_BOTTOM);
90
91          inst_op_id <= inst_id(POS_INST_OP_TOP downto POS_INST_OP_BOTTOM);
92
93          fin_jump_addr_id(Nbit - 1 downto Nbit - POS_PC_MSB_LENGTH -
    POS_JUMP_ADDRESS_LENGTH) <= pc_next_seq_id(POS_PC_MSB_TOP downto POS_PC_MSB_BOTTOM
    ) & inst_id(POS_JUMP_ADDRESS_TOP downto POS_JUMP_ADDRESS_BOTTOM);
94          fin_jump_addr_id(Nbit - POS_PC_MSB_LENGTH - POS_JUMP_ADDRESS_LENGTH - 1 downto
     0) <= (others => '0');
95
96          cu_opcode_id <= inst_id(POS_OPCODE_TOP downto POS_OPCODE_BOTTOM);
97          cu_funct_id <= inst_id(POS_FUNCT_TOP downto POS_FUNCT_BOTTOM);
98
99          pc_next_seq_id_ex <= pc_next_seq_id;
100     end behav;
```

./Code/MIPS_lite_common/if_stage.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  library work;
6  use work.util.all;
7
8  entity if_stage is
9      port (inst_mem_addr : out std_logic_vector(Nbit_address - 1 downto 0);
10           inst_mem_data : in std_logic_vector(Nbit - 1 downto 0); -- output
     instruction memory data
11           inst_if : out std_logic_vector(Nbit - 1 downto 0);
12           pc_next_seq_if : out std_logic_vector(Nbit - 1 downto 0);
13           pc_jump_branch_if : in std_logic_vector(Nbit - 1 downto 0);
14
15           inst_mem_rd_if : in std_logic; -- always to 1
16           inst_mem_rd_en : out std_logic; -- output for inst mem enable
17           pc_load_if : in std_logic; -- load signal for PC, always to 1
18           jump_branch_if : in std_logic;
19
20           clk : in std_logic;
21           rstn : in std_logic
22           );
23 end if_stage;
24
25 architecture behav of if_stage is
26     component regn_std_logic_vector
27         generic (N : integer := 32);
28         port (D : in std_logic_vector(N - 1 downto 0);
29               clock, resetN, en : in std_logic;
30               Q : out std_logic_vector(N - 1 downto 0));
31     end component;
32
33     component instruction_memory
34         port (clk : in std_logic;
35             RSn : in std_logic;
36             enable : in std_logic;
37             address : in std_logic_vector (Nbit_address-1 downto 0);
38             read_data : out std_logic_vector (Nbit-1 downto 0));
39     end component;
40
41     signal pc_out : std_logic_vector(Nbit - 1 downto 0);
42     signal pc_src : std_logic_vector(Nbit - 1 downto 0);
43     signal clkN : std_logic;
```

```
44      signal pc_next_seq_internal : std_logic_vector(Nbit - 1 downto 0);
45
46      begin
47          clkN <= clk xnor clock_PC;
48          pc_reg : regn_std_logic_vector
49                      generic map (N => Nbit)
50                      port map (D => pc_src,
51                              clock => clkN,
52                              resetN => rstn,
53                              en => pc_load_if,
54                              Q => pc_out);
55
56          -- unsigned sum for sequential address
57          pc_next_seq_internal <= std_logic_vector(unsigned(pc_out) + PC_OFFSET);
58          pc_next_seq_if <= pc_next_seq_internal;
59
60          pc_src <= pc_jump_branch_if when jump_branch_if = '1' else
    pc_next_seq_internal;
61
62          -- for instruction memory
63          inst_if <= inst_mem_data;
64          inst_mem_rd_en <= inst_mem_rd_if;
65          inst_mem_addr <= pc_out(Nbit_address - 1 downto 0);
66
67      end behav;
```

./Code/MIPS_lite_common/instruction_memory.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  library work;
6  use work.util.all;
7
8  entity instruction_memory is
9  port (clk : in std_logic;
10       RSn : in std_logic;
11       rd_enable : in std_logic;
12       rd_address : in std_logic_vector (Nbit_address-1 downto 0);
13       read_data : out std_logic_vector (Nbit-1 downto 0);
14       wr_address : in std_logic_vector(Nbit_address - 1 downto 0);
15       write_data : in std_logic_vector(Nbit - 1 downto 0);
16       wr_enable : in std_logic);
17 end entity instruction_memory;
18
19 architecture behavior of instruction_memory is
20
21 subtype word is std_logic_vector(word_length - 1 downto 0);
22 type memory is array (0 to 2**(length_memory+PC_N_BIT_OFFSET)-1) of word;
23
24 begin
25
26 process (clk, RSn)
27     variable mem0 : memory;
28 begin
29 --read_data <= (others=>'0');
30
31     if (RSn='1') then
32         if (clk'event and clk=clock_inst_mem) then
33             if (rd_enable='1' and wr_enable = '0') then
34                 -- inverted for big endian
```

```
35                  for i in 0 to (PC_OFFSET_INT - 1) loop
36                      read_data((i + 1) * word_length - 1 downto i * word_length) <=
    mem0(to_integer(unsigned(rd_address)) + PC_OFFSET_INT - 1 - i);
37                  end loop;
38              elsif (rd_enable='0' and wr_enable = '1') then
39                  -- inverted for big endian
40                  for i in 0 to (PC_OFFSET_INT - 1) loop
41                      mem0(to_integer(unsigned(wr_address)) + PC_OFFSET_INT - 1 - i) :=
    write_data((i + 1) * word_length - 1 downto i * word_length);
42                  end loop;
43              end if;
44          end if;
45      else
46          mem0 := (others => (others => '0'));
47          read_data <= (others => '0');
48      end if;
49  end process;
50
51  end architecture behavior;
```

./Code/MIPS_lite_common/mem_stage.vhd

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   library work;
6   use work.util.all;
7
8   entity mem_stage is
9       port (clk : in std_logic;
10            rstn : in std_logic;
11
12            mem_write_mem : in std_logic;
13            mem_read_mem : in std_logic;
14            alu_res_mem : in std_logic_vector(Nbit - 1 downto 0);
15            read_data : in std_logic_vector(Nbit - 1 downto 0);
16            rf_data2_mem : in std_logic_vector(Nbit - 1 downto 0);
17
18            rd_mem_data_mem : out std_logic_vector(Nbit - 1 downto 0);
19            address : out std_logic_vector(Nbit_address - 1 downto 0);
20            write_data : out std_logic_vector(Nbit - 1 downto 0);
21            enable_write, enable_read : out std_logic;
22
23            alu_res_mem_wb : out std_logic_vector(Nbit - 1 downto 0);
24            reg_dest_data_mem_wb : out std_logic_vector(bitNreg - 1 downto 0);
25            reg_dest_data_mem : in std_logic_vector(bitNreg - 1 downto 0);
26            pc_jump_branch_mem : in std_logic_vector(Nbit - 1 downto 0);
27            pc_jump_branch_mem_wb : out std_logic_vector(Nbit - 1 downto 0);
28            jump_branch_mem : in std_logic;
29            jump_branch_mem_wb : out std_logic
30            );
31  end mem_stage;
32
33  architecture behav of mem_stage is
34      component data_memory
35          port (clk : in std_logic;
36              RSn : in std_logic;
37              enable_write, enable_read : in std_logic;
38              address : in std_logic_vector (Nbit_address-1 downto 0);
39              write_data : in std_logic_vector (Nbit-1 downto 0);
40              read_data : out std_logic_vector (Nbit-1 downto 0));
```

```
    end component;
    begin
        enable_write <= mem_write_mem;
        enable_read <= mem_read_mem;

        address <= alu_res_mem(Nbit_address - 1 downto 0);
        rd_mem_data_mem <= read_data;

        write_data <= rf_data2_mem;

        alu_res_mem_wb <= alu_res_mem;
        reg_dest_data_mem_wb <= reg_dest_data_mem;
        pc_jump_branch_mem_wb <= pc_jump_branch_mem;
        jump_branch_mem_wb <= jump_branch_mem;

    end behav;
```

./Code/MIPS_lite_common/mips_lite.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library work;
use work.util.all;

entity mips_lite is
    port (inst_mem_rd_en : out std_logic; -- output for inst mem enable
          inst_mem_addr : out std_logic_vector(Nbit_address - 1 downto 0);
          inst_mem_data : in std_logic_vector(Nbit - 1 downto 0); -- output
    instruction memory data

          address : out std_logic_vector(Nbit_address - 1 downto 0);
          write_data : out std_logic_vector(Nbit - 1 downto 0);
          read_data : in std_logic_vector(Nbit - 1 downto 0);
          enable_write, enable_read : out std_logic;

          clk : in std_logic;
          rstn : in std_logic
          );
end mips_lite;

architecture behav of mips_lite is
    component datapath
        port (opcode : out std_logic_vector(OPCODE_LENGTH - 1 downto 0);
            func : out std_logic_vector(FUNCT_LENGTH - 1 downto 0);
            reg_dest : in std_logic;
            reg_write : in std_logic;
            ALU_src1 : in std_logic;
            ALU_src2 : in std_logic;
            extension : in std_logic;
            branch : in std_logic;
            jump : in std_logic;
            mem_write : in std_logic;
            mem_read : in std_logic;
            mem2reg : in std_logic;
            ALU_operation: in alu_opcode_states;

            inst_mem_rd_en : out std_logic; -- output for inst mem enable

            inst_mem_addr : out std_logic_vector(Nbit_address - 1 downto 0);
            inst_mem_data : in std_logic_vector(Nbit - 1 downto 0); -- output
```

```
    instruction memory data
43
44              address : out std_logic_vector(Nbit_address - 1 downto 0);
45              write_data : out std_logic_vector(Nbit - 1 downto 0);
46              read_data : in std_logic_vector(Nbit - 1 downto 0);
47              enable_write, enable_read : out std_logic;
48
49              inst_mem_rd : in std_logic; -- always to 1
50              pc_load : in std_logic; -- always to 1
51
52              if_id_reg_en : in std_logic;
53              id_ex_reg_en : in std_logic;
54              ex_mem_reg_en : in std_logic;
55              mem_wb_reg_en : in std_logic;
56
57              clk : in std_logic;
58              rstn : in std_logic
59              );
60      end component;
61
62      component control_unit
63          port (opcode : in std_logic_vector (OPCODE_LENGTH - 1 downto 0);
64                  func : in std_logic_vector (FUNCT_LENGTH - 1 downto 0);
65                  reg_dest : out std_logic;
66                  reg_write : out std_logic;
67                  ALU_src1 : out std_logic;
68                  ALU_src2 : out std_logic;
69                  extension : out std_logic;
70                  branch : out std_logic;
71                  jump : out std_logic;
72                  mem_write : out std_logic;
73                  mem_read : out std_logic;
74                  mem2reg : out std_logic;
75                  ALU_operation: out alu_opcode_states);
76      end component;
77
78      signal opcode : std_logic_vector(OPCODE_LENGTH - 1 downto 0);
79      signal func :  std_logic_vector(FUNCT_LENGTH - 1 downto 0);
80      signal reg_dest : std_logic;
81      signal reg_write : std_logic;
82      signal ALU_src1 : std_logic;
83      signal ALU_src2 : std_logic;
84      signal extension : std_logic;
85      signal branch : std_logic;
86      signal jump : std_logic;
87      signal mem_write : std_logic;
88      signal mem_read : std_logic;
89      signal mem2reg : std_logic;
90      signal ALU_operation: alu_opcode_states;
91
92      signal inst_mem_rd : std_logic; -- always to 1
93      signal pc_load : std_logic; -- always to 1
94
95      signal if_id_reg_en : std_logic;
96      signal id_ex_reg_en : std_logic;
97      signal ex_mem_reg_en : std_logic;
98      signal mem_wb_reg_en : std_logic;
99
100     begin
101         datapath_comp : datapath port map
102                         (opcode => opcode,
103                          func => func,
```

```
104                         reg_dest => reg_dest,
105                         reg_write => reg_write,
106                         ALU_src1 => ALU_src1,
107                         ALU_src2 => ALU_src2,
108                         extension => extension,
109                         branch => branch,
110                         jump => jump,
111                         mem_write => mem_write,
112                         mem_read => mem_read,
113                         mem2reg => mem2reg,
114                         ALU_operation => ALU_operation,
115                         inst_mem_rd_en => inst_mem_rd_en,
116                         inst_mem_addr => inst_mem_addr,
117                         inst_mem_data => inst_mem_data,
118                         address => address,
119                         write_data => write_data,
120                         read_data => read_data,
121                         enable_write => enable_write,
122                         enable_read => enable_read,
123                         inst_mem_rd => inst_mem_rd,
124                         pc_load => pc_load,
125                         if_id_reg_en => if_id_reg_en,
126                         id_ex_reg_en => id_ex_reg_en,
127                         ex_mem_reg_en => ex_mem_reg_en,
128                         mem_wb_reg_en => mem_wb_reg_en,
129                         clk => clk,
130                         rstn => rstn);
131
132         control_unit_comp : control_unit port map
133                             (opcode => opcode,
134                             func => func,
135                             reg_dest => reg_dest,
136                             reg_write => reg_write,
137                             ALU_src1 => ALU_src1,
138                             ALU_src2 => ALU_src2,
139                             extension => extension,
140                             branch => branch,
141                             jump => jump,
142                             mem_write => mem_write,
143                             mem_read => mem_read,
144                             mem2reg => mem2reg,
145                             ALU_operation => ALU_operation);
146
147         inst_mem_rd <= '1' when rstn = '1' else '0';
148         pc_load <= '1' when rstn = '1' else '0';
149         if_id_reg_en <= '1' when rstn = '1' else '0';
150         id_ex_reg_en <= '1' when rstn = '1' else '0';
151         ex_mem_reg_en <= '1' when rstn = '1' else '0';
152         mem_wb_reg_en <= '1' when rstn = '1' else '0';
153
154     end behav;
```

./Code/MIPS_lite_common/mips_lite_with_mem.vhd

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 library work;
6 use work.util.all;
7
8 entity mips_lite_with_mem is
```

```
9       port (clk : in std_logic;
10            rstn : in std_logic;
11            inst_mem_rstn : in std_logic;
12            inst_mem_wr_en : in std_logic;
13            inst_mem_wr_addr : in std_logic_vector(Nbit_address - 1 downto 0);
14            inst_mem_wr_data : in std_logic_vector(Nbit - 1 downto 0));
15  end mips_lite_with_mem;
16
17  architecture behav of mips_lite_with_mem is
18      component data_memory
19          port (clk : in std_logic;
20                RSn : in std_logic;
21                enable_write, enable_read : in std_logic;
22                address : in std_logic_vector (Nbit_address-1 downto 0);
23                write_data : in std_logic_vector (Nbit-1 downto 0);
24                read_data : out std_logic_vector (Nbit-1 downto 0));
25      end component;
26
27      component instruction_memory
28          port (clk : in std_logic;
29                RSn : in std_logic;
30                rd_enable : in std_logic;
31                rd_address : in std_logic_vector (Nbit_address-1 downto 0);
32                read_data : out std_logic_vector (Nbit-1 downto 0);
33                wr_address : in std_logic_vector(Nbit_address - 1 downto 0);
34                write_data : in std_logic_vector(Nbit - 1 downto 0);
35                wr_enable : in std_logic);
36      end component;
37
38      component mips_lite
39          port (inst_mem_rd_en : out std_logic; -- output for inst mem enable
40                inst_mem_addr : out std_logic_vector(Nbit_address - 1 downto 0);
41                inst_mem_data : in std_logic_vector(Nbit - 1 downto 0); -- output
        instruction memory data
42
43                address : out std_logic_vector(Nbit_address - 1 downto 0);
44                write_data : out std_logic_vector(Nbit - 1 downto 0);
45                read_data : in std_logic_vector(Nbit - 1 downto 0);
46                enable_write, enable_read : out std_logic;
47
48                clk : in std_logic;
49                rstn : in std_logic
50                );
51      end component;
52
53      signal inst_mem_rd_en : std_logic;
54      signal inst_mem_addr : std_logic_vector(Nbit_address - 1 downto 0);
55      signal inst_mem_data : std_logic_vector(Nbit - 1 downto 0);
56
57      signal address : std_logic_vector(Nbit_address - 1 downto 0);
58      signal write_data : std_logic_vector(Nbit - 1 downto 0);
59      signal read_data : std_logic_vector(Nbit - 1 downto 0);
60      signal enable_write : std_logic;
61      signal enable_read : std_logic;
62
63      begin
64          inst_mem : instruction_memory port map
65                     (clk => clk,
66                      RSn => inst_mem_rstn,
67                      rd_enable => inst_mem_rd_en,
68                      rd_address => inst_mem_addr,
69                      read_data => inst_mem_data,
```

```
                    wr_address => inst_mem_wr_addr,
                    wr_enable => inst_mem_wr_en,
                    write_data => inst_mem_wr_data);

    data_mem : data_memory port map
               (clk => clk,
                RSn => inst_mem_rstn,
                enable_write => enable_write,
                enable_read => enable_read,
                address => address,
                write_data => write_data,
                read_data => read_data);

    mips_comp : mips_lite port map
                (inst_mem_rd_en => inst_mem_rd_en,
                inst_mem_addr => inst_mem_addr,
                inst_mem_data  => inst_mem_data,

                address => address,
                write_data => write_data,
                read_data => read_data,
                enable_write => enable_write,
                enable_read => enable_read,

                clk => clk,
                rstn => rstn);


end behav;
```

./Code/MIPS_lite_common/regfile.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity regfile is
    generic (bitNregaddr : integer := 5; -- 2 ** bitNregaddr is total number of regs
             Nbitdata : integer := 32);
    port (addr_rd_reg1 : in std_logic_vector(bitNregaddr - 1 downto 0);
          addr_rd_reg2 : in std_logic_vector(bitNregaddr - 1 downto 0);
          addr_wr_reg : in std_logic_vector(bitNregaddr - 1 downto 0);
          data_wr_reg : in std_logic_vector(Nbitdata - 1 downto 0);
          data_rd_reg1 : out std_logic_vector(Nbitdata - 1 downto 0);
          data_rd_reg2 : out std_logic_vector(Nbitdata - 1 downto 0);
          write_en : in std_logic;
          clk : in std_logic;
          rstn : in std_logic);
end regfile;

architecture behav of regfile is
    component regn_std_logic_vector
        generic (N : integer := 32);
        port (D : in std_logic_vector(N - 1 downto 0);
              clock, resetN, en : in std_logic;
              Q : out std_logic_vector(N - 1 downto 0));
    end component;

    type reg_sig_buf is array(2 ** bitNregaddr - 1 downto 0) of
                        std_logic_vector(Nbitdata - 1 downto 0);
    signal data_out : reg_sig_buf;
    signal wr_en_reg : std_logic_vector(2 ** bitNregaddr - 1 downto 0);
```

```
31
32      begin
33          reg_gen : for i in 0 to 2 ** bitNregaddr - 1 generate
34              reg_comp : regn_std_logic_vector
35                              generic map (N => Nbitdata)
36                              port map (D => data_wr_reg,
37                                        clock => clk,
38                                        resetN => rstn,
39                                        en => wr_en_reg(i),
40                                        Q => data_out(i));
41          end generate;
42
43          write_enable : process(addr_wr_reg, write_en)
44              variable temp_wr_en_reg :
45                              std_logic_vector(2 ** bitNregaddr - 1 downto 0);
46          begin
47              temp_wr_en_reg := (others => '0');
48              temp_wr_en_reg(to_integer(unsigned(addr_wr_reg))) := '1';
49              if (write_en = '1') then
50                  wr_en_reg <= temp_wr_en_reg;
51              else
52                  wr_en_reg <= (others => '0');
53              end if;
54          end process;
55
56          output_selection1 : process(addr_rd_reg1, data_out)
57          begin
58              data_rd_reg1 <= data_out(to_integer(unsigned(addr_rd_reg1)));
59          end process;
60
61          output_selection2 : process(addr_rd_reg2, data_out)
62          begin
63              data_rd_reg2 <= data_out(to_integer(unsigned(addr_rd_reg2)));
64          end process;
65      end behav;
```

./Code/MIPS_lite_common/regn.vhd

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  USE ieee.numeric_std.all;
4
5  -- Flip Flop di tipo D, con parallelismo N e reset asincrono
6
7  ENTITY regn IS
8  GENERIC (N: INTEGER := 32);                    -- numero di bit del registro
9  PORT (D : IN SIGNED (N-1 DOWNTO 0);      -- ingresso
10     Clock, Resetn, EN : IN STD_LOGIC;          -- clock, reset, enable
11     Q : OUT SIGNED (N-1 DOWNTO 0));      -- uscita
12 END regn;
13
14 ARCHITECTURE Behavior OF regn IS
15 BEGIN
16 PROCESS (Clock, Resetn)
17 BEGIN
18
19     IF (Clock'EVENT AND Clock = '1') THEN  -- se c'è il fronte
20     IF (resetn = '0') then
21         q <= (others => '0');
22     elsIF (EN='1') THEN  -- e enable è attivo
23         Q <= D;
24         END IF;
```

```
END IF;
END PROCESS;
END Behavior;
```

./Code/MIPS_lite_common/regn_std_logic_vector.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

-- Flip Flop di tipo D, con parallelismo N e reset asincrono

ENTITY regn_std_logic_vector IS
GENERIC (N: INTEGER := 32);                         -- numero di bit del registro
PORT (D : IN std_logic_vector (N-1 DOWNTO 0);       -- ingresso
     Clock, Resetn, EN : IN STD_LOGIC;              -- clock, reset, enable
     Q : OUT std_logic_vector (N-1 DOWNTO 0));      -- uscita
END regn_std_logic_vector;

ARCHITECTURE Behavior OF regn_std_logic_vector IS
BEGIN
PROCESS (Clock, Resetn)
BEGIN

     IF (Clock'EVENT AND Clock = '1') THEN  -- se c'è il fronte
     IF (resetn = '0') then
          q <= (others => '0');
     elsIF (EN='1') THEN  -- e enable è attivo
          Q <= D;
          END IF;
END IF;
END PROCESS;
END Behavior;
```

./Code/MIPS_lite_common/sign_extension.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity sign_extension is
generic (IN_WIDTH : integer := 16;
         OUT_WIDTH : integer := 32);
port (data_in : in std_logic_vector (IN_WIDTH-1 downto 0);
     data_out : out std_logic_vector (OUT_WIDTH-1 downto 0);
     extension: in std_logic);
end entity sign_extension;

architecture behavior of sign_extension is

signal extension_sign, extension_zero : std_logic_vector (OUT_WIDTH-1 downto 0);

begin

     extension_sign(IN_WIDTH-1 downto 0) <= data_in;
     extension_sign(OUT_WIDTH-1 downto IN_WIDTH) <= (others => data_in(IN_WIDTH-1));

     extension_zero(IN_WIDTH-1 downto 0) <= data_in;
     extension_zero(OUT_WIDTH-1 downto IN_WIDTH) <= (others => '0');

     data_out <= extension_zero when extension='0'
             else extension_sign;
```

```

end architecture behavior;
```

./Code/MIPS_lite_common/tb_mips_lite.v

```
//`timescale 1ns


module tb_mips_lite ();

    wire A, clk, rstn, inst_mem_rstn, inst_mem_wr_en ;
    wire [8:0] inst_mem_wr_addr;
    wire [31:0] inst_mem_wr_data;
    wire inst_mem_rd_en;
     wire [8:0] inst_mem_addr;
     wire [31:0] inst_mem_data;

     wire [8:0] address;
     wire [31:0] write_data;
     wire [31:0] read_data;
     wire enable_write;
     wire enable_read;

    testbench TB (.A(A), .clk(clk), .rstn(rstn),
         .inst_mem_rstn(inst_mem_rstn), .inst_mem_wr_en(inst_mem_wr_en),
         .inst_mem_wr_addr(inst_mem_wr_addr), .inst_mem_wr_data(inst_mem_wr_data));

    instruction_memory inst_mem
                      (.clk(clk),
                       .RSn(inst_mem_rstn),
                       .rd_enable(inst_mem_rd_en),
                       .rd_address(inst_mem_addr),
                       .read_data(inst_mem_data),
                       .wr_address(inst_mem_wr_addr),
                       .wr_enable(inst_mem_wr_en),
                       .write_data(inst_mem_wr_data));

     data_memory data_mem
                      (.clk(clk),
                       .RSn(inst_mem_rstn),
                       .enable_write(enable_write),
                       .enable_read(enable_read),
                       .address(address),
                       .write_data(write_data),
                       .read_data(read_data));
    mips_lite UUT
                       (.inst_mem_rd_en(inst_mem_rd_en),
                       .inst_mem_addr(inst_mem_addr),
                       .inst_mem_data(inst_mem_data),

                       .address(address),
                       .write_data(write_data),
                       .read_data(read_data),
                       .enable_write(enable_write),
                       .enable_read(enable_read),
              .clk(clk),
                       .rstn(rstn));

    initial begin
    $read_lib_saif("../saif/NangateOpenCellLibrary.saif");
    $set_gate_level_monitoring("on");
    $set_toggle_region(UUT);
```

```
58      $toggle_start;
59      end
60
61      always @ ( A ) begin
62      if (A) begin
63      $toggle_stop;
64      $toggle_report("../saif/mips_lite_back.saif", 1.0e-9, "tb_mips_lite.UUT");
65      $stop;
66      end
67      end
68
69  endmodule
```

./Code/MIPS_lite_common/util.vhd

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   package util is
6
7       constant pipe_flag_logic : std_logic := '1';
8
9
10
11
12      constant pipe_flag : boolean := pipe_flag_logic = '1';
13
14      constant Nbit : integer := 32;
15      constant bitNreg : integer := 5;
16
17      constant Nbit_address : integer := 9;
18
19      type alu_opcode_states is (NOP, ADDOP, ANDOP, OROP, SHIFTRIGHTOP, XOROP,
20                                 SETLESSTHAN, SUBOP, SHIFTLEFT16, ABSOP);
21
22      constant alu_opcode_length : integer :=
23                           alu_opcode_states'pos(alu_opcode_states'right) + 1;
24
25      constant rising : std_logic := '1';
26      constant falling : std_logic := '0';
27
28      constant clock_inst_mem : std_logic := rising xor pipe_flag_logic;
29      constant clock_PC : std_logic := falling xor pipe_flag_logic;
30
31      constant clock_data_mem : std_logic := falling;
32
33      -- reg_dst
34      constant C_i_20_16 : STD_LOGIC := '1';
35      constant C_i_15_11 : STD_LOGIC := '0';
36
37      -- reg_write
38      constant C_reg_enable : STD_LOGIC := '1';
39      constant C_reg_disable : STD_LOGIC := '0';
40
41      -- ALU_src1
42      constant C_registers1 : STD_LOGIC := '0';
43      constant C_i_10_6 : STD_LOGIC := '1';
44
45      -- ALU src2
46      constant C_registers2 : STD_LOGIC := '1';
47      constant C_sign_extension : STD_LOGIC := '0';
```

```
48
49      -- branch
50      constant c_branch_yes : STD_LOGIC := '1';
51      constant C_branch_no : STD_LOGIC := '0';
52
53      -- jump
54      constant C_jump_yes : STD_LOGIC := '1';
55      constant C_jump_no : STD_LOGIC := '0';
56
57      -- mem_write
58      constant C_mw_enable : STD_LOGIC := '1';
59      constant C_mw_disable : STD_LOGIC := '0';
60
61      -- mem_read
62      constant C_mr_enable : STD_LOGIC := '1';
63      constant C_mr_disable : STD_LOGIC := '0';
64
65      -- mem2reg
66      constant C_memory : STD_LOGIC := '1';
67      constant C_result : STD_LOGIC := '0';
68
69      -- extension
70      constant C_ext_sign : STD_LOGIC := '1';
71      constant C_ext_zero : STD_LOGIC := '0';
72
73      -- positions for instruction/pc
74      -- regfile address1
75      constant POS_REG1_ADDR_TOP : integer := 25;
76      constant POS_REG1_ADDR_BOTTOM : integer := 21;
77
78      -- regfile address2
79      constant POS_REG2_ADDR_TOP : integer := 20;
80      constant POS_REG2_ADDR_BOTTOM : integer := 16;
81
82      -- destination 1 regfile
83      constant POS_REG_DEST_ADDR1_TOP : integer := 20;
84      constant POS_REG_DEST_ADDR1_BOTTOM : integer := 16;
85
86      -- destination 2 regfile
87      constant POS_REG_DEST_ADDR2_TOP : integer := 15;
88      constant POS_REG_DEST_ADDR2_BOTTOM : integer := 11;
89
90      -- operand in instruction
91      constant POS_INST_OP_TOP : integer := 10;
92      constant POS_INST_OP_BOTTOM : integer := 6;
93      constant INST_OP_LENGTH : integer := POS_INST_OP_TOP - POS_INST_OP_BOTTOM + 1;
94
95      -- immediate
96      constant POS_IMMEDIATE_TOP : integer := 15;
97      constant POS_IMMEDIATE_BOTTOM : integer := 0;
98      constant IMMEDIATE_LENGTH : integer := POS_IMMEDIATE_TOP - POS_IMMEDIATE_BOTTOM +
        1;
99
100     -- jump address
101     constant POS_JUMP_ADDRESS_TOP : integer := 25;
102     constant POS_JUMP_ADDRESS_BOTTOM : integer := 0;
103     constant POS_JUMP_ADDRESS_LENGTH : integer := POS_JUMP_ADDRESS_TOP -
        POS_JUMP_ADDRESS_BOTTOM + 1;
104
105     -- pc msb for jump address
106     constant POS_PC_MSB_TOP : integer := 31;
107     constant POS_PC_MSB_BOTTOM : integer := 28;
```

```
108     constant POS_PC_MSB_LENGTH : integer := POS_PC_MSB_TOP - POS_PC_MSB_BOTTOM + 1;
109
110     -- opcode
111     constant POS_OPCODE_TOP : integer := 31;
112     constant POS_OPCODE_BOTTOM : integer := 26;
113     constant OPCODE_LENGTH : integer := POS_OPCODE_TOP - POS_OPCODE_BOTTOM + 1;
114
115     -- funct
116     constant POS_FUNCT_TOP : integer := 5;
117     constant POS_FUNCT_BOTTOM : integer := 0;
118     constant FUNCT_LENGTH : integer := POS_FUNCT_TOP - POS_FUNCT_BOTTOM + 1;
119
120     -- ex_right_shift
121     constant POS_LEFT_SHIFT_TOP : integer := Nbit - 1 - 2;
122     constant POS_LEFT_SHIFT_BOTTOM : integer := 0;
123     constant POS_LEFT_SHIFT_LENGTH : integer := POS_LEFT_SHIFT_TOP -
    POS_LEFT_SHIFT_BOTTOM + 1;
124
125     -- PC offset
126     constant PC_N_BIT_OFFSET : integer := 2;
127     constant PC_OFFSET : unsigned(Nbit - 1 downto 0) :=  (PC_N_BIT_OFFSET => '1',
128                                                          others => '0');
129     constant PC_OFFSET_INT : integer := to_integer(PC_OFFSET);
130
131     -- instruction memory
132     constant length_memory : integer := 7;
133     constant word_length : integer := Nbit / PC_OFFSET_INT;
134 END util;
```

./Code/MIPS_lite_common/wb_stage.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  library work;
6  use work.util.all;
7
8  entity wb_stage is
9      port (clk : in std_logic;
10           rstn : in std_logic;
11
12           rd_mem_data_wb : in std_logic_vector(Nbit - 1 downto 0);
13           alu_res_wb : in std_logic_vector(Nbit - 1 downto 0);
14           mem2reg_wb : in std_logic;
15
16           rf_wr_data_wb : out std_logic_vector(Nbit - 1 downto 0);
17
18           reg_dest_data_wb : in std_logic_vector(bitNreg - 1 downto 0);
19           reg_dest_data_fb : out std_logic_vector(bitNreg - 1 downto 0)
20
21           );
22 end wb_stage;
23
24 architecture behav of wb_stage is
25     begin
26         rf_wr_data_wb <= rd_mem_data_wb when mem2reg_wb = C_memory else
27                          alu_res_wb;
28
29         reg_dest_data_fb <= reg_dest_data_wb;
30
31     end behav;
```

APPENDIX B

MIPS-lite without absolute value

./Code/MIPS_lite_noabs/alu.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library work;
use work.util.all;

entity alu is
    generic (Nbit : integer := 32);
    port (operand1, operand2 : in std_logic_vector(Nbit - 1 downto 0);
          result : out std_logic_vector(Nbit - 1 downto 0);
          zero : out std_logic;
          alu_opcode : in alu_opcode_states);
end alu;

architecture behav of alu is

    begin
        process(operand1, operand2, alu_opcode)
            variable temp, zeros : signed(Nbit - 1 downto 0);
            variable optemp1, optemp2 : signed(Nbit - 1 downto 0);
        begin
            zeros := (others => '0');
            optemp1 := signed(operand1);
            optemp2 := signed(operand2);

            if (alu_opcode = ADDOP) then
                temp := optemp1 + optemp2;
            elsif (alu_opcode = ANDOP) then
                for i in 0 to Nbit - 1 loop
                    temp(i) := optemp1(i) and optemp2(i);
                end loop;
            elsif (alu_opcode = OROP) then
                for i in 0 to Nbit - 1 loop
                    temp(i) := optemp1(i) or optemp2(i);
                end loop;
            elsif (alu_opcode = SHIFTRIGHTOP) then
                temp := shift_right(optemp2, to_integer(optemp1));
            elsif (alu_opcode = XOROP) then
                for i in 0 to Nbit - 1 loop
                    temp(i) := optemp1(i) xor optemp2(i);
```

```
42                 end loop;
43             elsif (alu_opcode = SETLESSTHAN) then
44                 if (optemp1 < optemp2) then
45                     temp(0) := '1';
46                 else
47                     temp(0) := '0';
48                 end if;
49                 temp(Nbit - 1 downto 1) := (others => '0');
50             elsif (alu_opcode = SUBOP) then
51                 temp := optemp1 - optemp2;
52             elsif (alu_opcode = SHIFTLEFT16) then
53                 temp := shift_left(optemp2, 16);
54 
55             else
56                 temp := (others => '0');
57             end if;
58 
59             result <= std_logic_vector(temp);
60             if (temp = zeros) then
61                 zero <= '1';
62             else
63                 zero <= '0';
64             end if;
65         end process;
66     end behav;
```

./Code/MIPS_lite_noabs/control_unit.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4 
5  library work;
6  use work.util.all;
7 
8  entity control_unit is
9  port (  opcode : in std_logic_vector (OPCODE_LENGTH - 1 downto 0);
10         func : in std_logic_vector (FUNCT_LENGTH - 1 downto 0);
11         reg_dest : out std_logic;
12         reg_write : out std_logic;
13         ALU_src1 : out std_logic;
14         ALU_src2 : out std_logic;
15         extension : out std_logic;
16         branch : out std_logic;
17         jump : out std_logic;
18         mem_write : out std_logic;
19         mem_read : out std_logic;
20         mem2reg : out std_logic;
21         ALU_operation: out alu_opcode_states);
22 end entity control_unit;
23 
24 architecture behavior of control_unit is
25 
26 signal opcode_long, func_long : std_logic_vector (OPCODE_LENGTH + 2 - 1 downto 0);
27 
28 begin
29 
30 opcode_long <= "00" & opcode;
31 func_long <= "00" & func;
32 
33 cu_process : process (opcode_long, func_long)
34 begin
```

```
35      case opcode_long is
36          when x"00" =>
37                      case func_long is
38                          -- nop
39                          when x"00" =>   reg_dest <= C_i_15_11;
40                                          reg_write <= C_reg_disable;
41                                          ALU_src1 <= C_registers1;
42                                          ALU_src2 <= C_registers2;
43                                          extension <= C_ext_zero;
44                                          branch <= C_branch_no;
45                                          jump <= C_jump_no;
46                                          mem_write <= C_mw_disable;
47                                          mem_read <= C_mr_disable;
48                                          mem2reg <= C_result;
49                                          ALU_operation <= NOP;
50                          -- add
51                          when x"20" =>   reg_dest <= C_i_15_11;
52                                          reg_write <= C_reg_enable;
53                                          ALU_src1 <= C_registers1;
54                                          ALU_src2 <= C_registers2;
55                                          extension <= C_ext_zero;
56                                          branch <= C_branch_no;
57                                          jump <= C_jump_no;
58                                          mem_write <= C_mw_disable;
59                                          mem_read <= C_mr_disable;
60                                          mem2reg <= C_result;
61                                          ALU_operation <= ADDOP;
62                          -- slt
63                          when x"2a" =>   reg_dest <= C_i_15_11;
64                                          reg_write <= C_reg_enable;
65                                          ALU_src1 <= C_registers1;
66                                          ALU_src2 <= C_registers2;
67                                          extension <= C_ext_zero;
68                                          branch <= C_branch_no;
69                                          jump <= C_jump_no;
70                                          mem_write <= C_mw_disable;
71                                          mem_read <= C_mr_disable;
72                                          mem2reg <= C_result;
73                                          ALU_operation <= SETLESSTHAN;
74                          -- xor
75                          when x"26" =>   reg_dest <= C_i_15_11;
76                                          reg_write <= C_reg_enable;
77                                          ALU_src1 <= C_registers1;
78                                          ALU_src2 <= C_registers2;
79                                          extension <= C_ext_zero;
80                                          branch <= C_branch_no;
81                                          jump <= C_jump_no;
82                                          mem_write <= C_mw_disable;
83                                          mem_read <= C_mr_disable;
84                                          mem2reg <= C_result;
85                                          ALU_operation <= XOROP;
86                          -- sra
87                          when x"03" =>   reg_dest <= C_i_15_11;
88                                          reg_write <= C_reg_enable;
89                                          ALU_src1 <= C_i_10_6;
90                                          ALU_src2 <= C_registers2;
91                                          extension <= C_ext_zero;
92                                          branch <= C_branch_no;
93                                          jump <= C_jump_no;
94                                          mem_write <= C_mw_disable;
95                                          mem_read <= C_mr_disable;
96                                          mem2reg <= C_result;
```

```
97                                      ALU_operation <= SHIFTRIGHTOP;
98
99                      when others =>  reg_dest <= C_i_15_11;
100                                     reg_write <= C_reg_disable;
101                                     ALU_src1 <= C_registers1;
102                                     ALU_src2 <= C_registers2;
103                                     extension <= C_ext_zero;
104                                     branch <= C_branch_no;
105                                     jump <= C_jump_no;
106                                     mem_write <= C_mw_disable;
107                                     mem_read <= C_mr_disable;
108                                     mem2reg <= C_result;
109                                     ALU_operation <= NOP;
110                 end case;
111         -- addi
112         when x"08" =>   reg_dest <= C_i_20_16;
113                         reg_write <= C_reg_enable;
114                         ALU_src1 <= C_registers1;
115                         ALU_src2 <= C_sign_extension;
116                         extension <= C_ext_sign;
117                         branch <= C_branch_no;
118                         jump <= C_jump_no;
119                         mem_write <= C_mw_disable;
120                         mem_read <= C_mr_disable;
121                         mem2reg <= C_result;
122                         ALU_operation <= ADDOP;
123         -- andi
124         when x"0c" =>   reg_dest <= C_i_20_16;
125                         reg_write <= C_reg_enable;
126                         ALU_src1 <= C_registers1;
127                         ALU_src2 <= C_sign_extension;
128                         extension <= C_ext_zero;
129                         branch <= C_branch_no;
130                         jump <= C_jump_no;
131                         mem_write <= C_mw_disable;
132                         mem_read <= C_mr_disable;
133                         mem2reg <= C_result;
134                         ALU_operation <= ANDOP;
135         -- beq
136         when x"04" =>   reg_dest <= C_i_15_11;
137                         reg_write <= C_reg_disable;
138                         ALU_src1 <= C_registers1;
139                         ALU_src2 <= C_registers2;
140                         extension <= C_ext_sign;
141                         branch <= C_branch_yes;
142                         jump <= C_jump_no;
143                         mem_write <= C_mw_disable;
144                         mem_read <= C_mr_disable;
145                         mem2reg <= C_result;
146                         ALU_operation <= SUBOP;
147         -- j
148         when x"02" =>   reg_dest <= C_i_15_11;
149                         reg_write <= C_reg_disable;
150                         ALU_src1 <= C_registers1;
151                         ALU_src2 <= C_registers2;
152                         extension <= C_ext_zero;
153                         branch <= C_branch_no;
154                         jump <= C_jump_yes;
155                         mem_write <= C_mw_disable;
156                         mem_read <= C_mr_disable;
157                         mem2reg <= C_result;
158                         ALU_operation <= NOP;
```

```
159         -- lui
160         when x"0f" =>   reg_dest <= C_i_20_16;
161                         reg_write <= C_reg_enable;
162                         ALU_src1 <= C_registers1;
163                         ALU_src2 <= C_sign_extension;
164                         extension <= C_ext_zero;
165                         branch <= C_branch_no;
166                         jump <= C_jump_no;
167                         mem_write <= C_mw_disable;
168                         mem_read <= C_mr_disable;
169                         mem2reg <= C_result;
170                         ALU_operation <= SHIFTLEFT16;
171         -- lw
172         when x"23" =>   reg_dest <= C_i_20_16;
173                         reg_write <= C_reg_enable;
174                         ALU_src1 <= C_registers1;
175                         ALU_src2 <= C_sign_extension;
176                         extension <= C_ext_sign;
177                         branch <= C_branch_no;
178                         jump <= C_jump_no;
179                         mem_write <= C_mw_disable;
180                         mem_read <= C_mr_enable;
181                         mem2reg <= C_memory;
182                         ALU_operation <= ADDOP;
183         -- ori
184         when x"0d" =>   reg_dest <= C_i_20_16;
185                         reg_write <= C_reg_enable;
186                         ALU_src1 <= C_registers1;
187                         ALU_src2 <= C_sign_extension;
188                         extension <= C_ext_zero;
189                         branch <= C_branch_no;
190                         jump <= C_jump_no;
191                         mem_write <= C_mw_disable;
192                         mem_read <= C_mr_disable;
193                         mem2reg <= C_result;
194                         ALU_operation <= OROP;
195         -- sw
196         when x"2b" =>   reg_dest <= C_i_20_16;
197                         reg_write <= C_reg_disable;
198                         ALU_src1 <= C_registers1;
199                         ALU_src2 <= C_sign_extension;
200                         extension <= C_ext_sign;
201                         branch <= C_branch_no;
202                         jump <= C_jump_no;
203                         mem_write <= C_mw_enable;
204                         mem_read <= C_mr_disable;
205                         mem2reg <= C_result;
206                         ALU_operation <= ADDOP;
207
208
209
210         when others =>  reg_dest <= C_i_15_11;
211                         reg_write <= C_reg_disable;
212                         ALU_src1 <= C_registers1;
213                         ALU_src2 <= C_registers2;
214                         extension <= C_ext_zero;
215                         branch <= C_branch_no;
216                         jump <= C_jump_no;
217                         mem_write <= C_mw_disable;
218                         mem_read <= C_mr_disable;
219                         mem2reg <= C_result;
220                         ALU_operation <= NOP;
```

```
221     end case;
222 end process cu_process;
223 
224 end architecture behavior;
```

APPENDIX C

MIPS-lite with absolute value

./Code/MIPS_lite_abs/alu.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library work;
use work.util.all;

entity alu is
    generic (Nbit : integer := 32);
    port (operand1, operand2 : in std_logic_vector(Nbit - 1 downto 0);
          result : out std_logic_vector(Nbit - 1 downto 0);
          zero : out std_logic;
          alu_opcode : in alu_opcode_states);
end alu;

architecture behav of alu is

    begin
        process(operand1, operand2, alu_opcode)
            variable temp, zeros : signed(Nbit - 1 downto 0);
            variable optemp1, optemp2 : signed(Nbit - 1 downto 0);
        begin
            zeros := (others => '0');
            optemp1 := signed(operand1);
            optemp2 := signed(operand2);

            if (alu_opcode = ADDOP) then
                temp := optemp1 + optemp2;
            elsif (alu_opcode = ANDOP) then
                for i in 0 to Nbit - 1 loop
                    temp(i) := optemp1(i) and optemp2(i);
                end loop;
            elsif (alu_opcode = OROP) then
                for i in 0 to Nbit - 1 loop
                    temp(i) := optemp1(i) or optemp2(i);
                end loop;
            elsif (alu_opcode = SHIFTRIGHTOP) then
                temp := shift_right(optemp2, to_integer(optemp1));
            elsif (alu_opcode = XOROP) then
                for i in 0 to Nbit - 1 loop
                    temp(i) := optemp1(i) xor optemp2(i);
```

```
42                 end loop;
43             elsif (alu_opcode = SETLESSTHAN) then
44                 if (optemp1 < optemp2) then
45                     temp(0) := '1';
46                 else
47                     temp(0) := '0';
48                 end if;
49                 temp(Nbit - 1 downto 1) := (others => '0');
50             elsif (alu_opcode = SUBOP) then
51                 temp := optemp1 - optemp2;
52             elsif (alu_opcode = SHIFTLEFT16) then
53                 temp := shift_left(optemp2, 16);
54             elsif (alu_opcode = ABSOP) then
55                 if (optemp1 < 0) then
56                     temp := -optemp1;
57                 else
58                     temp := optemp1;
59                 end if;
60             else
61                 temp := (others => '0');
62             end if;
63
64             result <= std_logic_vector(temp);
65             if (temp = zeros) then
66                 zero <= '1';
67             else
68                 zero <= '0';
69             end if;
70         end process;
71     end behav;
```

./Code/MIPS_lite_abs/control_unit.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  library work;
6  use work.util.all;
7
8  entity control_unit is
9  port (  opcode : in std_logic_vector (OPCODE_LENGTH - 1 downto 0);
10         func : in std_logic_vector (FUNCT_LENGTH - 1 downto 0);
11         reg_dest : out std_logic;
12         reg_write : out std_logic;
13         ALU_src1 : out std_logic;
14         ALU_src2 : out std_logic;
15         extension : out std_logic;
16         branch : out std_logic;
17         jump : out std_logic;
18         mem_write : out std_logic;
19         mem_read : out std_logic;
20         mem2reg : out std_logic;
21         ALU_operation: out alu_opcode_states);
22 end entity control_unit;
23
24 architecture behavior of control_unit is
25
26 signal opcode_long, func_long : std_logic_vector (OPCODE_LENGTH + 2 - 1 downto 0);
27
28 begin
29
```

```
30 opcode_long <= "00" & opcode;
31 func_long <= "00" & func;
32
33 cu_process : process (opcode_long, func_long)
34 begin
35     case opcode_long is
36         when x"00" =>
37                     case func_long is
38                         -- nop
39                         when x"00" =>   reg_dest <= C_i_15_11;
40                                         reg_write <= C_reg_disable;
41                                         ALU_src1 <= C_registers1;
42                                         ALU_src2 <= C_registers2;
43                                         extension <= C_ext_zero;
44                                         branch <= C_branch_no;
45                                         jump <= C_jump_no;
46                                         mem_write <= C_mw_disable;
47                                         mem_read <= C_mr_disable;
48                                         mem2reg <= C_result;
49                                         ALU_operation <= NOP;
50                         -- add
51                         when x"20" =>   reg_dest <= C_i_15_11;
52                                         reg_write <= C_reg_enable;
53                                         ALU_src1 <= C_registers1;
54                                         ALU_src2 <= C_registers2;
55                                         extension <= C_ext_zero;
56                                         branch <= C_branch_no;
57                                         jump <= C_jump_no;
58                                         mem_write <= C_mw_disable;
59                                         mem_read <= C_mr_disable;
60                                         mem2reg <= C_result;
61                                         ALU_operation <= ADDOP;
62                         -- slt
63                         when x"2a" =>   reg_dest <= C_i_15_11;
64                                         reg_write <= C_reg_enable;
65                                         ALU_src1 <= C_registers1;
66                                         ALU_src2 <= C_registers2;
67                                         extension <= C_ext_zero;
68                                         branch <= C_branch_no;
69                                         jump <= C_jump_no;
70                                         mem_write <= C_mw_disable;
71                                         mem_read <= C_mr_disable;
72                                         mem2reg <= C_result;
73                                         ALU_operation <= SETLESSTHAN;
74                         -- xor
75                         when x"26" =>   reg_dest <= C_i_15_11;
76                                         reg_write <= C_reg_enable;
77                                         ALU_src1 <= C_registers1;
78                                         ALU_src2 <= C_registers2;
79                                         extension <= C_ext_zero;
80                                         branch <= C_branch_no;
81                                         jump <= C_jump_no;
82                                         mem_write <= C_mw_disable;
83                                         mem_read <= C_mr_disable;
84                                         mem2reg <= C_result;
85                                         ALU_operation <= XOROP;
86                         -- sra
87                         when x"03" =>   reg_dest <= C_i_15_11;
88                                         reg_write <= C_reg_enable;
89                                         ALU_src1 <= C_i_10_6;
90                                         ALU_src2 <= C_registers2;
91                                         extension <= C_ext_zero;
```

```
92                                      branch <= C_branch_no;
93                                      jump <= C_jump_no;
94                                      mem_write <= C_mw_disable;
95                                      mem_read <= C_mr_disable;
96                                      mem2reg <= C_result;
97                                      ALU_operation <= SHIFTRIGHTOP;
98
99                          when others =>  reg_dest <= C_i_15_11;
100                                     reg_write <= C_reg_disable;
101                                     ALU_src1 <= C_registers1;
102                                     ALU_src2 <= C_registers2;
103                                     extension <= C_ext_zero;
104                                     branch <= C_branch_no;
105                                     jump <= C_jump_no;
106                                     mem_write <= C_mw_disable;
107                                     mem_read <= C_mr_disable;
108                                     mem2reg <= C_result;
109                                     ALU_operation <= NOP;
110                 end case;
111         -- addi
112         when x"08" =>   reg_dest <= C_i_20_16;
113                         reg_write <= C_reg_enable;
114                         ALU_src1 <= C_registers1;
115                         ALU_src2 <= C_sign_extension;
116                         extension <= C_ext_sign;
117                         branch <= C_branch_no;
118                         jump <= C_jump_no;
119                         mem_write <= C_mw_disable;
120                         mem_read <= C_mr_disable;
121                         mem2reg <= C_result;
122                         ALU_operation <= ADDOP;
123         -- andi
124         when x"0c" =>   reg_dest <= C_i_20_16;
125                         reg_write <= C_reg_enable;
126                         ALU_src1 <= C_registers1;
127                         ALU_src2 <= C_sign_extension;
128                         extension <= C_ext_zero;
129                         branch <= C_branch_no;
130                         jump <= C_jump_no;
131                         mem_write <= C_mw_disable;
132                         mem_read <= C_mr_disable;
133                         mem2reg <= C_result;
134                         ALU_operation <= ANDOP;
135         -- beq
136         when x"04" =>   reg_dest <= C_i_15_11;
137                         reg_write <= C_reg_disable;
138                         ALU_src1 <= C_registers1;
139                         ALU_src2 <= C_registers2;
140                         extension <= C_ext_sign;
141                         branch <= C_branch_yes;
142                         jump <= C_jump_no;
143                         mem_write <= C_mw_disable;
144                         mem_read <= C_mr_disable;
145                         mem2reg <= C_result;
146                         ALU_operation <= SUBOP;
147         -- j
148         when x"02" =>   reg_dest <= C_i_15_11;
149                         reg_write <= C_reg_disable;
150                         ALU_src1 <= C_registers1;
151                         ALU_src2 <= C_registers2;
152                         extension <= C_ext_zero;
153                         branch <= C_branch_no;
```

```
154                         jump <= C_jump_yes;
155                         mem_write <= C_mw_disable;
156                         mem_read <= C_mr_disable;
157                         mem2reg <= C_result;
158                         ALU_operation <= NOP;
159         -- lui
160         when x"0f" =>   reg_dest <= C_i_20_16;
161                         reg_write <= C_reg_enable;
162                         ALU_src1 <= C_registers1;
163                         ALU_src2 <= C_sign_extension;
164                         extension <= C_ext_zero;
165                         branch <= C_branch_no;
166                         jump <= C_jump_no;
167                         mem_write <= C_mw_disable;
168                         mem_read <= C_mr_disable;
169                         mem2reg <= C_result;
170                         ALU_operation <= SHIFTLEFT16;
171         -- lw
172         when x"23" =>   reg_dest <= C_i_20_16;
173                         reg_write <= C_reg_enable;
174                         ALU_src1 <= C_registers1;
175                         ALU_src2 <= C_sign_extension;
176                         extension <= C_ext_sign;
177                         branch <= C_branch_no;
178                         jump <= C_jump_no;
179                         mem_write <= C_mw_disable;
180                         mem_read <= C_mr_enable;
181                         mem2reg <= C_memory;
182                         ALU_operation <= ADDOP;
183         -- ori
184         when x"0d" =>   reg_dest <= C_i_20_16;
185                         reg_write <= C_reg_enable;
186                         ALU_src1 <= C_registers1;
187                         ALU_src2 <= C_sign_extension;
188                         extension <= C_ext_zero;
189                         branch <= C_branch_no;
190                         jump <= C_jump_no;
191                         mem_write <= C_mw_disable;
192                         mem_read <= C_mr_disable;
193                         mem2reg <= C_result;
194                         ALU_operation <= OROP;
195         -- sw
196         when x"2b" =>   reg_dest <= C_i_20_16;
197                         reg_write <= C_reg_disable;
198                         ALU_src1 <= C_registers1;
199                         ALU_src2 <= C_sign_extension;
200                         extension <= C_ext_sign;
201                         branch <= C_branch_no;
202                         jump <= C_jump_no;
203                         mem_write <= C_mw_enable;
204                         mem_read <= C_mr_disable;
205                         mem2reg <= C_result;
206                         ALU_operation <= ADDOP;
207
208                         -- chosen opcode for absolute value
209         when x"14" =>   reg_dest <= C_i_20_16;
210                         reg_write <= C_reg_enable;
211                         ALU_src1 <= C_registers1;
212                         ALU_src2 <= C_registers2;
213                         extension <= C_ext_zero;
214                         branch <= C_branch_no;
215                         jump <= C_jump_no;
```

```
216                         mem_write <= C_mw_disable;
217                         mem_read <= C_mr_disable;
218                         mem2reg <= C_result;
219                         ALU_operation <= ABSOP;
220
221         when others =>  reg_dest <= C_i_15_11;
222                         reg_write <= C_reg_disable;
223                         ALU_src1 <= C_registers1;
224                         ALU_src2 <= C_registers2;
225                         extension <= C_ext_zero;
226                         branch <= C_branch_no;
227                         jump <= C_jump_no;
228                         mem_write <= C_mw_disable;
229                         mem_read <= C_mr_disable;
230                         mem2reg <= C_result;
231                         ALU_operation <= NOP;
232     end case;
233 end process cu_process;
234
235 end architecture behavior;
```