Previ 3. Amb les instruccions creades a classe, escriviu un codi que pugui esperar la quantitat de temps demanada per la partitura i alterni diferents notes creant una melodia senzilla. Adjunteu tot el codi del mini AVR.

```
library ieee;

use ieee.std_logic_1164.all;

use ieee.numeric_std.all;

entity AVR_1 is

port (

clk : in std_logic ;

reset : in std_logic ;

-- Let ' s have some registers as outputs :

r16 : out std_logic_vector (7 downto 0);

r17 : out std_logic_vector (7 downto 0);

r18 : out std_logic_vector (7 downto 0);

r19 : out std_logic_vector (7 downto 0)

);

end AVR_1;

architecture arch of AVR_1 is

 type register_bank is

 array (15 downto 0) of std_logic_vector (7 downto 0);

 signal regs : register_bank ; -- Registers r16 .. r31

 type status_reg is -- Reg.status types

 record

 Z : std_logic ;

 C : std_logic ;

 end record ;

 signal pr_SR , nx_SR : status_reg ;

type out_mux_type is (mux_alu , mux_lit); -- CONTROL

signal out_mux : out_mux_type ;

signal d_reg,r_reg : std_logic_vector(3 downto 0):=(others=>'0');

signal reg_we : std_logic;

signal alu_op : std_logic_vector (2 downto 0):=(others=>'0');

signal k: std_logic_vector(7 downto 0):=(others=>'0');

--ALU signals

signal alu_in_a,alu_in_b,alu_out: std_logic_vector(7 downto 0):=(others=>'0');

signal add_temp: std_logic_vector(9 downto 0);

signal update_Z: std_logic :='0';

--ROM signals

signal nx_pc : std_logic_vector(7 downto 0):=(others=>'0');

signal pr_pc : std_logic_vector(7 downto 0):=(others=>'0');

signal pr_op : std_logic_vector(15 downto 0):=(others=>'0');

signal k_jmp : std_logic_vector(7 downto 0):=(others=>'0');

signal nx_reg : std_logic_vector(7 downto 0):=(others=>'0');

-- CONSTANTS

constant NOP : std_logic_vector (3 downto 0) := "0000";

constant ADC : std_logic_vector (3 downto 0) := "0001";

constant ALU_B : std_logic_vector (3 downto 0) := "0010"; -- New MOV
```

```
constant RJMP : std_logic_vector (3 downto 0) := "1100";

constant LDI : std_logic_vector (3 downto 0) := "1110";

constant BRANCH: std_logic_vector (3 downto 0) := "1111";

constant ALU_B_AND : std_logic_vector (1 downto 0) := "00";

constant ALU_B_EOR : std_logic_vector (1 downto 0) := "01";

constant ALU_B_OR : std_logic_vector (1 downto 0) := "10";

constant ALU_B_MOV : std_logic_vector (1 downto 0) := "11";

------

constant ALU_AND: std_logic_vector (2 downto 0) := "000";

constant ALU_ADC : std_logic_vector (2 downto 0) :=  "001";

constant ALU_MOV : std_logic_vector (2 downto 0) :=  "010";

constant ALU_OR : std_logic_vector (2 downto 0) :=  "011";

constant ALU_EOR : std_logic_vector (2 downto 0) :=  "100";

-- constant AND : std_logic_vector (3 downto 0) := " 0010 ";

--constant MOV : std_logic_vector (3 downto 0) := "0010";

-- constant OR : std_logic_vector (3 downto 0) := " 0010 ";

-- constant EOR : std_logic_vector (3 downto 0) := " 0010 ";

-- ALU_B replaced the old MOV instruction

--Senyals de debugging

  signal debug_carry, debug_zero: std_logic;

------------

begin

COUNTER:  process (clk , reset) -- Synchronous elements

  begin

  if reset ='1' then -- Initialize Processor

  pr_pc <= (others =>'0');

  pr_SR.C <='0';

  pr_SR.Z <='0';

  elsif (rising_edge (clk)) then

  pr_pc <= nx_pc ;

  pr_SR <= nx_SR ;

  end if;

end process COUNTER ;

  nx_pc <= std_logic_vector(unsigned(pr_pc) + 1);


  NEXT_PC : process (pr_pc , k_jmp)

  variable tmp_pc : std_logic_vector (8 downto 0);

  begin

  tmp_pc := std_logic_vector (signed(pr_pc &'1') +

  signed(k_jmp &'1'));

  nx_pc <= tmp_pc (8 downto 1);

  end process ;

-- REGISTER

RegW : process (clk) -- Register write

begin
```

```
if rising_edge (clk) then
 if reg_we = '1' then -- write
  regs (to_integer (unsigned (d_reg))) <= nx_reg ;
 end if;
end if;
end process RegW ;
-- Register read : ALU inputs
alu_in_a <= regs (to_integer(unsigned(d_reg)));
alu_in_b <= regs (to_integer(unsigned(r_reg)));

CONTROL : process (pr_op , pr_pc , pr_SR)
begin
out_mux <= mux_alu ;
r_reg <= pr_op (3 downto 0); -- Defaults
d_reg <= pr_op (7 downto 4);
reg_we <='0';
k_jmp <=(others =>'0');
case pr_op(15 downto 12) is
 when LDI => -- - - - - - - - - - - - - - - - - - - - - - - - LDI Instruction
  d_reg <= pr_op (7 downto 4);
  out_mux <= mux_lit ;
  k <= pr_op (11 downto 8) & pr_op (3 downto 0);
  reg_we<='1';
  -- -- when MOV => -- - - - - - - - - - - - - - - - - - - - - - - - - - - - MOV Instruction
  -- -- r_reg <= pr_op (3 downto 0); -- Source register
  -- -- d_reg <= pr_op (7 downto 4); -- Destination register
  -- -- out_mux <= mux_alu ;
  -- -- reg_we <='1'; -- Enable register write
  -- ALU_op <= ALU_MOV ;
 when NOP => null;
 when ADC => -- - - - - - - - - - - - - - - - - - - - - - - - - - - - ADC Instruction
  r_reg <= pr_op (3 downto 0);
  d_reg <= pr_op (7 downto 4);
  out_mux <= mux_alu ;
  reg_we <='1'; -- Enable register write
  ALU_op <= ALU_ADC ;
 when BRANCH => -- - - - - - - - - - - - - - - - - BRANCH Instruction
  if pr_op (10) ='0' then -- BREQ Instruction
   if pr_SR .Z ='1' then
   k_jmp (6 downto 0) <= pr_op (9 downto 3);
   k_jmp (7) <= pr_op (9);
   end if;
  else -- BRNE (Complete this instruction)
   if pr_SR.Z ='0' then
   k_jmp (6 downto 0) <= pr_op (9 downto 3);
   k_jmp (7) <= pr_op (9);
   end if;
  end if;
 when RJMP => -- RJMP (Complete this instruction)
  if pr_op(3)='1' then
   k_jmp<= "1111" &  pr_op(3 downto 0);
```

```
   else
    k_jmp<= "0000" & pr_op(3 downto 0);
   end if;
  when ALU_B => -- - - - - - - - - - - MOV , AND , EOR , OR Instructions
   reg_we <='1'; -- Enable register write
   case pr_op(11 downto 10) is
    when ALU_B_MOV =>
    ALU_op <= ALU_MOV ;
    when ALU_B_EOR =>
    ALU_op <= ALU_EOR ;
    when ALU_B_AND =>
     ALU_op <= ALU_AND ;
    when ALU_B_OR =>
     ALU_op <= ALU_OR ;

    when others => null ;
   end case ;
   when others=> null;
end case;
end process CONTROL ;
ALU : process (alu_op , alu_in_a , alu_in_b , pr_SR , add_temp)
begin
  nx_SR .C <= pr_SR .C; -- by default , preserve
  update_Z <='1'; -- Most operations update Z
  case ALU_op is
  when ALU_MOV => -- - - - - - - - - - - - - - - - MOV : in_b --> out
  alu_out <= alu_in_b ;
  update_Z <='0';
  when ALU_AND => -- - - - - - - - - - - - - - - - AND
  alu_out <= alu_in_a and alu_in_b ;
  when ALU_OR => -- - - - - - - - - - - - - - - - OR
  alu_out <= alu_in_a or alu_in_b ;
  when ALU_EOR => -- - - - - - - - - - - - - - - - EOR / XOR
  alu_out <= alu_in_a xor alu_in_b ;
when ALU_ADC => -- - - - - - - - - - - - - - - - - - - ADC : Carry in / out
add_temp <= std_logic_vector(unsigned('0' & alu_in_a & '1') +unsigned('0' & alu_in_b & pr_SR.C)); -- Carry In
alu_out <= add_temp (8 downto 1);
nx_SR .C <= add_temp (9); -- Update Carry Flag
if add_temp (8 downto 1) = x"00" then -- Update Zero Flag
nx_SR .Z <='1';
else
nx_SR .Z <='0';
end if;
when others => -- - - - - - - - - - - - - - - - - - - - - - - - - - Should not happen
alu_out <= (others =>'-'); -- Don' t care
end case ;
end process ;

UPD_z : process (update_Z , alu_out , pr_SR)
begin
```

```
if update_Z ='1' then -- - - - - - - - - Update Zero Flag
if alu_out = x"00" then
nx_SR .Z <='1';
else
nx_SR .Z <='0';
end if;
else -- - - - - - - - - Keep old value
nx_SR .Z <= pr_SR .Z;
end if;
end process ;
MUX : process (out_mux , alu_out , k)
begin
case out_mux is
when mux_alu => nx_reg <= alu_out ;
when mux_lit => nx_reg <= k;
when others => nx_reg <= (others =>'-');
end case ;
end process ;
-----------ROM------------
ROM : process (pr_pc) -- Program ROM
begin
case pr_pc is
when x"00" => pr_op <= LDI & "0100" & "0000" & "0001" ; -- LDI ,r16 x41
when x"01" => pr_op <= LDI & "1000" & "0001" & "0011" ; -- LDI r17, x83
when x"02" => pr_op <= ADC & "1111" & "0000" & "0000" ; -- ADC r16,r16
when x"03" => pr_op <= BRANCH & "00" & "00000001" & "01"; -- BREQ
when x"04" => pr_op <= RJMP & "----" & "1111" & "1101";
when x"05" => pr_op <= RJMP & "----" & "1111" & "1111";
when others => pr_op <= (others =>'-');
end case ;
end process ;
r16 <= regs(0);
r17 <= regs(1);
debug_carry <= pr_SR.C;
debug_zero <= pr_SR.Z;
end arch;
```

Previ 4. Feu un programa de test que comprovi que es va executant la melodia i que sou capaços d'esperar el temps previst entre notes. Demostreu-ho amb un testbench i una simulació adequada.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity AVR_1_tb is
end AVR_1_tb;
architecture arch of AVR_1_tb is
  component my_AVR_1
  port (
  clk : in std_logic ;
```

```
 reset : in std_logic ;
 -- Let ' s have some registers as outputs :
 r16 : out std_logic_vector ( 7 downto 0);
 r17 : out std_logic_vector ( 7 downto 0);
 r18 : out std_logic_vector ( 7 downto 0);
 r19 : out std_logic_vector ( 7 downto 0)
 );
end component;
for dut1: my_AVR_1 use entity work.AVR_1;
signal S16,S17,S18,S19: std_logic_vector(7 downto 0);
signal sreset: std_logic;
signal sclk: std_logic;
begin
dut1: my_AVR_1 port map(
clk=> sclk,
reset=>sreset,
r16=>S16,
r17=>S17,
r18=>S18,
r19=>S19);
clock_process: process
 begin          --the clock process
  sclk <= '0';
  wait for 500 ns;
  for i in 1 to 4000000 loop
   sclk <= not sclk;
    wait for 500 ns;
  end loop;
  wait;
 end process clock_process;
sreset<='1','0' after 10 ms ;
end arch;
```

Previ 5. Dissenyeu un circuit analògic capaç de sumar els quatre senyals generats per cadascun dels sintetitzadors.

No he sigut capaç de desargollar aquest tasca perquè no he entès el enunciat.