NANO PROCESSOR

GROUP NAME - **KYOJIN**

GROUP MEMBERS

- Tharuka E.A.A – 190623V
- Rubasinghe R.K.R.U – 190530H
- Perera P.D.N.D – 190458T

CONTRIBUTION

THARUKA E A

INSTRUCTION DECODER

PROGRAM ROM

LUT (LOOK-UP TABLE)

RUBASINGHE R K R U

REGISTER BANK

ADD/SUBTRACT UNIT

PERERA P D N D

PROGRAM COUNTER

3-bit register

3-bit adder

2 way 3-bit multiplexer

INTRODUCTION 4
INSTRUCTION DECODER 4
VHDL CODE 4
RTL SCHEMATICS 6
BEHAVIORAL SIMULATION CODE 6
TIMING DIAGRAM 8
PROGRAM ROM 8
ASSEMBLY PROGRAM AND MACHINE CODE REPRESENTATION 8
VHDL CODE 9
RTL SCHEMATICS 10
BEHAVIORAL SIMULATION CODE 10
TIMING DIAGRAM 11
LOOKUP TABLE 11
VHDL CODE 12
RTL SCHEMATICS 13
REGISTER BANK 13
VHDL CODE 13
RTL SCHEMATICS 16
BEHAVIORAL SIMULATION CODE 17
TIMING DIAGRAM 20
4–BIT ADD/SUBTRACT UNIT 20
VHDL CODE 20
RTL SCHEMATICS 23
BEHAVIORAL SIMULATION CODE 23
TIMING DIAGRAM 25
4-BIT RIPPLE CARRY ADDER 25
VHDL CODE 25
RTL SCHEMATIC 27
PROGRAM COUNTER 28
VHDL CODE 28
RTL SCHEMATICS 31
BEHAVIORAL SIMULATION CODE 31
TIMING DIAGRAM 33
PC REGISTER 33
VHDL CODE 33
RTL SCHEMATICS 34
BEHAVIORAL SIMULATION CODE 34

TIMING DIAGRAM 36
3–BIT ADDER 36
VHDL CODE 36
RTL SCHEMATICS 38
BEHAVIORAL SIMULATION CODE 38
TIMING DIAGRAM 40
K–WAY B–BIT MUX 40
8–WAY 4–BIT MUX 41
VHDL CODE *41*
RTL SCHEMATICS *42*
BEHAVIORAL SIMULATION CODE *42*
TIMING DIAGRAM *44*
2–WAY 4–BIT MUX 44
VHDL CODE *44*
RTL SCHEMATICS *45*
BEHAVIORAL SIMULATION CODE *45*
TIMING DIAGRAM *46*
2–WAY 3–BIT MUX 47
VHDL CODE *47*
RTL SCHEMATICS *47*
BEHAVIORAL SIMULATION CODE *48*
TIMING DIAGRAM *49*
REGISTER 49
VHDL CODE 49
RTL SCHEMATICS 50
BEHAVIORAL SIMULATION CODE 50
TIMING DIAGRAM 52
SLOW CLOCK 52
VHDL CODE 52
NANO PROCESSOR 53
VHDL CODE 53
RTL SCHEMATICS 59
BEHAVIORAL SIMULATION CODE 59
TIMING DIAGRAM 61
XDC FILE 61
CONCLUSION 63

INTRODUCTION

In this final lab, we were asked to design a simple microprocessor capable of executing a simple set of instructions.

This consists of a 4-bit Add/Subtract unit that is capable of adding and subtracting numbers using 2's complement method and also a 3-bit Program Counter (PC) that is designed using a 3-bit adder, a 3-bit register, and a 2 way 3-bit multiplexer.

An Instruction Decoder is created to activate necessary components based on the instructions provided by the program ROM which stores the Assembly program with the instructions we wish to execute. Buses are used to combine the components to reduce the complexity of the code.

Register bank is created using registers and a 3-8 decoder implemented in previous labs.

This nano processor is capable of executing four operations such as moving a value directly to a register, adding two values, getting the 2's complement of a value, and jumping to a specific instruction.

Therefore, to subtract two values, it is needed to take the negation of one value and add them.

INSTRUCTION DECODER

The instruction decoder accepts the instruction provided by the program ROM in each clock cycle.

It identifies the necessary components which are needed to be activated and signals which are needed to be passed to each activated component.

It contains a 2–4 decoder which is used to recognize the instruction opcode. Then it will distribute the provided register addresses and immediate values accordingly.

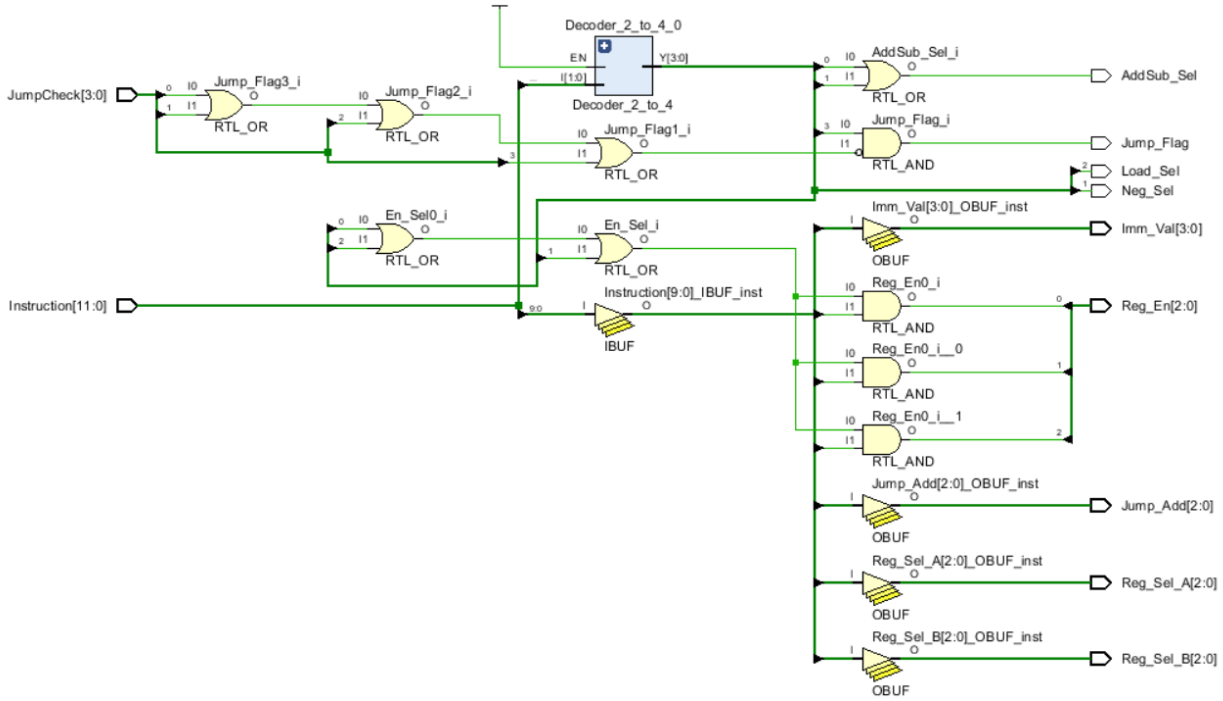
VHDL CODE

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


entity Ins_Decoder is
   Port ( Instruction     : in STD_LOGIC_VECTOR (11 downto 0);
          JumpCheck       : in STD_LOGIC_VECTOR (3 downto 0);
          Reg_En          : out STD_LOGIC_VECTOR (2 downto 0);
          Load_Sel        : out STD_LOGIC;
```

```
        Imm_Val         : out STD_LOGIC_VECTOR (3 downto 0);
        Reg_Sel_A       : out STD_LOGIC_VECTOR (2 downto 0);
        Reg_Sel_B       : out STD_LOGIC_VECTOR (2 downto 0);
        AddSub_Sel      : out STD_LOGIC;
        Jump_Flag       : out STD_LOGIC;
        Jump_Add        : out STD_LOGIC_VECTOR (2 downto 0);
        Neg_Sel         : out STD_LOGIC);
end Ins_Decoder;

architecture Behavioral of Ins_Decoder is

component Decoder_2_to_4 is
   Port ( I : in STD_LOGIC_VECTOR (1 downto 0);
        EN : in STD_LOGIC;
        Y : out STD_LOGIC_VECTOR (3 downto 0));
end component;

signal Add, Neg, Mov, Jmp, En_Sel : std_logic;
--En_Sel is for enable registers only if the instruction is not Jmp

begin

   Decoder_2_to_4_0: Decoder_2_to_4
      port map(
           I(0)   => Instruction(10),
           I(1)   => Instruction(11),
           EN     => '1',
           Y(0)   => Add,
           Y(1)   => Neg,
           Y(2)   => Mov,
           Y(3)   => Jmp);

   Load_Sel   <= Mov;
   AddSub_Sel  <= Add or Neg;
   Neg_Sel    <= Neg;

   En_Sel     <= Add or Mov or Neg;
```

```
  Imm_Val     <= Instruction(3 downto 0);

  Reg_En(0)   <= En_Sel and Instruction(7);
  Reg_En(1)   <= En_Sel and Instruction(8);
  Reg_En(2)   <= En_Sel and Instruction(9);

  Reg_Sel_A   <= Instruction(9 downto 7);
  Reg_Sel_B   <= Instruction(6 downto 4);

  --check the register R = 0 before jump
  Jump_Flag   <= Jmp and (not (JumpCheck(0) or JumpCheck(1) or JumpCheck(2) or JumpCheck(3)));
  --address to jump is in the first three digits of the instruction
  Jump_Add    <= Instruction(2 downto 0);

end Behavioral;
```

RTL SCHEMATICS

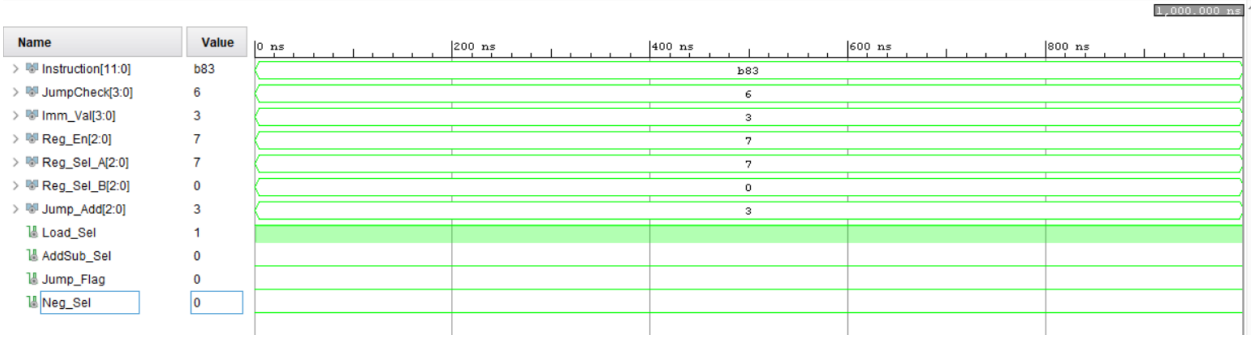


BEHAVIORAL SIMULATION CODE

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

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

architecture Behavioral of Sim_InsDecoder is

component Ins_Decoder is
    Port ( Instruction      : in STD_LOGIC_VECTOR (11 downto 0);
           JumpCheck        : in STD_LOGIC_VECTOR (3 downto 0);
           Reg_En           : out STD_LOGIC_VECTOR (2 downto 0);
           Load_Sel         : out STD_LOGIC;
           Imm_Val          : out STD_LOGIC_VECTOR (3 downto 0);
           Reg_Sel_A        : out STD_LOGIC_VECTOR (2 downto 0);
           Reg_Sel_B        : out STD_LOGIC_VECTOR (2 downto 0);
           AddSub_Sel       : out STD_LOGIC;
           Jump_Flag        : out STD_LOGIC;
           Jump_Add         : out STD_LOGIC_VECTOR (2 downto 0);
           Neg_Sel          : out STD_LOGIC);
end component;

signal Instruction                      : std_logic_vector(11 downto 0);
signal JumpCheck, Imm_Val               : std_logic_vector(3 downto 0);
signal Reg_En, Reg_Sel_A, Reg_Sel_B, Jump_Add   : std_logic_vector(2 downto 0);
signal Load_Sel, AddSub_Sel, Jump_Flag, Neg_Sel : std_logic;

begin

UUT : Ins_Decoder port map(
        Instruction => Instruction,
        JumpCheck   => JumpCheck,
        Reg_En      => Reg_En,
        Load_Sel    => Load_Sel,
        Imm_Val     => Imm_Val,
        Reg_Sel_A   => Reg_Sel_A,
```

```
      Reg_Sel_B   => Reg_Sel_B,
      AddSub_Sel  => AddSub_Sel,
      Jump_Flag   => Jump_Flag,
      Jump_Add    => Jump_Add,
      Neg_Sel     => Neg_Sel);

process
  begin
      Instruction <= "101110000011";
      JumpCheck   <= "0110";
  wait;
end process;

end Behavioral;
```

TIMING DIAGRAM



PROGRAM ROM

Machine codes for each instruction that is needed to be executed using the processor are stored in separate locations of the Program ROM.

There are seven ROM locations with 12 – bits for each instruction.

The first two bits indicate the opcode for the instruction. The next six bits represent the register addresses with 3 – bits for each. The last four bits are for the immediate value and the same last three bits are for the jump address.

ASSEMBLY PROGRAM AND MACHINE CODE REPRESENTATION

"101110000011" -> MOVI R7, 3 ; R7 <- 3 - 000 - 0

"100010000011" -> MOVI R1, 3 ; R1 <- 3 - 001 - 1

"100100000001" -> MOVI R2, 1 ; R2 <- 1 - 010 - 2

"010100000000" -> NEG R2 ; R2 <- -R2 - 011 - 3

"000010100000" -> ADD R1, R2 ; R1 <- R1 + R2 - 100 - 4

"001110010000" -> ADD R7, R1 ; R7 <- R7 + R1 - 101 - 5

"110010000110" -> JZR R1, 6 ; If R1 = 0 jump to line 6 - 110 - 6

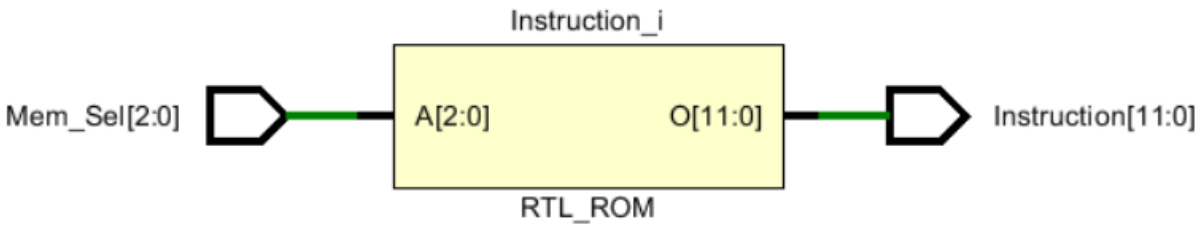
"110000000100" -> JZR R0, 4 ; If R0 = 0 jump to line 4 - 111 - 7

VHDL CODE

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

entity Program_ROM is
   Port ( Mem_Sel      : in STD_LOGIC_VECTOR (2 downto 0);
          Instruction  : out STD_LOGIC_VECTOR (11 downto 0));
end Program_ROM;

architecture Behavioral of Program_ROM is

type rom_type is array( 0 to 7 ) of std_logic_vector (11 downto 0);
   signal instruction_ROM : rom_type := (
      "101110000011", --10 111 000 0011 -> MOVI R7, 3    ; R7 <- 3              - 000 - 0
      "100010000011", --10 001 000 0011 -> MOVI R1, 3    ; R1 <- 3              - 001 - 1
      "100100000001", --10 010 000 0001 -> MOVI R2, 1    ; R2 <- 1              - 010 - 2
      "010100000000", --01 010 000 0000 -> NEG R2        ; R2 <- -R2            - 011 - 3
      "000010100000", --00 001 010 0000 -> ADD R1, R2    ; R1 <- R1 + R2        - 100 - 4
      "001110010000", --00 111 001 0000 -> ADD R7, R1    ; R7 <- R7 + R1        - 101 - 5
      "110010000110", --11 001 000 0110 -> JZR R1, 6     ; If R1 = 0 jump to line 6  - 110 - 6
      "110000000100"  --11 000 000 0100 -> JZR R0, 4     ; If R0 = 0 jump to line 4  - 111 - 7
      );
```

```
begin

Instruction <= instruction_ROM(to_integer(unsigned(Mem_Sel)));

end Behavioral;
```

RTL SCHEMATICS

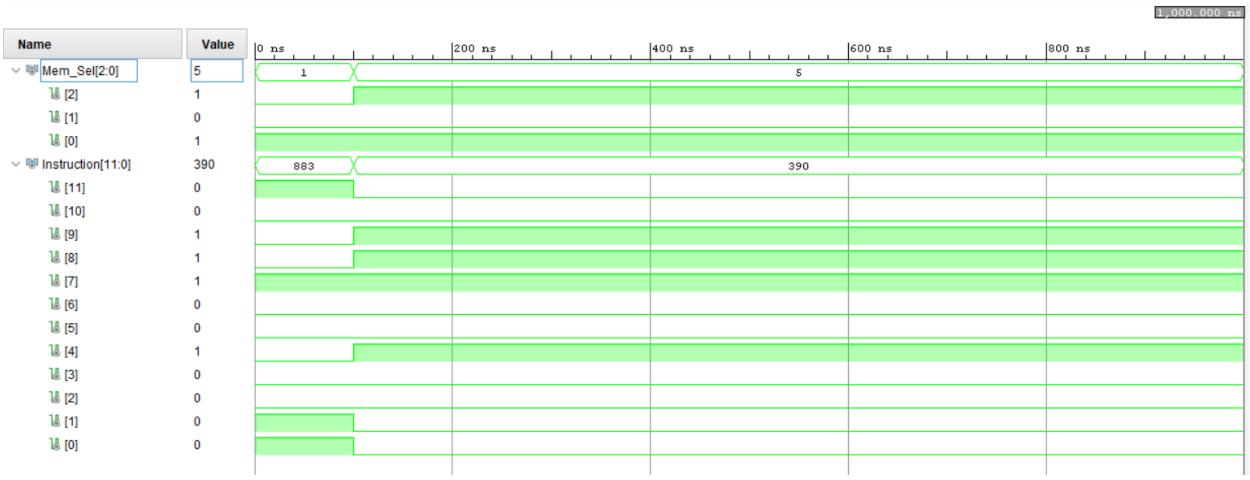


BEHAVIORAL SIMULATION CODE

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

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

architecture Behavioral of Sim_ProgramROM is

component Program_ROM is
    Port ( Mem_Sel      : in STD_LOGIC_VECTOR (2 downto 0);
           Instruction  : out STD_LOGIC_VECTOR (11 downto 0));
end component;

signal Mem_Sel : std_logic_vector(2 downto 0);
signal Instruction : std_logic_vector(11 downto 0);

begin
```

```
UUT : Program_ROM port map(
      Mem_Sel     => Mem_Sel,
      Instruction => Instruction);

process
   begin
      Mem_Sel <= "001";
      wait for 100ns;

      Mem_Sel <= "101";
   wait;
end process;

end Behavioral;
```

TIMING DIAGRAM



LOOKUP TABLE

Value of the Reg7 is directed to the Basys3 LEDs and 7 – segment display as output.

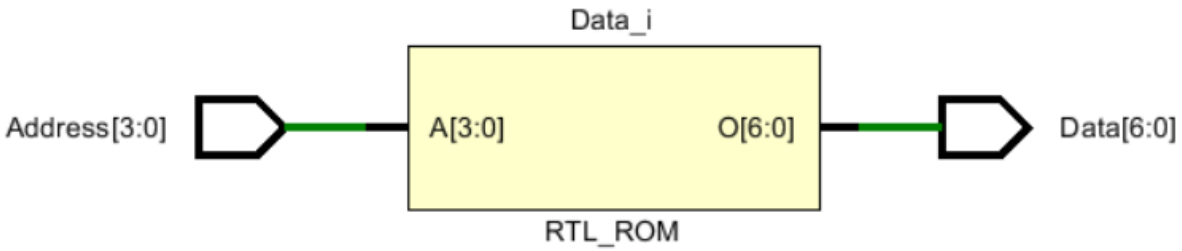
To achieve the correct functionality of the segment display, each output must be mapped using a look-up table.

Hence the 4 – bit output value is converted into a 7 – bit value where each bit represents each segment of the display.

VHDL CODE

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity LUT_7seg is
    Port ( Address : in STD_LOGIC_VECTOR (3 downto 0);
           Data : out STD_LOGIC_VECTOR (6 downto 0));
end LUT_7seg;

architecture Behavioral of LUT_7seg is

type rom_type is array (0 to 15) of std_logic_vector(6 downto 0);
signal sevenSegROM : rom_type := (
    "1000000", --0
    "1111001", --1
    "0100100", --2
    "0110000", --3
    "0011001", --4
    "0010010", --5
    "0000010", --6
    "1111000", --7
    "0000000", --8
    "0000100", --9
    "0001000", --A
    "0000011", --B
    "1000110", --C
    "0100001", --D
    "0000110", --E
    "0001110"  --F
    );

begin

Data <= sevenSegROM(to_integer(unsigned(Address)));

end Behavioral;
```

RTL SCHEMATICS



REGISTER BANK

Register bank contains eight 4 – bit registers (Reg0 to Reg7) implemented using D-Flipflops. A 3 – 8 decoder is used to enable each register separately according to the instruction.

In the register bank, Reg0 is hardcoded to “0000” to simplify the implementation of the NEG instruction.

Furthermore, Reg7, which holds the final output is directly connected to the Basys3 LEDs and the 7 – segment display using a lookup table.

VHDL CODE

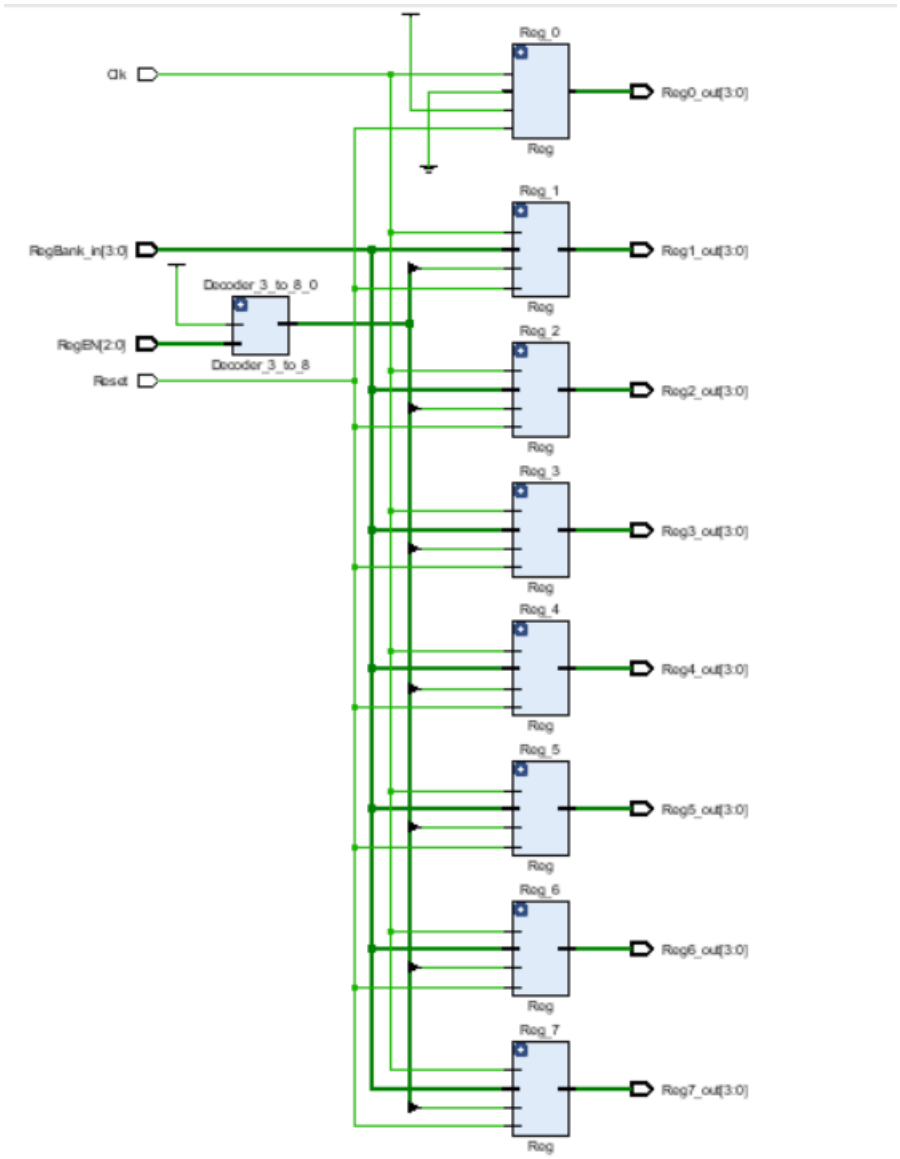
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity RegisterBank is
    Port ( RegBank_in   : in STD_LOGIC_VECTOR (3 downto 0);
           RegEN        : in STD_LOGIC_VECTOR (2 downto 0);
           Reset        : in STD_LOGIC;
           Clk          : in STD_LOGIC;
           Reg0_out     : out STD_LOGIC_VECTOR (3 downto 0);
           Reg1_out     : out STD_LOGIC_VECTOR (3 downto 0);
           Reg2_out     : out STD_LOGIC_VECTOR (3 downto 0);
```

```
        Reg3_out    : out STD_LOGIC_VECTOR (3 downto 0);
        Reg4_out    : out STD_LOGIC_VECTOR (3 downto 0);
        Reg5_out    : out STD_LOGIC_VECTOR (3 downto 0);
        Reg6_out    : out STD_LOGIC_VECTOR (3 downto 0);
        Reg7_out    : out STD_LOGIC_VECTOR (3 downto 0));
end RegisterBank;

architecture Behavioral of RegisterBank is

component Reg
   Port ( D    : in STD_LOGIC_VECTOR (3 downto 0);
          En   : in STD_LOGIC;
          R    : in STD_LOGIC;
          Clk  : in STD_LOGIC;
          Q    : out STD_LOGIC_VECTOR (3 downto 0));
end component;

 --works as register enabling unit
component Decoder_3_to_8
   Port ( I    : in STD_LOGIC_VECTOR (2 downto 0);
          EN   : in STD_LOGIC;
          Y    : out STD_LOGIC_VECTOR (7 downto 0));
end component;

signal in_RegEN : STD_LOGIC_VECTOR (7 downto 0); --enabler for each register

begin

   Decoder_3_to_8_0 : Decoder_3_to_8
      port map (
      I(2 downto 0)   => RegEN(2 downto 0),
      EN              => '1',
      Y(7 downto 0)   => in_RegEN(7 downto 0));

   Reg_0 : Reg
      port map(
         D(3 downto 0)   => "0000", -- Reg_0 is readOnly - to execute negate instruction
```

```
      En          => '1',  --set the Reg_0 to 0000 initially
      R           => Reset,
      Clk         => Clk,
      Q(3 downto 0)   => Reg0_out(3 downto 0));

Reg_1 : Reg
   port map(
      D(3 downto 0)   => RegBank_in(3 downto 0),
      En          => in_RegEN(1),
      R           => Reset,
      Clk         => Clk,
      Q(3 downto 0)   => Reg1_out(3 downto 0));

Reg_2 : Reg
   port map(
      D(3 downto 0)   => RegBank_in(3 downto 0),
      En          => in_RegEN(2),
      R           => Reset,
      Clk         => Clk,
      Q(3 downto 0)   => Reg2_out(3 downto 0));

Reg_3 : Reg
   port map(
      D(3 downto 0)   => RegBank_in(3 downto 0),
      En          => in_RegEN(3),
      R           => Reset,
      Clk         => Clk,
      Q(3 downto 0)   => Reg3_out(3 downto 0));

Reg_4 : Reg
   port map(
      D(3 downto 0)   => RegBank_in(3 downto 0),
      En          => in_RegEN(4),
      R           => Reset,
      Clk         => Clk,
      Q(3 downto 0)   => Reg4_out(3 downto 0));
```

```
  Reg_5 : Reg
    port map(
      D(3 downto 0)   => RegBank_in(3 downto 0),
      En              => in_RegEN(5),
      R               => Reset,
      Clk             => Clk,
      Q(3 downto 0)   => Reg5_out(3 downto 0));

  Reg_6 : Reg
    port map(
      D(3 downto 0)   => RegBank_in(3 downto 0),
      En              => in_RegEN(6),
      R               => Reset,
      Clk             => Clk,
      Q(3 downto 0)   => Reg6_out(3 downto 0));

  Reg_7 : Reg
    port map(
      D(3 downto 0)   => RegBank_in(3 downto 0),
      En              => in_RegEN(7),
      R               => Reset,
      Clk             => Clk,
      Q(3 downto 0)   => Reg7_out(3 downto 0));

end Behavioral;
```

RTL SCHEMATICS



BEHAVIORAL SIMULATION CODE

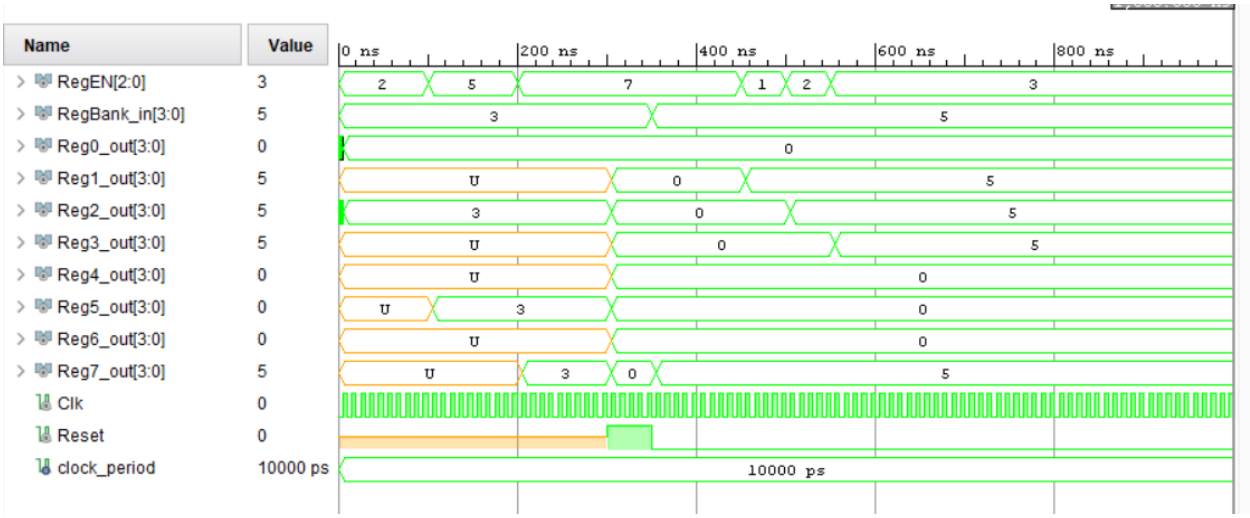
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


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


architecture Behavioral of Sim_RegisterBank is
```

```
component RegisterBank is
   Port ( RegBank_in   : in STD_LOGIC_VECTOR (3 downto 0);
          RegEN        : in STD_LOGIC_VECTOR (2 downto 0);
          Reset        : in STD_LOGIC;
          Clk          : in STD_LOGIC;
          Reg0_out     : out STD_LOGIC_VECTOR (3 downto 0);
          Reg1_out     : out STD_LOGIC_VECTOR (3 downto 0);
          Reg2_out     : out STD_LOGIC_VECTOR (3 downto 0);
          Reg3_out     : out STD_LOGIC_VECTOR (3 downto 0);
          Reg4_out     : out STD_LOGIC_VECTOR (3 downto 0);
          Reg5_out     : out STD_LOGIC_VECTOR (3 downto 0);
          Reg6_out     : out STD_LOGIC_VECTOR (3 downto 0);
          Reg7_out     : out STD_LOGIC_VECTOR (3 downto 0));
end component;

signal RegEN : std_logic_vector(2 downto 0);
signal RegBank_in, Reg0_out, Reg1_out, Reg2_out, Reg3_out, Reg4_out, Reg5_out, Reg6_out, Reg7_out : std_logic_vector(3 downto 0);
signal Clk, Reset : std_logic;

constant clock_period : time := 10ns;

begin

UUT : RegisterBank port map(
       RegBank_in  => RegBank_in,
       RegEN       => RegEN,
       Reset       => Reset,
       Clk         => Clk,
       Reg0_out    => Reg0_out,
       Reg1_out    => Reg1_out,
       Reg2_out    => Reg2_out,
       Reg3_out    => Reg3_out,
       Reg4_out    => Reg4_out,
       Reg5_out    => Reg5_out,
       Reg6_out    => Reg6_out,
       Reg7_out    => Reg7_out);
```

```
clock_process : process
   begin
      Clk <= '0';
      wait for clock_period / 2;

      Clk <= '1';
      wait for clock_period / 2;
end process;

sim : process
   begin
      RegBank_in <= "0011";
      RegEN     <= "010";
      wait for 100ns;

      RegEN  <= "101";
      wait for 100ns;

      RegEN   <= "111";
      wait for 100ns;

      Reset <= '1';
      wait for 50ns;
      Reset <= '0';

      RegBank_in <= "0101";
      RegEN      <= "111";
      wait for 100ns;

      RegEN <= "001";
      wait for 50ns;

      RegEN <= "010";
      wait for 50ns;

      RegEN <= "011";
```

```
   wait;
end process;


end Behavioral;
```

TIMING DIAGRAM



4–BIT ADD/SUBTRACT UNIT

Add/Subtract unit contains the ripple carry adder we designed in lab 03 by using four full adders.

Since the numbers are represented as signed integers, the output range of the add/subtract unit is from (-8) to 7.

Add/Subtract unit is capable of adding two 4 – bit numbers, which are provided by the two 8 – 4 multiplexers and indicate the carryout and zero using **overflow** and **zero** flags.

The **overflow** is calculated by taking **xor** between the carry outs of 3rd and 4th full adders as there is an overflow if either one of carry outs are 1 and **overflow** is 0 if and only if both carry outs are zero or one.

This unit also can calculate the negation of the given 4 – bit values.

Bitwise XOR operation is performed between the output of mux A and the **NegIn** which will be 1 only in a negate instruction.

Carry in of the adder is also NegIn and add/subtract unit is used to add the 1's complement of the value and 1 to get the 2's complement of that value which is the negation.

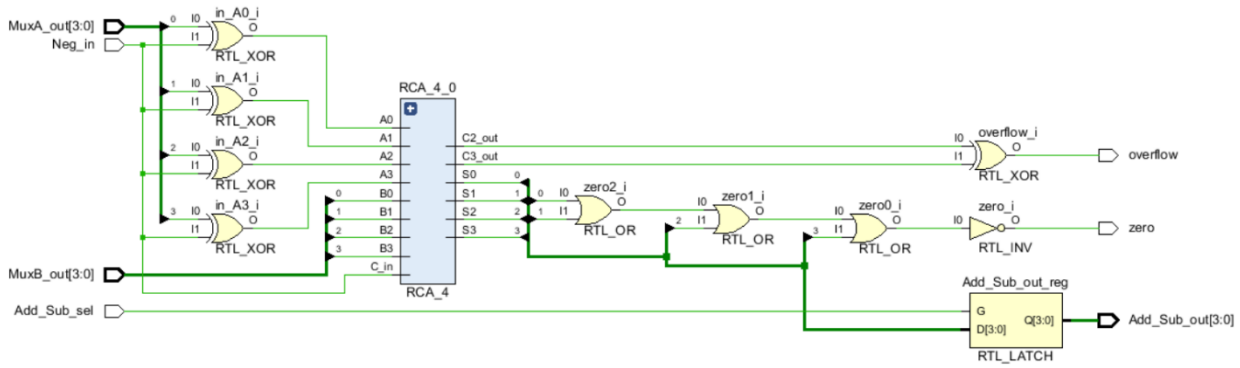
VHDL CODE

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

entity AddSubUnit is
   Port ( MuxA_out     : in STD_LOGIC_VECTOR (3 downto 0);     --outputs of the mux -> inputs for the add/sub unit
          MuxB_out     : in STD_LOGIC_VECTOR (3 downto 0);
          Add_Sub_sel  : in STD_LOGIC;                         --add/sub unit selector line
          Neg_in       : in STD_LOGIC;                         --to calculate the negation of a given number
          Add_Sub_out  : out STD_LOGIC_VECTOR (3 downto 0);    --output of the add/sub unit
          overflow     : out STD_LOGIC;                        --carry out
          zero         : out STD_LOGIC);                       --zero flag :- '1' if output is zero
end AddSubUnit;

architecture Behavioral of AddSubUnit is

component RCA_4
   Port ( A0      : in STD_LOGIC;
          A1      : in STD_LOGIC;
          A2      : in STD_LOGIC;
          A3      : in STD_LOGIC;
          B0      : in STD_LOGIC;
          B1      : in STD_LOGIC;
          B2      : in STD_LOGIC;
          B3      : in STD_LOGIC;
          C_in    : in STD_LOGIC;
          S0      : out STD_LOGIC;
          S1      : out STD_LOGIC;
          S2      : out STD_LOGIC;
          S3      : out STD_LOGIC;
          C2_out    : out STD_LOGIC;
          C3_out    : out STD_LOGIC);
end component;

signal in_A0, in_A1, in_A2, in_A3   : STD_LOGIC;
--define seperate equations in order to execute both add/sub and neg instruction

signal in_RCA_out      : STD_LOGIC_VECTOR (3 downto 0);
```

```
--output of the ripple carry adder
signal in_RCA_carryOut2, in_RCA_carryOut3  : STD_LOGIC;
--carry out of the ripple carry adder

begin
   -- calculates either addition or negation, based on the instruction
   in_A0 <= MuxA_out(0) xor Neg_in;
   in_A1 <= MuxA_out(1) xor Neg_in;
   in_A2 <= MuxA_out(2) xor Neg_in;
   in_A3 <= MuxA_out(3) xor Neg_in;

   RCA_4_0 : RCA_4
      port map(
         A0     => in_A0,
         A1     => in_A1,
         A2     => in_A2,
         A3     => in_A3,
         B0     => MuxB_out(0),
         B1     => MuxB_out(1),
         B2     => MuxB_out(2),
         B3     => MuxB_out(3),
         C_in    => Neg_in,
         S0     => in_RCA_out(0),
         S1     => in_RCA_out(1),
         S2     => in_RCA_out(2),
         S3     => in_RCA_out(3),
         C2_out  => in_RCA_carryOut2,
         C3_out  => in_RCA_carryOut3);

      process(Add_Sub_sel, in_RCA_out)
         begin

            if (Add_Sub_sel = '1') then
               Add_Sub_out <= in_RCA_out;
            end if;

         end process;
```

```
    zero       <= not(in_RCA_out(0) or in_RCA_out(1) or in_RCA_out(2) or in_RCA_out(3));
    overflow   <= in_RCA_carryOut2 xor in_RCA_carryOut3;

end Behavioral;
```

RTL SCHEMATICS



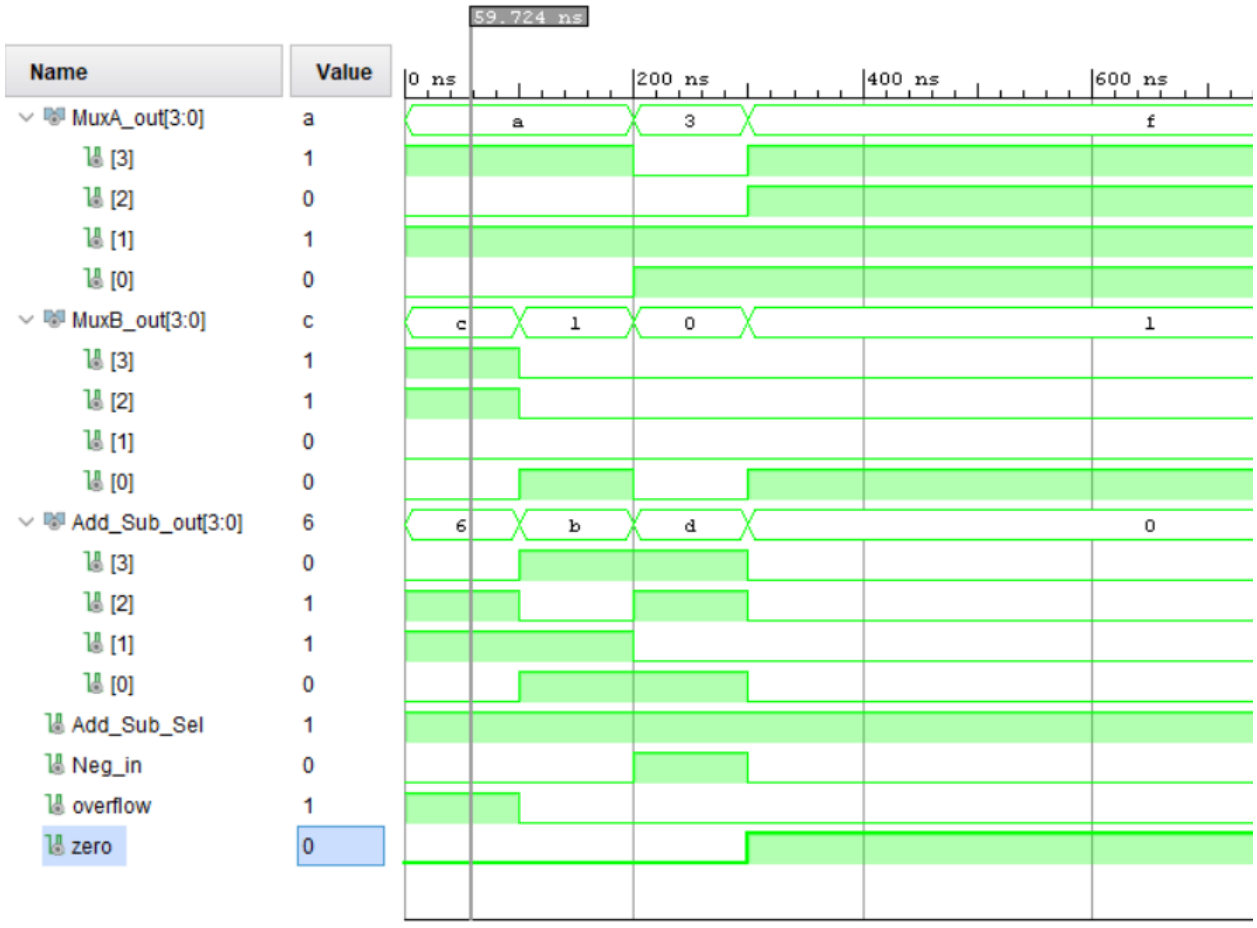
BEHAVIORAL SIMULATION CODE

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

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

architecture Behavioral of Sim_AddSubUnit is

component AddSubUnit is
   Port ( MuxA_out     : in STD_LOGIC_VECTOR (3 downto 0);     --outputs of the mux -> inputs for the add/sub unit
          MuxB_out     : in STD_LOGIC_VECTOR (3 downto 0);
          Add_Sub_sel  : in STD_LOGIC;                    --add/sub unit selector line
          Neg_in       : in STD_LOGIC;                    --to calculate the negation of a given number
          Add_Sub_out  : out STD_LOGIC_VECTOR (3 downto 0);    --output of the add/sub unit
          overflow     : out STD_LOGIC;                    --carry out
          zero         : out STD_LOGIC);
end component;
```

```
signal MuxA_out, MuxB_out, Add_Sub_out      : std_logic_vector(3 downto 0);
signal Add_Sub_Sel, Neg_in, overflow, zero  : std_logic;

begin

UUT : AddSubUnit port map(
      MuxA_out    => MuxA_out,
      MuxB_out    => MuxB_out,
      Add_Sub_Sel => Add_Sub_Sel,
      Neg_in      => Neg_in,
      Add_Sub_out => Add_Sub_out,
      overflow    => overflow,
      zero        => zero);

--number range that can be shown in 4 bits = (-8) to 7
process
   begin
      MuxA_out    <= "1010"; -- (-6)
      MuxB_out    <= "1100"; -- (-4)
      Add_Sub_sel <= '1';
      Neg_in      <= '0';
      wait for 100ns;

      MuxB_out    <= "0001"; -- 1
      wait for 100ns;

      MuxA_out    <= "0011"; -- 3
      MuxB_out    <= "0000"; -- 0
      Neg_in      <= '1';
      wait for 100ns;

      MuxA_out    <= "1111"; -- (-1)
      MuxB_out    <= "0001"; -- 1
      Neg_in      <= '0';
   wait;
end process;
```

```
end Behavioral;
```

TIMING DIAGRAM



4-BIT RIPPLE CARRY ADDER

The RCA_4 implemented in the previous lab was modified to use in the nano processor.

Carry out of the 3rd and 4th full adders are taken out as outputs and passed to the add/subtract unit to calculate the overflow.

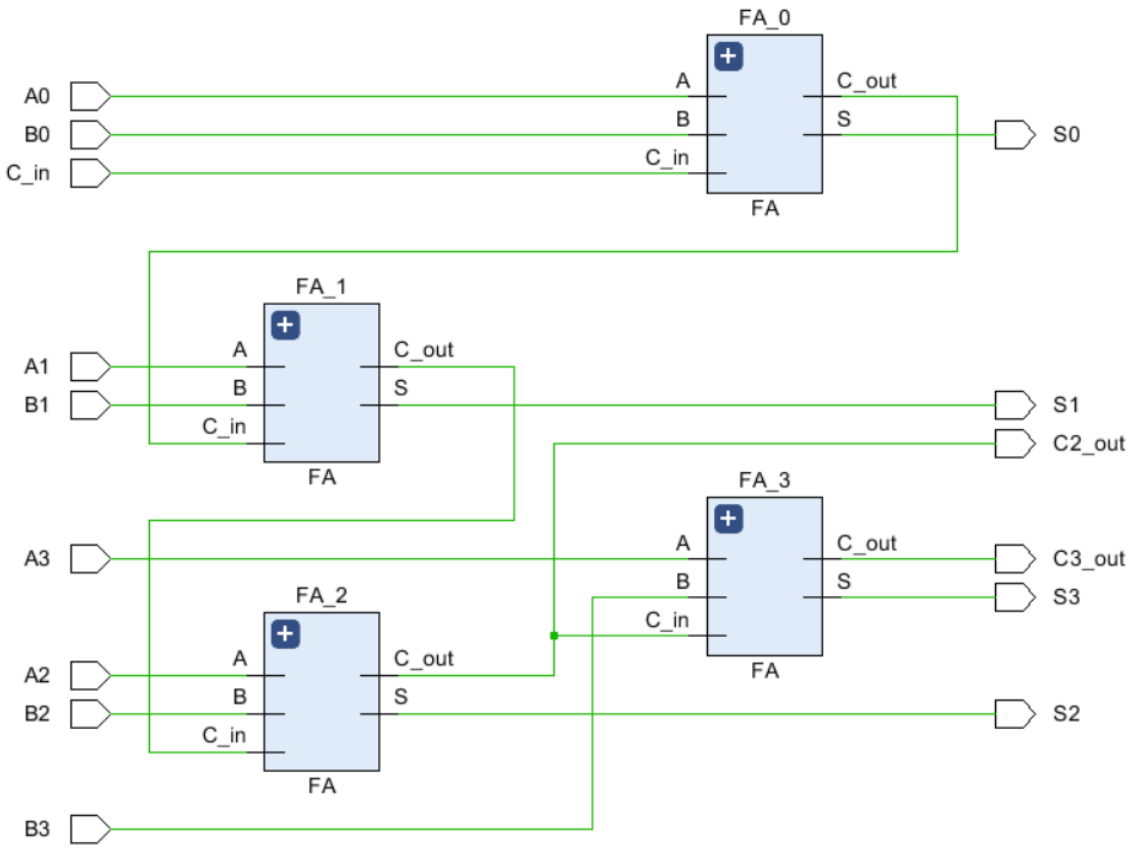
VHDL CODE

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


entity RCA_4 is
    Port ( A0      : in STD_LOGIC;
```

```
        A1      : in STD_LOGIC;
        A2      : in STD_LOGIC;
        A3      : in STD_LOGIC;
        B0      : in STD_LOGIC;
        B1      : in STD_LOGIC;
        B2      : in STD_LOGIC;
        B3      : in STD_LOGIC;
        C_in    : in STD_LOGIC;
        S0      : out STD_LOGIC;
        S1      : out STD_LOGIC;
        S2      : out STD_LOGIC;
        S3      : out STD_LOGIC;
        C2_out   : out STD_LOGIC;
        C3_out   : out STD_LOGIC);
end RCA_4;

architecture Behavioral of RCA_4 is

   component FA
      port (
      A      : in std_logic;
      B      : in std_logic;
      C_in   : in std_logic;
      S      : out std_logic;
      C_out  : out std_logic);
   end component;

signal FA0_S, FA0_C, FA1_S, FA1_C, FA2_S, FA2_C, FA3_S, FA3_C : std_logic;

begin
   FA_0 : FA
      port map (
        A      => A0,
        B      => B0,
        C_in   => C_in,
        S      => S0,
        C_Out  => FA0_C);
```

```
  FA_1 : FA
    port map (
      A      => A1,
      B      => B1,
      C_in   => FA0_C,
      S      => S1,
      C_Out  => FA1_C);

  FA_2 : FA
    port map (
      A      => A2,
      B      => B2,
      C_in   => FA1_C,
      S      => S2,
      C_Out  => FA2_C);

  FA_3 : FA
    port map (
      A      => A3,
      B      => B3,
      C_in   => FA2_C,
      S      => S3,
      C_Out  => C3_out);

  C2_out <= FA2_C;

end Behavioral;
```

RTL SCHEMATIC



PROGRAM COUNTER

The program counter decides which instruction gets executed next.

It constantly increments the pc–register based on the clock pulse and 2 – way 3–bit multiplexer checks whether the jump flag is enabled.

If the jump flag is enabled, pc – register jumps to the address provided by the instruction decoder, else pc–register will set to incremented address.

VHDL CODE

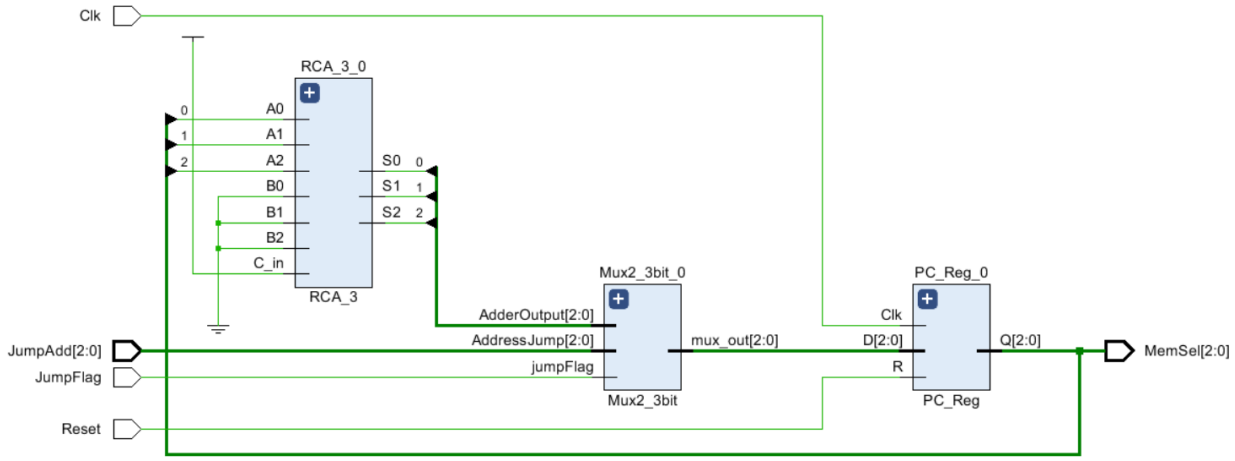
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ProgramCounter is
    Port ( Clk      : in STD_LOGIC;
           Reset    : in STD_LOGIC;
```

```
        JumpFlag : in STD_LOGIC;
        JumpAdd  : in STD_LOGIC_VECTOR (2 downto 0);
        MemSel   : out STD_LOGIC_VECTOR (2 downto 0));
end ProgramCounter;

architecture Behavioral of ProgramCounter is

component PC_Reg is
   Port (    D     : in STD_LOGIC_VECTOR (2 downto 0);
             R     : in STD_LOGIC;
             Clk   : in STD_LOGIC;
             Q     : out STD_LOGIC_VECTOR (2 downto 0));
end component;

component RCA_3 is
   Port ( A0      : in STD_LOGIC;
          A1      : in STD_LOGIC;
          A2      : in STD_LOGIC;

          B0      : in STD_LOGIC;
          B1      : in STD_LOGIC;
          B2      : in STD_LOGIC;

          C_in    : in STD_LOGIC;

          S0      : out STD_LOGIC;
          S1      : out STD_LOGIC;
          S2      : out STD_LOGIC;

          C_out   : out STD_LOGIC);
end component;

component Mux2_3bit is
   Port ( AdderOutput  : in STD_LOGIC_VECTOR (2 downto 0);
          AddressJump  : in STD_LOGIC_VECTOR (2 downto 0);
          jumpFlag     : in STD_LOGIC;
          mux_out      : out STD_LOGIC_VECTOR (2 downto 0));
```

```
end component;

signal Mux_in          : std_logic_vector(2 downto 0);
signal nextIns         : std_logic_vector(2 downto 0);
signal currIns         : std_logic_vector(2 downto 0);

begin

   PC_Reg_0 : PC_Reg
      port map(
         D   => nextIns,
         R   => Reset,
         Clk => Clk,
         Q   => currIns);

   RCA_3_0 : RCA_3
      port map(
         A0     => currIns(0),
         A1     => currIns(1),
         A2     => currIns(2),
         B0     => '0',
         B1     => '0',
         B2     => '0',
         C_in    => '1',
         S0     => Mux_in(0),
         S1     => Mux_in(1),
         S2     => Mux_in(2)); -- C_out is not included

   Mux2_3bit_0 : Mux2_3bit
      port map(
         AdderOutput => Mux_in,
         AddressJump => JumpAdd,
         jumpFlag    => JumpFlag,
         mux_out     => nextIns);

   MemSel <= currIns;
```

```
end Behavioral;
```

RTL SCHEMATICS



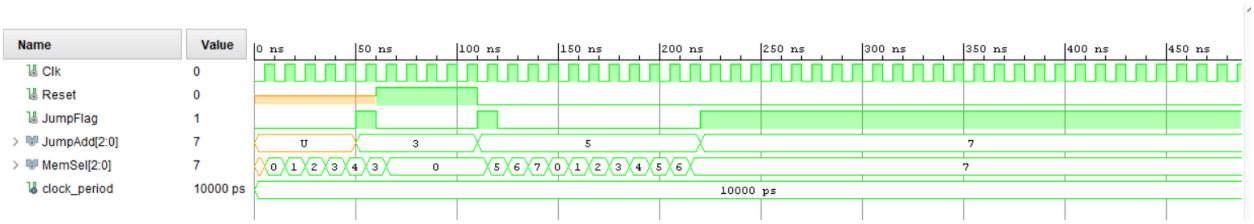
BEHAVIORAL SIMULATION CODE

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

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

architecture Behavioral of Sim_ProgramCounter is

component ProgramCounter is
    Port ( Clk      : in STD_LOGIC;
           Reset    : in STD_LOGIC;
           JumpFlag : in STD_LOGIC;
           JumpAdd  : in STD_LOGIC_VECTOR (2 downto 0);
           MemSel   : inout STD_LOGIC_VECTOR (2 downto 0));
end component;

signal Clk, Reset, JumpFlag : std_logic;
signal JumpAdd, MemSel      : std_logic_vector(2 downto 0);

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

```
begin

UUT : ProgramCounter port map(
      Clk        => Clk,
      Reset      => Reset,
      JumpFlag   => JumpFlag,
      JumpAdd    => JumpAdd,
      MemSel     => MemSel);

clock_process : process
   begin
      Clk <= '0';
      wait for clock_period / 2;

      Clk <= '1';
      wait for clock_period / 2;

end process;

sim : process
   begin
      JumpFlag <= '0';
      wait for 50ns;

      JumpFlag <= '1';
      JumpAdd  <= "011";
      wait for clock_period;

      JumpFlag <= '0';
      Reset <= '1';
      wait for 50ns;
      Reset <= '0';

      JumpFlag <= '1';
      JumpAdd  <= "101";
      wait for clock_period;
```

```
        JumpFlag <= '0';
        wait for 100ns;

        JumpFlag <= '1';
        JumpAdd  <= "111";
    wait;
end process;

end Behavioral;
```

TIMING DIAGRAM



PC REGISTER

PC–register is a 3–bit register that holds the ROM address of the next instruction to be executed.

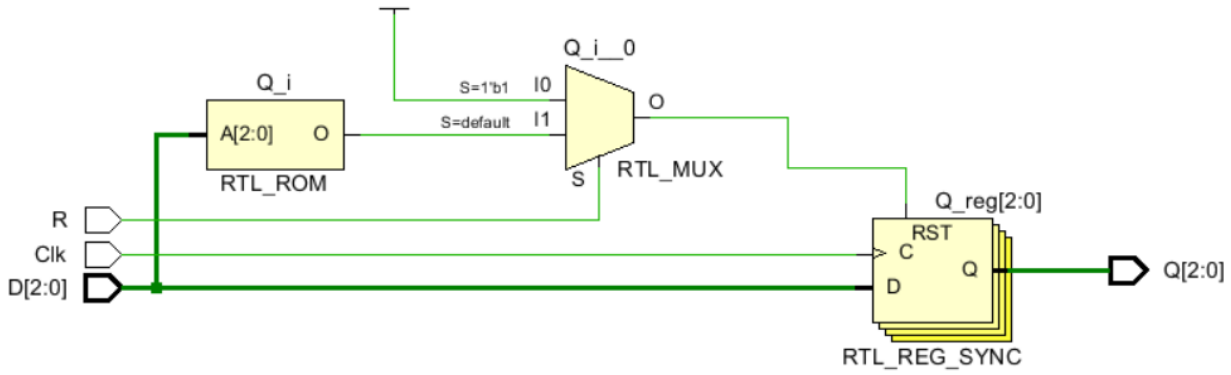
This can be reset to “000” using the Reset push button whenever the user wants.

VHDL CODE

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

entity PC_Reg is
    Port (    D     : in STD_LOGIC_VECTOR (2 downto 0);
              R     : in STD_LOGIC;                    --reset push button
              Clk   : in STD_LOGIC;
              Q     : out STD_LOGIC_VECTOR (2 downto 0));   --memory select
end PC_Reg;

architecture Behavioral of PC_Reg is
```

```
begin


   process (Clk) begin
      if (rising_edge(Clk)) then
         if R = '1' then
            Q <= "000";
         else
            if D = "UUU" then
               Q <= "000";
            else
               Q <= D;
            end if;
         end if;
      end if;
   end process;

end Behavioral;
```

RTL SCHEMATICS



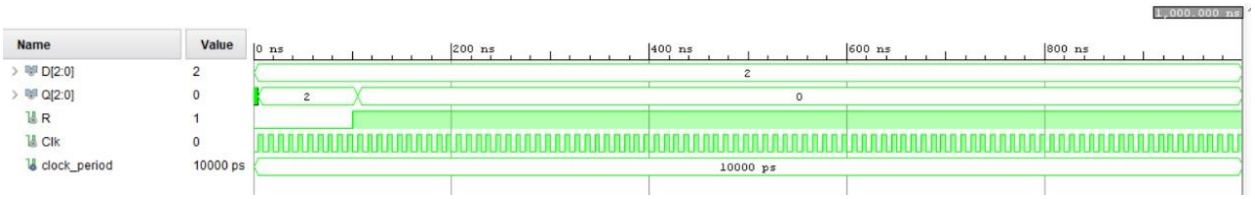
BEHAVIORAL SIMULATION CODE

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


entity Sim_PC_Reg is
--  Port ( );
```

```
end Sim_PC_Reg;

architecture Behavioral of Sim_PC_Reg is

component PC_Reg is
   Port (    D     : in STD_LOGIC_VECTOR (2 downto 0);
             R     : in STD_LOGIC;                    --reset push button
             Clk   : in STD_LOGIC;
             Q     : out STD_LOGIC_VECTOR (2 downto 0));   --memory select
end component;

signal D, Q           : std_logic_vector(2 downto 0);
signal R, Clk         : std_logic;

constant clock_period   : time := 10ns;

begin

UUT : PC_Reg port map(
      D   => D,
      R   => R,
      Clk => Clk,
      Q   => Q);

clock_process : process
   begin
      Clk <= '0';
      wait for clock_period / 2;

      Clk <= '1';
      wait for clock_period / 2;
end process;

sim : process
   begin
      D <= "010";
      R <= '0';
```

```
        wait for 100ns;


        R <= '1';
    wait;
end process;


end Behavioral;
```

TIMING DIAGRAM



3–BIT ADDER

This is a ripple carry adder with three full adders, which is used to increment the pc–register value by 1. The output of the adder is directed to the 2–way 3–bit mux.

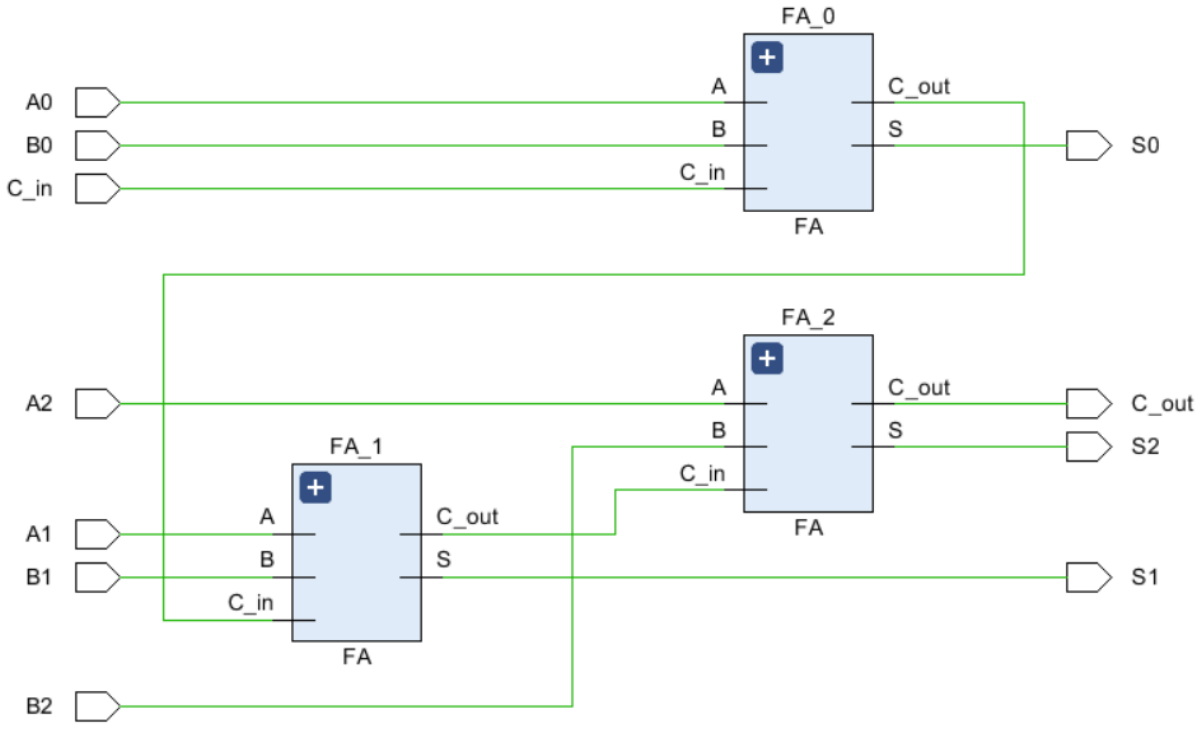
VHDL CODE

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


entity RCA_3 is
    Port ( A0      : in STD_LOGIC;
           A1      : in STD_LOGIC;
           A2      : in STD_LOGIC;

           B0      : in STD_LOGIC;
           B1      : in STD_LOGIC;
           B2      : in STD_LOGIC;

           C_in    : in STD_LOGIC;

           S0      : out STD_LOGIC;
           S1      : out STD_LOGIC;
```

```
        S2      : out STD_LOGIC;

        C_out   : out STD_LOGIC);
end RCA_3;

architecture Behavioral of RCA_3 is

   component FA
      port (
         A      : in std_logic;
         B      : in std_logic;
         C_in   : in std_logic;
         S      : out std_logic;
         C_out  : out std_logic);
   end component;

signal FA0_C, FA1_C        : std_logic;

begin

   FA_0 : FA
      port map (
         A => A0,
         B => B0,
         C_in => C_in,
         S => S0,
         C_Out => FA0_C);

   FA_1 : FA
      port map (
         A => A1,
         B => B1,
         C_in => FA0_C,
         S => S1,
         C_Out => FA1_C);

   FA_2 : FA
```

```
    port map (
      A => A2,
      B => B2,
      C_in => FA1_C,
      S => S2,
      C_Out => C_Out);

end Behavioral;
```

RTL SCHEMATICS



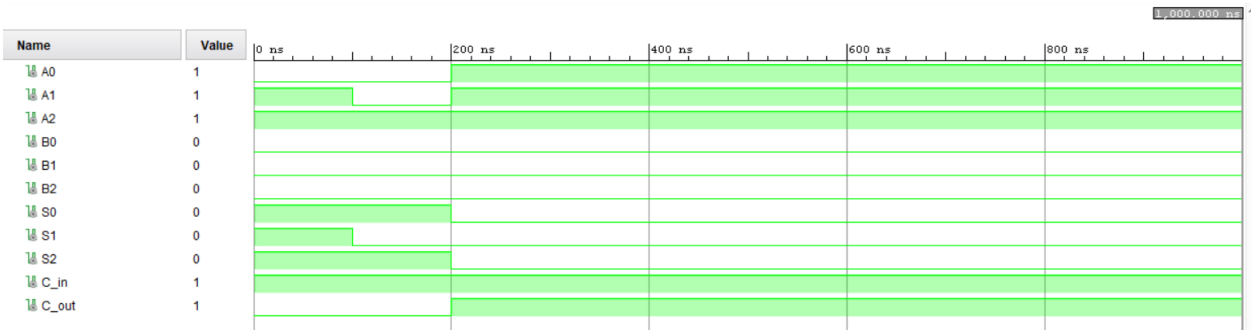
BEHAVIORAL SIMULATION CODE

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

entity Sim_RCA_3 is
--  Port ( );
end Sim_RCA_3;

architecture Behavioral of Sim_RCA_3 is
```

```
component RCA_3 is
   Port ( A0      : in STD_LOGIC;
          A1      : in STD_LOGIC;
          A2      : in STD_LOGIC;

          B0      : in STD_LOGIC;
          B1      : in STD_LOGIC;
          B2      : in STD_LOGIC;

          C_in    : in STD_LOGIC;

          S0      : out STD_LOGIC;
          S1      : out STD_LOGIC;
          S2      : out STD_LOGIC;

          C_out   : out STD_LOGIC);
end component;

signal A0, A1, A2, B0, B1, B2, S0, S1, S2, C_in, C_out : std_logic;

begin

UUT : RCA_3 port map(
      A0     => A0,
      A1     => A1,
      A2     => A2,
      B0     => B0,
      B1     => B1,
      B2     => B2,
      C_in   => C_in,
      S0     => S0,
      S1     => S1,
      S2     => S2,
      C_out  => C_out);

process
```

```
  begin
     B0 <= '0';
     B1 <= '0';
     B2 <= '0';
     C_in <= '1';
     A0 <= '0';
     A1 <= '1';
     A2 <= '1';
     wait for 100ns;

     A1 <= '0';
     wait for 100ns;

     A1 <= '1';
     A0 <= '1';
  wait;
end process;

end Behavioral;
```

TIMING DIAGRAM



K–WAY B–BIT MUX

The design was based on the multiplexers we implemented in lab 4.

The only difference is rather than using single data lines, a k–way b–bit mux uses **k** number of input busses and one output bus, which contains **b** number of bits.

Each mux has a path selector based on the **k** value.

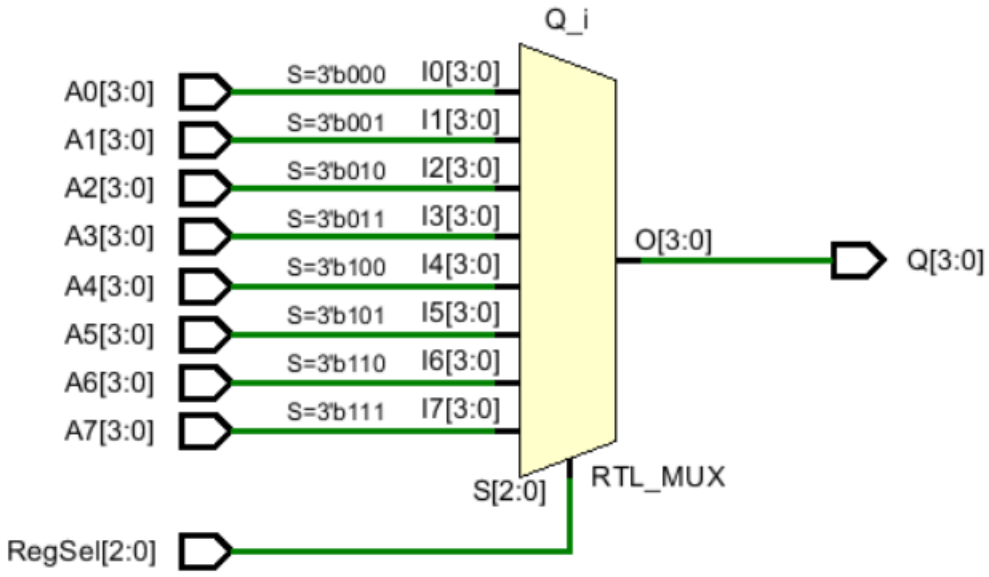
8–WAY 4–BIT MUX

VHDL CODE

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

entity Mux8_4bit is
    Port ( A0       : in STD_LOGIC_VECTOR (3 downto 0);
           A1       : in STD_LOGIC_VECTOR (3 downto 0);
           A2       : in STD_LOGIC_VECTOR (3 downto 0);
           A3       : in STD_LOGIC_VECTOR (3 downto 0);
           A4       : in STD_LOGIC_VECTOR (3 downto 0);
           A5       : in STD_LOGIC_VECTOR (3 downto 0);
           A6       : in STD_LOGIC_VECTOR (3 downto 0);
           A7       : in STD_LOGIC_VECTOR (3 downto 0);
           RegSel   : in STD_LOGIC_VECTOR (2 downto 0);
           Q        : out STD_LOGIC_VECTOR (3 downto 0));
end Mux8_4bit;

architecture Behavioral of Mux8_4bit is

begin

    process(A0, A1, A2, A3, A4, A5, A6, A7, RegSel)
        begin
          case RegSel is

              when "000" => Q <= A0;
              when "001" => Q <= A1;
              when "010" => Q <= A2;
              when "011" => Q <= A3;
              when "100" => Q <= A4;
              when "101" => Q <= A5;
              when "110" => Q <= A6;
```

```
            when "111" => Q <= A7;
            when others => Q <= "0000";

        end case;
    end process;

end Behavioral;
```

RTL SCHEMATICS



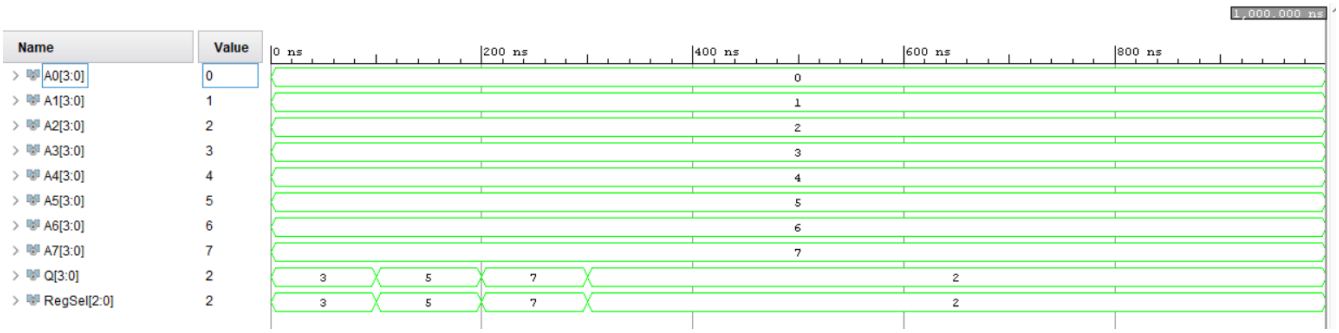
BEHAVIORAL SIMULATION CODE

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

entity Sim_Mux8_4bit is
--  Port ( );
end Sim_Mux8_4bit;

architecture Behavioral of Sim_Mux8_4bit is

component Mux8_4bit is
```

```
    Port ( A0       : in STD_LOGIC_VECTOR (3 downto 0);
           A1       : in STD_LOGIC_VECTOR (3 downto 0);
           A2       : in STD_LOGIC_VECTOR (3 downto 0);
           A3       : in STD_LOGIC_VECTOR (3 downto 0);
           A4       : in STD_LOGIC_VECTOR (3 downto 0);
           A5       : in STD_LOGIC_VECTOR (3 downto 0);
           A6       : in STD_LOGIC_VECTOR (3 downto 0);
           A7       : in STD_LOGIC_VECTOR (3 downto 0);
           RegSel   : in STD_LOGIC_VECTOR (2 downto 0);
           Q        : out STD_LOGIC_VECTOR (3 downto 0));
end component;

signal A0, A1, A2, A3, A4, A5, A6, A7, Q : std_logic_vector(3 downto 0);
signal RegSel                            : std_logic_vector(2 downto 0);

begin

UUT : Mux8_4bit port map(
        A0      => A0,
        A1      => A1,
        A2      => A2,
        A3      => A3,
        A4      => A4,
        A5      => A5,
        A6      => A6,
        A7      => A7,
        RegSel  => RegSel,
        Q       => Q);

process
    begin
        A0 <= "0000";
        A1 <= "0001";
        A2 <= "0010";
        A3 <= "0011";
        A4 <= "0100";
        A5 <= "0101";
```

```
    A6 <= "0110";
    A7 <= "0111";
    RegSel <= "011";
    wait for 100ns;

    RegSel <= "101";
    wait for 100ns;

    RegSel <= "111";
    wait for 100ns;

    RegSel <= "010";
    wait for 100ns;
  wait;
end process;

end Behavioral;
```

TIMING DIAGRAM



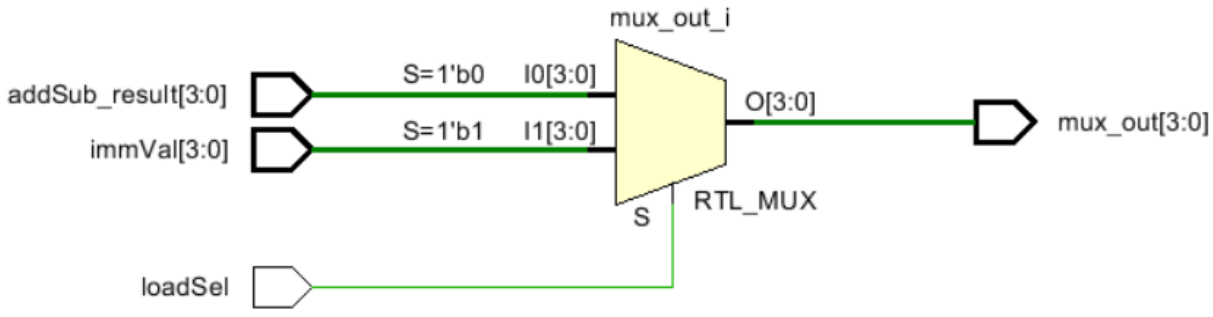
2–WAY 4–BIT MUX

VHDL CODE

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

entity Mux2_4bit is
    Port ( immVal          : in STD_LOGIC_VECTOR (3 downto 0);
```

```
        addSub_result    : in STD_LOGIC_VECTOR (3 downto 0);
        loadSel          : in STD_LOGIC; --loadSel <= Mov
        mux_out          : out STD_LOGIC_VECTOR (3 downto 0));
end Mux2_4bit;

architecture Behavioral of Mux2_4bit is

begin

   process(immVal, addSub_result, loadSel)
      begin
         case loadSel is

            when '0'       => mux_out <= addSub_result;
            when '1'       => mux_out <= immVal;
            when others    => mux_out <= "0000";

         end case;
      end process;

end Behavioral;
```

RTL SCHEMATICS

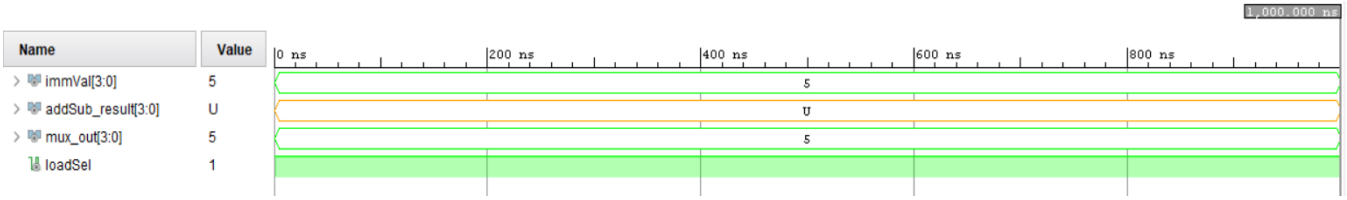


BEHAVIORAL SIMULATION CODE

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

```
entity Sim_Mux2_4bit is
--  Port ( );
end Sim_Mux2_4bit;

architecture Behavioral of Sim_Mux2_4bit is

component Mux2_4bit is
   Port ( immVal          : in STD_LOGIC_VECTOR (3 downto 0);
          addSub_result    : in STD_LOGIC_VECTOR (3 downto 0);
          loadSel          : in STD_LOGIC; --loadSel <= Mov
          mux_out          : out STD_LOGIC_VECTOR (3 downto 0));
end component;

signal immVal, addSub_result, mux_out   : std_logic_vector(3 downto 0);
signal loadSel                          : std_logic;

begin

UUT : Mux2_4bit port map(
       immVal          => immVal,
       addSub_result   => addSub_result,
       loadSel         => loadSel,
       mux_out         => mux_out);

process
   begin
       immVal <= "0101";
       loadSel <= '1';
   wait;
end process;

end Behavioral;
```

TIMING DIAGRAM

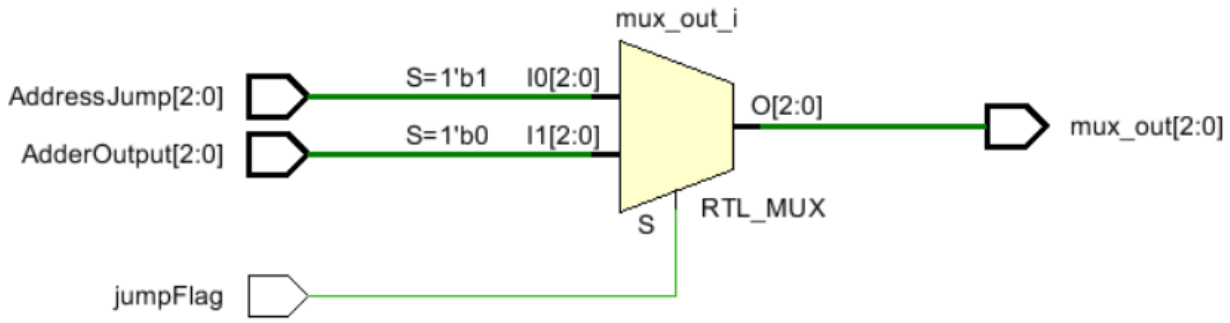


2–WAY 3–BIT MUX

VHDL CODE

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

entity Mux2_3bit is
    Port ( AdderOutput  : in STD_LOGIC_VECTOR (2 downto 0);
           AddressJump  : in STD_LOGIC_VECTOR (2 downto 0);
           jumpFlag     : in STD_LOGIC;
           mux_out      : out STD_LOGIC_VECTOR (2 downto 0));
end Mux2_3bit;

architecture Behavioral of Mux2_3bit is

begin

    process(AdderOutput, AddressJump, jumpFlag)
        begin
            case jumpFlag is

                when '1'    => mux_out <= AddressJump;
                when '0'    => mux_out <= AdderOutput;
                when others => mux_out <= "000";

            end case;
        end process;

end Behavioral;
```

RTL SCHEMATICS

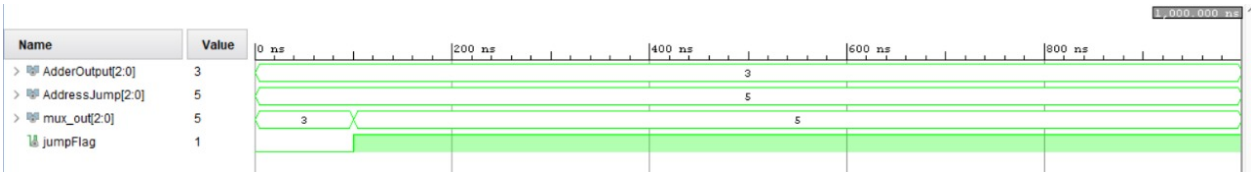


BEHAVIORAL SIMULATION CODE

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

entity Mux2_3bit is
    Port ( AdderOutput  : in STD_LOGIC_VECTOR (2 downto 0);
           AddressJump  : in STD_LOGIC_VECTOR (2 downto 0);
           jumpFlag     : in STD_LOGIC;
           mux_out      : out STD_LOGIC_VECTOR (2 downto 0));
end Mux2_3bit;

architecture Behavioral of Mux2_3bit is

begin

    process(AdderOutput, AddressJump, jumpFlag)
        begin
            case jumpFlag is

                when '1'    => mux_out <= AddressJump;
                when '0'    => mux_out <= AdderOutput;
                when others => mux_out <= "000";

            end case;
        end process;
```

```
end Behavioral;
```

TIMING DIAGRAM



REGISTER

A previously implemented register is used for the register component and modified to have a **Reset** option.

Reset input is connected to the same push button as the program counter.

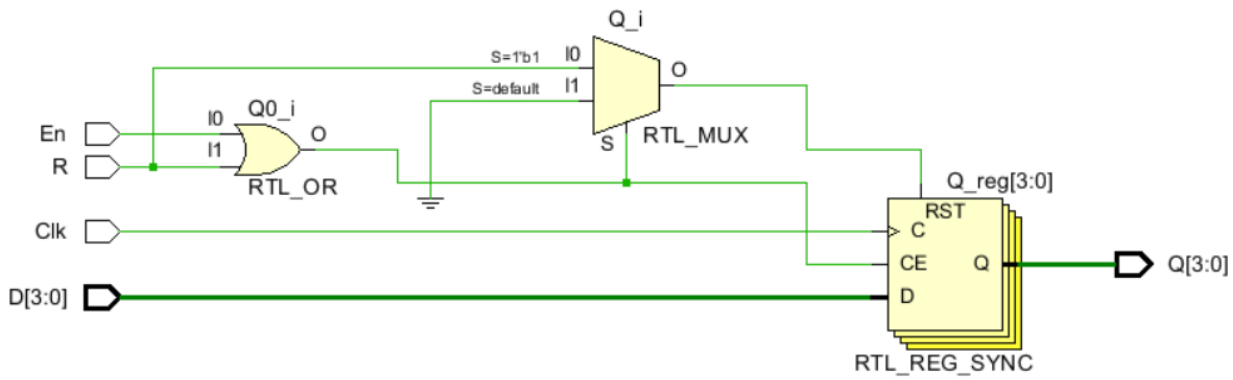
Pushing the reset button will cause all the registers to store the value **"0000"** in them.

VHDL CODE

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

entity Reg is
    Port ( D    : in STD_LOGIC_VECTOR (3 downto 0);
           En   : in STD_LOGIC;
           R    : in STD_LOGIC;
           Clk  : in STD_LOGIC;
           Q    : out STD_LOGIC_VECTOR (3 downto 0));
end Reg;

architecture Behavioral of Reg is

begin

    process (Clk) begin
        if (rising_edge(Clk)) then      --respond when clock rises
```

```
        if En = '1' or R = '1' then --Enable should be set
            if R = '1' then
                Q <= "0000";
            else
                Q <= D;
            end if;
        end if;
    end if;
  end process;

end Behavioral;
```

RTL SCHEMATICS



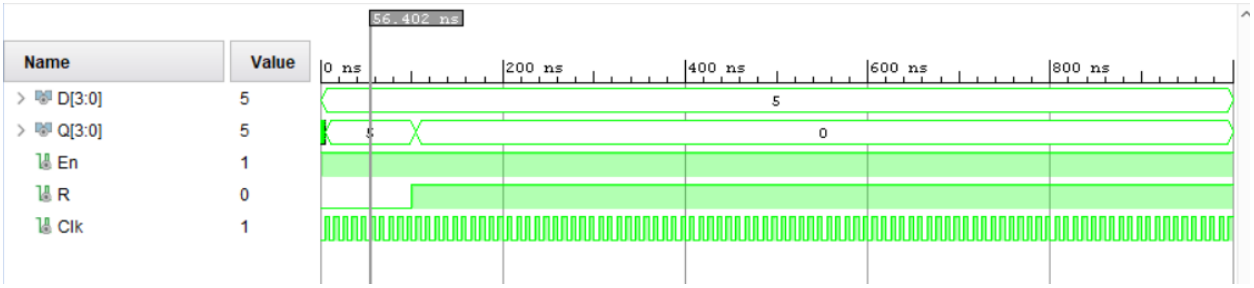
BEHAVIORAL SIMULATION CODE

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

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

architecture Behavioral of Sim_Register is

component Reg is
    Port ( D       : in STD_LOGIC_VECTOR (3 downto 0);
```

```
        En      : in STD_LOGIC;
        R       : in STD_LOGIC;
        Clk     : in STD_LOGIC;
        Q       : out STD_LOGIC_VECTOR (3 downto 0));
end component;

signal D, Q        : std_logic_vector(3 downto 0);
signal En, R, Clk   : std_logic;

signal clock_period : time := 10ns;

begin

UUT : Reg port map(
      D   => D,
      En  => En,
      R   => R,
      Clk => Clk,
      Q   => Q);

clock_process : process
   begin
      Clk <= '0';
      wait for clock_period / 2;

      Clk <= '1';
      wait for clock_period / 2;
end process;

sim : process
   begin
      D   <= "0101";
      En  <= '1';
      R   <= '0';
      wait for 100ns;

      R   <= '1';
```

```
    wait;
end process;


end Behavioral;
```

TIMING DIAGRAM



SLOW CLOCK

A slow clock was used to decrease the speed of the Basys3 internal clock process.

Since the frequency of the internal clock is 100MHz, the output of the Nano processor will not be visible to the naked eye.

Hence when generating the bit Stream, the frequency was reduced to 1Hz by setting the count to 50,000,000. However, to get the timing diagram under a clock period of 1000ns, the count was set to 2.

VHDL CODE

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


entity Slow_Clk is
    Port ( Clk_in   : in STD_LOGIC;
           Clk_out  : out STD_LOGIC);
end Slow_Clk;


architecture Behavioral of Slow_Clk is
```

```
signal count       : integer   := 1;
signal clk_status   : std_logic := '0';

begin
   --For 100MHz input clock, this generates 1Hz clock
   process (Clk_in) begin
      if (rising_edge(Clk_in)) then
         count <= count + 1;
         if (count = 2) then               --count 50M pulses(1/2 of period)
            clk_status <= not clk_status;   --Invert clock status
            Clk_out <= clk_status;
            count <= 1;                  --Reset counter
         end if;
      end if;
   end process;
end Behavioral;
```

NANO PROCESSOR

The overall processor accepts two inputs, which are the Reset push button and the Clock. It outputs the values of the Reg7, overflow, and zero flags.

VHDL CODE

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

entity NanoProcessor is
   Port ( Clk         : in STD_LOGIC;
          Reset       : in STD_LOGIC;
          Overflow    : out STD_LOGIC;
          Zero        : out STD_LOGIC;
          S_out       : out STD_LOGIC_VECTOR (3 downto 0);
          sevenSegOut  : out STD_LOGIC_VECTOR(6 downto 0));
end NanoProcessor;

architecture Behavioral of NanoProcessor is
```

```
component Slow_Clk is
    Port ( Clk_in : in STD_LOGIC;
           Clk_out : out STD_LOGIC);
end component;

signal slwClkOut : std_logic;

component Ins_Decoder is
    Port ( Instruction      : in STD_LOGIC_VECTOR (11 downto 0);
           JumpCheck        : in STD_LOGIC_VECTOR (3 downto 0);
           Reg_En           : out STD_LOGIC_VECTOR (2 downto 0);
           Load_Sel         : out STD_LOGIC;
           Imm_Val          : out STD_LOGIC_VECTOR (3 downto 0);
           Reg_Sel_A        : out STD_LOGIC_VECTOR (2 downto 0);
           Reg_Sel_B        : out STD_LOGIC_VECTOR (2 downto 0);
           AddSub_Sel       : out STD_LOGIC;
           Jump_Flag        : out STD_LOGIC;
           Jump_Add         : out STD_LOGIC_VECTOR (2 downto 0);
           Neg_Sel          : out STD_LOGIC);
end component;

signal ins                          : std_logic_vector(11 downto 0);
signal immVal                       : std_logic_vector(3 downto 0);
signal regEn, regSelA, regSelB, jmpAdd      : std_logic_vector(2 downto 0);
signal loadSel, addSubSel, jmpFlag, negSel  : std_logic;

component ProgramCounter is
    Port ( Clk      : in STD_LOGIC;
           Reset    : in STD_LOGIC;
           JumpFlag : in STD_LOGIC;
           JumpAdd  : in STD_LOGIC_VECTOR (2 downto 0);
           MemSel   : out STD_LOGIC_VECTOR (2 downto 0));
end component;

signal memSel : std_logic_vector(2 downto 0);

component Program_ROM is
```

```
    Port ( Mem_Sel      : in STD_LOGIC_VECTOR (2 downto 0);
           Instruction  : out STD_LOGIC_VECTOR (11 downto 0));
end component;

component RegisterBank is
    Port ( RegBank_in   : in STD_LOGIC_VECTOR (3 downto 0);
           RegEN        : in STD_LOGIC_VECTOR (2 downto 0);
           Reset        : in STD_LOGIC;
           Clk          : in STD_LOGIC;
           Reg0_out     : out STD_LOGIC_VECTOR (3 downto 0);
           Reg1_out     : out STD_LOGIC_VECTOR (3 downto 0);
           Reg2_out     : out STD_LOGIC_VECTOR (3 downto 0);
           Reg3_out     : out STD_LOGIC_VECTOR (3 downto 0);
           Reg4_out     : out STD_LOGIC_VECTOR (3 downto 0);
           Reg5_out     : out STD_LOGIC_VECTOR (3 downto 0);
           Reg6_out     : out STD_LOGIC_VECTOR (3 downto 0);
           Reg7_out     : out STD_LOGIC_VECTOR (3 downto 0));
end component;

signal regBankIn, r0, r1, r2, r3, r4, r5, r6, r7 : std_logic_vector(3 downto 0);

component Mux8_4bit is
    Port ( A0       : in STD_LOGIC_VECTOR (3 downto 0);
           A1       : in STD_LOGIC_VECTOR (3 downto 0);
           A2       : in STD_LOGIC_VECTOR (3 downto 0);
           A3       : in STD_LOGIC_VECTOR (3 downto 0);
           A4       : in STD_LOGIC_VECTOR (3 downto 0);
           A5       : in STD_LOGIC_VECTOR (3 downto 0);
           A6       : in STD_LOGIC_VECTOR (3 downto 0);
           A7       : in STD_LOGIC_VECTOR (3 downto 0);
           RegSel   : in STD_LOGIC_VECTOR (2 downto 0);
           Q        : out STD_LOGIC_VECTOR (3 downto 0));
end component;

signal muxOutA, muxOutB : std_logic_vector(3 downto 0);

component AddSubUnit is
```

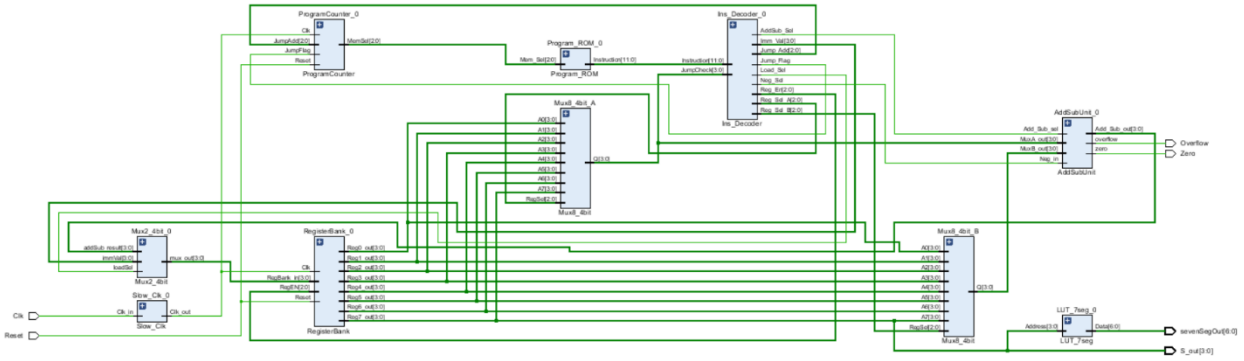
```
   Port ( MuxA_out     : in STD_LOGIC_VECTOR (3 downto 0);
          MuxB_out     : in STD_LOGIC_VECTOR (3 downto 0);
          Add_Sub_sel  : in STD_LOGIC;
          Neg_in       : in STD_LOGIC;
          Add_Sub_out  : out STD_LOGIC_VECTOR (3 downto 0);
          overflow     : out STD_LOGIC;
          zero         : out STD_LOGIC);
end component;

signal addSubOut       : std_logic_vector(3 downto 0);

component Mux2_4bit is
   Port ( immVal          : in STD_LOGIC_VECTOR (3 downto 0);
          addSub_result   : in STD_LOGIC_VECTOR (3 downto 0);
          loadSel         : in STD_LOGIC;
          mux_out         : out STD_LOGIC_VECTOR (3 downto 0));
end component;

component LUT_7seg is
   Port ( Address  : in STD_LOGIC_VECTOR (3 downto 0);
          Data     : out STD_LOGIC_VECTOR (6 downto 0));
end component;

begin

Slow_Clk_0 : Slow_Clk
   port map(
      Clk_in => Clk,         --connected from direct input
      Clk_out => slwClkOut);

RegisterBank_0 : RegisterBank
   port map(
      RegBank_in  => regBankIn,   --in
      RegEN       => regEn,       --in
      Reset       => Reset,
      Clk         => slwClkOut,   --in
      Reg0_out    => r0,
```

```
    Reg1_out    => r1,
    Reg2_out    => r2,
    Reg3_out    => r3,
    Reg4_out    => r4,
    Reg5_out    => r5,
    Reg6_out    => r6,
    Reg7_out    => r7);

Mux8_4bit_A : Mux8_4bit
  port map(
    A0 => r0,          --in
    A1 => r1,          --in
    A2 => r2,          --in
    A3 => r3,          --in
    A4 => r4,          --in
    A5 => r5,          --in
    A6 => r6,          --in
    A7 => r7,          --in
    RegSel => regSelA,  --in
    Q => muxOutA);

Mux8_4bit_B : Mux8_4bit
  port map(
    A0     => r0,        --in
    A1     => r1,        --in
    A2     => r2,        --in
    A3     => r3,        --in
    A4     => r4,        --in
    A5     => r5,        --in
    A6     => r6,        --in
    A7     => r7,        --in
    RegSel  => regSelB,    --in
    Q      => muxOutB);

AddSubUnit_0 : AddSubUnit
  port map(
    MuxA_out    => muxOutA,    --in
```

```
      MuxB_out    => muxOutB,     --in
      Add_Sub_sel => addSubSel,   --in
      Neg_in      => negSel,      --in
      Add_Sub_out => addSubOut,
      overflow    => Overflow,    --connected to direct output
      zero        => Zero);       --connected to direct output

Mux2_4bit_0 : Mux2_4bit
   port map(
      immVal => immVal,          --in
      addSub_result => addSubOut, --in
      loadSel => loadSel,        --in
      mux_out => regBankIn);

Ins_Decoder_0 : Ins_Decoder
   port map(
      Instruction => ins,     --in
      JumpCheck   => muxOutA,  --in
      Reg_En      => regEn,
      Load_Sel    => loadSel,
      Imm_Val     => immVal,
      Reg_Sel_A   => regSelA,
      Reg_Sel_B   => regSelB,
      AddSub_Sel  => addSubSel,
      Jump_Flag   => jmpFlag,
      Jump_Add    => jmpAdd,
      Neg_Sel     => negSel);

Program_ROM_0 : Program_ROM
   port map(
      Mem_Sel     => memSel,      --in
      Instruction => ins);

ProgramCounter_0 : ProgramCounter
   port map(
      Clk         => slwClkOut,   --in
      Reset       => Reset,       --connected from direct input
```

```
      JumpFlag    => jmpFlag,     --in
      JumpAdd     => jmpAdd,      --in
      MemSel      => memSel);

LUT_7seg_0 : LUT_7seg
   port map(
      Address => r7,
      Data    => sevenSegOut);

   S_out <= r7; --connected to direct output

end Behavioral;
```

RTL SCHEMATICS



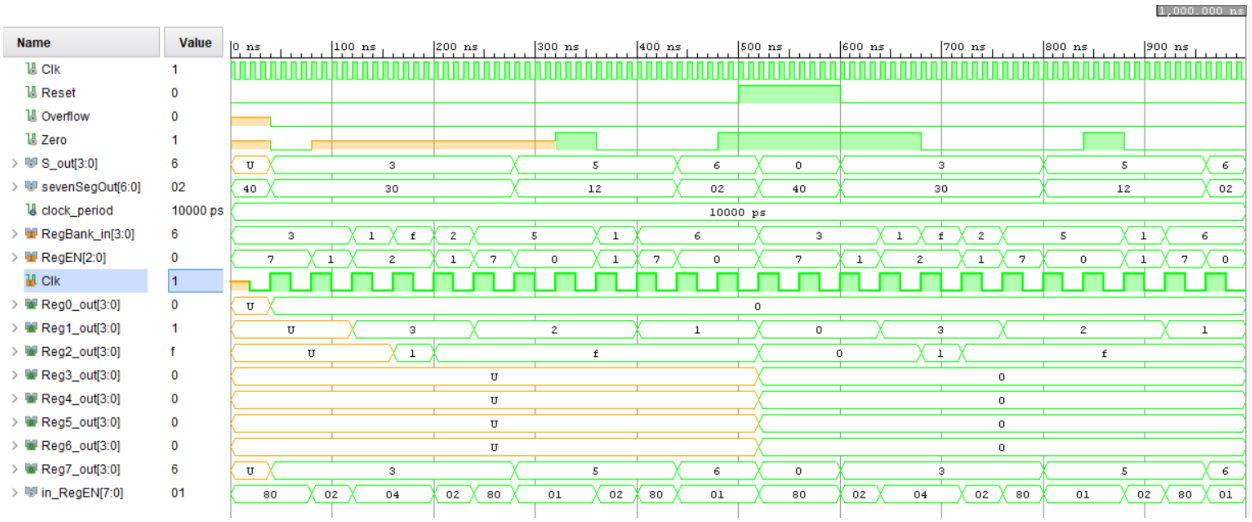
BEHAVIORAL SIMULATION CODE

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

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

architecture Behavioral of Sim_NanoProcessor is

component NanoProcessor is
   Port ( Clk         : in STD_LOGIC;
          Reset       : in STD_LOGIC;
```

```
        Overflow    : out STD_LOGIC;
        Zero        : out STD_LOGIC;
        S_out       : out STD_LOGIC_VECTOR (3 downto 0);
        sevenSegOut  : out STD_LOGIC_VECTOR(6 downto 0));
end component;

signal Clk, Reset, Overflow, Zero   : std_logic;
signal S_out                    : std_logic_vector(3 downto 0);
signal sevenSegOut              : std_logic_vector(6 downto 0);

constant clock_period           : time := 10ns;

begin

UUT : NanoProcessor port map(
      Clk       => Clk,
      Reset     => Reset,
      Overflow  => Overflow,
      Zero      => Zero,
      S_out     => S_out,
      sevenSegOut => sevenSegOut);

clock_process : process
   begin
      Clk <= '1';
      wait for clock_period / 2;

      Clk <= '0';
      wait for clock_period / 2;
end process;

sim : process
   begin
      Reset <= '0';
      wait for 500ns;
      Reset <= '1';
      wait for 100ns;
```

```
    Reset <= '0';
  wait;
end process;


end Behavioral;
```

TIMING DIAGRAM



XDC FILE

Overall nano processor has two inputs and four outputs that needs to be connected to the BASYS3 board.

Clock in the BASYS3 board is connected to the **Clk** input in the nano processor and is slowed down by the slow clock implemented inside.

Reset input is taken through a push button which is used to reset the registers in register bank and the program counter.

Since the final value of the program is stored in the **Reg7**, it is connected to a set of LEDs and to a 7-segment display.

Overflow and the **Zero** produced by add/subtract unit are connected to two LEDs.

```
## LEDs - Register 7
set_property PACKAGE_PIN U16 [get_ports {S_out[0]}]
   set_property IOSTANDARD LVCMOS33 [get_ports {S_out[0]}]
set_property PACKAGE_PIN E19 [get_ports {S_out[1]}]
```

```
   set_property IOSTANDARD LVCMOS33 [get_ports {S_out[1]}]
set_property PACKAGE_PIN U19 [get_ports {S_out[2]}]
   set_property IOSTANDARD LVCMOS33 [get_ports {S_out[2]}]
set_property PACKAGE_PIN V19 [get_ports {S_out[3]}]
   set_property IOSTANDARD LVCMOS33 [get_ports {S_out[3]}]

##Overflow
set_property PACKAGE_PIN P1 [get_ports {Overflow}]
   set_property IOSTANDARD LVCMOS33 [get_ports {Overflow}]
##Zero
set_property PACKAGE_PIN L1 [get_ports {Zero}]
   set_property IOSTANDARD LVCMOS33 [get_ports {Zero}]

##Push Button for Reset
set_property PACKAGE_PIN U18 [get_ports Reset]
   set_property IOSTANDARD LVCMOS33 [get_ports Reset]

##7 segment display for Register 7
set_property PACKAGE_PIN W7 [get_ports {sevenSegOut[0]}]
   set_property IOSTANDARD LVCMOS33 [get_ports {sevenSegOut[0]}]
set_property PACKAGE_PIN W6 [get_ports {sevenSegOut[1]}]
   set_property IOSTANDARD LVCMOS33 [get_ports {sevenSegOut[1]}]
set_property PACKAGE_PIN U8 [get_ports {sevenSegOut[2]}]
   set_property IOSTANDARD LVCMOS33 [get_ports {sevenSegOut[2]}]
set_property PACKAGE_PIN V8 [get_ports {sevenSegOut[3]}]
   set_property IOSTANDARD LVCMOS33 [get_ports {sevenSegOut[3]}]
set_property PACKAGE_PIN U5 [get_ports {sevenSegOut[4]}]
   set_property IOSTANDARD LVCMOS33 [get_ports {sevenSegOut[4]}]
set_property PACKAGE_PIN V5 [get_ports {sevenSegOut[5]}]
   set_property IOSTANDARD LVCMOS33 [get_ports {sevenSegOut[5]}]
set_property PACKAGE_PIN U7 [get_ports {sevenSegOut[6]}]
   set_property IOSTANDARD LVCMOS33 [get_ports {sevenSegOut[6]}]

## Clock signal
set_property PACKAGE_PIN W5 [get_ports Clk]
   set_property IOSTANDARD LVCMOS33 [get_ports Clk]
   create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports Clk]
```

CONCLUSION

After completing the lab, we were able to develop a nano processor that can do four simple operations.

Using pre-developed components in previous labs made it easier for us to combine them to work together in the nano processor.

All the different components were checked using simulations and finally, the nano processor was simulated to execute an assembly program with eight instructions.

This team project helped us to enhance skills such as communication, coordination, distributing responsibilities, and integrating components developed by separate team members as well.