CS2052 Computer Architecture

Department of Computer Science and Engineering, University of Moratuwa

Lab 9-10 – Nanoprocessor Design Competition

Group Name : Nanocache

Group Members	Index Number
R . A . N Sankalana	190564N
M . M Poorna S. Cooray	190110V
H .D .S Vidulanka	190646T

Prepared for : CS2052

Date of performance : 17 / 03 / 2021

Date of submission : 12 / 04 / 2021

Contents

Lab Tasks 3

Assembly Program 3

VHDL Codes for the NanoProcessor 4

1. 4-bit Add/Subtract Unit 4
2. 3-bit Adder 6
3. 3-bit Program Counter 8
4. K-way b-bit Multiplexer 9
5. Register Bank 18
6. Program ROM 21
7. Instruction Decoder 22
8. NanoProcessor 24

Timing Diagrams 30

1. 4-bit Add/Subtract Unit 30
2. 3-bit Adder 32
3. 3-bit Program Counter 34
4. K-way b-bit Multiplexer 36
5. Register Bank 38
6. Program ROM 41
7. Instruction Decoder 42
8. NanoProcessor 44

Conclusion 46

Contribution of Members 46

Lab Tasks

The main task of this lab is to design a 4-bit processor which is capable of executing a set of predefined instructions. The arithmetic unit of the processor only focuses on addition and subtraction. The processor comprises of the following key components.

- 4-bit Add/Subtract unit
- 3-bit adder
- 3-bit Program Counter (PC)
- k-way b-bit multiplexers
  - 2-way 3-bit multiplexer
  - 2-way 4-bit multiplexer
  - 8-way 4-bit multiplexer
- Register Bank
- Program ROM
- Instruction Decoder

The whole project was completed as a team where lab tasks were evenly divided among the team members.

Assembly Program

This assembly program was written to calculate the sum of all integers between 1 and 3. According to the lab report output was mapped to the Register 7 in the register bank. And the final value of sum was saved in that register. In this code we create a loop from instruction 4 to7, to decrement numbers from 3 to 1 and add to the R7 register. In instruction 7, always check if R0 for jump. But it is by default “0000” then always it comes to the instruction 7 it returns to the instruction 4. Then we create a terminating condition for that loop in instruction 6. It checks the value of R1 and if it is zero terminate the program.

Assembly Code :

```
MOVI  R7, 0        ; R7 ← 0
MOVI  R1, 3        ; R1 ← 3
MOVI  R2, 1        ; R2 ← 1
NEG   R2           ; R2 ← Neg R2
ADD   R7, R1       ; R7 ← R7 + R1
ADD   R1, R2       ; R1 ← R1 + R2
JZR   R1, 6        ; If R1 0 jump to 6
JZR   R0, 4        ; If  R0 jump to 4
```

Machine Code :

```
Instruction  0        "101110000000",
Instruction  1        "100010000011",
Instruction  2        "100100000001",
Instruction  3        "010100000000",
Instruction  4        "001110010000",
Instruction  5        "000010100000",
Instruction  6        "110010000110",
Instruction  7        "110000000100"
```

VHDL Codes for the NanoProcessor

1. 4-bit Add/Subtract Unit

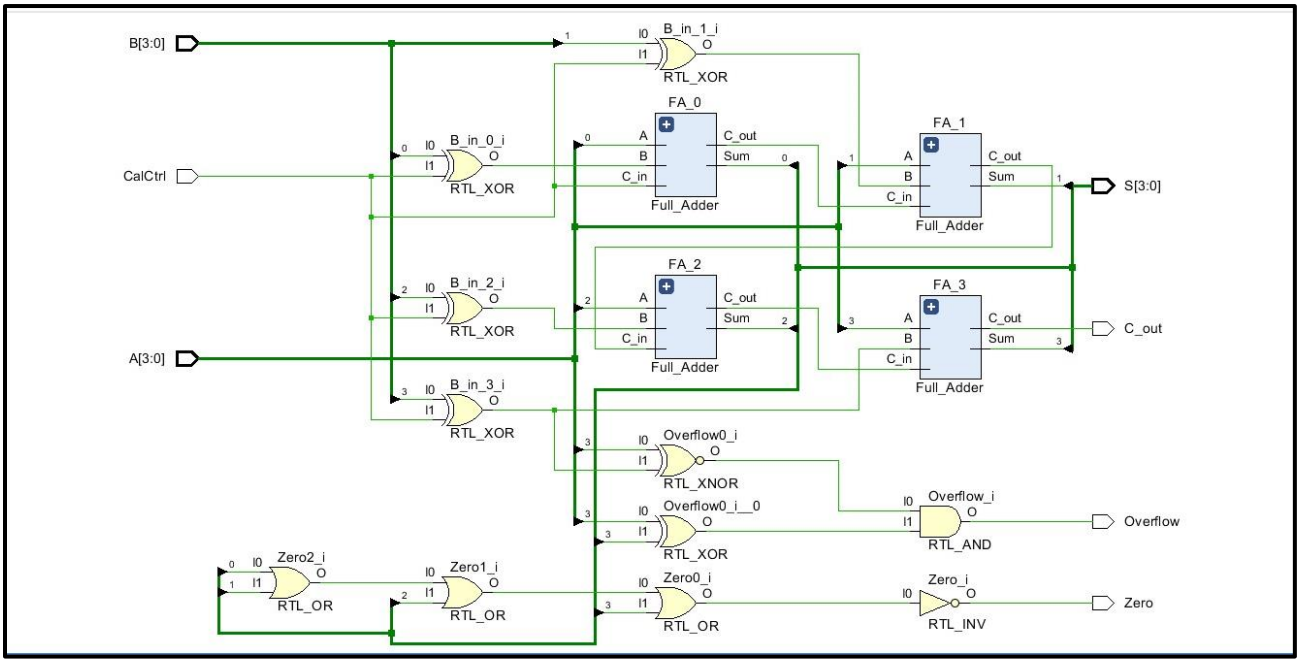
4-bit adder-subtractor can add and subtract 4-bit binary numbers. A: A0, A1, A2, A3 and B: B0, B1, B2, B3 and Calctrl are the inputs of the adder subtractor. The circuit consists of 4 full adders to perform operations of 4-bit numbers. Calctrl holds the binary value which determines the operation is addition or subtraction.

VHDL code for 4-bit add/sub unit.

```
entity Adder_Subtractor is
Port (  A : in STD_LOGIC_VECTOR (3 downto 0);
 B : in STD_LOGIC_VECTOR (3 downto 0);
 CalCtrl : in STD_LOGIC;
 C_out : out STD_LOGIC;
 Overflow : out STD_LOGIC;
 Zero : out STD_LOGIC;
 S : out STD_LOGIC_VECTOR (3 downto 0));
end Adder_Subtractor;

architecture Behavioral of Adder_Subtractor is
component Full_Adder is
port (   A: in std_logic;
         B: in std_logic;
         C_in: in std_logic;
         Sum: out std_logic;
         C_out: out std_logic);
end component;
```

```
SIGNAL B_in, S_out : std_logic_vector(3 downto 0);
SIGNAL FA0_S, FA0_C, FA1_S, FA1_C, FA2_S, FA2_C, FA3_S, FA3_C : std_logic;

begin
        B_in(0) <= B(0) XOR CalCtrl;
        B_in(1) <= B(1) XOR CalCtrl;
        B_in(2) <= B(2) XOR CalCtrl;
        B_in(3) <= B(3) XOR CalCtrl;

FA_0 : Full_Adder
        port map ( A => A(0),
                   B => B_in(0),
                   C_in => CalCtrl,
                   Sum => S_out(0),
                   C_Out => FA0_C);

FA_1 : Full_Adder
        port map (A => A(1),
                  B => B_in(1),
                  C_in => FA0_C,
                  Sum => S_out(1),
                  C_Out => FA1_C);

FA_2 : Full_Adder
        port map (A => A(2),
                  B => B_in(2),
                  C_in => FA1_C,
                  Sum => S_out(2),
                  C_Out => FA2_C);

FA_3 : Full_Adder
        port map (A => A(3),
                  B => B_in(3),
                  C_in => FA2_C,
                  Sum => S_out(3),
                  C_Out => C_out);

S(0) <= S_out(0);
S(1) <= S_out(1);
S(2) <= S_out(2);
S(3) <= S_out(3);
Overflow <= (A(3) XNOR (CalCtrl XOR B(3))) AND (A(3) XOR S_out(3));
Zero <= not (S_out(0) or S_out(1) or S_out(2) or S_out(3));

end Behavioral;
```

RTL Schematic Diagram for add/sub unit



2. 3-bit Adder

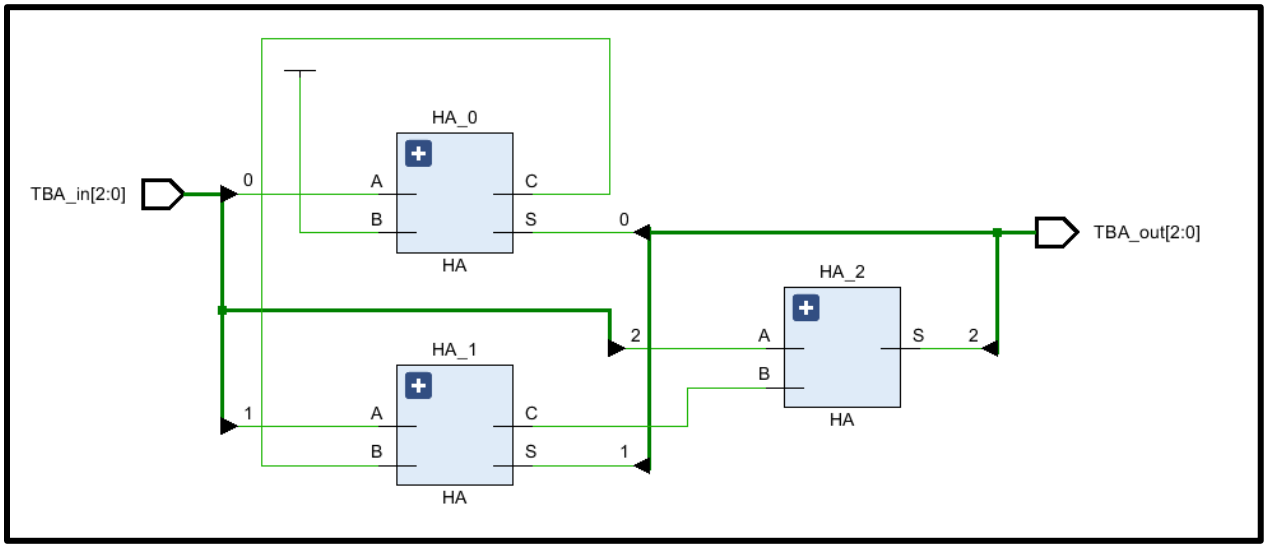
3-bit adder, with an input and an output, is used to increment the program counter. The component is implemented using the half adder which was designed in Lab 3.

Code for 3 – bit Adder

```
entity Adder_3_bit is
    Port ( TBA_in : in STD_LOGIC_VECTOR (2 downto 0);
           TBA_out : out STD_LOGIC_VECTOR (2 downto 0));
end Adder_3_bit;

architecture Behavioral of Adder_3_bit is


component HA
    Port ( A : in STD_LOGIC;
           B : in STD_LOGIC;
           S : out STD_LOGIC;
           C : out STD_LOGIC);
end component;
SIGNAL C_0 , C_1: std_logic;
```

```
begin
--adds one to the input
HA_0: HA
   Port map
      ( A => TBA_in(0),
        B =>'1',
        S => TBA_out(0),
        C => C_0);
HA_1: HA
    Port map
       ( A => TBA_in(1),
         B =>C_0,
         S => TBA_out(1),
         C => C_1);
HA_2: HA
     Port map
       ( A => TBA_in(2),
         B => C_1,
         S => TBA_out(2));

end Behavioral;
```

RTL Schematic Diagram of 3 – bit Adder



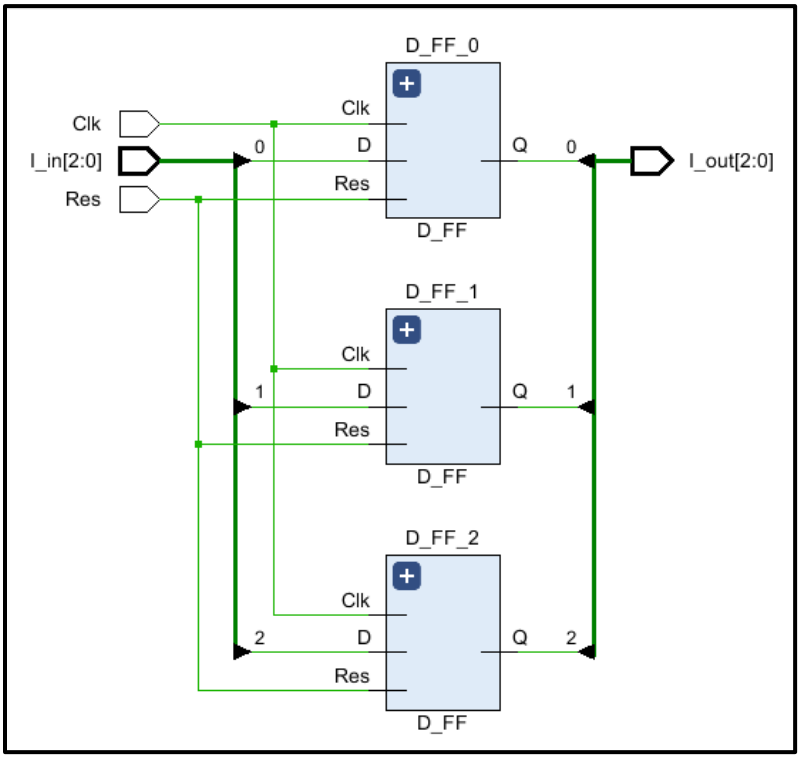
3. 3-bit Program Counter

Program counter keeps track of the instruction address that is being executing at the current moment by the processor. The counter can be set to 0 by the reset button. Hence D flip flops from the Lab 5 are used here.

Code for Program Counter

```
entity ProgramCounter is
    Port ( I_in : in STD_LOGIC_VECTOR (2 downto 0);
           Clk : in STD_LOGIC;
           Res : in STD_LOGIC;
           I_out : out STD_LOGIC_VECTOR (2 downto 0));
end ProgramCounter;
architecture Behavioral of ProgramCounter is
component D_FF
    Port ( D : in STD_LOGIC;
           Res : in STD_LOGIC;
           Clk : in STD_LOGIC;
           Q : out STD_LOGIC;
           Qbar : out STD_LOGIC);
end component;

begin
D_FF_0: D_FF
    port map(
        D => I_in(0),
        Res=>Res,
        Clk=>Clk,
        Q=>I_out(0));

D_FF_1: D_FF
    port map(
        D => I_in(1),
        Res=>Res,
        Clk=>Clk,
        Q=>I_out(1));

D_FF_2: D_FF
    port map(
        D => I_in(2),
        Res=>Res,
        Clk=>Clk,
        Q=>I_out(2));

end Behavioral;
```

RTL Schematic Diagram of 3-bit Program Counter



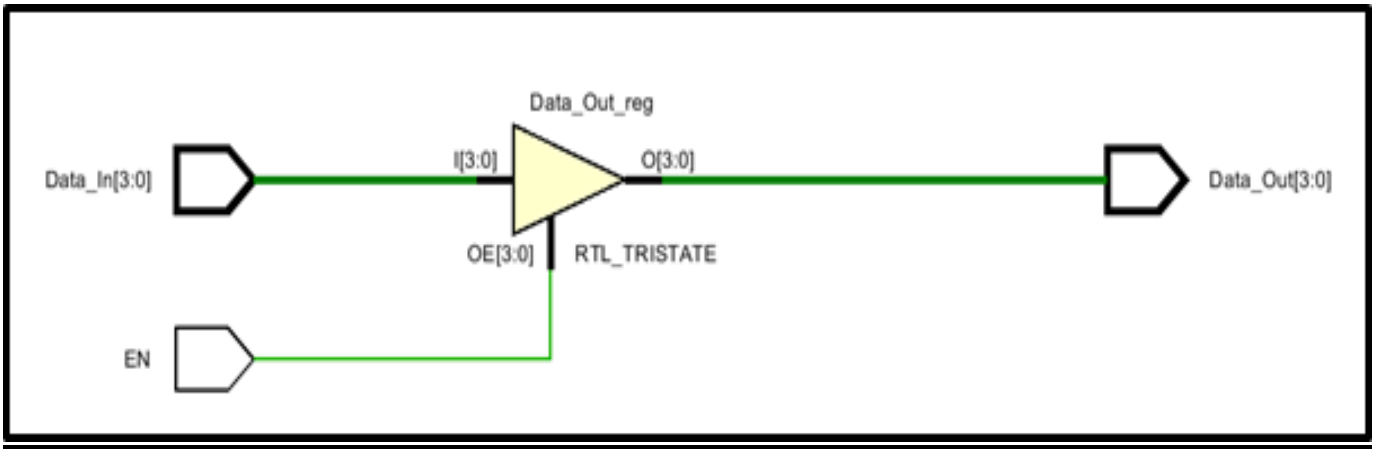
4. K-way b-bit Multiplexer

- There is main two ways to make a k-way b-bit multiplexers.
- In this lab we used Try state buffers to implement k-way b-bit multiplexers.
- To create that multiplexers only we need a decoder and “k” number of try state buffers.
- To make Nano processor we had to create 2-way 3-bit Multiplexer, 2-way 4-bitMultiplexer and 8-way 4-bit multiplexers.
- 2-way 3=bit mux used to select the jump address bus when jump flag raise otherwise it select the adder 3-bit output path.
- 2-way 3-bit mux used to select the Immediate Load when the Load select is high otherwise it selects the result bus of ALU.
- 8-way 4-bit mux is used to select the correct register when we input the address of it.

Code for Try State Buffer

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


entity Try_State_Buffer_4_bit is
    Port ( Data_In : in STD_LOGIC_VECTOR (3 downto 0);
           EN : in STD_LOGIC;
           Data_Out : out STD_LOGIC_VECTOR (3 downto 0));
end Try_State_Buffer_4_bit;

architecture Behavioral of Try_State_Buffer_4_bit is

begin
       Data_Out <= Data_In when (EN = '1') else "ZZZZ";
end Behavioral;
```

RTL Schematic Diagram of Try State Buffer



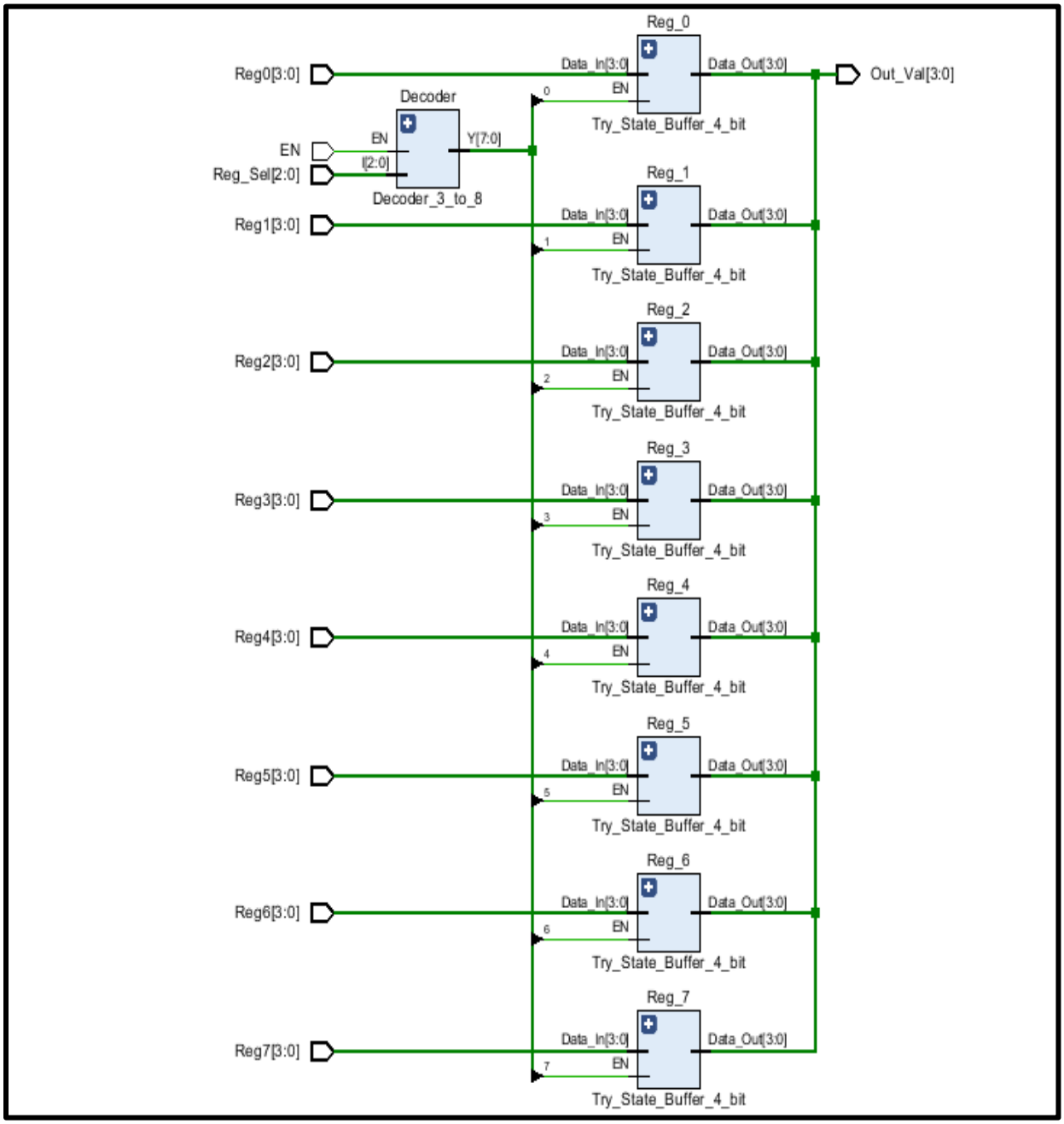
- ❖ The implementation of all k-way b-bit multiplexers are same . the VHDL codes of all multiplexers are below.

Code for 8-way 4-bit Multiplexer

```
entity Mux_8_way_4_bit is
    Port ( Reg0 : in STD_LOGIC_VECTOR (3 downto 0);
           Reg1 : in STD_LOGIC_VECTOR (3 downto 0);
           Reg2 : in STD_LOGIC_VECTOR (3 downto 0);
           Reg3 : in STD_LOGIC_VECTOR (3 downto 0);
           Reg4 : in STD_LOGIC_VECTOR (3 downto 0);
           Reg5 : in STD_LOGIC_VECTOR (3 downto 0);
           Reg6 : in STD_LOGIC_VECTOR (3 downto 0);
           Reg7 : in STD_LOGIC_VECTOR (3 downto 0);
           EN : in STD_LOGIC;
           Reg_Sel : in STD_LOGIC_VECTOR (2 downto 0);
           Out_Val : out STD_LOGIC_VECTOR (3 downto 0));
end Mux_8_way_4_bit;
architecture Behavioral of Mux_8_way_4_bit is
--add a decoder to enable registers separately
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;
--add a try state buffer to throw the correct input to common bus after enabling correct try
state buffer
component Try_State_Buffer_4_bit
    Port ( Data_In : in STD_LOGIC_VECTOR (3 downto 0);
           EN : in STD_LOGIC;
           Data_Out : out STD_LOGIC_VECTOR (3 downto 0));
end component;
    Signal Common_Data_Bus : STD_LOGIC_VECTOR (3 downto 0);
    signal RegSel :STD_LOGIC_VECTOR (7 downto 0);

begin
    --decode the input register and select correct try state buffer
    Decoder: Decoder_3_to_8
        port map (
            I => Reg_Sel,
            EN => EN,
            Y => RegSel );

    Reg_0: Try_State_Buffer_4_bit
        port map(
            Data_In => Reg0,
            EN => RegSel(0),
```

```
        Data_Out => Common_Data_Bus );
    Reg_1: Try_State_Buffer_4_bit
        port map(
            Data_In => Reg1,
            EN => RegSel(1),
            Data_Out => Common_Data_Bus );

    Reg_2: Try_State_Buffer_4_bit
        port map(
            Data_In => Reg2,
            EN => RegSel(2),
            Data_Out => Common_Data_Bus );

    Reg_3: Try_State_Buffer_4_bit
        port map(
            Data_In => Reg3,
            EN => RegSel(3),
            Data_Out => Common_Data_Bus );

    Reg_4: Try_State_Buffer_4_bit
        port map(
            Data_In => Reg4,
            EN => RegSel(4),
            Data_Out => Common_Data_Bus );

    Reg_5: Try_State_Buffer_4_bit
        port map(
            Data_In => Reg5,
            EN => RegSel(5),
            Data_Out => Common_Data_Bus );

    Reg_6: Try_State_Buffer_4_bit
        port map(
            Data_In => Reg6,
            EN => RegSel(6),
            Data_Out => Common_Data_Bus );

    Reg_7: Try_State_Buffer_4_bit
        port map(
            Data_In => Reg7,
            EN => RegSel(7),
            Data_Out => Common_Data_Bus );
--output common bus
    Out_Val <= Common_Data_Bus;

end Behavioral;
```

RTL Schematic Diagram of 8-way 4-bit Multiplexer

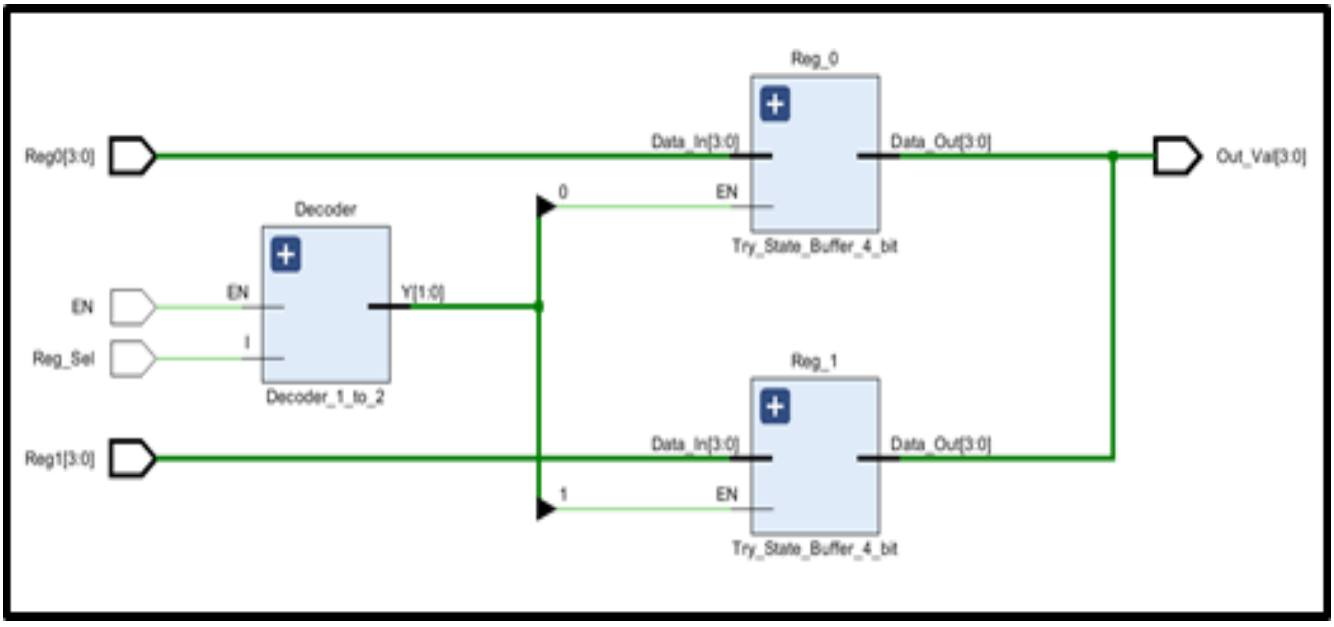


Code for 2-way 4-bit Multiplexer

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

entity Mux_2_way_4_bit is
    Port ( Reg0 : in STD_LOGIC_VECTOR (3 downto 0);
           Reg1 : in STD_LOGIC_VECTOR (3 downto 0);
           EN : in STD_LOGIC;
           Reg_Sel : in STD_LOGIC ;
           Out_Val : out STD_LOGIC_VECTOR (3 downto 0));
end Mux_2_way_4_bit;

architecture Behavioral of Mux_2_way_4_bit is
--add a decorder to enable registers separately
component Decoder_1_to_2
    Port ( I : in STD_LOGIC;
           EN : in STD_LOGIC;
           Y : out STD_LOGIC_VECTOR(1 downto 0));
end component;

--add a try state buffer to trow the correct input to common bus after enable
correct try state buffer
component Try_State_Buffer_4_bit
    Port ( Data_In : in STD_LOGIC_VECTOR (3 downto 0);
           EN : in STD_LOGIC;
           Data_Out : out STD_LOGIC_VECTOR (3 downto 0));
end component;

    Signal Common_Data_Bus : STD_LOGIC_VECTOR (3 downto 0);
    signal RegSel : STD_LOGIC_VECTOR (1 downto 0);

begin
    --decode the input register and select correct try state buffer
    Decoder: Decoder_1_to_2
        port map (
            I => Reg_Sel,
            EN => EN,
```

```
        Y => RegSel);
   Reg_0: Try_State_Buffer_4_bit
      port map(
         Data_In => Reg0,
         EN => RegSel(0),
         Data_Out => Common_Data_Bus );

   Reg_1: Try_State_Buffer_4_bit
      port map(
         Data_In => Reg1,
         EN => RegSel(1),
         Data_Out => Common_Data_Bus );

--output common bus
   Out_Val <= Common_Data_Bus;

end Behavioral;
```

RTL Schematic Diagram of 2-way 4-bit Multiplexer

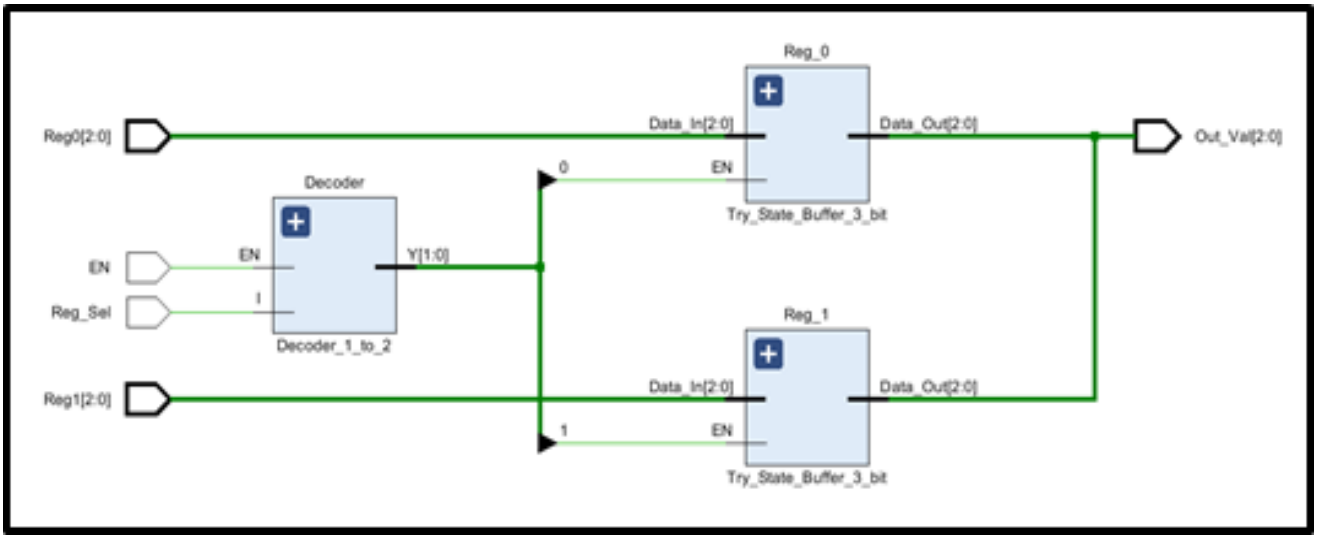


Code for 2-way 3-bit Multiplexer

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

entity Mux_2_way_3_bit is
    Port ( Reg0 : in STD_LOGIC_VECTOR (2 downto 0);
           Reg1 : in STD_LOGIC_VECTOR (2 downto 0);
           EN : in STD_LOGIC;
           Reg_Sel : in STD_LOGIC ;
           Out_Val : out STD_LOGIC_VECTOR (2 downto 0));
end Mux_2_way_3_bit;

architecture Behavioral of Mux_2_way_3_bit is

--add a decorder to enable registers separately
component Decoder_1_to_2
    Port ( I : in STD_LOGIC;
           EN : in STD_LOGIC;
           Y : out STD_LOGIC_VECTOR(1 downto 0));
end component;

--add a try state buffer to trow the correct input to common bus after enable correct try state
buffer
component Try_State_Buffer_3_bit
    Port ( Data_In : in STD_LOGIC_VECTOR (2 downto 0);
           EN : in STD_LOGIC;
           Data_Out : out STD_LOGIC_VECTOR (2 downto 0));
end component;

    Signal Common_Data_Bus : STD_LOGIC_VECTOR (2 downto 0);
    signal RegSel :STD_LOGIC_VECTOR (1 downto 0);

begin
    --decode the input register and select correct try state buffer
    Decoder: Decoder_1_to_2
        port map (
            I => Reg_Sel,
            EN => EN,
            Y => RegSel );

    Reg_0: Try_State_Buffer_3_bit
        port map(
            Data_In => Reg0,
            EN => RegSel(0),
            Data_Out => Common_Data_Bus );
```

```
    Reg_1: Try_State_Buffer_3_bit
       port map(
          Data_In => Reg1,
          EN => RegSel(1),
          Data_Out => Common_Data_Bus );

--output common bus
    Out_Val <= Common_Data_Bus;

end Behavioral;
```

RTL Schematic Diagram of 2-way 4-bit Multiplexer



5. Register Bank

8 Registers are used to create register bank and 3 to 8 decoder used to separate register addresses.

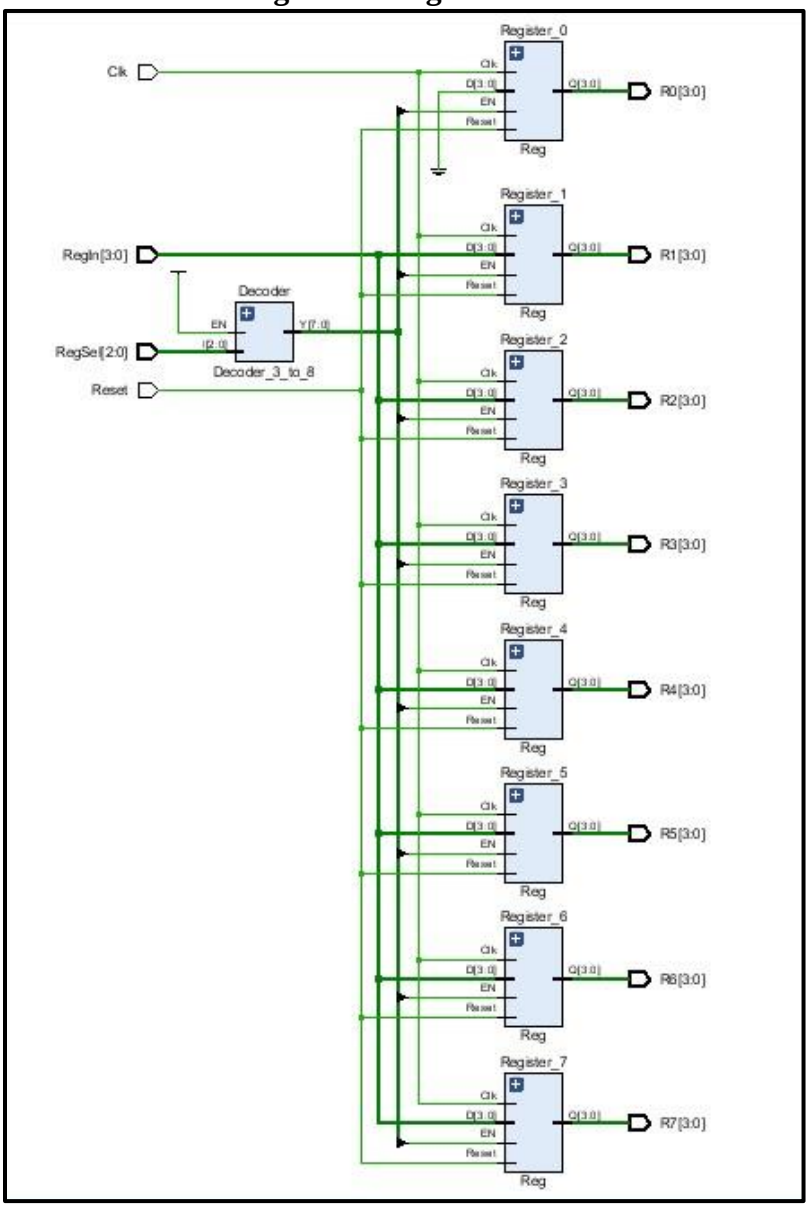
VHDL Code for Register Bank

```
entity Register_Bank is
Port ( RegIn : in STD_LOGIC_VECTOR (3 downto 0);
          Clk : in STD_LOGIC;
          Reset : in STD_LOGIC;
          RegSel : in STD_LOGIC_VECTOR (2 downto 0);
          R0 : out STD_LOGIC_VECTOR (3 downto 0);
          R1 : out STD_LOGIC_VECTOR (3 downto 0);
          R2 : out STD_LOGIC_VECTOR (3 downto 0);
          R3 : out STD_LOGIC_VECTOR (3 downto 0);
          R4 : out STD_LOGIC_VECTOR (3 downto 0);
          R5 : out STD_LOGIC_VECTOR (3 downto 0);
          R6 : out STD_LOGIC_VECTOR (3 downto 0);
          R7 : out STD_LOGIC_VECTOR (3 downto 0));
end Register_Bank;

architecture Behavioral of Register_Bank is

component Reg
   Port ( D : in STD_LOGIC_VECTOR (3 downto 0);
          EN : in STD_LOGIC;
          Reset: in std_logic;
          Clk : in STD_LOGIC;
          Q : out STD_LOGIC_VECTOR (3 downto 0));
end component;

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 EnableReg : std_logic_vector(7 downto 0);


begin
   Decoder: Decoder_3_to_8
      port map( I => RegSel,
                EN => '1',
                Y => EnableReg);
```

```
Register_0:Reg
   port map( D => "0000",
             EN => EnableReg(0),
             Reset => Reset,
             Clk => Clk,
             Q => R0);

Register_1:Reg
   port map( D => RegIn,
             EN => EnableReg(1),
             Reset => Reset,
             Clk => Clk,
             Q => R1);

Register_2:Reg
   port map( D => RegIn,
             EN => EnableReg(2),
             Reset => Reset,
             Clk => Clk,
             Q => R2);

Register_3:Reg
   port map( D => RegIn,
             EN => EnableReg(3),
             Reset => Reset,
             Clk => Clk,
             Q => R3);

Register_4:Reg
   port map( D => RegIn,
             EN => EnableReg(4),
             Reset => Reset,
             Clk => Clk,
             Q => R4);

Register_5:Reg
   port map( D => RegIn,
             EN => EnableReg(5),
             Reset => Reset,
             Clk => Clk,
             Q => R5);

Register_6:Reg
   port map( D => RegIn,
             EN => EnableReg(6),
             Reset => Reset,
```

```
            Clk => Clk,
            Q => R6);

   Register_7:Reg
      port map( D => RegIn,
            EN => EnableReg(7),
            Reset => Reset,
            Clk => Clk,
            Q => R7);

end Behavioral;
```

RTL Schematic Diagram of Register Bank



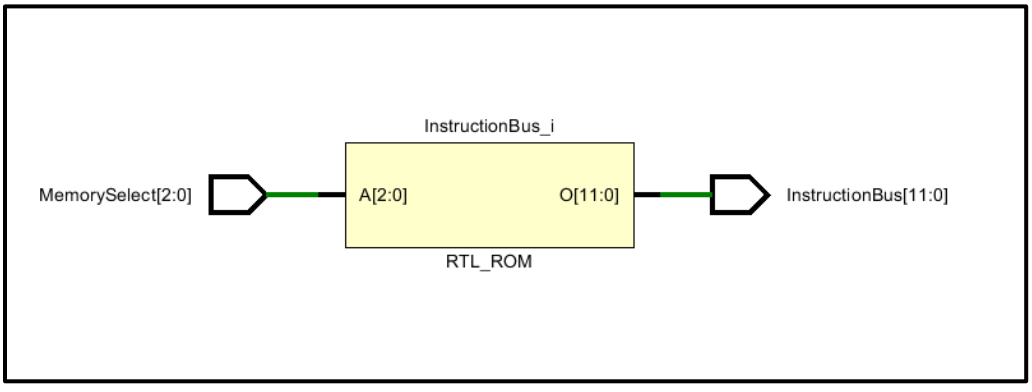
6. Program ROM

A set of predefined instructions are going to be executed in the nano processor. The task is to obtain the sum of 1-3 umbers. The instructions are written in assembly language and they are stored in the ROM.

Code for Program Rom

```
entity Prog_ROM is
 Port ( MemorySelect : in STD_LOGIC_VECTOR (2 downto 0);
        InstructionBus : out STD_LOGIC_VECTOR (11 downto 0));
end Prog_ROM;

architecture Behavioral of Prog_ROM is
type rom_type is array (0 to 7) of std_logic_vector(11 downto 0);

--a lookup table is used to store instructions
 signal ROM_bank : rom_type := (
  "101110000000", -- R7 <- 0
  "100010000011", -- R1 <- 3
  "100100000001", -- R2 <- 1
  "010100000000", --R2 <- Neg R2
  "001110010000", --R7 <- R7 + R1
  "000010100000", -- R1 <- R1 + R2
  "110010000110", -- If R1 0 jump to 6
  "110000000100"  --If  R0 jump to 4

 );
begin
   InstructionBus <= ROM_bank(to_integer(unsigned(MemorySelect)));

end Behavioral;
```

RTL Schematic Diagram of Program ROM



7. Instruction Decoder

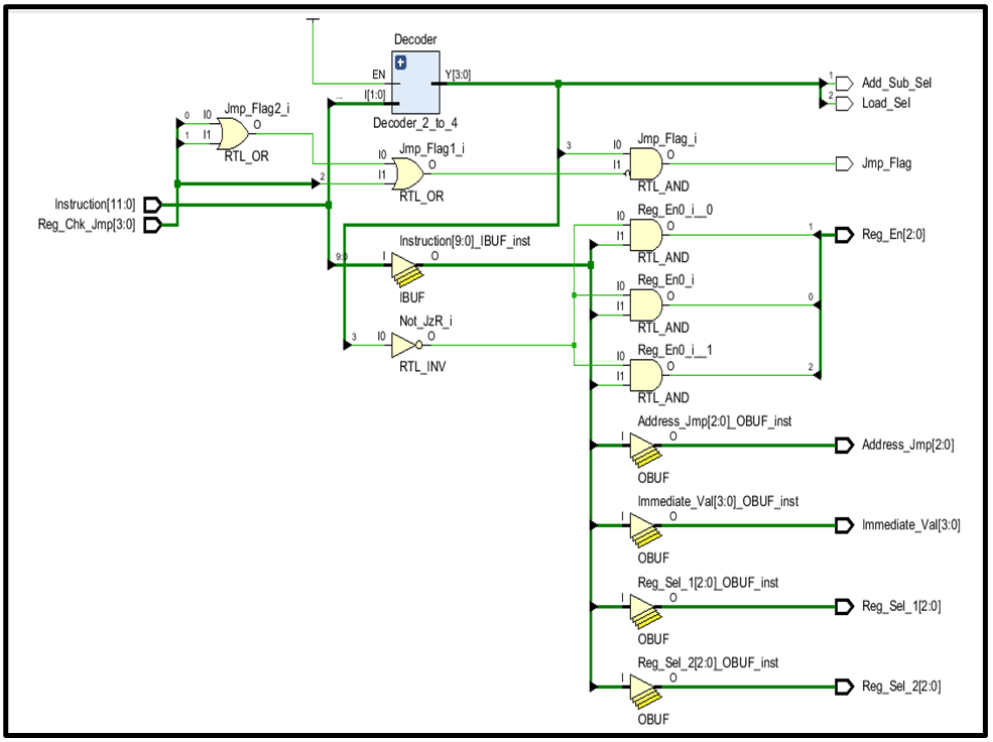
Instruction decoder is the instruction fetching unit in that processor. In this unit mainly do following things. decoding the instruction, sending instruction signals to the relevant components, and putting data into the data busses. After decoding the instruction first that unit select the operation from the op code using a 2 to 4 decoder. Then relative to the ADD, NEG, MOVI, JZR operations, it gives the instructions to components and add data into busses

Code for Instruction Decoder

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

entity Instruction_Decoder is
    Port ( Instruction : in STD_LOGIC_VECTOR (11 downto 0);
           Reg_Chk_Jmp : in STD_LOGIC_VECTOR (3 downto 0);
           Load_Sel : out STD_LOGIC;
           Add_Sub_Sel : out STD_LOGIC;
           Jmp_Flag : out STD_LOGIC;
           Reg_En : out STD_LOGIC_VECTOR (2 downto 0);
           Immediate_Val : out STD_LOGIC_VECTOR (3 downto 0);
           Reg_Sel_1 : out STD_LOGIC_VECTOR (2 downto 0);
           Reg_Sel_2 : out STD_LOGIC_VECTOR (2 downto 0);
           Address_Jmp : out STD_LOGIC_VECTOR (2 downto 0));
end Instruction_Decoder;

architecture Behavioral of Instruction_Decoder is
--add a decorder to select the correct opcode
component Decoder_2_to_4
    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,JzR,Not_JzR: std_logic;
begin
    --decode the opcode
    Decoder : Decoder_2_to_4
        port map(
                I => Instruction(11 downto 10),
                EN => '1',
                Y(0) => Add,  --if opcode is 00 then add will be 1
                Y(1) => Neg,  --if opcode is 01 then Neg will be 1
                Y(2) => Mov,  --if opcode is 10 then Mov will be 1
                Y(3) => JzR); --if opcode is 11 then JzR will be 1
```

```
    Load_Sel <= Mov;    --change the load to Immidiate value if and only if mov is 1
    Add_sub_Sel <= Neg; --Substract when the op code is 01

    --If op code is 11 (when jump instruction comes) regster enable will bw 0 for all regsters
    Not_JzR <= not JzR;
    Reg_EN(0) <= Not_JzR and Instruction(7);
    Reg_EN(1) <= Not_JzR and Instruction(8);
    Reg_EN(2) <= Not_JzR and Instruction(9);

    Immediate_Val <= Instruction(3 downto 0);   --set the immidiate value

    Reg_sel_1 <= Instruction(9 downto 7);   --set the Number 1
    Reg_sel_2 <= Instruction(6 downto 4);   --Set the Number 2

    -- create the jump flag using nputs of register for jump
    Jmp_Flag <= JzR and (not ( Reg_Chk_Jmp(0) or Reg_Chk_Jmp(1) or
Reg_Chk_Jmp(2)));
    -- Set the jump address if the instruction is a jump instruction
    Address_Jmp <= Instruction(2 downto 0);

end Behavioral;
```

RTL Schematic Diagram of Instruction Decoder



8. NanoProcessor

Code for NanoProcessor

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

entity NanoProcessor is
    Port ( Reset : in STD_LOGIC;
           Clk : in STD_LOGIC;
           Zero_Flag : out STD_LOGIC;
           Overflow_Flag : out STD_LOGIC;
           LED : out STD_LOGIC_VECTOR (3 downto 0);
           Carry_Out : out STD_LOGIC;
           SD_7 : out STD_LOGIC_VECTOR (6 downto 0));
end NanoProcessor;

architecture Behavioral of NanoProcessor is
component Slow_Clk
    port( Clk_in : in STD_LOGIC;
          Clk_out : out STD_LOGIC);
end component;

component ProgramCounter
    port( I_in : in STD_LOGIC_VECTOR (2 downto 0);
          Clk : in STD_LOGIC;
          Res : in STD_LOGIC;
          I_out : out STD_LOGIC_VECTOR (2 downto 0));
end component;

component Adder_3_bit
    Port ( TBA_in : in STD_LOGIC_VECTOR (2 downto 0);
           TBA_out : out STD_LOGIC_VECTOR (2 downto 0));
 end component;

component Mux_2_way_3_bit
     Port ( Reg0 : in STD_LOGIC_VECTOR (2 downto 0);
            Reg1 : in STD_LOGIC_VECTOR (2 downto 0);
            EN : in STD_LOGIC;
            Reg_Sel : in STD_LOGIC ;
            Out_Val : out STD_LOGIC_VECTOR (2 downto 0));
end component;

component Prog_ROM
    Port ( MemorySelect : in STD_LOGIC_VECTOR (2 downto 0);
           InstructionBus : out STD_LOGIC_VECTOR (11 downto 0));
```

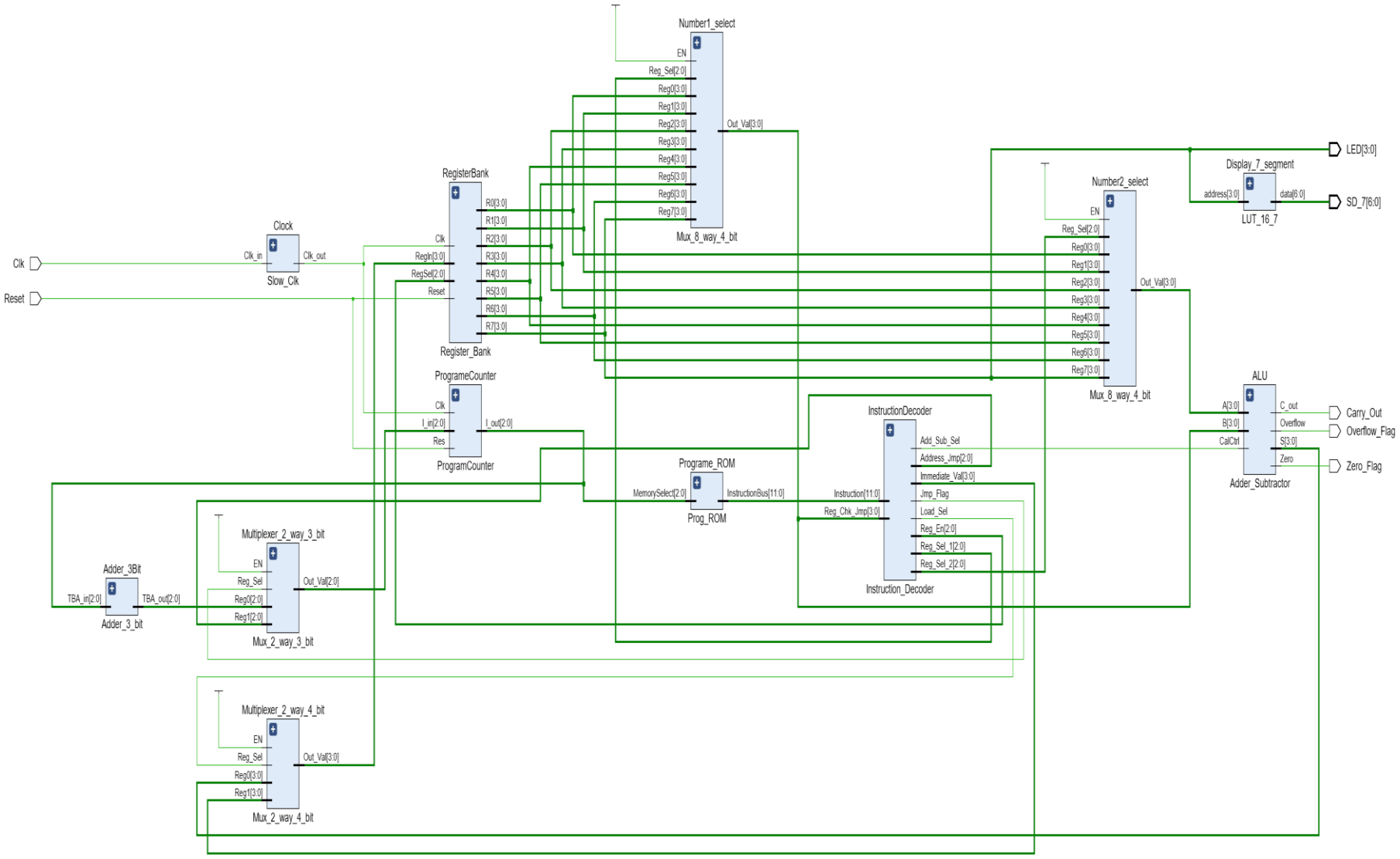
```
end component;

component Instruction_Decoder
   Port ( Instruction : in STD_LOGIC_VECTOR (11 downto 0);
          Reg_Chk_Jmp : in STD_LOGIC_VECTOR (3 downto 0);
          Load_Sel : out STD_LOGIC;
          Add_Sub_Sel : out STD_LOGIC;
          Jmp_Flag : out STD_LOGIC;
          Reg_En : out STD_LOGIC_VECTOR (2 downto 0);
          Immediate_Val : out STD_LOGIC_VECTOR (3 downto 0);
          Reg_Sel_1 : out STD_LOGIC_VECTOR (2 downto 0);
          Reg_Sel_2 : out STD_LOGIC_VECTOR (2 downto 0);
          Address_Jmp : out STD_LOGIC_VECTOR (2 downto 0));
end component;

component Mux_2_way_4_bit
   Port ( Reg0 : in STD_LOGIC_VECTOR (3 downto 0);
      Reg1 : in STD_LOGIC_VECTOR (3 downto 0);
      EN : in STD_LOGIC;
      Reg_Sel : in STD_LOGIC ;
      Out_Val : out STD_LOGIC_VECTOR (3 downto 0));
end component;

component Register_Bank
   Port ( RegIn : in STD_LOGIC_VECTOR (3 downto 0);
          Clk : in STD_LOGIC;
          Reset : in STD_LOGIC;
          RegSel : in STD_LOGIC_VECTOR (2 downto 0);
          R0 : out STD_LOGIC_VECTOR (3 downto 0);
          R1 : out STD_LOGIC_VECTOR (3 downto 0);
          R2 : out STD_LOGIC_VECTOR (3 downto 0);
          R3 : out STD_LOGIC_VECTOR (3 downto 0);
          R4 : out STD_LOGIC_VECTOR (3 downto 0);
          R5 : out STD_LOGIC_VECTOR (3 downto 0);
          R6 : out STD_LOGIC_VECTOR (3 downto 0);
          R7 : out STD_LOGIC_VECTOR (3 downto 0));
end component;

component Mux_8_way_4_bit
   Port ( Reg0 : in STD_LOGIC_VECTOR (3 downto 0);
          Reg1 : in STD_LOGIC_VECTOR (3 downto 0);
          Reg2 : in STD_LOGIC_VECTOR (3 downto 0);
          Reg3 : in STD_LOGIC_VECTOR (3 downto 0);
          Reg4 : in STD_LOGIC_VECTOR (3 downto 0);
          Reg5 : in STD_LOGIC_VECTOR (3 downto 0);
          Reg6 : in STD_LOGIC_VECTOR (3 downto 0);
          Reg7 : in STD_LOGIC_VECTOR (3 downto 0);
```

```
        EN : in STD_LOGIC;
        Reg_Sel : in STD_LOGIC_VECTOR (2 downto 0);
        Out_Val : out STD_LOGIC_VECTOR (3 downto 0));
end component;

component Adder_Subtractor
   Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
        B : in STD_LOGIC_VECTOR (3 downto 0);
        CalCtrl : in STD_LOGIC;
        C_out : out STD_LOGIC;
        Overflow : out STD_LOGIC;
        Zero : out STD_LOGIC;
        S : out STD_LOGIC_VECTOR (3 downto 0));
 end component;

 component LUT_16_7
   Port ( address : in STD_LOGIC_VECTOR (3 downto 0);
        data : Out STD_LOGIC_VECTOR (6 downto 0));
 end component;

   Signal Clk_out : std_logic;
   Signal Ins_next, Ins_Current, Normal_Ins : std_logic_vector (2 downto 0);

   Signal jump_flag, Add_Sub_Sellect, Load_Select : std_logic;
   Signal Instruction : std_logic_vector (11 downto 0);
   Signal Immediate_Value,  Calculate_value : std_logic_vector (3 downto 0);
   Signal Register_Enable, Num_1_Select, Num_2_Select, Jump_Address : std_logic_vector
(2 downto 0);

   Signal Register_save_Value, R0,R1,R2,R3,R4,R5,R6,R7 : std_logic_vector (3 downto 0);

   Signal Number1, Number2 : std_logic_vector (3 downto 0);
   Signal C_out : std_logic;   --carry out

begin
Clock :Slow_Clk port map(
            Clk_in => Clk,
            Clk_out => Clk_out);

Number1_select : Mux_8_way_4_bit port map(
            Reg0 => R0,
            Reg1 => R1,
            Reg2 => R2,
            Reg3 => R3,
            Reg4 => R4,
            Reg5 => R5,
            Reg6 => R6,
```

```
            Reg7 => R7,
            EN => '1',
            Reg_Sel => Num_1_Select,
            Out_Val => Number1);

Number2_select : Mux_8_way_4_bit port map(
            Reg0 => R0,
            Reg1 => R1,
            Reg2 => R2,
            Reg3 => R3,
            Reg4 => R4,
            Reg5 => R5,
            Reg6 => R6,
            Reg7 => R7,
            EN => '1',
            Reg_Sel => Num_2_Select,
            Out_Val => Number2);

RegisterBank : Register_Bank port map(
            RegIn => Register_save_Value,
            Clk => Clk_out,
            Reset => Reset,
            RegSel => Register_Enable,
            R0 => R0,
            R1 => R1,
            R2 => R2,
            R3 => R3,
            R4 => R4,
            R5 => R5,
            R6 => R6,
            R7 => R7);

ALU : Adder_Subtractor port map(
            A => Number2,
            B => Number1,
            CalCtrl => Add_Sub_Sellect,
            Overflow => Overflow_Flag,
            Zero =>Zero_Flag,
            S => Calculate_value,
            C_out => Carry_Out);

InstructionDecoder : Instruction_Decoder port map(
            Instruction => Instruction,
            Reg_Chk_Jmp => Number1 ,
            Load_Sel => Load_Select,
            Add_Sub_Sel => Add_Sub_Sellect,
            Jmp_Flag => jump_flag,
```

```
            Reg_En => Register_Enable,
            Immediate_Val => Immediate_Value,
            Reg_Sel_1 => Num_1_Select,
            Reg_Sel_2 => Num_2_Select,
            Address_Jmp => Jump_Address);

 Multiplexer_2_way_3_bit: Mux_2_way_3_bit port map(
            Reg0 => Normal_ins,
            Reg1 => Jump_Address,
            EN => '1',
            Reg_Sel => jump_flag,
            Out_Val => Ins_next);

 Multiplexer_2_way_4_bit : Mux_2_way_4_bit port map(
            Reg0 => Calculate_value,
            Reg1 => Immediate_Value,
            EN => '1',
            Reg_Sel => Load_Select,
            Out_val => Register_save_Value);

Display_7_segment : LUT_16_7 port map(
            address => R7,
            data => SD_7);

Programe_ROM : Prog_ROM port map(
            MemorySelect => Ins_Current,
            InstructionBus => Instruction);

ProgrameCounter : ProgramCounter port map(
            I_in=> Ins_next,
            Clk => Clk_out,
            Res => Reset,
            I_out => Ins_Current);

Adder_3Bit : Adder_3_bit port map(
            TBA_in => Ins_Current,
            TBA_out => Normal_Ins);

LED <= R7;

end Behavioral;
```

Schematic diagram of NanoProcessor



Timing Diagrams

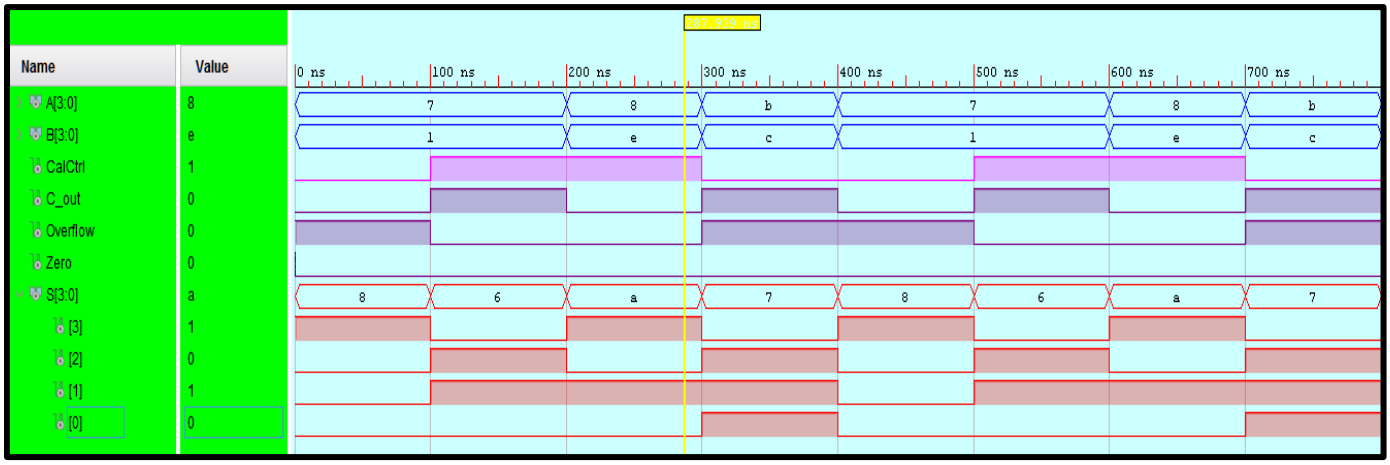
1. 4-bit Add/Subtract Unit

Simulation code for add/sub unit

```
entity Adder_Subtractor_Sim is
 -- port( );
end Adder_Subtractor_Sim;

architecture Behavioral of Adder_Subtractor_Sim is
component Adder_Subtractor

port(    A : in STD_LOGIC_VECTOR (3 downto 0);
         B : in STD_LOGIC_VECTOR (3 downto 0);
         CalCtrl : in STD_LOGIC;
         C_out : out STD_LOGIC;
         Overflow : out STD_LOGIC;
         Zero : out STD_LOGIC;
         S : out STD_LOGIC_VECTOR (3 downto 0));
end component;

signal A, B:  std_logic_vector(3 downto 0);
signal CalCtrl : std_logic;

begin
UUT : Adder_Subtractor port map(
         A => A,
         B => B,
         CalCtrl => CalCtrl,
         C_out => C_out,
         Overflow => Overflow,
         Zero => Zero,
         S => S
);
Sim : process
begin
         CalCtrl <= '0';
         A(0) <= '1';
         A(1) <= '1';
         A(2) <= '1';
         A(3) <= '0';
         B(0) <= '1';
         B(1) <= '0';
         B(2) <= '0';
         B(3) <= '0';
```

```
        wait for 100 ns;

        CalCtrl <= '1';
        A(0) <= '1';
        A(1) <= '1';
        A(2) <= '1';
        A(3) <= '0';
        B(0) <= '1';
        B(1) <= '0';
        B(2) <= '0';
        B(3) <= '0';
        wait for 100 ns;

        CalCtrl <= '1';
        A(0) <= '0';
        A(1) <= '0';
        A(2) <= '0';
        A(3) <= '1';
        B(0) <= '0';
        B(1) <= '1';
        B(2) <= '1';
        B(3) <= '1';
        wait for 100 ns;

        CalCtrl <= '0';
        A(0) <= '1';
        A(1) <= '1';
        A(2) <= '0';
        A(3) <= '1';
        B(0) <= '0';
        B(1) <= '0';
        B(2) <= '1';
        B(3) <= '1';
        wait for 100ns;
end process;
end Behavioral;
```

Timing diagram of add/sub unit



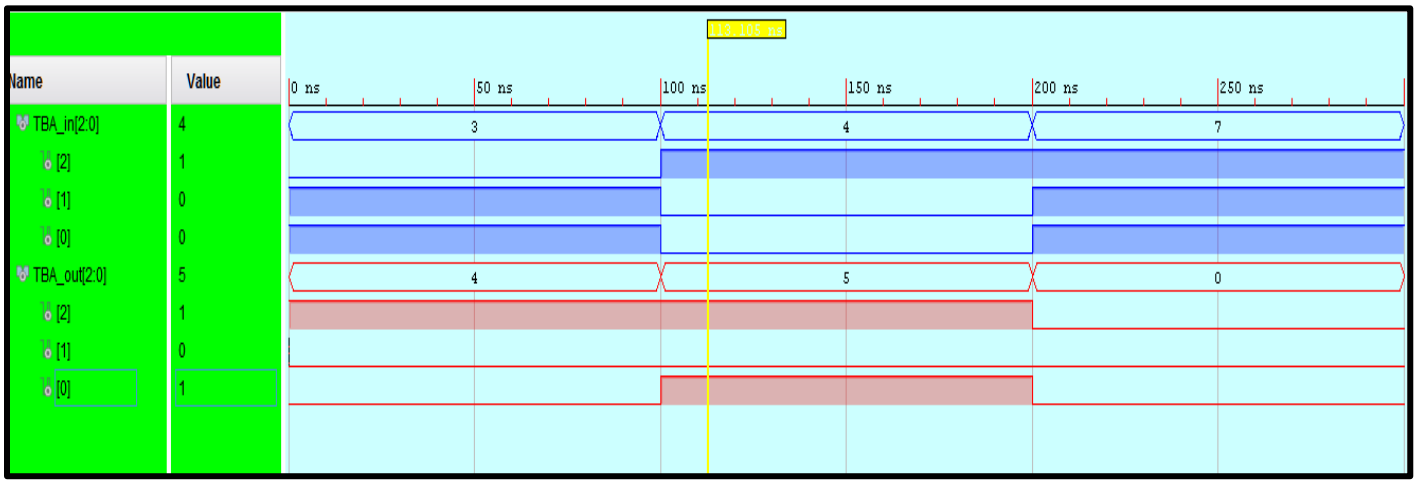
2. 3-bit Adder

Simulation code for 3-bit adder

```
entity Adder_3_bit_sim is
--  Port ( );
end Adder_3_bit_sim;

architecture Behavioral of Adder_3_bit_sim is

component Adder_3_bit
port(
   TBA_in : in STD_LOGIC_VECTOR (2 downto 0);
   TBA_out : out STD_LOGIC_VECTOR (2 downto 0));
end component;

signal TBA_in : std_logic_vector(2 downto 0);
signal TBA_out : std_logic_vector(2 downto 0);

begin

uut: Adder_3_bit
port map(
   TBA_in => TBA_in,
   TBA_out => TBA_out);

Process
```

```
Begin
    TBA_in <= "011";
    wait for 100ns;
    TBA_in <= "100";
    wait for 100ns;


    TBA_in <= "111";
    wait;


end process;

end Behavioral;
```

Timing diagram for 3-bit adder



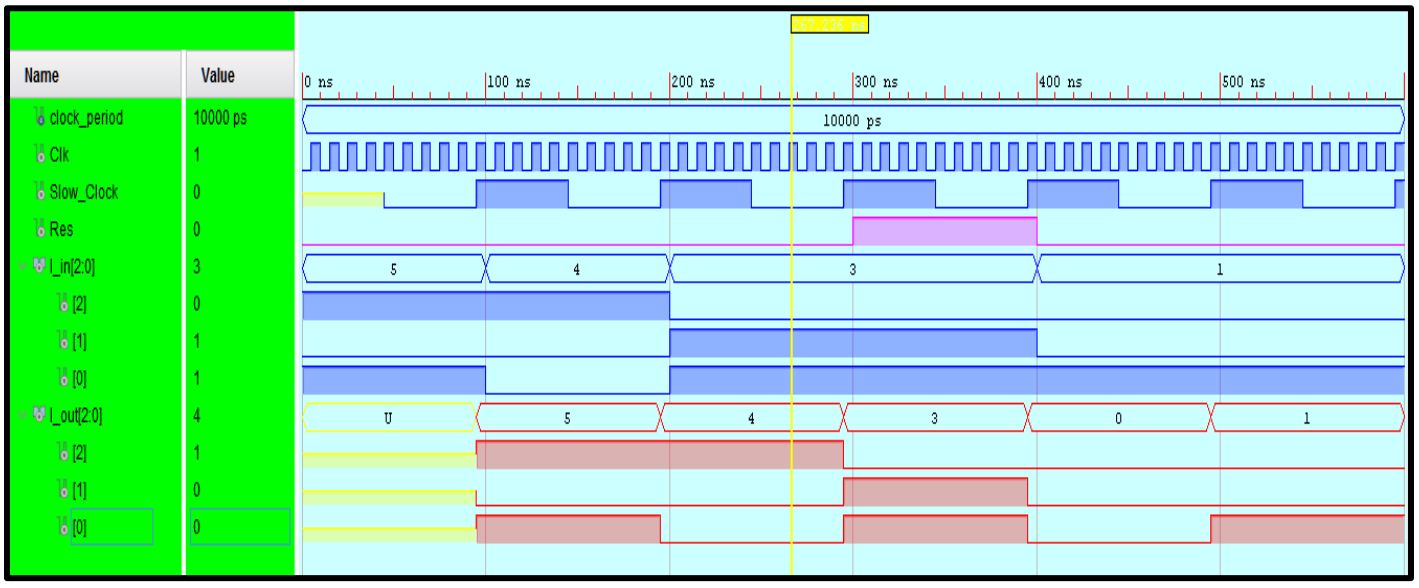
3. 3-bit Program Counter

Simulation code of 3-bit Program Counter

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

architecture Behavioral of ProgramCounter_sim is

component ProgramCounter
    Port ( I_in : in STD_LOGIC_VECTOR (2 downto 0);
           Clk : in STD_LOGIC;
           Res : in STD_LOGIC;
           I_out : out STD_LOGIC_VECTOR (2 downto 0));
end component;

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

signal I_in : STD_LOGIC_VECTOR (2 downto 0);
signal Clk, Slow_Clock : std_logic;
signal Res : STD_LOGIC:='0';
signal I_out :  STD_LOGIC_VECTOR (2 downto 0);
constant clock_period: time:= 10ns;

begin

    uut: ProgramCounter
    port map(
        I_in => I_in,
        Clk => Slow_Clock,
        Res => Res,
        I_out => I_out);
    SlowClock: Slow_Clk port map(
                Clk_in => Clk,
                Clk_out => Slow_Clock);

clock_process : process
    begin
        Clk <= '0';
        WAIT for clock_period/2;
        Clk <= '1';
        WAIT for clock_period/2;
    end process;
```

```
Sim :process
   begin

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

   I_in <= "100";
   wait for 100 ns;

   I_in <= "011";
   wait for 100 ns;

   Res <= '1';
   wait for 100 ns;
   Res <= '0';

   I_in <= "001";
   wait;

   end process;
end Behavioral;
```

Timing diagram of 3-bit Program Counter



4. K-way b-bit Multiplexer

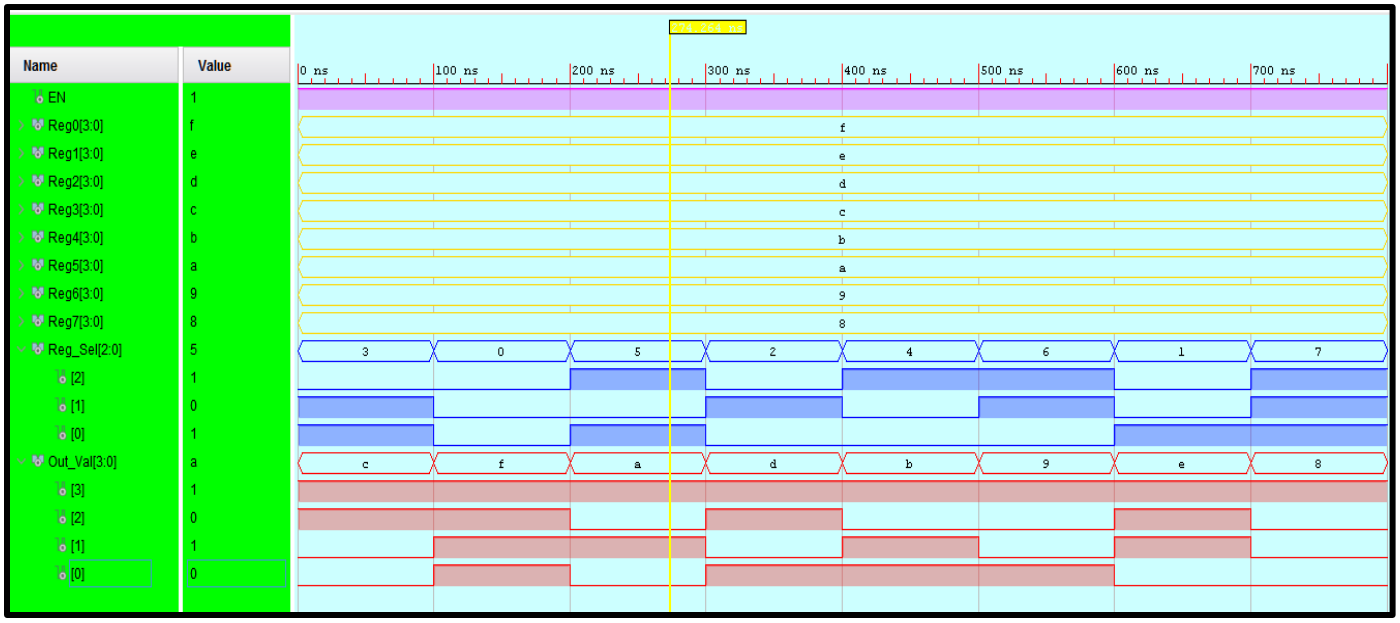
We use a decoder and try state buffers for all the k-way b-bit multiplexers. Here we show the time diagram of 8-way 4-bit Multiplexer. Other diagrams are same as this. First, we enter some values to registers of the multiplexer and after that we give different addresses to the multiplexer then the multiplexer gives the correct value in the given register.

Simulation code for 8-way 4-bit Multiplexer

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

entity Mux_8_way_4_bit_Sim is
--  Port ( );
end Mux_8_way_4_bit_Sim;

architecture Behavioral of Mux_8_way_4_bit_Sim is
component Mux_8_way_4_bit
    Port ( Reg0 : in STD_LOGIC_VECTOR (3 downto 0);
           Reg1 : in STD_LOGIC_VECTOR (3 downto 0);
           Reg2 : in STD_LOGIC_VECTOR (3 downto 0);
           Reg3 : in STD_LOGIC_VECTOR (3 downto 0);
           Reg4 : in STD_LOGIC_VECTOR (3 downto 0);
           Reg5 : in STD_LOGIC_VECTOR (3 downto 0);
           Reg6 : in STD_LOGIC_VECTOR (3 downto 0);
           Reg7 : in STD_LOGIC_VECTOR (3 downto 0);
           EN : in STD_LOGIC;
           Reg_Sel : in STD_LOGIC_VECTOR (2 downto 0);
           Out_Val : out STD_LOGIC_VECTOR (3 downto 0));
end component;

    Signal EN : std_logic;
    Signal Reg0, Reg1, Reg2, Reg3, Reg4, Reg5, Reg6, Reg7 ,Out_Val : std_logic_vector(3
downto 0);
    Signal Reg_Sel : std_logic_vector(2 downto 0);

begin
    UUT: Mux_8_way_4_bit port map(
        Reg0 => Reg0,
        Reg1 => Reg1,
        Reg2 => Reg2,
        Reg3 => Reg3,
        Reg4 => Reg4,
        Reg5 => Reg5,
        Reg6 => Reg6,
```

```
        Reg7 => Reg7,
        EN => EN,
        Reg_Sel => Reg_Sel,
        Out_Val => Out_Val);

    Sim:process
        begin
            Reg0 <= "1111";
            Reg1 <= "1110";
            Reg2 <= "1101";
            Reg3 <= "1100";
            Reg4 <= "1011";
            Reg5 <= "1010";
            Reg6 <= "1001";
            Reg7 <= "1000";
            EN <= '1';

            Reg_Sel <= "011";
            wait for 100ns;

            Reg_Sel <= "000";
            wait for 100ns;

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

            Reg_Sel <= "010";
            wait for 100ns;

            Reg_Sel <= "100";
            wait for 100ns;

            Reg_Sel <= "110";
            wait for 100ns;

            Reg_Sel <= "001";
            wait for 100ns;

            Reg_Sel <= "111";
            wait;
        end process;

end Behavioral;
```

Timing diagram of 8-way 4-bit Multiplexer



5. Register Bank

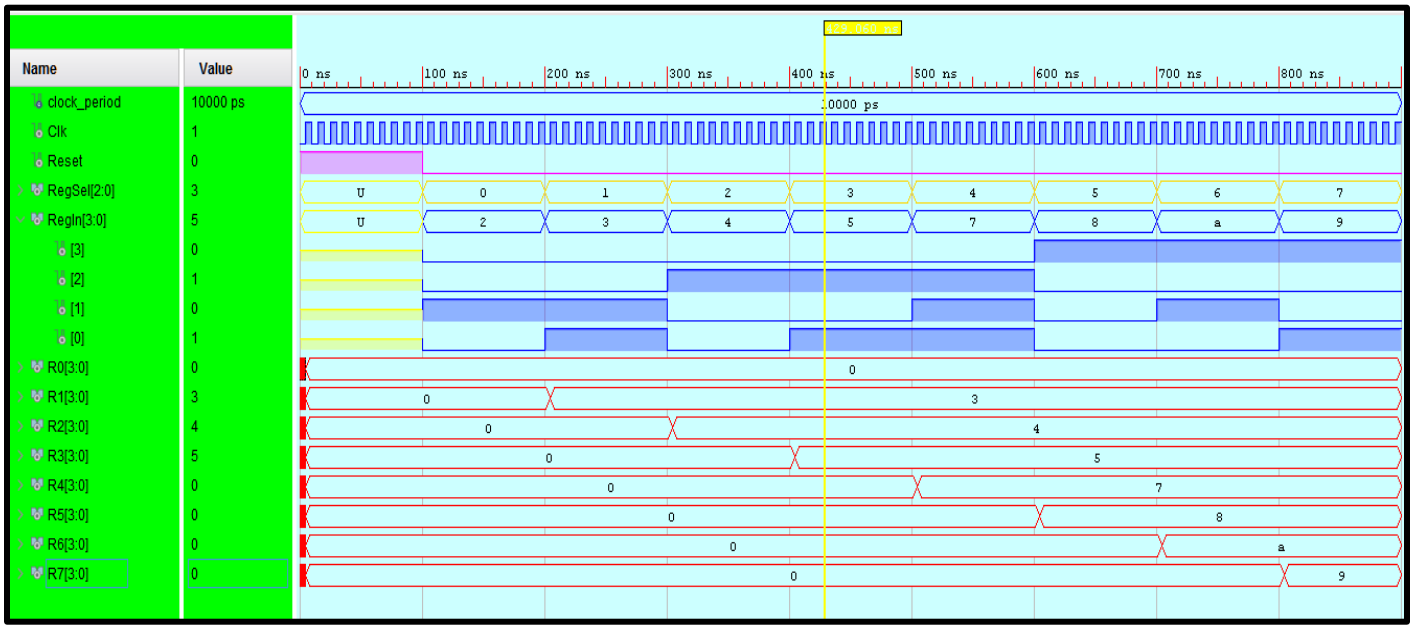
Simulation code for Register Bank

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

architecture Behavioral of Reg_Bank_Sim is
component Register_Bank
    Port ( RegIn : in STD_LOGIC_VECTOR (3 downto 0);
           Clk : in STD_LOGIC;
           Reset : in STD_LOGIC;
           RegSel : in STD_LOGIC_VECTOR (2 downto 0);
           R0 : out STD_LOGIC_VECTOR (3 downto 0);
           R1 : out STD_LOGIC_VECTOR (3 downto 0);
           R2 : out STD_LOGIC_VECTOR (3 downto 0);
           R3 : out STD_LOGIC_VECTOR (3 downto 0);
           R4 : out STD_LOGIC_VECTOR (3 downto 0);
           R5 : out STD_LOGIC_VECTOR (3 downto 0);
           R6 : out STD_LOGIC_VECTOR (3 downto 0);
           R7 : out STD_LOGIC_VECTOR (3 downto 0));
   end component;
```

```
signal Clk, Reset : std_logic;
signal RegIn, R0, R1, R2, R3, R4, R5, R6, R7 : std_logic_vector(3 downto 0);
signal RegSel : std_logic_vector(2 downto 0);
CONSTANT clock_period : TIME := 10ns;

begin
UUT: Register_Bank
port map(
   RegIn => RegIn,
   Clk => Clk,
   Reset => Reset,
   RegSel => RegSel,
   R0 => R0,
   R1 => R1,
   R2 => R2,
   R3 => R3,
   R4 => R4,
   R5 => R5,
   R6 => R6,
   R7 => R7);

clock_process : PROCESS
      BEGIN
         Clk <= '0';
         WAIT FOR clock_period/2;
         Clk <= '1';
         WAIT FOR clock_period/2;
      END PROCESS;

sim : PROCESS
         BEGIN
            Reset <= '1';
            WAIT FOR 100ns;

            Reset <= '0';
            RegIn <= "0010";
            RegSel <= "000";
            WAIT FOR 100ns;

            RegIn <= "0011";
            RegSel <= "001";
            WAIT FOR 100ns;

            RegIn <= "0100";
            RegSel <= "010";
            WAIT FOR 100ns;
```

```
        RegIn <= "0101";
        RegSel <= "011";
        WAIT FOR 100ns;

        RegIn <= "0111";
        RegSel <= "100";
        WAIT FOR 100ns;

        RegIn <= "1000";
        RegSel <= "101";
        WAIT FOR 100ns;

        RegIn <= "1010";
        RegSel <= "110";
        WAIT FOR 100ns;

        RegIn <= "1001";
        RegSel <= "111";

        WAIT;
    END PROCESS;
END Behavioral;
```

Timing diagram of Register Bank

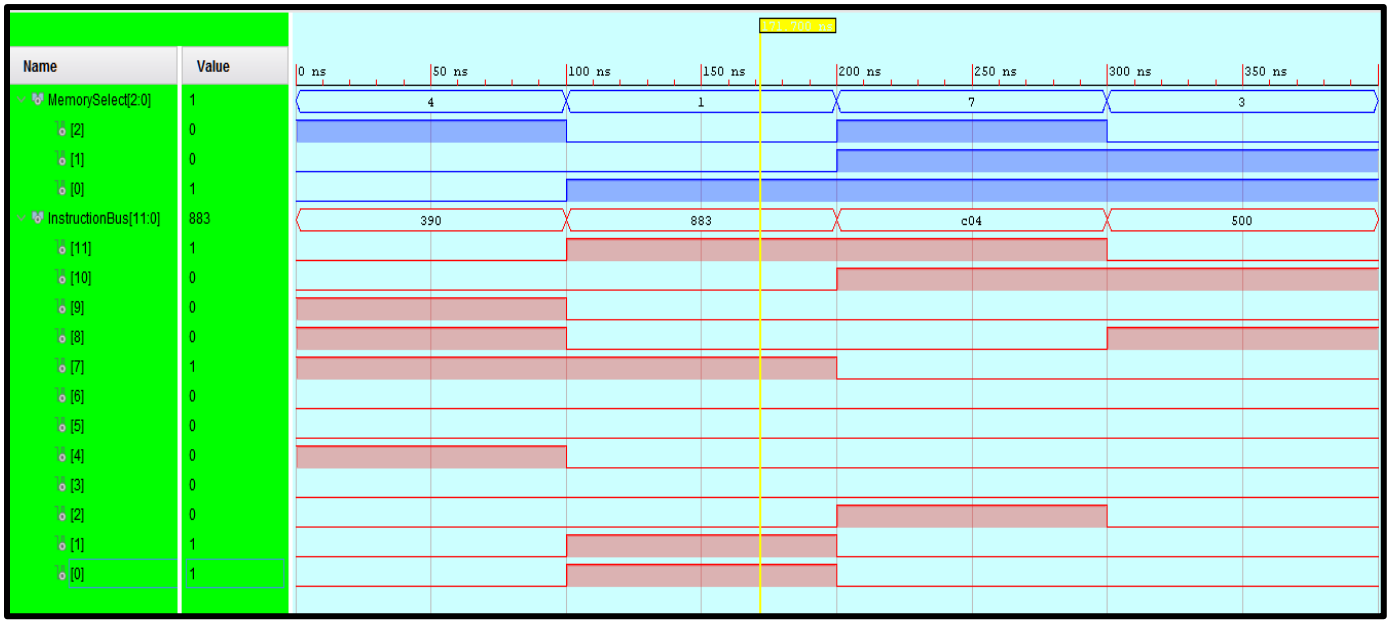


6. Program ROM

Simulation code for Program ROM

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

architecture Behavioral of Program_ROM_sim is
component Program_ROM
    Port ( MemorySelect : in STD_LOGIC_VECTOR (2 downto 0);
           InstructionBus : out STD_LOGIC_VECTOR (11 downto 0));
end component;

signal MemorySelect : STD_LOGIC_VECTOR (2 downto 0);
signal InstructionBus : STD_LOGIC_VECTOR (2 downto 0);

begin

uut: Program_ROM
    port map(
        MemorySelect => MemorySelect,
        InstructionBus => InstructionBus
    );

sim:process

begin

    MemorySelect <= "100";
    wait for 100ns;

    MemorySelect <= "001";
    wait for 100ns;

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

    MemorySelect <= "011";
    wait;

end process;

end Behavioral;
```

Timing diagram of Program ROM



7. Instruction Decoder

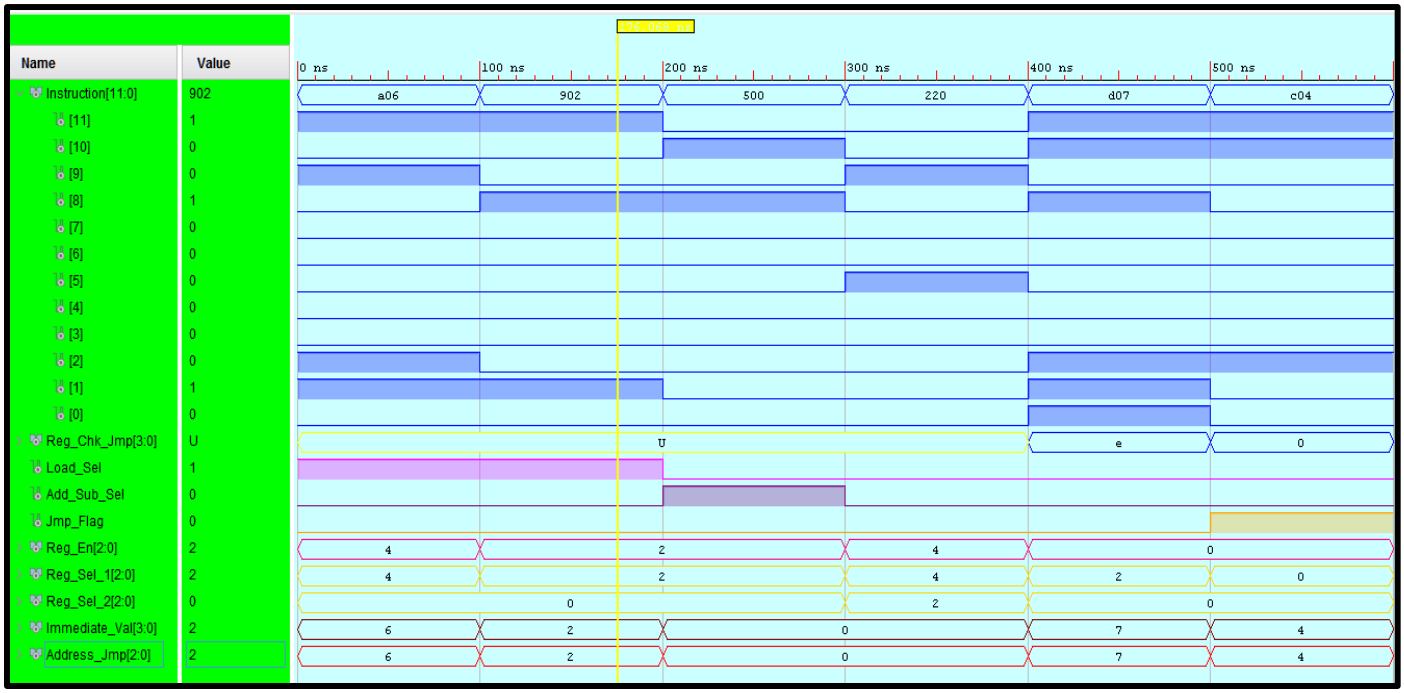
Instruction Decoder was simulated by giving 4 different instructions to the Instruction Decoder. The instruction providing part of the simulation file is follows.

Simulation code for Program Instruction Decoder

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

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

architecture Behavioral of Instruction_Decoder_Sim is
component Instruction_Decoder
    Port ( Instruction : in STD_LOGIC_VECTOR (11 downto 0);
           Reg_Chk_Jmp : in STD_LOGIC_VECTOR (3 downto 0);
           Load_Sel : out STD_LOGIC;
           Add_Sub_Sel : out STD_LOGIC;
           Jmp_Flag : out STD_LOGIC;
           Reg_En : out STD_LOGIC_VECTOR (2 downto 0);
           Immediate_Val : out STD_LOGIC_VECTOR (3 downto 0);
           Reg_Sel_1 : out STD_LOGIC_VECTOR (2 downto 0);
```

```
        Reg_Sel_2 : out STD_LOGIC_VECTOR (2 downto 0);
        Address_Jmp : out STD_LOGIC_VECTOR (2 downto 0));

end component;

   Signal Load_Sel, Add_Sub_Sel, Jmp_Flag : std_logic;
   Signal Reg_En, Reg_Sel_1, Reg_Sel_2, Address_Jmp : std_logic_vector(2 downto 0);
   Signal Reg_Chk_Jmp, Immediate_Val : std_logic_vector(3 downto 0);
   Signal Instruction : std_logic_vector( 11 downto 0);

begin
   UUT: Instruction_Decoder port map(
      Instruction => Instruction,
      Reg_Chk_Jmp => Reg_Chk_Jmp,
      Load_Sel => Load_Sel,
      Add_Sub_Sel => Add_Sub_Sel,
      Jmp_Flag => Jmp_Flag,
      Reg_En => Reg_En,
      Immediate_Val => Immediate_Val,
      Reg_Sel_1 => Reg_Sel_1,
      Reg_Sel_2 => Reg_Sel_2,
      Address_Jmp => Address_Jmp);

   sim :process
      begin
        --Assembly instruction is          MOVI R4, 6    ; R1 <- 6
              Instruction <= "101000000110";
              wait for 100ns;
        --Assembly instruction is          MOVI R2, 2    ; R1 <- 2
              Instruction <= "100100000010";
              wait for 100ns;
        --Assembly instruction is          NEG R2              ; R2 <- -R2
              Instruction <= "010100000000";
              wait for 100ns;
        --Assembly instruction is          ADD R4,R2     ; R4 <- R4 + R2
              Instruction <= "001000100000";
              wait for 100ns;
        --Assembly instruction is          JZR R2,7        ; If R2 = 0 jump to line 7
              Instruction <= "110100000111";
              Reg_Chk_Jmp <= "1110";
              wait for 100ns;
        --Assembly instruction is          JZR R0,4        ; If R0 = 0 jump to line 4
              Instruction <= "110000000100";
              Reg_Chk_Jmp <= "0000";
              wait;
   end process;
end Behavioral;
```

Timing diagram of Program Instruction Decoder



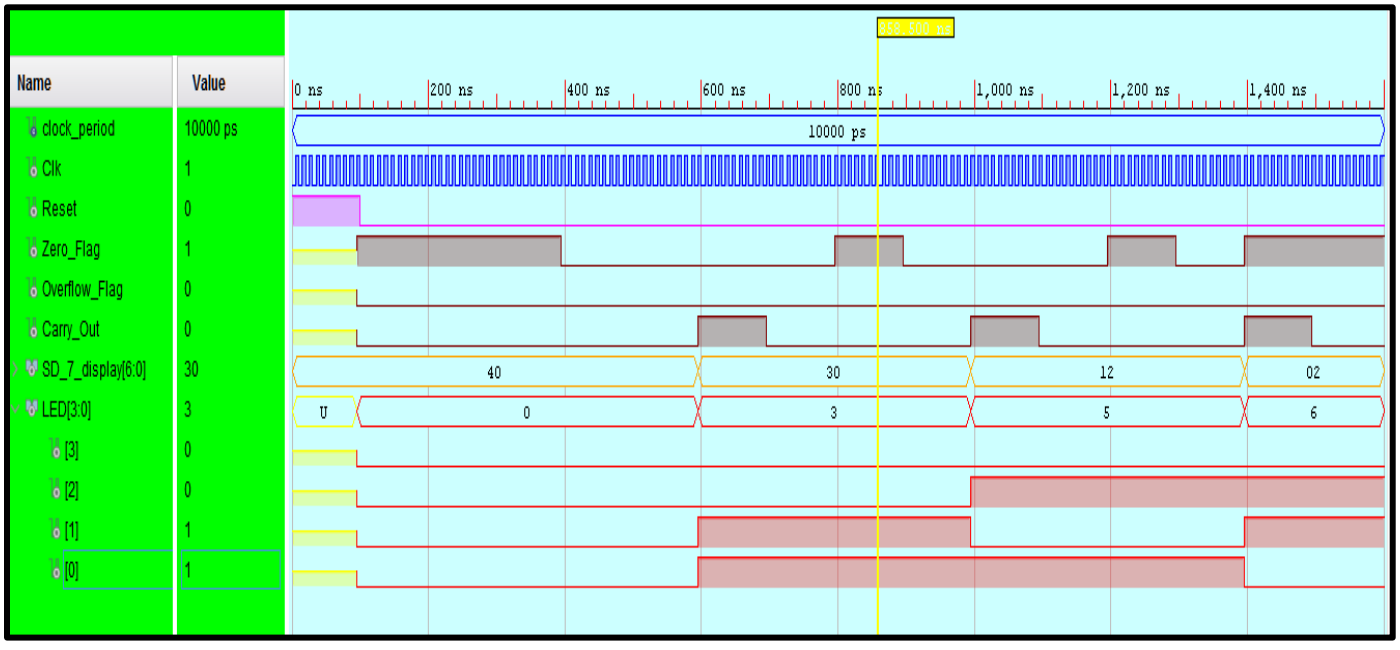
8. NanoProcessor

Simulation Code for NanoProcessor

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

architecture Behavioral of NanoProcessorSim is
component NanoProcessor
    port( Reset : in std_logic;
          Clk : in std_logic;
          Zero_Flag : out std_logic;
          Overflow_Flag : out std_logic;
          Carry_Out : out std_logic;
          LED : out STD_LOGIC_VECTOR (3 downto 0);
          SD_7 : out std_logic_vector( 6 downto 0));
end component;

Signal Reset, Clk, Zero_Flag, Overflow_Flag, Carry_Out: std_logic;
Signal SD_7_display: std_logic_vector (6 downto 0);
Signal LED: std_logic_vector (3 downto 0);
constant clock_period : time := 10ns;
```

```
begin
UUT: NanoProcessor port map(
                Reset => Reset,
                Clk => Clk,
                Zero_Flag => Zero_Flag,
                Overflow_Flag => Overflow_Flag,
                Carry_Out => Carry_Out,
                LED =>LED,
                SD_7 => SD_7_display);

clock_process: process
             begin
                Clk <= '0';
                wait for clock_period/2;
                Clk <= '1';
                wait for clock_period/2;
             end process;

Sim : process
       begin
          Reset <= '1';
          wait for 100ns;
          Reset <= '0';
          wait;
   end process;
end Behavioral;
```

Timing Diagram of NanoProcessor



Conclusion

- It is more apt to design the program counter using D flip flops since it needs to reset to 0 when required.
- MUX with tri state buffers are better than traditional MUX due to the reduction of unnecessary connections. At the same time, it makes the code more readable and understandable.
- The heavy usage of wires all around the circuit is mitigated through the usage of 3,4 and 12-bit buses.
- Instruction decoder is the most important component in the project as it stores all the machine language instructions of the processor.

Contribution of Members

As mentioned in the Lab this is a group project then we divide main components between three group members. After creating components, all of us create the Nano Processor by connecting predesigned components together and simulate it. Finally Code the constraint file and generate the bitstream.

1. R.A.N. Sankalana	• 4 - bit Add/Sub unit • Register bank
2. M.M. Poorna S. Cooray	• 3-bit adder • 3-bit counter • Program ROM
3. H.D.S. Vidulanka 4.	• Instruction Decoder, • K-way b-bit mux