In22-S2-CS1050 - Computer Organization and Digital Design

Lab 9 - 10

NANO-PROCESSOR

Design Competition
Lab Report

Abhayawickrama G.M.A.M. 220011G
Gangadari M.D.S. 220178X
Senevirathne S.M.P.U. 220599M
Weerawansa M.S.I. 220690J

5th of May 2024

TABLE OF CONTENTS

This page is intentionally left blank.

1. Basic Nano-processor

1.1. Introduction

- The objective of this lab aimed to design a 4-bit processor that can execute the following instructions.

1. **MOVI R, d** - Move immediate 4-bit value d to register R.
2. **ADD Ra, Rb** - Add values in register Ra & Rb and store the result in Ra.
3. **NEG R** - 2's complement of the binary value in register R.
4. **JZR R, d** - Jump to line d(of instructions) if the value in register R is 0.

- To build such a processor, the following components should be designed first.

1. 4-bit Add/Subtract Unit
2. 3-bit Adder
3. 3-bit Program Counter
4. 2-way 3-bit Multiplexer
5. 2-way 4-bit Multiplexer
6. 8-way 4-bit Multiplexer
7. Register Bank
8. Program ROM
9. Instruction Decoder

- After creating the above components, they were connected to the instruction decoder using buses.

- Then the design was simulated and tested on a Basys 3 board.

1.2. VHDL Design Source Codes

1.2.1. 4-bit Add/Subtract Unit

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

entity Add_Sub_4 is
    Port ( S_in1 : in STD_LOGIC_VECTOR (3 downto 0);
           S_in2 : in STD_LOGIC_VECTOR (3 downto 0);
           OpSel : in STD_LOGIC;
           S_out : out STD_LOGIC_VECTOR (3 downto 0);
           Carry : out STD_LOGIC;
           OverFlow : out STD_LOGIC;
           Zero : out STD_LOGIC);
end Add_Sub_4;

architecture Behavioral of Add_Sub_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 S0,S1,S2,S3,C0,C1,C2,C3,I0,I1,I2,I3 : std_logic;

begin

I0 <= OpSel XOR S_in1(0);
I1 <= OpSel XOR S_in1(1);
I2 <= OpSel XOR  S_in1(2);
I3 <= OpSel XOR  S_in1(3);

FA_0 : FA
port map(
    A => S_in2(0),
    B => I0,
    C_in => OpSel,
    S => S0,
    C_out => C0
    );

FA_1 : FA
port map(
    A => S_in2(1),
    B => I1,
    C_in => C0,
```

```
    S => S1,
    C_out => C1
    );

FA_2 : FA
port map(
    A => S_in2(2),
    B => I2,
    C_in => C1,
    S => S2,
    C_out => C2
    );

FA_3 : FA
port map(
    A => S_in2(3),
    B => I3,
    C_in => C2,
    S => S3,
    C_out => C3
    );

Zero <= not(S0 or S1 or S2 or S3);
Carry  <= C3;
S_out(0) <= S0;
S_out(1) <= S1;
S_out(2) <= S2;
S_out(3) <= S3;

OverFlow <=  C2 xor C3;

end Behavioral;
```

- Timing diagram can be found here.
- Elaborated design schematic can be found here.

1.2.2. 3-bit Adder for Program Counter

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

entity RCA_3 is
    Port ( I : in STD_LOGIC_VECTOR (2 downto 0);
           S_out : out STD_LOGIC_VECTOR (2 downto 0));
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_S,FA1_S,FA2_S,FA0_C,FA1_C,FA2_C,Carry : std_logic;
    SIGNAL S : std_logic_vector(2 downto 0);

begin

    FA_0 : FA
    port map(
        A => I(0),
        B => '1',
        C_in => '0',
        S => S(0),
        C_out => FA0_C );

    FA_1 : FA
    port map(
            A => I(1),
            B => '0',
            C_in => FA0_C,
            S => S(1),
            C_out => FA1_C );

    FA_2 : FA
     port map(
              A => I(2),
              B => '0',
              C_in => FA1_C,
              S => S(2),
              C_out => Carry );
```

```
    S_out <= S;

end Behavioral;
```

- Timing diagram can be found here.
- Elaborated design schematic can be found here.

1.2.3. 2-way 3-bit Multiplexer

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

entity MUX_2_way_3_bit is
    Port ( I0 : in STD_LOGIC_VECTOR (2 downto 0);
           I1 : in STD_LOGIC_VECTOR (2 downto 0);
           S : in STD_LOGIC;
           Y : out STD_LOGIC_VECTOR (2 downto 0));
end MUX_2_way_3_bit;

architecture Behavioral of MUX_2_way_3_bit is

begin


Y(0) <= ( I0(0) AND NOT S ) OR ( I1(0) AND S );
Y(1) <= ( I0(1) AND NOT S ) OR ( I1(1) AND S );
Y(2) <= ( I0(2) AND NOT S ) OR ( I1(2) AND S );

end Behavioral;
```

- Timing diagram can be found here.
- Elaborated design schematic can be found here.

1.2.4. 2-way 4-bit Multiplexer

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

entity MUX_2_way_4_bit is
    Port ( I0 : in STD_LOGIC_VECTOR (3 downto 0);
           I1 : in STD_LOGIC_VECTOR (3 downto 0);
           S : in STD_LOGIC;
           Y : out STD_LOGIC_VECTOR (3 downto 0));
end MUX_2_way_4_bit;

architecture Behavioral of MUX_2_way_4_bit is

begin

Y(0) <= ( I0(0) AND NOT S ) OR ( I1(0) AND S );
Y(1) <= ( I0(1) AND NOT S ) OR ( I1(1) AND S );
Y(2) <= ( I0(2) AND NOT S ) OR ( I1(2) AND S );
Y(3) <= ( I0(3) AND NOT S ) OR ( I1(3) AND S );

end Behavioral;
```

- Timing diagram can be found here.
- Elaborated design schematic can be found here.

1.2.5. 8-way 4-bit Multiplexer

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

entity MUX_8_way_4_bit is
    Port ( I0 : in STD_LOGIC_VECTOR (3 downto 0);
           I1 : in STD_LOGIC_VECTOR (3 downto 0);
           I2 : in STD_LOGIC_VECTOR (3 downto 0);
           I3 : in STD_LOGIC_VECTOR (3 downto 0);
           I4 : in STD_LOGIC_VECTOR (3 downto 0);
           I5 : in STD_LOGIC_VECTOR (3 downto 0);
           I6 : in STD_LOGIC_VECTOR (3 downto 0);
           I7 : in STD_LOGIC_VECTOR (3 downto 0);
           S_in : in STD_LOGIC_VECTOR (2 downto 0);
           Y_out : out STD_LOGIC_VECTOR (3 downto 0));
end MUX_8_way_4_bit;

architecture Behavioral of MUX_8_way_4_bit is

component MUX_8_to_1
    Port ( I : in STD_LOGIC_VECTOR (7 downto 0);
           S : in STD_LOGIC_VECTOR (2 downto 0);
           EN : in STD_LOGIC;
           Y : out STD_LOGIC);
end component;

--signal A, B, C, D : STD_LOGIC;
```

```
begin

MUX_8_to_1_0 : MUX_8_to_1
    PORT MAP(
        I(0) => I0(0),
        I(1) => I1(0),
        I(2) => I2(0),
        I(3) => I3(0),
        I(4) => I4(0),
        I(5) => I5(0),
        I(6) => I6(0),
        I(7) => I7(0),
        S => S_in,
        Y => Y_out(0),
        EN => '1'
        );

MUX_8_to_1_1 : MUX_8_to_1
    PORT MAP(
        I(0) => I0(1),
        I(1) => I1(1),
        I(2) => I2(1),
        I(3) => I3(1),
        I(4) => I4(1),
        I(5) => I5(1),
        I(6) => I6(1),
        I(7) => I7(1),
        S => S_in,
        Y => Y_out(1),
        EN => '1'
        );

MUX_8_to_1_2 : MUX_8_to_1
    PORT MAP(
        I(0) => I0(2),
        I(1) => I1(2),
        I(2) => I2(2),
        I(3) => I3(2),
        I(4) => I4(2),
        I(5) => I5(2),
        I(6) => I6(2),
        I(7) => I7(2),
        S => S_in,
        Y => Y_out(2),
        EN => '1'
        );

MUX_8_to_1_3 : MUX_8_to_1
    PORT MAP(
        I(0) => I0(3),
        I(1) => I1(3),
        I(2) => I2(3),
        I(3) => I3(3),
        I(4) => I4(3),
        I(5) => I5(3),
        I(6) => I6(3),
        I(7) => I7(3),
```

```
        S => S_in,
        Y => Y_out(3),
        EN => '1'
        );

end Behavioral;
```

- Timing diagram can be found here.
- Elaborated design schematic can be found here.

1.2.6. Register Bank

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

entity RegisterBank is
    Port ( S : in STD_LOGIC_VECTOR (2 downto 0);
           RB_in : in STD_LOGIC_VECTOR (3 downto 0);
           CLK_in : in STD_LOGIC;
           R0_out : out STD_LOGIC_VECTOR (3 downto 0);
           R1_out : out STD_LOGIC_VECTOR (3 downto 0);
           R2_out : out STD_LOGIC_VECTOR (3 downto 0);
           R3_out : out STD_LOGIC_VECTOR (3 downto 0);
           R4_out : out STD_LOGIC_VECTOR (3 downto 0);
           R5_out : out STD_LOGIC_VECTOR (3 downto 0);
           R6_out : out STD_LOGIC_VECTOR (3 downto 0);
           R7_out : out STD_LOGIC_VECTOR (3 downto 0));
end RegisterBank;

architecture Behavioral of RegisterBank is

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;

component Register_4_bit
    Port ( R_in : in STD_LOGIC_VECTOR (3 downto 0);
           EN : in STD_LOGIC;
           CLK : in STD_LOGIC;
           R_out : out STD_LOGIC_VECTOR (3 downto 0));
end component;
signal tempEN : STD_LOGIC_VECTOR (7 downto 0);
begin

Decoder_3_to_8_0 : Decoder_3_to_8
    port map(
        I => S,
        EN => '1',
        Y => tempEN);

Register_4_bit_0 : Register_4_bit
    port map(
        R_in => "0000",
        EN => '1',
```

```
        CLK => CLK_in,
        R_out => R0_out);

Register_4_bit_1 : Register_4_bit
    port map(
        R_in => RB_in,
        EN => tempEN(1),
        CLK => CLK_in,
        R_out => R1_out);

Register_4_bit_2 : Register_4_bit
    port map(
        R_in => RB_in,
        EN => tempEN(2),
        CLK => CLK_in,
        R_out => R2_out);

Register_4_bit_3 : Register_4_bit
    port map(
        R_in => RB_in,
        EN => tempEN(3),
        CLK => CLK_in,
        R_out => R3_out);

Register_4_bit_4 : Register_4_bit
    port map(
        R_in => RB_in,
        EN => tempEN(4),
        CLK => CLK_in,
        R_out => R4_out);

Register_4_bit_5 : Register_4_bit
    port map(
        R_in => RB_in,
        EN => tempEN(5),
        CLK => CLK_in,
        R_out => R5_out);

Register_4_bit_6 : Register_4_bit
    port map(
        R_in => RB_in,
        EN => tempEN(6),
        CLK => CLK_in,
        R_out => R6_out);

Register_4_bit_7 : Register_4_bit
    port map(
        R_in => RB_in,
        EN => tempEN(7),
        CLK => CLK_in,
        R_out => R7_out);

end Behavioral;
```

- Timing diagram can be found here.
- Elaborated design schematic can be found here.

1.2.7. Program ROM

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

entity PROM is
    Port ( I : in STD_LOGIC_VECTOR (2 downto 0);
           O : out STD_LOGIC_VECTOR (11 downto 0));
end PROM;

architecture Behavioral of PROM is

    type rom_type is array (0 to 7) of std_logic_vector (11 downto 0);
        signal program_ROM : rom_type := (
            "100010000010", --MOVI R1,2
            "101110000011",--MOVI R7,3
            "100100000001", --MOVI R2, 1
            "010100000000", --NEG R2
            "001110010000", --ADD R7, R1
            "000010100000", --ADD R1, R2
            "110010000110", --JZR R1, 7
            "110000000100" --JZR R0, 5
           );

begin
    O <= program_ROM(to_integer(unsigned(I)));

end Behavioral;
```

- Timing diagram can be found here.
- Elaborated design schematic can be found here.

1.2.8. Instruction Decoder

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

entity ins_decoder is
    Port ( Ins : in STD_LOGIC_VECTOR (11 downto 0);
           JMP_CHK : in STD_LOGIC_VECTOR (3 downto 0);
           Reg_En : out STD_LOGIC_VECTOR (2 downto 0);
           Load_Sel : out STD_LOGIC;
           JMP : out STD_LOGIC;
           Im_Val : out STD_LOGIC_VECTOR (3 downto 0);
           Mux_A : out STD_LOGIC_VECTOR (2 downto 0);
           Mux_B : out STD_LOGIC_VECTOR (2 downto 0);
           Sub : out STD_LOGIC);
end ins_decoder;

architecture Behavioral of ins_decoder is

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 decoder_line: std_logic_vector(3 downto 0);
signal Can_JMP: std_logic;

begin

D0 : Decoder_2_to_4
    port map(
    I => Ins(11 downto 10),
    EN => '1',
    Y => decoder_line);

Can_JMP <= not(JMP_CHK(0) or JMP_CHK(1) or JMP_CHK(2) or JMP_CHK(3));
--checks if every bit coming from mux A is zero

Reg_En <= Ins(9 downto 7);
Load_Sel <= decoder_line(2);
Im_Val <= Ins(3 downto 0);
Mux_A <= Ins(9 downto 7);
Mux_B <= Ins(6 downto 4);
Sub <= decoder_line(1);
JMP <= Can_JMP and decoder_line(3);

end Behavioral;
```

- Timing diagram can be found here.
- Elaborated design schematic can be found here.

1.2.9. Nano Processor

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

entity Nanoprocessor1 is
    Port ( Clk : in STD_LOGIC;
           Reset : in STD_LOGIC;
           Zero : out std_logic;
           Overflow : out std_logic;
           S_7Seg : out STD_LOGIC_VECTOR (6 downto 0);
           an : out std_logic_vector(3 downto 0);
           RD7Out: out std_logic_vector(3 downto 0));
end Nanoprocessor1;

architecture Behavioral of Nanoprocessor1 is

component ins_decoder
port(Ins : in STD_LOGIC_VECTOR (11 downto 0);
           JMP_CHK : in STD_LOGIC_VECTOR (3 downto 0);
           Reg_En : out STD_LOGIC_VECTOR (2 downto 0);
           Load_Sel : out STD_LOGIC;
           JMP : out STD_LOGIC;
           Im_Val : out STD_LOGIC_VECTOR (3 downto 0);
           Mux_A : out STD_LOGIC_VECTOR (2 downto 0);
           Mux_B : out STD_LOGIC_VECTOR (2 downto 0);
           Sub : out STD_LOGIC);
end component;

component PROM
port( I : in STD_LOGIC_VECTOR (2 downto 0);
          O : out STD_LOGIC_VECTOR (11 downto 0));
end component;

component MUX_2_way_4_bit
port(I0 : in STD_LOGIC_VECTOR (3 downto 0);
           I1 : in STD_LOGIC_VECTOR (3 downto 0);
           S : in STD_LOGIC;
           Y : out STD_LOGIC_VECTOR (3 downto 0));
end component;

component RegisterBank
Port ( S : in STD_LOGIC_VECTOR (2 downto 0);
           RB_in : in STD_LOGIC_VECTOR (3 downto 0);
           CLK_in : in STD_LOGIC;
           R0_out : out STD_LOGIC_VECTOR (3 downto 0);
           R1_out : out STD_LOGIC_VECTOR (3 downto 0);
           R2_out : out STD_LOGIC_VECTOR (3 downto 0);
           R3_out : out STD_LOGIC_VECTOR (3 downto 0);
           R4_out : out STD_LOGIC_VECTOR (3 downto 0);
           R5_out : out STD_LOGIC_VECTOR (3 downto 0);
           R6_out : out STD_LOGIC_VECTOR (3 downto 0);
           R7_out : out STD_LOGIC_VECTOR (3 downto 0));
end component;

component MUX_8_way_4_bit
```

```
port( I0 : in STD_LOGIC_VECTOR (3 downto 0);
          I1 : in STD_LOGIC_VECTOR (3 downto 0);
          I2 : in STD_LOGIC_VECTOR (3 downto 0);
          I3 : in STD_LOGIC_VECTOR (3 downto 0);
          I4 : in STD_LOGIC_VECTOR (3 downto 0);
          I5 : in STD_LOGIC_VECTOR (3 downto 0);
          I6 : in STD_LOGIC_VECTOR (3 downto 0);
          I7 : in STD_LOGIC_VECTOR (3 downto 0);
          S_in : in STD_LOGIC_VECTOR (2 downto 0);
          Y_out : out STD_LOGIC_VECTOR (3 downto 0));
end component;

component Add_Sub_4
Port ( S_in1 : in STD_LOGIC_VECTOR (3 downto 0);
           S_in2 : in STD_LOGIC_VECTOR (3 downto 0);
           OpSel : in STD_LOGIC;
           S_out : out STD_LOGIC_VECTOR (3 downto 0);
           Carry : out STD_LOGIC;
           OverFlow : out STD_LOGIC;
           Zero : out STD_LOGIC);
end component;

component ProgramCounter
Port ( Clk : in STD_LOGIC;
    Res : in std_logic;
           I : in STD_LOGIC_VECTOR (2 downto 0);
           Y : out STD_LOGIC_VECTOR (2 downto 0));
end component;

component RCA_3
Port ( I : in STD_LOGIC_VECTOR (2 downto 0);
           S_out : out STD_LOGIC_VECTOR (2 downto 0));
end component;

component MUX_2_way_3_bit
Port ( I0 : in STD_LOGIC_VECTOR (2 downto 0);
           I1 : in STD_LOGIC_VECTOR (2 downto 0);
           S : in STD_LOGIC;
           Y : out STD_LOGIC_VECTOR (2 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;

component CLOCK
    Port ( Clk_in  : in STD_LOGIC; --clock input signal with frequency f
           Clk_out : out STD_LOGIC); --modified clock output signal with
frequency f/n
end component;

--signals from INSTRUCTION DECODER
signal Reg_En, Mux_A, Mux_B : std_logic_vector(2 downto 0);
signal Load_Sel, Sub, Jump : std_logic;
signal Im_Val:std_logic_vector(3 downto 0);
```

```
--signals from MUX ABOVE DECODER
signal To_Register_Bank : std_logic_vector(3 downto 0);

--signals from REGISTER BANK
signal RD0, RD1, RD2, RD3, RD4, RD5, RD6, RD7: std_logic_vector(3 downto
0);

--signals from MUX_A and MUX_B
signal From_Mux_A, From_Mux_B: std_logic_vector(3 downto 0);

--signals from ADDER/SUBTRACTOR
signal From_Adder_Subtractor: std_logic_vector(3 downto 0);
signal Overflow_F, Zero_F, Carry_F:std_logic;

--signals from 3 BIT ADDER
signal From_3_Bit_Adder: std_logic_vector(2 downto 0);

--signals from MUX TO PROGRAM COUNTER
signal To_Program_Counter: std_logic_vector(2 downto 0);

--signals from PROGRAM COUNTER
signal From_Program_Counter: std_logic_vector(2 downto 0);

--signals from ROM
signal Instruction: std_logic_vector(11 downto 0);

--signals from Slow CLOCK
signal Clk_out : std_logic;

begin

Slow_Clock : CLOCK
port map(Clk_in => Clk,
        Clk_out => Clk_out);

Instruction_Decoder: ins_decoder
port map(Ins => Instruction,
           JMP_CHK => From_Mux_A,
           Reg_En => Reg_En,
           Load_Sel => Load_Sel,
           JMP => Jump,
           Im_Val => Im_Val,
           Mux_A => Mux_A,
           Mux_B => Mux_B,
           Sub => Sub);

Mux_Above_InstructionDecoder: MUX_2_way_4_bit
port map(I0 => From_Adder_Subtractor,
           I1 => Im_Val,
           S => Load_Sel,
           Y => To_Register_Bank);

Register_Bank: RegisterBank
Port map( S => Reg_En,
          RB_in => To_Register_Bank,
          CLK_in => Clk_out,
          R0_out =>RD0,
```

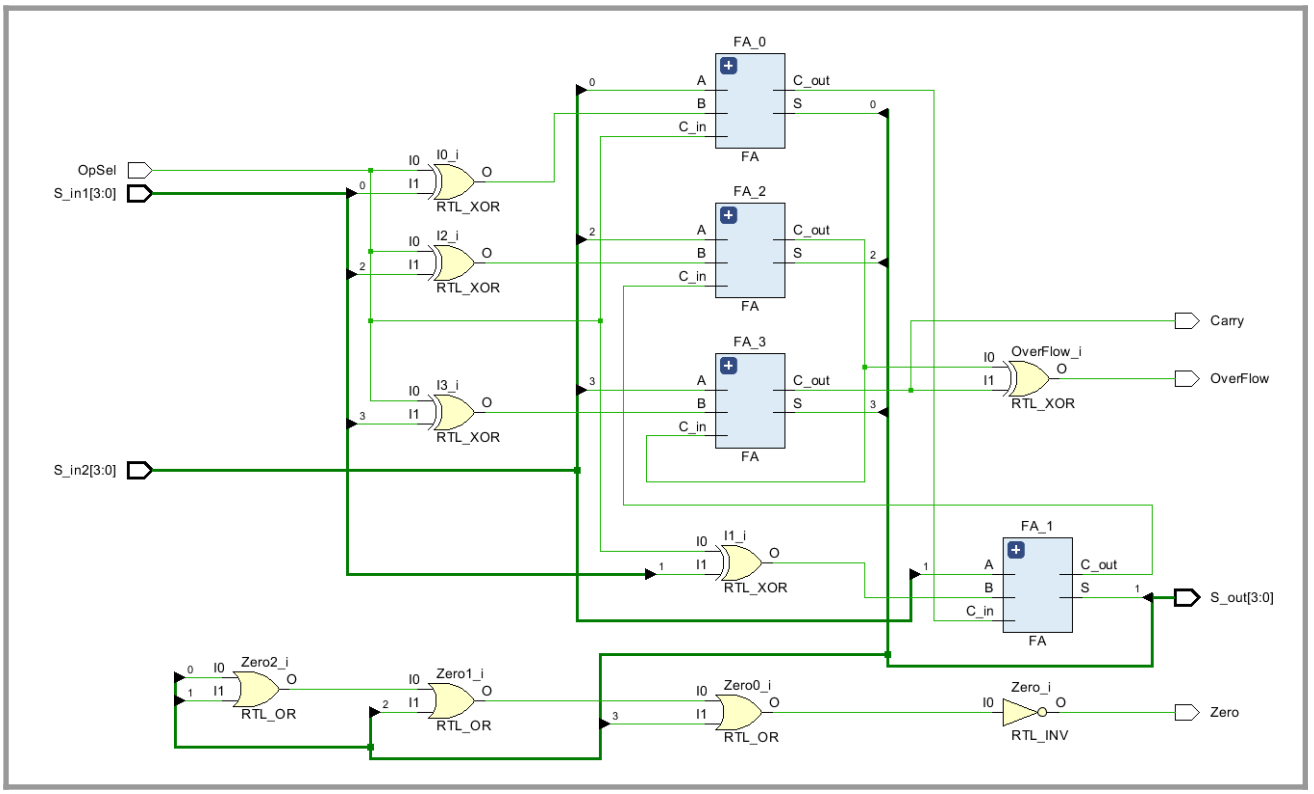
```
          R1_out =>RD1,
          R2_out =>RD2,
          R3_out =>RD3,
          R4_out =>RD4,
          R5_out =>RD5,
          R6_out =>RD6,
          R7_out =>RD7);

Mux_A_Above_Adder: MUX_8_way_4_bit
port map( I0 => RD0,
      I1 => RD1,
      I2 => RD2,
      I3 => RD3,
      I4 => RD4,
      I5 => RD5,
      I6 => RD6,
      I7 => RD7,
      S_in => Mux_A,
      Y_out => From_Mux_A);

Mux_B_Above_Adder: MUX_8_way_4_bit
  port map( I0 => RD0,
        I1 => RD1,
        I2 => RD2,
        I3 => RD3,
        I4 => RD4,
        I5 => RD5,
        I6 => RD6,
        I7 => RD7,
        S_in => Mux_B,
        Y_out => From_Mux_B);

RCA_4_Bit: Add_Sub_4
Port map(S_in1 => From_Mux_A,
       S_in2 => From_Mux_B,
       OpSel => Sub,
       S_out => From_Adder_Subtractor,
       Carry => Carry_F,
       OverFlow => Overflow_F,
       Zero => Zero_F);

Program_Counter: ProgramCounter
Port map(Clk => Clk_out,
    Res => Reset,
    I => To_Program_Counter,
    Y => From_Program_Counter);

Adder_For_Program_Counter: RCA_3
Port map( I =>From_Program_Counter,
           S_out =>From_3_Bit_Adder);

Mux_For_Program_Counter: MUX_2_way_3_bit
Port map( I0 => From_3_Bit_Adder,
      I1 => Im_Val(2 downto 0),
      S => Jump,
      Y => To_Program_Counter);
```

```
Program : PROM
port map( I => From_Program_Counter,
          O => Instruction);

To_Display : LUT_16_7
Port map( address => RD7,
     data => S_7Seg);
RD7Out <= RD7;
Zero <= Zero_F;
Overflow <= Overflow_F;
an <= "1110";

end Behavioral;
```

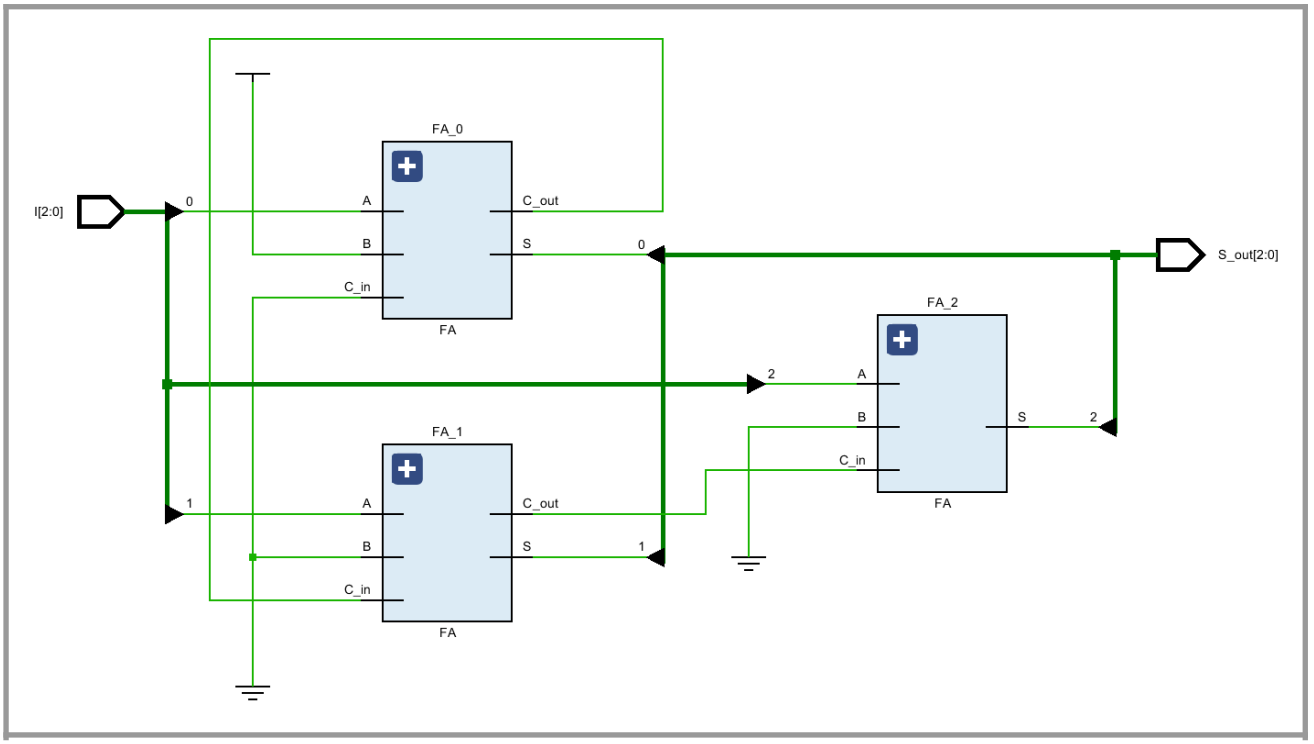
- Timing diagram can be found here.
- Elaborated design schematic can be found here.

1.3. Elaborated Design Schematics

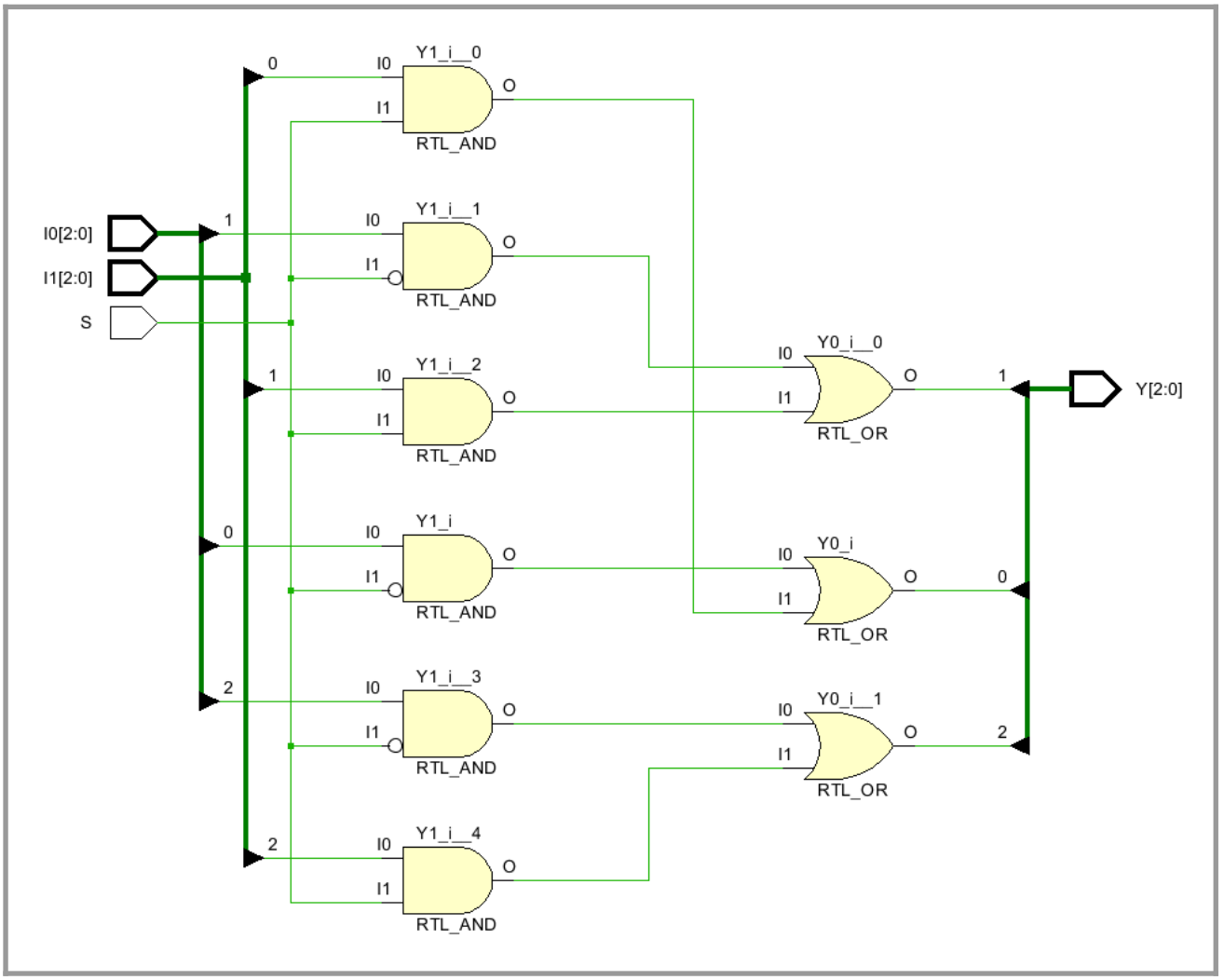
1.3.1. 4-bit Add/Subtract Unit



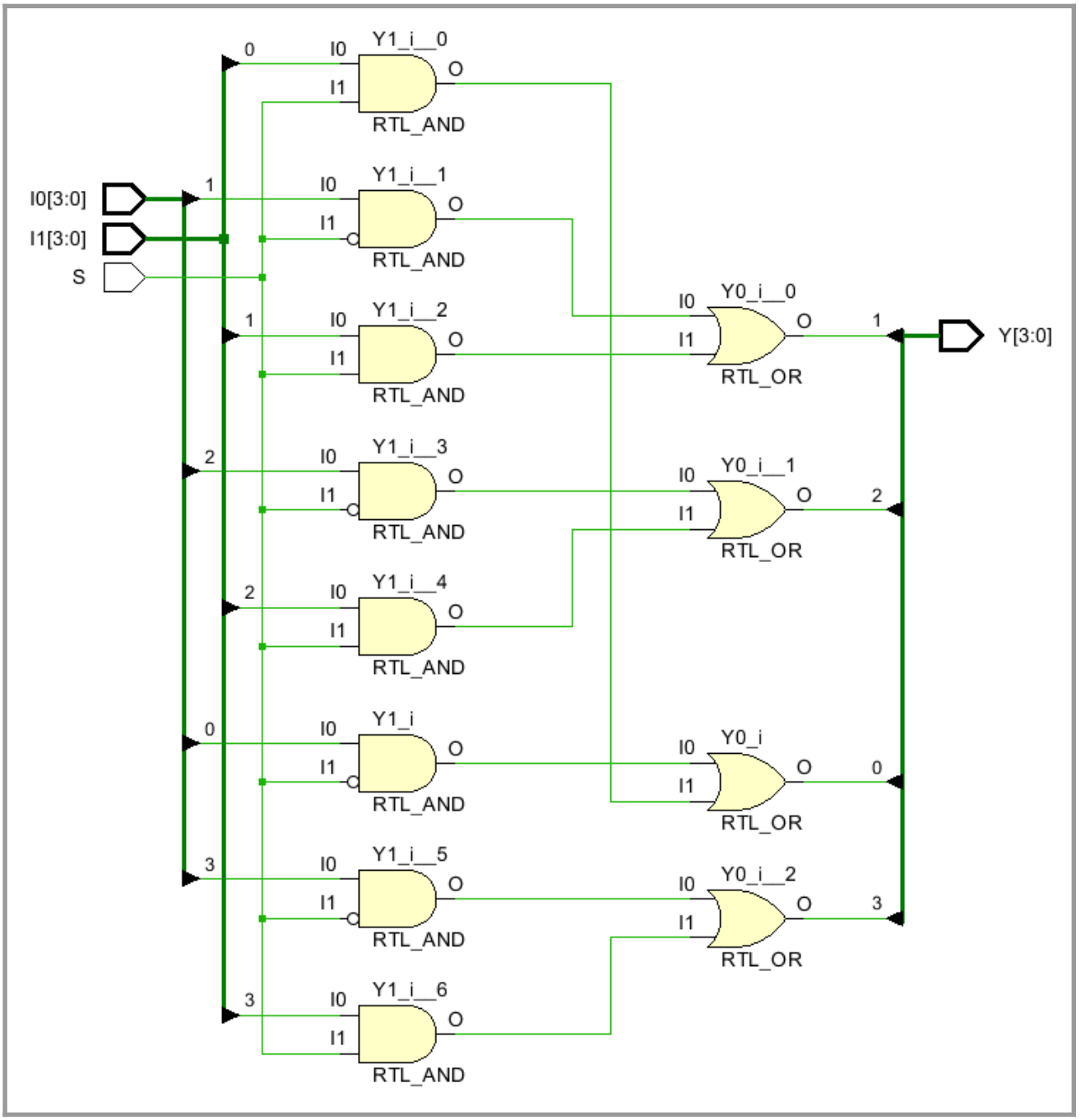
1.3.2. 3-bit Adder for Program Counter



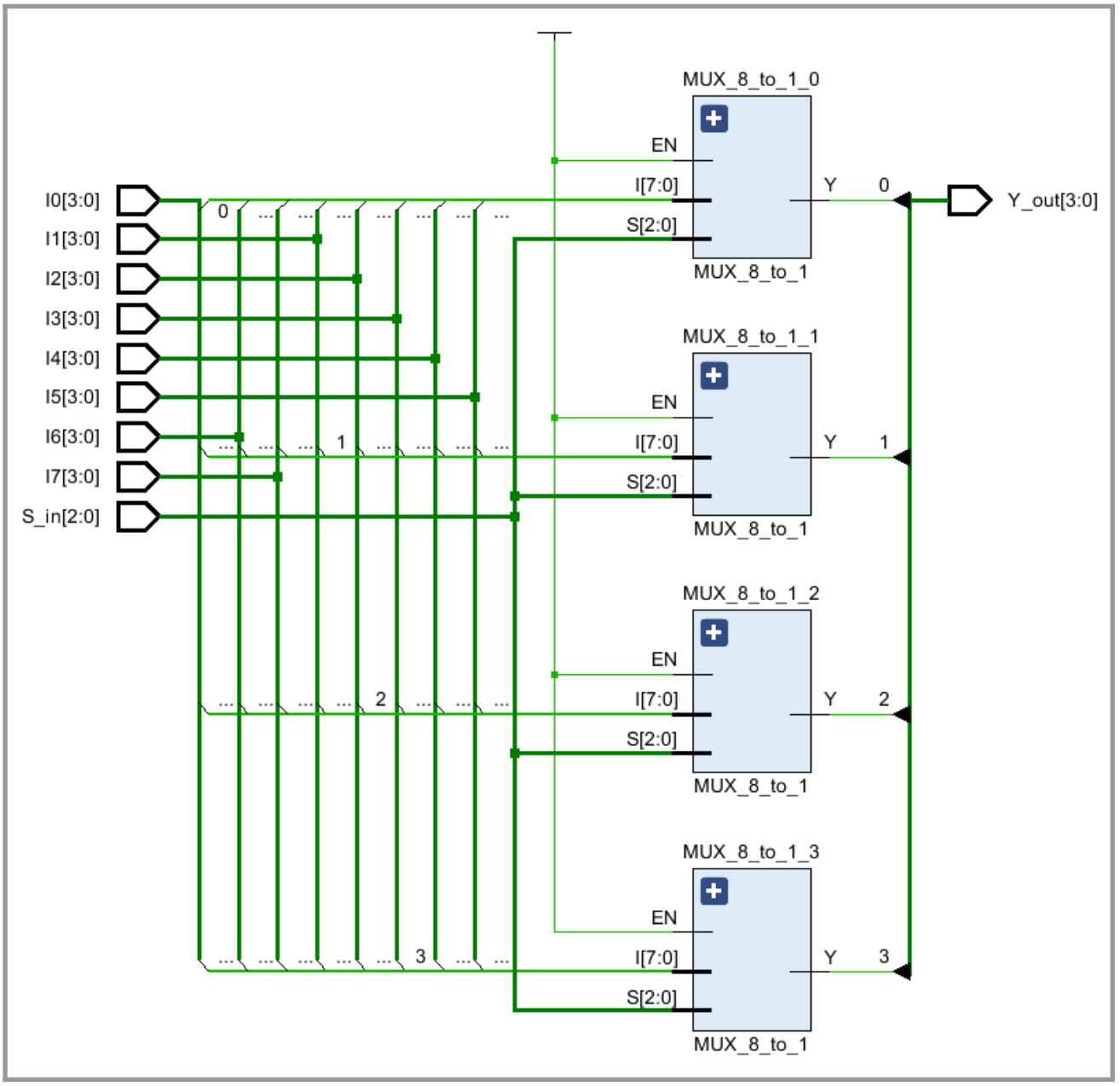
1.3.3. 2-way 3-bit Multiplexer



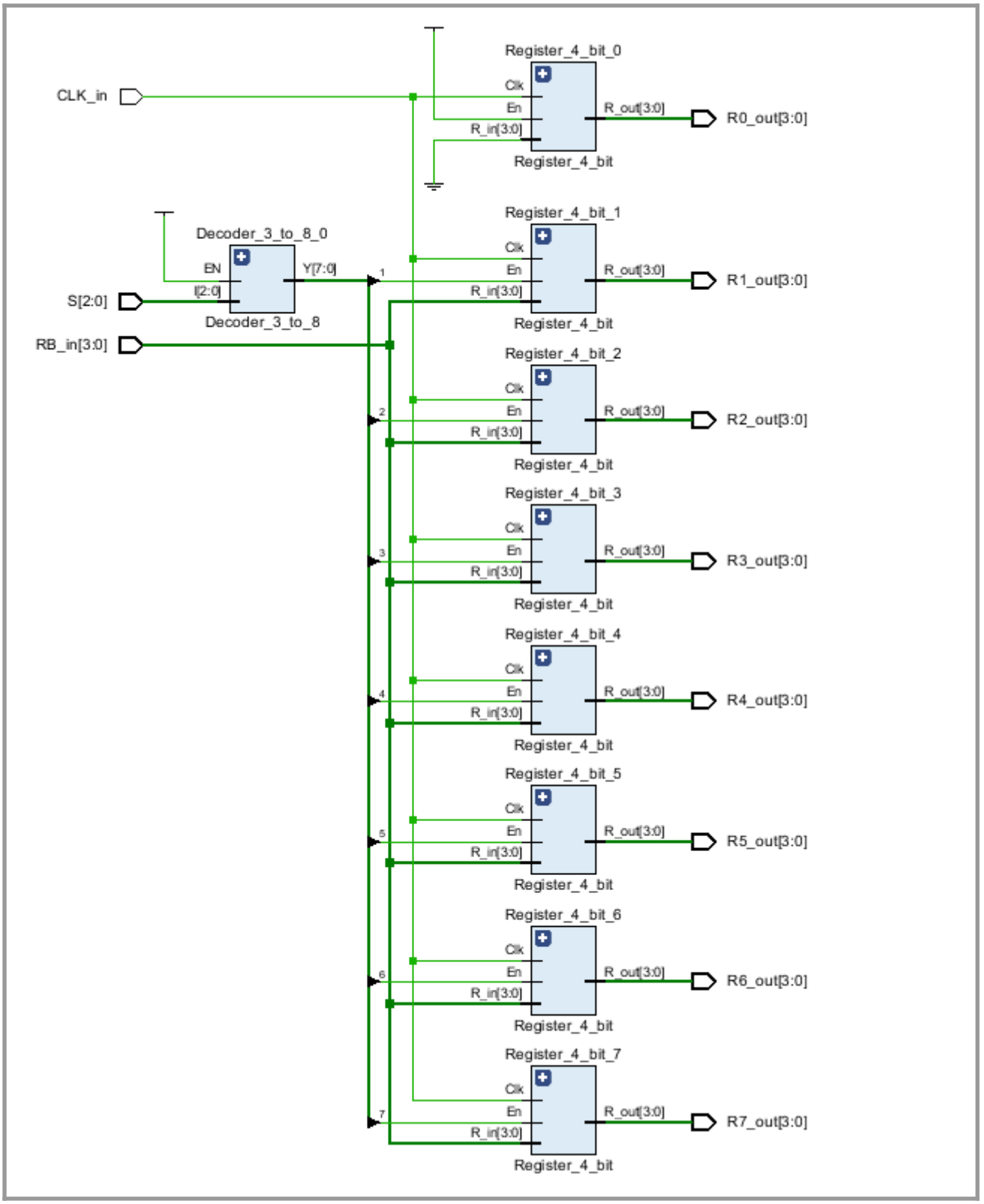
1.3.4. 2-way 4-bit Multiplexer



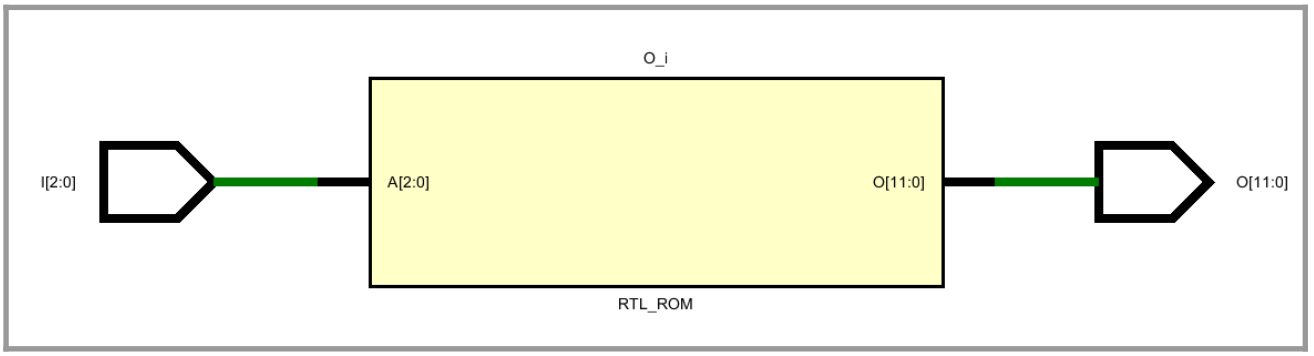
1.3.5. 8-way 4-bit Multiplexer



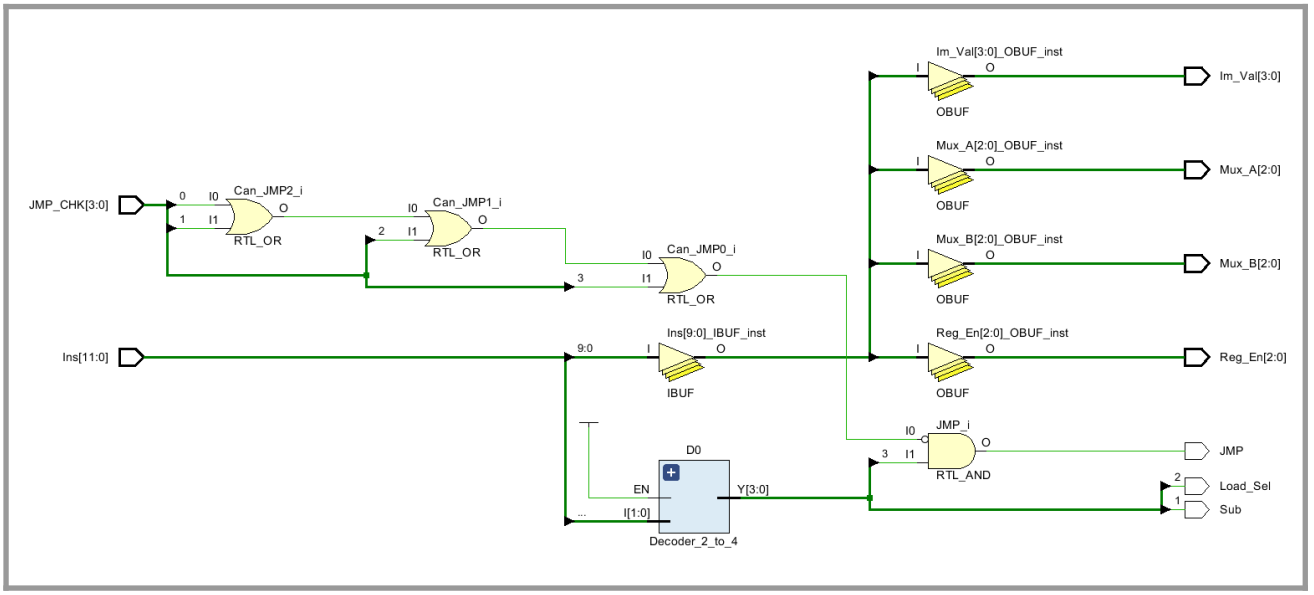
1.3.6. Register Bank



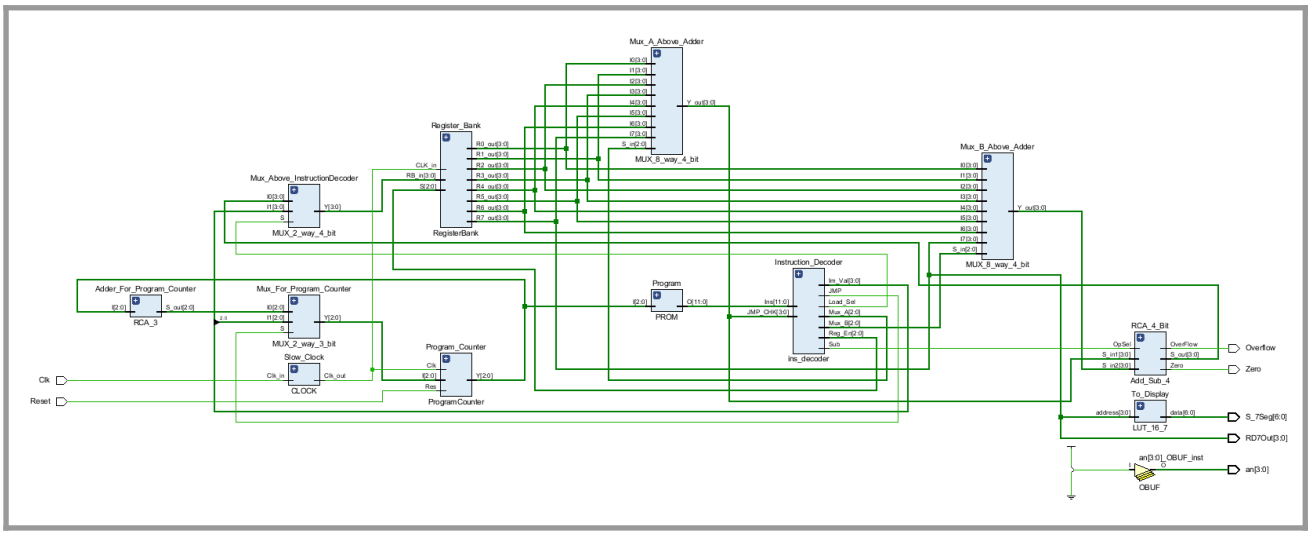
1.3.7. Program ROM



1.3.8. Instruction Decoder

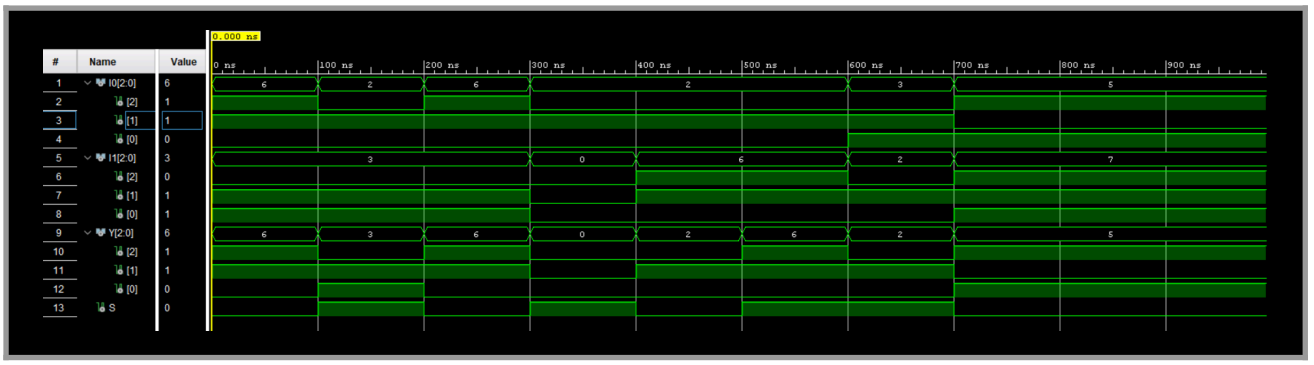


1.3.9. Nano Processor

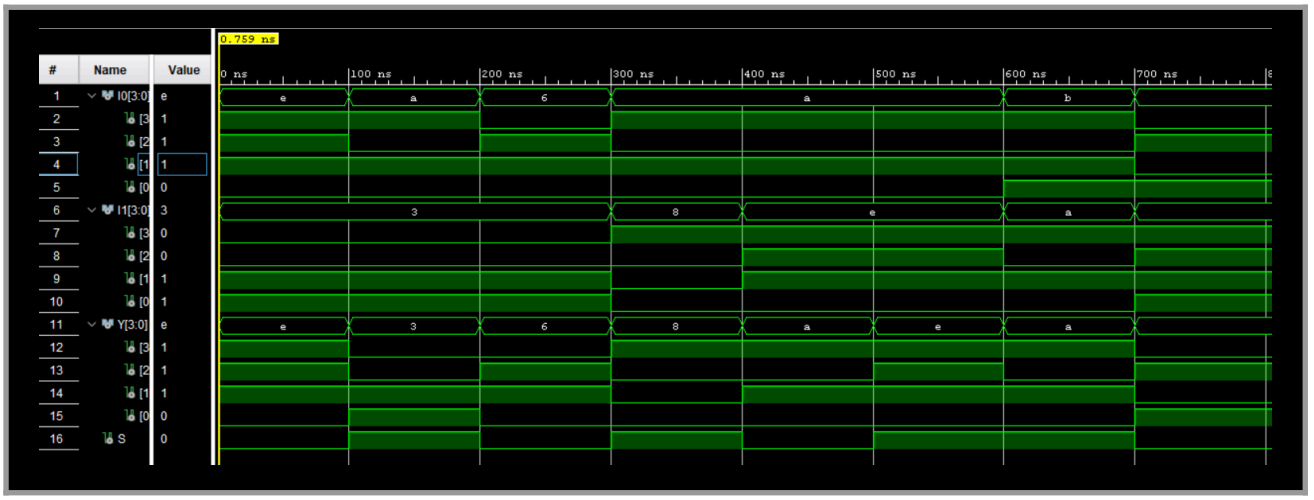


1.4. Timing Diagrams

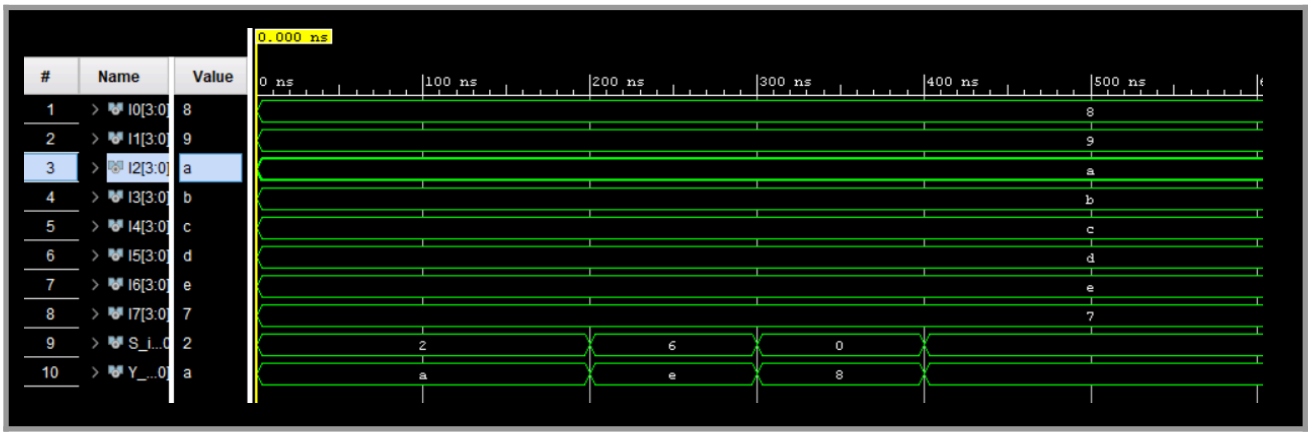
1.4.1. 2-way 3-bit Mux



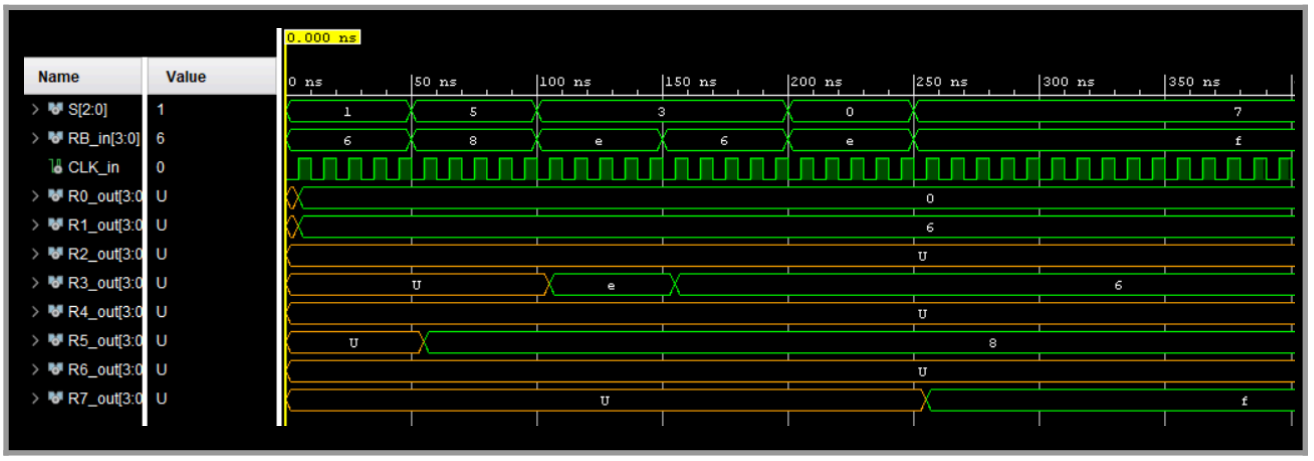
1.4.2. 2-way 4-bit Mux



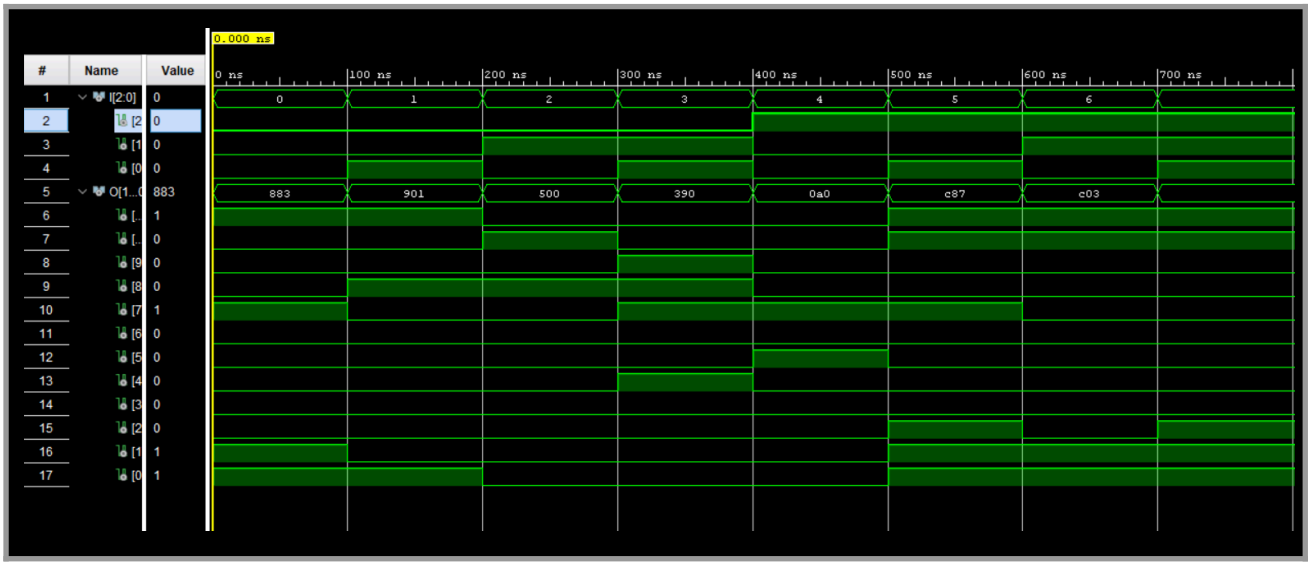
1.4.3. 8-way 4-bit Mux



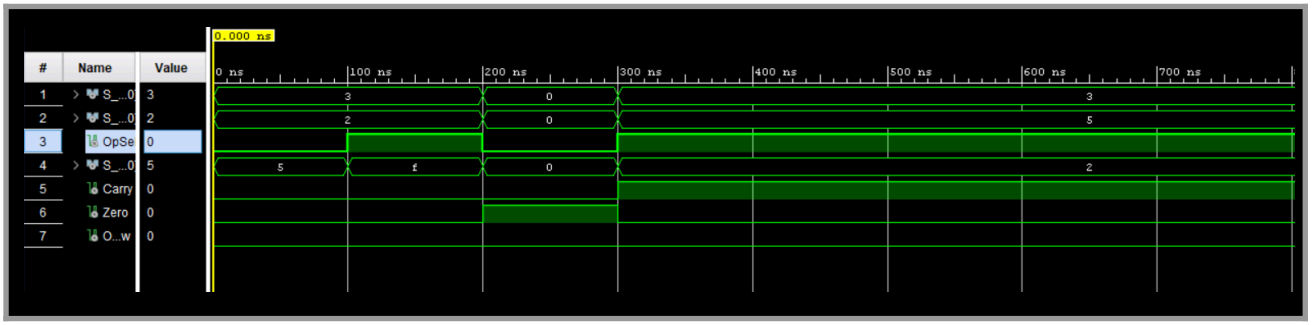
1.4.4. Register Bank



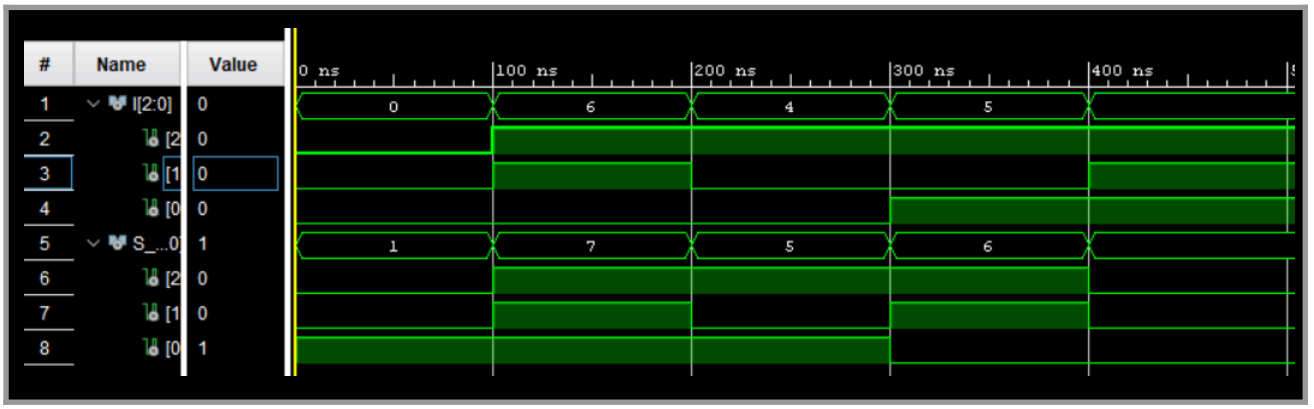
1.4.5. Program ROM



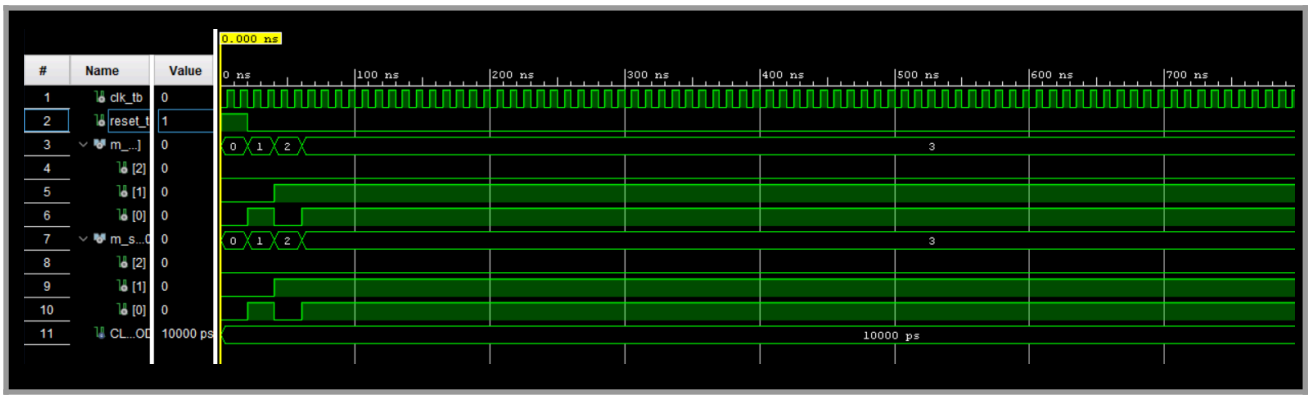
1.4.6. 4-bit Add/Subtract Unit



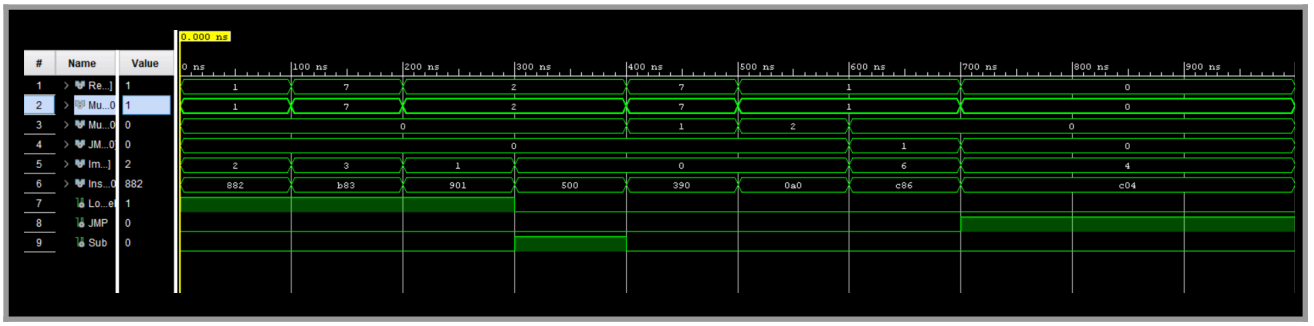
1.4.7. 3-bit Adder for Program Counter



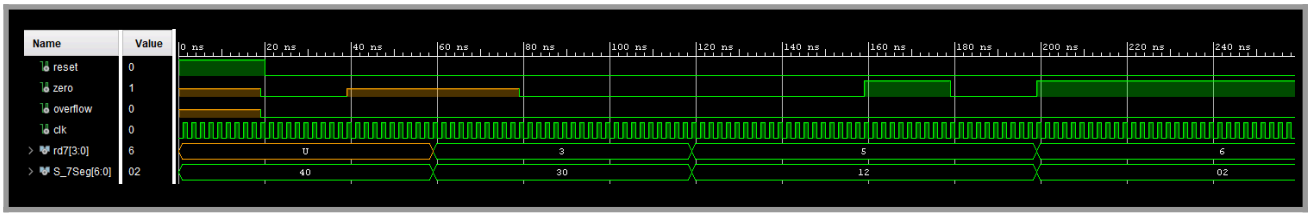
1.4.8. Program Counter



1.4.9. Instruction Decoder



1.4.10. Nano Processor



2. Improved Nano-processor

2.1. Processor Details

2.1.1. Features

Our improved version of the nano processor is programmed using 100% logic gates. (ie: We never used if/when/switch cases during programming, only logic expressions were used.) Every component was designed by fine tuning the logic using Karnaugh-maps. Therefore, every component in the elaborated design schematics can be broken down into 'building unit' logic gates. Hence, this design can be directly used as a blueprint for implementing the nanoprocessor on a silicon board. This is one of the main unique features which makes our design stand-out from the competition.

In addition, our design also contains the following features,

- Register Bank with eight 4-bit registers
- 4-bit Carry Look Ahead Adder/Subtractor
- 4-bit Comparator
- 16-line 14-bit wide Program ROM
- 13 Instructions

2.1.2. Improvements

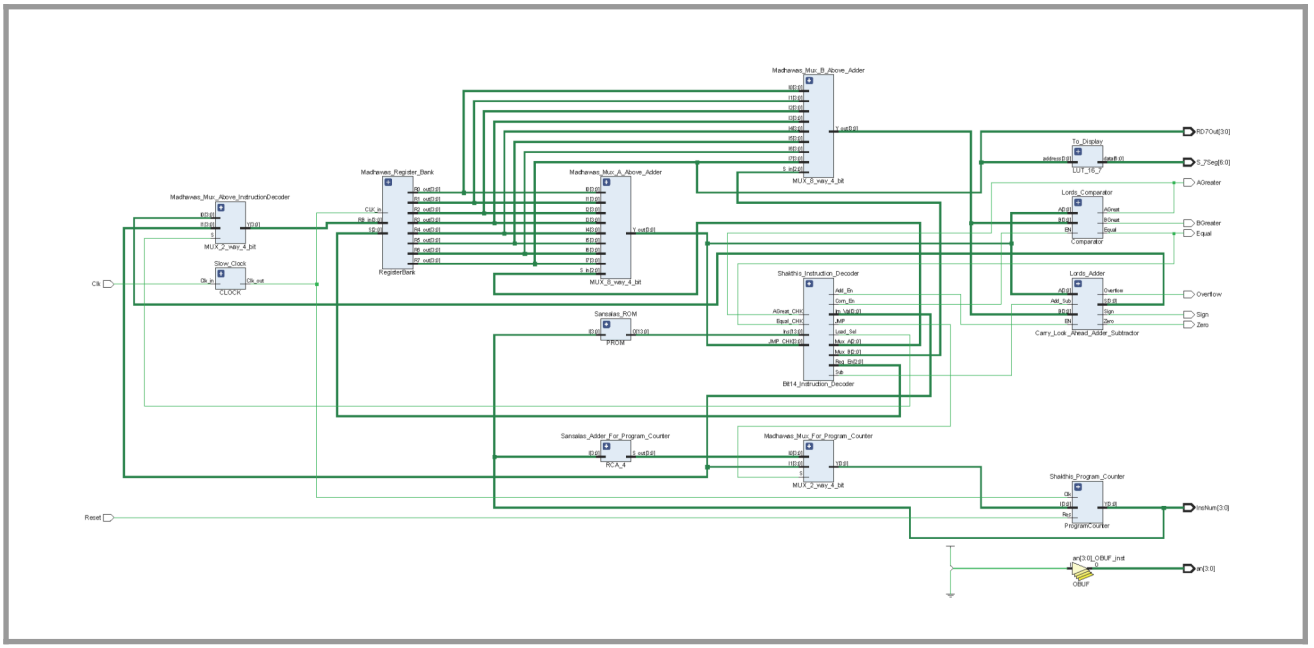
- Features a 4-bit **Carry Look Ahead Adder/Subtractor** for faster computation.
- The **four Comparator instructions**, "COM: Compare A & B", "IFAG: If A is Greater than B, then jump to", "IFE: If A & B are equal, then jump to" and "IFNE: If A & B are not equal, then jump to" gives freedom to the programmer to create a wide range of programmes.
- Features **16 line-Program ROM** to freely write programs.
- Power consumption is minimized in the following ways:
    - Carry Look Ahead Adder will be disabled when not in use.
    - Comparator will be disabled when not in use.

  Their enable inputs will become low when not in use, resulting in low voltages running inside the components.
- Every component was designed by fine tuning the logic using **Karnaugh-maps**. Logic expressions derived from K-maps involve a **minimal number of logic gates**. This results in conserving the following resources.
    - **Space**
    - **Silicon(materials)**
    - **Monetary expense**
    - **Power**

2.1.3. Instruction Set

Op Code	Func	Syntax	Details	Format
0000	ADD	ADD Ra , Rb	Add Ra+Rb & overwrite Ra	0000 RaRaRa RbRbRb 0000
0001	SUB	SUB Ra , Rb	Sub Rb - Ra & overwrite Ra	0001 RaRaRa RbRbRb 0000
0010	MOVI	MOV d to R	Move d to R	0010 R R R 0 0 0 dddd
0011	JZR	JZR R,d	Jump to line d if R=0	0011 R R R 0 0 0 dddd
0100	INC	INC R	Increment R by 1	0100 R R R 0 0 1 0000
0101	DEC	DEC R	Decrement R by 1	0101 R R R 0 1 0 0000
0110	NEG	NEG R	Negate R	0110 R R R 0 0 0 0000
0111	RES	RES R	Reset R to 0	0111 R R R 0 0 0 0000
1000	COM	COM Ra , Rb	Compare Ra & Rb	1000 RaRaRa RbRbRb 0000
1001	NOP	NOP	Do nothing	1001 0 0 0 0 0 0 0000
1010	IFAG	IFAG Ra, Rb, d	If value in Ra>Rb jump to line d	1010 RaRaRa RbRbRb dddd
1011	IFE	IFE Ra,Rb,d	If value in Ra=Rb jump to line d	1011 RaRaRa RbRbRb dddd
1100	IFNE	IFNE Ra, Rb, d	If value in Ra ≠ Rb, jump to line d	1100 RaRaRa RbRbRb dddd

2.1.4. Elaborated Schematic Diagram



- An enlarged image can be found here.

2.2. Slow Clock

2.2.1. Component Details

The Slow Clock slows down the 10 MHz clock signal of the Basys3 board to 1Hz before sending that signal to the components of the nano-processor. It achieves this by toggling its output between 1 and 0 once every 50 million clock cycles of the Basys3 board. This component exists merely for demonstrational purposes and is not a requirement for the functionality of the overall nano-processor. (Therefore, the elaborated design schematic and the timing diagram are not attached for this component.)

2.2.2. VHDL Design Source Code

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

entity CLOCK is
    Port ( Clk_in  : in STD_LOGIC; --clock input signal with frequency f
           Clk_out : out STD_LOGIC); --modified clock output signal with
frequency f/n
end CLOCK;

architecture Behavioral of CLOCK is
--signal count and Clk_status
    signal count : integer := 1;
    signal Clk_status : STD_LOGIC := '0';

begin
    process (Clk_in) begin
    if (rising_edge (Clk_in)) then
        count <= Count +1;
            if (count =50000000) then --change 50000000 to a desired n
value. Then the Clk_out frequency will be 1/n th of Clk_in frequency.
                Clk_status <= not Clk_status;
                Clk_out <= Clk_status;
                count <=1;
             end if;
    end if;
end process;

end Behavioral;
```

2.3. 4-bit Comparator

2.3.1. Component Details

This 4-bit comparator compares two 4-bit binary numbers(from registers) by iterating from the MSB(Most Significant Bit) to the LSB(Least Significant Bit). It has two bus inputs containing the two binary numbers to be compared. It also has an Enable input which makes all inputs and outputs(the three flags) Zero; which saves power when it's turned off (transferring Zero inside logic circuits requires less voltages.) After processing the two numbers, it sets one of the output flags to "1".

The **three output flags** are for the situations where "**A is Great**", "**B is Great**", and "**A and B are Equal**". The Comparator Latch is the logic unit which is used to compare the current two bits and transfer the current state of the comparison to the next step.

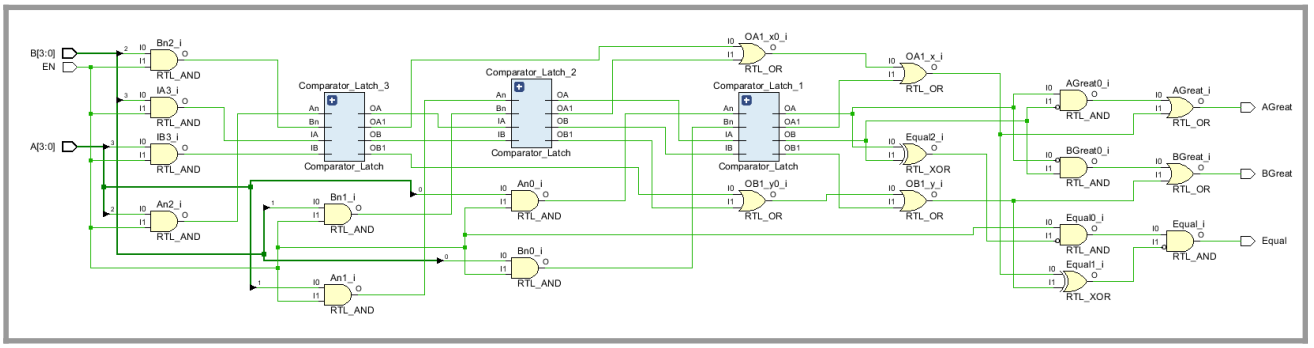
2.3.2. VHDL Design Source Code

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

entity Comparator is
    Port ( EN : in STD_LOGIC; --Enable input
           A : in STD_LOGIC_VECTOR (3 downto 0); --4 bit value: A
           B : in STD_LOGIC_VECTOR (3 downto 0); --4 bit value: B
           AGreat : out STD_LOGIC; --Is A the greater number?
           BGreat : out STD_LOGIC; --Is B the greater number?
           Equal : out STD_LOGIC); --Are the numbers equal?
end Comparator;

architecture Behavioral of Comparator is
component Comparator_Latch
    Port ( An : in STD_LOGIC; --n th bit of number A
       Bn : in STD_LOGIC; --n th bit of number B
       IA : in STD_LOGIC; --input status: A
       IB : in STD_LOGIC; --input status: B
       OA : out STD_LOGIC; --output status: A
       OA1 : out STD_LOGIC; --AGreat Special Case
       OB : out STD_LOGIC; --output status: B
       OB1 : out STD_LOGIC); --BGreat Special Case
end component;

SIGNAL IA3, IA2, IA1, IB3, IB2, IB1 : std_logic; --signal input statuses
of A and B
SIGNAL OA3, OA2, OA1, OB3, OB2, OB1, OA1_3, OA1_2, OA1_1, OB1_3, OB1_2,
OB1_1 : std_logic; --signal output statuses of A and B
SIGNAL An2, An1, An0, Bn2, Bn1, Bn0 : std_logic; --signal 2 downto 0
bits of A and B
SIGNAL OA1_x, OB1_y :std_logic; --special outputs
begin
    Comparator_Latch_3 : Comparator_Latch --mapping the bottom latch
    port map (
```

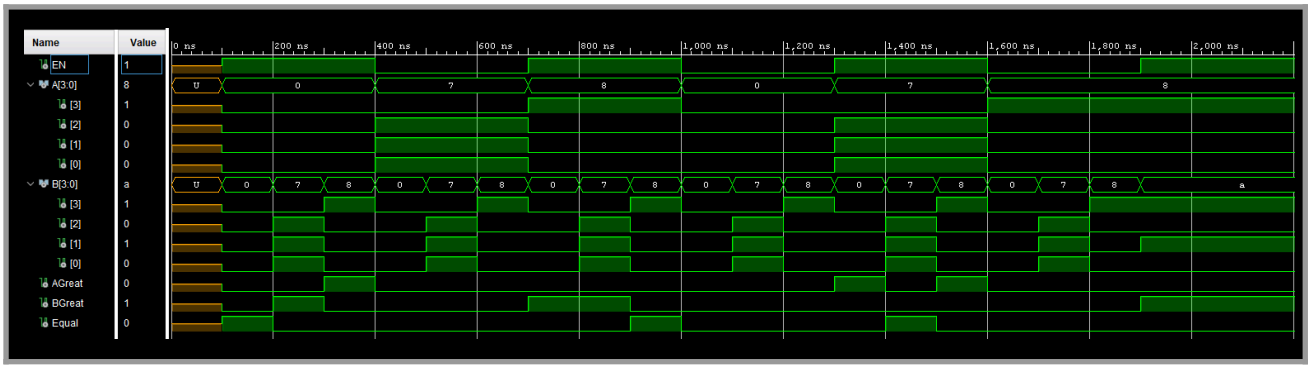
```
        An => An2,
        Bn => Bn2,
        IA => IA3,
        IB => IB3,
        OA => OA3,
        OA1 => OA1_3,
        OB => OB3,
        OB1 => OB1_3
    );
    Comparator_Latch_2 : Comparator_Latch --mapping the middle latch
    port map (
        An => An1,
        Bn => Bn1,
        IA => IA2,
        IB => IB2,
        OA => OA2,
        OA1 => OA1_2,
        OB => OB2,
        OB1 => OB1_2
    );
    Comparator_Latch_1 : Comparator_Latch --mapping the top latch
    port map (
        An => An0,
        Bn => Bn0,
        IA => IA1,
        IB => IB1,
        OA => OA1,
        OA1 => OA1_1,
        OB => OB1,
        OB1 => OB1_1
    );
--Assigning the inputs to the bottom latch
IA3 <= B(3) AND EN;
IB3 <= A(3) AND EN;
An2 <= A(2) AND EN;
Bn2 <= B(2) AND EN;
--Assigning the inputs to the middle latch
IA2 <= OA3;
IB2 <= OB3;
An1 <= A(1) AND EN;
Bn1 <= B(1) AND EN;
--Assigning the inputs to the top latch
IA1 <= OA2;
IB1 <= OB2;
An0 <= A(0) AND EN;
Bn0 <= B(0) AND EN;

--Defining OA1_x and OA1_y for the special cases
OA1_x <= OA1_3 OR OA1_2 OR OA1_1;
OB1_y <= OB1_3 OR OB1_2 OR OB1_1;
--Defining the flags; AGreat, BGreat and Equal
AGreat <= ((OA1 AND NOT(OB1)) OR OA1_x);
BGreat <= ((NOT(OA1) AND OB1) OR OB1_y);
Equal <= EN AND NOT(OA1 XOR OB1) AND NOT(OA1_x XOR OB1_y);
end Behavioral;
```

- The Comparator Latch can be found here.

2.3.3. Elaborated Design Schematic



2.3.4. Timing Diagram



2.4. 4-bit Carry Look Ahead Adder Subtractor

2.4.1. Component Details

The 4-bit Carry Look-Ahead Adder/Subtractor adds/subtracts two 4-bit binary numbers(from registers) by using a carry look-ahead logic which results in a faster processing time. It has two bus inputs containing the two binary numbers to be added/subtracted. Another input is a 1-bit binary value which tells which operation(out of "add" and "sub"; 1 for sub and 0 for add) should be performed. This also has an Enable input which makes all inputs and outputs Zero; which saves power when it's turned off (transferring Zero inside logic circuits requires less voltages.) Finally, after processing the operation, it outputs the 4-bit result into the registers as "S".

There are **three output flags** namely "**overflow**", "**zero**", and "**sign**". The Zero flag will be high when the result is algebraically zero. The Sign flag will be high when the result is algebraically negative. The Overflow flag will be high when the result is not in the decimal range [-8,+7]. As we use carry lookahead logic here, a special type of Full Adder was used to get the Parent and Generation bits.

2.4.2. VHDL Design Source Code

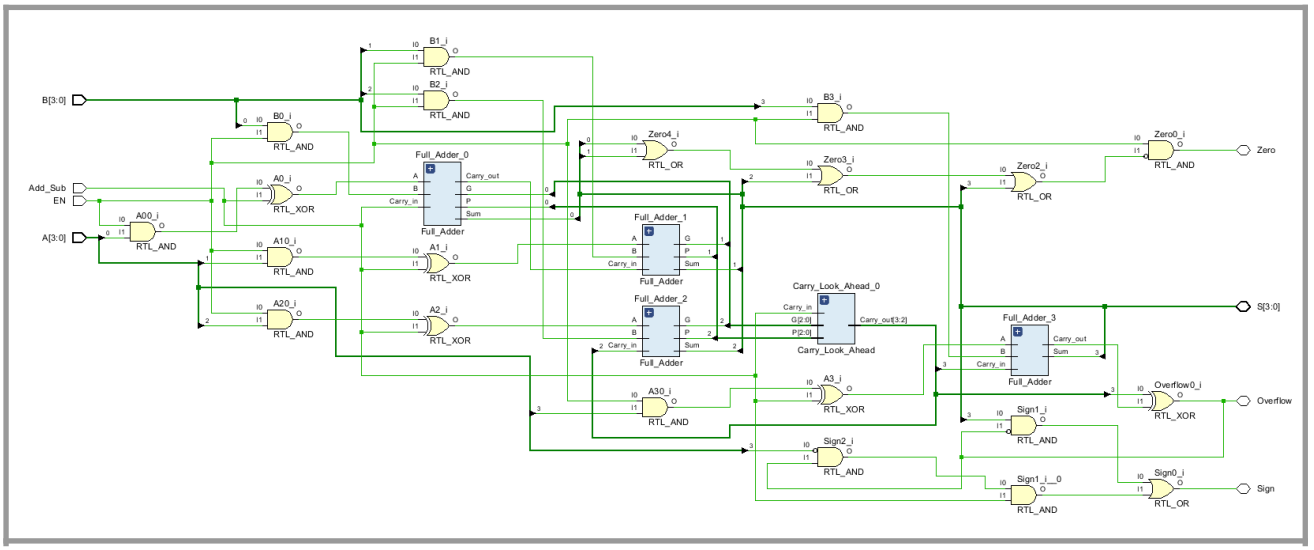
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Carry_Look_Ahead_Adder_Subtractor is
    Port ( A : in STD_LOGIC_VECTOR (3 downto 0); --Bus for the first
binary number
           B : in STD_LOGIC_VECTOR (3 downto 0); --Bus for the second
binary number
           Add_Sub : in STD_LOGIC; --Should we add B to A or should we
subtract B from A? (1 for subtraction, 0 for addition)
           EN : in STD_LOGIC; --Enable input
           S : inout STD_LOGIC_VECTOR (3 downto 0); --Bus for the binary
sum output
           Sign : inout STD_LOGIC; --Sign Bit: 1 for negative, 0 for
positive
           Overflow : inout STD_LOGIC; --Overflow bit: Is there an
overflow?
           --When the number is negative, is the number in range
[-0,-8]: if yes, there is no overflow => Overflow =0; if no, there is an
overflow => Overflow = 1
           --When the number is positive, is the number in range [+0,
7]: if yes, there is no overflow => Overflow =0; if no, there is an
overflow => Overflow = 1
           Zero : inout STD_LOGIC); --Is the output zero(0000) even when
Enable is 1?
end Carry_Look_Ahead_Adder_Subtractor;

architecture Behavioral of Carry_Look_Ahead_Adder_Subtractor is
component Full_Adder is
    Port ( A : in STD_LOGIC; --First input bit
           B : in STD_LOGIC; --Second input bit
```

```
            Carry_in : in STD_LOGIC; --Carry input bit
            Sum : out STD_LOGIC; --Sum output bit
            Carry_out : out STD_LOGIC; --Carry output bit
            P : out STD_LOGIC; --Propergate output bit
            G : out STD_LOGIC); --Generate output bit
end component;

component Carry_Look_Ahead is
    Port ( P : in STD_LOGIC_VECTOR (2 downto 0); --Propergate input bus
           G : in STD_LOGIC_VECTOR (2 downto 0); --Propergate output bus
           Carry_in : in STD_LOGIC; --Carry input bit
           Carry_out : out STD_LOGIC_VECTOR (3 downto 2)); --Carry
output bus
end component;

--Signal the inputs and outputs of internal components
SIGNAL A0, B0, C1, P0, G0: std_logic; --FA0
SIGNAL A1, B1, P1, G1: std_logic; --FA1
SIGNAL A2, B2, C2, P2, G2: std_logic; --FA2
SIGNAL A3, B3, C3, C4: std_logic; --FA3

--An and Bn are n th bits of the binary numbers A and B inserted to the
relevant(n th) Full Adder
--Cn is the carry bit generated by adding the n th bits of A and B
--Pn and Gn are propergation and generation bits of the relevant(n th)
Full Adder
begin
    Full_Adder_0 : Full_Adder --mapping first full adder
        port map(A => A0,
        B => B0,
        Carry_in => Add_Sub, --C0
        Sum => S(0),
        Carry_out => C1,
        P => P0,
        G => G0
    );
    Full_Adder_1 :Full_Adder --mapping second full adder
        port map(A => A1,
        B => B1,
        Carry_in => C1,
        Sum => S(1),
        P => P1,
        G => G1
    );
    Full_Adder_2 :Full_Adder --mapping third full adder
        port map(A => A2,
        B => B2,
        Carry_in => C2,
        Sum => S(2),
        P => P2,
        G => G2
    );
    Full_Adder_3 :Full_Adder --mapping last full adder
        port map(A => A3,
        B => B3,
        Carry_in => C3,
        Sum => S(3),
```

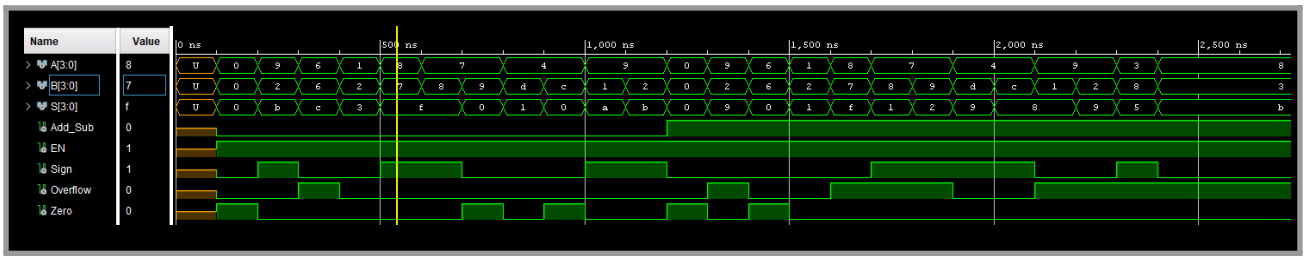
```
        Carry_out => C4
    );
    Carry_Look_Ahead_0 : Carry_Look_Ahead --mapping carry look ahead
logic unit
        port map(P(0) => P0, P(1) => P1, P(2) => P2,
        G(0) => G0, G(1) => G1, G(2) => G2,
        Carry_in => Add_Sub, --C0
        Carry_out(2) => C2, Carry_out(3) => C3
    );

--Defining inputs of FA0 (A0, B0)
A0 <= (EN AND A(0)) XOR Add_Sub;
B0 <= B(0) AND EN;

--Defining inputs of FA1 (A1, B1)
A1 <= (EN AND A(1)) XOR Add_Sub;
B1 <= B(1) AND EN;

--Defining inputs of FA2 (A2, B2)
A2 <= (EN AND A(2)) XOR Add_Sub;
B2 <= B(2) AND EN;

--Defining inputs of FA3 (A3, B3)
A3 <= (EN AND A(3)) XOR Add_Sub;
B3 <= B(3) AND EN;

--Defining the zero flag
Zero <= EN AND NOT(S(0) OR S(1) OR S(2) OR S(3));

--carry output of the Carry Look Ahead Adder Subtractor
Sign <= (S(3) AND NOT(Overflow)) OR (NOT(A(3)) AND Overflow AND
Add_Sub);

--Defining the overflow bit
Overflow <= C3 XOR (C4 );

end Behavioral;
```

- Full Adder for the 4-bit Carry Look Ahead Adder can be found here.
- The Carry Look Ahead Logic Unit can be found here.
- Half Adder can be found here.

2.4.3. Elaborated Design Schematic



2.4.4. Timing Diagram



2.5. 4-bit Adder for Program Counter

2.5.1. Component Details

The task of the 4-bit adder in the nanoprocessor involves incrementing the program counter by 1 after each instruction execution. This is typically achieved using an RCA circuit, which adds 1 to the current value stored in the program counter register. The design of this adder has optimized for space, and power efficiency to ensure reliable operation within the nanoprocessor's constraints.

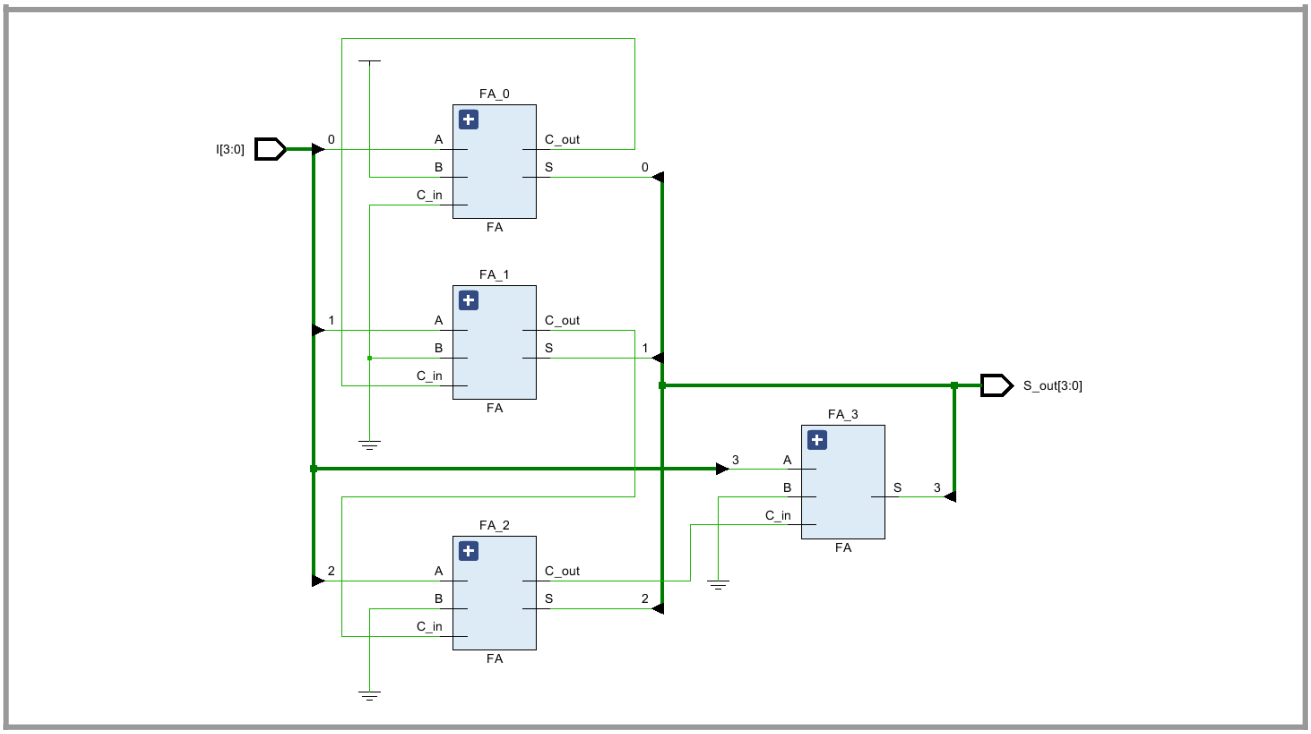
2.5.2. VHDL Design Source Codes

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

entity RCA_4 is
    Port ( I : in STD_LOGIC_VECTOR (3 downto 0);
           S_out : out STD_LOGIC_VECTOR (3 downto 0));
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,FA1_S,FA2_S,FA3_S,FA0_C,FA1_C,FA2_C,Carry : std_logic;
    SIGNAL S : std_logic_vector(3 downto 0);

begin

    FA_0 : FA
    port map(
        A => I(0),
        B => '1',
        C_in => '0',
        S => S(0),
        C_out => FA0_C );

    FA_1 : FA
    port map(
            A => I(1),
            B => '0',
            C_in => FA0_C,
            S => S(1),
            C_out => FA1_C );

    FA_2 : FA
     port map(
              A => I(2),
              B => '0',
```

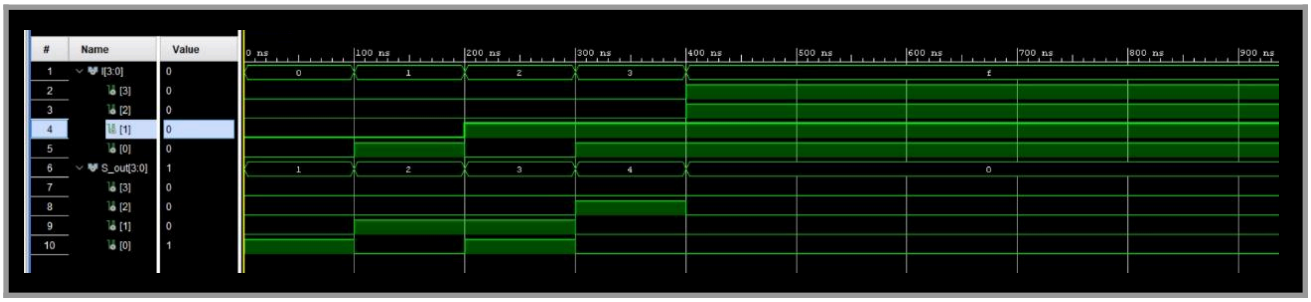
```
                C_in => FA1_C,
                S => S(2),
                C_out => FA2_C );
    FA_3 : FA
      port map(
                 A => I(3),
                 B => '0',
                 C_in => FA2_C,
                 S => S(3),
                 C_out => Carry );
     S_out <= S;
end Behavioral;
```

- Full Adder can be found here.
- Half Adder can be found here.

2.5.3. Elaborated Design Schematic



2.5.4. Timing Diagram



2.6. 4-bit Program Counter

2.6.1. Component Details

This component is a register capable of storing memory addresses up to 2^4 (or 16) distinct locations. It tracks the address of the next instruction to be fetched and executed.

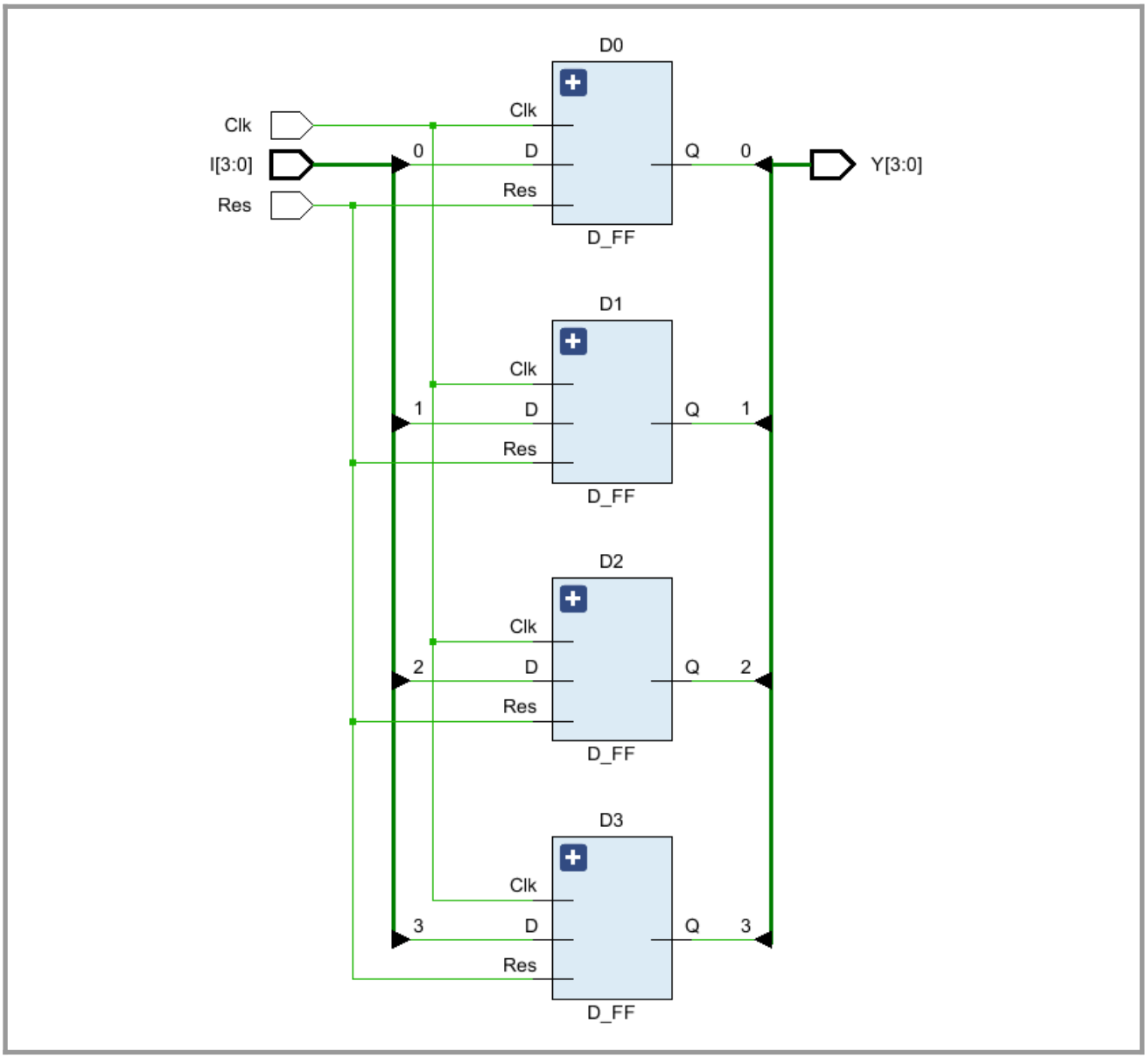
2.6.2. VHDL Design Source Code

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

entity ProgramCounter is
    Port ( Clk : in STD_LOGIC;
    Res : in std_logic;
           I : in STD_LOGIC_VECTOR (3 downto 0);
           Y : out STD_LOGIC_VECTOR (3 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

D0 :  D_FF
port map(
    D => I(0),
    Res => Res,
    Clk => Clk,
    Q => Y(0),
    Qbar => open);

D1 :  D_FF
port map(
    D => I(1),
    Res => Res,
    Clk => Clk,
    Q => Y(1),
    Qbar => open);

D2 :  D_FF
port map(
    D => I(2),
    Res => Res,
    Clk => Clk,
    Q => Y(2),
    Qbar => open);
```

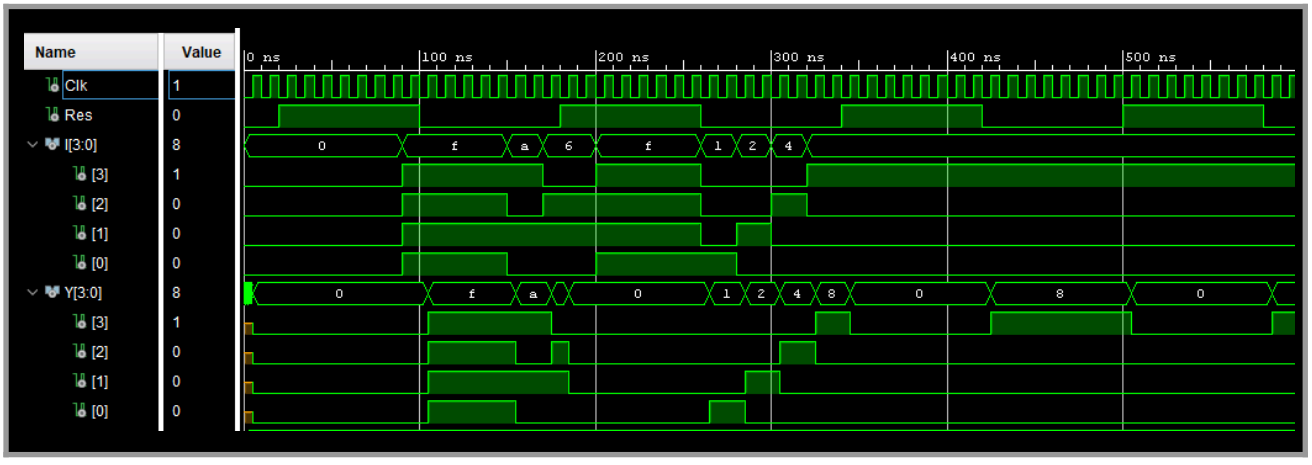
```
D3 :  D_FF
    port map(
        D => I(3),
        Res => Res,
        Clk => Clk,
        Q => Y(3),
        Qbar => open);

end Behavioral;
```

- D-flip Flop can be found here.

2.6.3. Elaborated Design Schematic



2.6.4 Timing Diagram



2.7. 2-way 4-bit Multiplexer

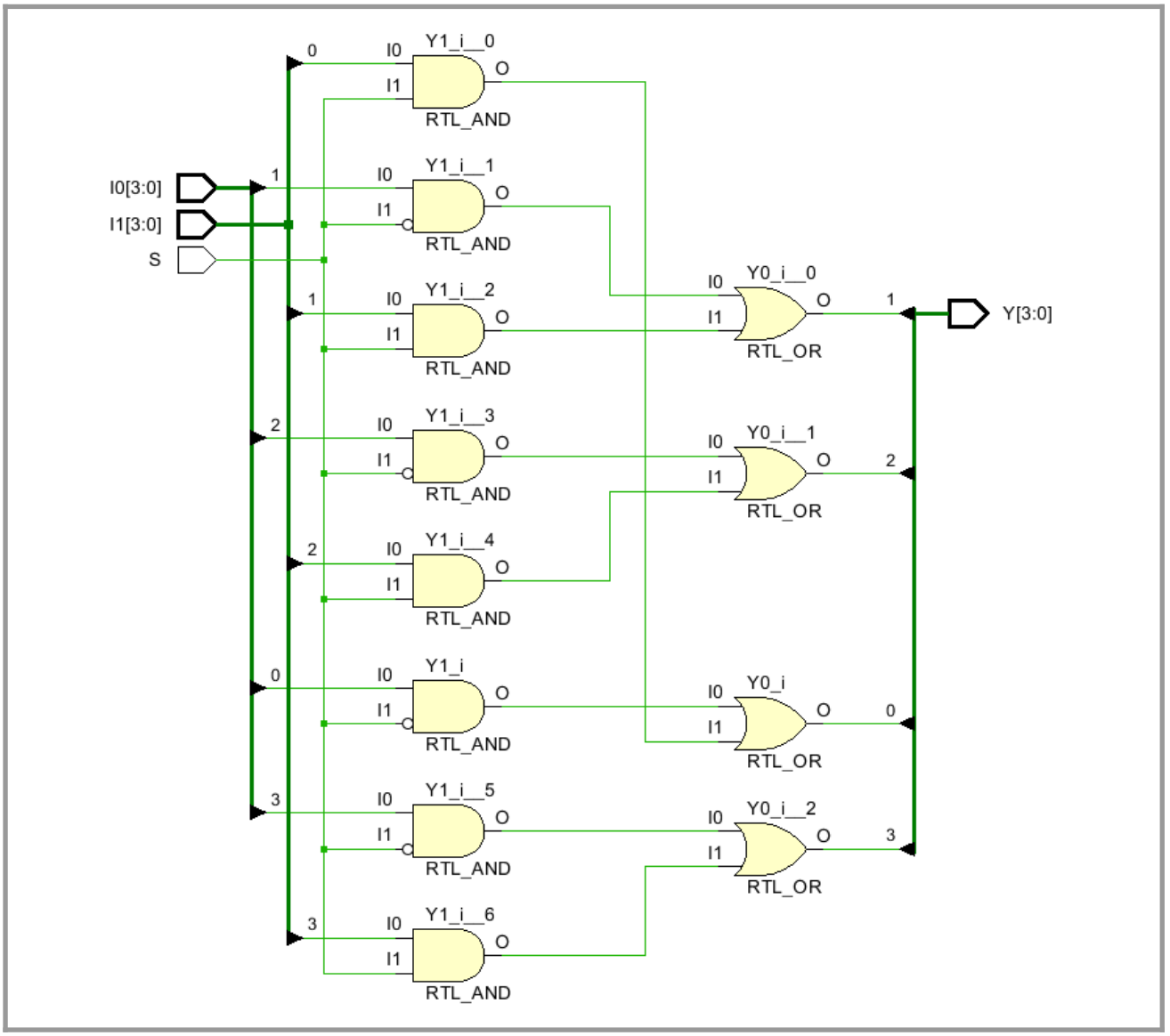
2.7.1. Component Details

This multiplexer serves the function of selecting one of two 4-bit input data sources (I0 and I1) based on the value of the select signal (S). The selected data is then routed to the output Y.

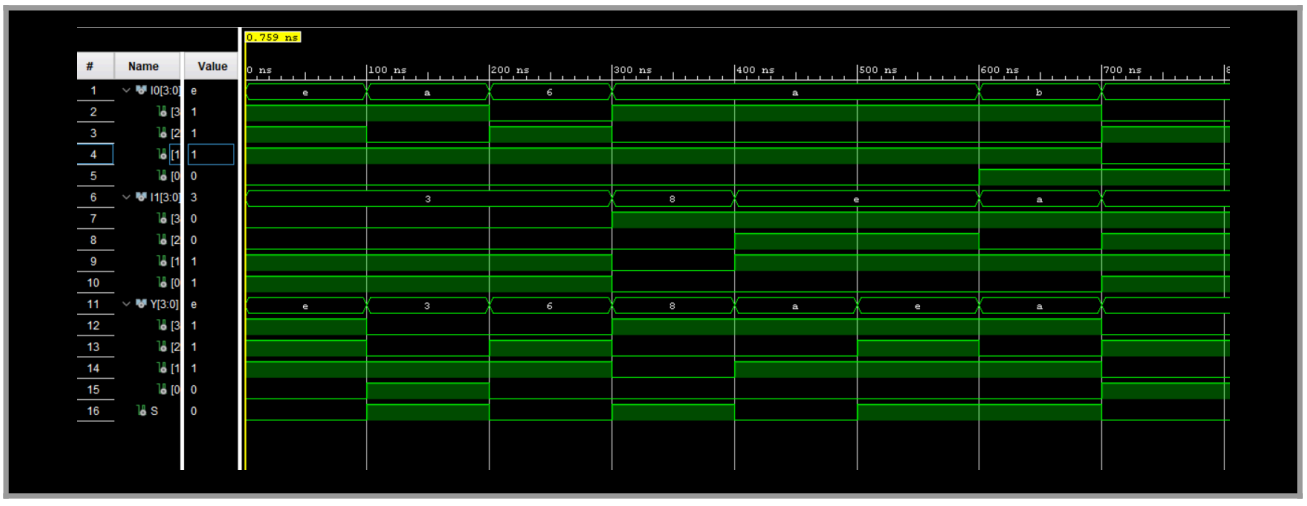
2.7.2. VHDL Design Source Code

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

entity MUX_2_way_4_bit is
    Port ( I0 : in STD_LOGIC_VECTOR (3 downto 0);
           I1 : in STD_LOGIC_VECTOR (3 downto 0);
           S : in STD_LOGIC;
           Y : out STD_LOGIC_VECTOR (3 downto 0));
end MUX_2_way_4_bit;

architecture Behavioral of MUX_2_way_4_bit is

begin

Y(0) <= ( I0(0) AND NOT S ) OR ( I1(0) AND S );
Y(1) <= ( I0(1) AND NOT S ) OR ( I1(1) AND S );
Y(2) <= ( I0(2) AND NOT S ) OR ( I1(2) AND S );
Y(3) <= ( I0(3) AND NOT S ) OR ( I1(3) AND S );

end Behavioral;
```

2.7.3. Elaborated Design Schematic



2.7.4. Timing Diagram



2.8. 8-way 4-bit Multiplexer

2.8.1. Component Details

The MUX_8_way_4_bit represents an 8-way, 4-bit multiplexer, which selects between eight sets of input vectors (I0 through I7) based on a 3-bit select signal (S_in). The selected output is denoted by Y_out.

Within its architecture, it uses an 8-to-1 multiplexer (MUX_8_to_1) component four times, one for each bit of the input vectors. Each instance of the MUX_8_to_1 component selects the appropriate bit from the eight input vectors based on the select signal S_in, and routes it to the corresponding bit of the output vector Y_out.

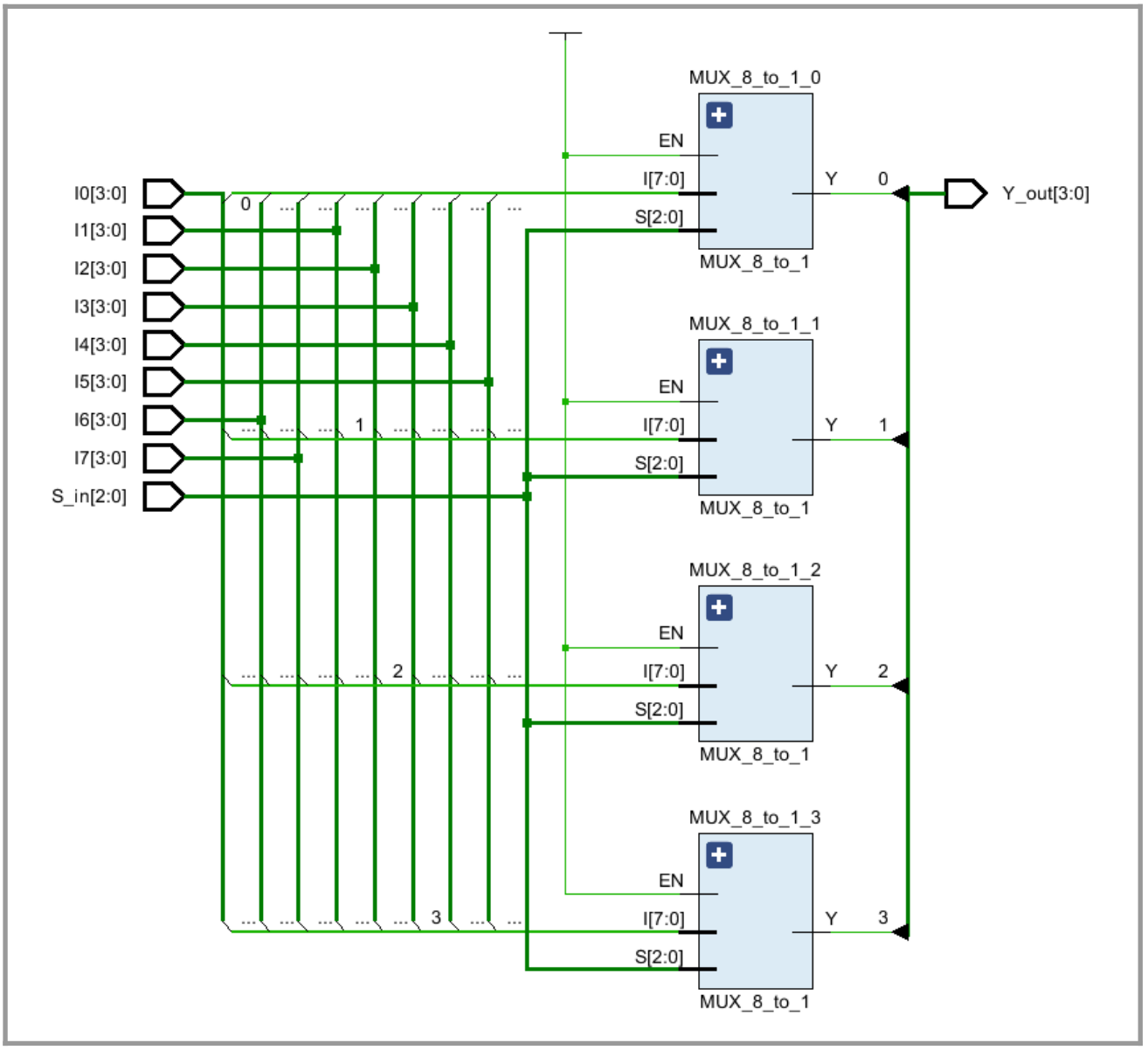
2.8.2. VHDL Design Source Code

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

entity MUX_8_way_4_bit is
    Port ( I0 : in STD_LOGIC_VECTOR (3 downto 0);
           I1 : in STD_LOGIC_VECTOR (3 downto 0);
           I2 : in STD_LOGIC_VECTOR (3 downto 0);
           I3 : in STD_LOGIC_VECTOR (3 downto 0);
           I4 : in STD_LOGIC_VECTOR (3 downto 0);
           I5 : in STD_LOGIC_VECTOR (3 downto 0);
           I6 : in STD_LOGIC_VECTOR (3 downto 0);
           I7 : in STD_LOGIC_VECTOR (3 downto 0);
           S_in : in STD_LOGIC_VECTOR (2 downto 0);
           Y_out : out STD_LOGIC_VECTOR (3 downto 0));
end MUX_8_way_4_bit;

architecture Behavioral of MUX_8_way_4_bit is

component MUX_8_to_1
    Port ( I : in STD_LOGIC_VECTOR (7 downto 0);
           S : in STD_LOGIC_VECTOR (2 downto 0);
           EN : in STD_LOGIC;
           Y : out STD_LOGIC);
end component;

--signal A, B, C, D : STD_LOGIC;

begin

MUX_8_to_1_0 : MUX_8_to_1
    PORT MAP(
        I(0) => I0(0),
        I(1) => I1(0),
        I(2) => I2(0),
        I(3) => I3(0),
        I(4) => I4(0),
        I(5) => I5(0),
```

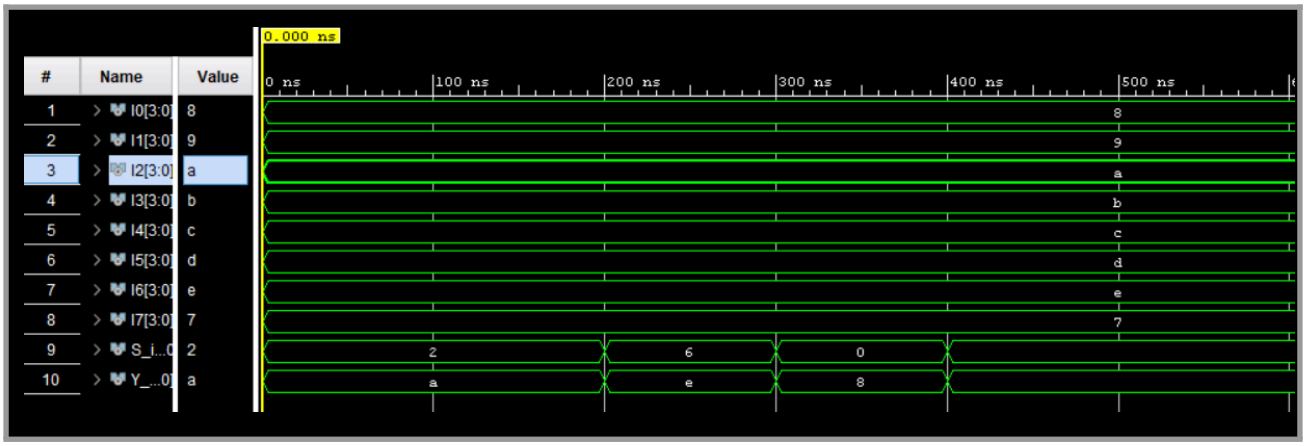
```
        I(6) => I6(0),
        I(7) => I7(0),
        S => S_in,
        Y => Y_out(0),
        EN => '1'
        );

MUX_8_to_1_1 : MUX_8_to_1
    PORT MAP(
        I(0) => I0(1),
        I(1) => I1(1),
        I(2) => I2(1),
        I(3) => I3(1),
        I(4) => I4(1),
        I(5) => I5(1),
        I(6) => I6(1),
        I(7) => I7(1),
        S => S_in,
        Y => Y_out(1),
        EN => '1'
        );

MUX_8_to_1_2 : MUX_8_to_1
    PORT MAP(
        I(0) => I0(2),
        I(1) => I1(2),
        I(2) => I2(2),
        I(3) => I3(2),
        I(4) => I4(2),
        I(5) => I5(2),
        I(6) => I6(2),
        I(7) => I7(2),
        S => S_in,
        Y => Y_out(2),
        EN => '1'
        );

MUX_8_to_1_3 : MUX_8_to_1
    PORT MAP(
        I(0) => I0(3),
        I(1) => I1(3),
        I(2) => I2(3),
        I(3) => I3(3),
        I(4) => I4(3),
        I(5) => I5(3),
        I(6) => I6(3),
        I(7) => I7(3),
        S => S_in,
        Y => Y_out(3),
        EN => '1'
        );

end Behavioral;
```

- MUX_8_to_1 can be found here.
- Decoder_3_to_8 can be found here.
- Decoder_2_to_4 can be found here.

2.8.3. Elaborated Design Schematic



2.8.4. Timing Diagram



2.9. Register Bank

2.9.1. Component Details

The RegisterBank represents a group of registers, capable of storing data simultaneously. It features eight 4-bit registers (R0_out through R7_out), each potentially holding different data based on the select signal S.

The architecture employs two components: a 3-to-8 decoder (Decoder_3_to_8) and a 4-bit register (Register_4_bit). The decoder translates the 3-bit select signal S into eight individual enable signals, which determine which registers will receive data updates. Each register (Register_4_bit) is controlled by a corresponding enable signal generated by the decoder. When enabled, the register either receives the data from the input (RB_in) or maintains its current value, depending on the control signal EN and clock signal CLK_in.

2.9.2. VHDL Design Source Code

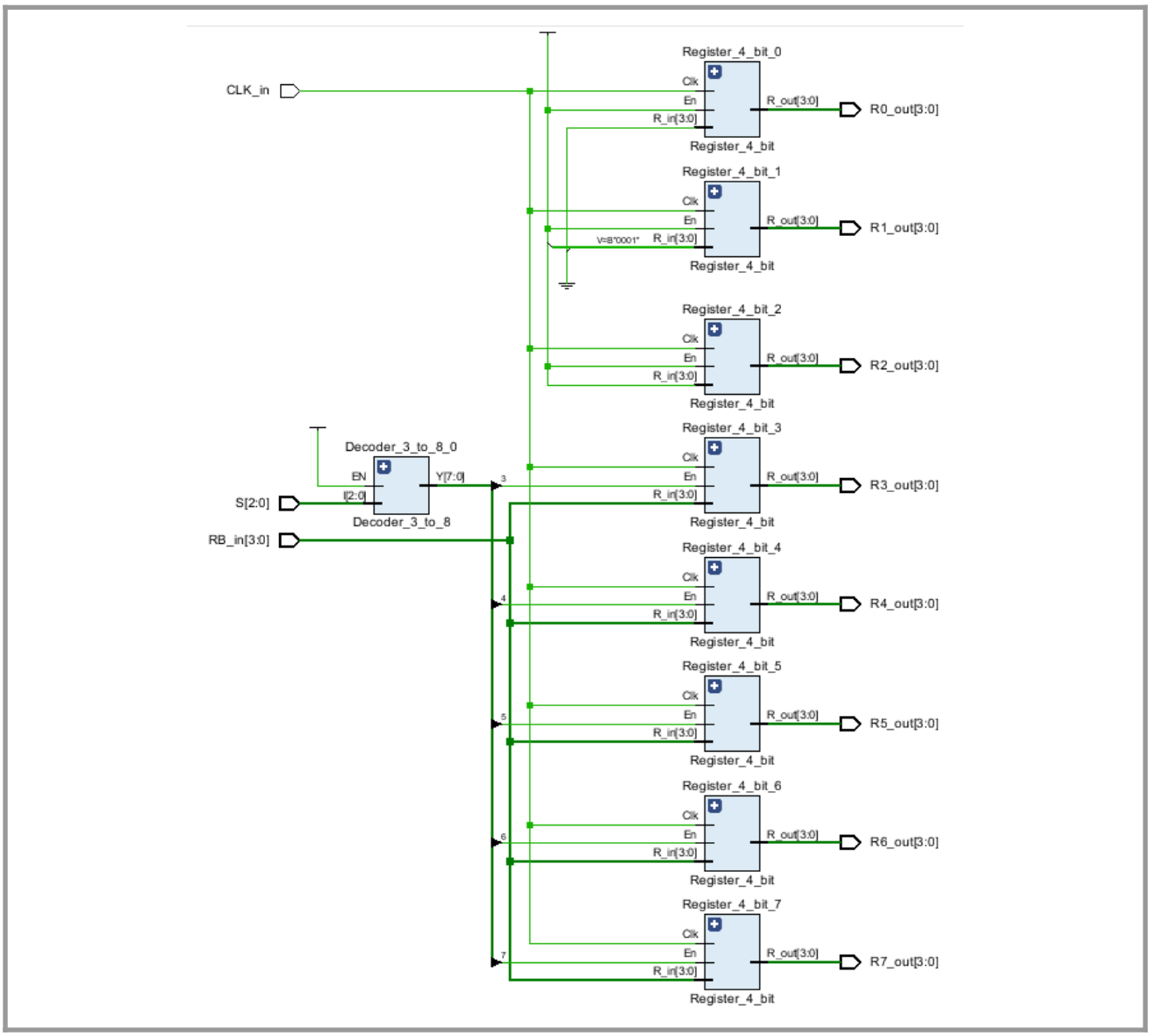
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity RegisterBank is
    Port ( S : in STD_LOGIC_VECTOR (2 downto 0);
           RB_in : in STD_LOGIC_VECTOR (3 downto 0);
           CLK_in : in STD_LOGIC;
           R0_out : out STD_LOGIC_VECTOR (3 downto 0);
           R1_out : out STD_LOGIC_VECTOR (3 downto 0);
           R2_out : out STD_LOGIC_VECTOR (3 downto 0);
           R3_out : out STD_LOGIC_VECTOR (3 downto 0);
           R4_out : out STD_LOGIC_VECTOR (3 downto 0);
           R5_out : out STD_LOGIC_VECTOR (3 downto 0);
           R6_out : out STD_LOGIC_VECTOR (3 downto 0);
           R7_out : out STD_LOGIC_VECTOR (3 downto 0));
end RegisterBank;

architecture Behavioral of RegisterBank is

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;

component Register_4_bit
    Port ( R_in : in STD_LOGIC_VECTOR (3 downto 0);
           EN : in STD_LOGIC;
           CLK : in STD_LOGIC;
           R_out : out STD_LOGIC_VECTOR (3 downto 0));
end component;

signal tempEN : STD_LOGIC_VECTOR (7 downto 0);

begin
```

```
Decoder_3_to_8_0 : Decoder_3_to_8
    port map(
        I => S,
        EN => '1',
        Y => tempEN);

Register_4_bit_0 : Register_4_bit
    port map(
        R_in => "0000",
        EN => '1',
        CLK => CLK_in,
        R_out => R0_out);

Register_4_bit_1 : Register_4_bit
    port map(
        R_in => "0001",
        EN => '1',
        CLK => CLK_in,
        R_out => R1_out);

--Register_4_bit_1 : Register_4_bit
--    port map(
--        R_in => RB_in,
--        EN => tempEN(1),
--        CLK => CLK_in,
--        R_out => R1_out);

Register_4_bit_2 : Register_4_bit
    port map(
        R_in => "1111",
        EN => '1',
        CLK => CLK_in,
        R_out => R2_out);

Register_4_bit_3 : Register_4_bit
    port map(
        R_in => RB_in,
        EN => tempEN(3),
        CLK => CLK_in,
        R_out => R3_out);

Register_4_bit_4 : Register_4_bit
    port map(
        R_in => RB_in,
        EN => tempEN(4),
        CLK => CLK_in,
        R_out => R4_out);

Register_4_bit_5 : Register_4_bit
    port map(
        R_in => RB_in,
        EN => tempEN(5),
        CLK => CLK_in,
        R_out => R5_out);

Register_4_bit_6 : Register_4_bit
```

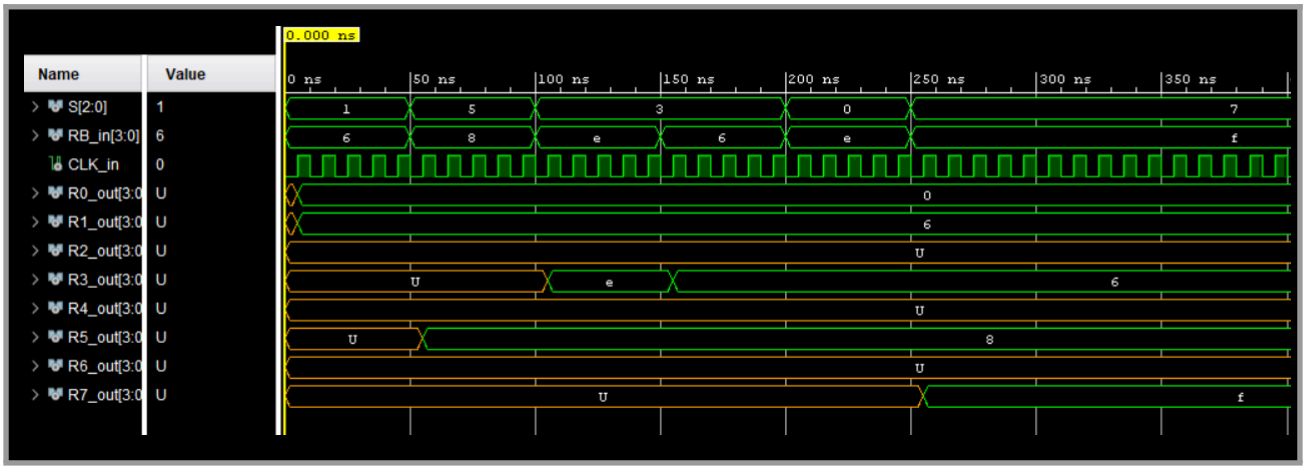
```
    port map(
        R_in => RB_in,
        EN => tempEN(6),
        CLK => CLK_in,
        R_out => R6_out);

Register_4_bit_7 : Register_4_bit
    port map(
        R_in => RB_in,
        EN => tempEN(7),
        CLK => CLK_in,
        R_out => R7_out);

end Behavioral;
```

- Decoder_3_to_8 can be found here.
- Register_4_bit can be found here.

2.9.3. Elaborated Design Schematic



2.9.4. Timing Diagram



2.10. Program ROM

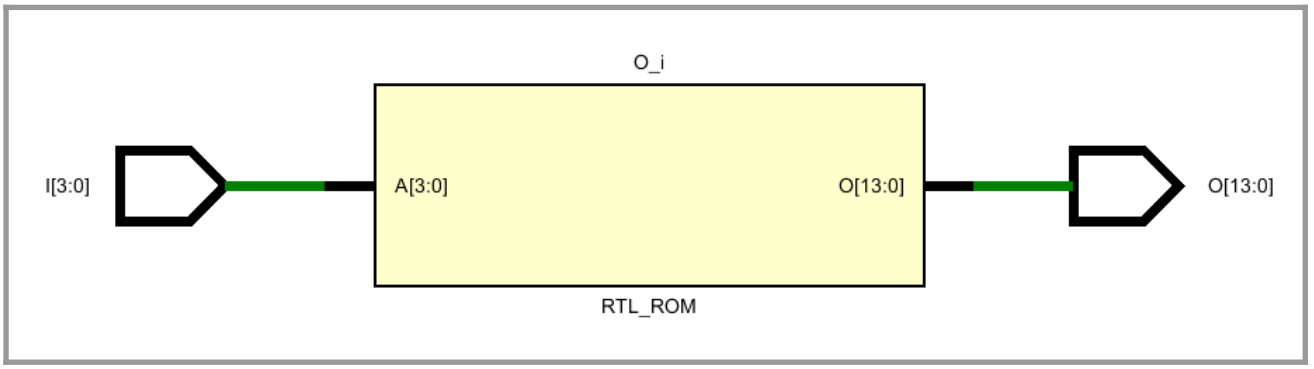
2.10.1 Component Details

This component contains the instructions necessary to run the Nano-processor. It is implemented using a LUT.

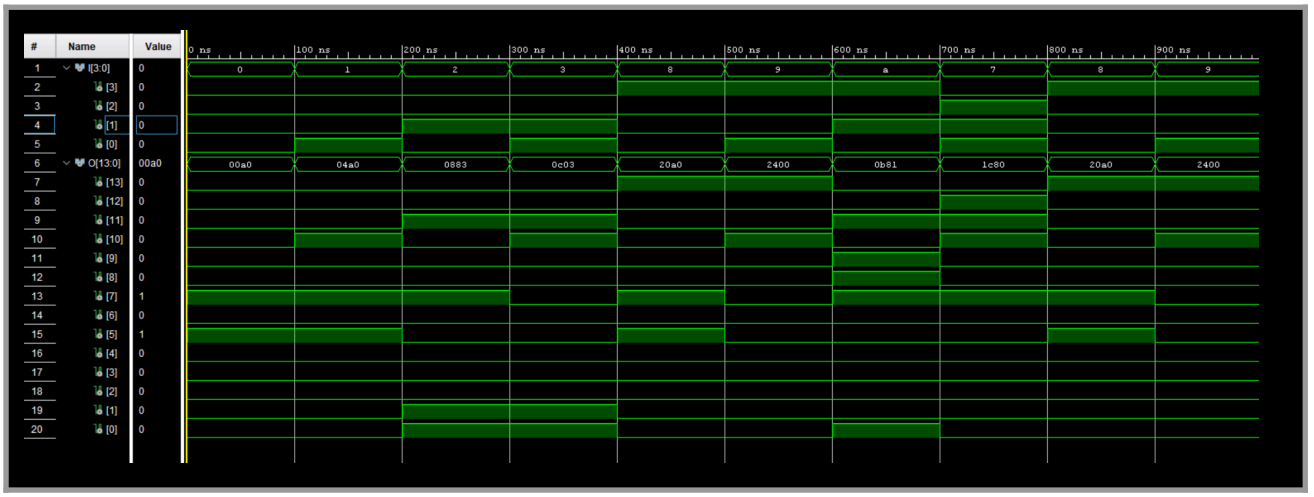
2.10.2. VHDL Design Source Code

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

entity PROM is
    Port ( I : in STD_LOGIC_VECTOR (3 downto 0);
           O : out STD_LOGIC_VECTOR (13 downto 0));
end PROM;

architecture Behavioral of PROM is

type rom_type is array (0 to 15) of std_logic_vector (13 downto 0);
    signal Program_ROM : rom_type :=(
    "00101110000000", --MOVI R7, 0
    "00101000000110", --MOVI R4, 6
    "00101110000001", --MOVI R7, 1
    "00101100000000", --MOVI R6, 0
    "00001111100000", --ADD R7, R6
    "01011000100000", --DEC R4
    "00011101110000", --SUB R6, R7
    "00111000001001", --JZR R4, 10
    "00110000000100", --JZR R0, 5
    "00110000001001", --JZR R0, 10
    "10010000000000", --FILLED IN WITH NOP
    "10010000000000", --FILLED IN WITH NOP
    "10010000000000", --FILLED IN WITH NOP
    "10010000000000", --FILLED IN WITH NOP
    "10010000000000", --FILLED IN WITH NOP
    "10010000000000" --FILLED IN WITH NOP

        );

begin

O <= program_ROM(to_integer(unsigned(I)));

end Behavioral;
```

2.10.3. Elaborated Design Schematic



2.10.4.Timing Diagram



2.11. LUT for 7-segment Display

2.11.1. Component Details

This LUT stores the bits that tell which segments of the 7-Segment Display on the Basys3 board should light up according to the data in Register 7 of the Register Bank.

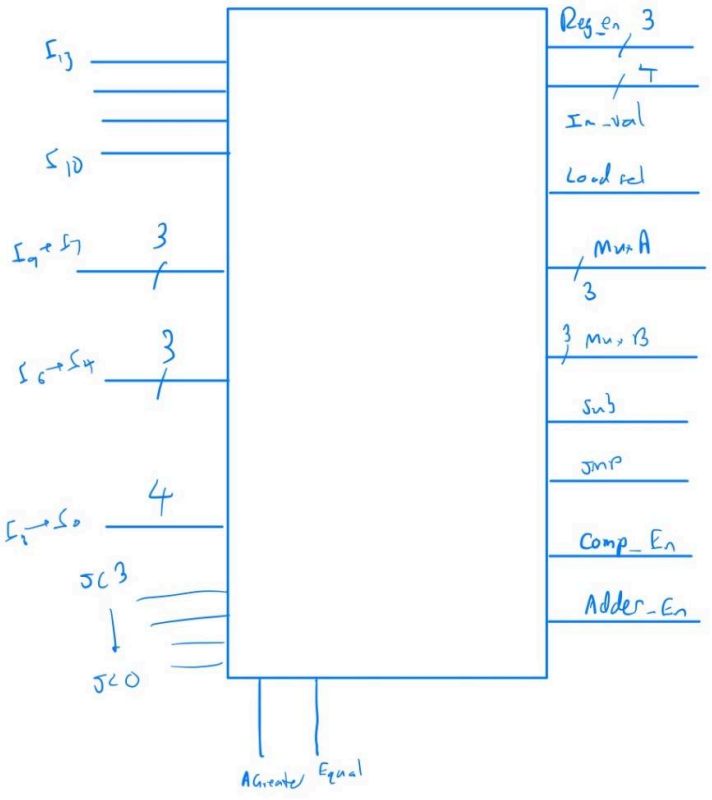
2.11.2. VHDL Design Source Code

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

entity LUT_16_7 is
    Port ( address : in STD_LOGIC_VECTOR (3 downto 0);
           data : out STD_LOGIC_VECTOR (6 downto 0));
end LUT_16_7;

architecture Behavioral of LUT_16_7 is

    type rom_type is array (0 to 15) of std_logic_vector(6 downto 0);

    signal sevenSegment_ROM : rom_type := (
                              "1000000",--0
                              "1111001",--1
                              "0100100",--2
                              "0110000",--3
                              "0011001",--4
                              "0010010",--5
                              "0000010",--6
                              "1111000",--7
                              "0000000",--8
                              "0010000",--9
                              "0001000",--a
                              "0000011",--b
                              "1000110",--c
                              "0100001",--d
                              "0000110",--e
                              "0001110"--f
);

begin

data <= sevenSegment_ROM(to_integer(unsigned(address)));

end Behavioral;
```

2.12. Instruction Decoder

2.12.1. Component Details

The instruction decoder is the main component that brings all the other components together.
It takes a 14-Bit instruction provided by the Program ROM. It is built purely out of logic gates. No If-Else statements exist in its design.



It consists the following inputs:

- I13 through I0 : The bits of the instruction coming from the Program ROM
- JC3 through JC0 : Bits from Mux_A used to check the Jump instruction
- AGreater : The AGreater output from the Comparator used for the IFAG instruction
- Equal: The Equal output from the Comparator used for the IFE and IFNE instructions

And the following outputs:

- Reg_en : Decides which register in the Register Bank is to be enabled for writing data.
- Im_Val : Holds the 4 LSBs of the instruction to be sent to the Mux above the decoder.
- Load_Sel : Chooses whether to send the Im_Val or Adder output data to the registers.
- Mux_A, Mux_B : Tells the Multiplexers above the Adder which registers to get data from.
- Sub : Tells the Adder which operation needs to be done(add or subtract.)
- JMP : Tells the Multiplexer for the Program Counter to choose between the 3-Bit Adder or Jump Address (Im_Val2 through Im_Val0)
- Adder_En, Comp_En: Disables the Adder and Comparator when not in use.

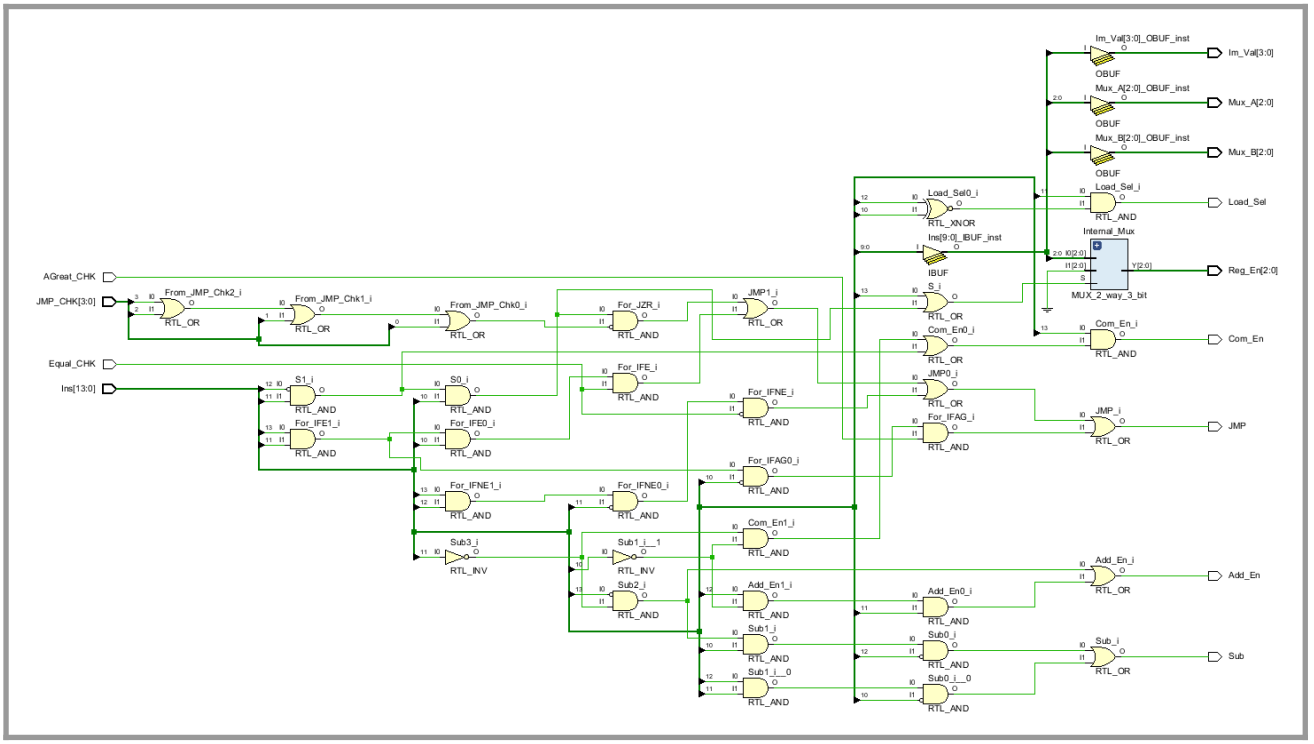
2.12.2. VHDL Design Source Code

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

entity Bit14_Instruction_Decoder is
    Port ( Ins : in STD_LOGIC_VECTOR (13 downto 0);
           JMP_CHK : in STD_LOGIC_VECTOR (3 downto 0);
           AGreat_CHK : in std_logic;
           Equal_CHK : in std_logic;
           Reg_En : out STD_LOGIC_VECTOR (2 downto 0);
           Im_Val : out STD_LOGIC_VECTOR (3 downto 0);
           Load_Sel : out STD_LOGIC;
           Mux_A : out STD_LOGIC_VECTOR (2 downto 0);
           Mux_B : out STD_LOGIC_VECTOR (2 downto 0);
           Sub : out STD_LOGIC;
           JMP : out STD_LOGIC;
           Com_En : out STD_LOGIC;
           Add_En : out STD_LOGIC);
end Bit14_Instruction_Decoder;

architecture Behavioral of Bit14_Instruction_Decoder is

component MUX_2_way_3_bit
Port ( I0 : in STD_LOGIC_VECTOR (2 downto 0);
           I1 : in STD_LOGIC_VECTOR (2 downto 0);
           S : in STD_LOGIC;
           Y : out STD_LOGIC_VECTOR (2 downto 0));
end component;

signal S, From_JMP_Chk: std_logic;
signal For_IFE, For_IFNE, For_IFAG, For_JZR: std_logic;

begin

Add_En <= ((not Ins(13)) and (not Ins(11))) or (Ins(12) and (not
Ins(10)) and Ins(11));
Com_En <= Ins(13) and (((not Ins(11)) and (not Ins(10))) or ((not
Ins(12)) and (Ins(11))));

Sub <= ((not Ins(13)) and (not Ins(11)) and Ins(10) and(not Ins(12))) or
(Ins(12) and Ins(11) and (not Ins(10)));
Load_Sel <= Ins(11) and (Ins(12) xnor Ins(10));

From_JMP_Chk <= not(JMP_CHK(3) or JMP_CHK(2) or JMP_CHK(1) or
JMP_CHK(0));
For_JZR <= ((not Ins(12)) and Ins(11) and Ins(10)) and From_JMP_CHK;
For_IFE <= Ins(13) and Ins(11) and Ins(10) and Equal_CHK;
For_IFNE <= Ins(13) and Ins(12) and (not Ins(11)) and (not Equal_CHK);
For_IFAG <= Ins(13) and Ins(11) and (not Ins(10)) and AGreat_CHK;

JMP <= For_JZR or For_IFE or For_IFNE or For_IFAG;

S <= Ins(13) or ((not Ins(12)) and Ins(11) and Ins(10));

Internal_Mux : MUX_2_way_3_bit
```

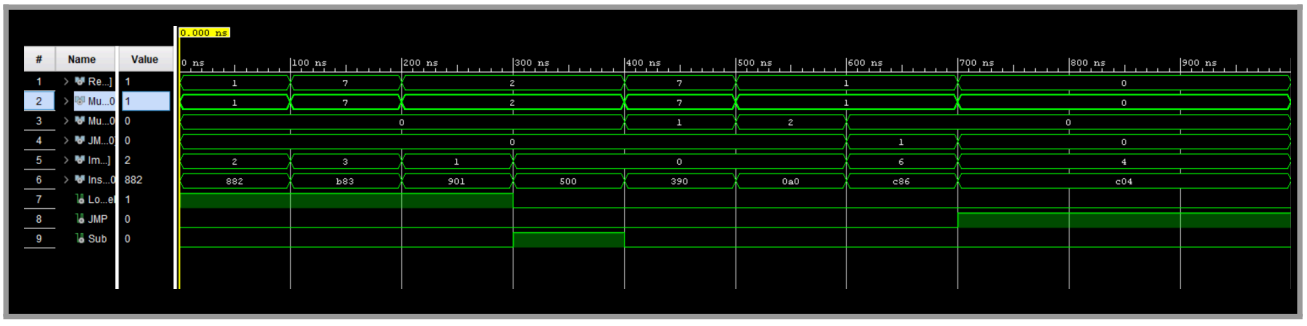
```
Port map( I0 => Ins(9 downto 7),
          I1 => "000",
          S => S,
          Y => Reg_En);

Im_Val <= Ins(3 downto 0);
Mux_A <= Ins(9 downto 7);
Mux_B <= Ins(6 downto 4);

end Behavioral;
```

- The 2-way 3-bit MUX can be found here.

2.12.3. Elaborated Design Schamatic



2.12.3. Timing Diagram



2.13. Nano-processor

2.13.1. Component Details

The final Nano-processor that uses all the above mentioned components.

2.13.2. VHDL Design Source Code

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

entity Nanoprocessor2 is
    Port ( Clk : in STD_LOGIC;
           Reset : in STD_LOGIC;
           RD7Out : out STD_LOGIC_VECTOR (3 downto 0);
           S_7Seg : out STD_LOGIC_VECTOR (6 downto 0);
           Overflow : out STD_LOGIC;
          --InsNum : out std_logic_vector(3 downto 0);
           Sign : out std_logic;
           Zero : out STD_LOGIC;
           AGreater : out STD_LOGIC;
           BGreater : out STD_LOGIC;
           an : out std_logic_vector(3 downto 0);
           Equal : out STD_LOGIC);

end Nanoprocessor2;

architecture Behavioral of Nanoprocessor2 is
component CLOCK
port(Clk_in: in std_logic;
    Clk_out : out std_logic);
end component;

component MUX_2_way_4_bit
port(I0 : in STD_LOGIC_VECTOR (3 downto 0);
           I1 : in STD_LOGIC_VECTOR (3 downto 0);
           S : in STD_LOGIC;
           Y : out STD_LOGIC_VECTOR (3 downto 0));
end component;

component PROM
port( I : in STD_LOGIC_VECTOR (3 downto 0);
          O : out STD_LOGIC_VECTOR (13 downto 0));
end component;

component RegisterBank
Port ( S : in STD_LOGIC_VECTOR (2 downto 0);
           RB_in : in STD_LOGIC_VECTOR (3 downto 0);
           CLK_in : in STD_LOGIC;
           R0_out : out STD_LOGIC_VECTOR (3 downto 0);
           R1_out : out STD_LOGIC_VECTOR (3 downto 0);
           R2_out : out STD_LOGIC_VECTOR (3 downto 0);
           R3_out : out STD_LOGIC_VECTOR (3 downto 0);
           R4_out : out STD_LOGIC_VECTOR (3 downto 0);
```

```
           R5_out : out STD_LOGIC_VECTOR (3 downto 0);
           R6_out : out STD_LOGIC_VECTOR (3 downto 0);
           R7_out : out STD_LOGIC_VECTOR (3 downto 0));
end component;

component MUX_8_way_4_bit
port( I0 : in STD_LOGIC_VECTOR (3 downto 0);
          I1 : in STD_LOGIC_VECTOR (3 downto 0);
          I2 : in STD_LOGIC_VECTOR (3 downto 0);
          I3 : in STD_LOGIC_VECTOR (3 downto 0);
          I4 : in STD_LOGIC_VECTOR (3 downto 0);
          I5 : in STD_LOGIC_VECTOR (3 downto 0);
          I6 : in STD_LOGIC_VECTOR (3 downto 0);
          I7 : in STD_LOGIC_VECTOR (3 downto 0);
          S_in : in STD_LOGIC_VECTOR (2 downto 0);
          Y_out : out STD_LOGIC_VECTOR (3 downto 0));
end component;

component ProgramCounter
Port ( Clk : in STD_LOGIC;
    Res : in std_logic;
           I : in STD_LOGIC_VECTOR (3 downto 0);
           Y : out STD_LOGIC_VECTOR (3 downto 0));
end component;

component RCA_4
Port ( I : in STD_LOGIC_VECTOR (3 downto 0);
           S_out : out STD_LOGIC_VECTOR (3 downto 0));
end component;

component MUX_2_way_3_bit
Port ( I0 : in STD_LOGIC_VECTOR (2 downto 0);
           I1 : in STD_LOGIC_VECTOR (2 downto 0);
           S : in STD_LOGIC;
           Y : out STD_LOGIC_VECTOR (2 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;

component Comparator
Port ( EN : in STD_LOGIC; --Enable input
           A : in STD_LOGIC_VECTOR (3 downto 0); --4 bit value: A
           B : in STD_LOGIC_VECTOR (3 downto 0); --4 bit value: B
           AGreat : out STD_LOGIC; --Is A the greater number?
           BGreat : out STD_LOGIC; --Is B the greater number?
           Equal : out STD_LOGIC); --Are the numbers equal?
end component;

component Bit14_Instruction_Decoder
    Port ( Ins : in STD_LOGIC_VECTOR (13 downto 0);
           JMP_CHK : in STD_LOGIC_VECTOR (3 downto 0);
           AGreat_CHK : in std_logic;
           Equal_CHK : in std_logic;
           Reg_En : out STD_LOGIC_VECTOR (2 downto 0);
```

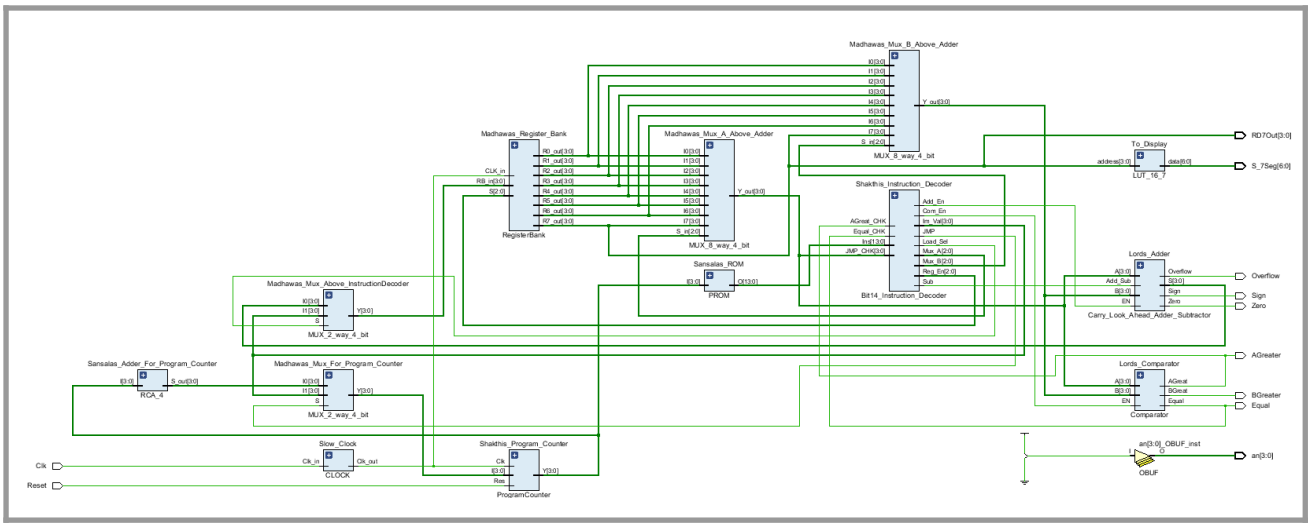
```
            Im_Val : out STD_LOGIC_VECTOR (3 downto 0);
            Load_Sel : out STD_LOGIC;
            Mux_A : out STD_LOGIC_VECTOR (2 downto 0);
            Mux_B : out STD_LOGIC_VECTOR (2 downto 0);
            Sub : out STD_LOGIC;
            JMP : out STD_LOGIC;
            Com_En : out STD_LOGIC;
            Add_En : out STD_LOGIC);
end component;

component Carry_Look_Ahead_Adder_Subtractor is
    Port ( A : in STD_LOGIC_VECTOR (3 downto 0); --Bus for the first
binary number
           B : in STD_LOGIC_VECTOR (3 downto 0); --Bus for the second
binary number
           Add_Sub : in STD_LOGIC; --Should we add B to A or should we
subtract B from A? (1 for subtraction, 0 for addition)
           EN : in STD_LOGIC; --Enable input
           S : inout STD_LOGIC_VECTOR (3 downto 0); --Bus for the binary
sum output
           Sign : inout STD_LOGIC; --Sign Bit: 1 for negative, 0 for
positive
           Overflow : inout STD_LOGIC; --Overflow bit: Is there an
overflow?
           --When the number is negative, is the number in range
[-0,-8]: if yes, there is no overflow => Overflow =0; if no, there is an
overflow => Overflow = 1
           --When the number is positive, is the number in range [+0,
7]: if yes, there is no overflow => Overflow =0; if no, there is an
overflow => Overflow = 1
           Zero : inout STD_LOGIC);
end component;

--signals from INSTRUCTION DECODER
signal Reg_En, Mux_A, Mux_B : std_logic_vector(2 downto 0);
signal Load_Sel, Sub, Jump, Add_En, Com_En : std_logic;
signal Im_Val:std_logic_vector(3 downto 0);

--signals from COMPARATOR
signal To_Equal_CHK, To_AGreat_CHK : std_logic;

--signals from MUX ABOVE DECODER
signal To_Register_Bank : std_logic_vector(3 downto 0);

--signals from REGISTER BANK
signal RD0, RD1, RD2, RD3, RD4, RD5, RD6, RD7: std_logic_vector(3 downto
0);

--signals from MUX_A and MUX_B
signal From_Mux_A, From_Mux_B: std_logic_vector(3 downto 0);

--signals from ADDER/SUBTRACTOR
signal From_Adder_Subtractor: std_logic_vector(3 downto 0);
signal Overflow_F, Zero_F, Carry_F, Sign_F:std_logic;

--signals from 3 BIT ADDER
signal From_4_Bit_Adder: std_logic_vector(3 downto 0);
```

```
--signals from MUX TO PROGRAM COUNTER
signal To_Program_Counter: std_logic_vector(3 downto 0);

--signals from PROGRAM COUNTER
signal From_Program_Counter: std_logic_vector(3 downto 0);

--signals from ROM
signal Instruction: std_logic_vector(13 downto 0);

--signals from CLOCK
signal Clk_out : std_logic;

begin

Slow_Clock : CLOCK
Port map(Clk_in => Clk,
        Clk_out => Clk_out);

Shakthis_Instruction_Decoder : Bit14_Instruction_Decoder
    Port map( Ins => Instruction,
           JMP_CHK => From_Mux_A,
           AGreat_CHK => To_AGreat_CHK,
           Equal_CHK => To_Equal_CHK,
           Reg_En => Reg_En,
           Im_Val => Im_Val,
           Load_Sel => Load_Sel,
           Mux_A => Mux_A,
           Mux_B => Mux_B,
           Sub => Sub,
           JMP => Jump,
           Com_En => Com_En,
           Add_En => Add_En);
Lords_Adder : Carry_Look_Ahead_Adder_Subtractor
Port map( A => From_Mux_A, --Bus for the first binary number
      B => From_Mux_B, --Bus for the second binary number
      Add_Sub => Sub, --Should we add B to A or should we subtract B
from A? (1 for subtraction, 0 for addition)
      EN =>Add_En, --Enable input
      S => From_Adder_Subtractor,--Bus for the binary sum output
      Sign => Sign_F, --Sign Bit: 1 for negative, 0 for positive
      Overflow => Overflow_F, --Overflow bit: Is there an overflow?
      --When the number is negative, is the number in range [-0,-8]: if
yes, there is no overflow => Overflow =0; if no, there is an overflow =>
Overflow = 1
      --When the number is positive, is the number in range [+0, 7]: if
yes, there is no overflow => Overflow =0; if no, there is an overflow =>
Overflow = 1
      Zero => Zero_F --is the number 0?
      );

Madhawas_Mux_Above_InstructionDecoder: MUX_2_way_4_bit
port map(I0 => From_Adder_Subtractor,
           I1 => Im_Val,
           S => Load_Sel,
           Y => To_Register_Bank);
```

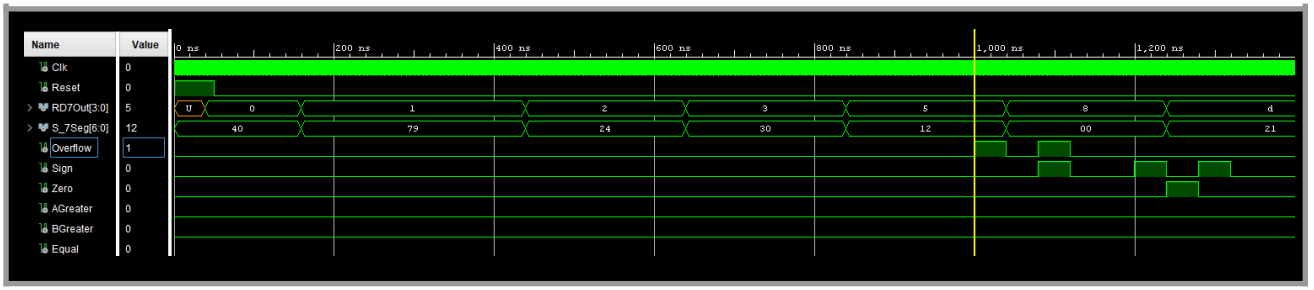
```
Madhawas_Register_Bank: RegisterBank
Port map( S => Reg_En,
          RB_in => To_Register_Bank,
          CLK_in => Clk_out,
          R0_out =>RD0,
          R1_out =>RD1,
          R2_out =>RD2,
          R3_out =>RD3,
          R4_out =>RD4,
          R5_out =>RD5,
          R6_out =>RD6,
          R7_out =>RD7);

Madhawas_Mux_A_Above_Adder: MUX_8_way_4_bit
port map( I0 => RD0,
      I1 => RD1,
      I2 => RD2,
      I3 => RD3,
      I4 => RD4,
      I5 => RD5,
      I6 => RD6,
      I7 => RD7,
      S_in => Mux_A,
      Y_out => From_Mux_A);

Madhawas_Mux_B_Above_Adder: MUX_8_way_4_bit
  port map( I0 => RD0,
        I1 => RD1,
        I2 => RD2,
        I3 => RD3,
        I4 => RD4,
        I5 => RD5,
        I6 => RD6,
        I7 => RD7,
        S_in => Mux_B,
        Y_out => From_Mux_B);

Shakthis_Program_Counter: ProgramCounter
        Port map(Clk => Clk_out,
            Res => Reset,
            I => To_Program_Counter,
            Y => From_Program_Counter);

Sansalas_Adder_For_Program_Counter: RCA_4
Port map( I =>From_Program_Counter,
           S_out =>From_4_Bit_Adder);

Madhawas_Mux_For_Program_Counter: MUX_2_way_4_bit
Port map( I0 => From_4_Bit_Adder,
      I1 => Im_Val(3 downto 0),
      S => Jump,
      Y => To_Program_Counter);

Sansalas_ROM : PROM
port map( I => From_Program_Counter,
          O => Instruction);
```

```
To_Display : LUT_16_7
Port map( address => RD7,
     data => S_7Seg);


Lords_Comparator : Comparator
Port map( EN => Com_En, --Enable input
          A => From_Mux_A, --4 bit value: A
          B => From_Mux_B, --4 bit value: B
          AGreat => To_AGreat_CHK, --Is A the greater number?
          BGreat => BGreater, --Is B the greater number?
          Equal => To_Equal_CHK); --Are the numbers equal?

RD7Out <= RD7;
Zero <= Zero_F;
Overflow <= Overflow_F;
Sign <= Sign_F;
an <= "1110";
--InsNum <= From_Program_Counter;
AGreater <= To_AGreat_CHK;
Equal <= To_Equal_CHK;

end Behavioral;
```

2.13.3. Elaborated Design Schematic



2.13.4. Timing Diagram



3. Sub-Components

3.1. Full Adder for 4-bit Carry Look Ahead Adder Subtractor

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

entity Full_Adder is
    Port ( A : in STD_LOGIC; --First input bit
           B : in STD_LOGIC; --Second input bit
           Carry_in : in STD_LOGIC; --Carry input bit
           Sum : out STD_LOGIC; --Sum output bit
           Carry_out : out STD_LOGIC; --Carry output bit
           P : out STD_LOGIC; --Propergate output bit
           G : out STD_LOGIC); --Generate output bit
end Full_Adder;

architecture Behavioral of Full_Adder is
component Half_Adder
    Port ( A : in STD_LOGIC; --First input bit
       B : in STD_LOGIC; --Second input bit
       Sum : out STD_LOGIC; --Sum output bit
       Carry : out STD_LOGIC); --Carry output bit
end component;

SIGNAL A1, A2, B1, B2: std_logic; --signal inputs of the two half adders
SIGNAL Sum1, Sum2, Carry1, Carry2: std_logic; --signal outputs of the
two half adders
begin
    Half_Adder_2 : Half_Adder --mapping the bottom half adder
    port map(
        A => A2,
        B => B2,
        Sum => Sum2,
        Carry => Carry2
    );
    Half_Adder_1 : Half_Adder --mapping the top half adder
    port map(
        A => A1,
        B => B1,
        Sum => Sum1,
        Carry => Carry1
    );
--Assigning the inputs to the bottom half adder
A2 <= A;
B2 <= B;

--Defining generation and propergation bits
P <= Sum2;
G <= Carry2;

--Assigning the inputs to the top half adder
B1 <= Carry_in;
A1 <= Sum2;
```

```
--Defining Carry_out and sum
Carry_out <= Carry2 OR Carry1;
Sum <= Sum1;
end Behavioral;
```

3.2. Carry Look Ahead Logic Unit for 4-bit Carry Look Ahead Adder Subtractor

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

entity Carry_Look_Ahead is
    Port ( P : in STD_LOGIC_VECTOR (2 downto 0); --Propergate input bus
           G : in STD_LOGIC_VECTOR (2 downto 0); --Propergate output bus
           Carry_in : in STD_LOGIC; --Carry input bit
           Carry_out : out STD_LOGIC_VECTOR (3 downto 2)); --Carry
output bus
end Carry_Look_Ahead;

architecture Behavioral of Carry_Look_Ahead is

begin
--Defining Carry_out(2)
Carry_out(2) <= (Carry_in AND P(0) AND P(1)) OR (G(0) AND P(1)) OR G(1);
--Defining Carry_out(1)
Carry_out(3) <= (Carry_in AND P(0) AND P(1) AND P(2)) OR (G(0) AND P(1)
AND P(2)) OR (G(1) AND P(2)) OR G(2);
end Behavioral;
```

3.3. D Flip-flop

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

entity D_FF is
    Port ( D : in STD_LOGIC;
           Res : in STD_LOGIC;
           Clk : in STD_LOGIC;
           Q : out STD_LOGIC;
           Qbar : out STD_LOGIC);
end D_FF;

architecture Behavioral of D_FF is

begin

process (Clk) begin
    if(rising_edge(Clk)) then
        if Res = '1' then
            Q <= '0';
```

```
            Qbar <= '1';
        else
            Q <= D;
            Qbar <= not D;
        end if;
    end if;
end process;


end Behavioral;
```

3.4. Half Adder

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

entity Half_Adder is
    Port ( A : in STD_LOGIC; --First input bit
           B : in STD_LOGIC; --Second input bit
           Sum : out STD_LOGIC; --Sum output bit
           Carry : out STD_LOGIC); --Carry output bit
end Half_Adder;

architecture Behavioral of Half_Adder is

begin
    --Defining sum and carry of the half adder
    Sum <= A XOR B;
    Carry <= A AND B;
end Behavioral;
```

3.5. Decoder_2_to_4

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

entity 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 Decoder_2_to_4;

architecture Behavioral of Decoder_2_to_4 is

begin
    Y(0) <= EN AND (NOT I(0)) AND (NOT I(1));
    Y(1) <= EN AND (I(0)) AND (NOT I(1));
    Y(2) <= EN AND (NOT I(0)) AND I(1);
    Y(3) <= EN AND I(0) AND I(1);

end Behavioral;
```

3.6. Decoder_3_to_8

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

entity Decoder_3_to_8 is
    Port ( I : in STD_LOGIC_VECTOR (2 downto 0);
           EN : in STD_LOGIC;
           Y : out STD_LOGIC_VECTOR (7 downto 0));
end Decoder_3_to_8;

architecture Behavioral of Decoder_3_to_8 is

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 I0, I1 : STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL Y0, Y1 : STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL I2, EN0, EN1 : STD_LOGIC;


begin

Decoder_2_to_4_0 : Decoder_2_to_4
    PORT MAP(
        I => I0,
        Y => Y0,
        EN => EN0
        );

Decoder_2_to_4_1 : Decoder_2_to_4
    PORT MAP(
        I => I1,
        Y => Y1,
        EN => EN1
        );

I2 <= I(2);
EN0 <= NOT(I(2)) AND EN;
EN1 <= I(2) AND EN;
I0 <= I(1 DOWNTO 0);
I1 <= I(1 DOWNTO 0);
Y(3 DOWNTO 0) <= Y0;
Y(7 DOWNTO 4) <= Y1;

end Behavioral;
```

3.7. Comparator Latch for the 4-bit Comparator

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

entity Comparator_Latch is
    Port ( An : in STD_LOGIC; --n th bit of number A
           Bn : in STD_LOGIC; --n th bit of number B
           IA : in STD_LOGIC; --input status: A
           IB : in STD_LOGIC; --input status: B
           OA : out STD_LOGIC; --output status: A
           OA1 : out STD_LOGIC; --AGreat Special Case
           OB : out STD_LOGIC; --output status: B
           OB1 : out STD_LOGIC); --BGreat Special Case
end Comparator_Latch;

architecture Behavioral of Comparator_Latch is

begin
    OA <= IA OR NOT(NOT(An) OR Bn OR IB); -- output status: A
    OB <= IB OR NOT(NOT(Bn) OR An OR IA); -- output status: B

    OA1 <= (IA AND An) AND (IB AND NOT(Bn)); --AGreat Special Case
    OB1 <= (IA AND NOT(An)) AND (IB AND Bn); --BGreat Special Case
end Behavioral;
```

3.8. Full Adder

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

entity FA is
    Port ( A : in STD_LOGIC;
           B : in STD_LOGIC;
           C_in : in STD_LOGIC;
           S : out STD_LOGIC;
           C_out : out STD_LOGIC);
end FA;

architecture Behavioral of FA is

component HA
    port (
    A: in std_logic;
    B: in std_logic;
    S: out std_logic;
    C: out std_logic);
end component;

SIGNAL HA0_S, HA0_C, HA1_S, HA1_C: std_logic;

begin
```

```
HA_0 : HA
    port map (
    A => A,
    B => B,
    S => HA0_S,
    C => HA0_C);

HA_1 : HA
    port map (
    A => HA0_S,
    B => C_in,
    S => HA1_S,
    C => HA1_C);

C_out <= HA0_C OR HA1_C;
S <= HA1_S;

end Behavioral;
```

3.9. MUX_8_to_1

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

entity Decoder_3_to_8 is
    Port ( I : in STD_LOGIC_VECTOR (2 downto 0);
           EN : in STD_LOGIC;
           Y : out STD_LOGIC_VECTOR (7 downto 0));
end Decoder_3_to_8;

architecture Behavioral of Decoder_3_to_8 is

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 I0, I1 : STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL Y0, Y1 : STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL I2, EN0, EN1 : STD_LOGIC;

begin

Decoder_2_to_4_0 : Decoder_2_to_4
    PORT MAP(
        I => I0,
        Y => Y0,
        EN => EN0
        );

Decoder_2_to_4_1 : Decoder_2_to_4
    PORT MAP(
        I => I1,
```

```
        Y => Y1,
        EN => EN1
        );

I2 <= I(2);
EN0 <= NOT(I(2)) AND EN;
EN1 <= I(2) AND EN;
I0 <= I(1 DOWNTO 0);
I1 <= I(1 DOWNTO 0);
Y(3 DOWNTO 0) <= Y0;
Y(7 DOWNTO 4) <= Y1;

end Behavioral;
```

3.10. 4-Bit Register

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

entity Register_4_bit is
    Port ( R_in : in STD_LOGIC_VECTOR (3 downto 0);
           En : in STD_LOGIC;
           Clk : in STD_LOGIC;
           R_out : out STD_LOGIC_VECTOR (3 downto 0));
end Register_4_bit;

architecture Behavioral of Register_4_bit is

begin

    process (clk) begin
        if (rising_edge(clk)) then
            if EN = '1' then
                R_out <= R_in;
            end if;
        end if;
    end process;

end Behavioral;
```

3.11. 2-way 3-bit MUX

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

entity MUX_2_way_3_bit is
    Port ( I0 : in STD_LOGIC_VECTOR (2 downto 0);
           I1 : in STD_LOGIC_VECTOR (2 downto 0);
           S : in STD_LOGIC;
           Y : out STD_LOGIC_VECTOR (2 downto 0));
end MUX_2_way_3_bit;

architecture Behavioral of MUX_2_way_3_bit is

begin

Y(0) <= ( I0(0) AND NOT S ) OR ( I1(0) AND S );
Y(1) <= ( I0(1) AND NOT S ) OR ( I1(1) AND S );
Y(2) <= ( I0(2) AND NOT S ) OR ( I1(2) AND S );

end Behavioral;
```

4. Legend for the Number System

7-segment Output	Decimal Representation	7-segment Output	Decimal Representation
0	0	8	-8
1	+1	9	-7
2	+2	a	-6
3	+3	b	-5
4	+4	c	-4
5	+5	d	-3
6	+6	e	-2
7	+7	f	-1

5. Compiler for Improved Nanoprocessor

5.1. Details

The Compiler is a Python program that converts Assembly code into machine language that can be directly pasted in the VHDL Source Code of the Program ROM. This was created for ease of program writing. It does this by reading from a text file named “code.txt” that contains the assembly code to be compiled, turns it into machine code and rewrites the same file. Note that this compiler works only for the Improved Nanoprocessor.

5.2. Source Code

```
import re

machineCode = []


def to_4_bit_signed_binary(number):
    if not (-8 <= number <= 7):
        raise ValueError("Number must be between -8 and 7 (inclusive).")

    if number >= 0:
        binary_string = format(number, '04b')
    else:
        two_complement = (number + 16) % 16
        binary_string = format(two_complement, '04b')

    return binary_string


def AddFunc(rest, no):
    if re.fullmatch(r'[r][0-7],[r][0-7]', rest):
        splitS = rest.split(",")
        firstR = int(splitS[0][-1])
        if firstR == 0 or firstR == 1 or firstR == 2:
            print(f"line {no}: WARNING: R{firstR} is a Read-Only Register.")
        secondR = int(splitS[1][-1])
        bs1 = bin(firstR)[2:].zfill(3)
        bs2 = bin(secondR)[2:].zfill(3)
        return "0000" + bs1 + bs2 + "0000"
    else:
        print(f"line {no}: ERROR")



def SubFunc(rest, no):
    if re.fullmatch(r'[r][0-7],[r][0-7]', rest):
        splitS = rest.split(",")
        firstR = int(splitS[0][-1])
        if firstR == 0 or firstR == 1 or firstR == 2:
            print(f"line {no}: WARNING: R{firstR} is a Read-Only Register.")
        secondR = int(splitS[1][-1])
        bs1 = bin(firstR)[2:].zfill(3)
        bs2 = bin(secondR)[2:].zfill(3)
        return "0001" + bs1 + bs2 + "0000"
    else:
        print(f"line {no}: ERROR")
```

```
def MoviFunc(rest, no):
    if re.fullmatch(r'r[0-7],-?[0-7]|r[0-7],-8', rest):
        splitS = rest.split(",")
        firstR = int(splitS[0][-1])
        if firstR == 0 or firstR == 1 or firstR == 2:
            print(f"line {no}: WARNING: R{firstR} is a Read-Only Register.")
        bs1 = bin(firstR)[2:].zfill(3)
        second = int(splitS[1])
        data = to_4_bit_signed_binary(second)
        return "0010" + bs1 + "000" + data+""
    else:
        print(f"line {no}: ERROR")


def JZRFunc(rest, no):
    if re.fullmatch(r'[r][0-7],[1-9]|[r][0-7],1[0-6]', rest):
        splitS = rest.split(",")
        firstR = int(splitS[0][-1])
        secondR = int(splitS[1]) - 1
        bs1 = bin(firstR)[2:].zfill(3)
        bs2 = bin(secondR)[2:].zfill(4)
        return "0011" + bs1 + "000" + bs2+""
    else:
        print(f"line {no}: ERROR")


def INC(rest, no):
    rest = rest.strip()
    if re.fullmatch(r'[r][0-7]', rest):
        firstR = int(rest[-1])
        if firstR == 0 or firstR == 1 or firstR == 2:
            print(f"line {no}: WARNING: R{firstR} is a Read-Only Register.")
        bs1 = bin(firstR)[2:].zfill(3)
        return "0100" + bs1 + "0010000"
    else:
        print(f"line {no}: ERROR")


def DEC(rest, no):
    rest = rest.strip()
    if re.fullmatch(r'[r][0-7]', rest):
        firstR = int(rest[-1])
        if firstR == 0 or firstR == 1 or firstR == 2:
            print(f"line {no}: WARNING: R{firstR} is a Read-Only Register.")
        bs1 = bin(firstR)[2:].zfill(3)
        return "0101" + bs1 + "0100000"
    else:
        print(f"line {no}: ERROR")


def NEG(rest, no):
    rest = rest.strip()
    if re.fullmatch(r'[r][0-7]', rest):
        firstR = int(rest[-1])
        if firstR == 0 or firstR == 1 or firstR == 2:
            print(f"line {no}: WARNING: R{firstR} is a Read-Only Register.")
        bs1 = bin(firstR)[2:].zfill(3)
        return "0110" + bs1 + "0000000"
    else:
        print(f"line {no}: ERROR")
```

```
def RES(rest, no):
    rest = rest.strip()
    if re.fullmatch(r'[r][0-7]', rest):
        firstR = int(rest[-1])
        if firstR == 0 or firstR == 1 or firstR == 2:
            print(f"line {no}: WARNING: R{firstR} is a Read-Only Register.")
        bs1 = bin(firstR)[2:].zfill(3)
        return "0111" + bs1 + "0000000"
    else:
        print(f"line {no}: ERROR")


def COM(rest, no):
    if re.fullmatch(r'[r][0-7],[r][0-7]', rest):
        splitS = rest.split(",")
        firstR = int(splitS[0][-1])
        secondR = int(splitS[1][-1])
        bs1 = bin(firstR)[2:].zfill(3)
        bs2 = bin(secondR)[2:].zfill(3)
        return "1000" + bs1 + bs2 + "0000"
    else:
        print(f"line {no}: ERROR")

def IFAG(rest, no):
    if re.fullmatch(r'[r][0-7],[r][0-7],[0-9]|[r][0-7],[r][0-7],1[0-6]', rest):
        splitS = rest.split(",")
        firstR = int(splitS[0][-1])
        secondR = int(splitS[1][-1])
        address = int(splitS[2]) - 1
        bs1 = bin(firstR)[2:].zfill(3)
        bs2 = bin(secondR)[2:].zfill(3)
        addressBin = bin(address)[2:].zfill(4)
        return "1010"+bs1+bs2+addressBin
    else:
        print(f"line {no}: ERROR")

def IFE(rest, no):
    if re.fullmatch(r'[r][0-7],[r][0-7],[0-9]|[r][0-7],[r][0-7],1[0-6]', rest):
        splitS = rest.split(",")
        firstR = int(splitS[0][-1])
        secondR = int(splitS[1][-1])
        address = int(splitS[2]) - 1
        bs1 = bin(firstR)[2:].zfill(3)
        bs2 = bin(secondR)[2:].zfill(3)
        addressBin = bin(address)[2:].zfill(4)
        return "1011"+bs1+bs2+addressBin
    else:
        print(f"line {no}: ERROR")

def IFNE(rest, no):
    if re.fullmatch(r'[r][0-7],[r][0-7],[0-9]|[r][0-7],[r][0-7],1[0-6]', rest):
        splitS = rest.split(",")
        firstR = int(splitS[0][-1])
        secondR = int(splitS[1][-1])
        address = int(splitS[2]) - 1
        bs1 = bin(firstR)[2:].zfill(3)
        bs2 = bin(secondR)[2:].zfill(3)
        addressBin = bin(address)[2:].zfill(4)
        return "1100"+bs1+bs2+addressBin
    else:
        print(f"line {no}: ERROR")
def is_not_single_word(string):
```

```
    # Define a regex pattern that matches any string containing whitespace
    # or punctuation characters other than a single word
    pattern = r'\s'

    # Use re.search to search for the pattern in the string
    if re.search(pattern, string):
        return True  # The string is not a single word (contains whitespace)
    else:
        return False  # The string is a single word (does not contain
whitespace)

def write_list_to_file(filename, data_list, lines):
    # Open the file in write mode
    with open(filename, 'w') as file:
        n = len(data_list)

        # Iterate through the list and write each element to the file
        for x in range(n-1):
            # Convert the item to a string and write it to the file with a
newline
            file.write(f"\"{data_list[x]}\", --{lines[x].upper().strip()}\n")
        file.write(f"\"{data_list[n-1]}\" --{lines[n-1].upper().strip()}")
        file.close()
def compileLine(line, no):
    rest = ""
    if is_not_single_word(line):
        splitS = line.split(" ", 1)
        rest = splitS[1].replace(" ", "")
    else:
        splitS = [line]
    function = splitS[0]
    match function:
        case "add":
            return AddFunc(rest, no)
        case "sub":
            return SubFunc(rest, no)
        case "movi":
            return MoviFunc(rest, no)
        case "jzr":
            return JZRFunc(rest, no)
        case "inc":
            return INC(rest, no)
        case "dec":
            return DEC(rest, no)
        case "neg":
            return NEG(rest, no)
        case "res":
            return RES(rest, no)
        case "com":
            return COM(rest, no)
        case "nop":
            return "10010000000000"
        case "ifag":
            return IFAG(rest, no)
        case "ife":
            return IFE(rest, no)
        case "ifne":
            return IFNE(rest, no)
        case _:
            print(f"line {no}: ERROR: Keyword {function.upper()} isn't
recognized.")
```

```
fileToCompile = open('code.txt', 'r')
lines = fileToCompile.readlines()
if len(lines) > 16:
   print("ERROR: Too many lines of code")
else:
   x = 0
   for line in lines:
       x += 1
       line = line.strip()
       line = line.lower()
       machineCode.append(compileLine(line, x))
   print(x)

   if None in machineCode:
       print("Compilation Failed")
   else:
       for n in range(16-x):
           machineCode.append("10010000000000")
           lines.append("Filled in with NOP")
       print(lines)
       write_list_to_file("code.txt", machineCode, lines)
       print("Compilation Successful")
```

6. Code Examples for the Improved Nano-Processor

6.1. Fibonacci Sequence

6.1.1. Assembly Code

```
MOVI R7, 0
MOVI R4, 5
MOVI R7, 1
MOVI R6, 0
ADD R7, R6
DEC R4
SUB R6, R7
JZR R4, 10
JZR R0, 5
JZR R0, 10
```

6.1.2. Machine Code (Compiler Output)

```
"00101110000000", --MOVI R7, 0
"00101000000101", --MOVI R4, 5
"00101110000001", --MOVI R7, 1
"00101100000000", --MOVI R6, 0
"00001111100000", --ADD R7, R6
"01011000100000", --DEC R4
"00011101110000", --SUB R6, R7
"00111000001001", --JZR R4, 10
"00110000000100", --JZR R0, 5
"00110000001001", --JZR R0, 10
"10010000000000", --FILLED IN WITH NOP
"10010000000000", --FILLED IN WITH NOP
"10010000000000", --FILLED IN WITH NOP
"10010000000000", --FILLED IN WITH NOP
"10010000000000", --FILLED IN WITH NOP
"10010000000000" --FILLED IN WITH NOP
```

6.2. Iteratively add 1+2+3

6.2.1. Assembly Code

```
MOVI R3,2
MOVI R7,3
MOVI R6, 1
NEG R6
ADD R7, R3
ADD R3, R6
JZR R3, 7
JZR R0, 5
```

6.2.2. Machine Code (Compiler Output)

```
"00100110000010", --MOVI R3,2
"00101110000011", --MOVI R7,3
"00101100000001", --MOVI R6, 1
"01101100000000", --NEG R6
"00001110110000", --ADD R7, R3
"00000111100000", --ADD R3, R6
"00110110000110", --JZR R3, 7
"00110000000100", --JZR R0, 5
"10010000000000", --FILLED IN WITH NOP
"10010000000000", --FILLED IN WITH NOP
"10010000000000", --FILLED IN WITH NOP
"10010000000000", --FILLED IN WITH NOP
"10010000000000", --FILLED IN WITH NOP
"10010000000000", --FILLED IN WITH NOP
"10010000000000" --FILLED IN WITH NOP
```

6.3. A Program to showcase If and Comparator related instructions

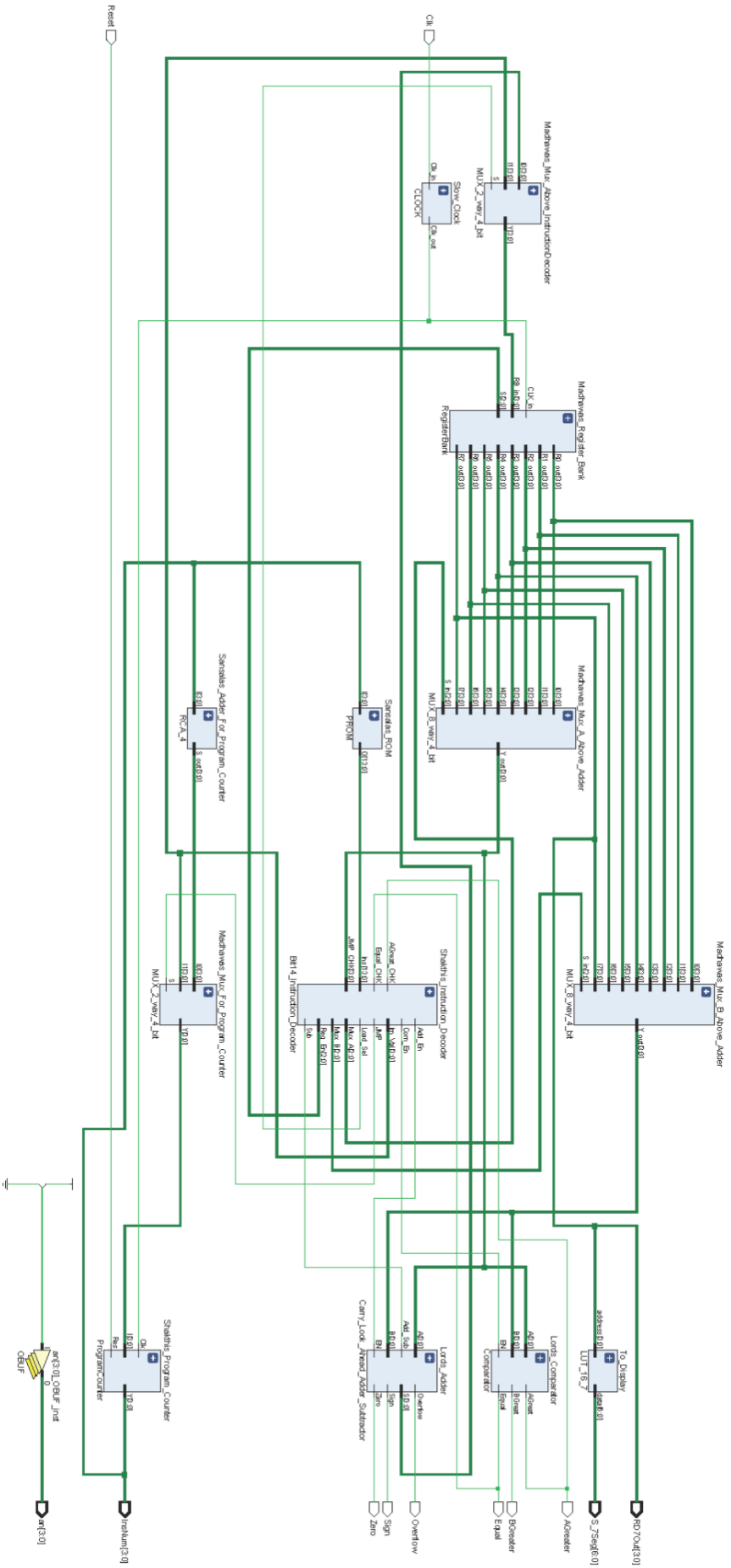
6.3.1. Assembly Code

```
MOVI R3, 3
MOVI R7, 5
MOVI R4, 5
IFAG R7, R3, 7
IFE R7, R3, 10
NOP
DEC R7
JZR R0, 4
NOP
INC R7
IFNE R7, R4, 10
IFE R7, R4, 14
NOP
INC R3
IFE R7, R3, 15
IFNE R7, R3, 4
```

6.3.2. Machine Code (Compiler Output)

```
"00100110000011", --MOVI R3, 3
"00101110000101", --MOVI R7, 5
"00101000000101", --MOVI R4, 5
"10101110110110", --IFAG R7, R3, 7
"10111110111001", --IFE R7, R3, 10
"10010000000000", --NOP
"01011110100000", --DEC R7
"00110000000011", --JZR R0, 4
"10010000000000", --NOP
"01001110010000", --INC R7
"11001111001001", --IFNE R7, R4, 10
"10111111001101", --IFE R7, R4, 14
"10010000000000", --NOP
"01000110010000", --INC R3
"10111110111110", --IFE R7, R3, 15
"11001110110011" --IFNE R7, R3, 4
```

Enlarged Elaborated Design Schematic



Contribution of Each Team Member

- **Abhayawickrama G.M.A.M. 220011G : 40 Hours**
  - 2-way 4-bit MUX
  - Register Bank
  - 8-way 4-bit MUX
- **Gangadari M.D.S. 220178X : 40 Hours**
  - Program ROM
  - 4-bit RCA for Program Counter
- **Senevirathne S.M.P.U. 220599M : 40 Hours**
  - 4-bit Carry Look Ahead Adder/Subtractor
  - 4-bit Comparator
- **Weerawansa M.S.I. 220690J : 40 Hours**
  - Instruction Decoder
  - Program Counter