Nano Processor

# Group name - **Kyojin**

## Group members

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

## Contribution

### Tharuka E A

Instruction Decoder

Program ROM

LUT (Look-up Table)

### Rubasinghe R K R U

Register Bank

Add/Subtract Unit

### **Perera P D N D**

Program Counter

3-bit register

3-bit adder

2 way 3-bit multiplexer

[Introduction 4](#_Toc70247338)

[Instruction Decoder 4](#_Toc70247339)

[VHDL code 4](#_Toc70247340)

[RTL Schematics 6](#_Toc70247341)

[Behavioral Simulation Code 6](#_Toc70247342)

[Timing Diagram 8](#_Toc70247343)

[PROGRAM ROM 8](#_Toc70247344)

[ASSEMBLY PROGRAM AND MACHINE CODE REPRESENTATION 8](#_Toc70247345)

[VHDL CODE 9](#_Toc70247346)

[RTL SCHEMATICS 10](#_Toc70247347)

[BEHAVIORAL SIMULATION CODE 10](#_Toc70247348)

[TIMING DIAGRAM 11](#_Toc70247349)

[LOOKUP TABLE 11](#_Toc70247350)

[VHDL CODE 12](#_Toc70247351)

[RTL SCHEMATICS 13](#_Toc70247352)

[REGISTER BANK 13](#_Toc70247353)

[VHDL CODE 13](#_Toc70247354)

[RTL SCHEMATICS 16](#_Toc70247355)

[BEHAVIORAL SIMULATION CODE 17](#_Toc70247356)

[TIMING DIAGRAM 20](#_Toc70247357)

[4–BIT ADD/SUBTRACT UNIT 20](#_Toc70247358)

[VHDL CODE 20](#_Toc70247359)

[RTL SCHEMATICS 23](#_Toc70247360)

[BEHAVIORAL SIMULATION CODE 23](#_Toc70247361)

[TIMING DIAGRAM 25](#_Toc70247362)

[4-BIT RIPPLE CARRY ADDER 25](#_Toc70247363)

[VHDL CODE 25](#_Toc70247364)

[RTL SCHEMATIC 27](#_Toc70247365)

[PROGRAM COUNTER 28](#_Toc70247366)

[VHDL CODE 28](#_Toc70247367)

[RTL SCHEMATICS 31](#_Toc70247368)

[BEHAVIORAL SIMULATION CODE 31](#_Toc70247369)

[TIMING DIAGRAM 33](#_Toc70247370)

[PC REGISTER 33](#_Toc70247371)

[VHDL CODE 33](#_Toc70247372)

[RTL SCHEMATICS 34](#_Toc70247373)

[BEHAVIORAL SIMULATION CODE 34](#_Toc70247374)

[TIMING DIAGRAM 36](#_Toc70247375)

[3–BIT ADDER 36](#_Toc70247376)

[VHDL CODE 36](#_Toc70247377)

[RTL SCHEMATICS 38](#_Toc70247378)

[BEHAVIORAL SIMULATION CODE 38](#_Toc70247379)

[TIMING DIAGRAM 40](#_Toc70247380)

[K–WAY B–BIT MUX 40](#_Toc70247381)

[8–WAY 4–BIT MUX 41](#_Toc70247382)

[VHDL CODE 41](#_Toc70247383)

[RTL SCHEMATICS 42](#_Toc70247384)

[BEHAVIORAL SIMULATION CODE 42](#_Toc70247385)

[TIMING DIAGRAM 44](#_Toc70247386)

[2–WAY 4–BIT MUX 44](#_Toc70247387)

[VHDL CODE 44](#_Toc70247388)

[RTL SCHEMATICS 45](#_Toc70247389)

[BEHAVIORAL SIMULATION CODE 45](#_Toc70247390)

[TIMING DIAGRAM 46](#_Toc70247391)

[2–WAY 3–BIT MUX 47](#_Toc70247392)

[VHDL CODE 47](#_Toc70247393)

[RTL SCHEMATICS 47](#_Toc70247394)

[BEHAVIORAL SIMULATION CODE 48](#_Toc70247395)

[TIMING DIAGRAM 49](#_Toc70247396)

[REGISTER 49](#_Toc70247397)

[VHDL CODE 49](#_Toc70247398)

[RTL SCHEMATICS 50](#_Toc70247399)

[BEHAVIORAL SIMULATION CODE 50](#_Toc70247400)

[TIMING DIAGRAM 52](#_Toc70247401)

[SLOW CLOCK 52](#_Toc70247402)

[VHDL CODE 52](#_Toc70247403)

[NANO PROCESSOR 53](#_Toc70247404)

[VHDL CODE 53](#_Toc70247405)

[RTL SCHEMATICS 59](#_Toc70247406)

[BEHAVIORAL SIMULATION CODE 59](#_Toc70247407)

[TIMING DIAGRAM 61](#_Toc70247408)

[XDC FILE 61](#_Toc70247409)

[CONCLUSION 63](#_Toc70247410)

# Introduction

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

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

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

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

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

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

Instruction Decoder

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

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

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

VHDL code

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Ins\_Decoder is

Port ( Instruction : in STD\_LOGIC\_VECTOR (11 downto 0);

JumpCheck : in STD\_LOGIC\_VECTOR (3 downto 0);

Reg\_En : out STD\_LOGIC\_VECTOR (2 downto 0);

Load\_Sel : out STD\_LOGIC;

Imm\_Val : out STD\_LOGIC\_VECTOR (3 downto 0);

Reg\_Sel\_A : out STD\_LOGIC\_VECTOR (2 downto 0);

Reg\_Sel\_B : out STD\_LOGIC\_VECTOR (2 downto 0);

AddSub\_Sel : out STD\_LOGIC;

Jump\_Flag : out STD\_LOGIC;

Jump\_Add : out STD\_LOGIC\_VECTOR (2 downto 0);

Neg\_Sel : out STD\_LOGIC);

end Ins\_Decoder;

architecture Behavioral of Ins\_Decoder is

component Decoder\_2\_to\_4 is

Port ( I : in STD\_LOGIC\_VECTOR (1 downto 0);

EN : in STD\_LOGIC;

Y : out STD\_LOGIC\_VECTOR (3 downto 0));

end component;

signal Add, Neg, Mov, Jmp, En\_Sel : std\_logic;

--En\_Sel is for enable registers only if the instruction is not Jmp

begin

Decoder\_2\_to\_4\_0: Decoder\_2\_to\_4

port map(

I(0) => Instruction(10),

I(1) => Instruction(11),

EN => '1',

Y(0) => Add,

Y(1) => Neg,

Y(2) => Mov,

Y(3) => Jmp);

Load\_Sel <= Mov;

AddSub\_Sel <= Add or Neg;

Neg\_Sel <= Neg;

En\_Sel <= Add or Mov or Neg;

Imm\_Val <= Instruction(3 downto 0);

Reg\_En(0) <= En\_Sel and Instruction(7);

Reg\_En(1) <= En\_Sel and Instruction(8);

Reg\_En(2) <= En\_Sel and Instruction(9);

Reg\_Sel\_A <= Instruction(9 downto 7);

Reg\_Sel\_B <= Instruction(6 downto 4);

--check the register R = 0 before jump

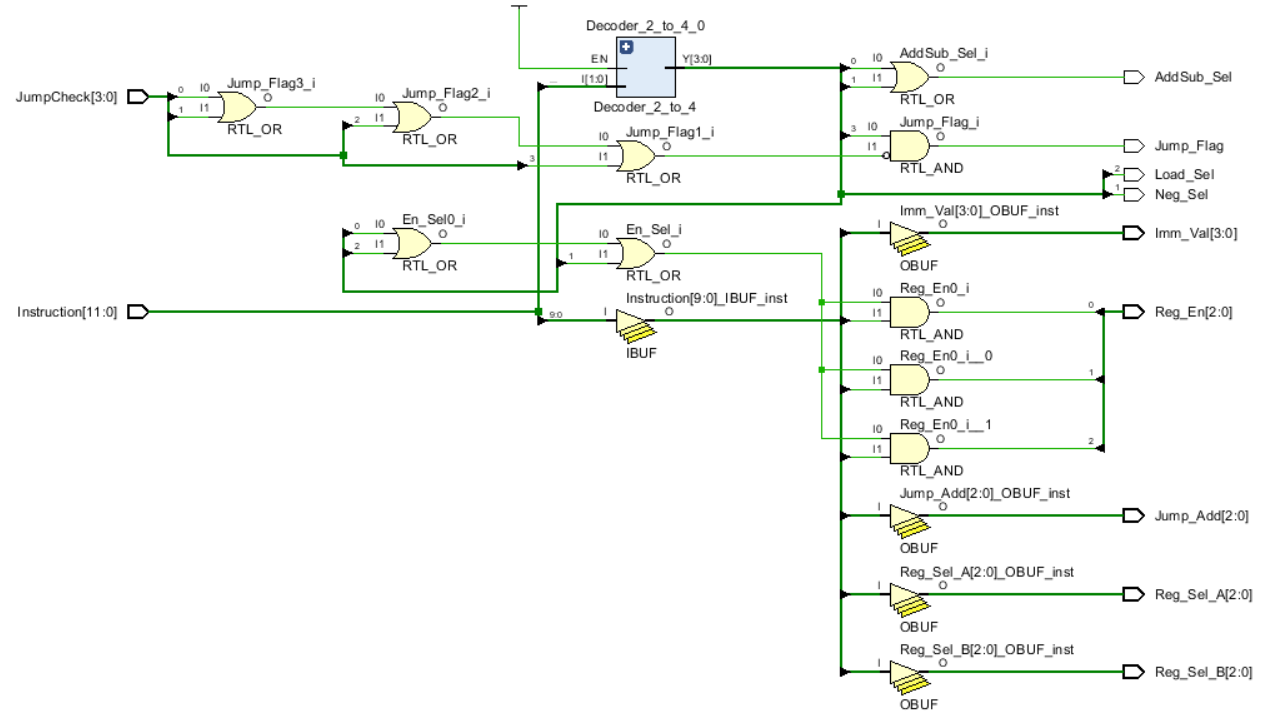
Jump\_Flag <= Jmp and (not (JumpCheck(0) or JumpCheck(1) or JumpCheck(2) or JumpCheck(3)));

--address to jump is in the first three digits of the instruction

Jump\_Add <= Instruction(2 downto 0);

end Behavioral;

RTL Schematics



Behavioral Simulation Code

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Sim\_InsDecoder is

-- Port ( );

end Sim\_InsDecoder;

architecture Behavioral of Sim\_InsDecoder is

component Ins\_Decoder is

Port ( Instruction : in STD\_LOGIC\_VECTOR (11 downto 0);

JumpCheck : in STD\_LOGIC\_VECTOR (3 downto 0);

Reg\_En : out STD\_LOGIC\_VECTOR (2 downto 0);

Load\_Sel : out STD\_LOGIC;

Imm\_Val : out STD\_LOGIC\_VECTOR (3 downto 0);

Reg\_Sel\_A : out STD\_LOGIC\_VECTOR (2 downto 0);

Reg\_Sel\_B : out STD\_LOGIC\_VECTOR (2 downto 0);

AddSub\_Sel : out STD\_LOGIC;

Jump\_Flag : out STD\_LOGIC;

Jump\_Add : out STD\_LOGIC\_VECTOR (2 downto 0);

Neg\_Sel : out STD\_LOGIC);

end component;

signal Instruction : std\_logic\_vector(11 downto 0);

signal JumpCheck, Imm\_Val : std\_logic\_vector(3 downto 0);

signal Reg\_En, Reg\_Sel\_A, Reg\_Sel\_B, Jump\_Add : std\_logic\_vector(2 downto 0);

signal Load\_Sel, AddSub\_Sel, Jump\_Flag, Neg\_Sel : std\_logic;

begin

UUT : Ins\_Decoder port map(

Instruction => Instruction,

JumpCheck => JumpCheck,

Reg\_En => Reg\_En,

Load\_Sel => Load\_Sel,

Imm\_Val => Imm\_Val,

Reg\_Sel\_A => Reg\_Sel\_A,

Reg\_Sel\_B => Reg\_Sel\_B,

AddSub\_Sel => AddSub\_Sel,

Jump\_Flag => Jump\_Flag,

Jump\_Add => Jump\_Add,

Neg\_Sel => Neg\_Sel);

process

begin

Instruction <= "101110000011";

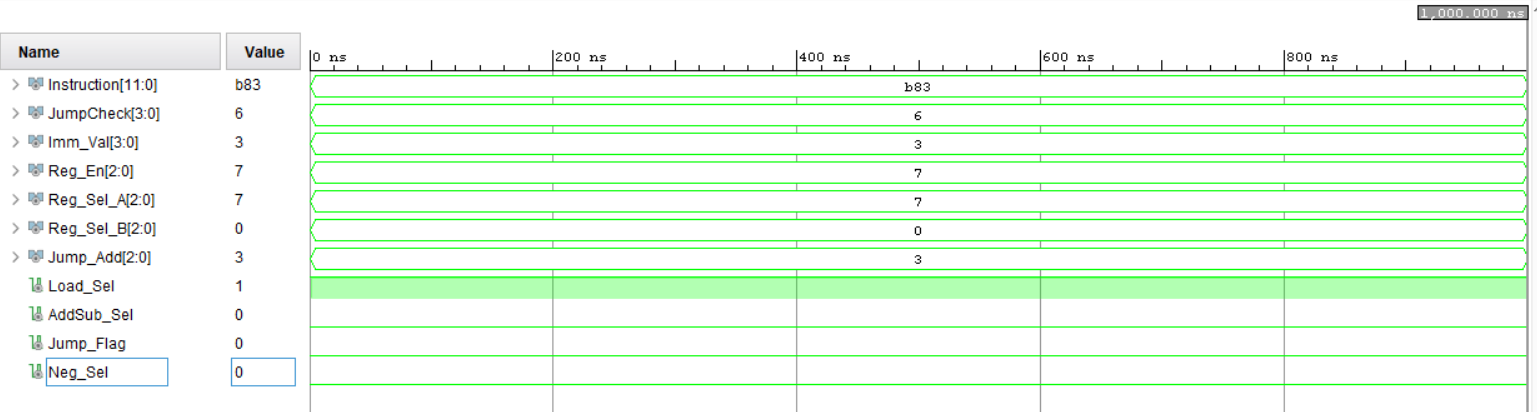
JumpCheck <= "0110";

wait;

end process;

end Behavioral;

Timing Diagram



# PROGRAM ROM

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

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

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

## ASSEMBLY PROGRAM AND MACHINE CODE REPRESENTATION

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

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

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

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

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

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

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

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

## VHDL CODE

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.NUMERIC\_STD.ALL;

entity Program\_ROM is

Port ( Mem\_Sel : in STD\_LOGIC\_VECTOR (2 downto 0);

Instruction : out STD\_LOGIC\_VECTOR (11 downto 0));

end Program\_ROM;

architecture Behavioral of Program\_ROM is

type rom\_type is array( 0 to 7 ) of std\_logic\_vector (11 downto 0);

signal instruction\_ROM : rom\_type := (

"101110000011", --10 111 000 0011 -> MOVI R7, 3 ; R7 <- 3 - 000 - 0

"100010000011", --10 001 000 0011 -> MOVI R1, 3 ; R1 <- 3 - 001 - 1

"100100000001", --10 010 000 0001 -> MOVI R2, 1 ; R2 <- 1 - 010 - 2

"010100000000", --01 010 000 0000 -> NEG R2 ; R2 <- -R2 - 011 - 3

"000010100000", --00 001 010 0000 -> ADD R1, R2 ; R1 <- R1 + R2 - 100 - 4

"001110010000", --00 111 001 0000 -> ADD R7, R1 ; R7 <- R7 + R1 - 101 - 5

"110010000110", --11 001 000 0110 -> JZR R1, 6 ; If R1 = 0 jump to line 6 - 110 - 6

"110000000100" --11 000 000 0100 -> JZR R0, 4 ; If R0 = 0 jump to line 4 - 111 - 7

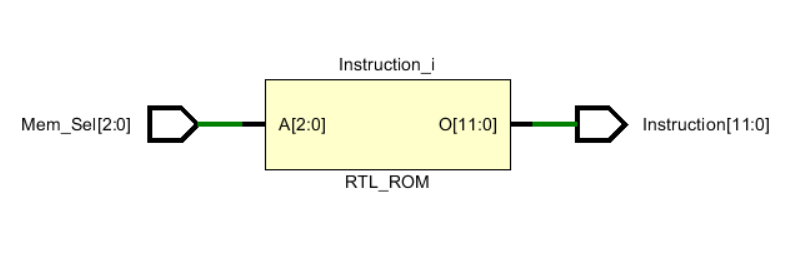
);

begin

Instruction <= instruction\_ROM(to\_integer(unsigned(Mem\_Sel)));

end Behavioral;

## RTL SCHEMATICS



## BEHAVIORAL SIMULATION CODE

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Sim\_ProgramROM is

-- Port ( );

end Sim\_ProgramROM;

architecture Behavioral of Sim\_ProgramROM is

component Program\_ROM is

Port ( Mem\_Sel : in STD\_LOGIC\_VECTOR (2 downto 0);

Instruction : out STD\_LOGIC\_VECTOR (11 downto 0));

end component;

signal Mem\_Sel : std\_logic\_vector(2 downto 0);

signal Instruction : std\_logic\_vector(11 downto 0);

begin

UUT : Program\_ROM port map(

Mem\_Sel => Mem\_Sel,

Instruction => Instruction);

process

begin

Mem\_Sel <= "001";

wait for 100ns;

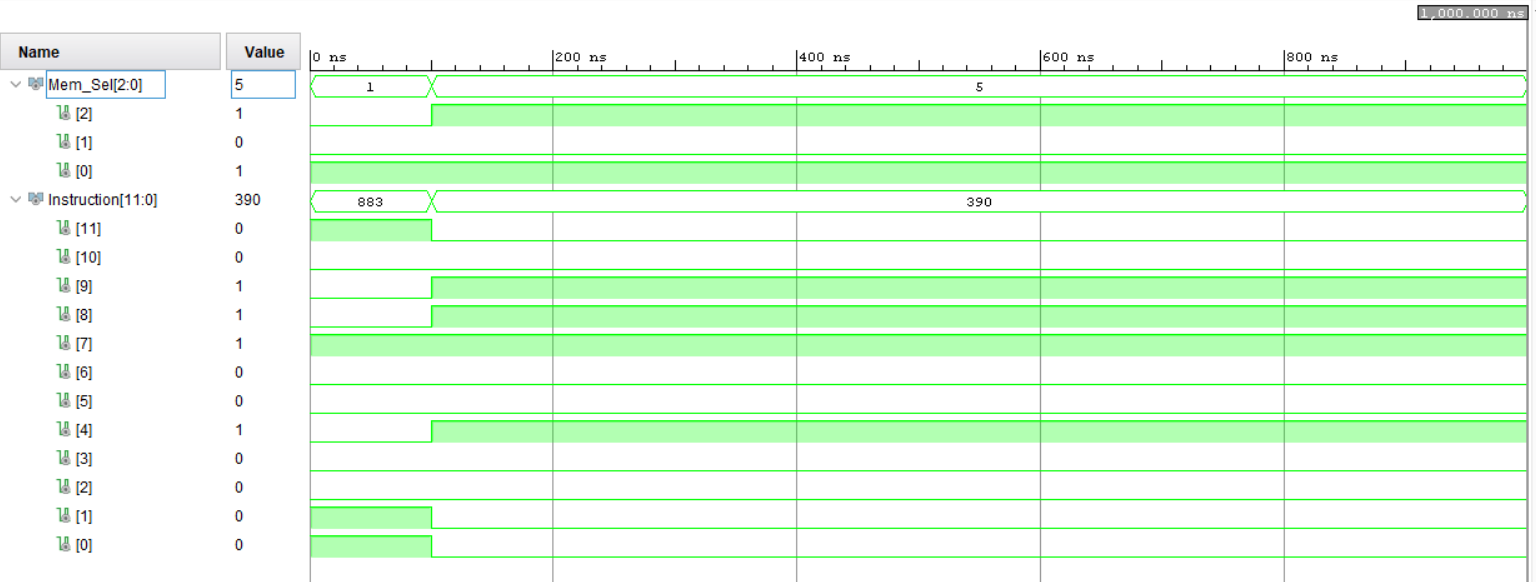
Mem\_Sel <= "101";

wait;

end process;

end Behavioral;

## TIMING DIAGRAM



# LOOKUP TABLE

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

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

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

## VHDL CODE

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use ieee.numeric\_std.all;

entity LUT\_7seg is

Port ( Address : in STD\_LOGIC\_VECTOR (3 downto 0);

Data : out STD\_LOGIC\_VECTOR (6 downto 0));

end LUT\_7seg;

architecture Behavioral of LUT\_7seg is

type rom\_type is array (0 to 15) of std\_logic\_vector(6 downto 0);

signal sevenSegROM : rom\_type := (

"1000000", --0

"1111001", --1

"0100100", --2

"0110000", --3

"0011001", --4

"0010010", --5

"0000010", --6

"1111000", --7

"0000000", --8

"0000100", --9

"0001000", --A

"0000011", --B

"1000110", --C

"0100001", --D

"0000110", --E

"0001110" --F

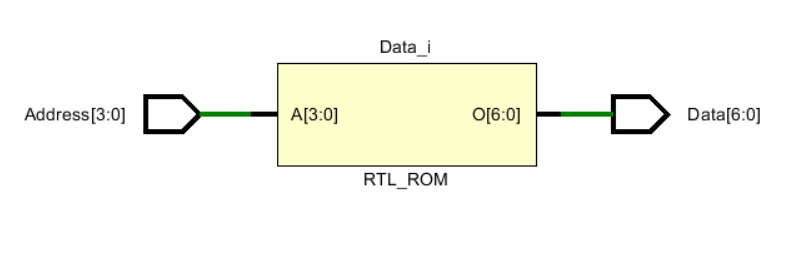
);

begin

Data <= sevenSegROM(to\_integer(unsigned(Address)));

end Behavioral;

## RTL SCHEMATICS



# REGISTER BANK

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

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

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

## VHDL CODE

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity RegisterBank is

Port ( RegBank\_in : in STD\_LOGIC\_VECTOR (3 downto 0);

RegEN : in STD\_LOGIC\_VECTOR (2 downto 0);

Reset : in STD\_LOGIC;

Clk : in STD\_LOGIC;

Reg0\_out : out STD\_LOGIC\_VECTOR (3 downto 0);

Reg1\_out : out STD\_LOGIC\_VECTOR (3 downto 0);

Reg2\_out : out STD\_LOGIC\_VECTOR (3 downto 0);

Reg3\_out : out STD\_LOGIC\_VECTOR (3 downto 0);

Reg4\_out : out STD\_LOGIC\_VECTOR (3 downto 0);

Reg5\_out : out STD\_LOGIC\_VECTOR (3 downto 0);

Reg6\_out : out STD\_LOGIC\_VECTOR (3 downto 0);

Reg7\_out : out STD\_LOGIC\_VECTOR (3 downto 0));

end RegisterBank;

architecture Behavioral of RegisterBank is

component Reg

Port ( D : in STD\_LOGIC\_VECTOR (3 downto 0);

En : in STD\_LOGIC;

R : in STD\_LOGIC;

Clk : in STD\_LOGIC;

Q : out STD\_LOGIC\_VECTOR (3 downto 0));

end component;

--works as register enabling unit

component Decoder\_3\_to\_8

Port ( I : in STD\_LOGIC\_VECTOR (2 downto 0);

EN : in STD\_LOGIC;

Y : out STD\_LOGIC\_VECTOR (7 downto 0));

end component;

signal in\_RegEN : STD\_LOGIC\_VECTOR (7 downto 0); --enabler for each register

begin

Decoder\_3\_to\_8\_0 : Decoder\_3\_to\_8

port map (

I(2 downto 0) => RegEN(2 downto 0),

EN => '1',

Y(7 downto 0) => in\_RegEN(7 downto 0));

Reg\_0 : Reg

port map(

D(3 downto 0) => "0000", -- Reg\_0 is readOnly - to execute negate instruction

En => '1', --set the Reg\_0 to 0000 initially

R => Reset,

Clk => Clk,

Q(3 downto 0) => Reg0\_out(3 downto 0));

Reg\_1 : Reg

port map(

D(3 downto 0) => RegBank\_in(3 downto 0),

En => in\_RegEN(1),

R => Reset,

Clk => Clk,

Q(3 downto 0) => Reg1\_out(3 downto 0));

Reg\_2 : Reg

port map(

D(3 downto 0) => RegBank\_in(3 downto 0),

En => in\_RegEN(2),

R => Reset,

Clk => Clk,

Q(3 downto 0) => Reg2\_out(3 downto 0));

Reg\_3 : Reg

port map(

D(3 downto 0) => RegBank\_in(3 downto 0),

En => in\_RegEN(3),

R => Reset,

Clk => Clk,

Q(3 downto 0) => Reg3\_out(3 downto 0));

Reg\_4 : Reg

port map(

D(3 downto 0) => RegBank\_in(3 downto 0),

En => in\_RegEN(4),

R => Reset,

Clk => Clk,

Q(3 downto 0) => Reg4\_out(3 downto 0));

Reg\_5 : Reg

port map(

D(3 downto 0) => RegBank\_in(3 downto 0),

En => in\_RegEN(5),

R => Reset,

Clk => Clk,

Q(3 downto 0) => Reg5\_out(3 downto 0));

Reg\_6 : Reg

port map(

D(3 downto 0) => RegBank\_in(3 downto 0),

En => in\_RegEN(6),

R => Reset,

Clk => Clk,

Q(3 downto 0) => Reg6\_out(3 downto 0));

Reg\_7 : Reg

port map(

D(3 downto 0) => RegBank\_in(3 downto 0),

En => in\_RegEN(7),

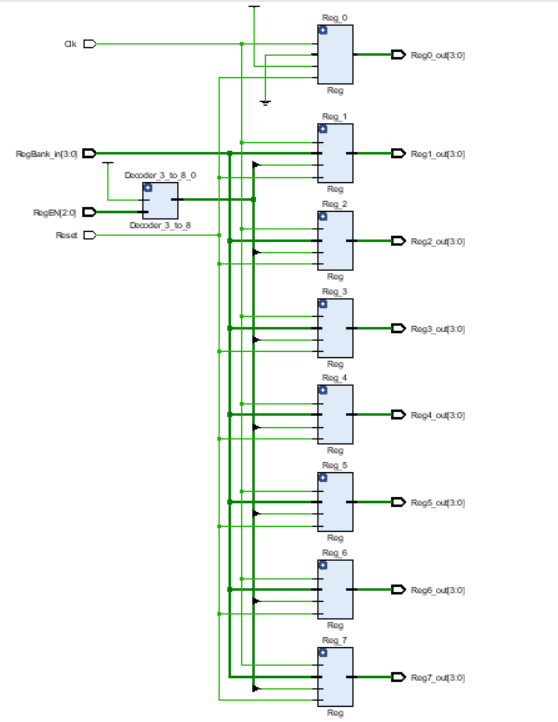
R => Reset,

Clk => Clk,

Q(3 downto 0) => Reg7\_out(3 downto 0));

end Behavioral;

## RTL SCHEMATICS



## BEHAVIORAL SIMULATION CODE

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Sim\_RegisterBank is

-- Port ( );

end Sim\_RegisterBank;

architecture Behavioral of Sim\_RegisterBank is

component RegisterBank is

Port ( RegBank\_in : in STD\_LOGIC\_VECTOR (3 downto 0);

RegEN : in STD\_LOGIC\_VECTOR (2 downto 0);

Reset : in STD\_LOGIC;

Clk : in STD\_LOGIC;

Reg0\_out : out STD\_LOGIC\_VECTOR (3 downto 0);

Reg1\_out : out STD\_LOGIC\_VECTOR (3 downto 0);

Reg2\_out : out STD\_LOGIC\_VECTOR (3 downto 0);

Reg3\_out : out STD\_LOGIC\_VECTOR (3 downto 0);

Reg4\_out : out STD\_LOGIC\_VECTOR (3 downto 0);

Reg5\_out : out STD\_LOGIC\_VECTOR (3 downto 0);

Reg6\_out : out STD\_LOGIC\_VECTOR (3 downto 0);

Reg7\_out : out STD\_LOGIC\_VECTOR (3 downto 0));

end component;

signal RegEN : std\_logic\_vector(2 downto 0);

signal RegBank\_in, Reg0\_out, Reg1\_out, Reg2\_out, Reg3\_out, Reg4\_out, Reg5\_out, Reg6\_out, Reg7\_out : std\_logic\_vector(3 downto 0);

signal Clk, Reset : std\_logic;

constant clock\_period : time := 10ns;

begin

UUT : RegisterBank port map(

RegBank\_in => RegBank\_in,

RegEN => RegEN,

Reset => Reset,

Clk => Clk,

Reg0\_out => Reg0\_out,

Reg1\_out => Reg1\_out,

Reg2\_out => Reg2\_out,

Reg3\_out => Reg3\_out,

Reg4\_out => Reg4\_out,

Reg5\_out => Reg5\_out,

Reg6\_out => Reg6\_out,

Reg7\_out => Reg7\_out);

clock\_process : process

begin

Clk <= '0';

wait for clock\_period / 2;

Clk <= '1';

wait for clock\_period / 2;

end process;

sim : process

begin

RegBank\_in <= "0011";

RegEN <= "010";

wait for 100ns;

RegEN <= "101";

wait for 100ns;

RegEN <= "111";

wait for 100ns;

Reset <= '1';

wait for 50ns;

Reset <= '0';

RegBank\_in <= "0101";

RegEN <= "111";

wait for 100ns;

RegEN <= "001";

wait for 50ns;

RegEN <= "010";

wait for 50ns;

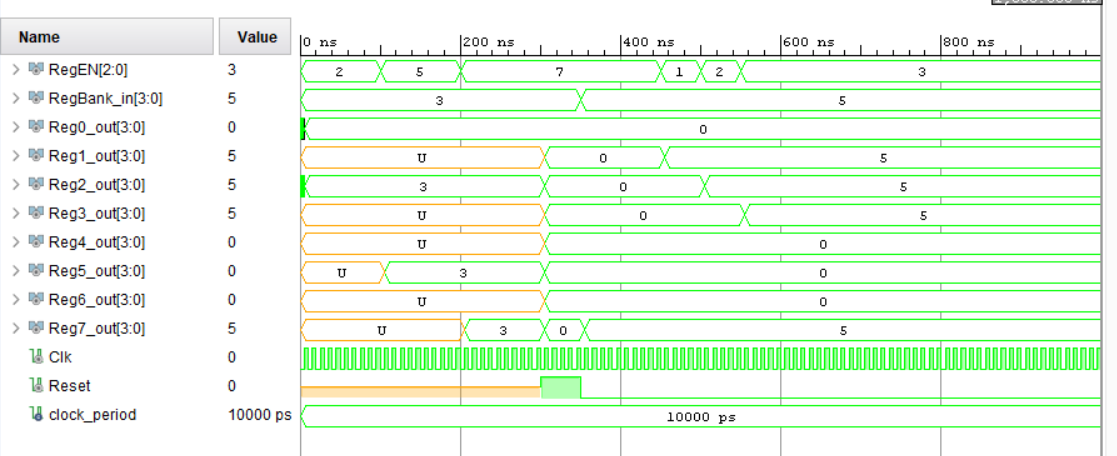
RegEN <= "011";

wait;

end process;

end Behavioral;

## TIMING DIAGRAM



# 4–BIT ADD/SUBTRACT UNIT

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

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

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

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

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

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

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

## VHDL CODE

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity AddSubUnit is

Port ( MuxA\_out : in STD\_LOGIC\_VECTOR (3 downto 0); --outputs of the mux -> inputs for the add/sub unit

MuxB\_out : in STD\_LOGIC\_VECTOR (3 downto 0);

Add\_Sub\_sel : in STD\_LOGIC; --add/sub unit selector line

Neg\_in : in STD\_LOGIC; --to calculate the negation of a given number

Add\_Sub\_out : out STD\_LOGIC\_VECTOR (3 downto 0); --output of the add/sub unit

overflow : out STD\_LOGIC; --carry out

zero : out STD\_LOGIC); --zero flag :- '1' if output is zero

end AddSubUnit;

architecture Behavioral of AddSubUnit is

component RCA\_4

Port ( A0 : in STD\_LOGIC;

A1 : in STD\_LOGIC;

A2 : in STD\_LOGIC;

A3 : in STD\_LOGIC;

B0 : in STD\_LOGIC;

B1 : in STD\_LOGIC;

B2 : in STD\_LOGIC;

B3 : in STD\_LOGIC;

C\_in : in STD\_LOGIC;

S0 : out STD\_LOGIC;

S1 : out STD\_LOGIC;

S2 : out STD\_LOGIC;

S3 : out STD\_LOGIC;

C2\_out : out STD\_LOGIC;

C3\_out : out STD\_LOGIC);

end component;

signal in\_A0, in\_A1, in\_A2, in\_A3 : STD\_LOGIC;

--define seperate equations in order to execute both add/sub and neg instruction

signal in\_RCA\_out : STD\_LOGIC\_VECTOR (3 downto 0);

--output of the ripple carry adder

signal in\_RCA\_carryOut2, in\_RCA\_carryOut3 : STD\_LOGIC;

--carry out of the ripple carry adder

begin

-- calculates either addition or negation, based on the instruction

in\_A0 <= MuxA\_out(0) xor Neg\_in;

in\_A1 <= MuxA\_out(1) xor Neg\_in;

in\_A2 <= MuxA\_out(2) xor Neg\_in;

in\_A3 <= MuxA\_out(3) xor Neg\_in;

RCA\_4\_0 : RCA\_4

port map(

A0 => in\_A0,

A1 => in\_A1,

A2 => in\_A2,

A3 => in\_A3,

B0 => MuxB\_out(0),

B1 => MuxB\_out(1),

B2 => MuxB\_out(2),

B3 => MuxB\_out(3),

C\_in => Neg\_in,

S0 => in\_RCA\_out(0),

S1 => in\_RCA\_out(1),

S2 => in\_RCA\_out(2),

S3 => in\_RCA\_out(3),

C2\_out => in\_RCA\_carryOut2,

C3\_out => in\_RCA\_carryOut3);

process(Add\_Sub\_sel, in\_RCA\_out)

begin

if (Add\_Sub\_sel = '1') then

Add\_Sub\_out <= in\_RCA\_out;

end if;

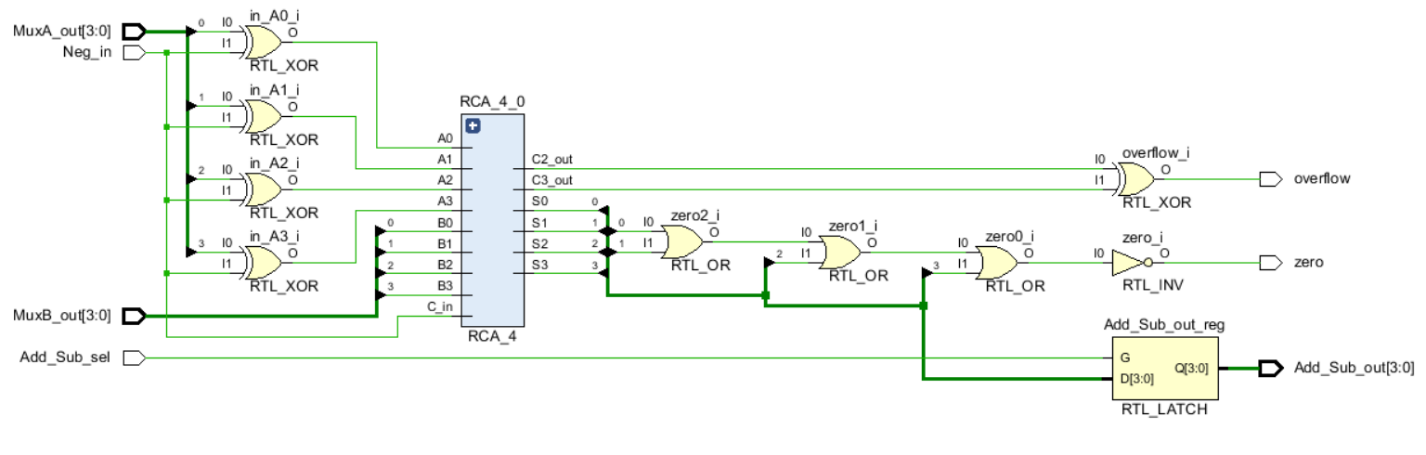
end process;

zero <= not(in\_RCA\_out(0) or in\_RCA\_out(1) or in\_RCA\_out(2) or in\_RCA\_out(3));

overflow <= in\_RCA\_carryOut2 xor in\_RCA\_carryOut3;

end Behavioral;

## RTL SCHEMATICS



## BEHAVIORAL SIMULATION CODE

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Sim\_AddSubUnit is

-- Port ( );

end Sim\_AddSubUnit;

architecture Behavioral of Sim\_AddSubUnit is

component AddSubUnit is

Port ( MuxA\_out : in STD\_LOGIC\_VECTOR (3 downto 0); --outputs of the mux -> inputs for the add/sub unit

MuxB\_out : in STD\_LOGIC\_VECTOR (3 downto 0);

Add\_Sub\_sel : in STD\_LOGIC; --add/sub unit selector line

Neg\_in : in STD\_LOGIC; --to calculate the negation of a given number

Add\_Sub\_out : out STD\_LOGIC\_VECTOR (3 downto 0); --output of the add/sub unit

overflow : out STD\_LOGIC; --carry out

zero : out STD\_LOGIC);

end component;

signal MuxA\_out, MuxB\_out, Add\_Sub\_out : std\_logic\_vector(3 downto 0);

signal Add\_Sub\_Sel, Neg\_in, overflow, zero : std\_logic;

begin

UUT : AddSubUnit port map(

MuxA\_out => MuxA\_out,

MuxB\_out => MuxB\_out,

Add\_Sub\_Sel => Add\_Sub\_Sel,

Neg\_in => Neg\_in,

Add\_Sub\_out => Add\_Sub\_out,

overflow => overflow,

zero => zero);

--number range that can be shown in 4 bits = (-8) to 7

process

begin

MuxA\_out <= "1010"; -- (-6)

MuxB\_out <= "1100"; -- (-4)

Add\_Sub\_sel <= '1';

Neg\_in <= '0';

wait for 100ns;

MuxB\_out <= "0001"; -- 1

wait for 100ns;

MuxA\_out <= "0011"; -- 3

MuxB\_out <= "0000"; -- 0

Neg\_in <= '1';

wait for 100ns;

MuxA\_out <= "1111"; -- (-1)

MuxB\_out <= "0001"; -- 1

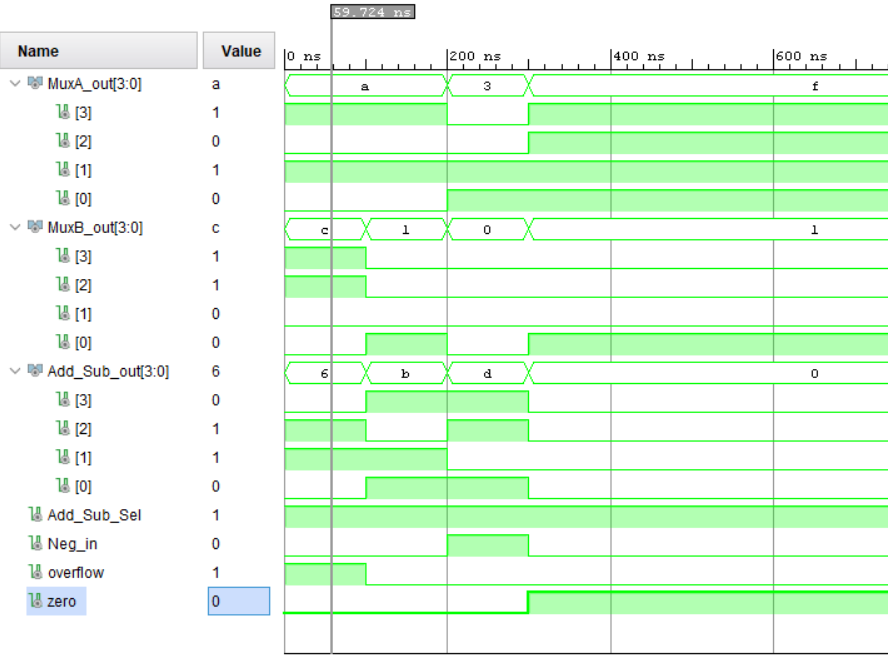
Neg\_in <= '0';

wait;

end process;

end Behavioral;

## TIMING DIAGRAM



# 4-BIT RIPPLE CARRY ADDER

The RCA\_4 implemented in the previous lab was modified to use in the nano processor.

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

## VHDL CODE

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity RCA\_4 is

Port ( A0 : in STD\_LOGIC;

A1 : in STD\_LOGIC;

A2 : in STD\_LOGIC;

A3 : in STD\_LOGIC;

B0 : in STD\_LOGIC;

B1 : in STD\_LOGIC;

B2 : in STD\_LOGIC;

B3 : in STD\_LOGIC;

C\_in : in STD\_LOGIC;

S0 : out STD\_LOGIC;

S1 : out STD\_LOGIC;

S2 : out STD\_LOGIC;

S3 : out STD\_LOGIC;

C2\_out : out STD\_LOGIC;

C3\_out : out STD\_LOGIC);

end RCA\_4;

architecture Behavioral of RCA\_4 is

component FA

port (

A : in std\_logic;

B : in std\_logic;

C\_in : in std\_logic;

S : out std\_logic;

C\_out : out std\_logic);

end component;

signal FA0\_S, FA0\_C, FA1\_S, FA1\_C, FA2\_S, FA2\_C, FA3\_S, FA3\_C : std\_logic;

begin

FA\_0 : FA

port map (

A => A0,

B => B0,

C\_in => C\_in,

S => S0,

C\_Out => FA0\_C);

FA\_1 : FA

port map (

A => A1,

B => B1,

C\_in => FA0\_C,

S => S1,

C\_Out => FA1\_C);

FA\_2 : FA

port map (

A => A2,

B => B2,

C\_in => FA1\_C,

S => S2,

C\_Out => FA2\_C);

FA\_3 : FA

port map (

A => A3,

B => B3,

C\_in => FA2\_C,

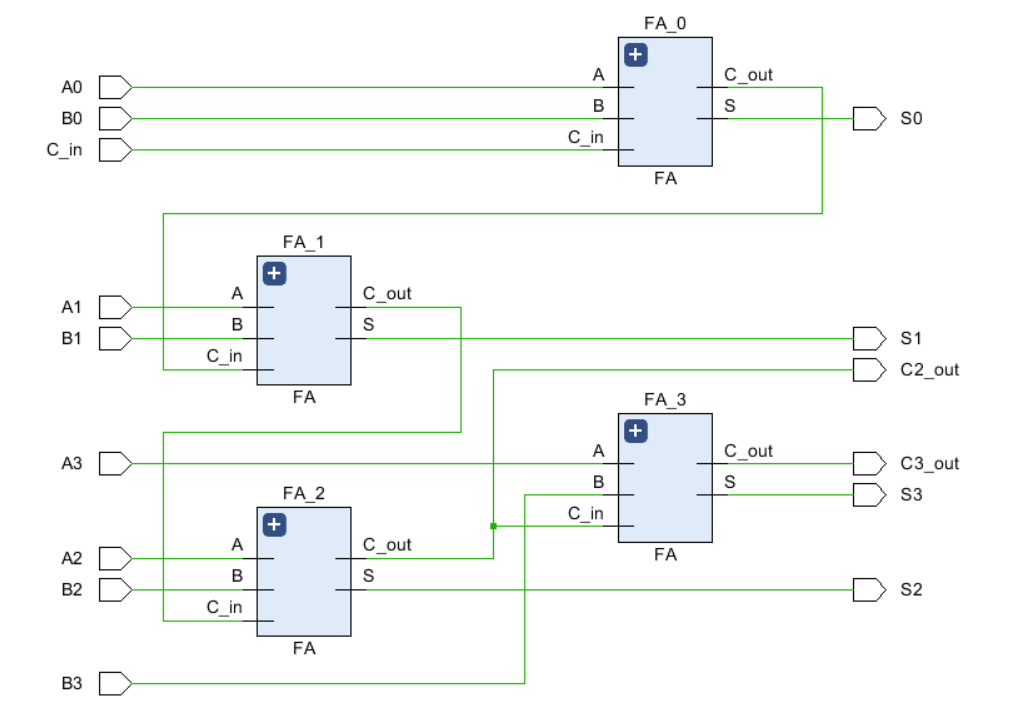
S => S3,

C\_Out => C3\_out);

C2\_out <= FA2\_C;

end Behavioral;

## RTL SCHEMATIC



PROGRAM COUNTER

The program counter decides which instruction gets executed next.

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

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

VHDL CODE

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity ProgramCounter is

Port ( Clk : in STD\_LOGIC;

Reset : in STD\_LOGIC;

JumpFlag : in STD\_LOGIC;

JumpAdd : in STD\_LOGIC\_VECTOR (2 downto 0);

MemSel : out STD\_LOGIC\_VECTOR (2 downto 0));

end ProgramCounter;

architecture Behavioral of ProgramCounter is

component PC\_Reg is

Port ( D : in STD\_LOGIC\_VECTOR (2 downto 0);

R : in STD\_LOGIC;

Clk : in STD\_LOGIC;

Q : out STD\_LOGIC\_VECTOR (2 downto 0));

end component;

component RCA\_3 is

Port ( A0 : in STD\_LOGIC;

A1 : in STD\_LOGIC;

A2 : in STD\_LOGIC;

B0 : in STD\_LOGIC;

B1 : in STD\_LOGIC;

B2 : in STD\_LOGIC;

C\_in : in STD\_LOGIC;

S0 : out STD\_LOGIC;

S1 : out STD\_LOGIC;

S2 : out STD\_LOGIC;

C\_out : out STD\_LOGIC);

end component;

component Mux2\_3bit is

Port ( AdderOutput : in STD\_LOGIC\_VECTOR (2 downto 0);

AddressJump : in STD\_LOGIC\_VECTOR (2 downto 0);

jumpFlag : in STD\_LOGIC;

mux\_out : out STD\_LOGIC\_VECTOR (2 downto 0));

end component;

signal Mux\_in : std\_logic\_vector(2 downto 0);

signal nextIns : std\_logic\_vector(2 downto 0);

signal currIns : std\_logic\_vector(2 downto 0);

begin

PC\_Reg\_0 : PC\_Reg

port map(

D => nextIns,

R => Reset,

Clk => Clk,

Q => currIns);

RCA\_3\_0 : RCA\_3

port map(

A0 => currIns(0),

A1 => currIns(1),

A2 => currIns(2),

B0 => '0',

B1 => '0',

B2 => '0',

C\_in => '1',

S0 => Mux\_in(0),

S1 => Mux\_in(1),

S2 => Mux\_in(2)); -- C\_out is not included

Mux2\_3bit\_0 : Mux2\_3bit

port map(

AdderOutput => Mux\_in,

AddressJump => JumpAdd,

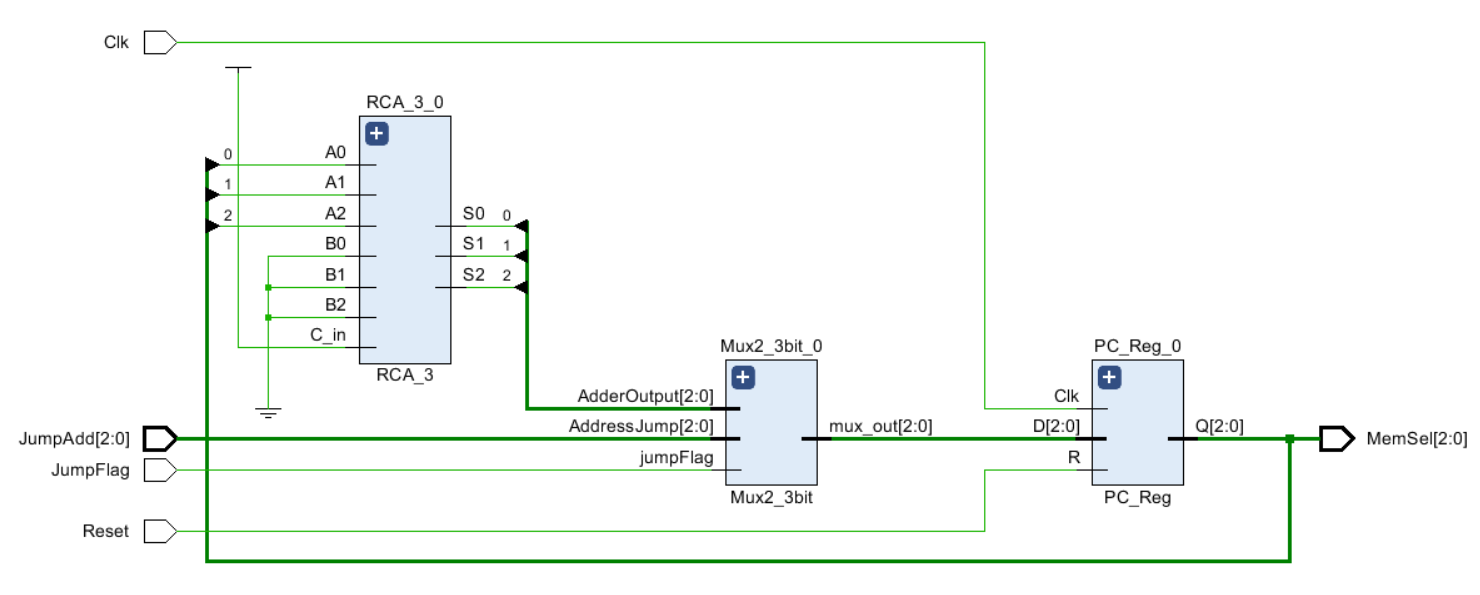
jumpFlag => JumpFlag,

mux\_out => nextIns);

MemSel <= currIns;

end Behavioral;

RTL SCHEMATICS



BEHAVIORAL SIMULATION CODE

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Sim\_ProgramCounter is

-- Port ( );

end Sim\_ProgramCounter;

architecture Behavioral of Sim\_ProgramCounter is

component ProgramCounter is

Port ( Clk : in STD\_LOGIC;

Reset : in STD\_LOGIC;

JumpFlag : in STD\_LOGIC;

JumpAdd : in STD\_LOGIC\_VECTOR (2 downto 0);

MemSel : inout STD\_LOGIC\_VECTOR (2 downto 0));

end component;

signal Clk, Reset, JumpFlag : std\_logic;

signal JumpAdd, MemSel : std\_logic\_vector(2 downto 0);

constant clock\_period : time := 10ns;

begin

UUT : ProgramCounter port map(

Clk => Clk,

Reset => Reset,

JumpFlag => JumpFlag,

JumpAdd => JumpAdd,

MemSel => MemSel);

clock\_process : process

begin

Clk <= '0';

wait for clock\_period / 2;

Clk <= '1';

wait for clock\_period / 2;

end process;

sim : process

begin

JumpFlag <= '0';

wait for 50ns;

JumpFlag <= '1';

JumpAdd <= "011";

wait for clock\_period;

JumpFlag <= '0';

Reset <= '1';

wait for 50ns;

Reset <= '0';

JumpFlag <= '1';

JumpAdd <= "101";

wait for clock\_period;

JumpFlag <= '0';

wait for 100ns;

JumpFlag <= '1';

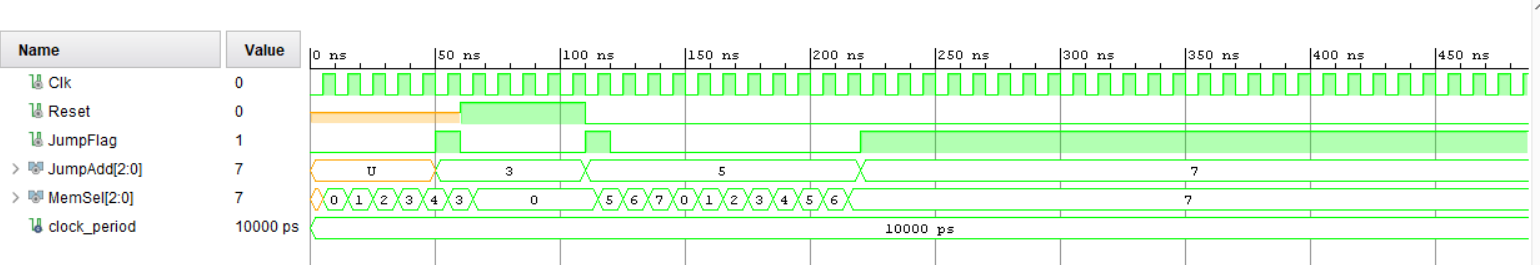
JumpAdd <= "111";

wait;

end process;

end Behavioral;

TIMING DIAGRAM



PC REGISTER

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

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

VHDL CODE

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity PC\_Reg is

Port ( D : in STD\_LOGIC\_VECTOR (2 downto 0);

R : in STD\_LOGIC; --reset push button

Clk : in STD\_LOGIC;

Q : out STD\_LOGIC\_VECTOR (2 downto 0)); --memory select

end PC\_Reg;

architecture Behavioral of PC\_Reg is

begin

process (Clk) begin

if (rising\_edge(Clk)) then

if R = '1' then

Q <= "000";

else

if D = "UUU" then

Q <= "000";

else

Q <= D;

end if;

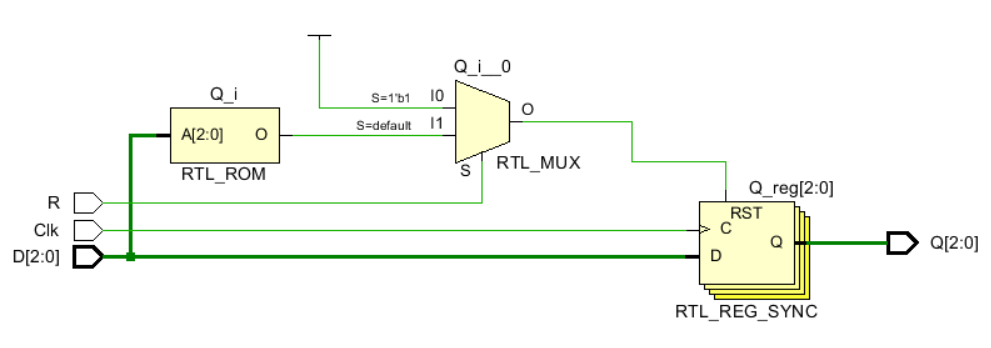
end if;

end if;

end process;

end Behavioral;

RTL SCHEMATICS



BEHAVIORAL SIMULATION CODE

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Sim\_PC\_Reg is

-- Port ( );

end Sim\_PC\_Reg;

architecture Behavioral of Sim\_PC\_Reg is

component PC\_Reg is

Port ( D : in STD\_LOGIC\_VECTOR (2 downto 0);

R : in STD\_LOGIC; --reset push button

Clk : in STD\_LOGIC;

Q : out STD\_LOGIC\_VECTOR (2 downto 0)); --memory select

end component;

signal D, Q : std\_logic\_vector(2 downto 0);

signal R, Clk : std\_logic;

constant clock\_period : time := 10ns;

begin

UUT : PC\_Reg port map(

D => D,

R => R,

Clk => Clk,

Q => Q);

clock\_process : process

begin

Clk <= '0';

wait for clock\_period / 2;

Clk <= '1';

wait for clock\_period / 2;

end process;

sim : process

begin

D <= "010";

R <= '0';

wait for 100ns;

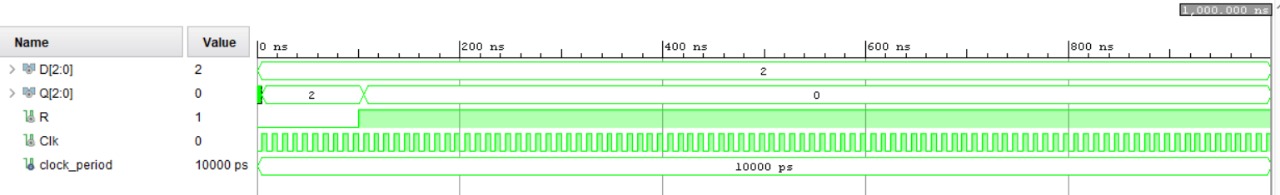
R <= '1';

wait;

end process;

end Behavioral;

TIMING DIAGRAM



# 3–BIT ADDER

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

## VHDL CODE

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity RCA\_3 is

Port ( A0 : in STD\_LOGIC;

A1 : in STD\_LOGIC;

A2 : in STD\_LOGIC;

B0 : in STD\_LOGIC;

B1 : in STD\_LOGIC;

B2 : in STD\_LOGIC;

C\_in : in STD\_LOGIC;

S0 : out STD\_LOGIC;

S1 : out STD\_LOGIC;

S2 : out STD\_LOGIC;

C\_out : out STD\_LOGIC);

end RCA\_3;

architecture Behavioral of RCA\_3 is

component FA

port (

A : in std\_logic;

B : in std\_logic;

C\_in : in std\_logic;

S : out std\_logic;

C\_out : out std\_logic);

end component;

signal FA0\_C, FA1\_C : std\_logic;

begin

FA\_0 : FA

port map (

A => A0,

B => B0,

C\_in => C\_in,

S => S0,

C\_Out => FA0\_C);

FA\_1 : FA

port map (

A => A1,

B => B1,

C\_in => FA0\_C,

S => S1,

C\_Out => FA1\_C);

FA\_2 : FA

port map (

A => A2,

B => B2,

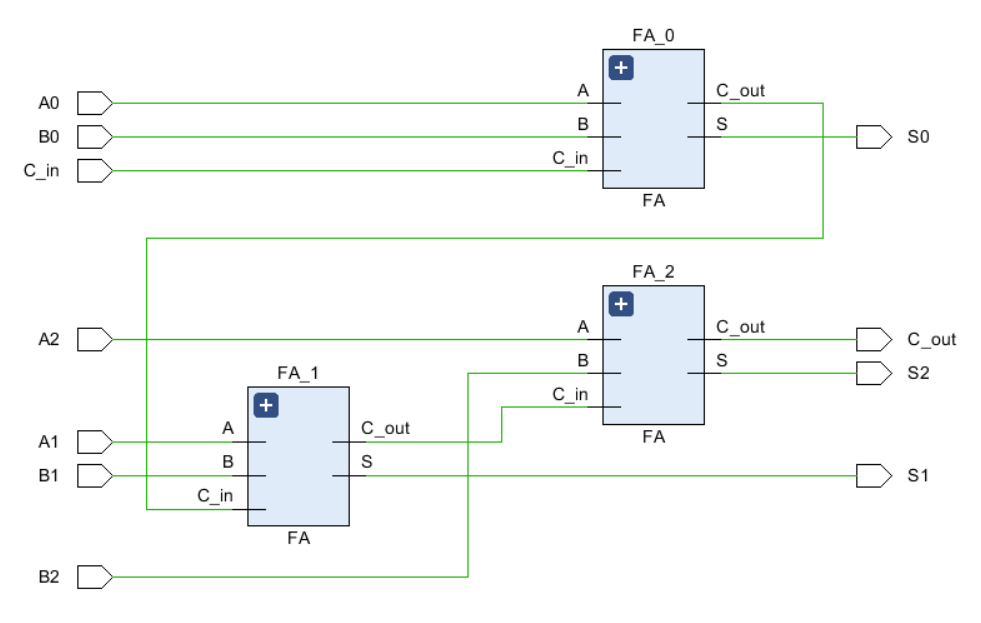
C\_in => FA1\_C,

S => S2,

C\_Out => C\_Out);

end Behavioral;

## RTL SCHEMATICS



## BEHAVIORAL SIMULATION CODE

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Sim\_RCA\_3 is

-- Port ( );

end Sim\_RCA\_3;

architecture Behavioral of Sim\_RCA\_3 is

component RCA\_3 is

Port ( A0 : in STD\_LOGIC;

A1 : in STD\_LOGIC;

A2 : in STD\_LOGIC;

B0 : in STD\_LOGIC;

B1 : in STD\_LOGIC;

B2 : in STD\_LOGIC;

C\_in : in STD\_LOGIC;

S0 : out STD\_LOGIC;

S1 : out STD\_LOGIC;

S2 : out STD\_LOGIC;

C\_out : out STD\_LOGIC);

end component;

signal A0, A1, A2, B0, B1, B2, S0, S1, S2, C\_in, C\_out : std\_logic;

begin

UUT : RCA\_3 port map(

A0 => A0,

A1 => A1,

A2 => A2,

B0 => B0,

B1 => B1,

B2 => B2,

C\_in => C\_in,

S0 => S0,

S1 => S1,

S2 => S2,

C\_out => C\_out);

process

begin

B0 <= '0';

B1 <= '0';

B2 <= '0';

C\_in <= '1';

A0 <= '0';

A1 <= '1';

A2 <= '1';

wait for 100ns;

A1 <= '0';

wait for 100ns;

A1 <= '1';

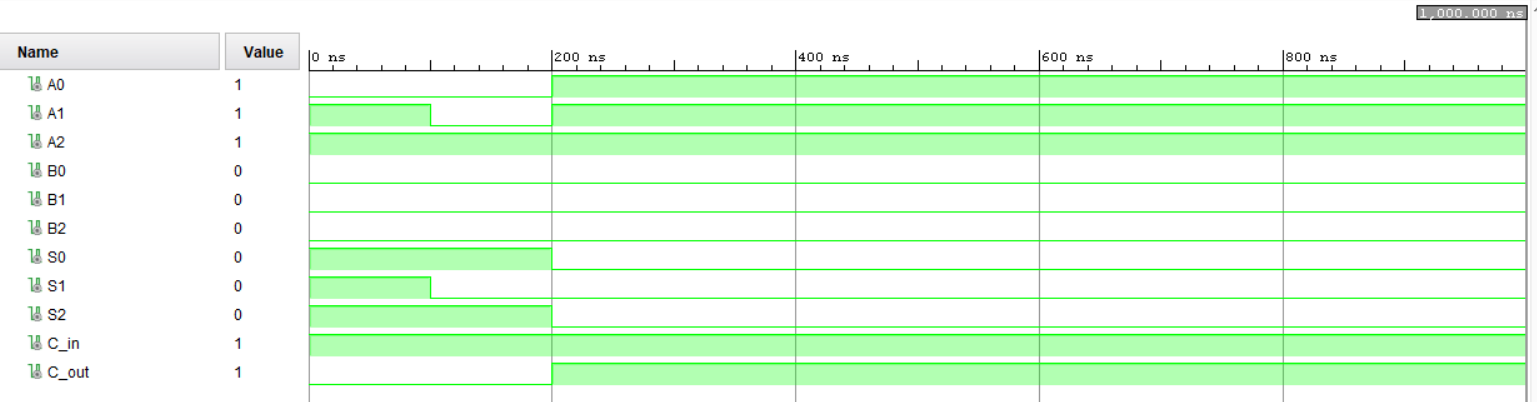
A0 <= '1';

wait;

end process;

end Behavioral;

## TIMING DIAGRAM



# K–WAY B–BIT MUX

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

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

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

## 8–WAY 4–BIT MUX

### VHDL CODE

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Mux8\_4bit is

Port ( A0 : in STD\_LOGIC\_VECTOR (3 downto 0);

A1 : in STD\_LOGIC\_VECTOR (3 downto 0);

A2 : in STD\_LOGIC\_VECTOR (3 downto 0);

A3 : in STD\_LOGIC\_VECTOR (3 downto 0);

A4 : in STD\_LOGIC\_VECTOR (3 downto 0);

A5 : in STD\_LOGIC\_VECTOR (3 downto 0);

A6 : in STD\_LOGIC\_VECTOR (3 downto 0);

A7 : in STD\_LOGIC\_VECTOR (3 downto 0);

RegSel : in STD\_LOGIC\_VECTOR (2 downto 0);

Q : out STD\_LOGIC\_VECTOR (3 downto 0));

end Mux8\_4bit;

architecture Behavioral of Mux8\_4bit is

begin

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

begin

case RegSel is

when "000" => Q <= A0;

when "001" => Q <= A1;

when "010" => Q <= A2;

when "011" => Q <= A3;

when "100" => Q <= A4;

when "101" => Q <= A5;

when "110" => Q <= A6;

when "111" => Q <= A7;

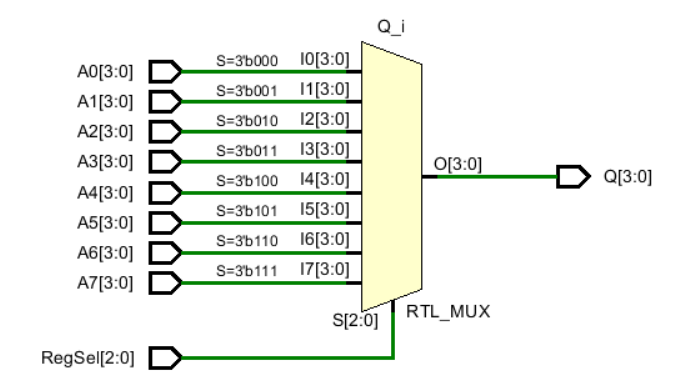
when others => Q <= "0000";

end case;

end process;

end Behavioral;

### RTL SCHEMATICS



### BEHAVIORAL SIMULATION CODE

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Sim\_Mux8\_4bit is

-- Port ( );

end Sim\_Mux8\_4bit;

architecture Behavioral of Sim\_Mux8\_4bit is

component Mux8\_4bit is

Port ( A0 : in STD\_LOGIC\_VECTOR (3 downto 0);

A1 : in STD\_LOGIC\_VECTOR (3 downto 0);

A2 : in STD\_LOGIC\_VECTOR (3 downto 0);

A3 : in STD\_LOGIC\_VECTOR (3 downto 0);

A4 : in STD\_LOGIC\_VECTOR (3 downto 0);

A5 : in STD\_LOGIC\_VECTOR (3 downto 0);

A6 : in STD\_LOGIC\_VECTOR (3 downto 0);

A7 : in STD\_LOGIC\_VECTOR (3 downto 0);

RegSel : in STD\_LOGIC\_VECTOR (2 downto 0);

Q : out STD\_LOGIC\_VECTOR (3 downto 0));

end component;

signal A0, A1, A2, A3, A4, A5, A6, A7, Q : std\_logic\_vector(3 downto 0);

signal RegSel : std\_logic\_vector(2 downto 0);

begin

UUT : Mux8\_4bit port map(

A0 => A0,

A1 => A1,

A2 => A2,

A3 => A3,

A4 => A4,

A5 => A5,

A6 => A6,

A7 => A7,

RegSel => RegSel,

Q => Q);

process

begin

A0 <= "0000";

A1 <= "0001";

A2 <= "0010";

A3 <= "0011";

A4 <= "0100";

A5 <= "0101";

A6 <= "0110";

A7 <= "0111";

RegSel <= "011";

wait for 100ns;

RegSel <= "101";

wait for 100ns;

RegSel <= "111";

wait for 100ns;

RegSel <= "010";

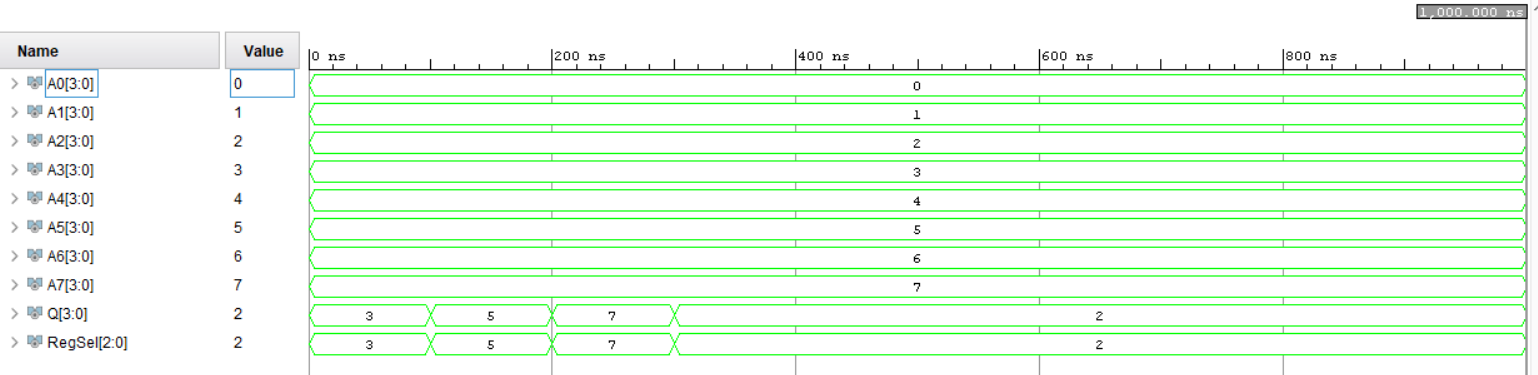
wait for 100ns;

wait;

end process;

end Behavioral;

### TIMING DIAGRAM



## 2–WAY 4–BIT MUX

### VHDL CODE

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Mux2\_4bit is

Port ( immVal : in STD\_LOGIC\_VECTOR (3 downto 0);

addSub\_result : in STD\_LOGIC\_VECTOR (3 downto 0);

loadSel : in STD\_LOGIC; --loadSel <= Mov

mux\_out : out STD\_LOGIC\_VECTOR (3 downto 0));

end Mux2\_4bit;

architecture Behavioral of Mux2\_4bit is

begin

process(immVal, addSub\_result, loadSel)

begin

case loadSel is

when '0' => mux\_out <= addSub\_result;

when '1' => mux\_out <= immVal;

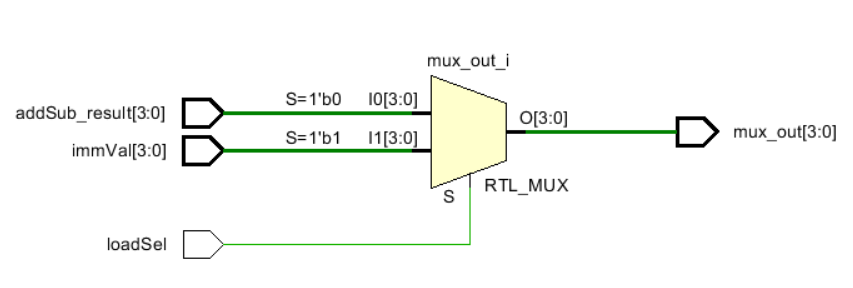
when others => mux\_out <= "0000";

end case;

end process;

end Behavioral;

### RTL SCHEMATICS



### BEHAVIORAL SIMULATION CODE

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Sim\_Mux2\_4bit is

-- Port ( );

end Sim\_Mux2\_4bit;

architecture Behavioral of Sim\_Mux2\_4bit is

component Mux2\_4bit is

Port ( immVal : in STD\_LOGIC\_VECTOR (3 downto 0);

addSub\_result : in STD\_LOGIC\_VECTOR (3 downto 0);

loadSel : in STD\_LOGIC; --loadSel <= Mov

mux\_out : out STD\_LOGIC\_VECTOR (3 downto 0));

end component;

signal immVal, addSub\_result, mux\_out : std\_logic\_vector(3 downto 0);

signal loadSel : std\_logic;

begin

UUT : Mux2\_4bit port map(

immVal => immVal,

addSub\_result => addSub\_result,

loadSel => loadSel,

mux\_out => mux\_out);

process

begin

immVal <= "0101";

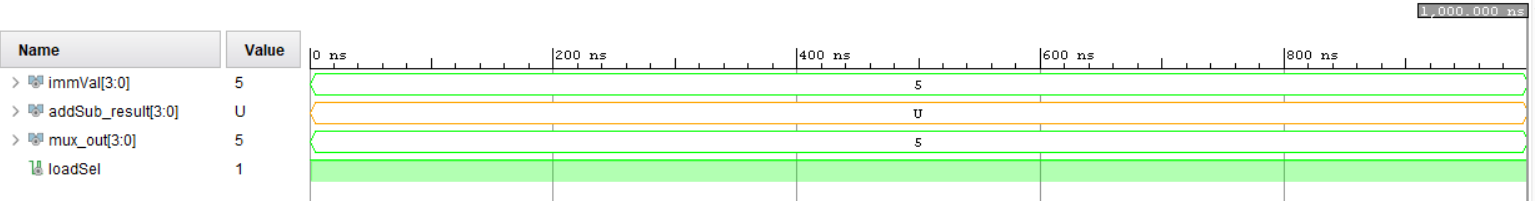
loadSel <= '1';

wait;

end process;

end Behavioral;

### TIMING DIAGRAM



## 2–WAY 3–BIT MUX

### VHDL CODE

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Mux2\_3bit is

Port ( AdderOutput : in STD\_LOGIC\_VECTOR (2 downto 0);

AddressJump : in STD\_LOGIC\_VECTOR (2 downto 0);

jumpFlag : in STD\_LOGIC;

mux\_out : out STD\_LOGIC\_VECTOR (2 downto 0));

end Mux2\_3bit;

architecture Behavioral of Mux2\_3bit is

begin

process(AdderOutput, AddressJump, jumpFlag)

begin

case jumpFlag is

when '1' => mux\_out <= AddressJump;

when '0' => mux\_out <= AdderOutput;

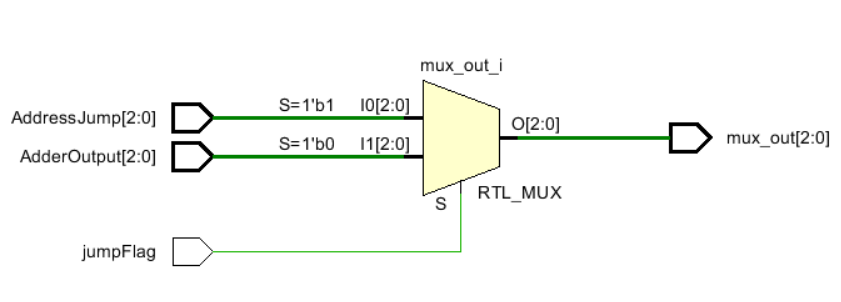
when others => mux\_out <= "000";

end case;

end process;

end Behavioral;

### RTL SCHEMATICS



### BEHAVIORAL SIMULATION CODE

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Mux2\_3bit is

Port ( AdderOutput : in STD\_LOGIC\_VECTOR (2 downto 0);

AddressJump : in STD\_LOGIC\_VECTOR (2 downto 0);

jumpFlag : in STD\_LOGIC;

mux\_out : out STD\_LOGIC\_VECTOR (2 downto 0));

end Mux2\_3bit;

architecture Behavioral of Mux2\_3bit is

begin

process(AdderOutput, AddressJump, jumpFlag)

begin

case jumpFlag is

when '1' => mux\_out <= AddressJump;

when '0' => mux\_out <= AdderOutput;

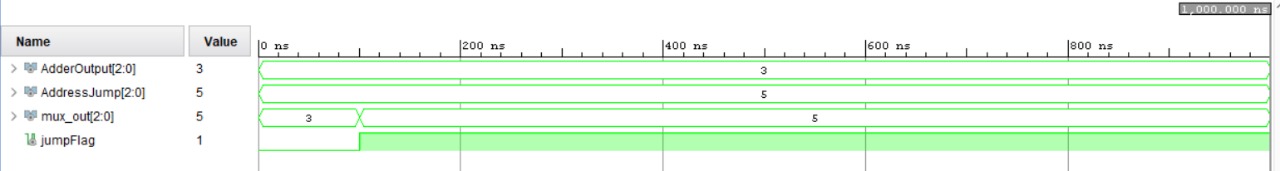
when others => mux\_out <= "000";

end case;

end process;

end Behavioral;

### TIMING DIAGRAM



# REGISTER

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

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

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

## VHDL CODE

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Reg is

Port ( D : in STD\_LOGIC\_VECTOR (3 downto 0);

En : in STD\_LOGIC;

R : in STD\_LOGIC;

Clk : in STD\_LOGIC;

Q : out STD\_LOGIC\_VECTOR (3 downto 0));

end Reg;

architecture Behavioral of Reg is

begin

process (Clk) begin

if (rising\_edge(Clk)) then --respond when clock rises

if En = '1' or R = '1' then --Enable should be set

if R = '1' then

Q <= "0000";

else

Q <= D;

end if;

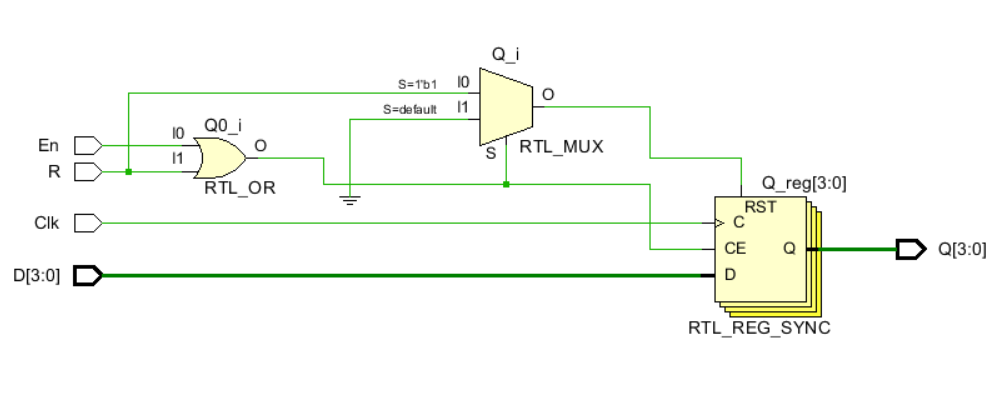
end if;

end if;

end process;

end Behavioral;

## RTL SCHEMATICS



## BEHAVIORAL SIMULATION CODE

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Sim\_Register is

-- Port ( );

end Sim\_Register;

architecture Behavioral of Sim\_Register is

component Reg is

Port ( D : in STD\_LOGIC\_VECTOR (3 downto 0);

En : in STD\_LOGIC;

R : in STD\_LOGIC;

Clk : in STD\_LOGIC;

Q : out STD\_LOGIC\_VECTOR (3 downto 0));

end component;

signal D, Q : std\_logic\_vector(3 downto 0);

signal En, R, Clk : std\_logic;

signal clock\_period : time := 10ns;

begin

UUT : Reg port map(

D => D,

En => En,

R => R,

Clk => Clk,

Q => Q);

clock\_process : process

begin

Clk <= '0';

wait for clock\_period / 2;

Clk <= '1';

wait for clock\_period / 2;

end process;

sim : process

begin

D <= "0101";

En <= '1';

R <= '0';

wait for 100ns;

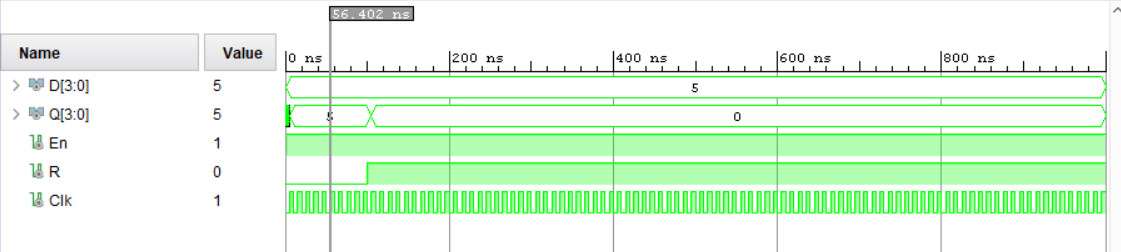
R <= '1';

wait;

end process;

end Behavioral;

## TIMING DIAGRAM



# SLOW CLOCK

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

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

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

## VHDL CODE

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Slow\_Clk is

Port ( Clk\_in : in STD\_LOGIC;

Clk\_out : out STD\_LOGIC);

end Slow\_Clk;

architecture Behavioral of Slow\_Clk is

signal count : integer := 1;

signal clk\_status : std\_logic := '0';

begin

--For 100MHz input clock, this generates 1Hz clock

process (Clk\_in) begin

if (rising\_edge(Clk\_in)) then

count <= count + 1;

if (count = 2) then --count 50M pulses(1/2 of period)

clk\_status <= not clk\_status; --Invert clock status

Clk\_out <= clk\_status;

count <= 1; --Reset counter

end if;

end if;

end process;

end Behavioral;

# NANO PROCESSOR

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

## VHDL CODE

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity NanoProcessor is

Port ( Clk : in STD\_LOGIC;

Reset : in STD\_LOGIC;

Overflow : out STD\_LOGIC;

Zero : out STD\_LOGIC;

S\_out : out STD\_LOGIC\_VECTOR (3 downto 0);

sevenSegOut : out STD\_LOGIC\_VECTOR(6 downto 0));

end NanoProcessor;

architecture Behavioral of NanoProcessor is

component Slow\_Clk is

Port ( Clk\_in : in STD\_LOGIC;

Clk\_out : out STD\_LOGIC);

end component;

signal slwClkOut : std\_logic;

component Ins\_Decoder is

Port ( Instruction : in STD\_LOGIC\_VECTOR (11 downto 0);

JumpCheck : in STD\_LOGIC\_VECTOR (3 downto 0);

Reg\_En : out STD\_LOGIC\_VECTOR (2 downto 0);

Load\_Sel : out STD\_LOGIC;

Imm\_Val : out STD\_LOGIC\_VECTOR (3 downto 0);

Reg\_Sel\_A : out STD\_LOGIC\_VECTOR (2 downto 0);

Reg\_Sel\_B : out STD\_LOGIC\_VECTOR (2 downto 0);

AddSub\_Sel : out STD\_LOGIC;

Jump\_Flag : out STD\_LOGIC;

Jump\_Add : out STD\_LOGIC\_VECTOR (2 downto 0);

Neg\_Sel : out STD\_LOGIC);

end component;

signal ins : std\_logic\_vector(11 downto 0);

signal immVal : std\_logic\_vector(3 downto 0);

signal regEn, regSelA, regSelB, jmpAdd : std\_logic\_vector(2 downto 0);

signal loadSel, addSubSel, jmpFlag, negSel : std\_logic;

component ProgramCounter is

Port ( Clk : in STD\_LOGIC;

Reset : in STD\_LOGIC;

JumpFlag : in STD\_LOGIC;

JumpAdd : in STD\_LOGIC\_VECTOR (2 downto 0);

MemSel : out STD\_LOGIC\_VECTOR (2 downto 0));

end component;

signal memSel : std\_logic\_vector(2 downto 0);

component Program\_ROM is

Port ( Mem\_Sel : in STD\_LOGIC\_VECTOR (2 downto 0);

Instruction : out STD\_LOGIC\_VECTOR (11 downto 0));

end component;

component RegisterBank is

Port ( RegBank\_in : in STD\_LOGIC\_VECTOR (3 downto 0);

RegEN : in STD\_LOGIC\_VECTOR (2 downto 0);

Reset : in STD\_LOGIC;

Clk : in STD\_LOGIC;

Reg0\_out : out STD\_LOGIC\_VECTOR (3 downto 0);

Reg1\_out : out STD\_LOGIC\_VECTOR (3 downto 0);

Reg2\_out : out STD\_LOGIC\_VECTOR (3 downto 0);

Reg3\_out : out STD\_LOGIC\_VECTOR (3 downto 0);

Reg4\_out : out STD\_LOGIC\_VECTOR (3 downto 0);

Reg5\_out : out STD\_LOGIC\_VECTOR (3 downto 0);

Reg6\_out : out STD\_LOGIC\_VECTOR (3 downto 0);

Reg7\_out : out STD\_LOGIC\_VECTOR (3 downto 0));

end component;

signal regBankIn, r0, r1, r2, r3, r4, r5, r6, r7 : std\_logic\_vector(3 downto 0);

component Mux8\_4bit is

Port ( A0 : in STD\_LOGIC\_VECTOR (3 downto 0);

A1 : in STD\_LOGIC\_VECTOR (3 downto 0);

A2 : in STD\_LOGIC\_VECTOR (3 downto 0);

A3 : in STD\_LOGIC\_VECTOR (3 downto 0);

A4 : in STD\_LOGIC\_VECTOR (3 downto 0);

A5 : in STD\_LOGIC\_VECTOR (3 downto 0);

A6 : in STD\_LOGIC\_VECTOR (3 downto 0);

A7 : in STD\_LOGIC\_VECTOR (3 downto 0);

RegSel : in STD\_LOGIC\_VECTOR (2 downto 0);

Q : out STD\_LOGIC\_VECTOR (3 downto 0));

end component;

signal muxOutA, muxOutB : std\_logic\_vector(3 downto 0);

component AddSubUnit is

Port ( MuxA\_out : in STD\_LOGIC\_VECTOR (3 downto 0);

MuxB\_out : in STD\_LOGIC\_VECTOR (3 downto 0);

Add\_Sub\_sel : in STD\_LOGIC;

Neg\_in : in STD\_LOGIC;

Add\_Sub\_out : out STD\_LOGIC\_VECTOR (3 downto 0);

overflow : out STD\_LOGIC;

zero : out STD\_LOGIC);

end component;

signal addSubOut : std\_logic\_vector(3 downto 0);

component Mux2\_4bit is

Port ( immVal : in STD\_LOGIC\_VECTOR (3 downto 0);

addSub\_result : in STD\_LOGIC\_VECTOR (3 downto 0);

loadSel : in STD\_LOGIC;

mux\_out : out STD\_LOGIC\_VECTOR (3 downto 0));

end component;

component LUT\_7seg is

Port ( Address : in STD\_LOGIC\_VECTOR (3 downto 0);

Data : out STD\_LOGIC\_VECTOR (6 downto 0));

end component;

begin

Slow\_Clk\_0 : Slow\_Clk

port map(

Clk\_in => Clk, --connected from direct input

Clk\_out => slwClkOut);

RegisterBank\_0 : RegisterBank

port map(

RegBank\_in => regBankIn, --in

RegEN => regEn, --in

Reset => Reset,

Clk => slwClkOut, --in

Reg0\_out => r0,

Reg1\_out => r1,

Reg2\_out => r2,

Reg3\_out => r3,

Reg4\_out => r4,

Reg5\_out => r5,

Reg6\_out => r6,

Reg7\_out => r7);

Mux8\_4bit\_A : Mux8\_4bit

port map(

A0 => r0, --in

A1 => r1, --in

A2 => r2, --in

A3 => r3, --in

A4 => r4, --in

A5 => r5, --in

A6 => r6, --in

A7 => r7, --in

RegSel => regSelA, --in

Q => muxOutA);

Mux8\_4bit\_B : Mux8\_4bit

port map(

A0 => r0, --in

A1 => r1, --in

A2 => r2, --in

A3 => r3, --in

A4 => r4, --in

A5 => r5, --in

A6 => r6, --in

A7 => r7, --in

RegSel => regSelB, --in

Q => muxOutB);

AddSubUnit\_0 : AddSubUnit

port map(

MuxA\_out => muxOutA, --in

MuxB\_out => muxOutB, --in

Add\_Sub\_sel => addSubSel, --in

Neg\_in => negSel, --in

Add\_Sub\_out => addSubOut,

overflow => Overflow, --connected to direct output

zero => Zero); --connected to direct output

Mux2\_4bit\_0 : Mux2\_4bit

port map(

immVal => immVal, --in

addSub\_result => addSubOut, --in

loadSel => loadSel, --in

mux\_out => regBankIn);

Ins\_Decoder\_0 : Ins\_Decoder

port map(

Instruction => ins, --in

JumpCheck => muxOutA, --in

Reg\_En => regEn,

Load\_Sel => loadSel,

Imm\_Val => immVal,

Reg\_Sel\_A => regSelA,

Reg\_Sel\_B => regSelB,

AddSub\_Sel => addSubSel,

Jump\_Flag => jmpFlag,

Jump\_Add => jmpAdd,

Neg\_Sel => negSel);

Program\_ROM\_0 : Program\_ROM

port map(

Mem\_Sel => memSel, --in

Instruction => ins);

ProgramCounter\_0 : ProgramCounter

port map(

Clk => slwClkOut, --in

Reset => Reset, --connected from direct input

JumpFlag => jmpFlag, --in

JumpAdd => jmpAdd, --in

MemSel => memSel);

LUT\_7seg\_0 : LUT\_7seg

port map(

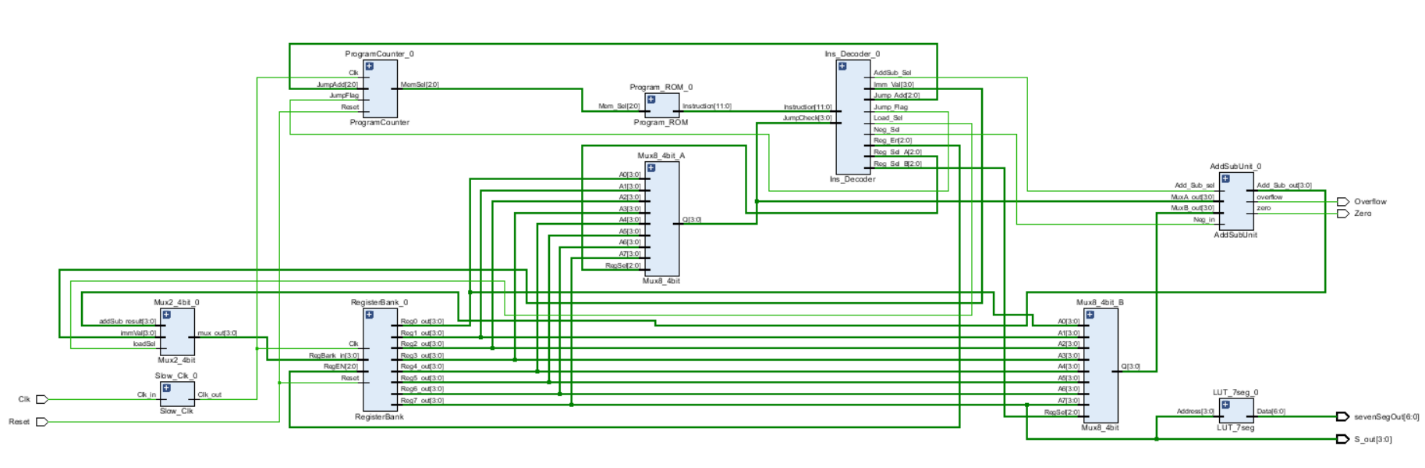
Address => r7,

Data => sevenSegOut);

S\_out <= r7; --connected to direct output

end Behavioral;

## RTL SCHEMATICS



## BEHAVIORAL SIMULATION CODE

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Sim\_NanoProcessor is

-- Port ( );

end Sim\_NanoProcessor;

architecture Behavioral of Sim\_NanoProcessor is

component NanoProcessor is

Port ( Clk : in STD\_LOGIC;

Reset : in STD\_LOGIC;

Overflow : out STD\_LOGIC;

Zero : out STD\_LOGIC;

S\_out : out STD\_LOGIC\_VECTOR (3 downto 0);

sevenSegOut : out STD\_LOGIC\_VECTOR(6 downto 0));

end component;

signal Clk, Reset, Overflow, Zero : std\_logic;

signal S\_out : std\_logic\_vector(3 downto 0);

signal sevenSegOut : std\_logic\_vector(6 downto 0);

constant clock\_period : time := 10ns;

begin

UUT : NanoProcessor port map(

Clk => Clk,

Reset => Reset,

Overflow => Overflow,

Zero => Zero,

S\_out => S\_out,

sevenSegOut => sevenSegOut);

clock\_process : process

begin

Clk <= '1';

wait for clock\_period / 2;

Clk <= '0';

wait for clock\_period / 2;

end process;

sim : process

begin

Reset <= '0';

wait for 500ns;

Reset <= '1';

wait for 100ns;

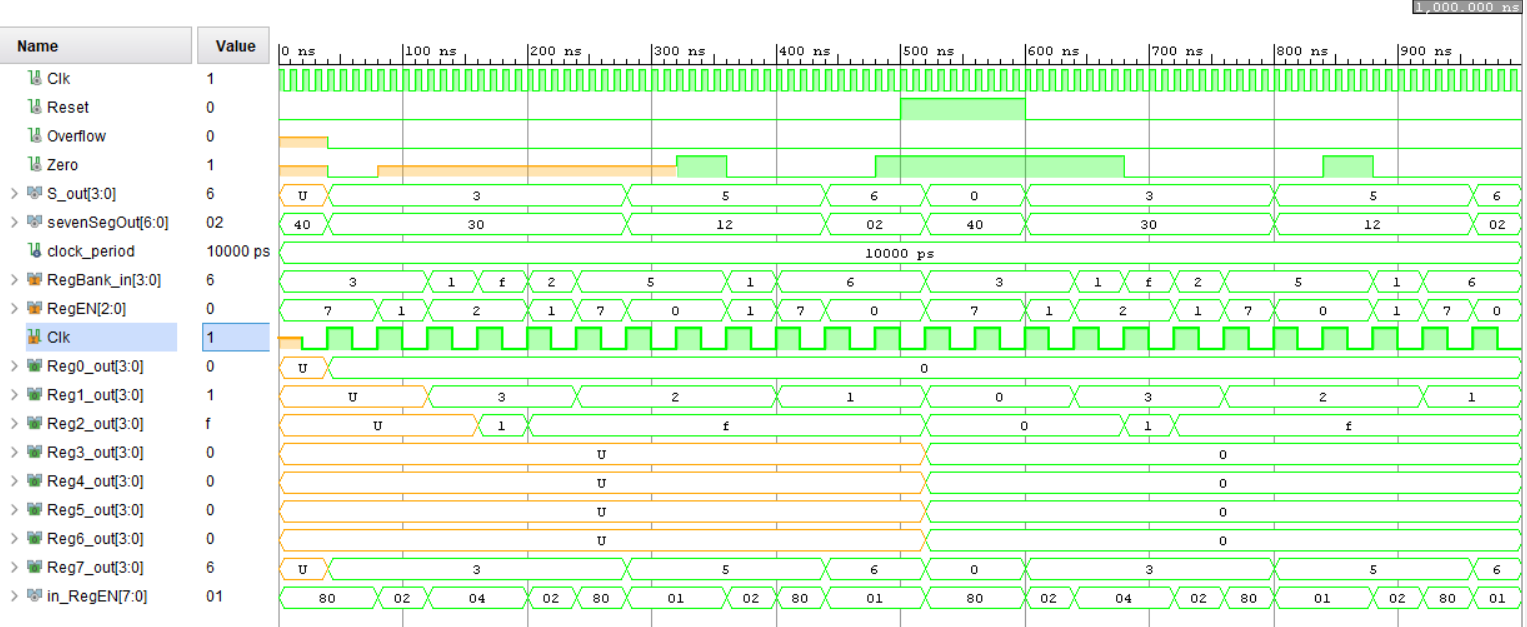
Reset <= '0';

wait;

end process;

end Behavioral;

## TIMING DIAGRAM



# XDC FILE

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

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

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

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

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

## LEDs - Register 7

set\_property PACKAGE\_PIN U16 [get\_ports {S\_out[0]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {S\_out[0]}]

set\_property PACKAGE\_PIN E19 [get\_ports {S\_out[1]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {S\_out[1]}]

set\_property PACKAGE\_PIN U19 [get\_ports {S\_out[2]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {S\_out[2]}]

set\_property PACKAGE\_PIN V19 [get\_ports {S\_out[3]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {S\_out[3]}]

##Overflow

set\_property PACKAGE\_PIN P1 [get\_ports {Overflow}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {Overflow}]

##Zero

set\_property PACKAGE\_PIN L1 [get\_ports {Zero}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {Zero}]

##Push Button for Reset

set\_property PACKAGE\_PIN U18 [get\_ports Reset]

set\_property IOSTANDARD LVCMOS33 [get\_ports Reset]

##7 segment display for Register 7

set\_property PACKAGE\_PIN W7 [get\_ports {sevenSegOut[0]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {sevenSegOut[0]}]

set\_property PACKAGE\_PIN W6 [get\_ports {sevenSegOut[1]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {sevenSegOut[1]}]

set\_property PACKAGE\_PIN U8 [get\_ports {sevenSegOut[2]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {sevenSegOut[2]}]

set\_property PACKAGE\_PIN V8 [get\_ports {sevenSegOut[3]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {sevenSegOut[3]}]

set\_property PACKAGE\_PIN U5 [get\_ports {sevenSegOut[4]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {sevenSegOut[4]}]

set\_property PACKAGE\_PIN V5 [get\_ports {sevenSegOut[5]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {sevenSegOut[5]}]

set\_property PACKAGE\_PIN U7 [get\_ports {sevenSegOut[6]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {sevenSegOut[6]}]

## Clock signal

set\_property PACKAGE\_PIN W5 [get\_ports Clk]

set\_property IOSTANDARD LVCMOS33 [get\_ports Clk]

create\_clock -add -name sys\_clk\_pin -period 10.00 -waveform {0 5} [get\_ports Clk]

# CONCLUSION

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

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

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

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