**LAB 9-10: NANOPROCESSOR DESIGN**

**LAB REPORT**

CS1050 Computer Organization and Digital Design

**Team 30**

H.M.M.M.Herath 230242A

J.M.A.N.N.Jayamanne 230272N

J.M.T.D.Chandrasiri 230101R

M.U.A.Ahamed 230025L

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**INTRODUCTION**

In this lab, we designed and implemented a 4-bit nano processor capable of executing 4 basic instructions, providing digital logic design, processor architecture, and hardware description language (VHDL) programming. The project involved constructing a simplified yet fully functional processor from the ground up, integrating key components such as arithmetic logic units (ALUs), registers, multiplexers, and control logic. The final design was simulated for correctness using XSim and deployed on a BASYS 3 FPGA board for real-world validation.

**Nano-processor Architecture**

The nano-processor consists of the following components:

1. **Register Bank**
   * Comprises eight 4-bit registers (R0–R7), with R0 hardwired to zero for use in operations like negation (NEG) and conditional jumps (JZR).
   * Implements tri-state buffers for controlled data flow, ensuring only one register drives the bus simultaneously.
2. **4-bit Add/Subtract Unit**
   * Built using a ripple-carry adder (RCA) with 2’s complement arithmetic for signed number operations.
   * Detects overflow and output flags for error handling.
3. **Program Counter (PC)** 
   * A 3-bit synchronous counter with reset functionality, allowing program restart via a pushbutton.
   * Supports conditional jumps (JZR) by checking if a register’s value is zero.
4. **Instruction Decoder**
   * Translates 12-bit machine code into control signals for:
     + Using register enable (RRR), only the required register should be enabled
     + For ADD and NEG instructions, relevant inputs are selected from the multiplexers, and the output is sent to the correct register.
     + Write values to registers
     + Program flow control (Address to jump or increment the Program Counter
5. **Program ROM**
   * Store the assembly programs.
   * Implemented as a lookup table (LUT) in VHDL, preloaded with instructions.
6. **Data Path & Multiplexers**
   * Uses 8-way 4-bit multiplexers to route data between registers and the ALU.
   * A 2-way 3-bit multiplexer controls the PC’s next address (increment or jump).

**Implementation & Verification**

1. **Simulation**
   * Verified correct register updates, ALU operations, and branching.
   * Checked overflow handling in signed arithmetic.
2. **FPGA Deployment (BASYS 3)**
   * Connected R7 output to LEDs and 7-segment display for real-time monitoring.
   * Used 7-segment display for debugging.
   * Implemented a slow clock (~1 Hz) for step-by-step execution visibility.

**Team Contributions**

| **Team Member** | **Contributions** |
| --- | --- |
| H.M.M.M.Herath |  |
| J.M.A.N.N.Jayamanne | Designed VHDL source codes and test bench codes for components including Register Bank, 7-segment Display, Program ROM. Implemented additional features for 7 segment display and Program ROM. Assembled the PDF report |
| J.M.T.D.Chandrasiri |  |
| M.U.A.Ahamed |  |

**Additional Features**

We have enabled two seven segment displays so we can show the negative sign. Thereby enabling us to display the values of Register 7 ranging from -8 to +7. We have also added an extra Program to the ROM with the R7 showing a negative value and also the program is running in an infinite loop.

We introduced two features to improve functionality and user interactivity. Firstly, we used two 7-segment displays to represent negative values stored in Register 7. Instead of displaying only positive values, our design utilizes two of the available 7-segment displays, where the first digit display the absolute value of the number in decimal, and the second digit conditionally shows a minus sign if the value is negative. This dual-display approach offers clear, intuitive feedback for interpreting signed numbers. Secondly, we implemented support for an additional program that runs independently when the change\_program signal is activated. This secondary program is specifically designed to display the sign handling capability, operating within an infinite loop. Together, these enhancements significantly improve the usability and debugging transparency of the Nanoprocessor system.

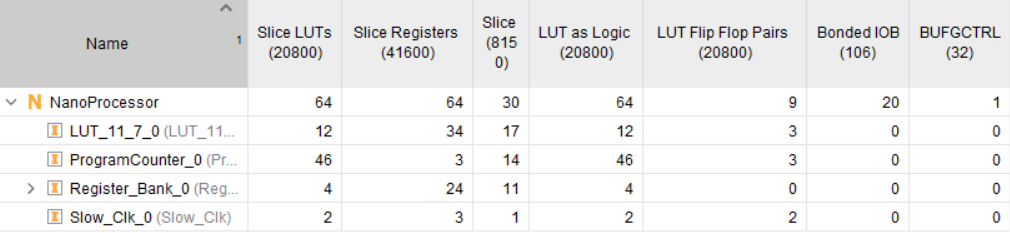
**Resource Utilization Reports**

**With Minimum Requirements**

A screenshot of a table

AI-generated content may be incorrect.

**With Additional Features**



**Assembly Program**

MOVI R7,3 => R7=3 (Move the value 3 to register 7)

MOVI R1,1 => R1=1(Move the value 1 to register 1)

NEG R1 => R1=-1 (Make register 1’s value negative)

MOVI R2,2 => R2=2 (Move the value 2 to register 2)

ADD R7,R2 => R7=5 (Add values in Register 7 and 2 and save the answer in register 7)

ADD R2,R1 => R2=1 (Add values in Register 2 and 1 and save the in register 7)

ADD R7,R2 => R7=6 (Add values in Register 7 and 2 and save the answer in register 7)

JZR R0,7 => (Goes to the 7th instruction if R0==0 which is not, so program will stay on the 7th instruction)

**Machine Code**

"101110000011"

"100010000001"

"010010000000"

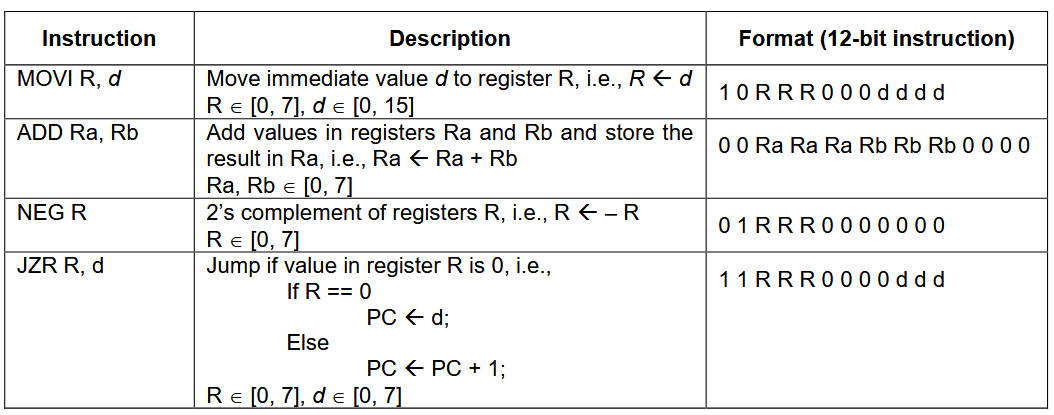
"100100000010"

"001110100000"

"000100010000"

"001110100000"

"110000000111"

In the machine code representation, each line represents a single instruction encoded in binary. The first two bits usually denote the opcode, followed by the necessary register address, operand or immediate value if applicable. These machine code instructions are what the processor executes directly.

**Strategies Used to Optimize resource consumption**

**Overall design**

In this design, we optimized resource utilization by using a 2 bit unsigned signal instead of a larger integer which use a 32 bit counter. This significantly reduces the number of flip-flops required for implementation in hardware, minimizing area and power consumption. By adjusting the bit width to the minimum needed for the desired frequency division (in this case, toggling the output after counting to 3), we achieve an efficient design that conserves FPGA resources while maintaining the intended clock slowing functionality.

**3-bit Adder**

In this 3 bit adder design, we optimized the circuit by avoiding a traditional ripple carry adder and instead implemented a custom incrementor. Since our requirement was only to add 1 to the 3 bit input R, we used simple combinational logic to generate the output Y directly. This approach uses just 2 XOR gates, 1 AND gate, and 1 NOT gate, compared to a ripple carry adder which would typically require 3 full adders, each consisting of multiple XOR, AND, and OR gates. Overall, this significantly reduces the total number of logic gates used, leading to a smaller, faster, and more power efficient design tailored specifically for incrementing by 1.

**Design sources**

**NanoProcessor.VHDL**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use work.Reg\_types.all;

entity NanoProcessor is

Port ( Clk : in STD\_LOGIC;

change\_program : in STD\_LOGIC;

Reset : in STD\_LOGIC;

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

S\_7seg : out STD\_LOGIC\_VECTOR (6 downto 0);

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

Overflow : out STD\_LOGIC;

Zero : out STD\_LOGIC);

end NanoProcessor;

architecture Behavioral of NanoProcessor is

component Slow\_Clk is

Port ( Clk\_in : in STD\_LOGIC;

Clk\_Out : out STD\_LOGIC);

end component;

component ProgramCounter is

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

Res : in STD\_LOGIC;

Clk : in STD\_LOGIC;

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

end component;

component Program\_ROM is

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

change\_program : in STD\_LOGIC;

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

end component;

component Instruction\_Decoder is

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

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

Load\_select : out STD\_LOGIC;

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

Register\_select\_A : out STD\_LOGIC\_VECTOR (2 downto 0);

Register\_select\_B : out STD\_LOGIC\_VECTOR (2 downto 0);

Neg : out STD\_LOGIC;

Reg\_check\_jmp : in STD\_LOGIC\_VECTOR (3 downto 0);

jmp\_flag : out STD\_LOGIC;

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

end component;

component Register\_Bank is

Port ( Clk : in STD\_LOGIC;

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

Reset : in STD\_LOGIC;

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

Data\_Bus : out Reg\_array);

end component;

component Mux\_8\_to\_4 is

Port ( R0 : in STD\_LOGIC\_VECTOR (3 downto 0);

R1 : in STD\_LOGIC\_VECTOR (3 downto 0);

R2 : in STD\_LOGIC\_VECTOR (3 downto 0);

R3 : in STD\_LOGIC\_VECTOR (3 downto 0);

R4 : in STD\_LOGIC\_VECTOR (3 downto 0);

R5 : in STD\_LOGIC\_VECTOR (3 downto 0);

R6 : in STD\_LOGIC\_VECTOR (3 downto 0);

R7 : in STD\_LOGIC\_VECTOR (3 downto 0);

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

M8\_4\_out : out STD\_LOGIC\_VECTOR (3 downto 0));

end component;

component bit4\_Add\_Sub\_Unit is

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

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

Neg : in STD\_LOGIC;

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

Overflow : out STD\_LOGIC;

Zero : out STD\_LOGIC);

end component;

component Mux\_2\_to\_4 is

Port ( B0 : in STD\_LOGIC\_VECTOR (3 downto 0);

B1 : in STD\_LOGIC\_VECTOR (3 downto 0);

LoadSel : in STD\_LOGIC;

M2\_4\_out : out STD\_LOGIC\_VECTOR (3 downto 0));

end component;

component Mux\_2\_to\_3 is

Port ( A0 : in STD\_LOGIC\_VECTOR (2 downto 0);

A1 : in STD\_LOGIC\_VECTOR (2 downto 0);

JMP\_FLAG : in STD\_LOGIC;

M2\_3\_out : out STD\_LOGIC\_VECTOR (2 downto 0));

end component;

component bit3\_Adder is

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

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

end component;

component LUT\_11\_7 is

Port ( Clk : in STD\_LOGIC;

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

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

S\_7seg : out STD\_LOGIC\_VECTOR (6 downto 0));

end component;

signal pc\_out, pc\_in : STD\_LOGIC\_VECTOR (2 downto 0);

signal slw\_clk : std\_logic;

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

signal Register\_enable : STD\_LOGIC\_VECTOR (2 downto 0);

signal Load\_select : std\_logic;

signal Immediate\_value : STD\_LOGIC\_VECTOR (3 downto 0);

signal Register\_select\_A, Register\_select\_B : STD\_LOGIC\_VECTOR (2 downto 0);

signal Neg : std\_logic;

signal jmp\_flag : std\_logic;

signal jmp\_addr : STD\_LOGIC\_VECTOR (2 downto 0);

signal Data : STD\_LOGIC\_VECTOR (3 downto 0);

signal Data\_Bus : Reg\_array;

signal M8\_4\_out\_A, M8\_4\_out\_B : STD\_LOGIC\_VECTOR (3 downto 0);

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

signal next\_addr : STD\_LOGIC\_VECTOR (2 downto 0);

begin

Slow\_Clk\_0 : Slow\_Clk

Port map ( Clk\_in => Clk,

Clk\_Out => slw\_clk);

ProgramCounter\_0 : ProgramCounter

Port map ( D => pc\_in,

Res => Reset,

Clk => slw\_clk,

Q => pc\_out);

Program\_ROM\_0 : Program\_ROM

Port map ( address => pc\_out,

change\_program => change\_program,

data => Instruction\_Bus);

Instruction\_Decoder\_0 : Instruction\_Decoder

Port map ( Instruction\_Bus => Instruction\_Bus,

Register\_enable => Register\_enable,

Load\_select => Load\_select,

Immediate\_value => Immediate\_value,

Register\_select\_A => Register\_select\_A,

Register\_select\_B => Register\_select\_B,

Neg => Neg,

Reg\_check\_jmp => M8\_4\_out\_A,

jmp\_flag => jmp\_flag,

jmp\_addr => jmp\_addr);

Register\_Bank\_0 : Register\_Bank

Port map ( Clk => slw\_clk,

Reg\_EN => Register\_enable,

Reset => Reset,

input => Data,

Data\_Bus => Data\_Bus);

Mux\_8\_to\_4\_A : Mux\_8\_to\_4

Port map ( R0 => Data\_Bus(0),

R1 => Data\_Bus(1),

R2 => Data\_Bus(2),

R3 => Data\_Bus(3),

R4 => Data\_Bus(4),

R5 => Data\_Bus(5),

R6 => Data\_Bus(6),

R7 => Data\_Bus(7),

Sel => Register\_select\_A,

M8\_4\_out => M8\_4\_out\_A);

Mux\_8\_to\_4\_B : Mux\_8\_to\_4

Port map ( R0 => Data\_Bus(0),

R1 => Data\_Bus(1),

R2 => Data\_Bus(2),

R3 => Data\_Bus(3),

R4 => Data\_Bus(4),

R5 => Data\_Bus(5),

R6 => Data\_Bus(6),

R7 => Data\_Bus(7),

Sel => Register\_select\_B,

M8\_4\_out => M8\_4\_out\_B);

bit4\_Add\_Sub\_Unit\_0 : bit4\_Add\_Sub\_Unit

Port map ( A => M8\_4\_out\_A,

B => M8\_4\_out\_B,

Neg => Neg,

S => S,

Overflow => Overflow,

Zero => Zero);

Mux\_2\_to\_4\_0 : Mux\_2\_to\_4

Port map ( B0 => S,

B1 => Immediate\_value,

LoadSel => Load\_select,

M2\_4\_out => Data);

Mux\_2\_to\_3\_0 : Mux\_2\_to\_3

Port map ( A0 => next\_addr,

A1 => jmp\_addr,

JMP\_FLAG => jmp\_flag,

M2\_3\_out => pc\_in);

bit3\_Adder\_0 : bit3\_Adder

Port map ( R => pc\_out,

Y => next\_addr);

LUT\_11\_7\_0 : LUT\_11\_7

Port map ( Clk => Clk,

number => Data\_Bus(7),

anode => anode,

S\_7seg => S\_7seg);

LED <= Data\_Bus(7);

end Behavioral;

**Slow\_Clk.VHDL**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.NUMERIC\_STD.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 : unsigned(25 downto 0) := "00000000000000000000000000";

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

begin

Clk\_out <= clk\_status;

process (clk\_in) begin

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

count <= count + 1;

if(count = "10111110101111000010000000") then

clk\_status <= not clk\_status;

count <= "00000000000000000000000000";

end if;

end if;

end process;

end Behavioral;

**ProgramCounter.VHDL**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity ProgramCounter is

Port ( D : in STD\_LOGIC\_VECTOR (2 downto 0):= "000";

Res : in STD\_LOGIC;

Clk : in STD\_LOGIC;

Q : out STD\_LOGIC\_VECTOR (2 downto 0):= "000");

end ProgramCounter;

architecture Behavioral of ProgramCounter is

begin

-- Here we have modified the code used in Lab 5. Instead of using DFFs, a bus is used.

process(Clk) begin

if(rising\_edge(Clk)) then

if(Res ='1') then

Q <= "000";

else

Q <= D;

end if;

end if;

end process;

end Behavioral;

**Program\_ROM.VHDL**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use ieee.numeric\_std.all;

entity Program\_ROM is

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

change\_program : in STD\_LOGIC:='0';

data : 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 First\_Program : rom\_type := (

"101110000011", --MOVI R7,3 => R7=3

"100010000001", --MOVI R1,1 => R1=1

"010010000000", --NEG R1 => R1=-1

"100100000010", --MOVI R2,2 => R2=2

"001110100000", --ADD R7,R2 => R7=5

"000100010000", --ADD R2,R1 => R2=1

"001110100000", --ADD R7,R2 => R7=6

"110000000111" --JZR R0,7 => GOES TO 7th INSTRUCTION IF R0==0 WHICH IS NOT

);

signal Second\_Program : rom\_type:=(

"101110000001", --MOVI R7,1 => R7=1

"101010000010", --MOVI R5,2 => R5=2

"011110000000", --NEG R7 => R7=-1

"101100000011", --MOVI R6,3 => R6=3

"001111010000", --ADD R7,R5 => R7=1

"001111100000", --ADD R7,R6 => R7=4

"011110000000", --NEG R7 => R7=-4

"110000000000" --JZR R0,0 => GOES TO 0th instruction if R0==0 which is

);

begin

process(change\_program,address)

begin

if change\_program ='1' then

data <= Second\_Program(to\_integer(unsigned(address)));

else

data <= First\_Program(to\_integer(unsigned(address)));

end if;

end process;

end Behavioral;

**Instruction\_Decoder.VHDL**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Instruction\_Decoder is

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

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

Load\_select : out STD\_LOGIC;

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

Register\_select\_A : out STD\_LOGIC\_VECTOR (2 downto 0);

Register\_select\_B : out STD\_LOGIC\_VECTOR (2 downto 0);

Neg : out STD\_LOGIC;

Reg\_check\_jmp : in STD\_LOGIC\_VECTOR (3 downto 0);

jmp\_flag : out STD\_LOGIC;

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

end Instruction\_Decoder;

architecture Behavioral of Instruction\_Decoder is

begin

Register\_enable <= Instruction\_Bus(9 downto 7);

Load\_select <= Instruction\_Bus(11);

Immediate\_value <= Instruction\_Bus(3 downto 0);

Register\_select\_A <= Instruction\_Bus(9 downto 7);

Register\_select\_B <= Instruction\_Bus(6 downto 4);

Neg <= Instruction\_Bus(10);

jmp\_flag <= '1' when Reg\_check\_jmp = "0000" and Instruction\_Bus(11 downto 10) = "11" else '0';

jmp\_addr <= Instruction\_Bus(2 downto 0);

end Behavioral;

**Register\_Bank.VHDL**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use work.Reg\_types.all;

entity Register\_Bank is

Port ( Clk : in STD\_LOGIC;

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

Reset : in STD\_LOGIC;

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

Data\_Bus : out Reg\_array);

end Register\_Bank;

architecture Behavioral of Register\_Bank is

component Decoder\_3\_to\_8

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

EN : in STD\_LOGIC := '1';

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

end component;

component Reg

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

EN : in STD\_LOGIC;

Clk : in STD\_LOGIC;

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

end component;

signal Reg\_EN\_out : STD\_LOGIC\_VECTOR (7 downto 0);

signal Input\_line : STD\_LOGIC\_VECTOR (3 downto 0);

signal Data\_Buses : Reg\_array;

begin

Input\_line <= "0000" when ( Reset='1' ) else input;

Decoder : Decoder\_3\_to\_8

port map(

I=>Reg\_EN,

Y=>Reg\_EN\_out);

Reg\_0 : Reg

port map(

D=>"0000",

EN=>'1',

Clk =>Clk,

Q=>Data\_Buses(0));

gen\_regs: for i in 1 to 7 generate

reg\_inst: Reg

port map (

D => Input\_line,

EN => Reg\_EN\_out(i),

Clk => Clk,

Q=>Data\_Buses(i));

end generate;

Data\_Bus <= Data\_Buses;

end Behavioral;

**Reg.VHDL**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Reg is

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

EN : 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

if EN = '1' then

Q<=D;

end if;

end if;

end process;

end Behavioral;

**Mux\_8\_to\_4.VHDL**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Mux\_8\_to\_4 is

Port ( R0 : in STD\_LOGIC\_VECTOR (3 downto 0);

R1 : in STD\_LOGIC\_VECTOR (3 downto 0);

R2 : in STD\_LOGIC\_VECTOR (3 downto 0);

R3 : in STD\_LOGIC\_VECTOR (3 downto 0);

R4 : in STD\_LOGIC\_VECTOR (3 downto 0);

R5 : in STD\_LOGIC\_VECTOR (3 downto 0);

R6 : in STD\_LOGIC\_VECTOR (3 downto 0);

R7 : in STD\_LOGIC\_VECTOR (3 downto 0);

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

M8\_4\_out : out STD\_LOGIC\_VECTOR (3 downto 0));

end Mux\_8\_to\_4;

architecture Behavioral of Mux\_8\_to\_4 is

begin

M8\_4\_out <= R0 when Sel = "000" else

R1 when Sel = "001" else

R2 when Sel = "010" else

R3 when Sel = "011" else

R4 when Sel = "100" else

R5 when Sel = "101" else

R6 when Sel = "110" else

R7 when Sel = "111" else

(others => 'U');

end Behavioral;

**MUX\_2\_to\_4.VHDL**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Mux\_2\_to\_4 is

Port ( B0 : in STD\_LOGIC\_VECTOR (3 downto 0);

B1 : in STD\_LOGIC\_VECTOR (3 downto 0);

LoadSel : in STD\_LOGIC;

M2\_4\_out : out STD\_LOGIC\_VECTOR (3 downto 0));

end Mux\_2\_to\_4;

architecture Behavioral of Mux\_2\_to\_4 is

begin

M2\_4\_out <= B0 when LoadSel = '0' else B1;

end Behavioral;

**MUX\_2\_to\_3.VHDL**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Mux\_2\_to\_3 is

Port ( A0 : in STD\_LOGIC\_VECTOR (2 downto 0);

A1 : in STD\_LOGIC\_VECTOR (2 downto 0);

JMP\_FLAG : in STD\_LOGIC;

M2\_3\_out : out STD\_LOGIC\_VECTOR (2 downto 0));

end Mux\_2\_to\_3;

architecture Behavioral of Mux\_2\_to\_3 is

begin

M2\_3\_out <= A0 when JMP\_FLAG = '0' else A1;

end Behavioral;

**bit4\_Add\_Sub\_Unit.VHDL**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity bit4\_Add\_Sub\_Unit is

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

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

Neg : in STD\_LOGIC;

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

Overflow : out STD\_LOGIC;

Zero : out STD\_LOGIC);

end bit4\_Add\_Sub\_Unit;

architecture Behavioral of bit4\_Add\_Sub\_Unit is

component 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 component;

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

signal A0\_xor\_Neg, A1\_xor\_Neg, A2\_xor\_Neg, A3\_xor\_Neg : std\_logic;

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

begin

A0\_xor\_Neg <= A(0) xor Neg;

A1\_xor\_Neg <= A(1) xor Neg;

A2\_xor\_Neg <= A(2) xor Neg;

A3\_xor\_Neg <= A(3) xor Neg;

FA\_0 : FA

port map (

A => A0\_xor\_Neg,

B => B(0),

C\_in => Neg,

S => Sum(0),

C\_out => FA0\_C);

FA\_1 : FA

port map (

A => A1\_xor\_Neg,

B => B(1),

C\_in => FA0\_C,

S => Sum(1),

C\_out => FA1\_C);

FA\_2 : FA

port map (

A => A2\_xor\_Neg,

B => B(2),

C\_in => FA1\_C,

S => Sum(2),

C\_out => FA2\_C);

FA\_3 : FA

port map (

A => A3\_xor\_Neg,

B => B(3),

C\_in => FA2\_C,

S => Sum(3),

C\_out => FA3\_C);

Overflow <= FA2\_C xor FA3\_C;

Zero <= '1' when Sum = "0000" else '0';

S <= Sum;

end Behavioral;

**FA.VHDL**

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);

S <= HA1\_S;

C\_out <= HA0\_C or HA1\_C;

end Behavioral;

**HA.VHDL**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity HA is

Port ( A : in STD\_LOGIC;

B : in STD\_LOGIC;

S : out STD\_LOGIC;

C : out STD\_LOGIC);

end HA;

architecture Behavioral of HA is

begin

S <= A xor B;

C <= A and B;

end Behavioral;

**bit3\_Adder.VHDL**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity bit3\_Adder is

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

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

end bit3\_Adder;

architecture Behavioral of bit3\_Adder is

begin

Y(0) <= not R(0);

Y(1) <= R(1) xor R(0);

Y(2) <= R(2) xor (R(1) and R(0));

end Behavioral;

**LUT\_11\_7.VHDL (7segment display source file)**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.NUMERIC\_STD.ALL;

entity LUT\_11\_7 is

Port ( Clk : in STD\_LOGIC;

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

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

S\_7seg : out STD\_LOGIC\_VECTOR (6 downto 0));

end LUT\_11\_7;

architecture Behavioral of LUT\_11\_7 is

type rom\_type is array (0 to 10) 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

"0111111", -- minus sign '-'

"1111111" -- shows nothing

);

signal count : integer := 1;

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

signal Clk\_out : std\_logic:= '1';

signal digit0 : integer range 0 to 15 := 0;

signal is\_negative : boolean := false;

begin

-- Clock divider and digit index rotator (roughly 1kHz)

process(clk)

begin

if rising\_edge(clk) then

count <= count +1;

if(count = 500000) then

clk\_status<= not clk\_status;

Clk\_out <= clk\_status;

count <=1;

end if;

end if;

end process;

-- Decode number into digits and sign

process(number)

variable num\_signed : integer;

begin

num\_signed := to\_integer(signed(number));

if num\_signed < 0 then

is\_negative <= true;

num\_signed := -num\_signed;

else

is\_negative <= false;

end if;

digit0 <= num\_signed;

end process;

-- Multiplexing logic

process(Clk\_out, digit0, is\_negative)

begin

case Clk\_out is

when '0' => -- Rightmost digit (LSB)

anode <= "1110";

S\_7seg <= sevenSegment\_ROM(digit0);

when '1' => -- Next digit (MSB or minus sign)

anode <= "1101";

if is\_negative then

S\_7seg <= sevenSegment\_ROM(9); -- minus sign

else

S\_7seg <= sevenSegment\_ROM(10); --shows nothing

end if;

when others => --turn off unused

anode <= "1111";

S\_7seg <= "1111111";

end case;

end process;

end Behavioral;

**Timing Diagrams**

**TB\_Nano\_Processor.VHDL**

**A screenshot of a computer screen

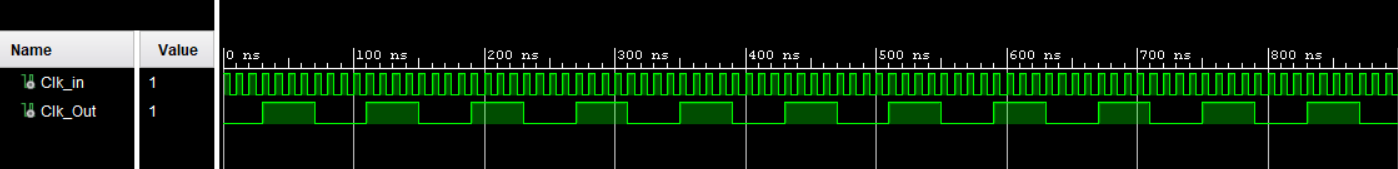
AI-generated content may be incorrect.**

**TB\_Instruction\_Decoder.VHDL**

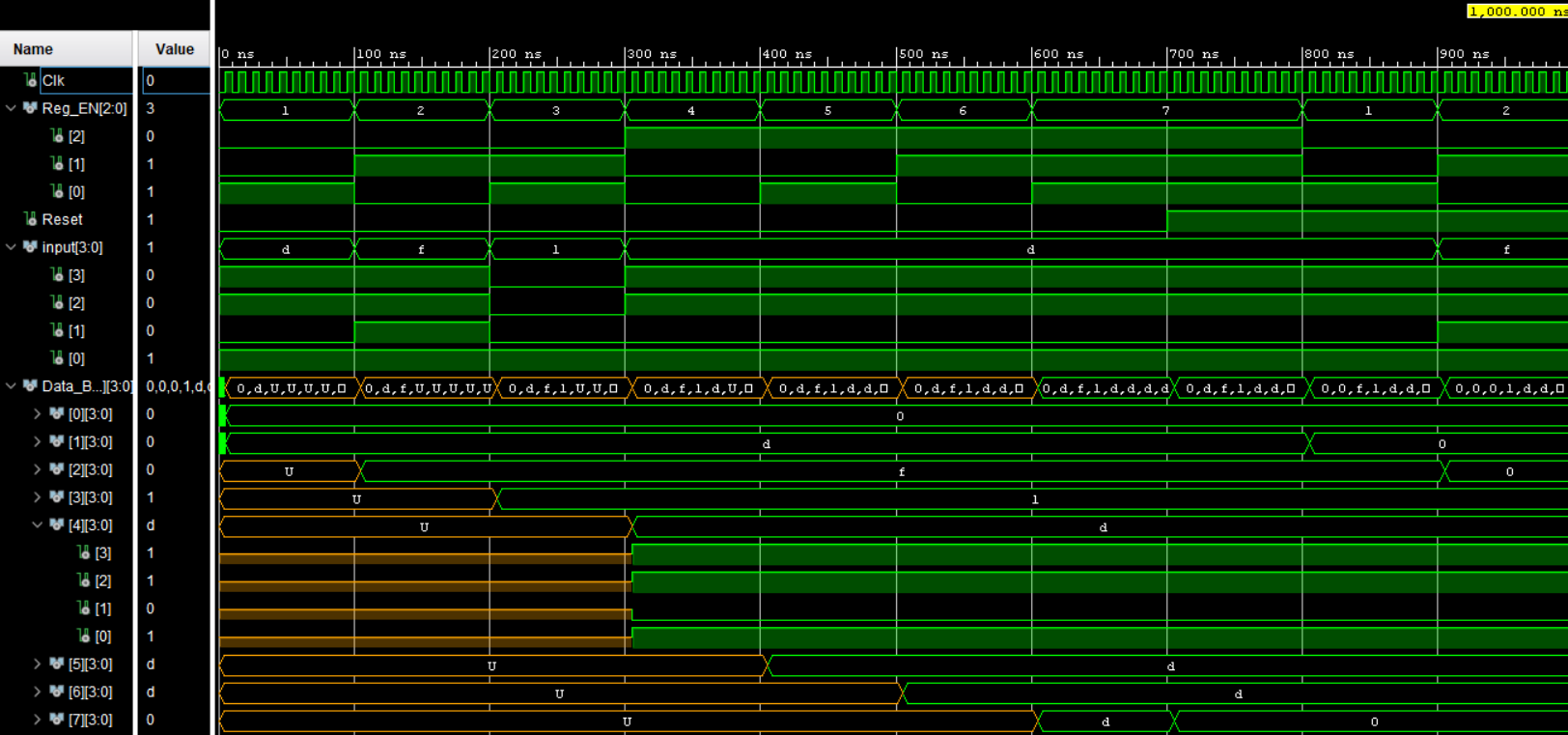
**A screenshot of a computer

AI-generated content may be incorrect.**

**TB\_Slow\_Clock.VHDL**

****

**Register Bank**

****

**TB\_bit3\_Adder.VHDL**

**A screenshot of a computer

AI-generated content may be incorrect.**

**TB\_bit4\_Add\_Sub\_Unit.VHDL**

**A screenshot of a computer

AI-generated content may be incorrect.**

**NanoProcessor Elaborated Design**

A diagram of a computer circuit

AI-generated content may be incorrect.**Implemented Designs**

**NanoProcessor**

A computer diagram of a computer

AI-generated content may be incorrect.

**Slow Clock**

**A diagram of a computer network

AI-generated content may be incorrect.**

**Program Counter**

**A diagram of a computer

AI-generated content may be incorrect.**

**Program ROM Elaborated Design**

**A diagram of a computer circuit

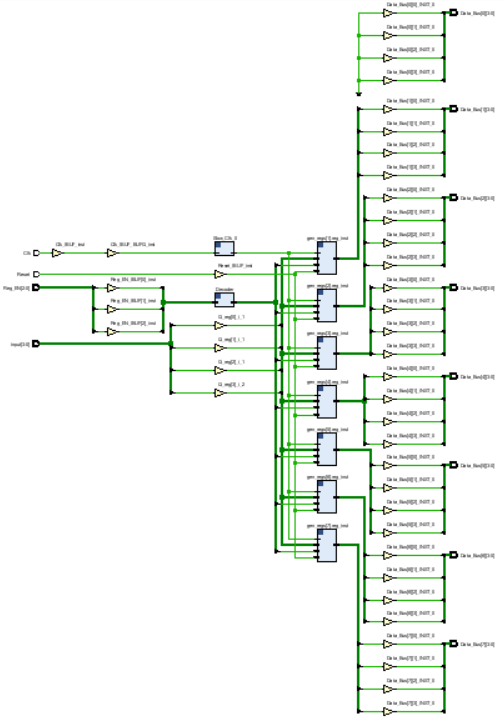
AI-generated content may be incorrect.**

**Instruction Decoder**

**A diagram of a computer

AI-generated content may be incorrect.**

**Register Bank**

****

**Register**

**A diagram of a computer

AI-generated content may be incorrect.**

**2-way 4-bit Multiplexer**

**A diagram of a computer

AI-generated content may be incorrect.**

**3 to 8 Decoder**

**A diagram of a decode system

AI-generated content may be incorrect.**

**8-way 4-bit Multiplexer**

**A diagram of a computer circuit

AI-generated content may be incorrect.**

**Half Adder**

**A diagram of a circuit

AI-generated content may be incorrect.**

**Full Adder**

**A diagram of a computer circuit

AI-generated content may be incorrect.**

**4-bit Add/Sub Unit**

**A diagram of a computer

AI-generated content may be incorrect.**

**3-bit Adder**

**A diagram of a computer

AI-generated content may be incorrect.**

**2-way 3-bit Multiplexer**

**A diagram of a computer

AI-generated content may be incorrect.**

**7 Segment Display**

**A green lines and squares

AI-generated content may be incorrect.**

**Constraints File (Basys3.xdc)**

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]

## Switches

set\_property PACKAGE\_PIN V17 [get\_ports {change\_program}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {change\_program}]

## LEDs

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

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

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

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

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

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

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

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

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

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

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

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

##7 segment display

set\_property PACKAGE\_PIN W7 [get\_ports {S\_7seg[0]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {S\_7seg[0]}]

set\_property PACKAGE\_PIN W6 [get\_ports {S\_7seg[1]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {S\_7seg[1]}]

set\_property PACKAGE\_PIN U8 [get\_ports {S\_7seg[2]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {S\_7seg[2]}]

set\_property PACKAGE\_PIN V8 [get\_ports {S\_7seg[3]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {S\_7seg[3]}]

set\_property PACKAGE\_PIN U5 [get\_ports {S\_7seg[4]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {S\_7seg[4]}]

set\_property PACKAGE\_PIN V5 [get\_ports {S\_7seg[5]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {S\_7seg[5]}]

set\_property PACKAGE\_PIN U7 [get\_ports {S\_7seg[6]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {S\_7seg[6]}]

set\_property PACKAGE\_PIN U2 [get\_ports {anode[0]}]

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

set\_property PACKAGE\_PIN U4 [get\_ports {anode[1]}]

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

set\_property PACKAGE\_PIN V4 [get\_ports {anode[2]}]

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

set\_property PACKAGE\_PIN W4 [get\_ports {anode[3]}]

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

##Buttons

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

set\_property IOSTANDARD LVCMOS33 [get\_ports Reset]

**Conclusion**

The project successfully achieved its goal of designing a 4-bit processor capable of executing 4 simple instructions. By developing key subcomponents like the instruction decoder and program counter, we mastered our skills in VHDL implementations. Our approach to the project involved modular design, careful component integration and thorough testing to ensure the reliability and performance of the Nanoprocessor system. The emphasis on teamwork facilitated efficient collaboration and integration of individual contributions of each team member, which resulted in an efficiently functioning nano processor with verified functionality on simulation and development board. Additionally, integrating a user-friendly display for negative values indicates our attention to usability and user experience.