Nano Processor

Table Of Content

1. Introduction

1.1 Team Details

Name	Index Number
Himath Nimpura	230139N
Kalana LIyanage	230461T
Tharupahan Jayawardana	230295L
Nisal Wilochana	230230K

1.2 Project Overview

The objective of this lab was to design and implement a 4-bit nanoprocessor capable of executing a limited instruction set, including arithmetic and conditional jump operations. The processor was built using VHDL, simulated, and tested on the BASYS 3 FPGA board.

Key tasks:

- Build an arithmetic unit for addition and subtraction.
- Implement a 3-bit program counter.
- Design multi-bit multiplexers.
- Develop instruction decoding logic.
- Create and run an assembly program on the processor.

To further extend the functionality of the nanoprocessor beyond basic arithmetic and control, several logical and comparison operations were integrated into the design:

- Implemented bitwise operations: AND, OR, and XOR.
- Added comparison operations: equal to (==), greater than (>), and less than (<).

2. Instruction Set Architecture

2.1 Original Instruction Set

Instruction	Description	Format (12-bit instruction)
MOVI R, *d*	Move immediate value *d* to register R, i.e., *R* <- *d* R ∈ [0, 7], d ∈ [0, 15]	1 0 R R R 0 0 0 d d d d
ADD Ra, Rb	Add values in registers Ra and Rb and store the result in Ra, i.e., Ra <- Ra + Rb Ra, Rb ∈ [0, 7]	0 0 Ra Ra Ra Rb Rb Rb 0 0 0 0
NEG R	2’s complement of registers R, i.e., R <- -R R ∈ [0, 7]	0 1 R R R 0 0 0 0 0 0 0
JZR R, d	Jump if value in register R is 0, i.e., If R == 0 PC <- d; Else PC <- PC + 1; R ∈ [0, 7], d ∈ [0, 7]	1 1 R R R 0 0 0 0 d d d

2.2 Extended Instruction Set

Instruction	Description	Format (12-bit instruction)
MOVI R, *d*	Move immediate value *d* to register R, i.e., *R* <- *d* R ∈ [0, 7], d ∈ [0, 15]	1 0 R R R 0 0 0 d d d d
ADD Ra, Rb	Add values in registers Ra and Rb and store the result in Ra, i.e., Ra <- Ra + Rb Ra, Rb ∈ [0, 7]	0 0 Ra Ra Ra Rb Rb Rb 0 0 0 0
SUB Ra, Rb	Subtract Rb from Ra and store result in Ra, i.e., Ra ← Ra - Rb Ra, Rb ∈ [0, 7]	0 0 Ra Ra Ra Rb Rb Rb 0 0 0 1
NEG R	2’s complement of registers R, i.e., R <- -R R ∈ [0, 7]	0 1 R R R 0 0 0 0 0 0 0
AND Ra, Rb	Bitwise AND of Ra and Rb, store result in Ra, i.e., Ra ← Ra & Rb Ra, Rb ∈ [0, 7]	0 0 Ra Ra Ra Rb Rb Rb 0 0 1 0
OR Ra, Rb	Bitwise OR of Ra and Rb, store result in Ra, i.e., Ra ← Ra | Rb	0 0 Ra Ra Ra Rb Rb Rb 0 0 1 1
XOR Ra, Rb	Bitwise XOR of Ra and Rb, store result in Ra, i.e., Ra ← Ra ^ Rb Ra, Rb ∈ [0, 7]	0 0 Ra Ra Ra Rb Rb Rb 0 1 0 0
MUL Ra, Rb	Multiply values in registers Ra and Rb and store the result in Ra, i.e., Ra <- Ra * Rb Ra, Rb ∈ [0, 7]	0 0 Ra Ra Ra Rb Rb Rb 0 1 0 1
CMP Ra, Rb	Compare Ra and Rb, set flags accordingly Ra, Rb ∈ [0, 7]	0 0 Ra Ra Ra Rb Rb Rb 0 1 1 1
JZR R, d	Jump if value in register R is 0, i.e., If R == 0 PC <- d; Else PC <- PC + 1; R ∈ [0, 7], d ∈ [0, 7]	1 1 R R R 0 0 0 0 d d d

2.3 Instruction Encoding

This table defines the primary operation categories using 2-bit opcodes:

Instruction Type	OP Code (2 bits)
ALU Operations	00
NEG	01
MOVI	10
JZR	11

This table defines the specific ALU operations using 3-bit function codes and 4-bit sub-opcodes:

ALU Operation	**Operation Code**	**Sub-opcode**
ADD	000	0000
SUB	001	0001
NEG	010	-
AND	011	0010
OR	100	0011
XOR	101	0100
MUL	110	0101
CMP	111	0111

2.4 Extended Instruction Set Program Examples

The program in section 3.2 demonstrates all extended operations using registers R6 and R7. Below is a detailed explanation of each instruction:

```
"101110000101", -- 0: MOVI R7, 5  ; R7 ← 5 (0101)
```

Explanation: Loads the immediate value 5 into register R7. This initializes R7 with a positive value for subsequent operations.

```
"101100000011", -- 1: MOVI R6, 3  ; R6 ← 3 (0011)
```

Explanation: Loads the immediate value 3 into register R6. This initializes R6 with another positive value.

```
"001111100001", -- 2: SUB R7, R6  ; R7 ← R7 - R6 = 2 (0010)
```

Explanation: Subtracts the value in R6 (3) from R7 (5), storing the result (2) back in R7. This tests the SUB operation.

```
"001111100101", -- 3: MUL R7, R6  ; R7 ← R7 * R6 = 6 (0110)
```

Explanation: Multiplies the value in R7 (2) by the value in R6 (3), storing the result (6) back in R7. This tests the MUL operation.

```
"001101100010", -- 4: AND R6, R7  ; R6 ← R6 AND R7 = 2 (0010)
```

Explanation: Performs a bitwise AND between R6 (3 = 0011) and R7 (6 = 0110), resulting in 0010 (2) which is stored in R6. This tests the AND operation.

```
"001110110011", -- 5: OR R7, R6   ; R7 ← R7 OR R6 = 6 (0110)
```

Explanation: Performs a bitwise OR between R7 (6 = 0110) and R6 (2 = 0010), resulting in 0110 (6) which is stored in R7. This tests the OR operation.

```
"001101100100", -- 6: XOR R6, R7  ; R6 ← R6 XOR R7 = 4 (0100)
```

Explanation: Performs a bitwise XOR between R6 (2 = 0010) and R7 (6 = 0110), resulting in 0100 (4) which is stored in R6. This tests the XOR operation.

```
"001110110111"  -- 7: CMP R7, R6  ; Flags: EQ=0, LT=0, GT=1
```

Explanation: Compare the values in R7 (6) and R6 (4), setting the comparison flags accordingly. Since R7 > R6, the GreaterThan flag is set to 1 while Equal and LessThan flags remain 0. This tests the CMP operation.

3. Assembly Program & Machine Code

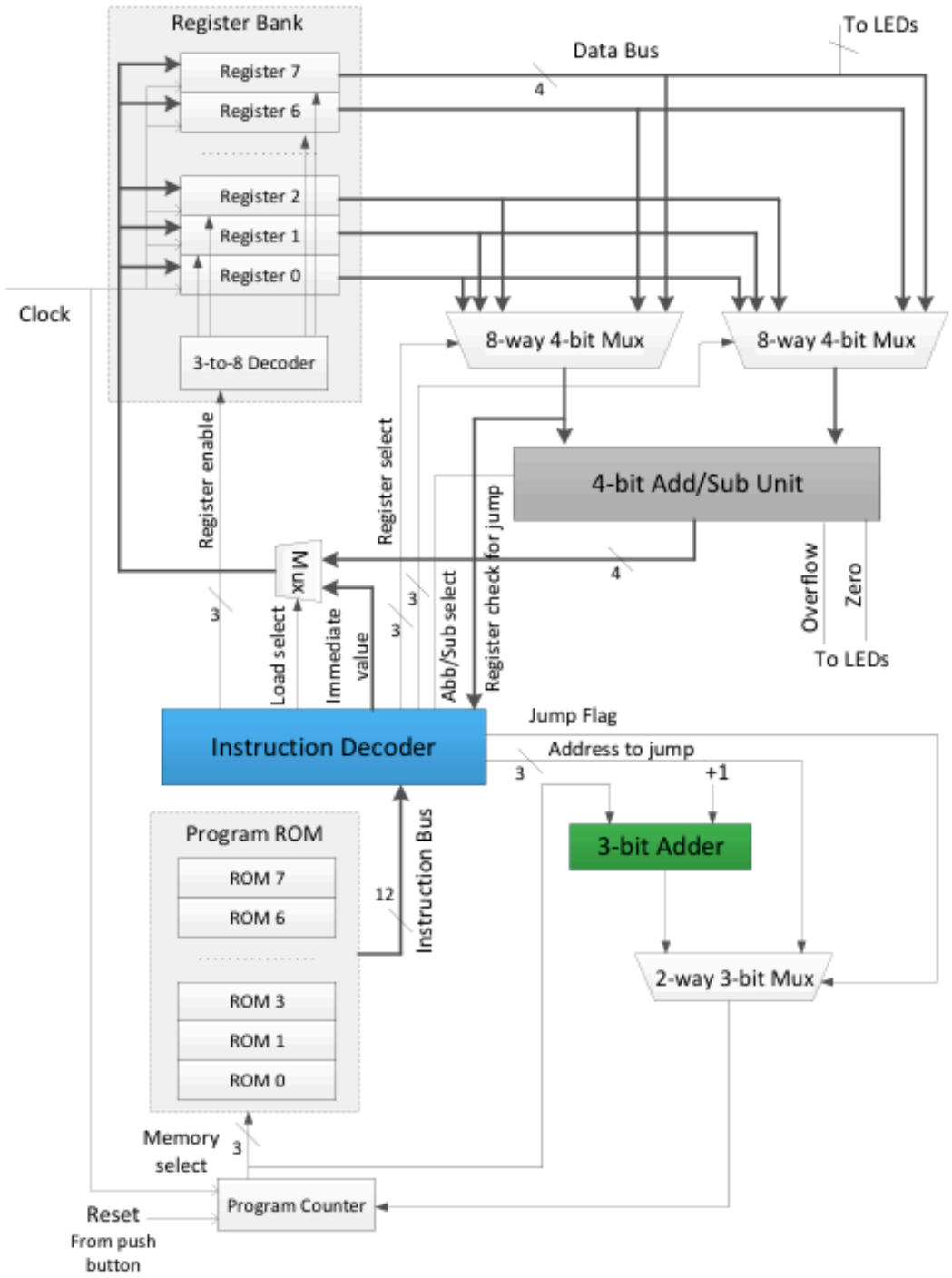
3.1 Original Version

```
        "101110000000", -- MOVI R1 , 0  ; Initialize R7 to 0
        "100010000001", -- MOVI R1 , 1  ; Initialize R1 to 1
        "100100000010", -- MOVI R1 , 2  ; Initialize R2 to 2
        "100110000011", -- MOVI R2 , 3  ; Initialize R3 to 3
        "001110010000", -- ADD R7 , R1  ; R7 <= R1 + R7
        "001110100000", -- ADD R7 , R2  ; R7 <= R2 + R7
        "001110110000", -- ADD R7 , R3  ; R7 <= R3 + R7
        "110000000100"  -- JZR R0 , 7   ; If R0 == 0, jump to address 4
```

3.2 Extended Version

```
    "101110000101",     -- 0: MOVI R7,5  ; R7 ← 5 (0101)
    "101100000011",     -- 1: MOVI R6,3  ; R6 ← 3 (0011)
    "001111100000",     -- 2: ADD R7,R6  ; R7 ← R7 + R6 = 8  (1000)
    "001101110001",     -- 3: SUB R6,R7  ; R6 ← R6 - R7 = -5 (1011)
    "001111100101",     -- 4: MUL R7,R6  ; R7 ← R7 * R6 = 8 (with overflow)
    "001101110011",     -- 5: OR R6,R7   ; R6 ← R6 OR R7 = 11 OR 8 = 11 (1011)
    "001111100100",     -- 6: XOR R7,R6  ; R7 ← R7 XOR R6 = 8 XOR 11 = 3 (0011)
    "001111000111"      -- 7: CMP R7,R6  ; Compare R7(3) with R6(11): GT=1
```

4. High-level Architecture (Original Nano Processor)



5. Constants and Bus Definitions

5.1 Original Constants Package

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

package Constants is

    -- Clock
    constant clk_period : time := 10 ns;
    constant clk_half_period : time := clk_period / 2;

    -- OP Codes
    constant MOVI_OP : std_logic_vector(1 downto 0) := "10";
    constant ADD_OP : std_logic_vector(1 downto 0) := "00";
    constant NEG_OP : std_logic_vector(1 downto 0) := "01";
    constant JZR_OP : std_logic_vector(1 downto 0) := "11";

end package Constants;
```

5.2 Extended Constants Package

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
package Constants is
    -- Clock
    constant clk_period : time := 10 ns;
    constant clk_half_period : time := clk_period / 2;
    -- OP Codes
    constant MOVI_OP : std_logic_vector(1 downto 0) := "10";
    constant ALU_OP : std_logic_vector(1 downto 0) := "00";
    constant NEG_OP : std_logic_vector(1 downto 0) := "01";
    constant JZR_OP : std_logic_vector(1 downto 0) := "11";

    -- ALU operation codes (3 bits)
    constant ALU_ADD : std_logic_vector(2 downto 0) := "000";  -- Addition
    constant ALU_SUB : std_logic_vector(2 downto 0) := "001";  -- Subtraction
    constant ALU_NEG : std_logic_vector(2 downto 0) := "010";  -- Negation (2's complement)
    -- Bitwise Operations
    constant ALU_AND : std_logic_vector(2 downto 0) := "011";  -- Bitwise AND
    constant ALU_OR  : std_logic_vector(2 downto 0) := "100";  -- Bitwise OR
    constant ALU_XOR : std_logic_vector(2 downto 0) := "101";  -- Bitwise XOR
    -- Multiplication operation
    constant ALU_MUL : std_logic_vector(2 downto 0) := "110";  -- Multiplication
    --comparison operations
    constant ALU_CMP : std_logic_vector(2 downto 0) := "111";  -- Compare

    -- ALU sub-opcodes (last 4 bits)
    constant SUBOP_ADD : std_logic_vector(3 downto 0) := "0000";  -- ADD instruction
    constant SUBOP_SUB : std_logic_vector(3 downto 0) := "0001";  -- SUB instruction
    constant SUBOP_AND : std_logic_vector(3 downto 0) := "0010";  -- AND instruction
    constant SUBOP_OR  : std_logic_vector(3 downto 0) := "0011";  -- OR instruction
    constant SUBOP_XOR : std_logic_vector(3 downto 0) := "0100";  -- XOR instruction
    constant SUBOP_MUL : std_logic_vector(3 downto 0) := "0101";  -- MULTIPLY instruction
    constant SUBOP_CMP : std_logic_vector(3 downto 0) := "0111";  -- COMPARE instruction
end package Constants;
```

5.3 Bus Definitions Package

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

package BusDefinitions is
    -- Standard width vector definitions
    subtype std_logic_vector_3 is std_logic_vector(2 downto 0);   -- 3-bit vector
    subtype std_logic_vector_4 is std_logic_vector(3 downto 0);   -- 4-bit vector
    subtype std_logic_vector_8 is std_logic_vector(7 downto 0);   -- 8-bit vector
    subtype std_logic_vector_12 is std_logic_vector(11 downto 0); -- 12-bit vector

    -- Composite data structures
    type reg_array_8x4 is array (7 downto 0) of std_logic_vector_4;  -- 8x4-bit register array
    type mem_array_4x8 is array (3 downto 0) of std_logic_vector_8;  -- 4x8-bit memory array

    -- Architecture-specific bus definitions
    subtype RegisterFile is reg_array_8x4;         -- Collection of 8 registers, each 4 bits wide
    subtype ProgramCounter is std_logic_vector_3;  -- 3-bit address pointer for program memory (0-7)
    subtype InstructionWord is std_logic_vector_12; -- 12-bit machine instruction format
    subtype DataBus is std_logic_vector_4;       -- 4-bit data value used in calculations
    subtype RegisterSelect is std_logic_vector_3;  -- 3-bit identifier to select registers (R0-R7)
    subtype OperationSelection is std_logic_vector_3;

end package BusDefinitions;
```

6. Original Nanoprocessor Top Module

Design Source Code

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.BusDefinitions.all;

entity NanoProcessor is
    port(
        Clock           : in  std_logic;  -- System clock
        Reset           : in  std_logic;  -- Asynchronous reset
        Overflow        : out std_logic;  -- Overflow flag from ALU
        Zero            : out std_logic;  -- Zero flag from ALU
        Equal           : out STD_LOGIC;
        LessThan        : out STD_LOGIC;
        GreaterThan     : out STD_LOGIC;
        S_7Seg          : out STD_LOGIC_VECTOR(6 downto 0);
        anode           : out STD_LOGIC_VECTOR(3 downto 0);
        Data            : out DataBus;     -- Output data (Register 7 contents)
        Test            :out DataBus

    );
end NanoProcessor;

architecture Behavioral of NanoProcessor is
    -- Clock and reset signals
    signal s_Clk : std_logic;
    signal s_Res : std_logic;
    signal s_SlowClk : std_logic;

    -- Program counter and addressing signals
    signal s_PCCurrent        : ProgramCounter;  -- Current program counter value
    signal s_PCNext           : ProgramCounter;  -- PC + 1 value
    signal s_JumpAddr         : ProgramCounter;  -- Jump target address
    signal s_JumpEn           : std_logic;       -- Jump control signal
    signal s_SelectedAddr     : ProgramCounter;  -- Selected next address
```

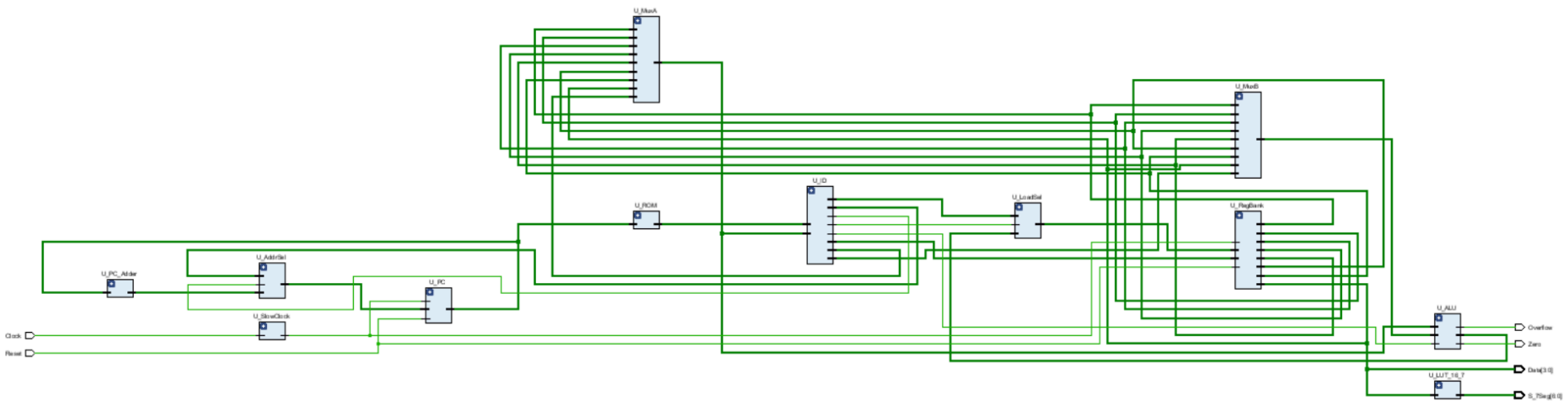
```
    -- Instruction and decoding signals
    signal s_Instruction      : InstructionWord;   -- Current instruction
    signal s_RegEn            : RegisterSelect;    -- Register write enable
    signal s_RegSelA          : RegisterSelect;    -- First operand select
    signal s_RegSelB          : RegisterSelect;    -- Second operand select
    signal s_OpSelect         : OperationSelection;         -- Operation select (0=ADD, 1=SUB)
    signal s_ImmValue         : DataBus;           -- Immediate value
    signal s_LoadSel          : std_logic;         -- Load select (0=imm, 1=ALU)

    -- Data path signals
    signal s_RegFileOutputs   : RegisterFile;      -- All register values
    signal s_OperandA         : DataBus;           -- First ALU operand
    signal s_OperandB         : DataBus;           -- Second ALU operand
    signal s_ALUResult        : DataBus;           -- ALU result
    signal s_WriteData        : DataBus;           -- Data to write to register

    -- Components
    component Program_Counter
        Port (
            PC_Next    : in  ProgramCounter;
            Res        : in  STD_LOGIC;
            Clk        : in  STD_LOGIC;
            PC_Current : out ProgramCounter
        );
    end component;

    component PC_Adder
        port (
            current_address : in  ProgramCounter;
            next_address    : out ProgramCounter
        );
    end component;

    component Address_Selector
        port (
            Sequential_Address : in  ProgramCounter;
            Jump_Address       : in  ProgramCounter;
            Jump_Enable        : in  std_logic;
            Selected_Address   : out ProgramCounter
        );
    end component;

    component Program_ROM
        port(
            program_counter : in  ProgramCounter;
            instruction_out : out InstructionWord
        );
    end component;

    component Instruction_Decoder
        port (
            Instruction           : in  InstructionWord;
            Register_Value_For_Jump : in  DataBus;
            Register_Enable       : out RegisterSelect;
            Register_Select_A     : out RegisterSelect;
            Register_Select_B     : out RegisterSelect;
            Operation_Select      : out OperationSelection;
            Immediate_Value       : out DataBus;
            Jump_Enable           : out STD_LOGIC;
            Jump_Address          : out ProgramCounter;
            Load_Select           : out STD_LOGIC
        );
    end component;

    component Load_Selector
        Port (
            RegisterValue  : in  DataBus;
            ImmediateValue : in  DataBus;
            LoadSelect     : in  STD_LOGIC;
```

```
        OutputData     : out DataBus
    );
end component;

component Register_Bank
    Port (
        Data             : in  DataBus;
        Reset            : in  STD_LOGIC;
        Reg_En           : in  RegisterSelect;
        Clock            : in  STD_LOGIC;
        Register_Outputs : out RegisterFile
    );
end component;

component RegisterData_Multiplexer
    Port (
        DataSources   : in  RegisterFile;
        SelectAddress : in  RegisterSelect;
        OutputData    : out DataBus
    );
end component;

component ALU
    Port(
        A         : in  DataBus;
        B         : in  DataBus;
        Operation : in  OperationSelection;
        Result    : out DataBus;
        Zero      : out STD_LOGIC;
        Overflow  : out STD_LOGIC;
        Equal           : out STD_LOGIC;
        LessThan        : out STD_LOGIC;
        GreaterThan     : out STD_LOGIC
    );
end component;

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

component LUT_16_7
    Port (
        address : in  DataBus;
        data    : out STD_LOGIC_VECTOR (6 downto 0)
    );
end component;
    );
    end component;

begin
    -- Connect top-level ports to internal signals
    s_Clk <= Clock;
    s_Res <= Reset;
    anode <= "1110";

   U_SlowClock : Slow_Clk
        port map (
        Clk_in  => s_Clk,
        Clk_out => s_SlowClk
        );


    -- Output register 7 data
    Data <= s_RegFileOutputs(7);
    Test <= s_RegFileOutputs(6);

    -- Program counter and instruction fetch
    U_PC : Program_Counter
```

```
    port map (
        PC_Next    => s_SelectedAddr,
        Res        => s_Res,
        Clk        => s_SlowClk,
        PC_Current => s_PCCurrent
    );

U_PC_Adder : PC_Adder
    port map (
        current_address => s_PCCurrent,
        next_address    => s_PCNext
    );

U_AddrSel : Address_Selector
    port map (
        Sequential_Address => s_PCNext,
        Jump_Address       => s_JumpAddr,
        Jump_Enable        => s_JumpEn,
        Selected_Address   => s_SelectedAddr
    );

U_ROM : Program_ROM
    port map(
        program_counter => s_PCCurrent,
        instruction_out => s_Instruction
    );

-- Instruction decode
U_ID : Instruction_Decoder
    port map(
        Instruction           => s_Instruction,
        Register_Value_For_Jump => s_OperandA,
        Register_Enable       => s_RegEn,
        Register_Select_A     => s_RegSelA,
        Register_Select_B     => s_RegSelB,
        Operation_Select      => s_OpSelect,
        Immediate_Value       => s_ImmValue,
        Jump_Enable           => s_JumpEn,
        Jump_Address          => s_JumpAddr,
        Load_Select           => s_LoadSel
    );

-- Register file and ALU data path
U_LoadSel : Load_Selector
    port map(
        RegisterValue  => s_ALUResult,
        ImmediateValue => s_ImmValue,
        LoadSelect     => s_LoadSel,
        OutputData     => s_WriteData
    );

U_RegBank : Register_Bank
    port map(
        Data             => s_WriteData,
        Reset            => s_Res,
        Reg_En           => s_RegEn,
        Clock            => s_SlowClk,
        Register_Outputs => s_RegFileOutputs
    );

U_MuxA : RegisterData_Multiplexer
    port map(
        DataSources   => s_RegFileOutputs,
        SelectAddress => s_RegSelA,
        OutputData    => s_OperandA
    );

U_MuxB : RegisterData_Multiplexer
    port map(
```

```
            DataSources   => s_RegFileOutputs,
            SelectAddress => s_RegSelB,
            OutputData    => s_OperandB
        );

    U_ALU : ALU
        port map(
            A       => s_OperandA,
            B       => s_OperandB,
            Operation      => s_OpSelect,
            Result        => s_ALUResult,
            Zero     => Zero,
            Overflow => Overflow,
            Equal     =>Equal,
            LessThan   =>LessThan,
            GreaterThan  =>GreaterThan
        );

     U_LUT_16_7 : LUT_16_7
           port map(
               address => s_RegFileOutputs(7),
               data => S_7Seg);
end Behavioral;
```

Elaborated Design Schematic



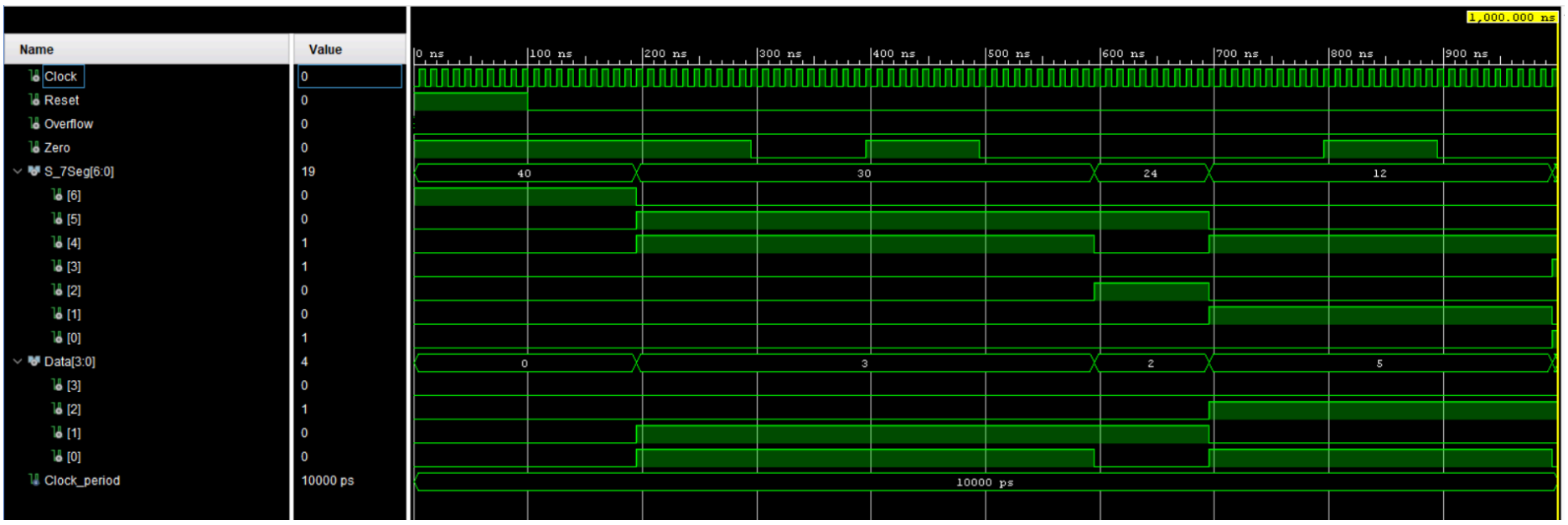
Behavioral Simulation

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.BusDefinitions.all;

entity NanoProcessor_TB is
-- No ports for testbench
end NanoProcessor_TB;

architecture Behavioral of NanoProcessor_TB is
    -- Component declaration for the Unit Under Test (UUT)
    component NanoProcessor
        port(
            Clock    : in  std_logic;
            Reset    : in  std_logic;
            Overflow : out std_logic;
            Zero     : out std_logic;
```

```
            Data     : out DataBus
        );
    end component;

    -- Inputs
    signal tb_Clock    : std_logic := '0';
    signal tb_Reset    : std_logic := '0';

    -- Outputs
    signal tb_Overflow : std_logic;
    signal tb_Zero     : std_logic;
    signal tb_Data     : DataBus;

    -- Clock period definition
    constant CLOCK_PERIOD : time := 10 ns;

begin
    -- Instantiate the Unit Under Test (UUT)
    UUT: NanoProcessor
    port map (
        Clock    => tb_Clock,
        Reset    => tb_Reset,
        Overflow => tb_Overflow,
        Zero     => tb_Zero,
        Data     => tb_Data
    );

    -- Clock process
    clk_process: process
    begin
        tb_Clock <= '0';
        wait for CLOCK_PERIOD/2;
        tb_Clock <= '1';
        wait for CLOCK_PERIOD/2;
    end process;

    -- Stimulus process
    stim_proc: process
    begin
        -- Hold reset for 2 clock cycles
        tb_Reset <= '1';
        wait for CLOCK_PERIOD*2;

        -- Release reset
        tb_Reset <= '0';

        -- Wait for program execution (adjust based on program length)
        -- This allows for approximately 20 instructions to execute
        wait for CLOCK_PERIOD*20;

        -- Optional: Re-apply reset to restart program
        tb_Reset <= '1';
        wait for CLOCK_PERIOD*2;
        tb_Reset <= '0';

        -- Continue execution
        wait for CLOCK_PERIOD*20;

        -- End simulation
        wait;
    end process;

    -- Monitor process (optional)
    monitor_proc: process(tb_Clock)
    begin
        if rising_edge(tb_Clock) then
            if tb_Reset = '0' then
            end if;
        end if;
```

```
    end process;
end Behavioral;
```

Timing Diagram



7. Extended Nanoprocessor Top Module

Design Source Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.BusDefinitions.all;

entity NanoProcessor is
    port(
        Clock    : in  std_logic;  -- System clock
        Reset    : in  std_logic;  -- Asynchronous reset
        Overflow : out std_logic;  -- Overflow flag from ALU
        Zero     : out std_logic;  -- Zero flag from ALU
        Equal           : out STD_LOGIC;
        LessThan        : out STD_LOGIC;
        GreaterThan     : out STD_LOGIC;
        S_7Seg  : out STD_LOGIC_VECTOR(6 downto 0);
        anode   : out STD_LOGIC_VECTOR(3 downto 0);
        Data_7     : out DataBus;     -- Output data (Register 7 contents)
        Data_6     :out DataBus       -- Output data (Register 6 contents)

    );
end NanoProcessor;

architecture Behavioral of NanoProcessor is
    -- Clock and reset signals
    signal s_Clk : std_logic;
    signal s_Res : std_logic;
    signal s_SlowClk : std_logic;

    -- Program counter and addressing signals
    signal s_PCCurrent        : ProgramCounter;  -- Current program counter value
    signal s_PCNext           : ProgramCounter;  -- PC + 1 value
    signal s_JumpAddr         : ProgramCounter;  -- Jump target address
    signal s_JumpEn           : std_logic;       -- Jump control signal
    signal s_SelectedAddr     : ProgramCounter;  -- Selected next address

    -- Instruction and decoding signals
```

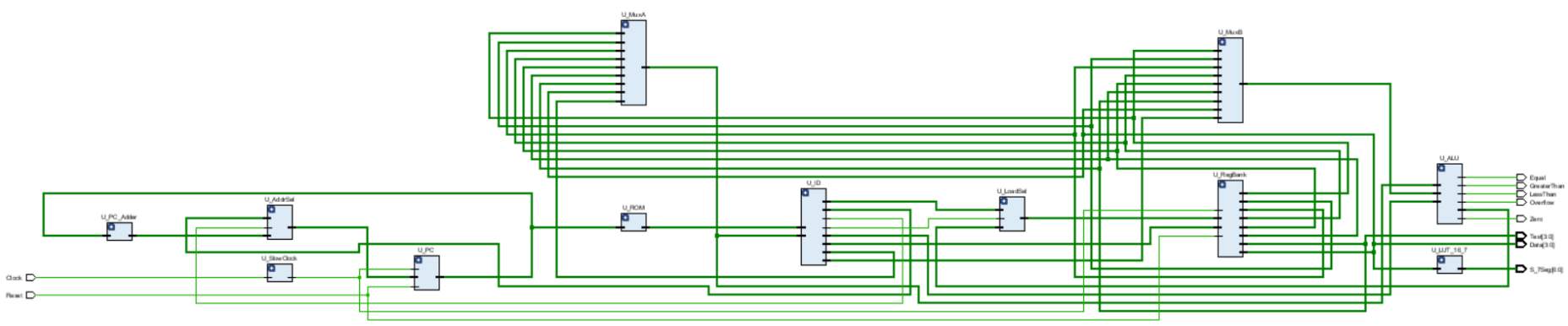
```
    signal s_Instruction      : InstructionWord;   -- Current instruction
    signal s_RegEn            : RegisterSelect;    -- Register write enable
    signal s_RegSelA          : RegisterSelect;    -- First operand select
    signal s_RegSelB          : RegisterSelect;    -- Second operand select
    signal s_OpSelect         : OperationSelection;         -- Operation select (0=ADD, 1=SUB)
    signal s_ImmValue         : DataBus;           -- Immediate value
    signal s_LoadSel          : std_logic;         -- Load select (0=imm, 1=ALU)

    -- Data path signals
    signal s_RegFileOutputs   : RegisterFile;      -- All register values
    signal s_OperandA         : DataBus;           -- First ALU operand
    signal s_OperandB         : DataBus;           -- Second ALU operand
    signal s_ALUResult        : DataBus;           -- ALU result
    signal s_WriteData        : DataBus;           -- Data to write to register

    -- Components
    component Program_Counter
        Port (
            PC_Next    : in  ProgramCounter;
            Res        : in  STD_LOGIC;
            Clk        : in  STD_LOGIC;
            PC_Current : out ProgramCounter
        );
    end component;

    component PC_Adder
        port (
            current_address : in  ProgramCounter;
            next_address    : out ProgramCounter
        );
    end component;

    component Address_Selector
        port (
            Sequential_Address : in  ProgramCounter;
            Jump_Address       : in  ProgramCounter;
            Jump_Enable        : in  std_logic;
            Selected_Address   : out ProgramCounter
        );
    end component;

    component Program_ROM
        port(
            program_counter : in  ProgramCounter;
            instruction_out : out InstructionWord
        );
    end component;

    component Instruction_Decoder
        port (
            Instruction           : in  InstructionWord;
            Register_Value_For_Jump : in  DataBus;
            Register_Enable       : out RegisterSelect;
            Register_Select_A     : out RegisterSelect;
            Register_Select_B     : out RegisterSelect;
            Operation_Select      : out OperationSelection;
            Immediate_Value       : out DataBus;
            Jump_Enable           : out STD_LOGIC;
            Jump_Address          : out ProgramCounter;
            Load_Select           : out STD_LOGIC
        );
    end component;

    component Load_Selector
        Port (
            RegisterValue  : in  DataBus;
            ImmediateValue : in  DataBus;
            LoadSelect     : in  STD_LOGIC;
            OutputData     : out DataBus
```

```
    );
end component;

component Register_Bank
    Port (
        Data             : in  DataBus;
        Reset            : in  STD_LOGIC;
        Reg_En           : in  RegisterSelect;
        Clock            : in  STD_LOGIC;
        Register_Outputs : out RegisterFile
    );
end component;

component RegisterData_Multiplexer
    Port (
        DataSources   : in  RegisterFile;
        SelectAddress : in  RegisterSelect;
        OutputData    : out DataBus
    );
end component;

component ALU
    Port(
        A         : in  DataBus;
        B         : in  DataBus;
        Operation : in  OperationSelection;
        Result    : out DataBus;
        Zero      : out STD_LOGIC;
        Overflow  : out STD_LOGIC;
        Equal           : out STD_LOGIC;
        LessThan        : out STD_LOGIC;
        GreaterThan     : out STD_LOGIC
    );
end component;

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

component LUT_16_7
    Port (
        address : in  DataBus;
        data    : out STD_LOGIC_VECTOR (6 downto 0)
    );
end component;

begin
    -- Connect top-level ports to internal signals
    s_Clk <= Clock;
    s_Res <= Reset;
    anode <= "1110";

   U_SlowClock : Slow_Clk
        port map (
        Clk_in  => s_Clk,
        Clk_out => s_SlowClk
        );


    -- Output register 7 data
    Data_7 <= s_RegFileOutputs(7);
    Data_6 <= s_RegFileOutputs(6);

    -- Program counter and instruction fetch
    U_PC : Program_Counter
        port map (
```

```
        PC_Next    => s_SelectedAddr,
        Res        => s_Res,
        Clk        => s_SlowClk,
        PC_Current => s_PCCurrent
    );

U_PC_Adder : PC_Adder
    port map (
        current_address => s_PCCurrent,
        next_address    => s_PCNext
    );

U_AddrSel : Address_Selector
    port map (
        Sequential_Address => s_PCNext,
        Jump_Address       => s_JumpAddr,
        Jump_Enable        => s_JumpEn,
        Selected_Address   => s_SelectedAddr
    );

U_ROM : Program_ROM
    port map(
        program_counter => s_PCCurrent,
        instruction_out => s_Instruction
    );

-- Instruction decode
U_ID : Instruction_Decoder
    port map(
        Instruction           => s_Instruction,
        Register_Value_For_Jump => s_OperandA,
        Register_Enable       => s_RegEn,
        Register_Select_A     => s_RegSelA,
        Register_Select_B     => s_RegSelB,
        Operation_Select      => s_OpSelect,
        Immediate_Value       => s_ImmValue,
        Jump_Enable           => s_JumpEn,
        Jump_Address          => s_JumpAddr,
        Load_Select           => s_LoadSel
    );

-- Register file and ALU data path
U_LoadSel : Load_Selector
    port map(
        RegisterValue  => s_ALUResult,
        ImmediateValue => s_ImmValue,
        LoadSelect     => s_LoadSel,
        OutputData     => s_WriteData
    );

U_RegBank : Register_Bank
    port map(
        Data             => s_WriteData,
        Reset            => s_Res,
        Reg_En           => s_RegEn,
        Clock            => s_SlowClk,
        Register_Outputs => s_RegFileOutputs
    );

U_MuxA : RegisterData_Multiplexer
    port map(
        DataSources   => s_RegFileOutputs,
        SelectAddress => s_RegSelA,
        OutputData    => s_OperandA
    );

U_MuxB : RegisterData_Multiplexer
    port map(
        DataSources   => s_RegFileOutputs,
```

```
            SelectAddress => s_RegSelB,
            OutputData    => s_OperandB
        );

    U_ALU : ALU
        port map(
            A       => s_OperandA,
            B       => s_OperandB,
            Operation      => s_OpSelect,
            Result        => s_ALUResult,
            Zero     => Zero,
            Overflow => Overflow,
            Equal     =>Equal,
            LessThan   =>LessThan,
            GreaterThan  =>GreaterThan
        );

     U_LUT_16_7 : LUT_16_7
           port map(
               address => s_RegFileOutputs(7),
               data => S_7Seg);
end Behavioral;
```

Elaborated Design Schematic

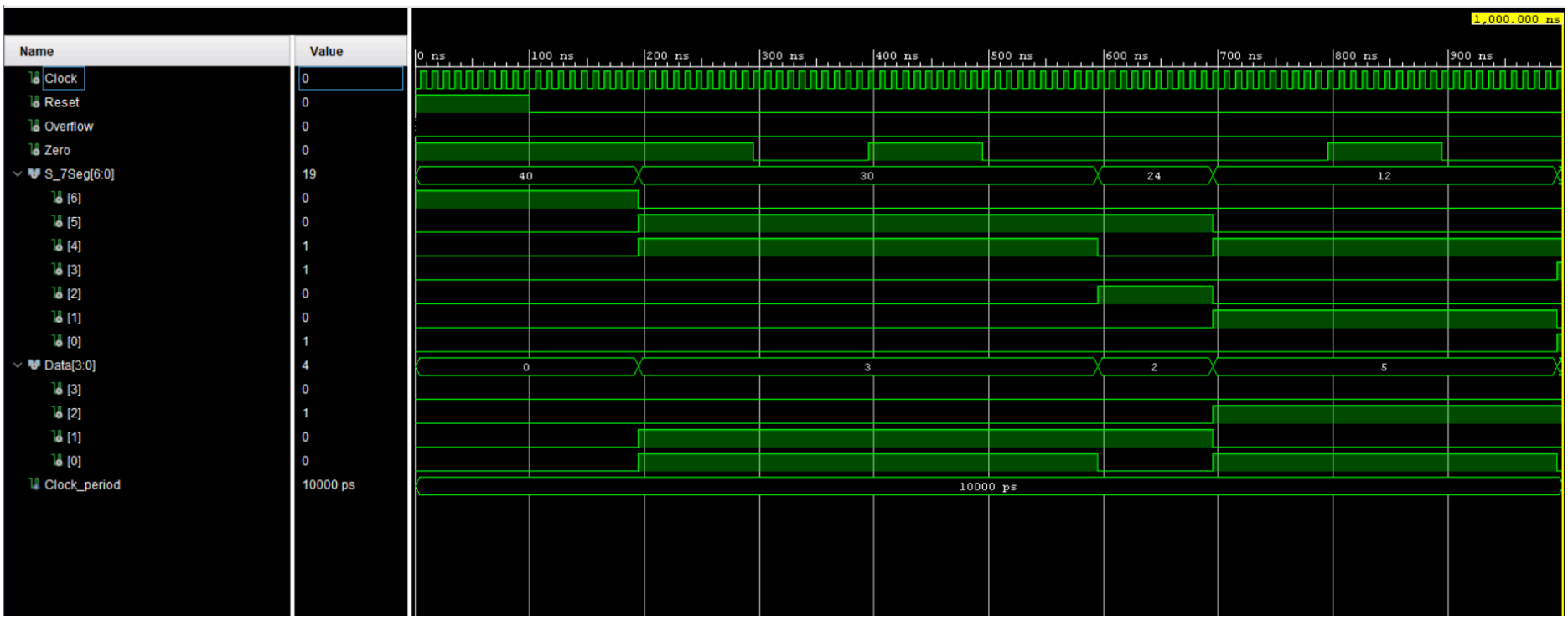


Behavioral Simulation

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.BusDefinitions.all;

entity NanoProcessor_TB is
-- No ports in testbench
end NanoProcessor_TB;

architecture Behavioral of NanoProcessor_TB is
    -- Component declaration
    component NanoProcessor
        port(
            Clock        : in  std_logic;
            Reset        : in  std_logic;
            Overflow     : out std_logic;
            Zero         : out std_logic;
            Equal        : out STD_LOGIC;
            LessThan     : out STD_LOGIC;
            GreaterThan  : out STD_LOGIC;
            S_7Seg       : out STD_LOGIC_VECTOR(6 downto 0);
```

```
            Data_7       : out DataBus;
            Data_6       : out DataBus
        );
    end component;

    -- Input signals
    signal Clock    : std_logic := '0';
    signal Reset    : std_logic := '0';

    -- Output signals
    signal Overflow     : std_logic;
    signal Zero         : std_logic;
    signal Equal        : std_logic;
    signal LessThan     : std_logic;
    signal GreaterThan  : std_logic;
    signal S_7Seg       : STD_LOGIC_VECTOR(6 downto 0);
    signal Data_7       : DataBus;
    signal Data_6       : DataBus;

begin
    -- Instantiate NanoProcessor
    UUT: NanoProcessor port map(
        Clock        => Clock,
        Reset        => Reset,
        Overflow     => Overflow,
        Zero         => Zero,
        Equal        => Equal,
        LessThan     => LessThan,
        GreaterThan  => GreaterThan,
        S_7Seg       => S_7Seg,
        Data_7       => Data_7,
        Data_6       => Data_6
    );

    -- Clock process
    Clock_process: process
    begin
        Clock <= '0';
        wait for 50 ns;
        Clock <= '1';
        wait for 50 ns;
    end process;

    -- Stimulus process
    Stimulus_process: process
    begin
        -- Initial reset
        Reset <= '1';
        wait for 100 ns;

        -- Release reset and let the program run
        Reset <= '0';

        -- Wait for program execution
        wait for 2000 ns;

        -- End simulation
        wait;
    end process;

end Behavioral;
```

Timing Diagram



8. Core Components - Original Nano Processor

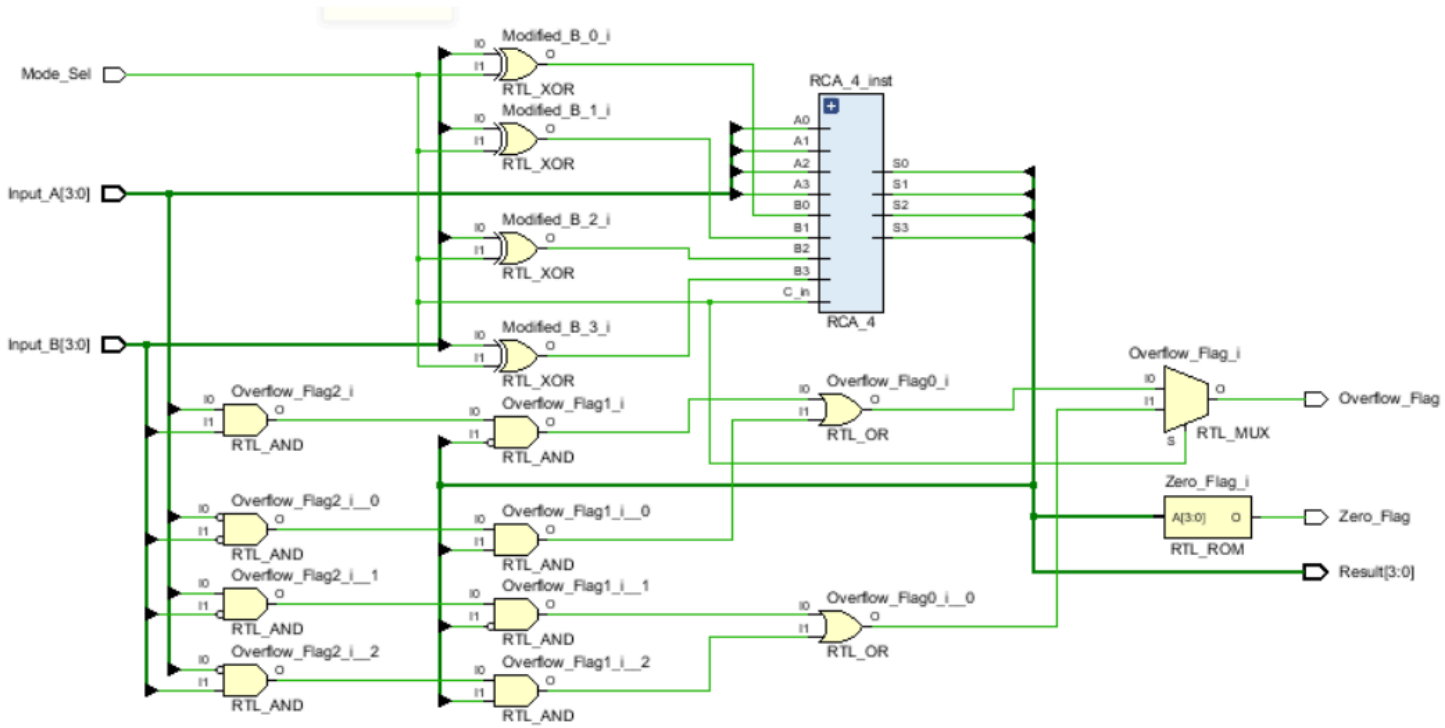
8.1 Add/Subtract Unit (4-bit)

Design Source Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.BusDefinitions.all;

entity Add_Sub_4_bit is
    Port(
        Input_A    : in  DataBus;
        Input_B    : in  DataBus;
        Mode_Sel   : in  STD_LOGIC;
        Result     : out DataBus;
        Zero_Flag  : out STD_LOGIC;
        Overflow_Flag : out STD_LOGIC
    );
end Add_Sub_4_bit;

architecture Behavioral of Add_Sub_4_bit 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;
            C_out : out STD_LOGIC
        );
    END COMPONENT;
```

```
    SIGNAL Modified_B, Temp_Result: STD_LOGIC_VECTOR(3 DOWNTO 0);
    SIGNAL RCA_Cout : STD_LOGIC; -- Carry output from RCA
begin
    RCA_4_inst : RCA_4 port map(
        A0 => Input_A(0),
        A1 => Input_A(1),
        A2 => Input_A(2),
        A3 => Input_A(3),
        B0 => Modified_B(0),
        B1 => Modified_B(1),
        B2 => Modified_B(2),
        B3 => Modified_B(3),
        C_in => Mode_Sel,
        S0 => Temp_Result(0),
        S1 => Temp_Result(1),
        S2 => Temp_Result(2),
        S3 => Temp_Result(3),
        C_out => RCA_Cout -- Store Carry Output in a signal instead of directly connecting to Overflow_Flag
    );

    -- XOR each bit of Input_B with the Mode_Sel signal for 2's complement in subtraction
    Modified_B(0) <= Input_B(0) XOR Mode_Sel;
    Modified_B(1) <= Input_B(1) XOR Mode_Sel;
    Modified_B(2) <= Input_B(2) XOR Mode_Sel;
    Modified_B(3) <= Input_B(3) XOR Mode_Sel;

    -- Process to determine if result is zero
    process (Temp_Result)
    begin
        if Temp_Result = "0000" then
            Zero_Flag <= '1';
        else
            Zero_Flag <= '0';
        end if;
    end process;

    -- True overflow detection for both addition and subtraction
    -- For addition: Overflow occurs when adding two numbers of the same sign results in a different sign
    -- For subtraction: Overflow occurs when subtracting two numbers of different signs results in a sign not
matching A
    process (Input_A, Input_B, Temp_Result, Mode_Sel)
    begin
        if Mode_Sel = '0' then
            -- Addition mode: Check if inputs have same sign but result has different sign
            Overflow_Flag <= (Input_A(3) AND Input_B(3) AND (NOT Temp_Result(3))) OR ((NOT Input_A(3)) AND (NOT
Input_B(3)) AND Temp_Result(3));
        else
            -- Subtraction mode: Check if inputs have different signs and result's sign doesn't match Input_A
            -- Since B is already complemented for subtraction, we need to check NOT(Input_B(3))
            Overflow_Flag <= (Input_A(3) AND (NOT Input_B(3)) AND (NOT Temp_Result(3))) OR ((NOT Input_A(3)) AND
Input_B(3) AND Temp_Result(3));
        end if;
    end process;

    -- Transfer temporary result to output
    Result <= Temp_Result;
end Behavioral;
```

Elaborated Design Schematic

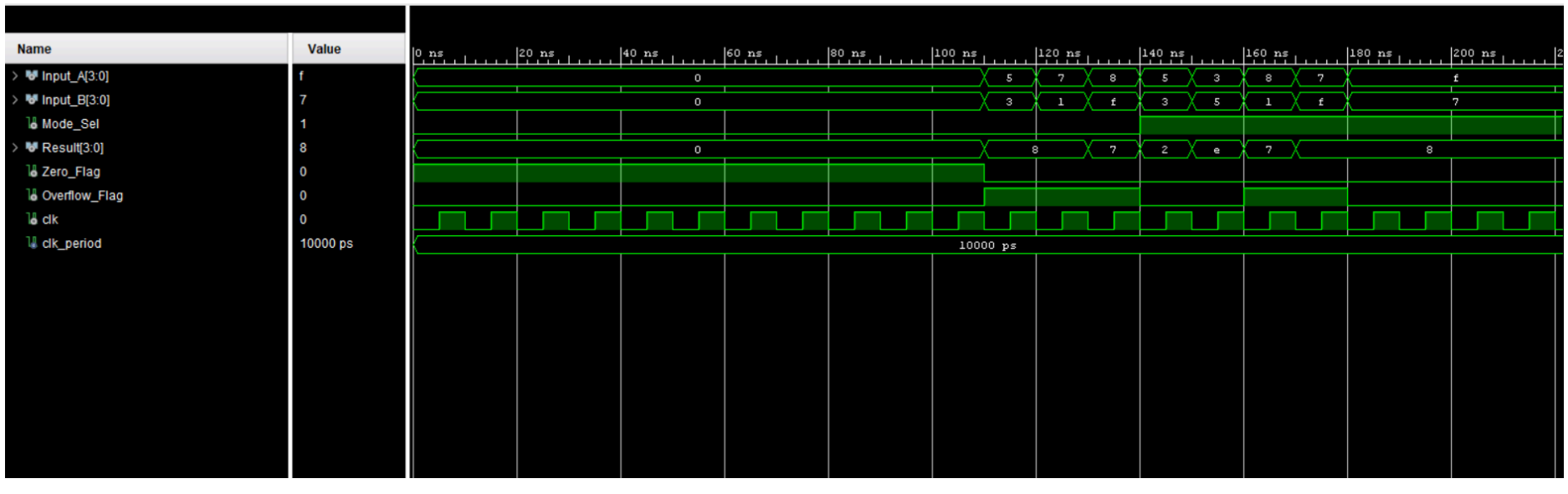


Behavioural Simulation Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.BusDefinitions.all;

entity Add_Sub_4_bit is
    Port(
        Input_A    : in  DataBus;
        Input_B    : in  DataBus;
        Mode_Sel   : in  STD_LOGIC;
        Result     : out DataBus;
        Zero_Flag  : out STD_LOGIC;
        Overflow_Flag : out STD_LOGIC
    );
end Add_Sub_4_bit;

architecture Behavioral of Add_Sub_4_bit 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;
            C_out : out STD_LOGIC
        );
    END COMPONENT;

    SIGNAL Modified_B, Temp_Result: STD_LOGIC_VECTOR(3 DOWNTO 0);
    SIGNAL RCA_Cout : STD_LOGIC; -- Carry output from RCA
begin
    RCA_4_inst : RCA_4 port map(
        A0 => Input_A(0),
        A1 => Input_A(1),
```

```
        A2 => Input_A(2),
        A3 => Input_A(3),
        B0 => Modified_B(0),
        B1 => Modified_B(1),
        B2 => Modified_B(2),
        B3 => Modified_B(3),
        C_in => Mode_Sel,
        S0 => Temp_Result(0),
        S1 => Temp_Result(1),
        S2 => Temp_Result(2),
        S3 => Temp_Result(3),
        C_out => RCA_Cout -- Store Carry Output in a signal instead of directly connecting to Overflow_Flag
    );

    -- XOR each bit of Input_B with the Mode_Sel signal for 2's complement in subtraction
    Modified_B(0) <= Input_B(0) XOR Mode_Sel;
    Modified_B(1) <= Input_B(1) XOR Mode_Sel;
    Modified_B(2) <= Input_B(2) XOR Mode_Sel;
    Modified_B(3) <= Input_B(3) XOR Mode_Sel;

    -- Process to determine if result is zero
    process (Temp_Result)
    begin
        if Temp_Result = "0000" then
            Zero_Flag <= '1';
        else
            Zero_Flag <= '0';
        end if;
    end process;

    -- True overflow detection for both addition and subtraction
    -- For addition: Overflow occurs when adding two numbers of the same sign results in a different sign
    -- For subtraction: Overflow occurs when subtracting two numbers of different signs results in a sign not
matching A
    process (Input_A, Input_B, Temp_Result, Mode_Sel)
    begin
        if Mode_Sel = '0' then
            -- Addition mode: Check if inputs have same sign but result has different sign
            Overflow_Flag <= (Input_A(3) AND Input_B(3) AND (NOT Temp_Result(3))) OR
                             ((NOT Input_A(3)) AND (NOT Input_B(3)) AND Temp_Result(3));
        else
            -- Subtraction mode: Check if inputs have different signs and result's sign doesn't match Input_A
            -- Since B is already complemented for subtraction, we need to check NOT(Input_B(3))
            Overflow_Flag <= (Input_A(3) AND (NOT Input_B(3)) AND (NOT Temp_Result(3))) OR
                             ((NOT Input_A(3)) AND Input_B(3) AND Temp_Result(3));
        end if;
    end process;

    -- Transfer temporary result to output
    Result <= Temp_Result;
end Behavioral;
```

Timing Diagram

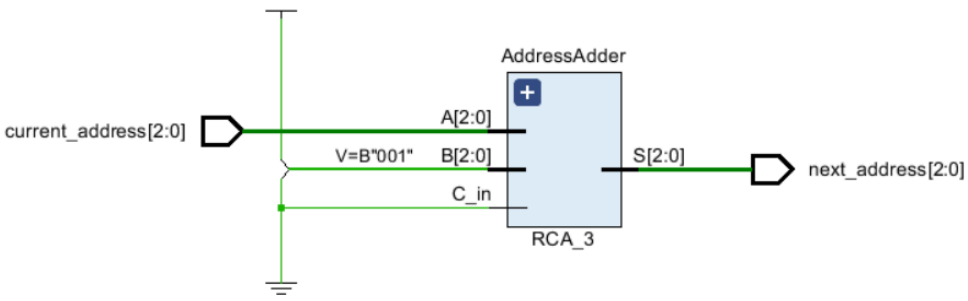


8.2 PC Adder (3-bit)

Design Source Code

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.BusDefinitions.ProgramCounter;

entity PC_Adder is
    port (
        current_address : in  ProgramCounter;  -- Current program counter value
        next_address    : out ProgramCounter   -- Next program counter value (current + 1)
    );
end PC_Adder;

architecture Behavioral of PC_Adder is
    COMPONENT RCA_3
        Port (
            A     : in  STD_LOGIC_VECTOR(2 downto 0);
            B     : in  STD_LOGIC_VECTOR(2 downto 0);
            C_in  : in  STD_LOGIC;
            S     : out STD_LOGIC_VECTOR(2 downto 0);
            C_out : out STD_LOGIC
        );
    END COMPONENT;

    constant INCREMENT_VALUE : std_logic_vector(2 downto 0) := "001";  -- Increment by 1
begin
    AddressAdder : RCA_3 port map (
        A     => current_address,
        B     => INCREMENT_VALUE,
        C_in  => '0',               -- No carry-in for simple addition
        S     => next_address,      -- Sum output is the incremented address
        C_out => open               -- Carry-out not used
    );
end Behavioral;
```

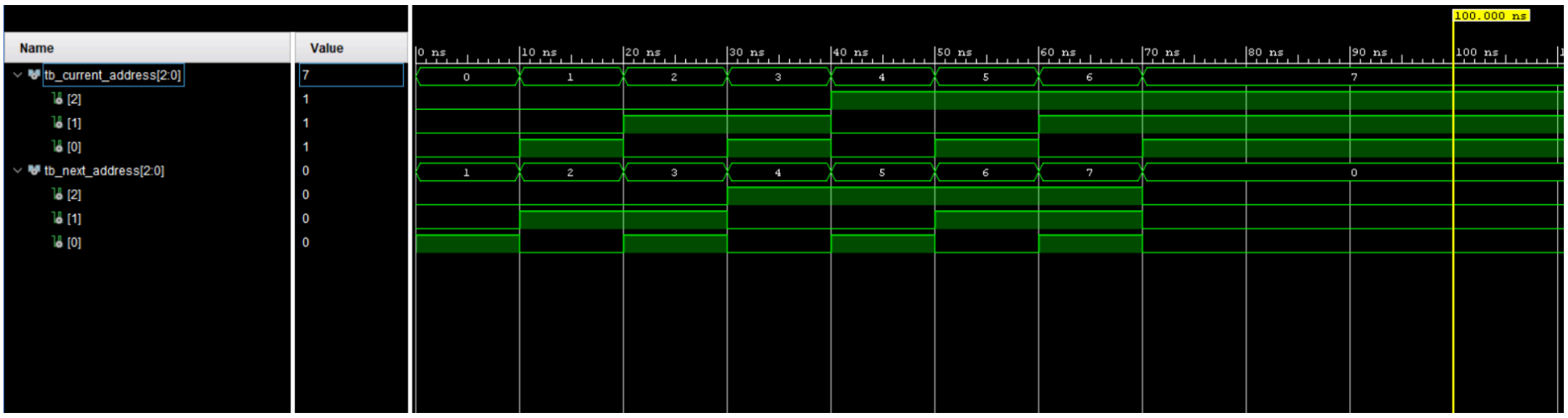
Elaborated Design Schematic



Behavioral Simulation Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.BusDefinitions.all;

entity PC_Adder_TB is
end PC_Adder_TB;

architecture Behavioral of PC_Adder_TB is

    -- Component declaration
    component PC_Adder is
        port (
            current_address : in  ProgramCounter;
            next_address    : out ProgramCounter
        );
    end component;

    -- Signals to connect to UUT
    signal tb_current_address : ProgramCounter := (others => '0');
    signal tb_next_address    : ProgramCounter;

begin

    -- Instantiate the Unit Under Test (UUT)
    UUT: PC_Adder
        port map (
            current_address => tb_current_address,
            next_address    => tb_next_address
        );

    -- Stimulus process
    stim_proc: process
    begin
        for i in 0 to 7 loop
            tb_current_address <= std_logic_vector(to_unsigned(i, tb_current_address'length));
            wait for 10 ns;
        end loop;

        wait;
    end process;

end Behavioral;
```

Timing Diagram

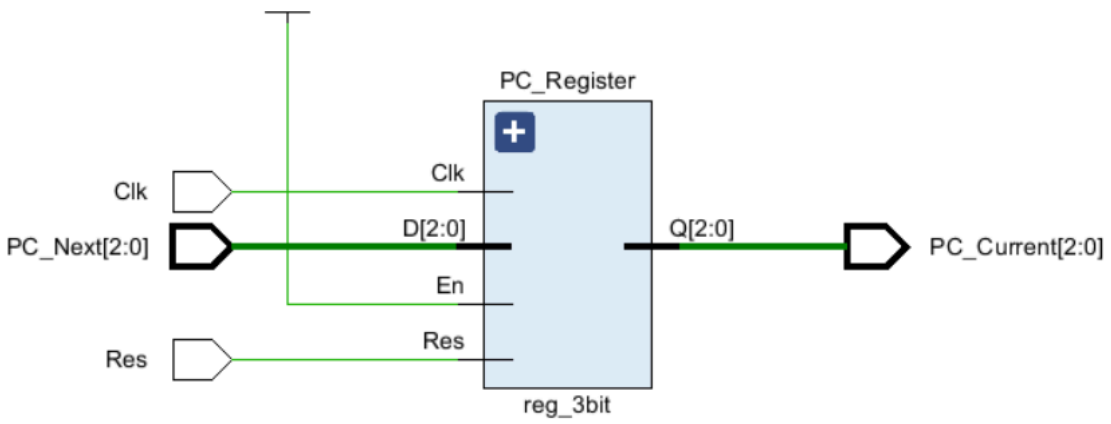


8.3 Program Counter

Design Source Code

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.BusDefinitions.all;

entity Program_Counter is
    Port (
        PC_Next    : in  ProgramCounter;
        Res        : in  STD_LOGIC;
        Clk        : in  STD_LOGIC;
        PC_Current : out ProgramCounter
    );
end Program_Counter;

architecture Behavioral of Program_Counter is
    component reg_3bit
        Port (
            D   : in  STD_LOGIC_VECTOR(2 downto 0);
            Res : in  STD_LOGIC;
            En  : in  STD_LOGIC;
            Clk : in  STD_LOGIC;
            Q   : out STD_LOGIC_VECTOR(2 downto 0)
        );
    end component;
begin
    PC_Register : reg_3bit
        port map(
            D   => PC_Next,
            Res => Res,
            En  => '1',
            Clk => Clk,
            Q   => PC_Current
        );
end Behavioral;
```

Elaborated Design Schematic

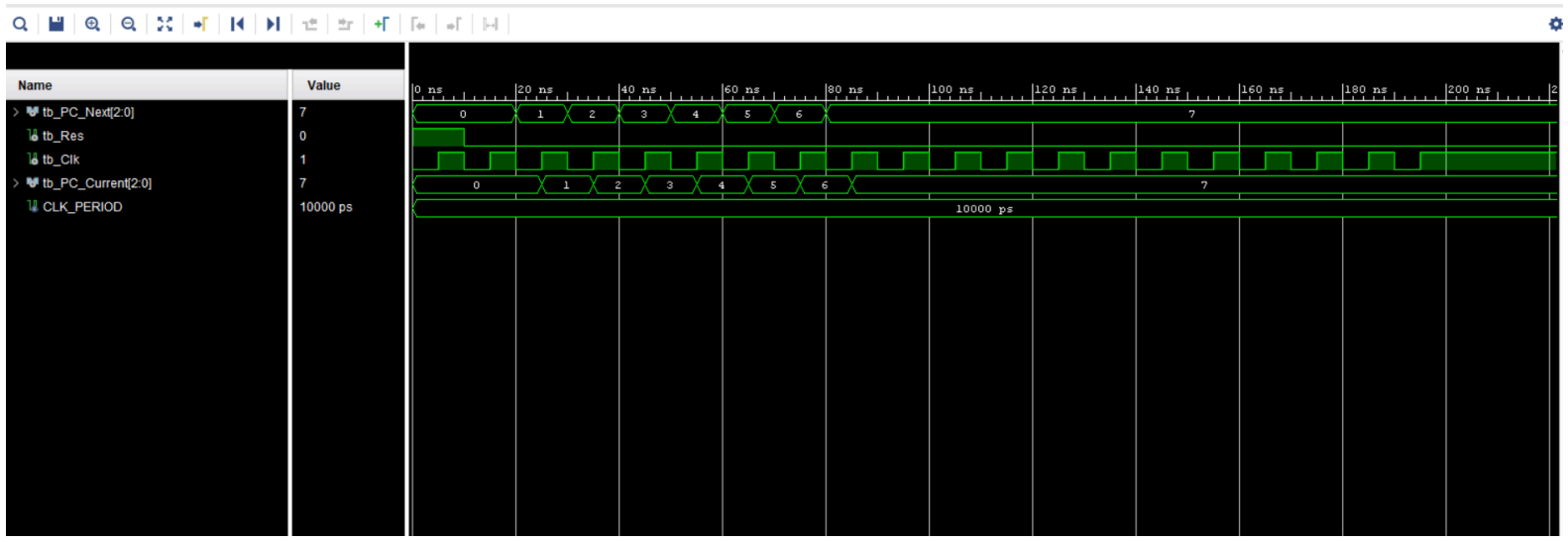


Behavioural Simulation Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.BusDefinitions.all;

entity Program_Counter_TB is
end Program_Counter_TB;

architecture Behavioral of Program_Counter_TB is

    -- Component under test
    component Program_Counter is
        Port (
            PC_Next    : in  ProgramCounter;
            Res        : in  STD_LOGIC;
            Clk        : in  STD_LOGIC;
            PC_Current : out ProgramCounter
        );
    end component;

    -- Signals
    signal tb_PC_Next    : ProgramCounter := (others => '0');
    signal tb_Res        : STD_LOGIC := '0';
    signal tb_Clk        : STD_LOGIC := '0';
    signal tb_PC_Current : ProgramCounter;

    -- Clock period
    constant CLK_PERIOD : time := 10 ns;

begin

    -- Instantiate the Unit Under Test (UUT)
    UUT: Program_Counter
        port map (
            PC_Next    => tb_PC_Next,
            Res        => tb_Res,
            Clk        => tb_Clk,
            PC_Current => tb_PC_Current
        );

    -- Clock generation process
    clk_process: process
    begin
        while now < 200 ns loop
            tb_Clk <= '0';
            wait for CLK_PERIOD / 2;
            tb_Clk <= '1';
            wait for CLK_PERIOD / 2;
        end loop;
        wait;
    end process;

    -- Stimulus process
    stim_proc: process
    begin
        -- Apply Reset
        tb_Res <= '1';
        wait for CLK_PERIOD;
        tb_Res <= '0';

        -- Load new PC values on each clock cycle
        for i in 0 to 7 loop
            tb_PC_Next <= std_logic_vector(to_unsigned(i, 3));
            wait for CLK_PERIOD;
        end loop;
```

```
        wait;
    end process;

end Behavioral;
```

Timing Diagram



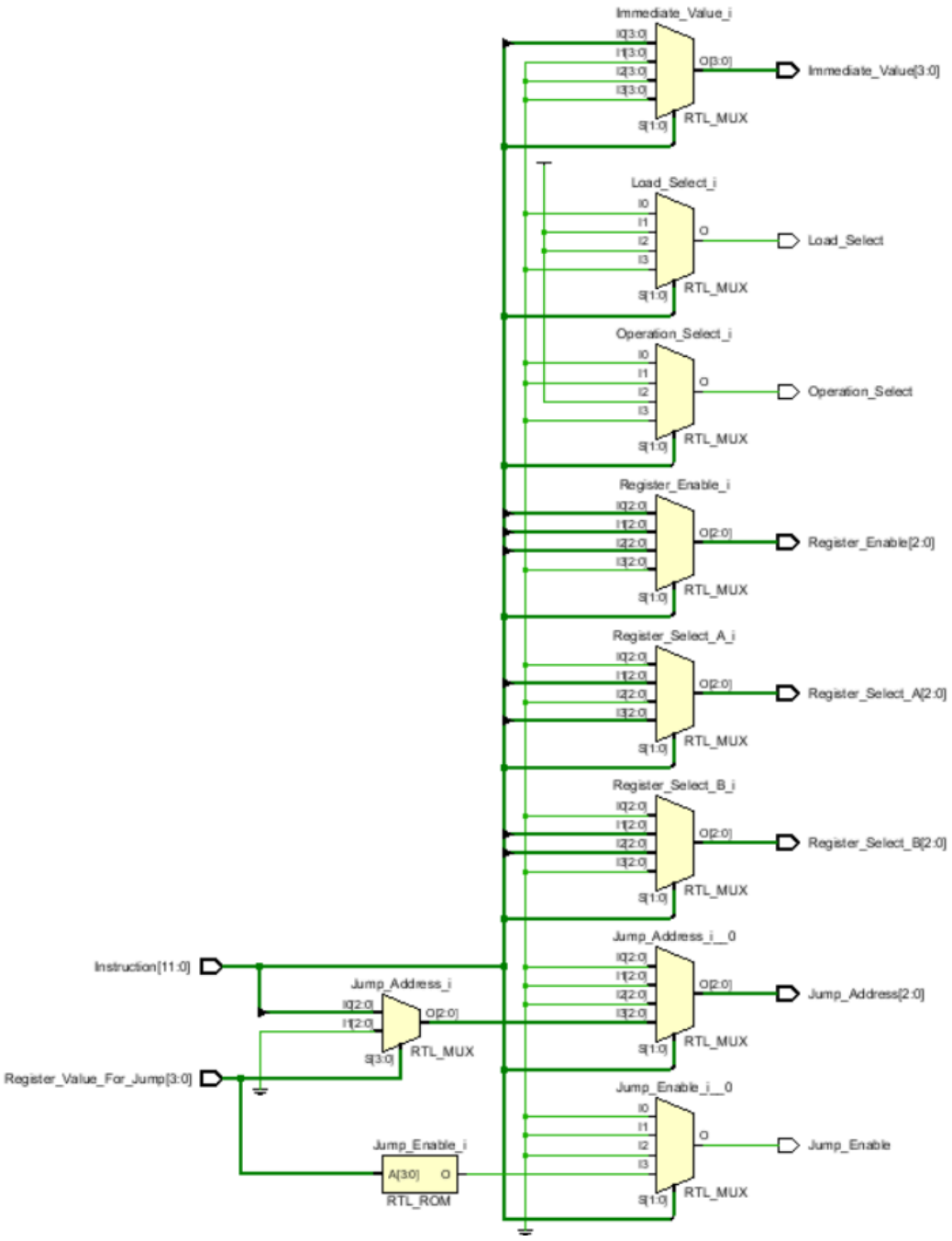
8.4 Instruction Decoder

Design Source Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.BusDefinitions.all;
use work.constants.all;

entity Instruction_Decoder is
    port (
        Instruction           : in  InstructionWord;  -- Instruction
        Register_Value_For_Jump : in  DataBus;        -- Register value to check for jump condition
        Register_Enable       : out RegisterSelect;   -- Register write enable
        Register_Select_A     : out RegisterSelect;   -- First operand register select
        Register_Select_B     : out RegisterSelect;   -- Second operand register select
        Operation_Select      : out STD_LOGIC;        -- '0' for ADD, '1' for SUB
        Immediate_Value       : out DataBus;          -- Immediate value for operations
        Jump_Enable           : out STD_LOGIC;        -- Jump control flag
        Jump_Address          : out ProgramCounter;   -- Address to jump to
        Load_Select           : out STD_LOGIC         -- Select between immediate or register data
    );
end entity Instruction_Decoder;

architecture Behavioral of Instruction_Decoder is
    signal Opcode : std_logic_vector(1 downto 0);     -- Instruction opcode
begin
    Opcode <= Instruction(11 downto 10);              -- Extract opcode from instruction

    decode: process(Opcode, Register_Value_For_Jump, Instruction)
    begin
        -- Default values to prevent latches
        Jump_Enable      <= '0';                      -- Disable jump by default
        Immediate_Value  <= "0000";                   -- Zero immediate value
        Load_Select      <= '0';                      -- Default to immediate load mode
        Register_Enable  <= "000";                    -- No register enabled by default
        Operation_Select <= '0';                      -- Default to ADD operation
        Register_Select_A <= "000";                   -- Default to register R0
        Register_Select_B <= "000";                   -- Default to register R0
        Jump_Address     <= "000";                    -- Default jump to address 0

        case Opcode is
            when MOVI_OP =>                                   -- Move Immediate
                Immediate_Value <= Instruction(3 downto 0);
```

```
            Load_Select     <= '0';                         -- Immediate load mode
            Register_Enable <= Instruction(9 downto 7);

        when ADD_OP =>                                      -- Add operation
            Register_Select_A <= Instruction(9 downto 7);
            Register_Select_B <= Instruction(6 downto 4);
            Operation_Select  <= '0';                       -- Addition
            Load_Select       <= '1';                       -- Register load mode
            Register_Enable   <= Instruction(9 downto 7);

        when NEG_OP =>                                      -- Negate operation
            Register_Select_A <= "000";                     -- Select R0 (zero)
            Register_Select_B <= Instruction(9 downto 7);
            Operation_Select  <= '1';                       -- Subtraction
            Load_Select       <= '1';                       -- Register load mode
            Register_Enable   <= Instruction(9 downto 7);

        when JZR_OP =>                                      -- Jump if Zero
            Register_Select_A <= Instruction(9 downto 7);
            Register_Enable   <= "000";                     -- No register writes

            if Register_Value_For_Jump = "0000" then
                Jump_Enable   <= '1';                       -- Enable jump
                Jump_Address  <= Instruction(2 downto 0);
            else
                Jump_Enable   <= '0';                        -- No jump
            end if;

        when others =>
            -- All outputs already have default values
    end case;
end process decode;
end architecture Behavioral;
```

Elaborated Design Schematic

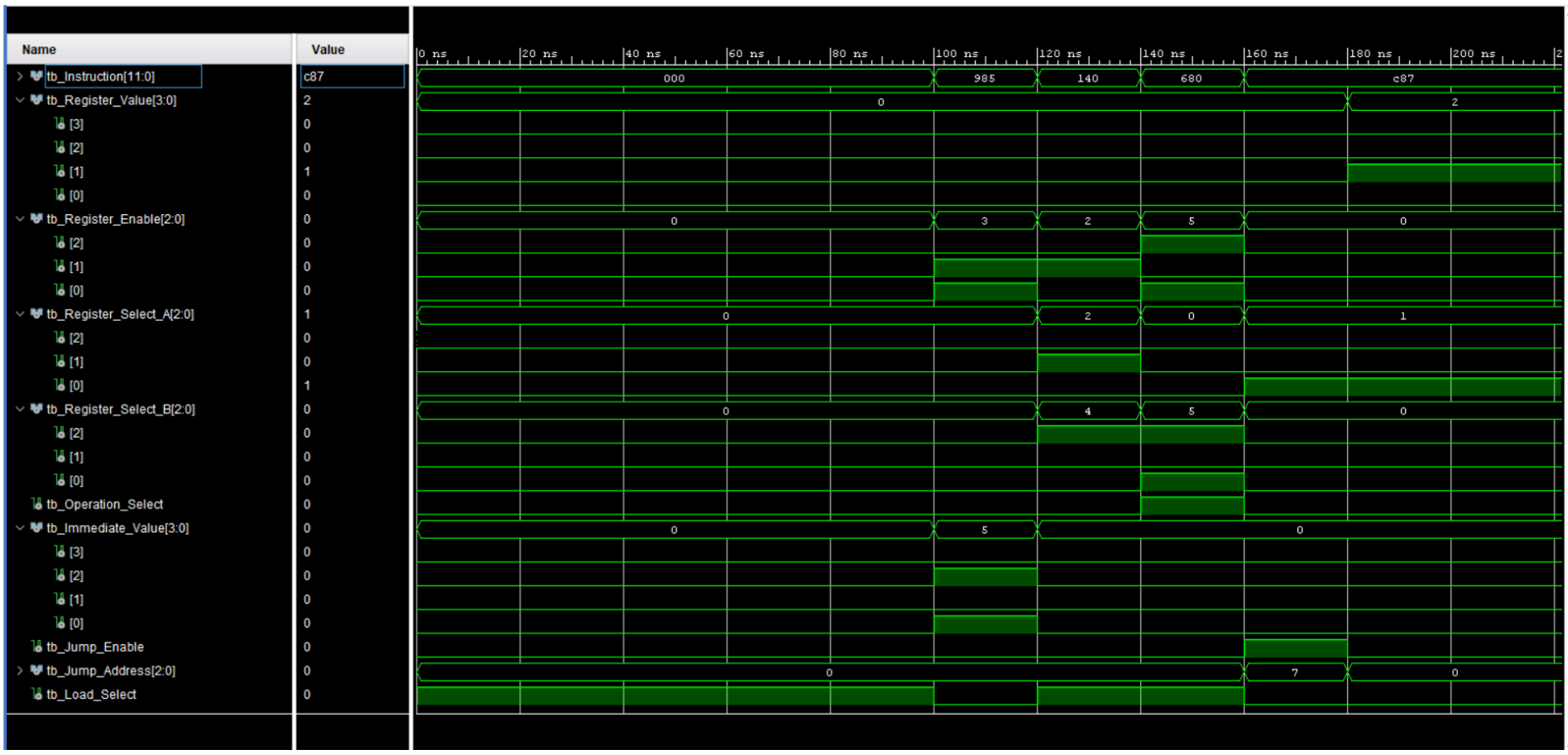


Behavioural Simulation Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.BusDefinitions.all;
use work.constants.all;

entity Instruction_Decoder_TB is
end Instruction_Decoder_TB;

architecture Behavioral of Instruction_Decoder_TB is
    -- Component declaration
    component Instruction_Decoder
        port (
            Instruction           : in  InstructionWord;
            Register_Value_For_Jump : in  DataBus;
            Register_Enable       : out RegisterSelect;
            Register_Select_A     : out RegisterSelect;
            Register_Select_B     : out RegisterSelect;
            Operation_Select      : out STD_LOGIC;
            Immediate_Value       : out DataBus;
            Jump_Enable           : out STD_LOGIC;
            Jump_Address          : out ProgramCounter;
            Load_Select           : out STD_LOGIC
        );
    end component;

    -- Inputs
    signal tb_Instruction : InstructionWord := (others => '0');
    signal tb_Register_Value : DataBus := (others => '0');

    -- Outputs
```

```
    signal tb_Register_Enable : RegisterSelect;
    signal tb_Register_Select_A : RegisterSelect;
    signal tb_Register_Select_B : RegisterSelect;
    signal tb_Operation_Select : STD_LOGIC;
    signal tb_Immediate_Value : DataBus;
    signal tb_Jump_Enable : STD_LOGIC;
    signal tb_Jump_Address : ProgramCounter;
    signal tb_Load_Select : STD_LOGIC;

begin
    -- Instantiate the Unit Under Test (UUT)
    uut: Instruction_Decoder
        port map (
            Instruction => tb_Instruction,
            Register_Value_For_Jump => tb_Register_Value,
            Register_Enable => tb_Register_Enable,
            Register_Select_A => tb_Register_Select_A,
            Register_Select_B => tb_Register_Select_B,
            Operation_Select => tb_Operation_Select,
            Immediate_Value => tb_Immediate_Value,
            Jump_Enable => tb_Jump_Enable,
            Jump_Address => tb_Jump_Address,
            Load_Select => tb_Load_Select
        );

    -- Stimulus process
    stim_proc: process
    begin
        -- Wait 100 ns for initialization
        wait for 100 ns;

        -- Test MOVI instruction
        -- MOVI R3, 5 (Move immediate value 5 to R3)
        tb_Instruction <= "100110000101";
        wait for 20 ns;

        -- Test ADD instruction
        -- ADD R2, R4 (Add R2 = R2 + R4)
        tb_Instruction <= "000101000000";
        wait for 20 ns;

        -- Test NEG instruction
        -- NEG R5 (R5 = -R5 = 0 - R5)
        tb_Instruction <= "011010000000";
        wait for 20 ns;

        -- Test JZR instruction with zero register
        -- JZR R1, 7 (Jump to address 7 if R1 = 0)
        tb_Instruction <= "110010000111";
        tb_Register_Value <= "0000";  -- R1 contains 0
        wait for 20 ns;

        -- Test JZR instruction with non-zero register
        -- JZR R1, 7 (Don't jump if R1 ? 0)
        tb_Instruction <= "110010000111";
        tb_Register_Value <= "0010";  -- R1 contains non-zero
        wait for 20 ns;

        wait;
    end process;
end Behavioral;
```

Timing Diagram



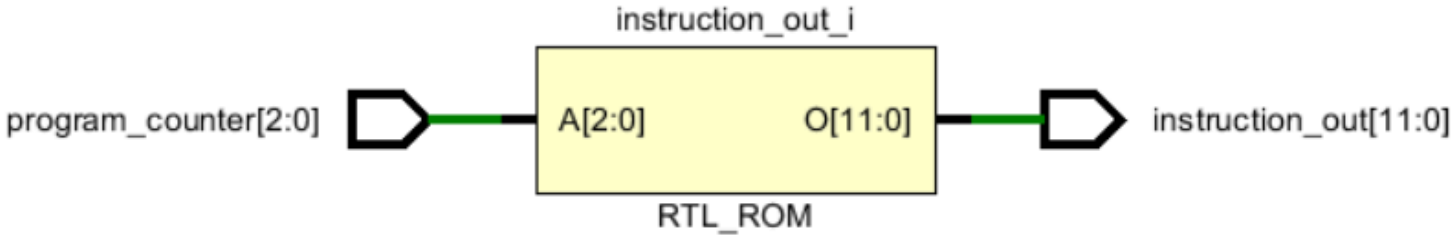
8.5 Program ROM

Design Source code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.BusDefinitions.all;

entity Program_ROM is
    port(
        program_counter : in ProgramCounter;   -- Address input from program counter
        instruction_out : out InstructionWord  -- 12-bit instruction output
    );
end Program_ROM;

architecture Behavioral of Program_ROM is
    -- ROM memory type stores 8 instructions of 12 bits each
    type instruction_memory_type is array (0 to 7) of std_logic_vector(11 downto 0);

    -- Program that adds numbers from 1 to 3
    signal program_instructions : instruction_memory_type := (
        "101110000011", -- MOVI R7,3  ; Load value 3 into R7 (loop counter)
        "100010000001", -- MOVI R1,1  ; Load immediate value 1 into R1
        "010010000000", -- NEG R1     ; Negate R1 to get -1
        "100100000011", -- MOVI R2,3  ; Load immediate value 3 into R2 (initial value)
        "000100010000", -- ADD R2,R1  ; Subtract 1 from R2 (using ADD with negative value)
        "001110100000", -- ADD R7,R2  ; Add R2 to accumulator in R7
        "110100000110", -- JZR R2,6   ; Jump to address 6 if R2 is zero
        "110000000100"  -- JZR R0,4   ; Jump to address 4 if R0 is zero (always true)
    );

begin
```

```
    -- Map ROM address (program counter) to instruction output
    instruction_out <= program_instructions(to_integer(unsigned(program_counter)));
end Behavioral;
```

Elaborated Design Schematic



Behavioural Simulation Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.BusDefinitions.all;
use work.Constants.all;

entity Program_ROM_TB is
end Program_ROM_TB;

architecture Behavioral of Program_ROM_TB is
    -- Component declaration
    component Program_ROM is
        port(
            program_counter : in ProgramCounter;
            instruction_out : out InstructionWord
        );
    end component;

    -- Testbench signals
    signal tb_pc : ProgramCounter := (others => '0');
    signal tb_instruction : InstructionWord;

begin
    -- Instantiate Unit Under Test (UUT)
    UUT: Program_ROM
        port map (
            program_counter => tb_pc,
            instruction_out => tb_instruction
        );

    -- Stimulus process
    stim_proc: process
    begin
        for i in 0 to 7 loop
            tb_pc <= std_logic_vector(to_unsigned(i, tb_pc'length));
            wait for 10 ns;
        end loop;
```

```
        wait;
    end process;

end Behavioral;
```

Timing Diagram

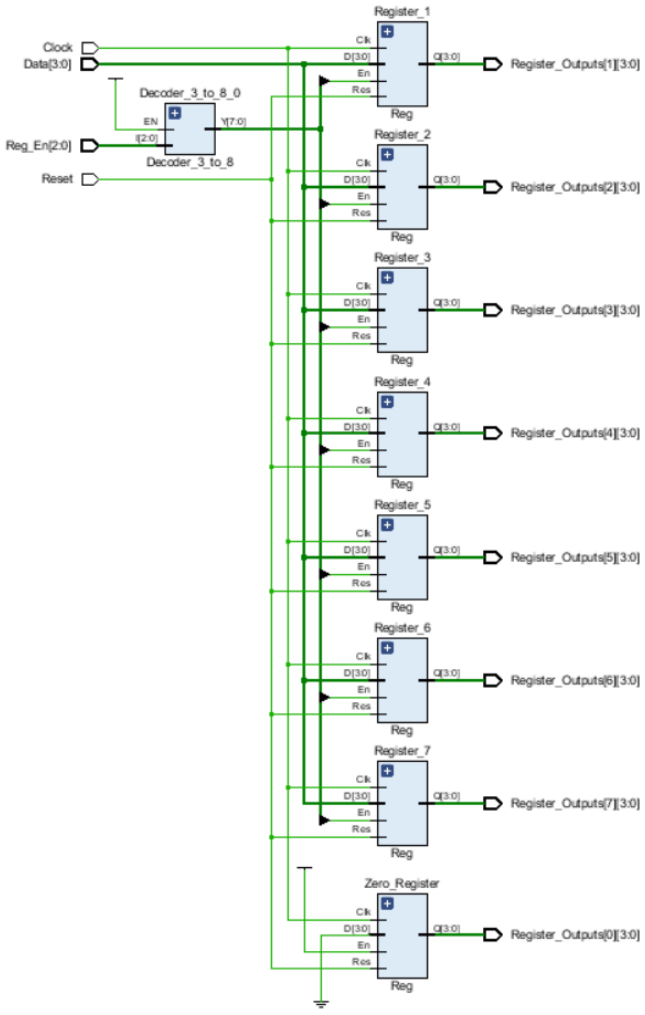
8.6 Register Bank

Design Source Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.BusDefinitions.all;

entity Register_Bank is
    Port (
        Data        : in  DataBus;  -- Data input to write to registers
        Reset       : in  STD_LOGIC;        -- Asynchronous reset signal
        Reg_En      : in  RegisterSelect;   -- Reg selection address
        Clock       : in  STD_LOGIC;         -- System clock signal
        Register_Outputs : out RegisterFile  -- Array of reg contents
    );
end Register_Bank;

architecture Behavioral of Register_Bank is
    -- Component declarations
    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 Reg
        Port (
            D   : in  STD_LOGIC_VECTOR(3 downto 0);
            Res : in  STD_LOGIC;
            En  : in  STD_LOGIC;
            Clk : in  STD_LOGIC;
            Q   : out STD_LOGIC_VECTOR(3 downto 0)
        );
    end component;

    -- Internal signals
    signal Reg_Sel : STD_LOGIC_VECTOR(7 downto 0);  -- One-hot register selection signals

begin
    -- Address decoder - converts 3-bit address to one-hot selection
    Decoder_3_to_8_0 : Decoder_3_to_8
        port map(
            I  => Reg_En,
            EN => '1',        -- Always enabled
            Y  => Reg_Sel
        );

    -- Register 0 - Always contains zero
    Zero_Register : Reg
        port map(
            D   => "0000",    -- Hardwired to zero
            Res => Reset,
            En  => '1',       -- Always enabled
            Clk => Clock,
            Q   => Register_Outputs(0)
        );

    -- Registers 1-7 - General purpose registers
Register_1 : Reg
        port map(
            D   => Data,         -- Data input
            Res => Reset,        -- Reset signal
            En  => Reg_Sel(1),   -- Enable for Register 1
            Clk => Clock,        -- Clock signal
            Q   => Register_Outputs(1)
```

```
    );

-- Register 2
Register_2 : Reg
    port map(
        D   => Data,         -- Data input
        Res => Reset,        -- Reset signal
        En  => Reg_Sel(2),   -- Enable for Register 2
        Clk => Clock,        -- Clock signal
        Q   => Register_Outputs(2)
    );

-- Register 3
Register_3 : Reg
    port map(
        D   => Data,         -- Data input
        Res => Reset,        -- Reset signal
        En  => Reg_Sel(3),   -- Enable for Register 3
        Clk => Clock,        -- Clock signal
        Q   => Register_Outputs(3)
    );

-- Register 4
Register_4 : Reg
    port map(
        D   => Data,         -- Data input
        Res => Reset,        -- Reset signal
        En  => Reg_Sel(4),   -- Enable for Register 4
        Clk => Clock,        -- Clock signal
        Q   => Register_Outputs(4)
    );

-- Register 5
Register_5 : Reg
    port map(
        D   => Data,         -- Data input
        Res => Reset,        -- Reset signal
        En  => Reg_Sel(5),   -- Enable for Register 5
        Clk => Clock,        -- Clock signal
        Q   => Register_Outputs(5)
    );

-- Register 6
Register_6 : Reg
    port map(
        D   => Data,         -- Data input
        Res => Reset,        -- Reset signal
        En  => Reg_Sel(6),   -- Enable for Register 6
        Clk => Clock,        -- Clock signal
        Q   => Register_Outputs(6)
    );

-- Register 7
Register_7 : Reg
    port map(
        D   => Data,         -- Data input
        Res => Reset,        -- Reset signal
        En  => Reg_Sel(7),   -- Enable for Register 7
        Clk => Clock,        -- Clock signal
        Q   => Register_Outputs(7)
    );

end Behavioral;
```

Elaborated Design Schematic



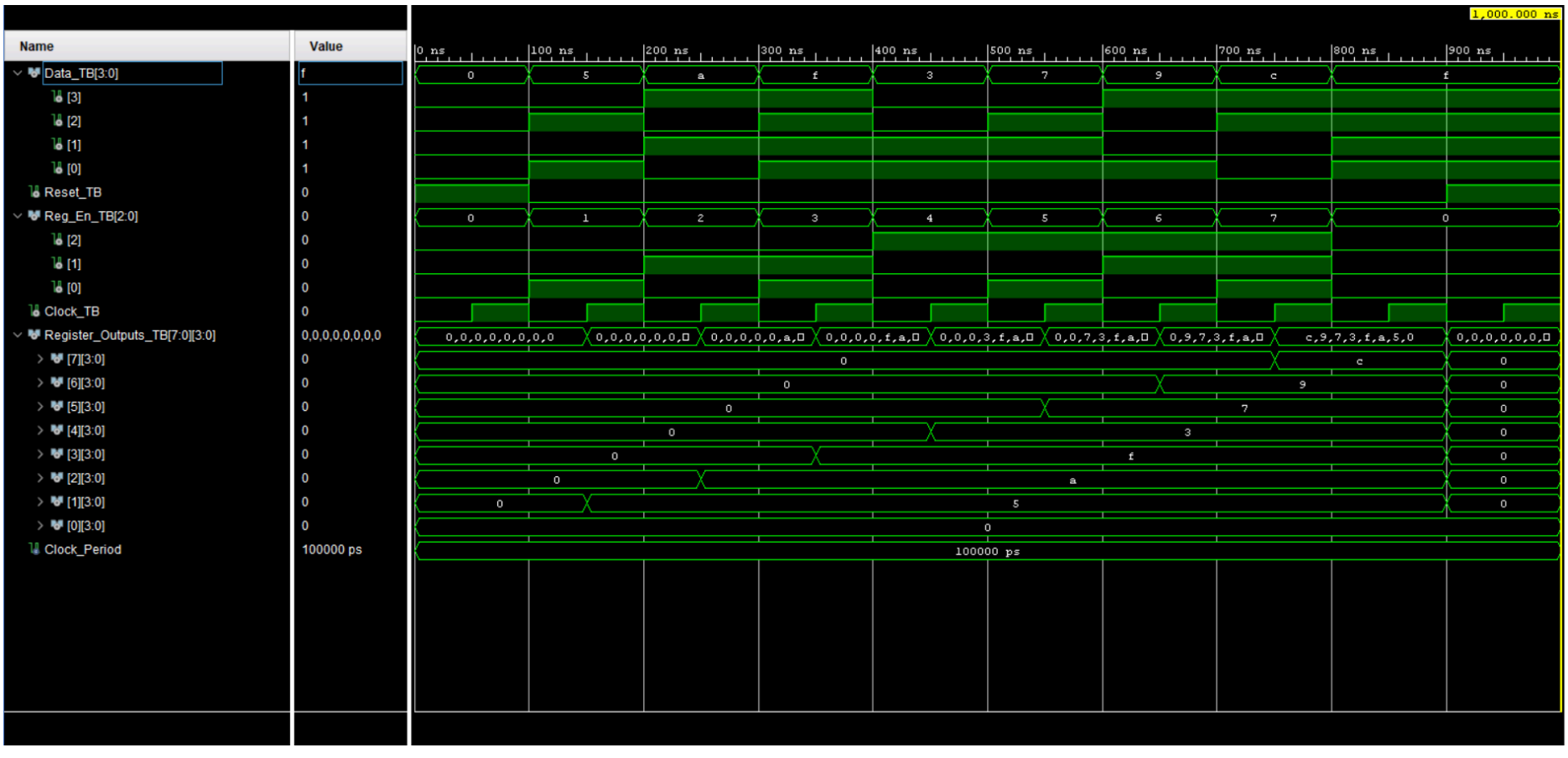
Behavioural Simulation Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.BusDefinitions.all;

entity Register_Bank_TB is
end Register_Bank_TB;

architecture Behavioral of Register_Bank_TB is
    -- Component Declaration
    component Register_Bank
        Port (
            Data             : in  DataBus;
            Reset            : in  STD_LOGIC;
            Reg_En           : in  RegisterSelect;
            Clock            : in  STD_LOGIC;
            Register_Outputs : out RegisterFile
        );
    end component;

    -- Test signals
    signal Data_TB           : DataBus := "0000";
    signal Reset_TB          : STD_LOGIC := '0';
    signal Reg_En_TB         : RegisterSelect := "000";
    signal Clock_TB          : STD_LOGIC := '0';
    signal Register_Outputs_TB : RegisterFile;

    -- Clock period
    constant Clock_Period : time := 100 ns;

begin
    -- Instantiate Register Bank
    UUT: Register_Bank port map (
        Data             => Data_TB,
```

```
        Reset            => Reset_TB,
        Reg_En           => Reg_En_TB,
        Clock            => Clock_TB,
        Register_Outputs => Register_Outputs_TB
    );

    -- Clock generation - run for longer to accommodate all test cases
    Clock_process: process
    begin
        while now < 1200 ns loop  -- Extended simulation time
            Clock_TB <= '0';
            wait for Clock_Period/2;
            Clock_TB <= '1';
            wait for Clock_Period/2;
        end loop;
        wait; -- Stop the clock after simulation time
    end process;

    -- Test sequence
    Stim_process: process
    begin
        -- Initial reset
        Reset_TB <= '1';
        wait for 100 ns;  -- Full clock cycle for reset
        Reset_TB <= '0';

        -- Change values after falling edge, capture on rising edge
        wait until falling_edge(Clock_TB);

        -- Test Register 1
        Reg_En_TB <= "001";  -- Select R1
        Data_TB <= "0101";   -- Value 5
        wait for Clock_Period;  -- Wait for full clock cycle

        -- Test Register 2
        wait until falling_edge(Clock_TB);
        Reg_En_TB <= "010";  -- Select R2
        Data_TB <= "1010";   -- Value 10
        wait for Clock_Period;

        -- Test Register 3
        wait until falling_edge(Clock_TB);
        Reg_En_TB <= "011";  -- Select R3
        Data_TB <= "1111";   -- Value 15
        wait for Clock_Period;

        -- Test Register 4
        wait until falling_edge(Clock_TB);
        Reg_En_TB <= "100";  -- Select R4
        Data_TB <= "0011";   -- Value 3
        wait for Clock_Period;

        -- Test Register 5
        wait until falling_edge(Clock_TB);
        Reg_En_TB <= "101";  -- Select R5
        Data_TB <= "0111";   -- Value 7
        wait for Clock_Period;

        -- Test Register 6
        wait until falling_edge(Clock_TB);
        Reg_En_TB <= "110";  -- Select R6
        Data_TB <= "1001";   -- Value 9
        wait for Clock_Period;

        -- Test Register 7
        wait until falling_edge(Clock_TB);
        Reg_En_TB <= "111";  -- Select R7
        Data_TB <= "1100";   -- Value 12
        wait for Clock_Period;
```

```
        -- Try to write to Register 0 (should remain 0)
        wait until falling_edge(Clock_TB);
        Reg_En_TB <= "000";  -- Select R0
        Data_TB <= "1111";   -- Try to write 15
        wait for Clock_Period;

        -- Test reset after writing
        wait until falling_edge(Clock_TB);
        Reset_TB <= '1';
        wait for Clock_Period;
        Reset_TB <= '0';

        wait;
    end process;
end Behavioral;
```

Timing Diagram

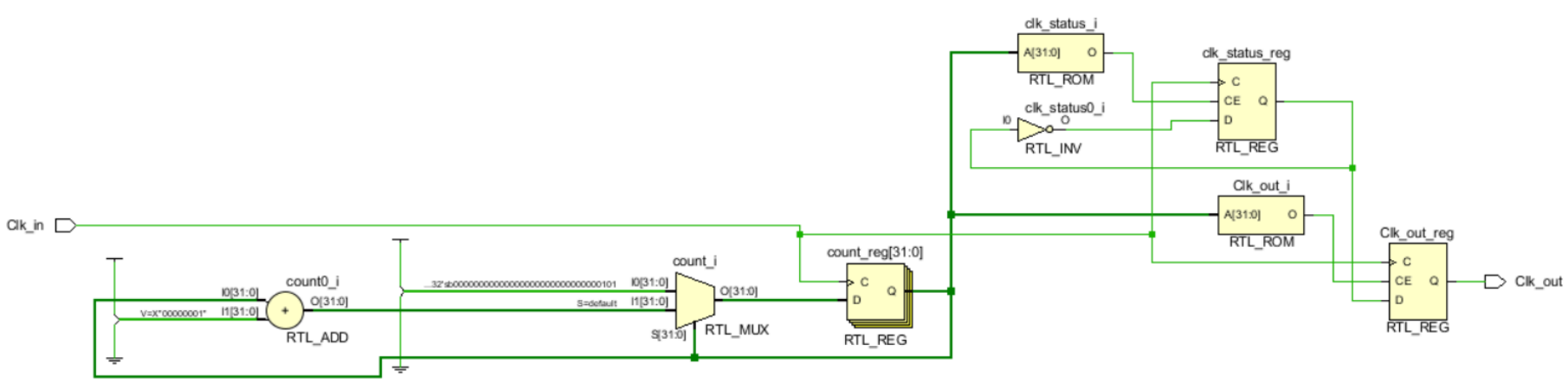


8.7 Slow Clock

Design Source 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
    process (Clk_in) begin
        if rising_edge(Clk_in) then
            count <= count + 1;                -- Increment counter
            if(count = 50000000) then
                clk_status <= not clk_status; -- Invert clock status
                Clk_out <= clk_status;
                count <= 1;                    -- Reset counter
            end if;
        end if;
    end process;

end Behavioral;
```

Elaborated Design Schematic

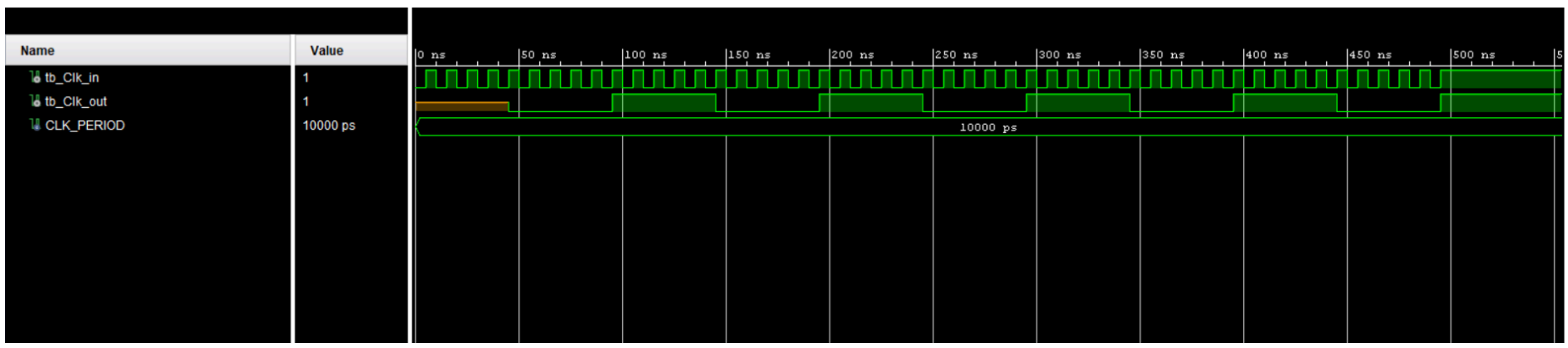


Behavioural Simulation Code

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

entity Slow_Clk_TB is
end Slow_Clk_TB;

architecture Behavioral of Slow_Clk_TB is

    -- Component under test
    component Slow_Clk is
```

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

    -- Testbench signals
    signal tb_Clk_in  : STD_LOGIC := '0';
    signal tb_Clk_out : STD_LOGIC;

    -- Clock period
    constant CLK_PERIOD : time := 10 ns;

begin

    -- Instantiate the Unit Under Test (UUT)
    UUT: Slow_Clk
        port map (
            Clk_in  => tb_Clk_in,
            Clk_out => tb_Clk_out
        );

    -- Generate input clock (50 MHz => 10 ns period)
    clk_process: process
    begin
        while now < 500 ns loop
            tb_Clk_in <= '0';
            wait for CLK_PERIOD / 2;
            tb_Clk_in <= '1';
            wait for CLK_PERIOD / 2;
        end loop;
        wait;
    end process;

end Behavioral;
```

Timing Diagram



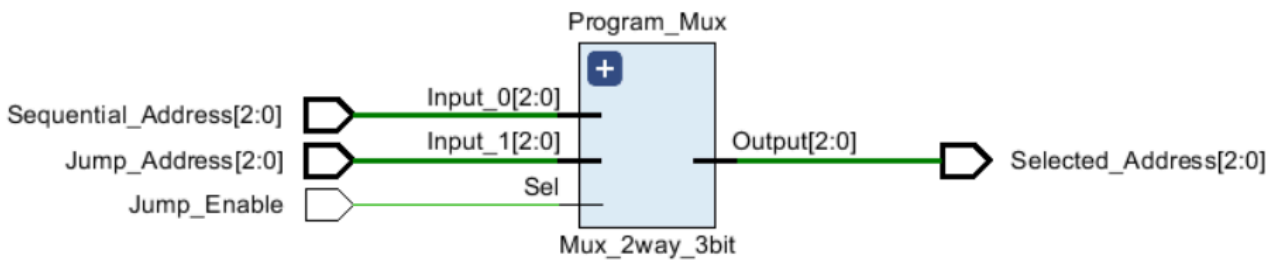
8.8 Address Selector

Design Source Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.BusDefinitions.ProgramCounter;

entity Address_Selector is
    port (
        Sequential_Address : in  ProgramCounter; -- PC+1 address for sequential execution
        Jump_Address       : in  ProgramCounter; -- Target address when jump occurs
        Jump_Enable        : in  std_logic;      -- Jump control signal (1=Jump, 0=Continue)
        Selected_Address   : out ProgramCounter  -- Address selected for next execution
```

```
    );
end entity Address_Selector;

architecture Behavioral of Address_Selector is
    component Mux_2way_3bit is
        port (
            Input_0 : in  STD_LOGIC_VECTOR(2 downto 0); -- First 3-bit input
            Input_1 : in  STD_LOGIC_VECTOR(2 downto 0); -- Second 3-bit input
            Sel     : in  STD_LOGIC;                    -- Selection control (1 bit for 2 inputs)
            Output  : out STD_LOGIC_VECTOR(2 downto 0)  -- 3-bit output
        );
    end component Mux_2way_3bit;
begin
    Program_Mux : Mux_2way_3bit
        port map (
            Input_0 => Sequential_Address,
            Input_1 => Jump_Address,
            Sel     => Jump_Enable,
            Output  => Selected_Address
        );
end architecture Behavioral;
```

Elaborated Design Schematic

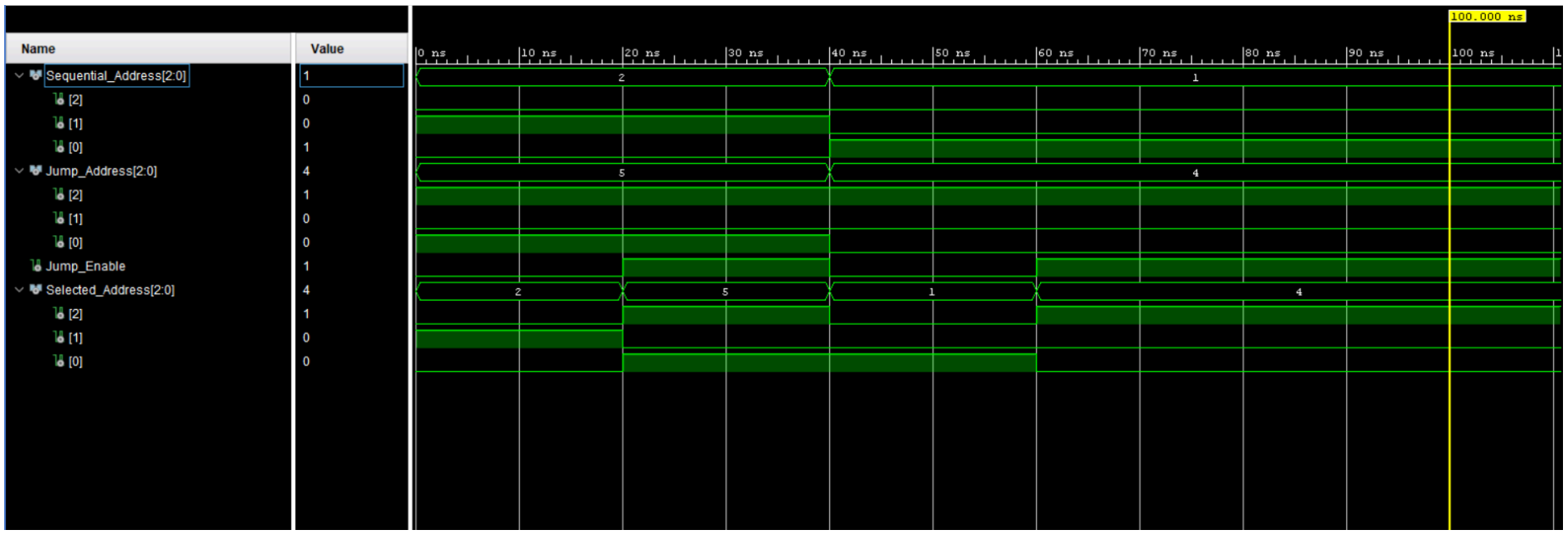


Behavioural Simulation Code

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

-- You must compile/work with BusDefinitions before this TB if it's referenced

entity Address_Selector_TB is
end Address_Selector_TB;

architecture Behavioral of Address_Selector_TB is

    -- Component under test
    component Address_Selector is
        port (
            Sequential_Address : in  STD_LOGIC_VECTOR(2 downto 0);
            Jump_Address       : in  STD_LOGIC_VECTOR(2 downto 0);
            Jump_Enable        : in  std_logic;
            Selected_Address   : out STD_LOGIC_VECTOR(2 downto 0)
        );
    end component;

    -- Signals for testing
    signal Sequential_Address : STD_LOGIC_VECTOR(2 downto 0) := "000";
```

```
    signal Jump_Address       : STD_LOGIC_VECTOR(2 downto 0) := "111";
    signal Jump_Enable        : STD_LOGIC := '0';
    signal Selected_Address   : STD_LOGIC_VECTOR(2 downto 0);

begin

    -- Instantiate UUT
    UUT: Address_Selector
        port map (
            Sequential_Address => Sequential_Address,
            Jump_Address       => Jump_Address,
            Jump_Enable        => Jump_Enable,
            Selected_Address   => Selected_Address
        );

    -- Stimulus process
    stimulus: process
    begin
        -- Case 1: Jump_Enable = 0 (sequential address selected)
        Sequential_Address <= "010";
        Jump_Address       <= "101";
        Jump_Enable        <= '0';
        wait for 20 ns;

        -- Case 2: Jump_Enable = 1 (jump address selected)
        Jump_Enable <= '1';
        wait for 20 ns;

        -- Case 3: Change addresses again
        Sequential_Address <= "001";
        Jump_Address       <= "100";
        Jump_Enable        <= '0';
        wait for 20 ns;

        Jump_Enable <= '1';
        wait for 20 ns;

        wait;
    end process;

end Behavioral;
```

Timing Diagram

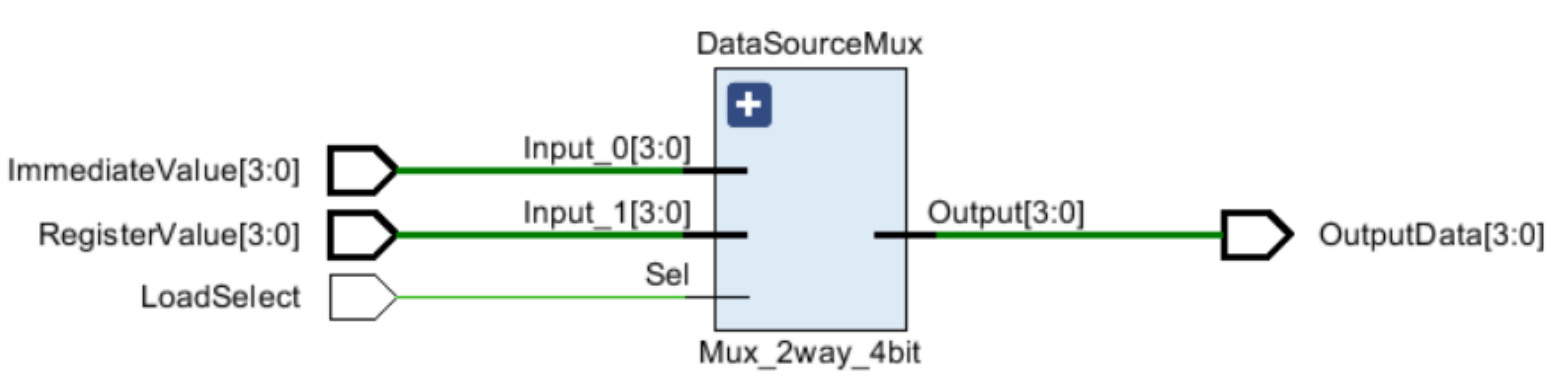


8.9 Load Selector

Design Source Code

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.BusDefinitions.all;

entity Load_Selector is
    Port (
        RegisterValue  : in  DataBus;          -- Input from register (for ADD/NEG)
        ImmediateValue : in  DataBus;          -- Immediate value (for MOVI)
        LoadSelect     : in  STD_LOGIC;        -- Selection control from instruction decoder
        OutputData     : out DataBus           -- Selected output to ALU
    );
end Load_Selector;

architecture Behavioral of Load_Selector is
    -- Component declaration for 2-way 4-bit MUX
    component Mux_2way_4bit
        Port (
            Input_0 : in  STD_LOGIC_VECTOR(3 downto 0);
            Input_1 : in  STD_LOGIC_VECTOR(3 downto 0);
            Sel     : in  STD_LOGIC;
            Output  : out STD_LOGIC_VECTOR(3 downto 0)
        );
    end component;

begin
    -- Instantiate the 2-way 4-bit MUX
    DataSourceMux: Mux_2way_4bit port map (
        Input_1 => RegisterValue,     -- When LoadSelect=1, use register value
        Input_0 => ImmediateValue,    -- When LoadSelect=0, use immediate value
        Sel     => LoadSelect,
        Output  => OutputData
    );

end Behavioral;
```

Elaborated Design Schematic

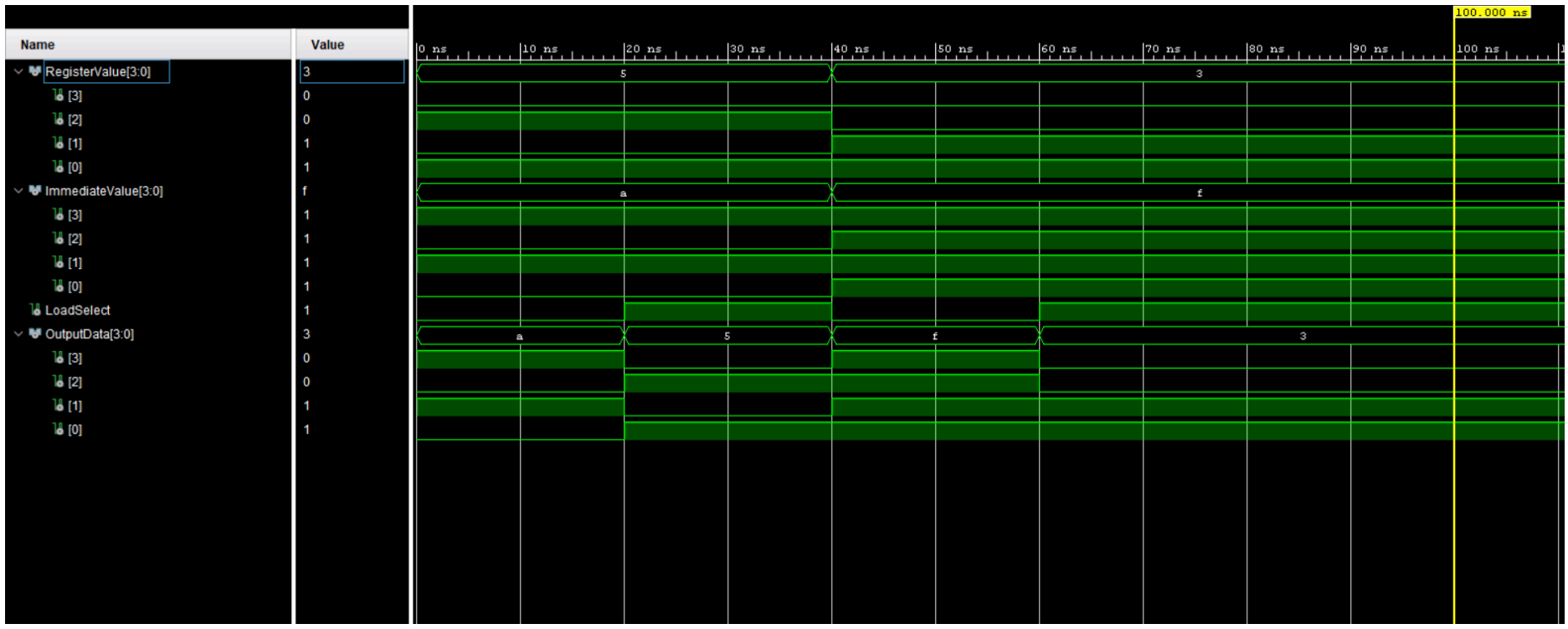


Behavioural Simulation Code

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

entity Load_Selector_TB is
end Load_Selector_TB;

architecture Behavioral of Load_Selector_TB is
```

```
    -- Component under test
    component Load_Selector is
        Port (
            RegisterValue  : in  STD_LOGIC_VECTOR(3 downto 0);
            ImmediateValue : in  STD_LOGIC_VECTOR(3 downto 0);
            LoadSelect     : in  STD_LOGIC;
            OutputData     : out STD_LOGIC_VECTOR(3 downto 0)
        );
    end component;

    -- Signals for the testbench
    signal RegisterValue  : STD_LOGIC_VECTOR(3 downto 0) := "0000";
    signal ImmediateValue : STD_LOGIC_VECTOR(3 downto 0) := "0000";
    signal LoadSelect     : STD_LOGIC := '0';
    signal OutputData     : STD_LOGIC_VECTOR(3 downto 0);

begin

    -- Instantiate the Unit Under Test
    UUT: Load_Selector
        port map (
            RegisterValue  => RegisterValue,
            ImmediateValue => ImmediateValue,
            LoadSelect     => LoadSelect,
            OutputData     => OutputData
        );

    -- Stimulus process
    stimulus: process
    begin
        -- Test case 1: LoadSelect = 0 → Output = ImmediateValue
        ImmediateValue <= "1010";  -- 10
        RegisterValue  <= "0101";  -- 5
        LoadSelect     <= '0';
        wait for 20 ns;

        -- Test case 2: LoadSelect = 1 → Output = RegisterValue
        LoadSelect <= '1';
        wait for 20 ns;

        -- Test case 3: Change values
        ImmediateValue <= "1111";  -- 15
        RegisterValue  <= "0011";  -- 3
        LoadSelect     <= '0';
        wait for 20 ns;

        LoadSelect <= '1';
        wait for 20 ns;

        wait;
    end process;

end Behavioral;
```

Timing Diagram

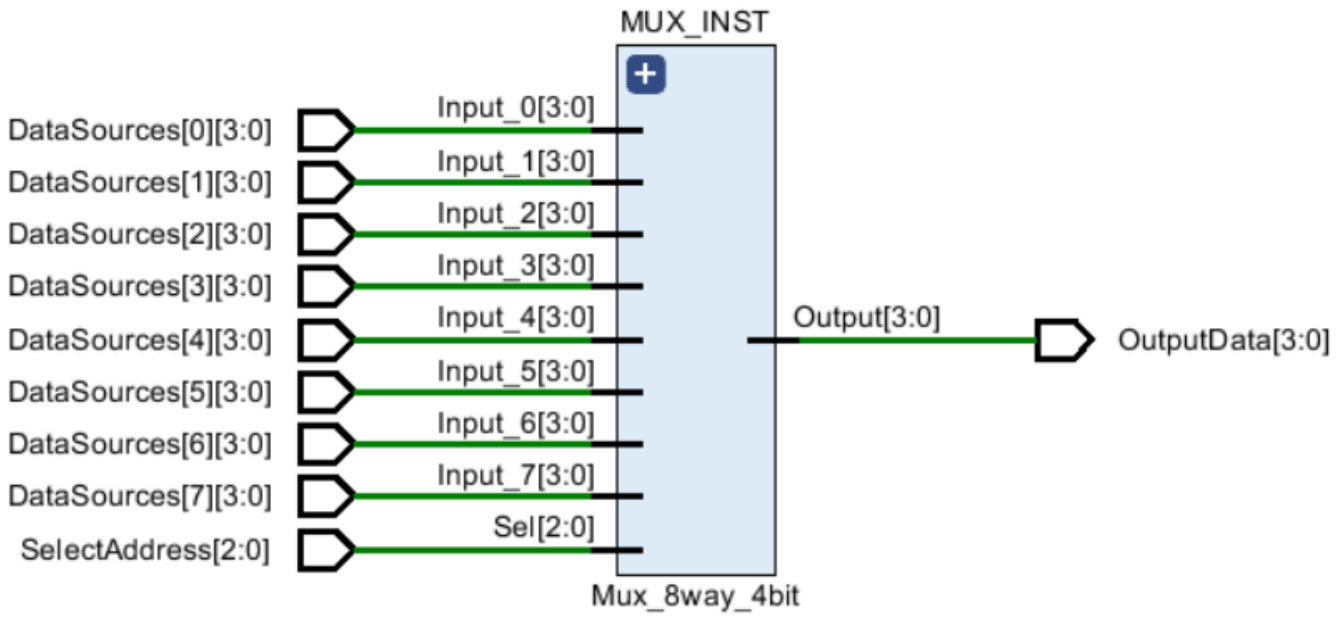


8.10 Register Data Multiplexer

Design Source Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.BusDefinitions.all;

entity RegisterData_Multiplexer is
    Port (
        DataSources   : in  RegisterFile;
        SelectAddress : in  RegisterSelect;
        OutputData    : out DataBus
    );
end RegisterData_Multiplexer;

architecture Behavioral of RegisterData_Multiplexer is
    component Mux_8way_4bit
        port(
            Input_0    : in  STD_LOGIC_VECTOR(3 downto 0);
            Input_1    : in  STD_LOGIC_VECTOR(3 downto 0);
            Input_2    : in  STD_LOGIC_VECTOR(3 downto 0);
            Input_3    : in  STD_LOGIC_VECTOR(3 downto 0);
            Input_4    : in  STD_LOGIC_VECTOR(3 downto 0);
            Input_5    : in  STD_LOGIC_VECTOR(3 downto 0);
            Input_6    : in  STD_LOGIC_VECTOR(3 downto 0);
            Input_7    : in  STD_LOGIC_VECTOR(3 downto 0);
            Sel        : in  STD_LOGIC_VECTOR(2 downto 0);
            Output     : out STD_LOGIC_VECTOR(3 downto 0)
        );
    end component;
begin
    -- Instantiate the multiplexer component and connect the ports
    MUX_INST: Mux_8way_4bit port map(
        Input_0 => DataSources(0),
        Input_1 => DataSources(1),
        Input_2 => DataSources(2),
        Input_3 => DataSources(3),
        Input_4 => DataSources(4),
        Input_5 => DataSources(5),
        Input_6 => DataSources(6),
        Input_7 => DataSources(7),
        Sel     => SelectAddress,
        Output  => OutputData
    );
end Behavioral;
```

Elaborated Design Schematic

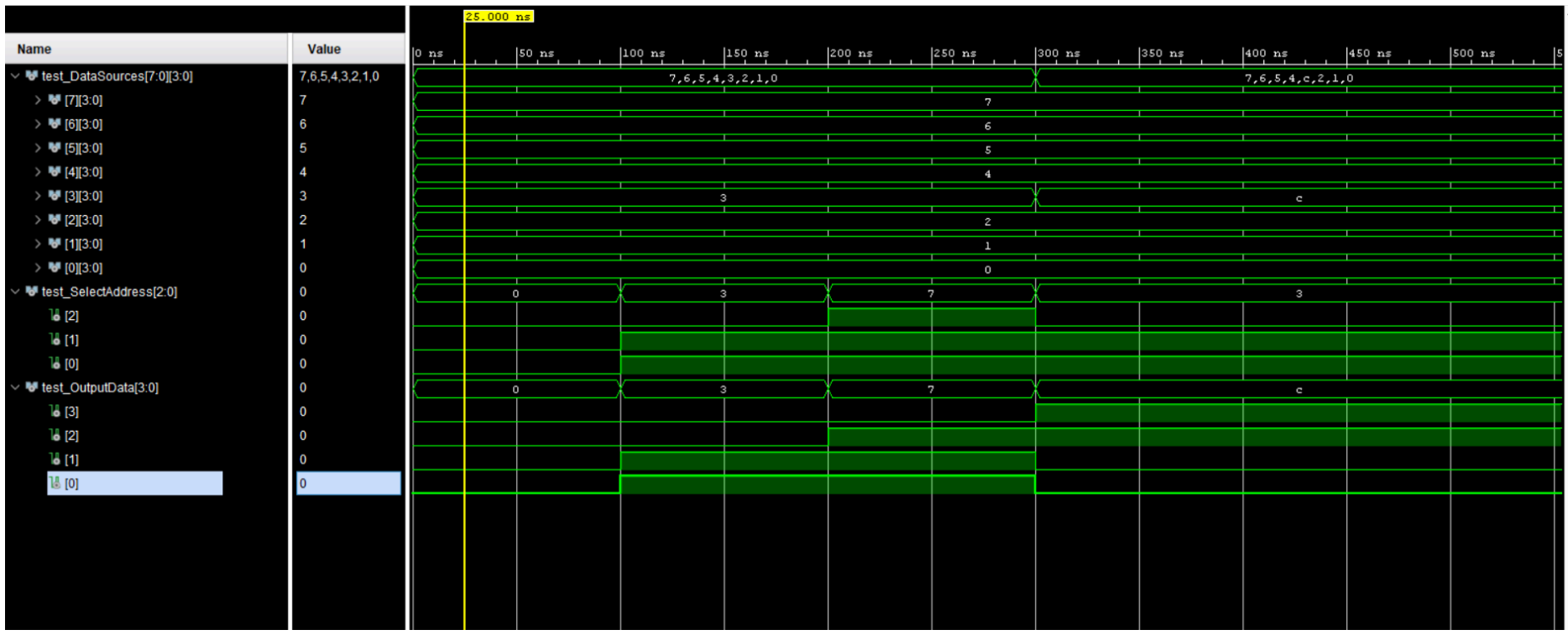


Behavioural Simulation Code

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

-- Use the same definitions as in your main design
use work.BusDefinitions.all;

entity RegisterData_Multiplexer_TB is
end RegisterData_Multiplexer_TB;

architecture Behavioral of RegisterData_Multiplexer_TB is

    -- Component under test
    component RegisterData_Multiplexer
        Port (
            DataSources   : in  RegisterFile;
            SelectAddress : in  RegisterSelect;
            OutputData    : out DataBus
        );
    end component;

    -- Testbench signals
    signal DataSources   : RegisterFile := (
        0 => "0001",  -- R0
        1 => "0010",  -- R1
        2 => "0011",  -- R2
        3 => "0100",  -- R3
        4 => "0101",  -- R4
        5 => "0110",  -- R5
        6 => "0111",  -- R6
        7 => "1000"   -- R7
    );
    signal SelectAddress : RegisterSelect := "000";
    signal OutputData    : DataBus;

begin

    -- Instantiate the DUT
    DUT: RegisterData_Multiplexer
        port map (
            DataSources   => DataSources,
            SelectAddress => SelectAddress,
            OutputData    => OutputData
        );

    -- Stimulus process
    stim_proc: process
```

```
    begin
        for i in 0 to 7 loop
            SelectAddress <= std_logic_vector(to_unsigned(i, 3));
            wait for 20 ns;
        end loop;
        wait;
    end process;

end Behavioral;
```

Timing Diagram

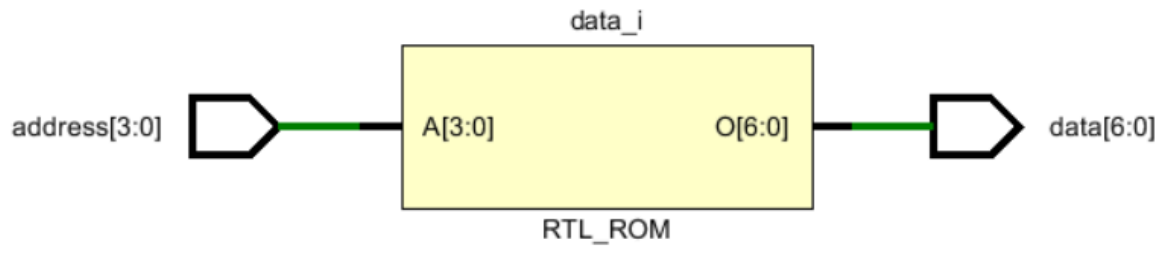


8.11 Lookup Table - 7 Segment display

Design Source Code

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

entity LUT_16_7 is
    Port ( address : in DataBus;
           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;
```

Elaborated Design Schematic

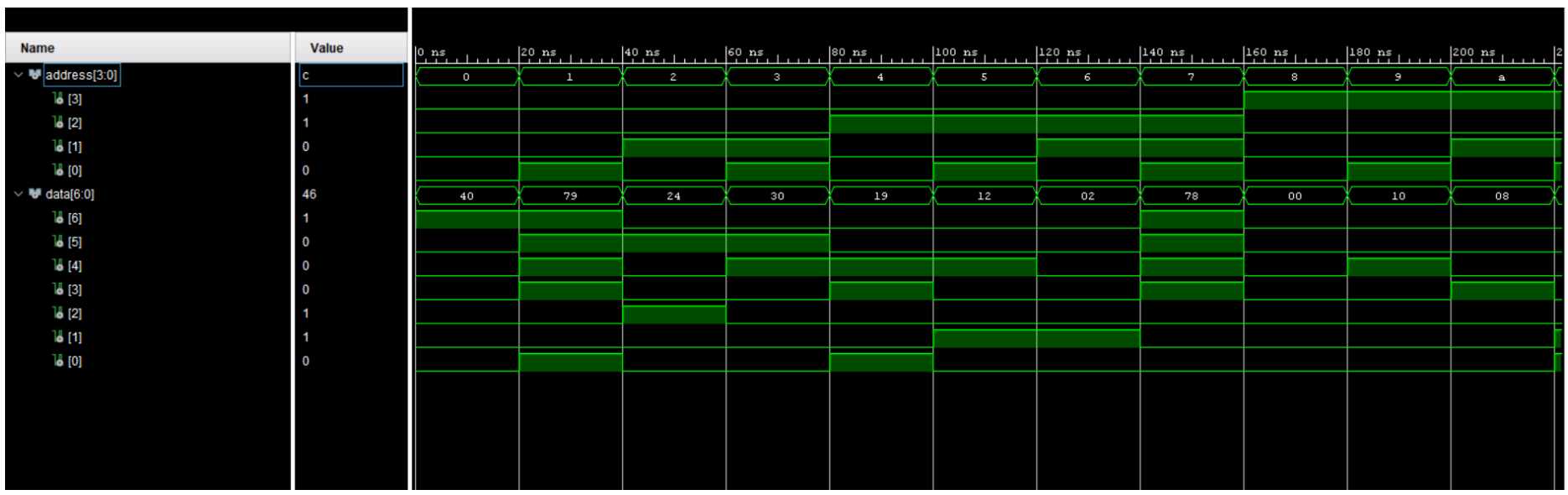


Behavioural Simulation Code

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

entity LUT_16_7_TB is
end LUT_16_7_TB;
```

```vhdl
architecture Behavioral of LUT_16_7_TB is

    -- Component declaration
    component LUT_16_7 is
        Port (
            address : in  STD_LOGIC_VECTOR(3 downto 0);
            data    : out STD_LOGIC_VECTOR(6 downto 0)
        );
    end component;

    -- Signals for stimulus and observation
    signal address : STD_LOGIC_VECTOR(3 downto 0) := "0000";
    signal data    : STD_LOGIC_VECTOR(6 downto 0);

begin

    -- Instantiate the unit under test
    UUT: LUT_16_7
        port map (
            address => address,
            data    => data
        );

    -- Test process
    stimulus: process
    begin
        for i in 0 to 15 loop
            address <= std_logic_vector(to_unsigned(i, 4));
            wait for 20 ns;
        end loop;

        wait;
    end process;

end Behavioral;
```

Timing Diagram



9. Core Components - Extended Nano Processor

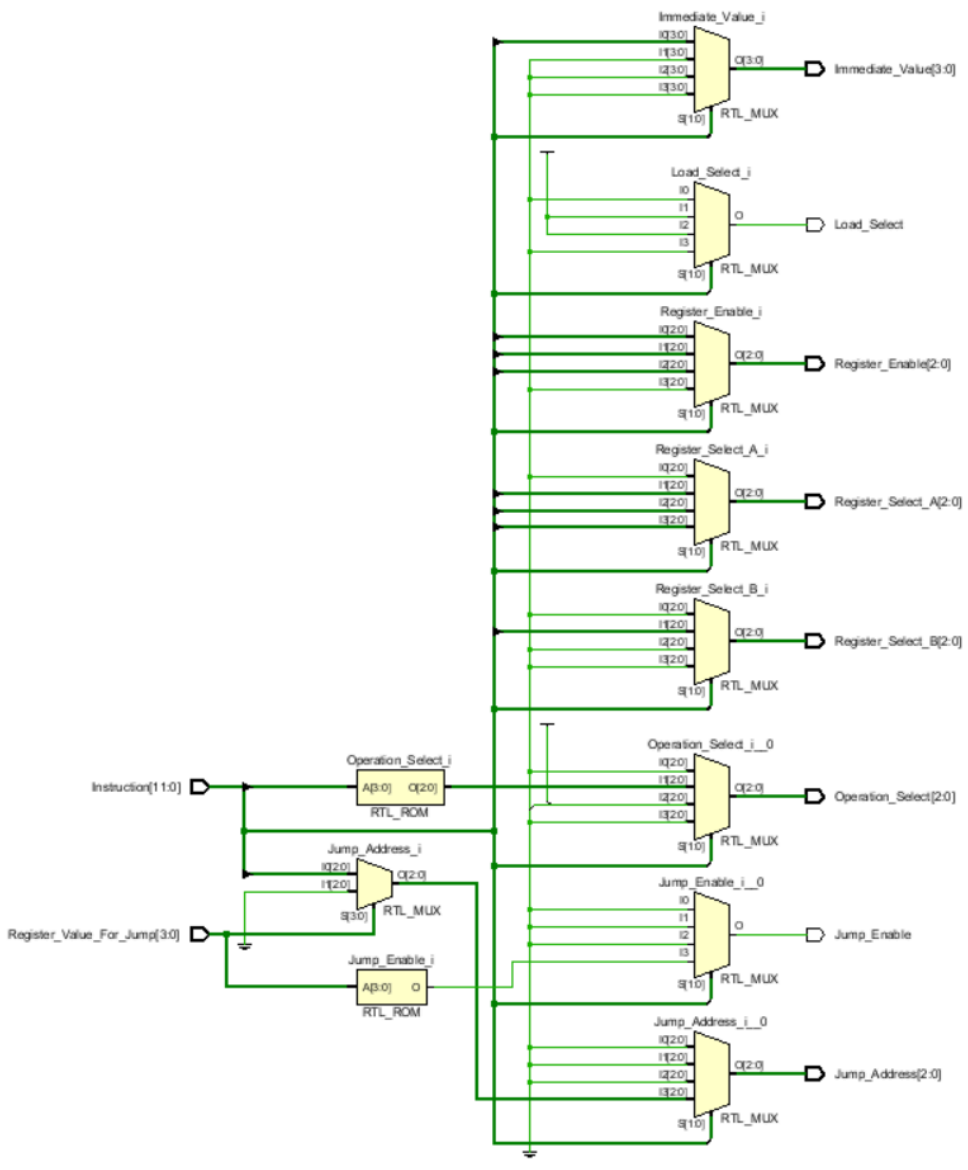
9.1 Extended Instruction Decoder

Design Source Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;use work.BusDefinitions.all;
use work.constants.all;

entity Instruction_Decoder is
    port (
        Instruction           : in  InstructionWord;  -- Instruction
        Register_Value_For_Jump : in  DataBus;        -- Register value to check for jump condition
        Register_Enable       : out RegisterSelect;   -- Register write enable
        Register_Select_A     : out RegisterSelect;   -- First operand register select
        Register_Select_B     : out RegisterSelect;   -- Second operand register select
        Operation_Select      : out OperationSelection;  -- Operation selector (3 bits)
        Immediate_Value       : out DataBus;          -- Immediate value for operations
        Jump_Enable           : out STD_LOGIC;        -- Jump control flag
        Jump_Address          : out ProgramCounter;   -- Address to jump to
        Load_Select           : out STD_LOGIC         -- Select between immediate or register data
    );
end entity Instruction_Decoder;

architecture Behavioral of Instruction_Decoder is
    signal Opcode    : std_logic_vector(1 downto 0);  -- Instruction opcode
    signal SubOpcode : std_logic_vector(3 downto 0);  -- Sub-opcode for ALU operations
begin
    Opcode    <= Instruction(11 downto 10);           -- Extract opcode from instruction
    SubOpcode <= Instruction(3 downto 0);             -- Extract sub-opcode (4 bits)

    decode: process(Opcode, Register_Value_For_Jump, Instruction, SubOpcode)
    begin
        -- Default values to prevent latches
        Jump_Enable      <= '0';                      -- Disable jump by default
        Immediate_Value  <= "0000";                   -- Zero immediate value
        Load_Select      <= '0';                      -- Default to immediate load mode
        Register_Enable  <= "000";                    -- No register enabled by default
        Operation_Select <= "000";                    -- Default to ADD operation
        Register_Select_A <= "000";                   -- Default to register R0
        Register_Select_B <= "000";                   -- Default to register R0
        Jump_Address     <= "000";                    -- Default jump to address 0

        case Opcode is
            when MOVI_OP =>                                  -- Move Immediate
                Immediate_Value <= Instruction(3 downto 0);
                Load_Select     <= '0';                      -- Immediate load mode
                Register_Enable <= Instruction(9 downto 7);

            when ALU_OP =>                                   -- ALU operation
                Register_Select_A <= Instruction(9 downto 7);
                Register_Select_B <= Instruction(6 downto 4);
                Load_Select       <= '1';                    -- Register load mode
                Register_Enable   <= Instruction(9 downto 7);

                case SubOpcode is
                    when SUBOP_ADD =>
                        Operation_Select <= ALU_ADD;
                    when SUBOP_SUB =>
                        Operation_Select <= ALU_SUB;
                    when SUBOP_AND =>
                        Operation_Select <= ALU_AND;
                    when SUBOP_OR =>
                        Operation_Select <= ALU_OR;
                    when SUBOP_XOR =>
                        Operation_Select <= ALU_XOR;
                    when SUBOP_MUL =>
                        Operation_Select <= ALU_MUL;
```

```
            when SUBOP_CMP =>
                Operation_Select <= ALU_CMP;
            when others =>
                Operation_Select <= ALU_ADD;        -- Default to ADD if unknown sub-opcode
        end case;

    when NEG_OP =>                                  -- Negate operation
        Register_Select_A <= Instruction(9 downto 7);
        Register_Select_B <= "000";                 -- Not used for NEG
        Operation_Select  <= ALU_NEG;               -- Negation operation
        Load_Select       <= '1';                   -- Register load mode
        Register_Enable   <= Instruction(9 downto 7);

    when JZR_OP =>                                  -- Jump if Zero
        Register_Select_A <= Instruction(9 downto 7);
        Register_Enable   <= "000";                 -- No register writes

        if Register_Value_For_Jump = "0000" then
            Jump_Enable   <= '1';                   -- Enable jump
            Jump_Address  <= Instruction(2 downto 0);
        else
            Jump_Enable   <= '0';                   -- No jump
        end if;

        when others =>
            -- All outputs already have default values
    end case;
  end process decode;
end architecture Behavioral;
```

Elaborated Design Schematic

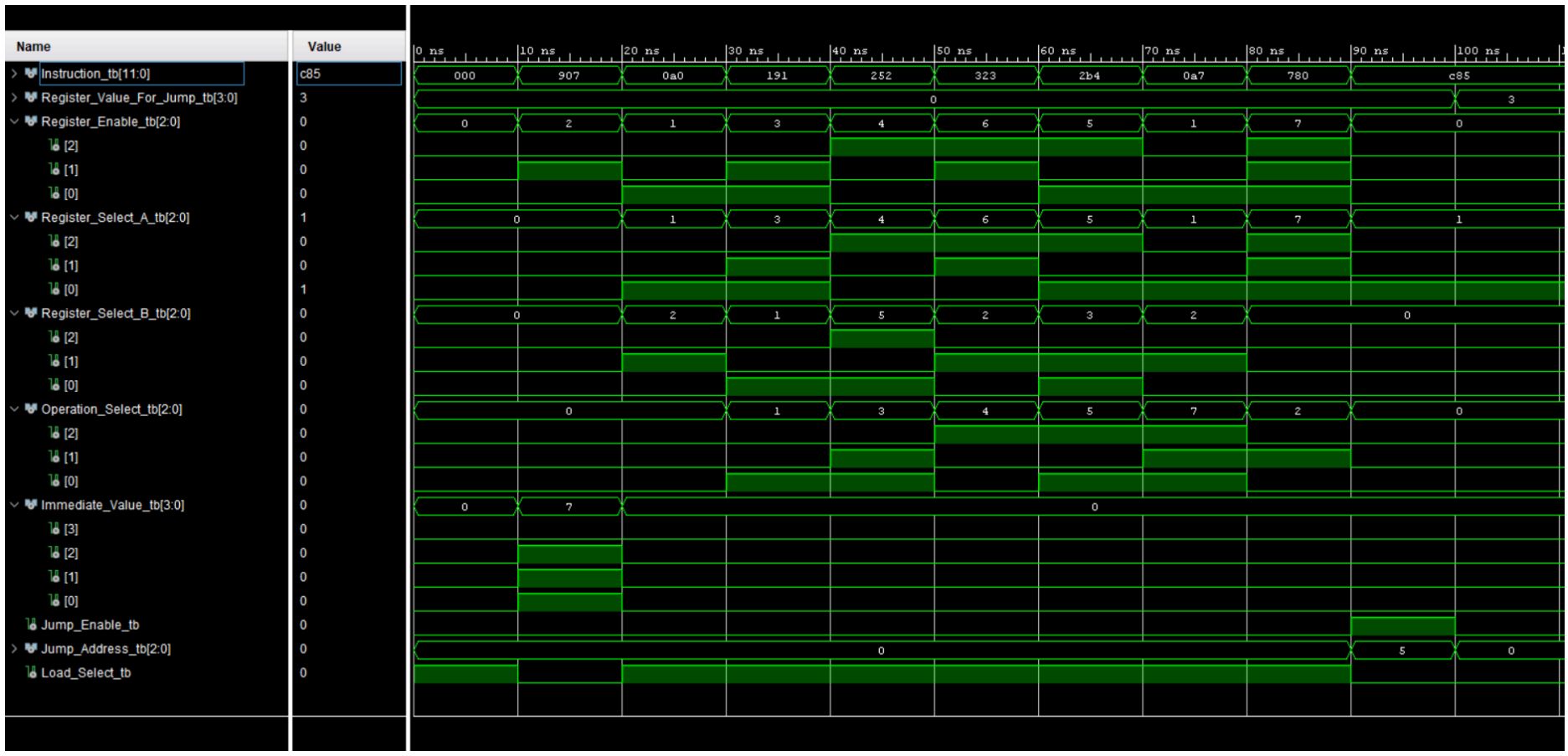


Behavioural Simulation code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.BusDefinitions.all;
use work.constants.all;

entity Instruction_Decoder_TB is
end Instruction_Decoder_TB;

architecture Behavioral of Instruction_Decoder_TB is

    -- DUT component declaration
    component Instruction_Decoder
        port (
            Instruction           : in  InstructionWord;
            Register_Value_For_Jump : in  DataBus;
            Register_Enable       : out RegisterSelect;
            Register_Select_A     : out RegisterSelect;
            Register_Select_B     : out RegisterSelect;
            Operation_Select      : out OperationSelection;
            Immediate_Value       : out DataBus;
            Jump_Enable           : out STD_LOGIC;
            Jump_Address          : out ProgramCounter;
            Load_Select           : out STD_LOGIC
        );
    end component;

    -- Signals to connect to DUT
    signal Instruction           : InstructionWord := (others => '0');
    signal Register_Value_For_Jump : DataBus := (others => '0');
    signal Register_Enable       : RegisterSelect;
    signal Register_Select_A     : RegisterSelect;
    signal Register_Select_B     : RegisterSelect;
    signal Operation_Select      : OperationSelection;
    signal Immediate_Value       : DataBus;
    signal Jump_Enable           : STD_LOGIC;
    signal Jump_Address          : ProgramCounter;
    signal Load_Select           : STD_LOGIC;

begin

    -- Instantiate the DUT
    DUT: Instruction_Decoder
        port map (
            Instruction           => Instruction,
            Register_Value_For_Jump => Register_Value_For_Jump,
            Register_Enable       => Register_Enable,
            Register_Select_A     => Register_Select_A,
            Register_Select_B     => Register_Select_B,
            Operation_Select      => Operation_Select,
            Immediate_Value       => Immediate_Value,
            Jump_Enable           => Jump_Enable,
            Jump_Address          => Jump_Address,
            Load_Select           => Load_Select
        );

-- Stimulus process
stim_proc: process
begin
    -- MOVI R3, #5 => Opcode=00, Reg=011, Imm=0101
    Instruction <= "00" & "011" & "000" & "0101";
    wait for 20 ns;

    -- ADD R2, R1 => Opcode=01, A=010, B=001, SubOpcode=0000
    Instruction <= "01" & "010" & "001" & "0000";
    wait for 20 ns;
```

```
        -- SUB R4, R5 => Opcode=01, A=100, B=101, SubOpcode=0001
        Instruction <= "01" & "100" & "101" & "0001";
        wait for 20 ns;

        -- NEG R6 => Opcode=10, A=110
        Instruction <= "10" & "110" & "000" & "0000";
        wait for 20 ns;

        -- JZR R7, Jump to address 3 if R7 = 0
        Instruction <= "11" & "111" & "000" & "0011";
        Register_Value_For_Jump <= "0000";  -- will trigger jump
        wait for 20 ns;

        -- JZR R7, but R7 ≠ 0 so jump not triggered
        Register_Value_For_Jump <= "1001";  -- won't jump
        wait for 20 ns;

        wait;
    end process;

end Behavioral;
```

Timing Diagram



9.2 Arithmetic and Logic Unit

Design Source Code

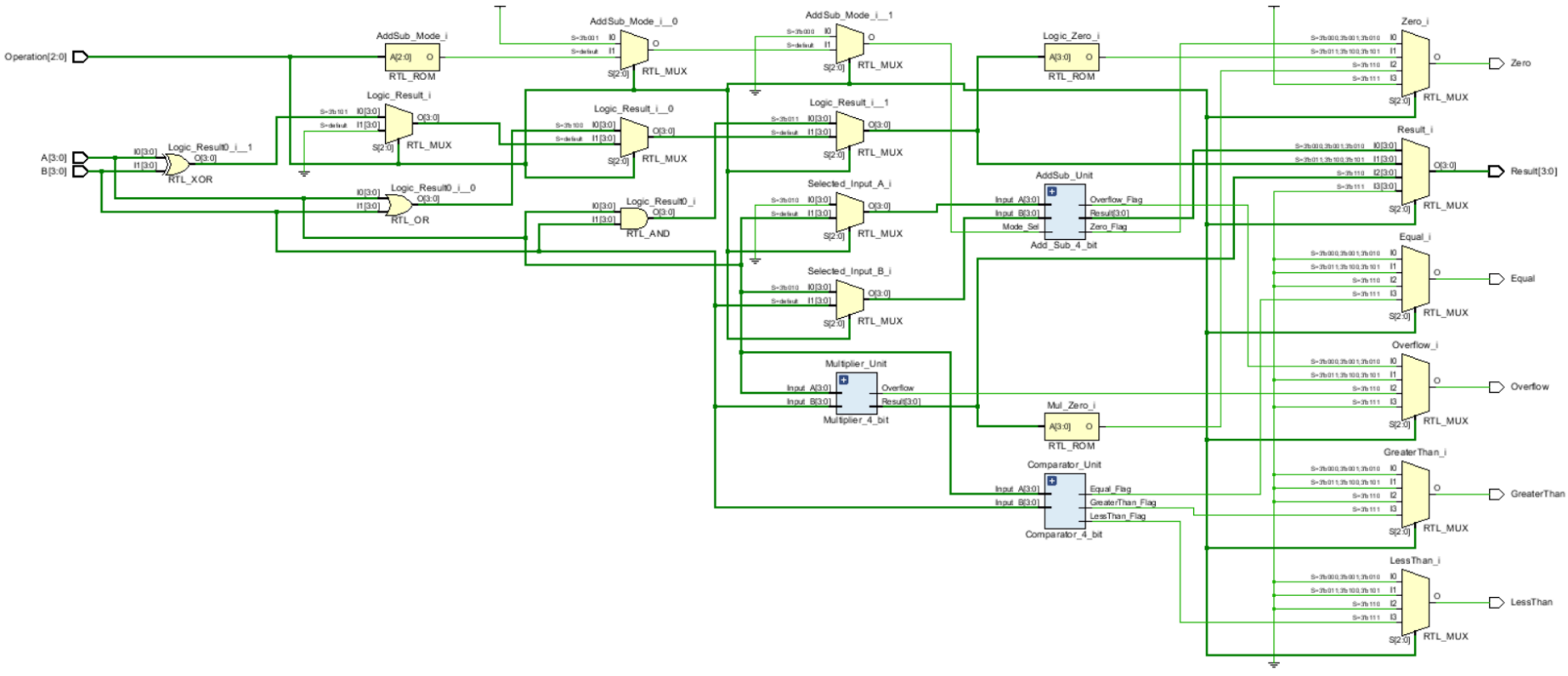
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.BusDefinitions.all;
use work.constants.all;

entity ALU is
    Port(
        A               : in  DataBus;
        B               : in  DataBus;
        Operation       : in  OperationSelection;
        Result          : out DataBus;
```

```
        Zero            : out STD_LOGIC;
        Overflow        : out STD_LOGIC;

        Equal           : out STD_LOGIC;
        LessThan        : out STD_LOGIC;
        GreaterThan     : out STD_LOGIC
    );
end ALU;

architecture Behavioral of ALU is
    component Add_Sub_4_bit
        Port(
            Input_A        : in  DataBus;
            Input_B        : in  DataBus;
            Mode_Sel       : in  STD_LOGIC;
            Result         : out DataBus;
            Zero_Flag      : out STD_LOGIC;
            Overflow_Flag  : out STD_LOGIC
        );
    end component;

    component Comparator_4_bit
        Port(
            Input_A          : in  DataBus;
            Input_B          : in  DataBus;
            Equal_Flag       : out STD_LOGIC;
            LessThan_Flag    : out STD_LOGIC;
            GreaterThan_Flag : out STD_LOGIC
        );
     end component;

    component Multiplier_4_bit
        Port(
            Input_A        : in  DataBus;
            Input_B        : in  DataBus;
            Result         : out DataBus;
            Overflow       : out STD_LOGIC
        );
    end component;

    -- Internal signals
    signal AddSub_Mode      : STD_LOGIC;
    signal AddSub_Result    : DataBus;
    signal AddSub_Zero      : STD_LOGIC;
    signal AddSub_Overflow  : STD_LOGIC;
    signal Zero_Value       : DataBus := "0000";
    signal Logic_Result     : DataBus;
    signal Logic_Zero       : STD_LOGIC;
    signal CMP_Equal        : STD_LOGIC;
    signal CMP_LessThan     : STD_LOGIC;
    signal CMP_GreaterThan  : STD_LOGIC;
    signal Mul_Result       : DataBus;
    signal Mul_Overflow     : STD_LOGIC;
    signal Mul_Zero         : STD_LOGIC;

    -- Signals for input selection to the AddSub unit
    signal Selected_Input_A : DataBus;
    signal Selected_Input_B : DataBus;
begin
    -- Input selection for Add_Sub_4_bit based on operation
    Input_Selection: process(A, B, Operation)
    begin
        -- Default assignment
        Selected_Input_A <= A;
        Selected_Input_B <= B;

        -- Special case for NEG operation
        if Operation = ALU_NEG then
            -- Method 1: 0 - A
```

```
        Selected_Input_A <= Zero_Value;  -- Use 0 as first input
        Selected_Input_B <= A;           -- Use A as second input
    end if;
end process Input_Selection;

AddSub_Unit : Add_Sub_4_bit
    port map(
        Input_A        => Selected_Input_A,
        Input_B        => Selected_Input_B,
        Mode_Sel       => AddSub_Mode,
        Result         => AddSub_Result,
        Zero_Flag      => AddSub_Zero,
        Overflow_Flag  => AddSub_Overflow
    );

Comparator_Unit : Comparator_4_bit
    port map(
        Input_A        => A,
        Input_B        => B,
        Equal_Flag     => CMP_Equal,
        LessThan_Flag  => CMP_LessThan,
        GreaterThan_Flag => CMP_GreaterThan
    );

Multiplier_Unit : Multiplier_4_bit
    port map(
        Input_A        => A,
        Input_B        => B,
        Result         => Mul_Result,
        Overflow       => Mul_Overflow
    );

AddSub_Mode_Control : process(Operation)
begin
    if Operation = ALU_ADD then
        AddSub_Mode <= '0';  -- Addition mode
    elsif Operation = ALU_SUB then
        AddSub_Mode <= '1';  -- Subtraction mode
    elsif Operation = ALU_NEG then
        AddSub_Mode <= '1';  -- Use subtraction mode for negation (0-A)
    else
        -- Default to addition for any other operation
        AddSub_Mode <= '0';
    end if;
end process AddSub_Mode_Control;

-- Logic operations processing
Logic_Operations: process(A, B, Operation)
begin
    if Operation = ALU_AND then
        Logic_Result <= A and B;
    elsif Operation = ALU_OR then
        Logic_Result <= A or B;
    elsif Operation = ALU_XOR then
        Logic_Result <= A xor B;
    else
        Logic_Result <= (others => '0');  -- Default case
    end if;
end process Logic_Operations;

-- Output multiplexing
Logic_Zero_Detection: process(Logic_Result)
begin
    if Logic_Result = Zero_Value then
        Logic_Zero <= '1';  -- Result is zero
    else
        Logic_Zero <= '0';  -- Result is non-zero
    end if;
end process Logic_Zero_Detection;
```

```
    -- Multiplication zero detection
    Mul_Zero_Detection: process(Mul_Result)
    begin
        if Mul_Result = Zero_Value then
            Mul_Zero <= '1';  -- Result is zero
        else
            Mul_Zero <= '0';  -- Result is non-zero
        end if;
    end process Mul_Zero_Detection;

    Result_Selection : process(Operation, Logic_Zero, Logic_Result, AddSub_Result, AddSub_Zero, AddSub_Overflow,
CMP_Equal, CMP_LessThan, CMP_GreaterThan, Mul_Result, Mul_Zero, Mul_Overflow)
    begin
        case Operation is
            when ALU_ADD | ALU_SUB | ALU_NEG =>
                Result <= AddSub_Result;
                Zero <= AddSub_Zero;
                Overflow <= AddSub_Overflow;
                Equal <= '0';
                LessThan <= '0';
                GreaterThan <= '0';

            when ALU_AND | ALU_OR | ALU_XOR =>
                Result <= Logic_Result;
                Zero <= Logic_Zero;
                Overflow <= '0';
                Equal <= '0';
                LessThan <= '0';
                GreaterThan <= '0';

            when ALU_MUL =>
                Result <= Mul_Result;
                Zero <= Mul_Zero;
                Overflow <= Mul_Overflow;
                Equal <= '0';
                LessThan <= '0';
                GreaterThan <= '0';

            when ALU_CMP =>
                Result <= Zero_Value;
                Zero <= '1';
                Overflow <= '0';
                Equal <= CMP_Equal;
                LessThan <= CMP_LessThan;
                GreaterThan <= CMP_GreaterThan;

            when others =>
                -- For any undefined operation, return zero with appropriate flags
                Result <= Zero_Value;
                Zero <= '1';
                Overflow <= '0';
                Equal <= '0';
                LessThan <= '0';
                GreaterThan <= '0';
        end case;
    end process Result_Selection;
end architecture Behavioral;
```

Elaborated Design Schematic



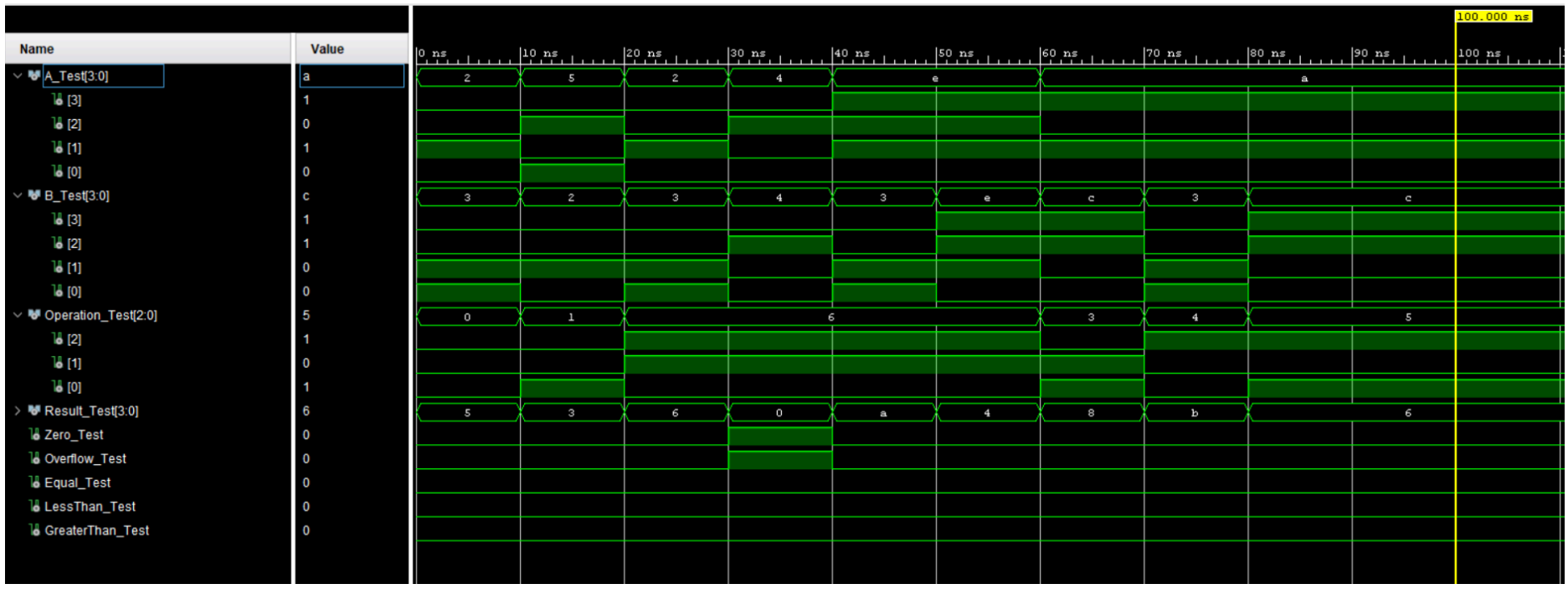
Behavioural Simulation Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.BusDefinitions.all;
use work.Constants.all;

entity Instruction_Decoder_TB is
--  Testbench has no ports
end Instruction_Decoder_TB;

architecture Behavioral of Instruction_Decoder_TB is
    -- Component declaration for the Unit Under Test (UUT)
    component Instruction_Decoder
        port (
            Instruction           : in  InstructionWord;
            Register_Value_For_Jump : in  DataBus;
            Register_Enable       : out RegisterSelect;
            Register_Select_A     : out RegisterSelect;
            Register_Select_B     : out RegisterSelect;
            Operation_Select      : out OperationSelection;
            Immediate_Value       : out DataBus;
            Jump_Enable           : out STD_LOGIC;
            Jump_Address          : out ProgramCounter;
            Load_Select           : out STD_LOGIC
        );
    end component;

    -- Test signals
    signal Instruction_tb         : InstructionWord := (others => '0');
    signal Register_Value_For_Jump_tb : DataBus := "0000";
    signal Register_Enable_tb     : RegisterSelect;
    signal Register_Select_A_tb   : RegisterSelect;
    signal Register_Select_B_tb   : RegisterSelect;
    signal Operation_Select_tb    : OperationSelection;
    signal Immediate_Value_tb     : DataBus;
    signal Jump_Enable_tb         : STD_LOGIC;
    signal Jump_Address_tb        : ProgramCounter;
    signal Load_Select_tb         : STD_LOGIC;

    -- Helper function to convert opcode to string
    function opcode_to_string(op: std_logic_vector(1 downto 0)) return string is
    begin
        case op is
```

```
        when MOVI_OP => return "MOVI";
        when ALU_OP  => return "ALU";
        when NEG_OP  => return "NEG";
        when JZR_OP  => return "JZR";
        when others  => return "Unknown";
    end case;
end function;

-- Helper function to convert ALU operation to string
function alu_op_to_string(op: std_logic_vector(2 downto 0)) return string is
begin
    case op is
        when ALU_ADD => return "ADD";
        when ALU_SUB => return "SUB";
        when ALU_NEG => return "NEG";
        when ALU_AND => return "AND";
        when ALU_OR  => return "OR";
        when ALU_XOR => return "XOR";
        when ALU_CMP => return "CMP";
        when others  => return "Unknown";
    end case;
end function;

begin
    -- Instantiate the Unit Under Test (UUT)
    uut: Instruction_Decoder
        port map (
            Instruction => Instruction_tb,
            Register_Value_For_Jump => Register_Value_For_Jump_tb,
            Register_Enable => Register_Enable_tb,
            Register_Select_A => Register_Select_A_tb,
            Register_Select_B => Register_Select_B_tb,
            Operation_Select => Operation_Select_tb,
            Immediate_Value => Immediate_Value_tb,
            Jump_Enable => Jump_Enable_tb,
            Jump_Address => Jump_Address_tb,
            Load_Select => Load_Select_tb
        );

    -- Stimulus process
    stim_proc: process
        -- Helper procedure to format and display decoder outputs
        procedure display_output(test_name: string) is
            variable op_name: string(1 to 10) := (others => ' ');
        begin
            op_name(1 to 3) := opcode_to_string(Instruction_tb(11 downto 10))(1 to 3);

        end procedure;

    begin
        -- Wait for global reset
        wait for clk_period;



        -- Test 1: MOVI Instruction - Move immediate value 7 to register R2
        Instruction_tb <= "10" & "010" & "000" & "0111";  -- MOVI R2, 7
        wait for clk_period;
        display_output("MOVI R2, 7");

        -- Test 2: ALU ADD Instruction - R1 = R1 + R2
        Instruction_tb <= "00" & "001" & "010" & "0000";  -- ADD R1, R2
        wait for clk_period;
        display_output("ADD R1, R2");

        -- Test 3: ALU SUB Instruction - R3 = R3 - R1
        Instruction_tb <= "00" & "011" & "001" & "0001";  -- SUB R3, R1
        wait for clk_period;
        display_output("SUB R3, R1");
```

```
        -- Test 4: ALU AND Instruction - R4 = R4 & R5
        Instruction_tb <= "00" & "100" & "101" & "0010";  -- AND R4, R5
        wait for clk_period;
        display_output("AND R4, R5");

        -- Test 5: ALU OR Instruction - R6 = R6 | R2
        Instruction_tb <= "00" & "110" & "010" & "0011";  -- OR R6, R2
        wait for clk_period;
        display_output("OR R6, R2");

        -- Test 6: ALU XOR Instruction - R5 = R5 ^ R3
        Instruction_tb <= "00" & "101" & "011" & "0100";  -- XOR R5, R3
        wait for clk_period;
        display_output("XOR R5, R3");

        -- Test 7: ALU CMP Instruction - Compare R1 and R2
        Instruction_tb <= "00" & "001" & "010" & "0111";  -- CMP R1, R2
        wait for clk_period;
        display_output("CMP R1, R2");

        -- Test 8: NEG Instruction - R7 = -R7
        Instruction_tb <= "01" & "111" & "000" & "0000";  -- NEG R7
        wait for clk_period;
        display_output("NEG R7");

        -- Test 9: JZR Instruction with zero register - Jump if R1 == 0 to address 5
        Instruction_tb <= "11" & "001" & "000" & "0101";  -- JZR R1, 5
        Register_Value_For_Jump_tb <= "0000";  -- R1 is zero, should jump
        wait for clk_period;
        display_output("JZR R1, 5 (with R1==0)");

        -- Test 10: JZR Instruction with non-zero register - Jump if R1 == 0 to address 5
        Instruction_tb <= "11" & "001" & "000" & "0101";  -- JZR R1, 5
        Register_Value_For_Jump_tb <= "0011";  -- R1 is non-zero, should not jump
        wait for clk_period;
        display_output("JZR R1, 5 (with R1!=0)");



        -- End simulation
        wait;
    end process;
end Behavioral;
```

Timing Diagram

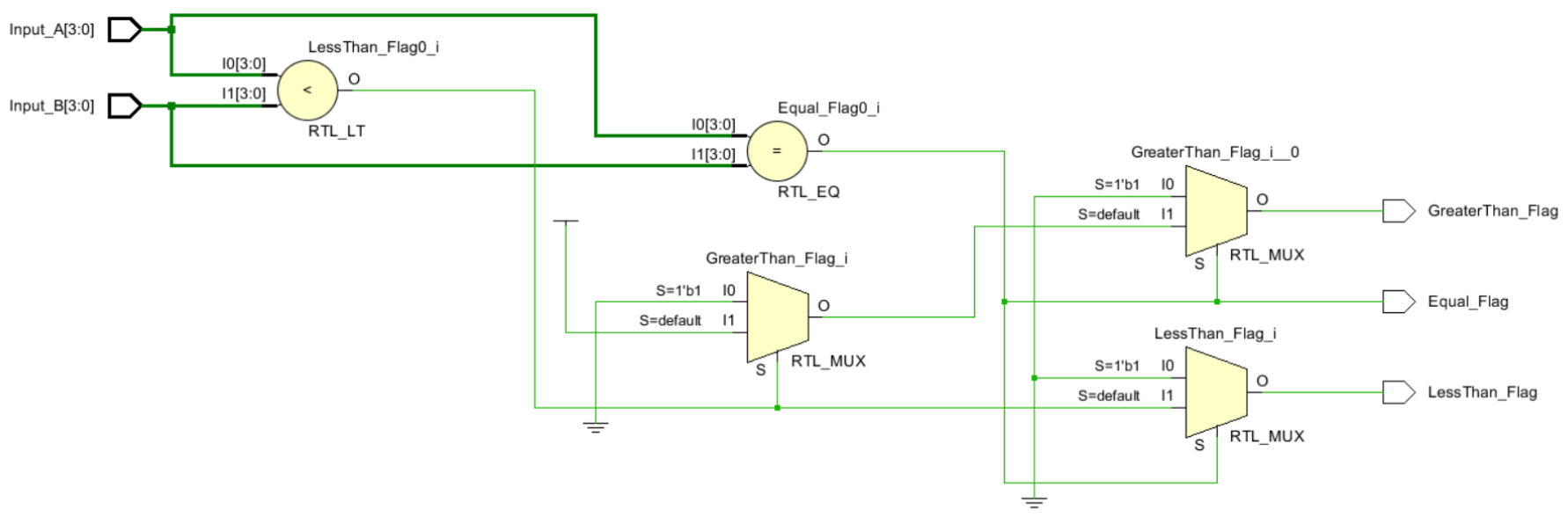


9.3 Comparator

Design Source File

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL; -- Needed for signed comparisons
use work.BusDefinitions.all;

entity Comparator_4_bit is
    Port(
        Input_A          : in  DataBus;
        Input_B          : in  DataBus;
        Equal_Flag       : out STD_LOGIC;
        LessThan_Flag    : out STD_LOGIC;
        GreaterThan_Flag : out STD_LOGIC
    );
end Comparator_4_bit;

architecture Behavioral of Comparator_4_bit is
begin
    Compare_Process: process(Input_A, Input_B)
    begin
        if SIGNED(Input_A) = SIGNED(Input_B) then
            Equal_Flag       <= '1';
            LessThan_Flag    <= '0';
            GreaterThan_Flag <= '0';
        elsif SIGNED(Input_A) < SIGNED(Input_B) then
            Equal_Flag       <= '0';
            LessThan_Flag    <= '1';
            GreaterThan_Flag <= '0';
        else
            Equal_Flag       <= '0';
            LessThan_Flag    <= '0';
            GreaterThan_Flag <= '1';
        end if;
    end process;
end Behavioral;
```

Elaborated Design



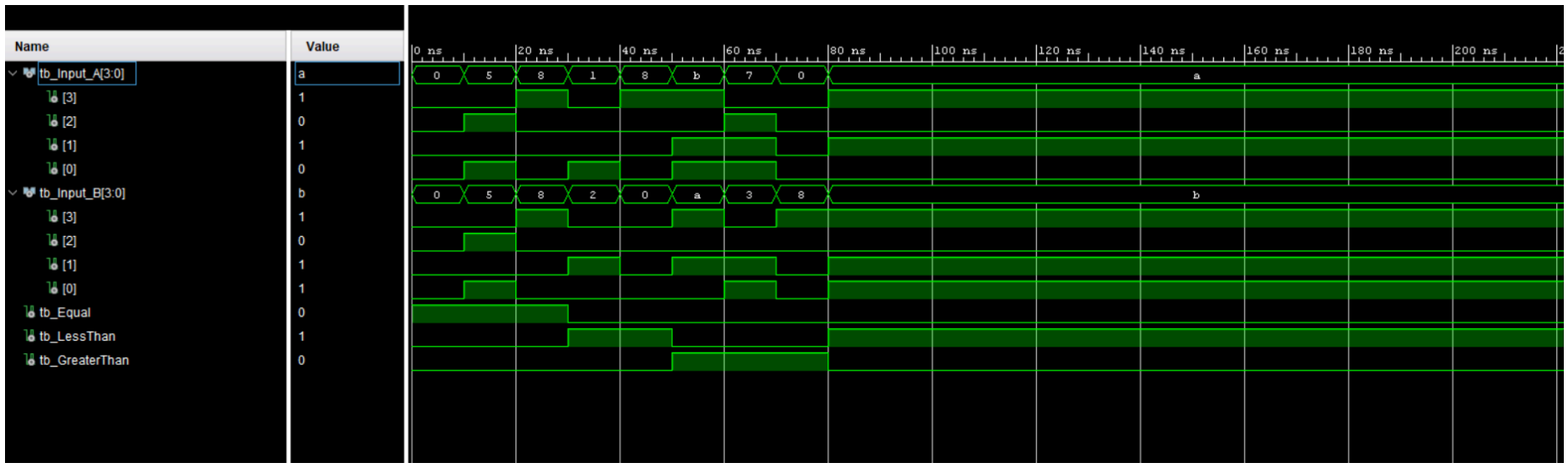
Behavioural Source file

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.BusDefinitions.all;
```

```vhdl
entity Comparator_4_bit_TB is
-- Testbench has no ports
end Comparator_4_bit_TB;

architecture Behavioral of Comparator_4_bit_TB is
    -- Component declaration for the Unit Under Test
    component Comparator_4_bit
        Port(
            Input_A           : in  DataBus;
            Input_B           : in  DataBus;
            Equal_Flag        : out STD_LOGIC;
            LessThan_Flag     : out STD_LOGIC;
            GreaterThan_Flag  : out STD_LOGIC
        );
    end component;

    -- Test signals
    signal tb_Input_A          : DataBus := (others => '0');
    signal tb_Input_B          : DataBus := (others => '0');
    signal tb_Equal            : STD_LOGIC;
    signal tb_LessThan         : STD_LOGIC;
    signal tb_GreaterThan      : STD_LOGIC;

begin
    -- Instantiate the Unit Under Test (UUT)
    UUT: Comparator_4_bit
    port map (
        Input_A          => tb_Input_A,
        Input_B          => tb_Input_B,
        Equal_Flag       => tb_Equal,
        LessThan_Flag    => tb_LessThan,
        GreaterThan_Flag => tb_GreaterThan
    );

    -- Stimulus process
    stim_proc: process
    begin
        -- Test case 1: Equal (both 0)
        tb_Input_A <= "0000";
        tb_Input_B <= "0000";
        wait for 10 ns;

        -- Test case 2: Equal (both positive)
        tb_Input_A <= "0101";
        tb_Input_B <= "0101";
        wait for 10 ns;

        -- Test case 3: Equal (both negative)
        tb_Input_A <= "1000";
        tb_Input_B <= "1000";
        wait for 10 ns;

        -- Test case 4: A < B (positive numbers)
        tb_Input_A <= "0001";
        tb_Input_B <= "0010";
        wait for 10 ns;

        -- Test case 5: A < B (negative vs zero)
        tb_Input_A <= "1000";
        tb_Input_B <= "0000";
        wait for 10 ns;

        -- Test case 6: A < B (negative vs negative)
        tb_Input_A <= "1011";
        tb_Input_B <= "1010";
        wait for 10 ns;

        -- Test case 7: A > B (positive numbers)
        tb_Input_A <= "0111";
```

```
        tb_Input_B <= "0011";
        wait for 10 ns;

        -- Test case 8: A > B (zero vs negative)
        tb_Input_A <= "0000";
        tb_Input_B <= "1000";
        wait for 10 ns;

        -- Test case 9: A > B (negative vs negative)
        tb_Input_A <= "1010";
        tb_Input_B <= "1011";
        wait for 10 ns;

        -- End of test
        wait;
    end process;

end Behavioral;
```

Timing Diagram



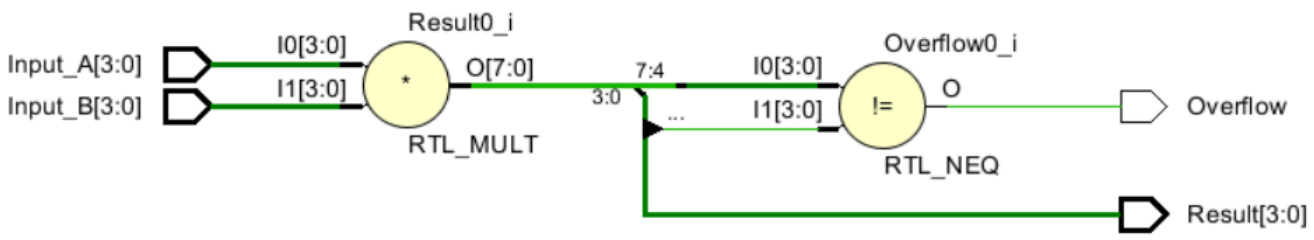
9.4 Multiplier

Design Source File

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.BusDefinitions.all;

entity Multiplier_4_bit is
    Port(
        Input_A   : in  DataBus;      -- 4-bit input A
        Input_B   : in  DataBus;      -- 4-bit input B
        Result    : out DataBus;      -- 4-bit result (lower 4 bits of multiplication)
        Overflow  : out STD_LOGIC     -- Indicates if multiplication result exceeds 4 bits
    );
end entity Multiplier_4_bit;

architecture Behavioral of Multiplier_4_bit is
    signal Full_Result : signed(7 downto 0);  -- Full 8-bit signed result
    signal Sign_Extension : std_logic_vector(3 downto 0);  -- For overflow checking
begin
    process(Input_A, Input_B)
        variable temp_A       : signed(3 downto 0);
        variable temp_B       : signed(3 downto 0);
        variable temp_Result  : signed(7 downto 0);
    begin
        -- Convert 4-bit input to signed
```

```
        temp_A := signed(Input_A);
        temp_B := signed(Input_B);

        -- Perform multiplication
        temp_Result := temp_A * temp_B;

        -- Store result
        Full_Result <= temp_Result;
    end process;

    -- Create sign extension vector for comparison
    Sign_Extension <= (others => Full_Result(3));

    -- Assign lower 4 bits to Result (convert signed to std_logic_vector)
    Result <= std_logic_vector(Full_Result(3 downto 0));

    -- Overflow detection: compare upper 4 bits with sign extension
    Overflow <= '1' when std_logic_vector(Full_Result(7 downto 4)) /= Sign_Extension else '0';

end architecture Behavioral;
```

Elaborated Design Schematic

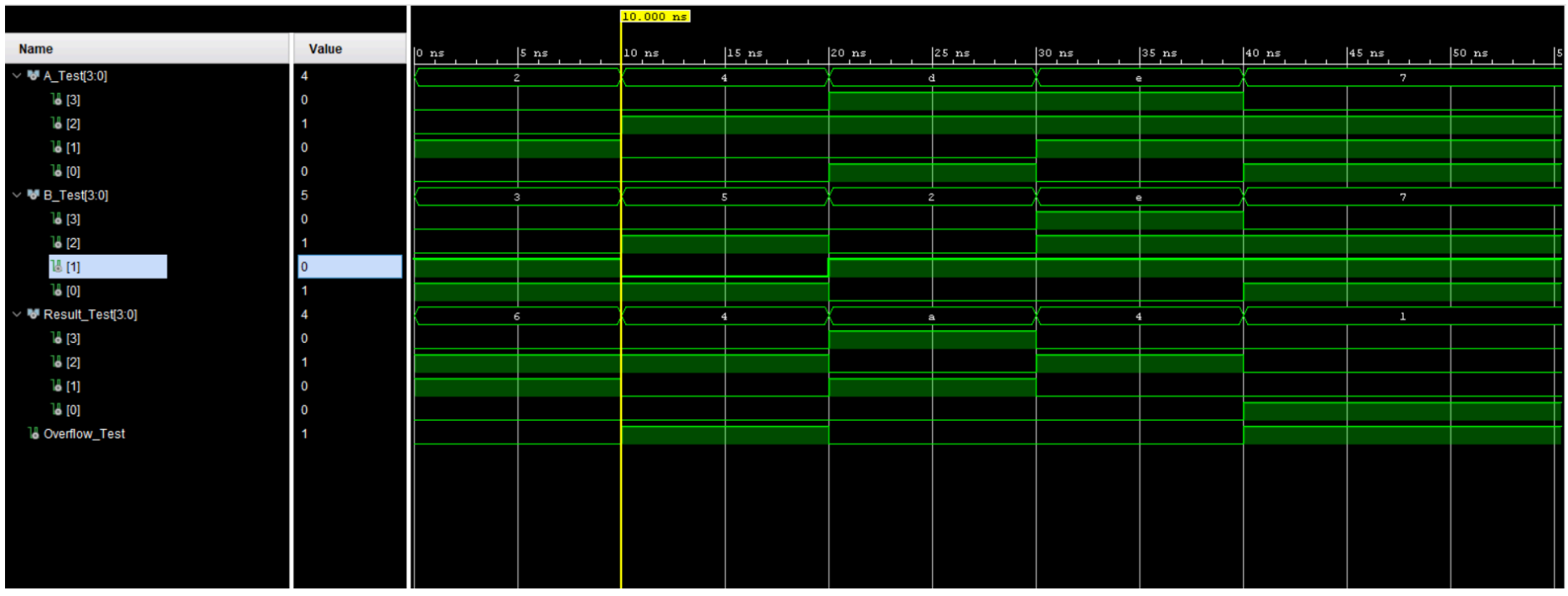


Behavioural Simulation Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.BusDefinitions.all;
use work.Constants.all;

entity Multiplier_4_bit_TB is
    --  No ports for a testbench
end entity Multiplier_4_bit_TB;

architecture Behavioral of Multiplier_4_bit_TB is

    -- Component declaration
    component Multiplier_4_bit is
        Port(
            Input_A        : in  DataBus;
            Input_B        : in  DataBus;
            Result         : out DataBus;
            Overflow       : out STD_LOGIC
        );
    end component;

    -- Signals to drive and observe DUT
    signal A_Test        : DataBus := "0000";
    signal B_Test        : DataBus := "0000";
    signal Result_Test   : DataBus;
    signal Overflow_Test : STD_LOGIC;
```

```
begin

    -- Instantiate the DUT
    UUT: Multiplier_4_bit
        port map (
            Input_A   => A_Test,
            Input_B   => B_Test,
            Result    => Result_Test,
            Overflow  => Overflow_Test
        );

    -- Stimulus process
    process
    begin
        -- Test 1: 2 * 3
        A_Test <= "0010";  -- 2
        B_Test <= "0011";  -- 3
        wait for 10 ns;

        -- Test 2: 4 * 5
        A_Test <= "0100";  -- 4
        B_Test <= "0101";  -- 5
        wait for 10 ns;

        -- Test 3: -3 * 2 (two's complement: -3 = 1101)
        A_Test <= "1101";  -- -3
        B_Test <= "0010";  -- 2
        wait for 10 ns;

        -- Test 4: -2 * -2 (should be +4)
        A_Test <= "1110";  -- -2
        B_Test <= "1110";  -- -2
        wait for 10 ns;

        -- Test 5: 7 * 7 (Overflow expected)
        A_Test <= "0111";  -- 7
        B_Test <= "0111";  -- 7
        wait for 10 ns;

        wait;
    end process;

end architecture Behavioral;
```

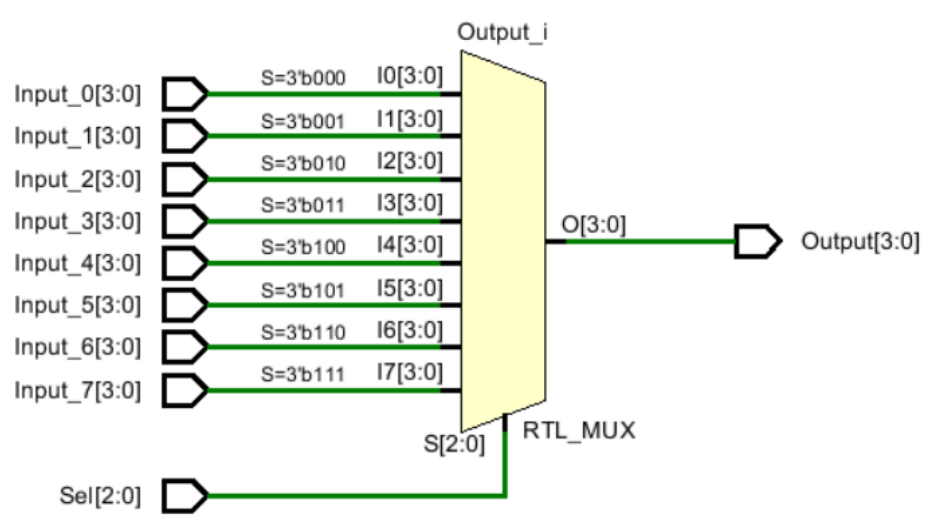
Timing Diagram



10. Building Blocks

10.1 Multiplexer (8 Way 4-bit)

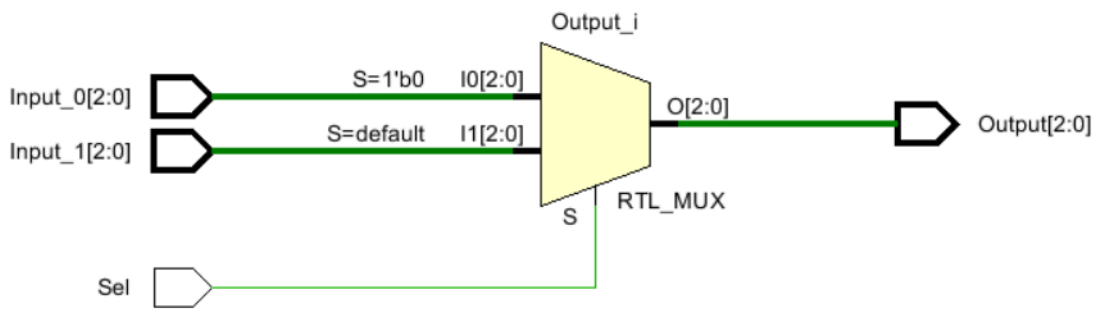
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Mux_8way_4bit is
    Port (
        Input_0    : in  STD_LOGIC_VECTOR(3 downto 0);
        Input_1    : in  STD_LOGIC_VECTOR(3 downto 0);
        Input_2    : in  STD_LOGIC_VECTOR(3 downto 0);
        Input_3    : in  STD_LOGIC_VECTOR(3 downto 0);
        Input_4    : in  STD_LOGIC_VECTOR(3 downto 0);
        Input_5    : in  STD_LOGIC_VECTOR(3 downto 0);
        Input_6    : in  STD_LOGIC_VECTOR(3 downto 0);
        Input_7    : in  STD_LOGIC_VECTOR(3 downto 0);
        Sel        : in  STD_LOGIC_VECTOR(2 downto 0);  -- 3 bits needed for 8 inputs (log2 8 = 3)
        Output     : out STD_LOGIC_VECTOR(3 downto 0)
    );
end Mux_8way_4bit;

architecture Behavioral of Mux_8way_4bit is
begin
    -- Using a selected signal assignment with case
    with Sel select
        Output <=
            Input_0 when "000",
            Input_1 when "001",
            Input_2 when "010",
            Input_3 when "011",
            Input_4 when "100",
            Input_5 when "101",
            Input_6 when "110",
            Input_7 when "111",
            (others => 'X') when others;  -- For completeness, though this case shouldn't occur
end Behavioral;
```



10.2 Multiplexer (2 Way 3-bit)

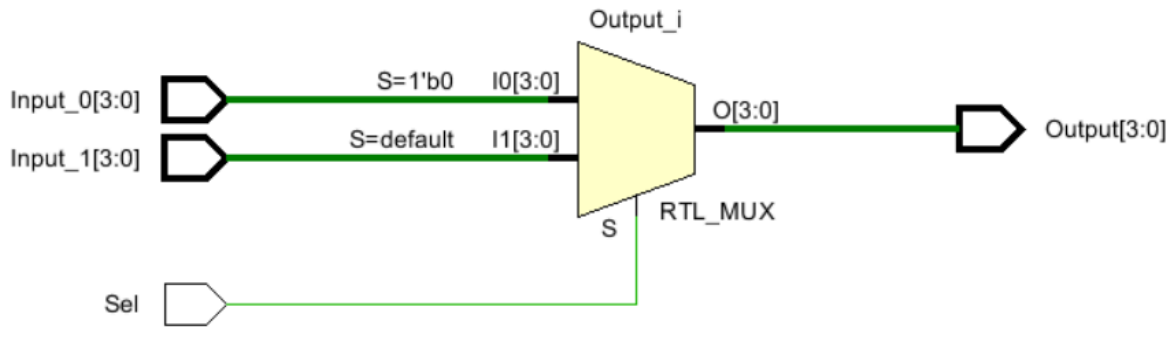
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Mux_2way_3bit is
    Port (
        Input_0    : in  STD_LOGIC_VECTOR(2 downto 0);  -- First 3-bit input
```

```
        Input_1    : in  STD_LOGIC_VECTOR(2 downto 0);  -- Second 3-bit input
        Sel        : in  STD_LOGIC;                     -- Selection control (1 bit for 2 inputs)
        Output     : out STD_LOGIC_VECTOR(2 downto 0)   -- 3-bit output
    );
end Mux_2way_3bit;

architecture Behavioral of Mux_2way_3bit is
begin
    -- Behavioral implementation
    Output <= Input_0 when Sel = '0' else Input_1;
end Behavioral;
```



10.3 Multiplexer (2 Way 4-bit)

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

entity Mux_2way_4bit is
    Port (
        Input_0    : in  STD_LOGIC_VECTOR(3 downto 0);  -- First 4-bit input
        Input_1    : in  STD_LOGIC_VECTOR(3 downto 0);  -- Second 4-bit input
        Sel        : in  STD_LOGIC;                     -- Selection control
        Output     : out STD_LOGIC_VECTOR(3 downto 0)   -- 4-bit output
    );
end Mux_2way_4bit;

architecture Behavioral of Mux_2way_4bit is
begin
    -- Behavioral implementation
    Output <= Input_0 when Sel = '0' else Input_1;
end Behavioral;
```



10.4 Half Adder

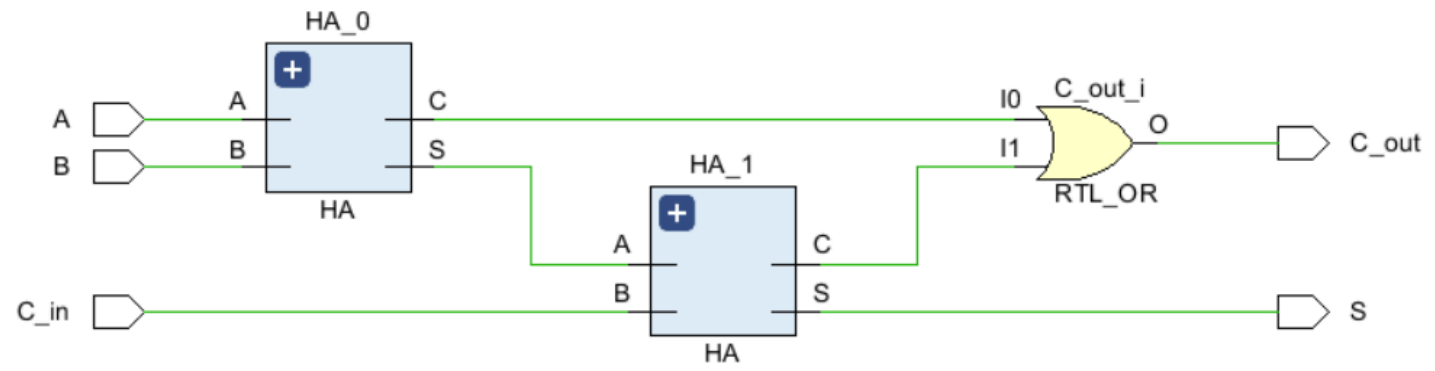
```
library IEEE;
```

```vhdl
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;
```

10.5 Full Adder

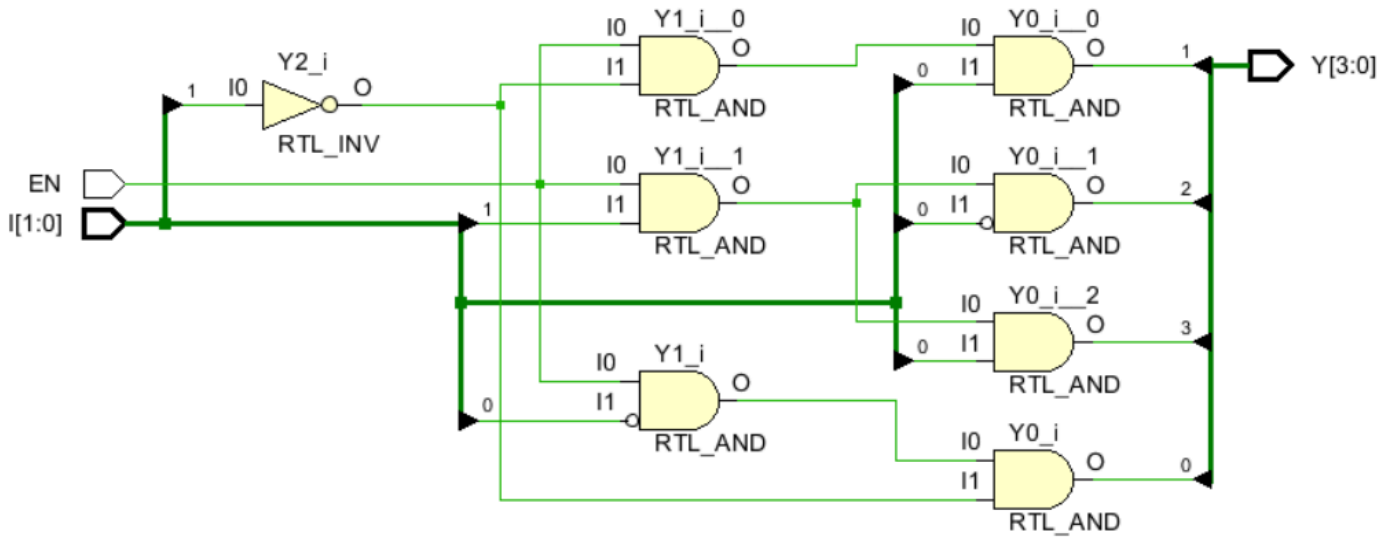
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


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;
```



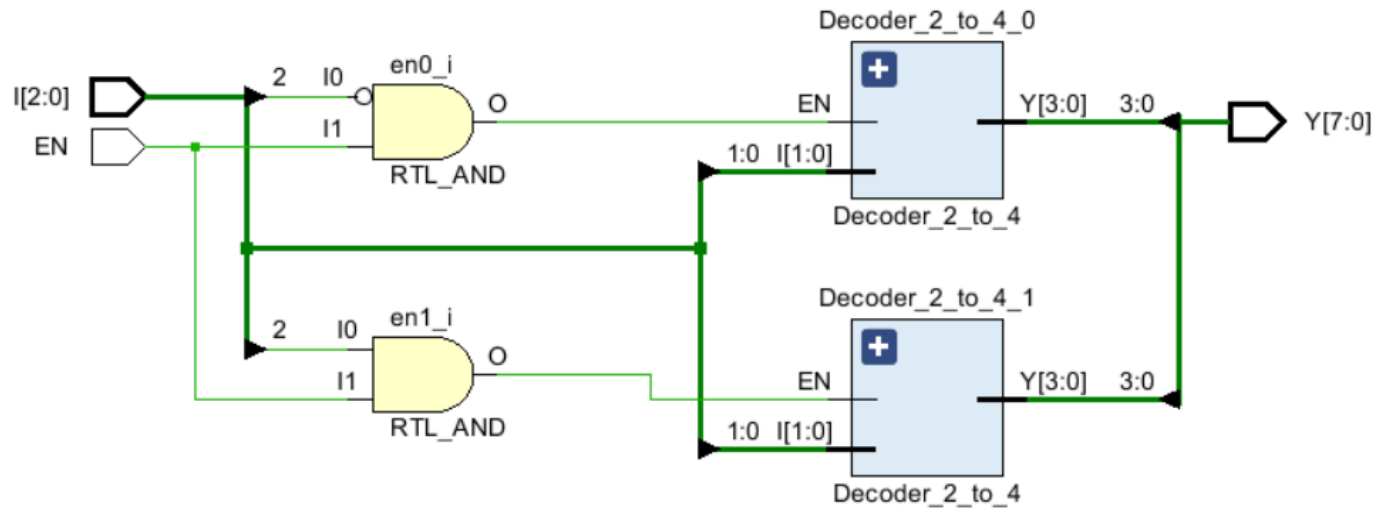
10.6 Decoder (2 to 4)

```
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
    process(I, EN)
    begin
        Y(0) <= EN and (not I(0)) and (not I(1));  -- For input "00"
        Y(1) <= EN and (not I(1)) and I(0);        -- For input "01"
        Y(2) <= EN and I(1) and (not I(0));        -- For input "10"
        Y(3) <= EN and I(1) and I(0);              -- For input "11" - FIXED!
    end process;
end Behavioral;
```



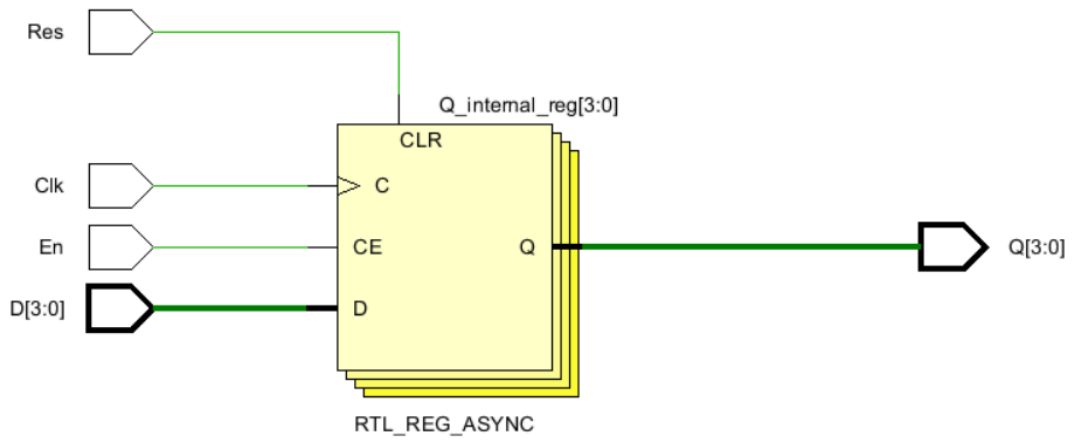
10.7 Decoder (3 to 8)

```
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 en0 , en1 , I2 : STD_LOGIC;
begin
    Decoder_2_to_4_0 : Decoder_2_to_4
    port map(
        I => I0,
        EN => en0,
        Y => Y0 );
    Decoder_2_to_4_1 : Decoder_2_to_4
    port map(
        I => I1,
        EN => en1,
        Y => Y1 );
    en0 <= NOT(I(2)) AND EN;
    en1 <= I(2) AND EN;
    I0 <= I(1 downto 0);
    I1 <= I(1 downto 0);
    I2 <= I(2);
    Y(3 downto 0) <= Y0;
    Y(7 downto 4) <= Y1;
end Behavioral;
```



10.8 Register (4-bit)

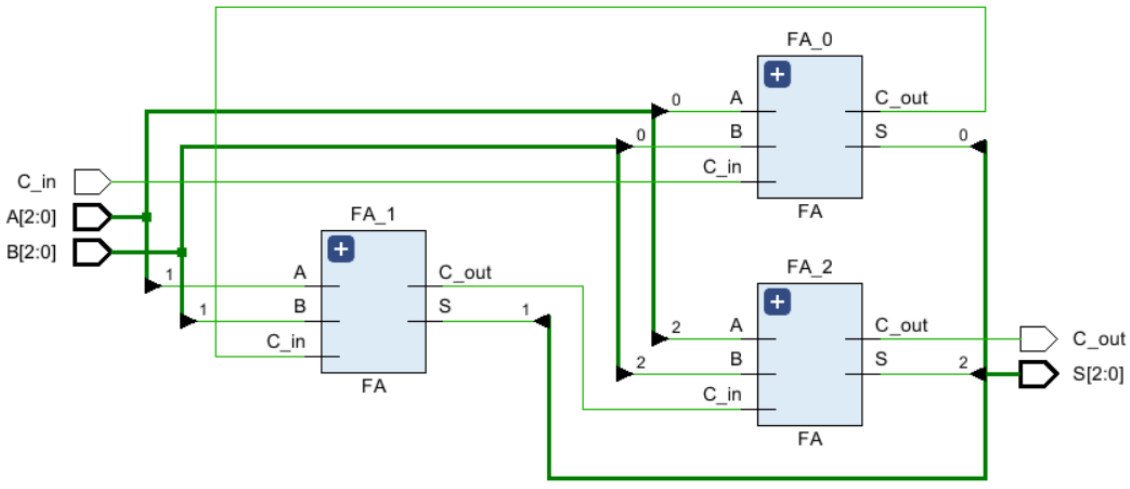
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

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

architecture Behavioral of Reg is
    signal Q_internal : STD_LOGIC_VECTOR(3 downto 0);
begin

    process(Clk, Res)
    begin
        if Res = '1' then
            Q_internal <= (others => '0');
        elsif rising_edge(Clk) then
            if En = '1' then
                Q_internal <= D;
            end if;
        end if;
    end process;

    Q <= Q_internal;

end Behavioral;
```



10.9 Ripple Carry Adder (3-bit)

Design Source Code

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

entity RCA_3 is
    Port (
        A     : in  STD_LOGIC_VECTOR(2 downto 0);  -- 3-bit input A as a bus
        B     : in  STD_LOGIC_VECTOR(2 downto 0);  -- 3-bit input B as a bus
        C_in  : in  STD_LOGIC;                     -- Carry input
        S     : out STD_LOGIC_VECTOR(2 downto 0);  -- 3-bit sum output as a bus
        C_out : out STD_LOGIC                      -- Carry output
    );
end RCA_3;

architecture Behavioral of RCA_3 is
    -- Full Adder component declaration
    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;

    -- Internal carry signals
    SIGNAL carry : std_logic_vector(2 downto 0);

begin
    -- First Full Adder
    FA_0 : FA
        port map (
            A     => A(0),
            B     => B(0),
            C_in  => C_in,   -- External carry input
            S     => S(0),
            C_Out => carry(0)
        );

    -- Second Full Adder
    FA_1 : FA
        port map (
            A     => A(1),
            B     => B(1),
            C_in  => carry(0),
            S     => S(1),
            C_Out => carry(1)
        );

    -- Third Full Adder
    FA_2 : FA
        port map (
            A     => A(2),
            B     => B(2),
            C_in  => carry(1),
            S     => S(2),
            C_Out => C_out
        );

end Behavioral;
```

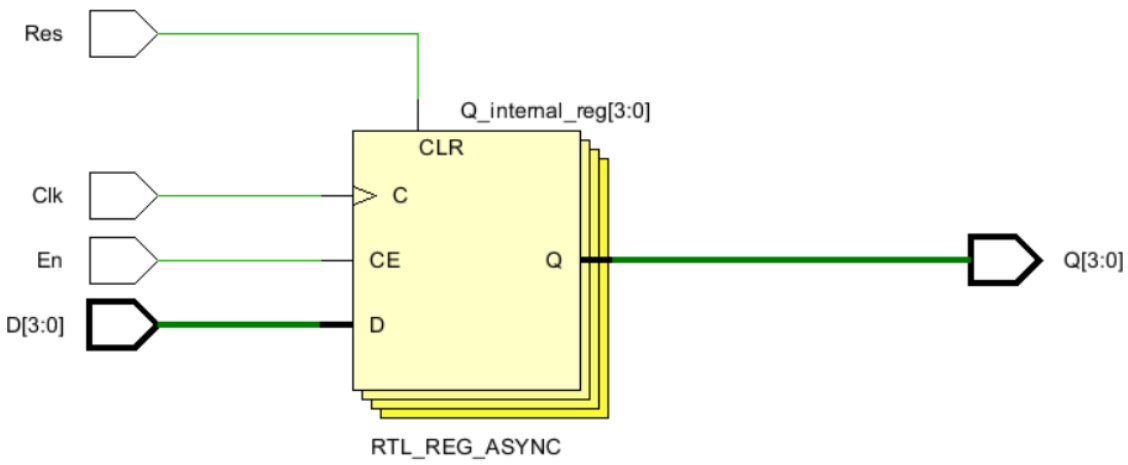


10.10 Ripple Carry Adder (4-bit)

Design Source 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;
           C_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, -- Set to ground
            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 => C_out);
end Behavioral;
```



11. Constraints File

Original Nano processor

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


## LEDs
set_property PACKAGE_PIN U16 [get_ports {Data[0]}]
      set_property IOSTANDARD LVCMOS33 [get_ports {Data[0]}]
set_property PACKAGE_PIN E19 [get_ports {Data[1]}]
      set_property IOSTANDARD LVCMOS33 [get_ports {Data[1]}]
set_property PACKAGE_PIN U19 [get_ports {Data[2]}]
      set_property IOSTANDARD LVCMOS33 [get_ports {Data[2]}]
set_property PACKAGE_PIN V19 [get_ports {Data[3]}]
      set_property IOSTANDARD LVCMOS33 [get_ports {Data[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]
```

Extended Nano Processor

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



## LEDs / Outputs
set_property PACKAGE_PIN U16 [get_ports {Data_7[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Data_7[0]}]

set_property PACKAGE_PIN E19 [get_ports {Data_7[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Data_7[1]}]

set_property PACKAGE_PIN U19 [get_ports {Data_7[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Data_7[2]}]

set_property PACKAGE_PIN V19 [get_ports {Data_7[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Data_7[3]}]

set_property PACKAGE_PIN W18 [get_ports {Data_6[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Data_6[0]}]

set_property PACKAGE_PIN U15 [get_ports {Data_6[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Data_6[1]}]

set_property PACKAGE_PIN U14 [get_ports {Data_6[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Data_6[2]}]

set_property PACKAGE_PIN V14 [get_ports {Data_6[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Data_6[3]}]

set_property PACKAGE_PIN U3 [get_ports Equal]
set_property IOSTANDARD LVCMOS33 [get_ports Equal]

set_property PACKAGE_PIN P3 [get_ports LessThan]
set_property IOSTANDARD LVCMOS33 [get_ports LessThan]

set_property PACKAGE_PIN N3 [get_ports GreaterThan]
```

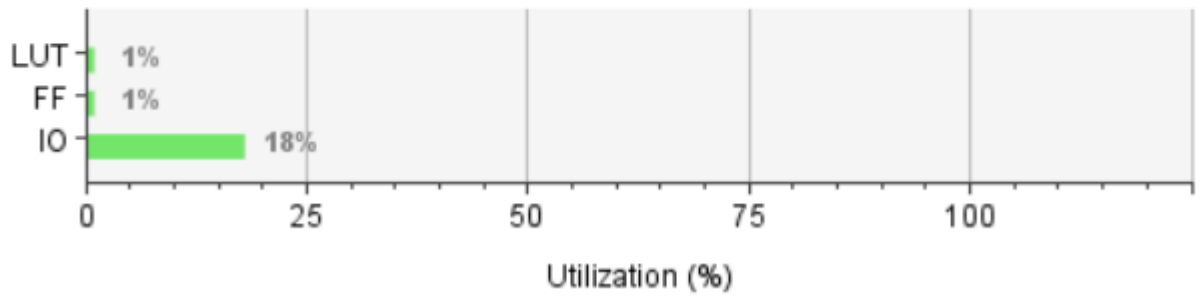
```
set_property IOSTANDARD LVCMOS33 [get_ports GreaterThan]

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]
```

12. Resource Utilization

Original Design

Name	Slice LUTs (20800)	Slice Registers (41600)	Slice (8150)	LUT as Logic (20800)	LUT Flip Flop Pairs (20800)	Bonded IOB (106)	BUFGCTRL (32)
NanoProcessor	33	49	24	33	7	19	1
U_PC (Program_Coun...	18	3	6	18	2	0	0
U_RegBank (Register...	4	12	5	4	0	0	0
U_SlowClock (Slow_C...	11	34	16	11	3	0	0

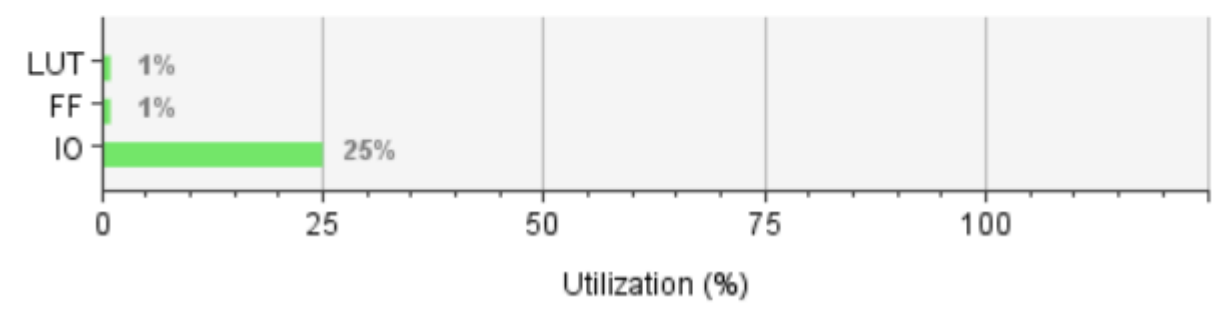
Resource	Utilization	Available	Utilization %
LUT	33	20800	0.16
FF	49	41600	0.12
IO	19	106	17.92



Extended Design

Name	Slice LUTs (20800)	Slice Registers (41600)	Slice (8150)	LUT as Logic (20800)	LUT Flip Flop Pairs (20800)	Bonded IOB (106)	BUFGCTRL (32)
NanoProcessor	63	44	31	63	3	26	1
U_ALU (ALU)	1	0	3	1	0	0	0
U_PC (Program_Coun...	48	3	14	48	2	0	0
U_RegBank (Register...	4	8	6	4	0	0	0
U_SlowClock (Slow_C...	10	33	14	10	1	0	0

Resource	Utilization	Available	Utilization %
LUT	63	20800	0.30
FF	44	41600	0.11
IO	26	106	24.53



13. Conclusion

Project Summary

Our extended nanoprocessor successfully implements a comprehensive instruction set while maintaining compatibility with the original 4-bit architecture. The addition of multiplication, logical operations (AND, OR, XOR), and comparison functionality transformed the basic processor into a more capable computing platform, all while preserving the 12-bit instruction format.

Key Achievements

- **Extended ALU:** Implemented a modular ALU supporting 7 distinct operations
- **Signed Operations:** Added full support for signed arithmetic with proper overflow detection
- **Comparison Logic:** Implemented dedicated comparison flags to enhance control flow capabilities
- **Backward Compatibility:** Maintained compatibility with original instruction format

This project provided practical experience in hardware-software co-design, architectural extension strategies, and making effective resource-performance trade-offs. The modular approach facilitated component testing and integration, reinforcing good design practices that would apply to larger systems.

Though modest in scale, this nanoprocessor embodies fundamental principles of computer architecture and serves as an effective educational platform for understanding the hardware foundations of computing systems.

14. Team Contributions

Name	Contributions	Hours Spent
Himath Nimpura	Instruction Decoders,ALUs, Top Files	24 hours
Kalana Liyanage	Program Counter, 3 bit Adder Deocoders , Constraint file	24 hours
Tharupahan Jayawardana	Program ROM, Register Bank, 7 Segment LUT, Report, Presentation	24 hours
Nisal Wilochana	Multiplexers , Adders Slow Clock , 4 bit Register	24 hours