NANO PROCESSOR PROJECT

FINAL REPORT

DATE: 5TH OF MAY, 2024

Prepared by: Group 44

Course/Department: Computer Science & Engineering

Group Members :

- Dissanayake.D.M.A.K - 220135N
- Ekanayake.L.A.D - 220155B
- Jayasinghe.U.H.N - 220264H
- Lihinikaduarachchi.L.A.G.H - 220361D

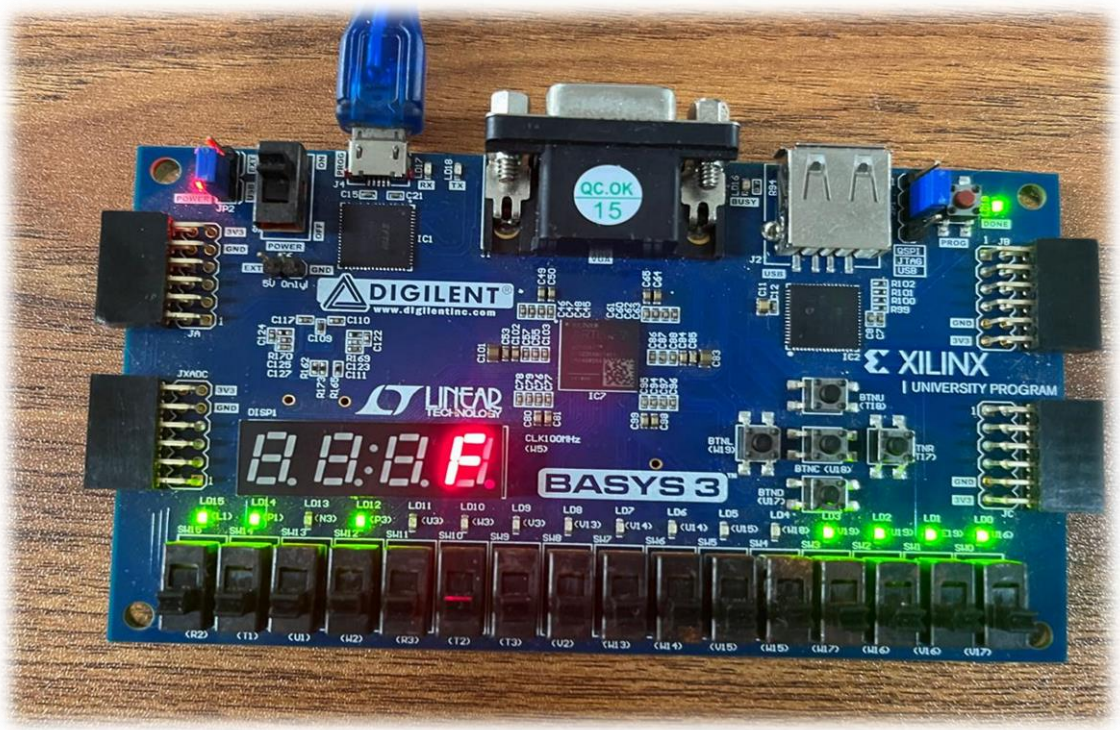


Table of Content

1) Introduction

The modern-day computers are made by interfacing software components with hardware devices that work together to provide the machine with the possibility of receiving data as input and output as results. A microprocessor controls all the functions of the CPU or any other digital device. It functions as an artificial brain which is entirely controlled by a single integrated circuit.

In this lab, we have designed a 4-bit processor capable of executing 8 instructions. Our processor consists of the following components.

1) 4-bit add/subtract unit- Add/subtract unit performs addition and subtraction operations. When only addition operations are considered, our processor can perform addition operation on integers represented using 4 bits. When subtraction operations are performed, integers with only 3 bits can be handled as the MSB is used as the sign bit.

2) Program ROM- This module acts as the memory component in our nano processor. This is used to store the Assembly Programs as hard coded machine code. We have implemented the Program ROM as a LUT.

3) 3-bit Program Counter- Program counter stores the memory address of the next instruction to be executed. As we are using a 3-bit PC, we can include 8 instructions in our program.

4) 3-bit adder -This module is used to increment the Program Counter. This has been implemented using a 4-bit RCA.

5) Instruction Decoder- Instruction Decoder is used to decode the instructions fetched from the Program ROM. It enables the necessary components of the nano processor based on the decoded instruction.

6) Register Bank- Register bank consists of 8,4-bit registers. Value of R0 has been hardcoded to all 0's. This includes a 3-to-8 decoder which enables the required register from the register bank based on the input to the register bank.

7) k-way b-bit multiplexers- We have used several multiplexers in the nano processor to select a specific input from k inputs by using log2k selection lines.

2) Assembly Program

Given below are 3 types of code we have used to implement our program.

- End with no Restart

Assembly code	Machine code
MOVI R1, 3	"0101010000011"
MOVI R2, 1	"0100100000001"
NEG R2	"0010100000000"
ADD R7, R1	"0001111010000"
ADD R1, R2	"0001010100000"
JZR R1, 7	"0111010000111"
JZR R0, 3	"0110000000011"
END	"1110000000000"

- End with Auto Restart

Assembly code	Machine code
MOVI R1, 3	"0101010000011"
MOVI R2, 1	"0100100000001"
NEG R2	"0010100000000"
ADD R7, R1	"0001111010000"
ADD R1, R2	"0001010100000"
JZR R1, 7	"0111010000111"
JZR R0, 3	"0110000000011"
JZR R0, 0	"0110000000000"

- End with Manual Restart

Assembly code	Machine code
MOVI R1, 3	"0101010000101"
MOVI R2, 1	"0100100000001"
NEG R2	"0010100000000"
ADD R7, R1	"0001111010000"
ADD R1, R2	"0001010100000"
JZR R1, 7	"0111010000111"
JZR R0, 3	"0110000000011"
JZR R0, 7	"0110000000111"

3) Additional Features

3.1) Extension of the instruction set

Instead of using 12-bit instructions, we have used 13-bit instructions with 3-bits as the op code for the instruction. Due to the use of 3-bits as the op code, we can include 8 instructions in the instruction set for our nano processor. The relevant op codes are as follows.

- 010 -> MOV
- 000 -> ADD
- 001 -> NEG
- 011 -> JZR
- 100 -> SUB
- 101 -> JMP
- 110 -> JNZR
- 111 -> END

The format of the instructions and the relevant descriptions are given in the following table.

Instruction	Description	Format (13-bit instruction)
MOVI R, d	Move immediate value d to register R, i.e., R ←d,R in [0, 7], d in [0, 15]	0 1 0 R R R 0 0 0 d d d d
ADD Ra, Rb	Add the values in registers Ra and Rb and store in Ra, i.e Ra←Ra+Rb, Ra,Rb in [0,7]	0 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 in [0, 7]	0 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 in [0, 7], d in [0, 7]	0 1 1 R R R 0 0 0 0 d d d
SUB Ra,Rb	Subtract the value in the register Rb from the value in the register Ra and store in Ra,i.e. Ra←Ra-Rb, Ra,Rb in [0,7]	1 0 0 Ra Ra Ra Rb Rb Rb 0 0 0 0
JMP d	Jump to the instruction stored in address d, i.e. PC←d, d in [0,7]	1 0 1 0 0 0 0 0 0 0 d d d
JNZR R, d	Jump if the value in register R is not equal to 0, i.e. if R ≠ 0, PC←d; Else,PC←Pc←+1, R in [0,7], d in [0,7]	1 1 0 R R R 0 0 0 0 d d d
END	Terminates the process of the processor.	1 1 1 0 0 0 0 0 0 0 0 0 0

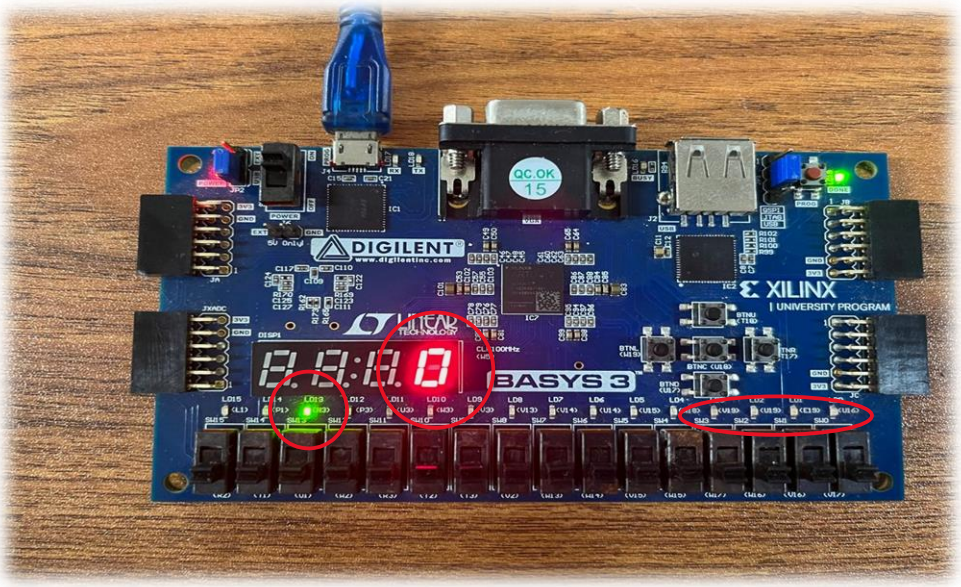
Additionally, a larger instruction size can facilitate more efficient memory access and alignment, potentially reducing memory access overhead and enhancing overall system performance. Furthermore, the increased instruction size allows for greater flexibility in designing instruction formats, enabling future scalability and evolution of the microprocessor architecture.

3.2) Including 4 flags

We have implemented the 4-bit add/subtract unit such that it generates 4 flags: Carry flag, Zero flag, Negative flag and Parity flag (Odd parity detector).

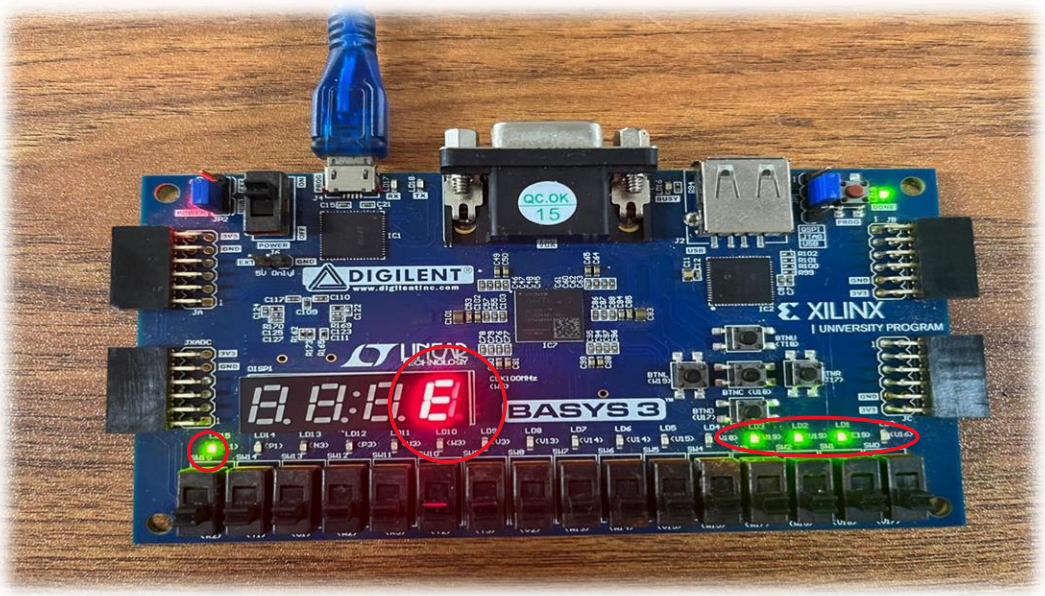
- **Carry Flag**: The Carry flag provides valuable information about arithmetic operations involving larger numbers or when performing multi-byte addition/subtraction. It indicates whether there is a carry-out from the most significant bit during addition or a borrow during subtraction. This flag is crucial for implementing multi-precision arithmetic and handling overflow conditions.

- **Zero Flag**: The Zero flag indicates whether the result of the arithmetic operation is zero. It enables efficient decision-making in program flow control, such as branching or conditional execution. Detecting zero results is essential for a wide range of algorithms and applications, allowing for optimized processing and resource allocation.



- **Negative Flag**: The Negative flag signifies whether the result is a negative integer in 2's complement representation. This flag is essential for signed arithmetic operations, providing insight into the sign of the result. It enables handling of negative numbers and facilitates accurate representation of arithmetic outcomes, ensuring compatibility with standard arithmetic conventions.

- **Parity Flag (Odd Parity Detector)**: The Parity flag detects whether the number of set bits in the result is odd. While not as commonly used as the other flags, it can be valuable in specific applications where data integrity and error detection are paramount. Parity checking can help detect single-bit errors in data transmission or storage, enhancing system reliability and fault tolerance.



Incorporating these flags into our implementation enhances the versatility, functionality, and reliability of the add/subtract unit. They empower users to make informed decisions based on the outcome of arithmetic operations, facilitate error detection, and ensure compatibility with established arithmetic standards. Overall, the inclusion of these flags enriches the computational capabilities of the processor and contributes to its effectiveness in diverse computing tasks.

3.3) Use of Generic MUXs

Instead of building MUXs using decoders, we have implemented a generic MUX.

Implementing the nano processor using generic multiplexers (MUXs) instead of relying solely on decoders offers several advantages:

1.**Flexibility in Design**-Generic MUXs provide greater flexibility in designing the nano processor's architecture. Unlike decoders, which have a fixed number of output lines determined by the input size, MUXs allow for arbitrary selection of input channels. This flexibility enables the nano processor's design to accommodate varying requirements and adapt to changes in specifications more easily.

2.**Customization of Input Selection**-With generic MUXs, the selection of input channels is not restricted to a predetermined pattern as with decoders. This means that the nano processor can selectively choose inputs based on dynamic conditions or programmable logic, enhancing its versatility and adaptability in different scenarios. Additionally, the ability to customise input selection enables optimisation of performance and resource utilization based on specific application requirements.

3. **Reduced Hardware Complexity**-In certain cases, using generic MUXs may result in reduced hardware complexity compared to using decoders. Decoders require additional logic gates to decode input signals and activate specific output lines, which can contribute to increased circuit complexity and power consumption. By employing MUXs directly, the nano processor's design may be more streamlined, leading to improved efficiency and ease of implementation.

4.**Ease of Integration and Debugging**-Generic MUXs are widely available and commonly used in digital circuit design, making them easier to integrate into the nano processor's overall architecture. Furthermore, debugging and troubleshooting may be simplified when using generic MUXs, as they offer a straightforward interface for selecting input signals and observing output behaviour. This can expedite the development process and facilitate rapid iteration and refinement of the nano processor design.

5.**Scalability and Future Expansion**-By leveraging generic MUXs, the nano processor's design can be more easily scaled and expanded to accommodate future requirements or enhancements. MUXs offer a scalable solution for multiplexing input signals, allowing for the addition of new features or increased functionality without significant redesign or restructuring of the overall architecture. This scalability ensures that the nano processor remains adaptable and capable of evolving alongside emerging technologies and application demands.

Overall, utilizing generic MUXs in the implementation of the nano processor offers advantages in terms of flexibility, customization, hardware simplicity, ease of integration, and scalability. These benefits contribute to the efficiency, versatility, and long-term viability of the nano processor design.

3.4) Multiplexers implemented with Tri state Buffers

Implementing the nano processor's multiplexers with tri-state buffers instead of generic MUXs provides several advantages and enhancements to its architecture and functionality:

1. **Enhanced Efficiency and Reduced Power Consumption**:

Tri-state buffers offer improved efficiency and reduced power consumption compared to generic MUXs. Tri-state buffers allow for the isolation of input lines when not in use, preventing unnecessary power consumption by inactive input channels. This feature contributes to the overall energy efficiency of the nano processor, particularly in scenarios where certain input channels remain unused for extended periods.

2. **Dynamic Input Control**:

Tri-state buffers enable dynamic control over input lines, allowing for efficient management of data flow within the nano processor. By selectively enabling or disabling specific input channels using tri-state buffers, the nano processor can adapt its input configuration in real-time based on changing requirements or operational conditions. This dynamic input control enhances the versatility and responsiveness of the nano processor, enabling it to efficiently handle diverse workloads and tasks.

3. **Improved Signal Integrity and Noise Immunity**:

Tri-state buffers enhance signal integrity and noise immunity within the nano processor's architecture. By isolating inactive input lines, tri-state buffers minimize signal interference and crosstalk, ensuring reliable data transmission and processing. This feature is particularly advantageous in high-speed or noise-sensitive applications where maintaining signal integrity is critical for optimal performance.

4. **Simplified Integration and Interfacing**:

The use of tri-state buffers simplifies the integration and interfacing of the multiplexer within the nano processor's design. Tri-state buffers provide a straightforward interface for connecting input and output lines, facilitating seamless communication between different components of the nano processor. This simplification streamlines the development and integration process, reducing complexity and time-to-market for the nano processor.

5. **Scalability and Expandability**:

Tri-state buffers offer scalability and expandability for the nano processor's multiplexer, enabling easy integration of additional input channels or functionality as needed. The modular nature of tri-state buffers allows for simple expansion of the multiplexer without requiring significant redesign or restructuring of the overall architecture. This scalability ensures that the nano processor remains adaptable to future requirements and can evolve alongside emerging technologies and application demands.

Overall, implementing the nano processor's multiplexer with tri-state buffers offers significant enhancements in efficiency, dynamic control, signal integrity, integration simplicity, and scalability. These advantages contribute to the improved performance, versatility, and long-term viability of the nano processor design, positioning it as a robust solution for various computing tasks and applications.

3.4) Implementing a Synchronized Instruction Decoder

Synchronizing the instruction decoder by connecting it to the same clock input as other components of the nano processor offers several advantages:

- **Synchronous Operation**: Synchronizing the instruction decoder ensures that it operates in sync with other components of the processor. This synchronous operation simplifies the overall system design and ensures proper timing relationships between different parts of the processor.
- **Timing Control**: By using the same clock signal for the instruction decoder and other components, we can precisely control the timing of instruction decoding, execution, and

other operations within the processor. This helps in avoiding timing hazards and ensures reliable and predictable behavior of the processor.

- **Ease of Design**: Designing a synchronous system is generally simpler compared to an asynchronous one. Synchronizing the instruction decoder with the rest of the processor components allows for a more straightforward design approach, reducing the chances of timing issues and race conditions.
- **Reduced Complexity**: Synchronous designs often have less complexity than asynchronous ones because they do not require additional circuitry for handling asynchronous signals and timing issues. This leads to simpler implementation and easier debugging and verification of the processor design.
- **Improved Performance**: Synchronizing the instruction decoder allows for better optimization of critical paths within the processor, leading to improved performance. By coordinating the timing of operations, we can minimize latency and maximize throughput in the processor.
- **Facilitates Pipelining**: If the processor is pipelined, having a synchronous instruction decoder is essential for ensuring proper pipeline operation. Synchronization enables the smooth flow of instructions through the pipeline stages, maintaining correct data dependencies and avoiding pipeline stalls.

In summary, synchronizing the instruction decoder with the clock signal used by other components of the nano processor offers benefits such as synchronous operation, precise timing control, simplified design, reduced complexity, improved performance, and support for pipelining.

3.5)Optimization of Synchronization

1) "Clk_slow_bar" connected as the input clock signal to the register bank

While other components of the nano processor are connected to Clk_slow, the register bank is connected to the Clk_Slow_bar. There are several advantages of this implementation.

- **Improved Timing Margins**: By using the complement of the clock signal for the register bank, we can stagger the timing of its operations relative to other components. This can help in avoiding critical timing conflicts and improving timing margins, reducing the risk of setup and hold time violations. As a result, the overall reliability and stability of the nano processor can be enhanced.
- **Reduced Power Consumption**: Operating the register bank on the complement of the clock signal can help in reducing dynamic power consumption. By toggling these components only, when necessary, rather than on every clock cycle, overall power consumption can be minimized. This is particularly beneficial in low-power applications or when power efficiency is a key design consideration.
- **Enhanced Noise Immunity**: Using the complement of the clock signal for the register bank can help in reducing susceptibility to clock-related noise and glitches. By desynchronizing the operation of the register bank from the main clock signal, any transient noise or glitches present on the clock signal are less likely to affect its

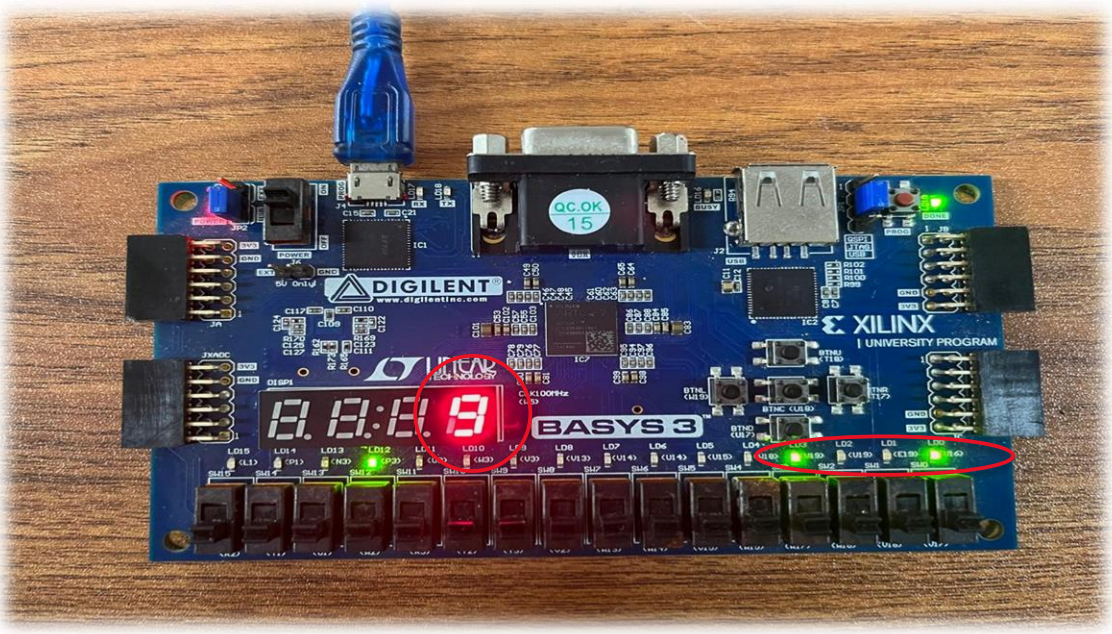
functionality. This can result in improved robustness and reliability of the nano processor in noisy environments.

- **Facilitates Asynchronous Operation:** Connecting the register bank to the complement of the clock signal enables it to operate asynchronously with respect to the rest of the processor. This can be advantageous in scenarios where the component needs to respond to external events or operate at a different frequency than the rest of the processor. Asynchronous operation allows for greater flexibility in design and can simplify interfacing with external devices or asynchronous subsystems.
- **Optimized Performance**: Certain components of the nano processor may have specific timing requirements or performance constraints that can be better met by operating them on the complement of the clock signal. By tailoring the timing of these components independently, we can optimize their performance without impacting the overall operation of the processor. This can lead to improved overall system performance and efficiency.

In summary, connecting the register bank to the complement of the clock signal can offer advantages in terms of timing margins, power consumption, noise immunity, asynchronous operation, and performance optimization. These benefits contribute to the overall robustness, reliability, and efficiency of the nano processor design.

2) Optimal slowing down of Basys3 in-built clock to the ratio 1*10^8:1

The internal clock of the Basys3 board runs at 100MHz. We have implemented a slow clock with an optimal ratio to emit an output only when a certain number of changes in the input is detected. There are several advantages of implementing a slow clock with an optimal rate in the nano processor design.



- **Compatibility with External Interfaces:** Slower clock rates are often compatible with a wider range of external interfaces and peripherals. Many external devices operate at

lower frequencies, so using a slow clock ensures compatibility and simplifies interfacing with external components. This can streamline system integration and reduce the need for additional clock conversion circuitry.

- **Reduced Power Consumption**: A slower clock rate typically results in lower power consumption. By slowing down the clock, the nano processor can operate more efficiently.
- **Improved Timing Margins**: Slowing down the clock allows for more relaxed timing constraints, reducing the likelihood of timing violations such as setup and hold time violations. This can simplify the design process and help ensure reliable operation of the nano processor, particularly in complex or high-speed designs where timing issues may arise.
- **Enhanced Reliability**: Lower clock frequencies can improve the reliability of the nano processor by reducing the likelihood of errors or glitches caused by high-speed operation. Slower clock rates allow for more time for signals to stabilize, reducing the risk of signal integrity issues and improving overall system stability.
- **Simplified Design and Verification**: Designing with a slower clock simplifies the design process and makes verification easier. Complex timing constraints associated with high-speed designs are reduced, making it easier to design and verify the functionality of the nano processor.

In summary, implementing a slow clock in the nano processor design offers advantages such as reduced power consumption, improved timing margins, enhanced reliability, simplified design and verification, compatibility with external interfaces, and cost savings.

3.6) Working with SIGNED INTEGERS

In our implementation of the 4-bit add/subunit, when only addition operations are considered, it has the capability to perform addition on integers represented using 4 bits. Each of these 4 bits can represent a value from 0 to 15. However, when subtraction operations are introduced, a portion of the available bits must be allocated to represent the sign of the integer. This is typically achieved by designating the Most Significant Bit,"MSB" as the sign bit, where 0 signifies a positive number and 1 indicates a negative number. Consequently, when subtraction is allowed, the range of representable integers is halved, as only 3 bits remain to represent the magnitude of the number. This limitation stems from the necessity to reserve a bit for sign representation, thereby reducing the available bits for numerical value storage. Thus, while our processor can handle 4-bit integers for addition operations, it can accommodate 3-bit integers for operations involving subtraction due to the allocation of the MSB as the sign bit.

3.7) **Connection through data busses**

In the Microprocessor we have used ***Single 2D BUS with 8, 4-bit registers***. If busses are not used, 32 lines are required for data transmission. Instead, we can use ***one 2D BUS with 8, 4-bit registers*** for data transmission from the register bank to the two 8-way 4-bit MUXs.

Also, in many other stages we have used ***4-bit and 3-bit BUSSES*** in this design.

There are several advantages of connecting the components through data busses.

- **Reduced Complexity**: Using a single bus instead of 32 individual lines simplifies the overall architecture of the system. This reduces the complexity of routing and managing data transmission within the register bank and the MUXs.
- **Space Efficiency**: Implementing a single bus consumes less physical space compared to having 32 individual lines. This is particularly advantageous in compact or constrained environments, such as in embedded systems or integrated circuits.
- **Improved Signal Integrity**: Consolidating data transmission onto a single bus can enhance signal integrity by reducing the potential for signal interference or crosstalk between lines. This leads to more reliable data transmission and reduces the likelihood of errors.
- **Enhanced Scalability**: The use of a single bus allows for easier scalability of the system. Adding additional components or expanding the system's capabilities can be achieved more seamlessly by simply extending the bus width or adding more bus lines, rather than dealing with a multitude of individual connections.
- **Simplified Control Logic**: With a single bus architecture, the control logic required to manage data transmission becomes more streamlined and easier to implement. This simplification can lead to improved system performance and reduced development time.

Overall, the implementation of a single bus consisting of 8, 4-bit busses offer advantages in terms of reduced complexity, space efficiency, signal integrity, scalability, and simplified control logic, making it a favorable choice for data transmission within the system.

3.8) **Implementing components with a reset option**

In our design we used Reset option for many components such as **Program Counter, Instruction Decoder, Register Bank, etc...** for better removal of garbage and better performance.

There may be garbage values stored in the registers of these components. Reset option allows to set the values stored in all these registers to zero. This improvement offers several benefits.

- **Initialization and Stability**: The reset option ensures that the processor starts from a known state every time it is powered on or reset. This helps in initializing the system reliably and ensures stable operation by clearing any potential garbage values or unintended states that might have occurred during previous executions.
- **Error Correction and Debugging**: In case of unexpected behavior or errors during program execution, the reset option provides a quick and effective means to reset the affected components to a known state. This aids in debugging and troubleshooting issues, as it allows developers to isolate problems and identify their root causes more efficiently.
- **Improved Predictability**: Resetting the PC, instruction decoder, and register bank to zero ensures predictability in the execution flow of the processor. It eliminates any residual effects from previous operations or program executions, allowing subsequent instructions to be executed reliably and without interference from past states.
- **Enhanced Security**: Resetting sensitive components like the PC and registers helps prevent potential security vulnerabilities that could arise from residual data or

unintended states. By clearing out any existing values, the reset option mitigates the risk of manipulation of critical system parameters.

- **Simplified Development and Testing**: The availability of a reset option simplifies the development and testing process for software running on the processor. We can reset the system to a clean state before running tests or executing code, ensuring consistent and reproducible results across multiple test runs.

Overall, implementing components with a reset option in our nano processor enhances its reliability, stability, predictability, security, and ease of development.

4) Components Details & Integration

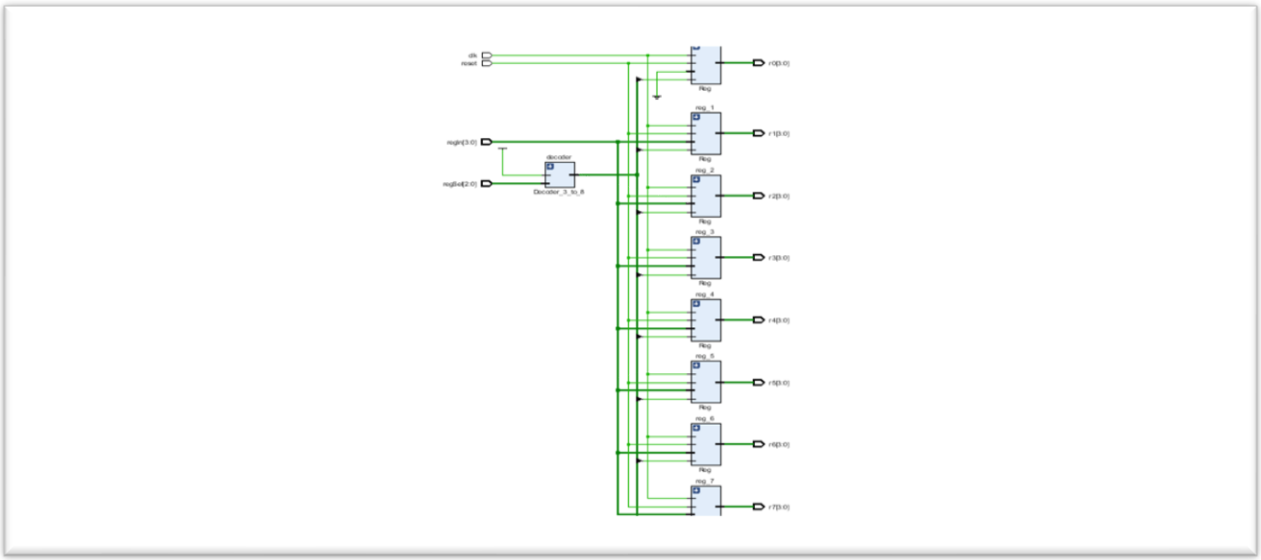
Register Bank

Design Source

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Reg_Bank is Port ( regIn : in STD_LOGIC_VECTOR (3 downto 0); --
Value to be written
regSel : in STD_LOGIC_VECTOR (2 downto 0); -- Register address select
clk : in STD_LOGIC; reset : in STD_LOGIC; -- Reset all registers to 0000
r0 : out STD_LOGIC_VECTOR (3 downto 0);
r1 : out STD_LOGIC_VECTOR (3 downto 0);
r2 : out STD_LOGIC_VECTOR (3 downto 0);
r3 : out STD_LOGIC_VECTOR (3 downto 0);
r4 : out STD_LOGIC_VECTOR (3 downto 0);
r5 : out STD_LOGIC_VECTOR (3 downto 0);
r6 : out STD_LOGIC_VECTOR (3 downto 0);
r7 : out STD_LOGIC_VECTOR (3 downto 0) );
end Reg_Bank;
architecture Behavioral of Reg_Bank is -- Component declarations
component Decoder_3_to_8 is
Port ( input : in STD_LOGIC_VECTOR (2 downto 0);
       enable : in STD_LOGIC;
       output : out STD_LOGIC_VECTOR (7 downto 0) );
end component;
component Reg is
Port ( d : in STD_LOGIC_VECTOR (3 downto 0);
       en : in STD_LOGIC;
       clk : in STD_LOGIC;
       clr : in STD_LOGIC;
       q : out STD_LOGIC_VECTOR (3 downto 0) );
end component;
-- Signal declarations
signal regEnSig : STD_LOGIC_VECTOR (7 downto 0);
begin
-- Instantiate the Decoder_3_to_8 component
decoder : Decoder_3_to_8
port map ( input => regSel,
       enable => '1',
       output => regEnSig );
-- Instantiate the Reg components for each register
reg_0 : Reg
port map ( d => "0000",
       en => regEnSig(0),
       clk => clk,
       clr => reset,
       q => r0 );
reg_1 : Reg
```

```
port map ( d => regIn,
      en => regEnSig(1),
      clk => clk,
      clr => reset,
      q => r1 );
reg_2 : Reg
port map ( d => regIn,
      en => regEnSig(2),
      clk => clk,
      clr => reset,
      q => r2 );
reg_3 : Reg
port map ( d => regIn,
      en => regEnSig(3),
      clk => clk,
      clr => reset,
      q => r3 );
reg_4 : Reg
port map ( d => regIn,
      en => regEnSig(4),
      clk => clk,
      clr => reset,
      q => r4 );
reg_5 : Reg
port map ( d => regIn,
      en => regEnSig(5),
      clk => clk,
      clr => reset,
      q => r5 );
reg_6 : Reg
port map ( d => regIn,
      en => regEnSig(6),
      clk => clk,
      clr => reset,
      q => r6 );
reg_7 : Reg
port map ( d => regIn,
      en => regEnSig(7),
      clk => clk,
      clr => reset,
      q => r7 );
end Behavioral;
```

Design Diagram



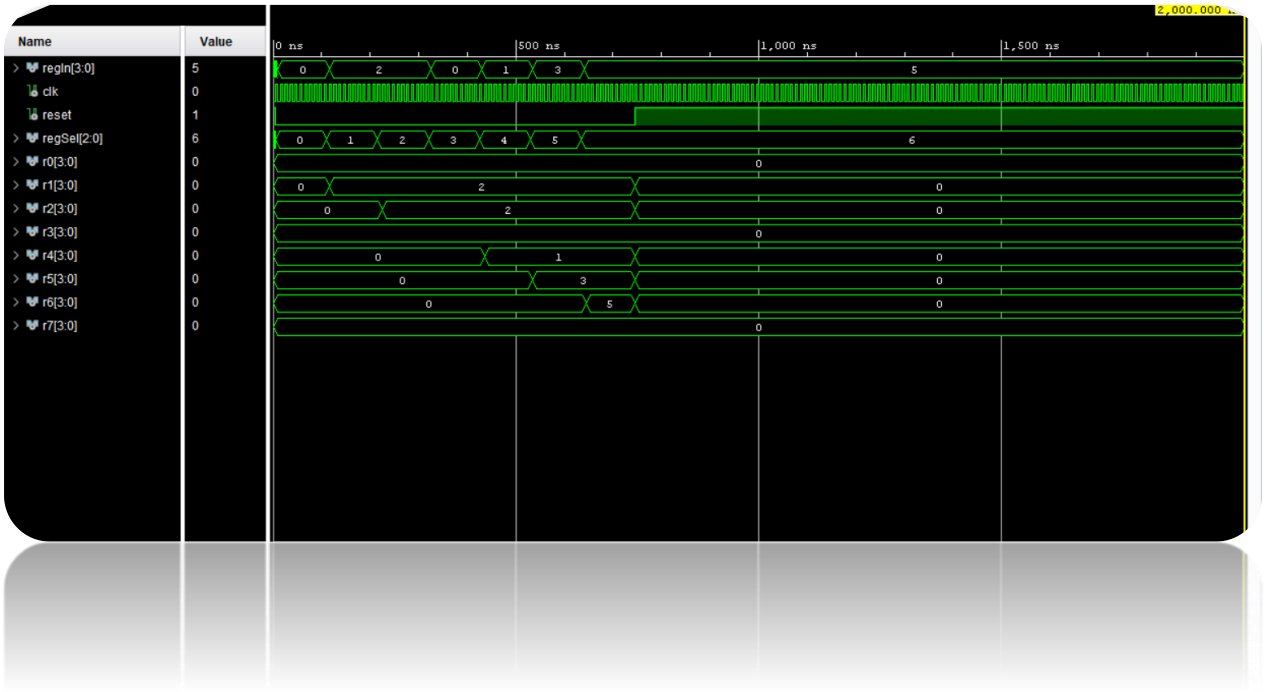
Test Bench File

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY TB_Reg_Bank IS
-- Port ( );
END TB_Reg_Bank;
ARCHITECTURE Behavioral OF TB_Reg_Bank IS
COMPONENT Reg_Bank
PORT ( regIn : IN STD_LOGIC_VECTOR (3 DOWNTO 0); -- Value to be written
regSel : IN STD_LOGIC_VECTOR (2 DOWNTO 0); -- Register address select
clk : IN STD_LOGIC;
reset : IN STD_LOGIC; -- Reset all registers to 0000
r0 : OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
r1 : OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
r2 : OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
r3 : OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
r4 : OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
r5 : OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
r6 : OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
r7 : OUT STD_LOGIC_VECTOR (3 DOWNTO 0) );
END COMPONENT;
SIGNAL regIn : STD_LOGIC_VECTOR (3 DOWNTO 0);
SIGNAL clk, reset : STD_LOGIC := '0';
SIGNAL regSel : STD_LOGIC_VECTOR (2 DOWNTO 0);
```

```
SIGNAL r0, r1, r2, r3, r4, r5, r6, r7 : STD_LOGIC_VECTOR (3 DOWNTO 0);
BEGIN
UUT : Reg_Bank
PORT MAP( regIn => regIn,
      clk => clk,
      regSel => regSel,
      reset => reset,
      r0 => r0,
      r1 => r1,
      r2 => r2,
      r3 => r3,
      r4 => r4,
      r5 => r5,
      r6 => r6,
      r7 => r7 );
PROCESS BEGIN
WAIT FOR 5ns;
clk <= NOT(clk);
END PROCESS;
PROCESS BEGIN
reset <= '1';
WAIT FOR 5ns;
reset <= '0';
-- Index Number : 220135N -- Index Number : 220155B -- Index Number :
220264H -- Index Number : 220361D
regSel <= "000";
WAIT FOR 5ns;
regIn <= "0000"; -- 0
WAIT FOR 100ns;
regSel <= "001";
WAIT FOR 5ns;
regIn <= "0010"; -- 2
WAIT FOR 100ns;
regSel <= "010";
WAIT FOR 5ns;
regIn <= "0010"; -- 2
WAIT FOR 100ns;
regSel <= "011";
WAIT FOR 5ns;
regIn <= "0000"; -- 0
WAIT FOR 100ns;
regSel <= "100";
WAIT FOR 5ns;
regIn <= "0001"; -- 1
WAIT FOR 100ns;
regSel <= "101";
WAIT FOR 5ns;
regIn <= "0011"; -- 3
WAIT FOR 100ns;
regSel <= "110";
WAIT FOR 5ns;
regIn <= "0101"; -- 5
WAIT FOR 5ns;
```

```
regSel <= "110";
WAIT FOR 100ns;
reset <= '1';
WAIT;
END PROCESS;
END Behavioral;
```

Timing Graph



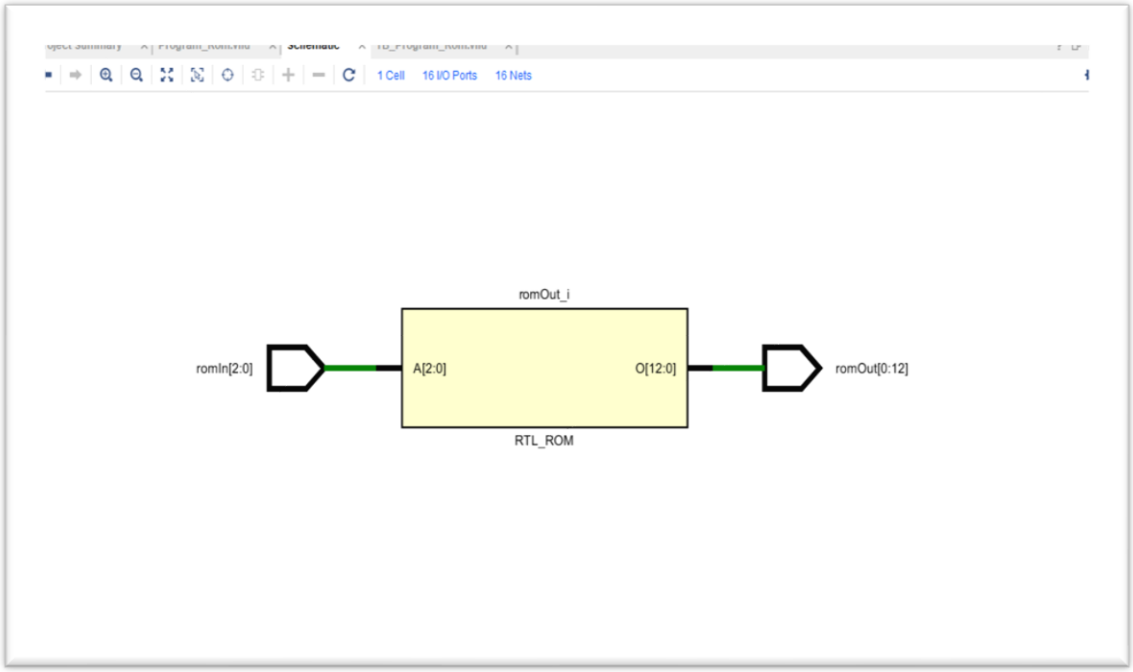
Program ROM

Design Source

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity Program_Rom is
      Port ( romIn : in STD_LOGIC_VECTOR (2 downto 0); -- ROM address input
      romOut : out STD_LOGIC_VECTOR (0 to 12) -- ROM data output );
end Program_Rom;
architecture Behavioral of Program_Rom is ---- Program that displays
numbers by decrementing 10 by 1
type RomType is array (0 to 7) of std_logic_vector(0 to 12);
      signal programRom : RomType := ( "0101110000110", -- 0 -- MOVI R7, 10
      "0100100000001", -- 1 -- MOVI R2, 1
      "0010100000000", -- 2 -- NEG R2
      "0001110100000", -- 3 -- ADD R7, R2
      "0111110000110", -- 4 -- JZR R7, 6
```

```
        "0110000000011", -- 5 -- JZR R0, 3
        "0110000000110", -- 6 -- JZR R0, 6
        "1110000000000" -- 7 -- END );
begin
romOut <= programRom(to_integer(unsigned(romIn)));
end Behavioral;
```

Design Diagram

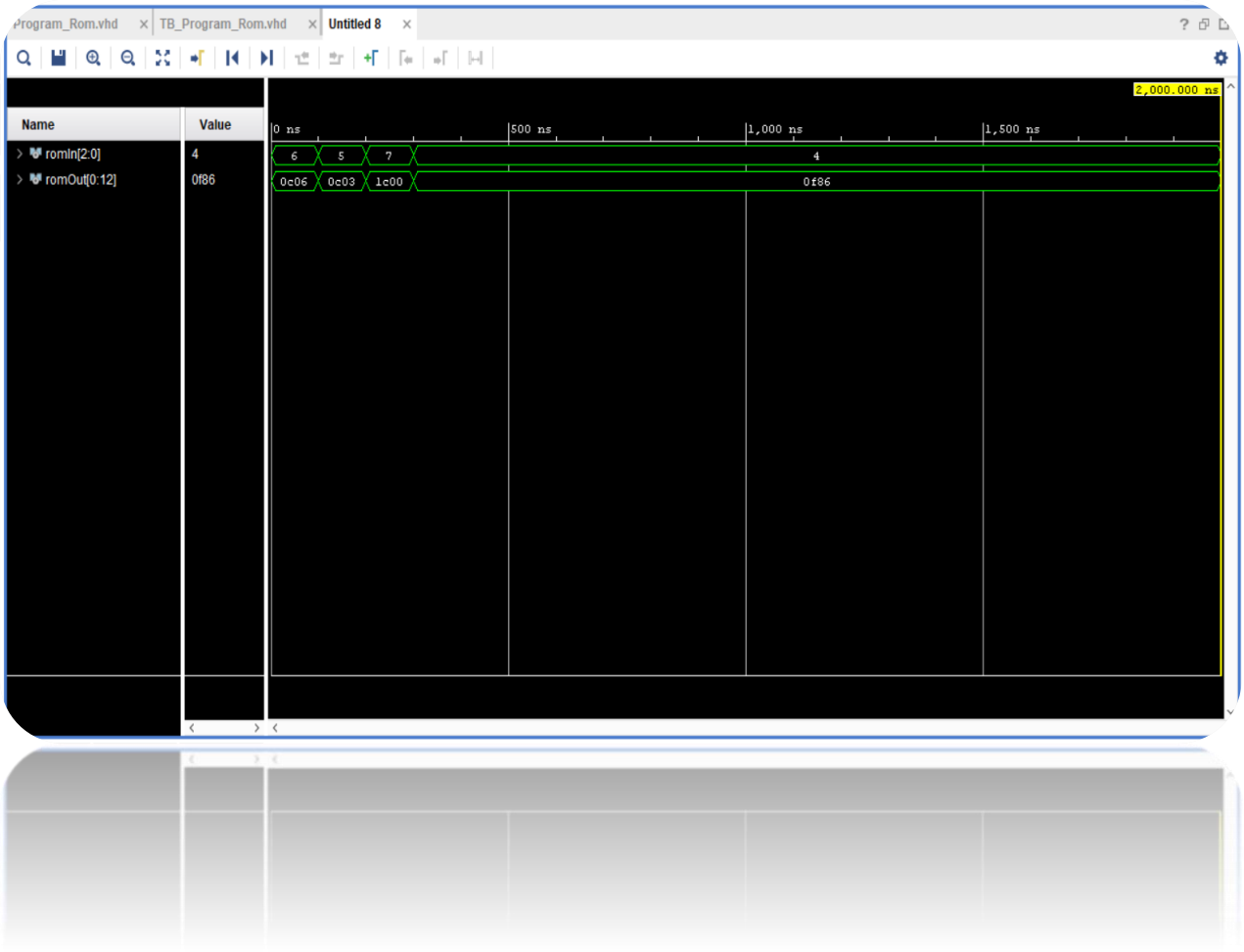


Test Bench File

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity TB_Program_Rom is
-- Port ( );
end TB_Program_Rom;
architecture Behavioral of TB_Program_Rom is
component Program_Rom
Port ( romIn : in STD_LOGIC_VECTOR (2 downto 0);
       romOut : out STD_LOGIC_VECTOR (0 to 12) );
end component;
signal romIn : STD_LOGIC_VECTOR (2 downto 0);
signal romOut : STD_LOGIC_VECTOR (0 to 12);
begin
UUT: Program_Rom
port map ( romIn => romIn,
       romOut => romOut );
process begin
```

```
-- Index number: 220135N = 110 101 101 111 100 111
romIn <= "110"; -- Set romIn to "110"
wait for 100ns;
romIn <= "101"; -- Set romIn to "101"
wait for 100ns;
romIn <= "111"; -- Set romIn to "111"
wait for 100ns;
romIn <= "100"; -- Set romIn to "100"
wait;
end process;
end Behavioral;
```

Timing Graph



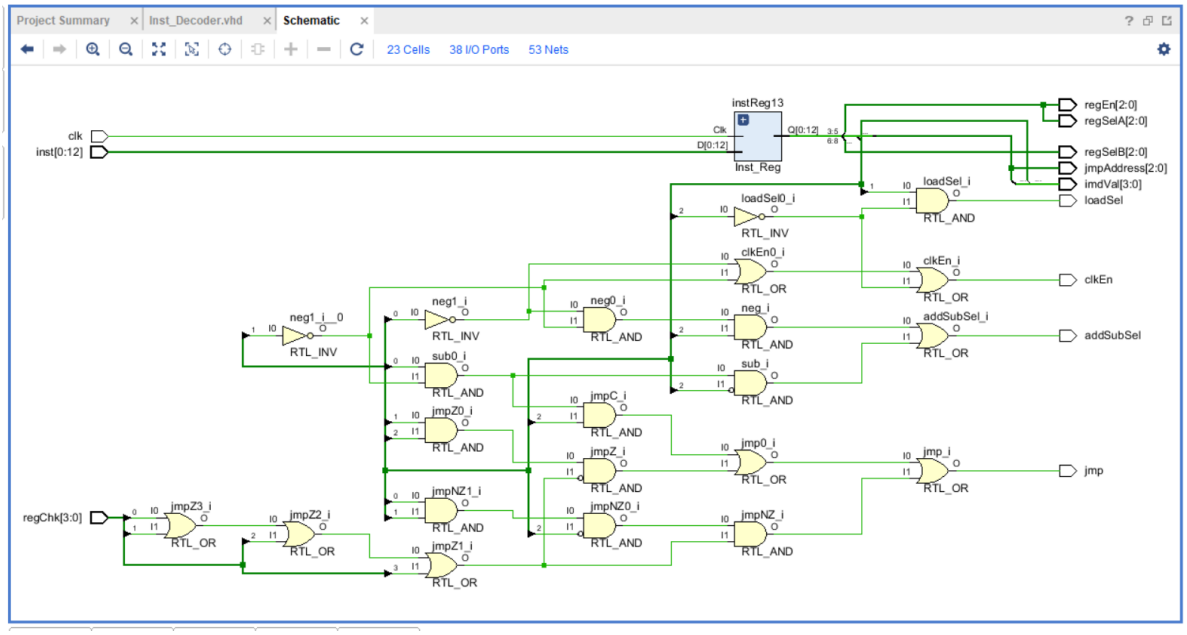
Instruction Decoder

Design Source

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Inst_Decoder is
Port ( inst: in STD_LOGIC_VECTOR (0 to 12); -- Instruction
clk: in STD_LOGIC; -- Clock signal
regChk: in STD_LOGIC_VECTOR (3 downto 0); -- Check register value for JZR
regSelA: out STD_LOGIC_VECTOR (2 downto 0);--Select register to load into
MUX A
regSelB: out STD_LOGIC_VECTOR (2 downto 0); -- Select register to load into
MUX B
imdVal: out STD_LOGIC_VECTOR (3 downto 0); -- Immediate value
regEn: out STD_LOGIC_VECTOR (2 downto 0); -- Enable register for write
loadSel: out STD_LOGIC; -- Choose between Imd value or Add/Sub Unit result
addSubSel: out STD_LOGIC; -- Add Sub selector
clkEn: out STD_LOGIC; -- Enable/Disable clock
jmp: out STD_LOGIC; -- Jump flag
jmpAddress: out STD_LOGIC_VECTOR (2 downto 0) -- Address to jump );
end Inst_Decoder;

architecture Behavioral of Inst_Decoder is
component Inst_Reg
Port ( d: in STD_LOGIC_VECTOR (0 to 12);
      clk: in STD_LOGIC;
      q: out STD_LOGIC_VECTOR (0 to 12) );
end component;
signal i: STD_LOGIC_VECTOR (0 to 12);
signal jmpZ, jmpNZ, jmpC, sub, neg: STD_LOGIC;
begin
-- 13-bit instruction register
instReg13: Inst_Reg
port map ( d => inst,
      clk => clk,
      q => i );
-- Mapping 1st three bits of instruction -
--010 - MOV
-- 000 - ADD
-- 001 - NEG
-- 011 - JZR
-- 100 - SUB
-- 101 - JMP
-- 110 - JNZR
-- 111 - END
sub <= i(0) AND NOT i(1) AND NOT i(2); -- Output is 1 when opcode is 100
neg <= NOT i(0) AND NOT i(1) AND i(2); -- Output is 1 when opcode is 001
addSubSel <= neg OR sub;
loadSel <= i(1) AND NOT i(2); -- Output is 1 when opcode is 010
```

```
clkEn <= NOT i(0) OR NOT i(1) OR NOT i(2); -- Output is 0 when opcode is
111
jmpC <= i(0) AND NOT i(1) AND i(2); -- Output is 1 when opcode is 101
jmpZ <= (i(1) AND i(2)) AND NOT (regChk(0) OR regChk(1) OR regChk(2) OR
regChk(3)); -- Output is 1 when opcode is 011 and checked register value is
0000
jmpNZ <= (i(0) AND i(1) AND NOT i(2)) AND (regChk(0) OR regChk(1) OR
regChk(2) OR regChk(3)); -- Output is 1 when opcode is 110 and checked
register value is not 0000
jmp <= jmpC OR jmpZ OR jmpNZ; -- Mapping bits for register A and B
regSelA <= i(3) & i(4) & i(5);
regSelB <= i(6) & i(7) & i(8);
regEn <= i(3) & i(4) & i(5); -- Mapping last 4 bits
imdVal <= i(9) & i(10) & i(11) & i(12); -- Value to load if MOVI
instruction is used
jmpAddress <= i(10) & i(11) & i(12); -- Address to jump to if JMP
instruction is used
end Behavioral;
```

Design Diagram

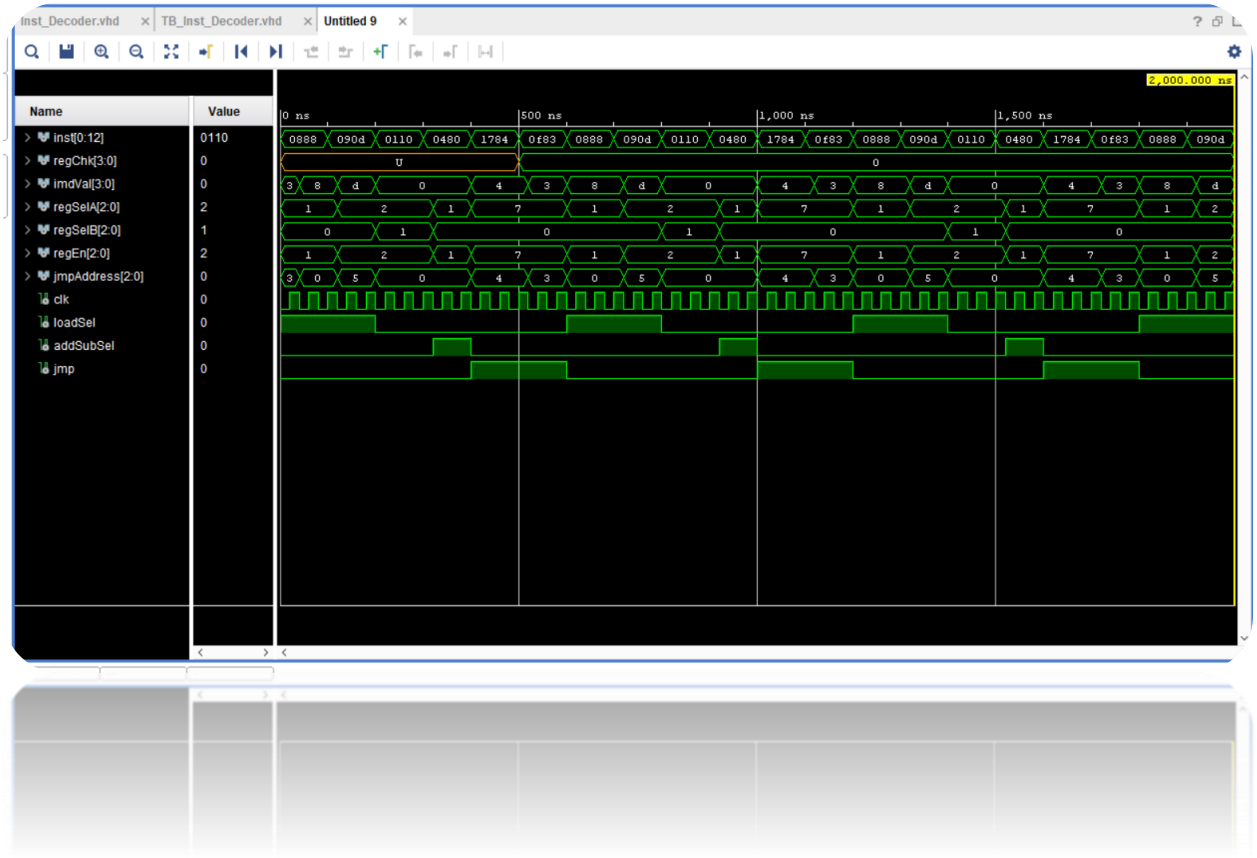


Test Bench File

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity TB_Inst_Decoder is
-- Port ( );
end TB_Inst_Decoder;
architecture Behavioral of TB_Inst_Decoder is
component Inst_Decoder
Port ( inst : in STD_LOGIC_VECTOR (0 to 12); -- Instruction
clk : in STD_LOGIC;
regChk : in STD_LOGIC_VECTOR (3 downto 0); -- Check register value for JZR
regSelA : out STD_LOGIC_VECTOR (2 downto 0);
regSelB : out STD_LOGIC_VECTOR (2 downto 0);
imdVal : out STD_LOGIC_VECTOR (3 downto 0); -- Immediate value
regEn : out STD_LOGIC_VECTOR (2 downto 0); -- Enable register for write
loadSel : out STD_LOGIC; -- Choose between Imd value or Add/Sub Unit result
addSubSel : out STD_LOGIC; -- Add Sub selector
jmp : out STD_LOGIC; -- Jump flag
jmpAddress : out STD_LOGIC_VECTOR (2 downto 0) -- Address to jump );
end component;
signal inst : STD_LOGIC_VECTOR (0 to 12);
signal regChk, imdVal : STD_LOGIC_VECTOR (3 downto 0);
signal regSelA, regSelB, regEn, jmpAddress : STD_LOGIC_VECTOR (2 downto 0);
signal clk, loadSel, addSubSel, jmp : STD_LOGIC := '0';
begin
uut: Inst_Decoder
port map ( inst => inst,
      clk => clk,
      regChk => regChk,
      regSelA => regSelA,
      regSelB => regSelB,
      imdVal => imdVal,
      regEn => regEn,
      loadSel => loadSel,
      addSubSel => addSubSel,
      jmp => jmp,
      jmpAddress => jmpAddress );
process begin
wait for 20ns;
clk <= not clk;
end process;
process begin
-- Index number : 220135N = 110101101111100111
-- Index number : 220155B = 110101101111111011
-- Index number : 220264H = 110101110001101000
-- Index number : 220361D = 110101110011001001
--From the index number binary form we get 13 bits to create a instruction
from 2nd index
inst <= "0100010001000"; -- MOV 8 to Reg 1
wait for 100ns;
```

```
inst <= "0100100001101"; -- MOV -3 to Reg 2
wait for 100ns;
inst <= "0000100010000"; -- ADD Reg 2 and Reg 1
wait for 100ns;
inst <= "0010010000000"; -- NEG Reg 1
wait for 100ns;
inst <= "1011110000100"; -- JMP Reg to Inst 4
wait for 100ns;
inst <= "0111110000011"; -- (JZR) JMP to Inst 3 if Reg 7 is zero
regChk <= "0000";
wait for 100ns;
end process;
end Behavioral;
```

Timing Graph

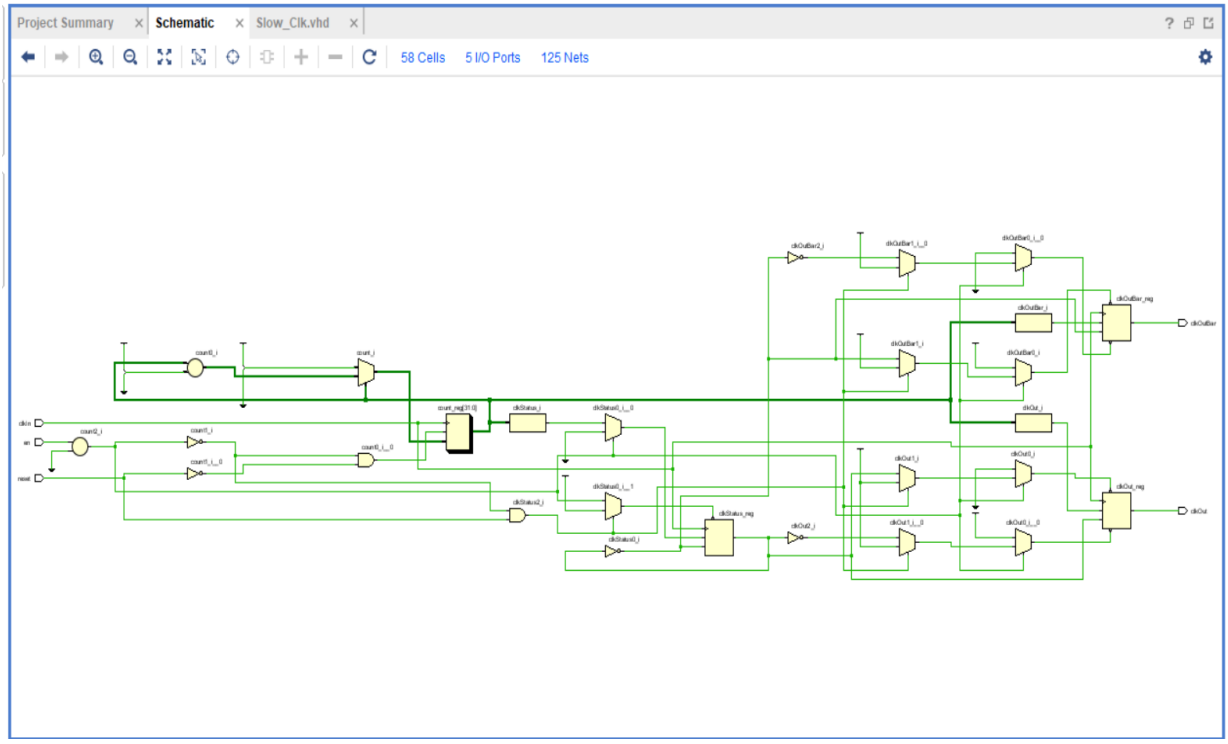


Slow Clock

Design Source

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Slow_Clk is
Port ( clkIn : in STD_LOGIC := '0';
       en : in STD_LOGIC := '1';
       reset : in STD_LOGIC;
       clkOut : out STD_LOGIC := '0';
       clkOutBar : out STD_LOGIC := '1' );
end Slow_Clk;
architecture Behavioral of Slow_Clk is
signal count : integer := 1;
signal clkStatus : STD_LOGIC := '0';
begin
process (clkIn, reset)
begin
if (en = '0') then
       clkOut <= '0';
       clkOutBar <= '1';
elsif (reset = '1') then
       clkStatus <= '0';
       clkOut <= clkStatus;
       clkOutBar <= not clkStatus;
elsif (rising_edge(clkIn)) then
       count <= count + 1; -- if(count = 100000000) then -- For Basys3 board
       if(count = 5) then -- For simulation in Vivado
              clkStatus <= not clkStatus;
              clkOut <= clkStatus;
              clkOutBar <= not clkStatus;
              count <= 1;
       end if;
end if;
end process;
end Behavioral;
```

Design Diagram

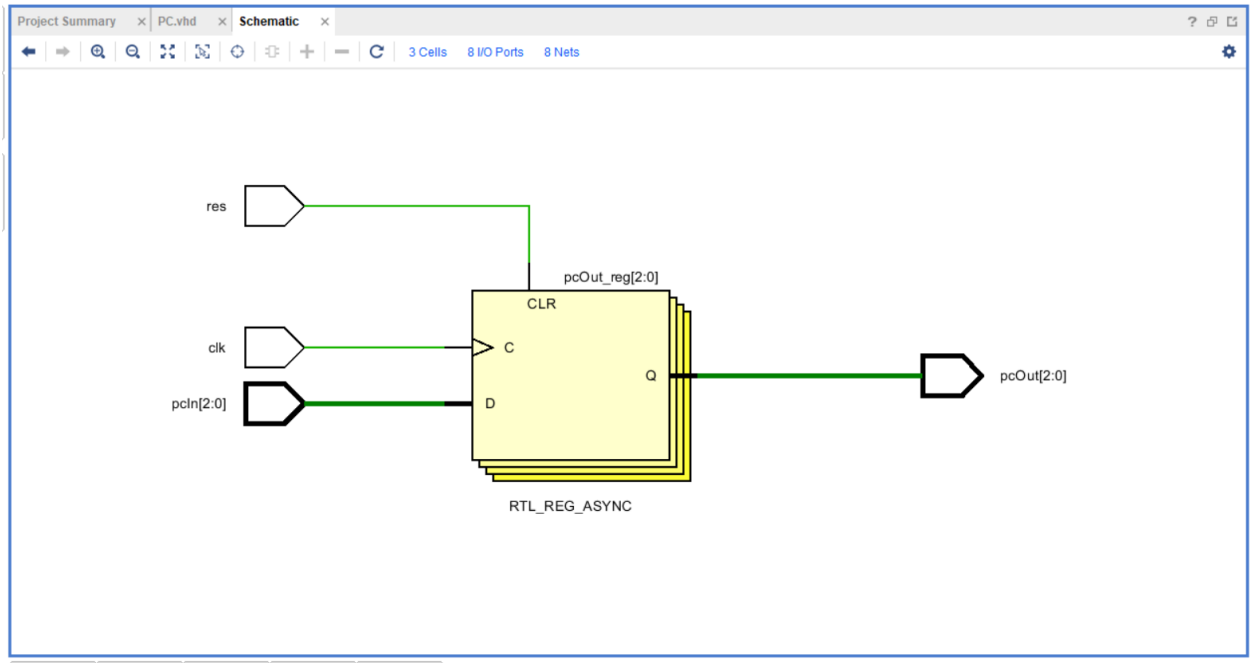


Program Counter

Design Source

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity PC is Port ( pcIn : in STD_LOGIC_VECTOR (2 downto 0) := "000";
      clk : in STD_LOGIC;
      res : in STD_LOGIC;
      pcOut : out STD_LOGIC_VECTOR (2 downto 0) := "000" );
end PC;
architecture Behavioral of PC is
begin
process (clk, res)
begin
if (res = '1') then
      pcOut <= "000";
elsif (rising_edge(clk)) then
      pcOut <= pcIn;
end if;
end process;
end Behavioral;
```

Design Diagram

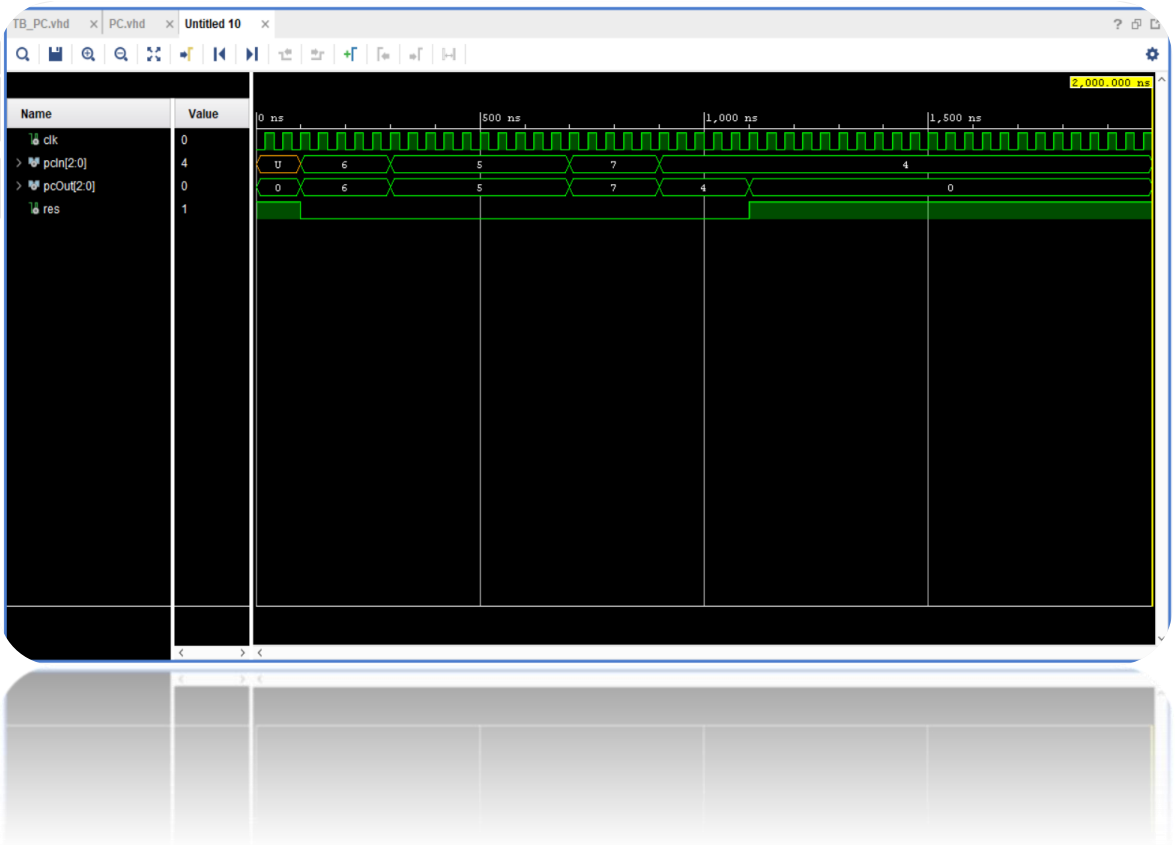


Test Bench File

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity TB_PC is
-- Port ( );
end TB_PC;
architecture Behavioral of TB_PC is
component PC
Port ( pcIn : in STD_LOGIC_VECTOR (2 downto 0);
       clk : in STD_LOGIC;
       res : in STD_LOGIC;
       pcOut : out STD_LOGIC_VECTOR (2 downto 0) );
end component;
signal clk : STD_LOGIC := '0';
signal pcIn, pcOut : STD_LOGIC_VECTOR (2 downto 0);
signal res : STD_LOGIC;
begin
UUT: PC
port map ( pcIn => pcIn,
       clk => clk,
       res => res,
       pcOut => pcOut );
process
begin
wait for 20ns;
```

```
clk <= NOT clk;
end process;
process
begin
-- Index number: 220135N = 110 101 101 111 100 111
res <= '1'; -- Set res to 1
wait for 100ns;
res <= '0'; -- Clear res
pcIn <= "110"; -- Set pcIn to "110"
wait for 200ns;
pcIn <= "101"; -- Set pcIn to "101"
wait for 200ns;
pcIn <= "101"; -- Set pcIn to "101"
wait for 200ns;
pcIn <= "111"; -- Set pcIn to "111"
wait for 200ns;
pcIn <= "100"; -- Set pcIn to "100"
wait for 200ns;
res <= '1'; -- Set res to 1
wait ;
end process;
end Behavioral;
```

Timing Graph

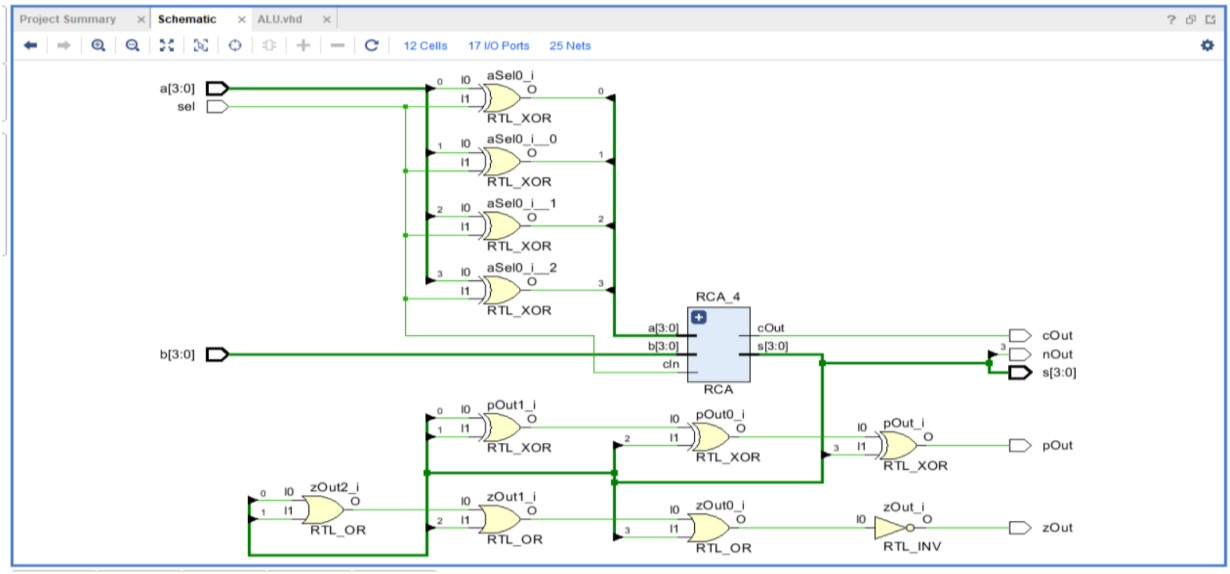


4-bit Adder/Substractor

Design Source

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity ALU is
port ( a : in STD_LOGIC_VECTOR (3 downto 0);
       b : in STD_LOGIC_VECTOR (3 downto 0);
       sel : in STD_LOGIC; -- Add/Sub Selector (sel=0 for add, sel=1 for
subtract)
       s : out STD_LOGIC_VECTOR (3 downto 0);
       cOut : out STD_LOGIC; -- Carry flag
       zOut : out STD_LOGIC; -- Zero flag
       nOut : out STD_LOGIC; -- Negative flag
       pOut : out STD_LOGIC -- Parity flag (odd parity detector) );
end ALU;
architecture Behavioral of ALU is
component RCA
port ( a : in STD_LOGIC_VECTOR (3 downto 0);
       b : in STD_LOGIC_VECTOR (3 downto 0);
       cIn : in STD_LOGIC;
       s : out STD_LOGIC_VECTOR (3 downto 0);
       cOut : out STD_LOGIC );
end component;
signal aSel, sOut: STD_LOGIC_VECTOR (3 downto 0);
begin
aSel(0) <= a(0) XOR sel;
aSel(1) <= a(1) XOR sel;
aSel(2) <= a(2) XOR sel;
aSel(3) <= a(3) XOR sel;
RCA_4: RCA
port map ( a =>aSel ,
       b =>b,
       cIn => sel,
       s => sOut,
       cOut => cOut );
s <= sOut;
zOut <= NOT (sOut(0) OR sOut(1) OR sOut(2) OR sOut(3));
nOut <= sOut(3);
pOut <= sOut(0) XOR sOut(1) XOR sOut(2) XOR sOut(3);
end Behavioral;
```

Design Diagram

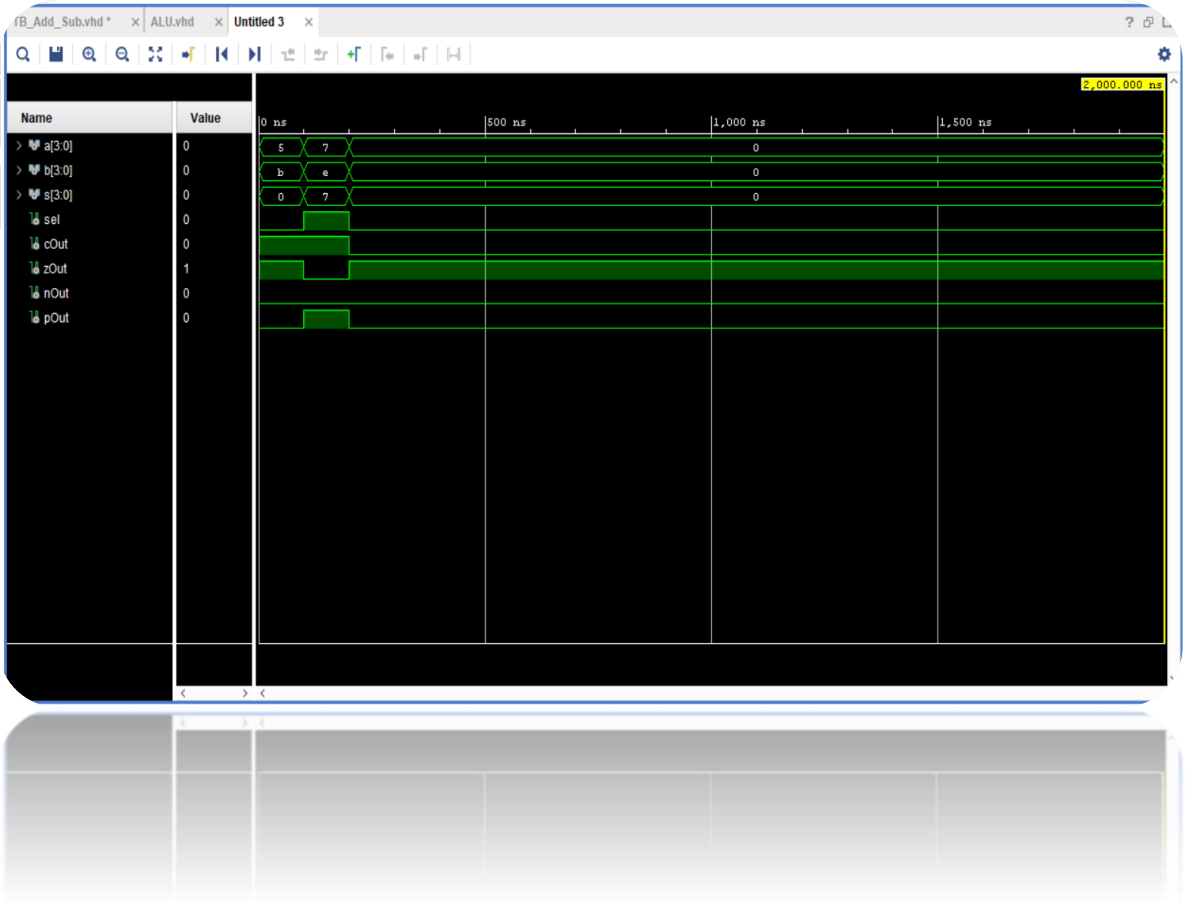


Test Bench File

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity TB_ALU is end TB_ALU;
architecture Behavioral of TB_ALU is
component ALU
Port ( a : in STD_LOGIC_VECTOR (3 downto 0);
       b : in STD_LOGIC_VECTOR (3 downto 0);
       sel : in STD_LOGIC; -- Add/Sub Selector
       s : out STD_LOGIC_VECTOR (3 downto 0);
       cOut : out STD_LOGIC; -- Carry flag
       zOut : out STD_LOGIC; -- Zero flag
       nOut : out STD_LOGIC; -- Negative flag
       pOut : out STD_LOGIC -- Parity flag );
end component;
signal a, b, s : STD_LOGIC_VECTOR (3 downto 0);
signal sel, cOut, zOut, nOut, pOut: STD_LOGIC;
begin
UUT: ALU
PORT MAP( a => a,
       b => b,
       sel => sel,
       s => s,
       cOut => cOut,
       zOut => zOut,
       nOut => nOut,
```

```
      pOut => pOut );
process
begin
-- sel = 0 : Adder
-- sel = 1 : Subtractor (Negates a)
-- Index number : 220135N = 11 0101 1011 1110 0111
-- 0101 + 1011 = 5 + 11 = 16 == 0 in 4 bit
sel <= '0';
a <= "0101";
b <= "1011";
wait for 100ns; -- 1110 - 0111 = 14 - 7 = 7 subtract a from B
sel <= '1';
a <= "0111";
b <= "1110";
wait for 100ns;
-- Reset
sel <= '0';
a <= "0000";
b <= "0000";
wait;
end process;
end Behavioral;
```

Timing Graph

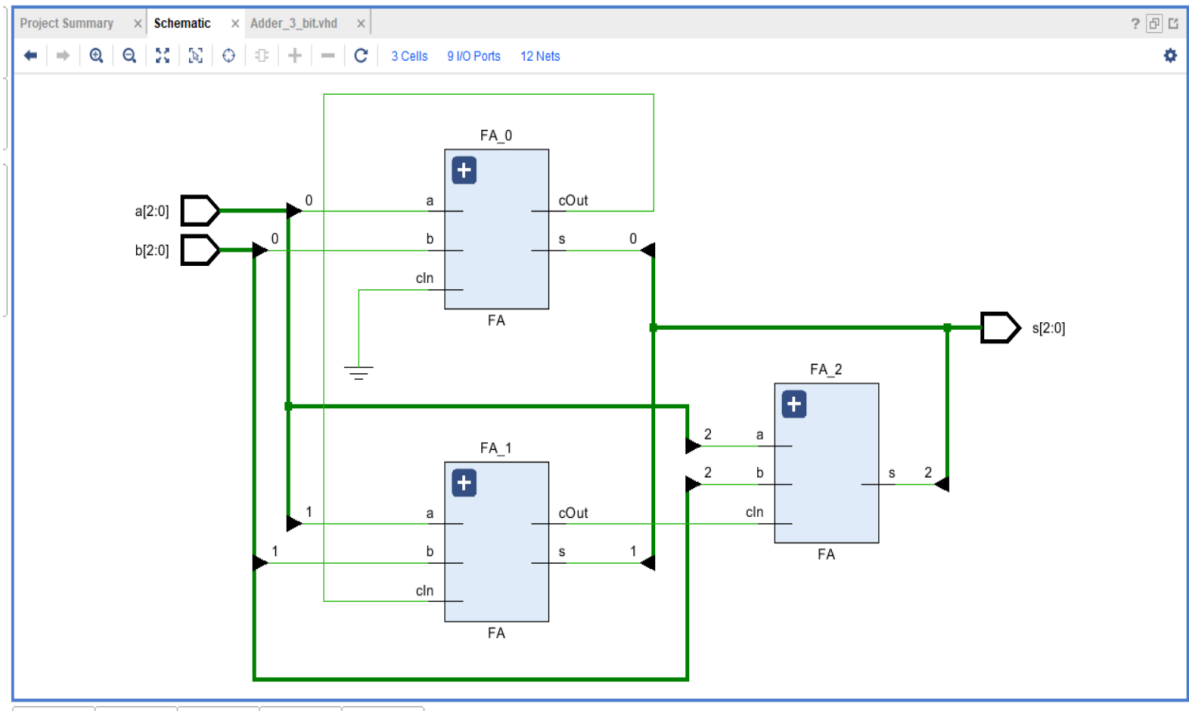


3-bit Adder

Design Source

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Adder_3_bit is
Port ( a : in STD_LOGIC_VECTOR (2 downto 0); -- Input A
       b : in STD_LOGIC_VECTOR (2 downto 0); -- Input B
       s : out STD_LOGIC_VECTOR (2 downto 0) -- Output S );
end Adder_3_bit;
architecture Behavioral of Adder_3_bit is
component FA
Port ( a : in STD_LOGIC; -- Input A
       b : in STD_LOGIC; -- Input B
       cIn : in STD_LOGIC; -- Carry-in
       s : out STD_LOGIC; -- Sum output
       cOut : out STD_LOGIC -- Carry-out );
end component;
signal fa0_c, fa1_c, fa2_c : STD_LOGIC; -- Internal carry signals
begin
-- Full Adder 0
FA_0 : FA
port map ( a => a(0),
       b => b(0),
       cIn => '0', -- Set to ground (no initial carry)
       s => s(0),
       cOut => fa0_c );
-- End FA_0
-- Full Adder 1
FA_1 : FA
port map ( a => a(1),
       b => b(1),
       cIn => fa0_c, -- Carry from FA_0
       s => s(1),
       cOut => fa1_c );
-- End FA_1
-- Full Adder 2
FA_2 : FA
port map ( a => a(2),
       b => b(2),
       cIn => fa1_c, -- Carry from FA_1
       s => s(2),
       cOut => fa2_c );
-- End FA_2
end Behavioral;
```

Design Diagram

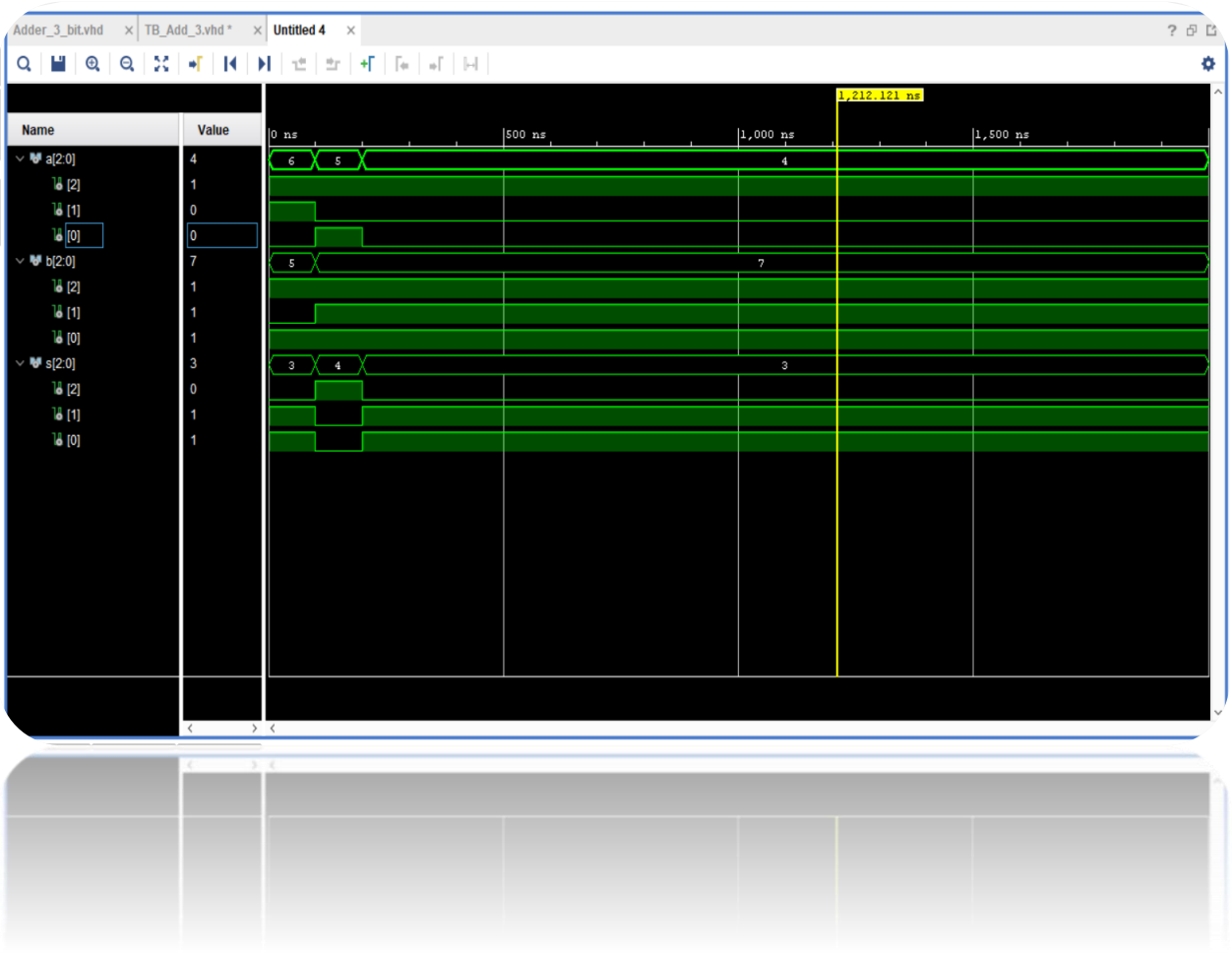


Test Bench File

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity TB_Add_3 is
-- Port ( );
end TB_Add_3;
architecture Behavioral of TB_Add_3 is
component Adder_3_bit
port ( a : in STD_LOGIC_VECTOR (2 downto 0); -- Input A
       b : in STD_LOGIC_VECTOR (2 downto 0); -- Input B
       s : out STD_LOGIC_VECTOR (2 downto 0) -- Sum output );
end component;
signal a, b, s : STD_LOGIC_VECTOR (2 downto 0) := "000"; -- Input and
output signals
begin
UUT: Adder_3_bit
port map ( a => a,
       b => b,
       s => s );
process
begin
-- Test Cases
```

```
-- Index number : 220135N = 110 101 101 111 100 111
-- 110 + 101 = 6 + 5 = 11 --> overloaded --> 3
a <= "110";
b <= "101";
wait for 100ns;
-- 101 + 111 = 5 + 7 = 12 --> overloaded --> 3
a <= "101";
b <= "111";
wait for 100ns;
-- 100 + 111 = 4 + 7 = 11 --> overloaded --> 3
a <= "100";
b <= "111";
wait;
end process;
end Behavioral;
```

Timing Graph

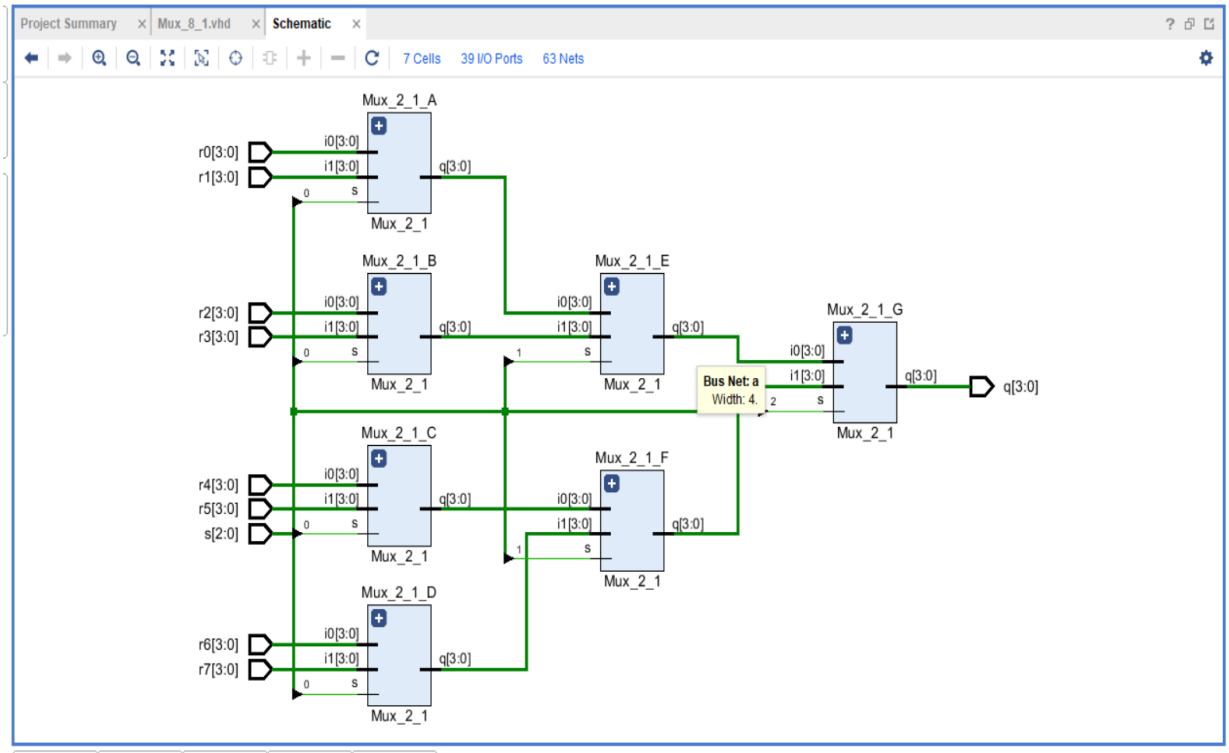


Mux 8-way 4-bit

Design Source

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
-- 8 way mux using 2 way bux all in 4 bits
entity Mux_8_1 is
Port ( s : in STD_LOGIC_VECTOR (2 downto 0);
       r0,r1,r2,r3,r4,r5,r6,r7 : in std_logic_vector (3 downto 0);
       q : out STD_LOGIC_vector(3 downto 0));
end Mux_8_1;
architecture Behavioral of Mux_8_1 is
component Mux_2_1 is
Port ( i0 : in STD_LOGIC_VECTOR (3 downto 0);
       i1 : in STD_LOGIC_VECTOR (3 downto 0);
       s : in STD_LOGIC;
       q : out STD_LOGIC_VECTOR (3 downto 0));
end component;
SIGNAL d01, d23, d45, d67, a, b:STD_LOGIC_VECTOR(3 DOWNTO 0);
begin
Mux_2_1_A:Mux_2_1
port map(
r0,r1,s(0),d01);

Mux_2_1_B:Mux_2_1
port map(r2,r3,s(0),d23);

Mux_2_1_C:Mux_2_1
port map(r4,r5,s(0),d45);
Mux_2_1_D:Mux_2_1
port map(r6,r7,s(0),d67);

Mux_2_1_E:Mux_2_1
port map(d01,d23,s(1),a);

Mux_2_1_F:Mux_2_1
port map(d45,d67,s(1),b);

Mux_2_1_G:Mux_2_1
port map(a,b,s(2),q);
end Behavioral;
```

Design Diagram

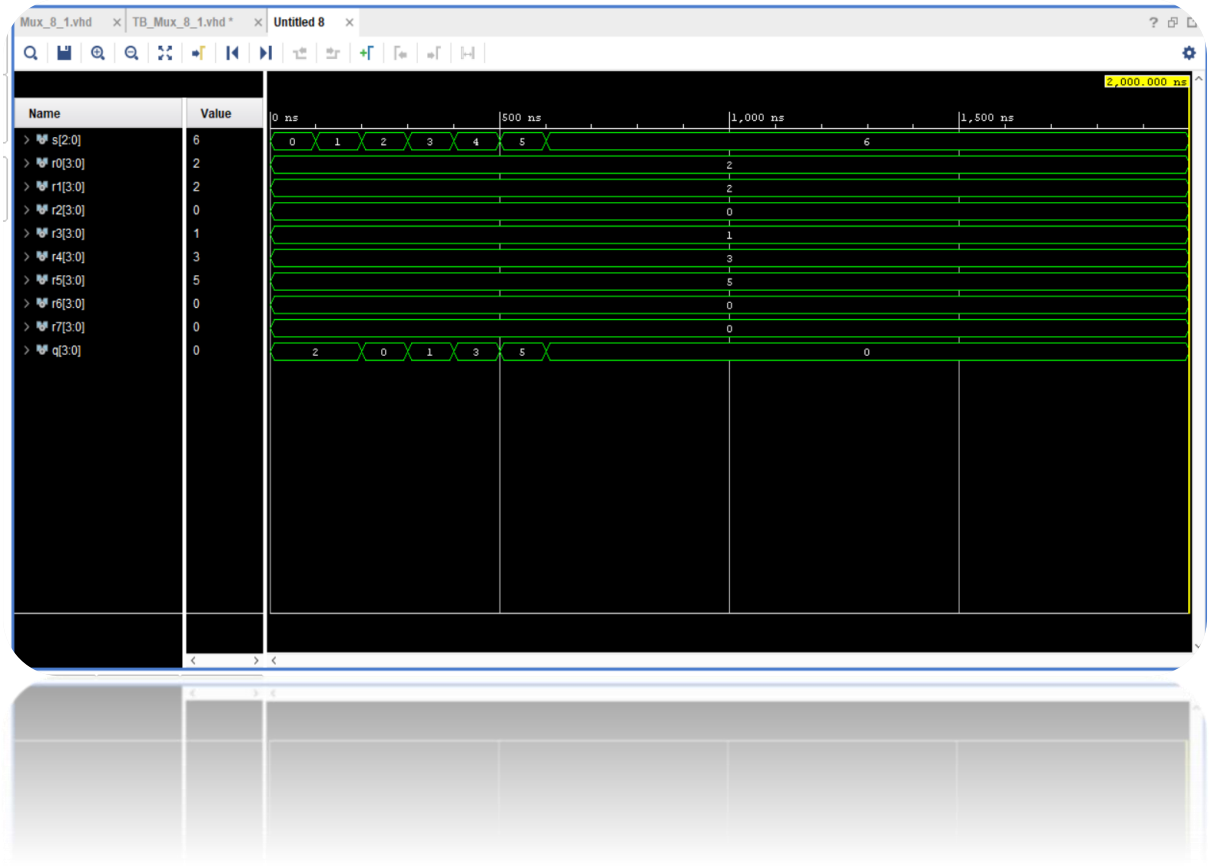


Test Bench File

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity TB_Mux_8_1 is
-- Port ( );
end TB_Mux_8_1;
architecture Behavioral of TB_Mux_8_1 is
component Mux_8_1
Port ( s : in STD_LOGIC_VECTOR (2 downto 0);
       r0,r1,r2,r3,r4,r5,r6,r7 : in std_logic_vector (3 downto 0);
       q : out STD_LOGIC_vector(3 downto 0));
end component;
signal s : STD_LOGIC_VECTOR (2 downto 0);
signal r0,r1,r2,r3,r4,r5,r6,r7 : std_logic_vector (3 downto 0);
signal q : STD_LOGIC_vector(3 downto 0);
begin
UUT: Mux_8_1
PORT MAP(s,r0,r1,r2,r3,r4,r5,r6,r7,q);
PROCESS BEGIN
--Index number is 220135N
r0 <= "0010";
```

```
r1 <= "0010";
r2 <= "0000";
r3 <= "0001";
r4 <= "0011";
r5 <= "0101";
r6 <= "0000";
r7 <= "0000";
s <= "000";
WAIT FOR 100ns;
s <= "001";
WAIT FOR 100ns;
s <= "010";
WAIT FOR 100ns;
s <= "011";
WAIT FOR 100ns;
s <= "100";
WAIT FOR 100ns;
s <= "101";
WAIT FOR 100ns;
s <= "110";
WAIT;
END PROCESS;
end Behavioral;
```

Timing Graph

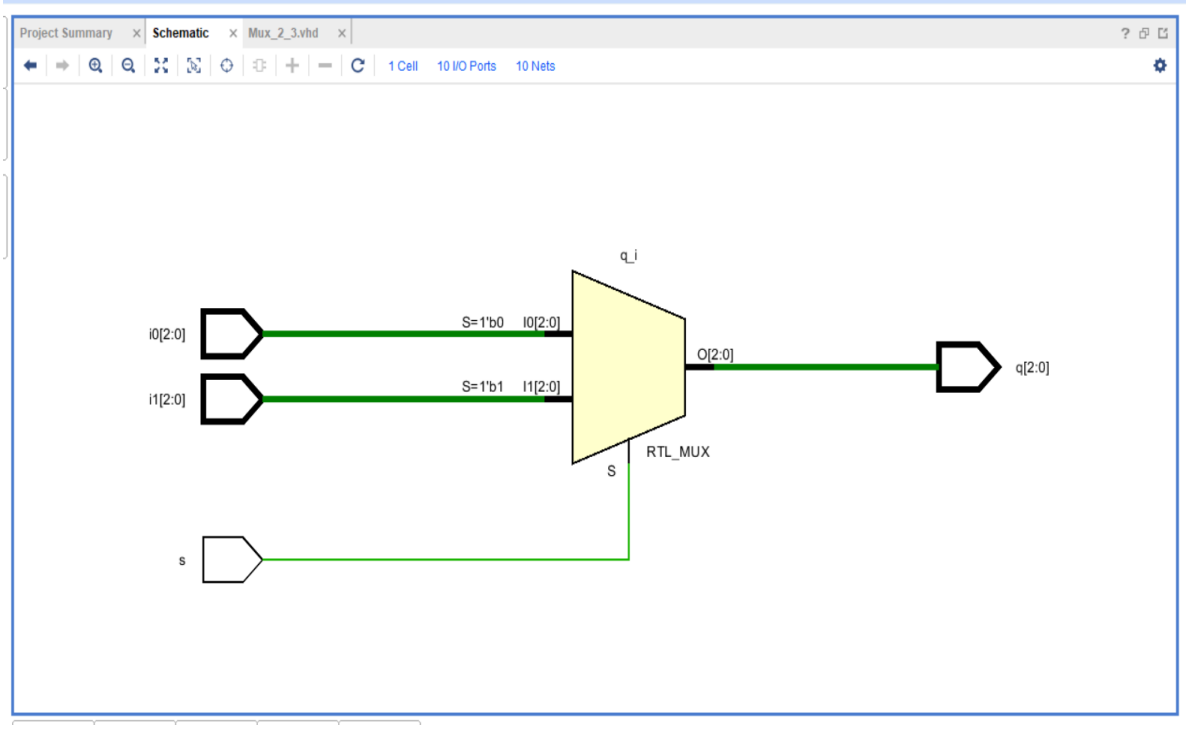


Mux 2-way 3-bit

Design Source

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Mux_2_3 is
Port ( i0 : in STD_LOGIC_VECTOR (2 downto 0);
       i1 : in STD_LOGIC_VECTOR (2 downto 0);
       s : in STD_LOGIC;
       q : out STD_LOGIC_VECTOR (2 downto 0));
end Mux_2_3;
architecture Behavioral of Mux_2_3 is
begin
       process(i0,i1,s)
       begin
             case s is
                   when '0' => q <= i0;
                   when '1' => q <= i1;
                   when others => q <= "ZZZ";
             end case;
       end process;
end Behavioral;
```

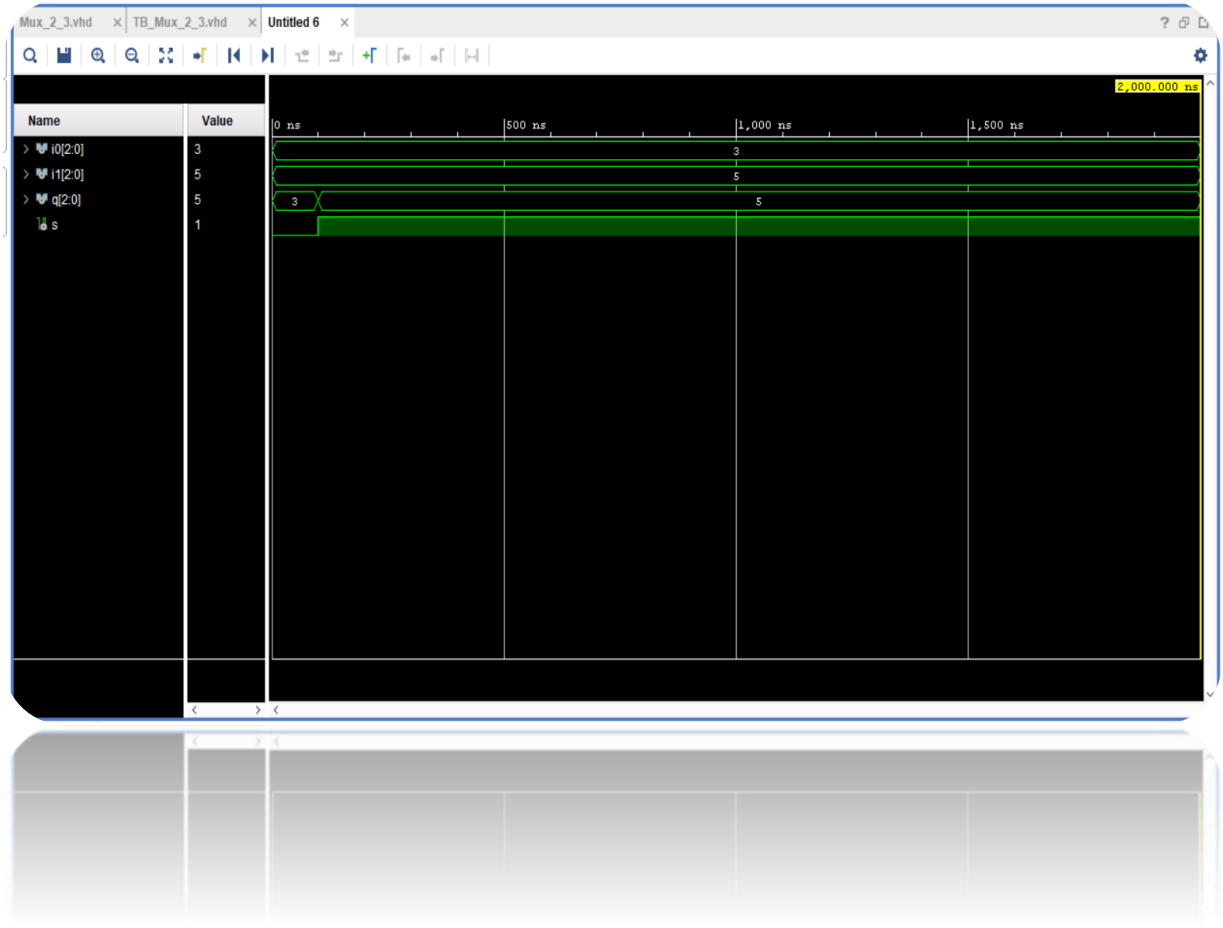
Design Diagram



Test Bench File

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity TB_Mux_2_3 is
-- Port ( );
end TB_Mux_2_3;
architecture behavioral of TB_Mux_2_3 is
component mux_2_3
      port ( i0 : in STD_LOGIC_VECTOR (2 downto 0);
             i1 : in STD_LOGIC_VECTOR (2 downto 0);
             s : in STD_LOGIC;
             q : out STD_LOGIC_VECTOR (2 downto 0));
end component;
signal i0, i1, q : STD_LOGIC_VECTOR (2 downto 0);
signal s : STD_LOGIC;
begin
uut: mux_2_3
port map( i0 => i0, i1 => i1, s => s, q => q );
-- Inputs
i0 <= "011";
i1 <= "101";
process
begin
s <= '0'; -- i0 is selected
wait for 100ns;
s <= '1'; -- i1 is selected
wait;
end process;
end behavioral;
```

Timing Graph



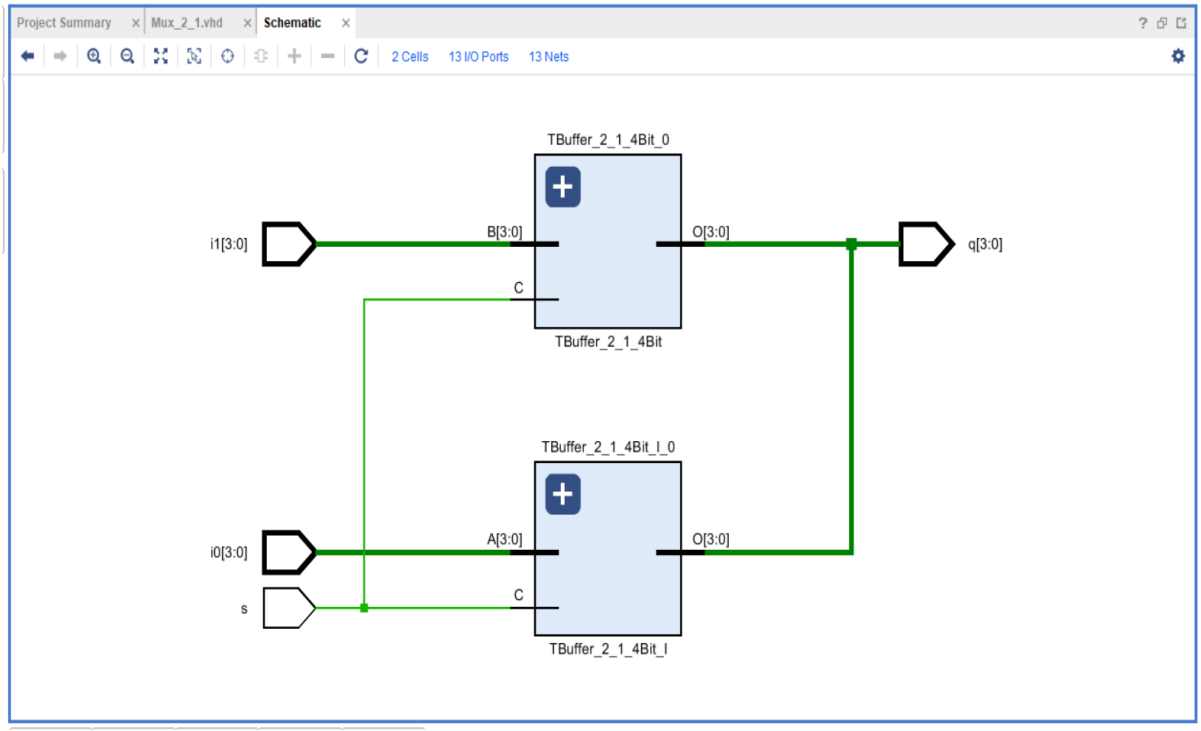
Mux 2-way 4-bit

Design Source

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Mux_2_1 is
Port ( i0 : in STD_LOGIC_VECTOR (3 downto 0);
       i1 : in STD_LOGIC_VECTOR (3 downto 0);
       s : in STD_LOGIC;
       q : out STD_LOGIC_VECTOR (3 downto 0));
end Mux_2_1;
architecture Behavioral of Mux_2_1 is
component TBuffer_2_1_4Bit_I is
Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
       C : in STD_LOGIC;
       O : out STD_LOGIC_VECTOR (3 downto 0));
end component;
```

```
component TBuffer_2_1_4Bit is
Port ( B : in STD_LOGIC_VECTOR (3 downto 0);
       C : in STD_LOGIC;
       O : out STD_LOGIC_VECTOR (3 downto 0));
end component;
signal A_out,B_out:STD_LOGIC_VECTOR (3 downto 0);
begin
TBuffer_2_1_4Bit_I_0: TBuffer_2_1_4Bit_I
port map(i0,s,A_out);

TBuffer_2_1_4Bit_0: TBuffer_2_1_4Bit
PORT MAP(i1,s,B_out);

q<= A_out;
q<= B_out;
end Behavioral;
```

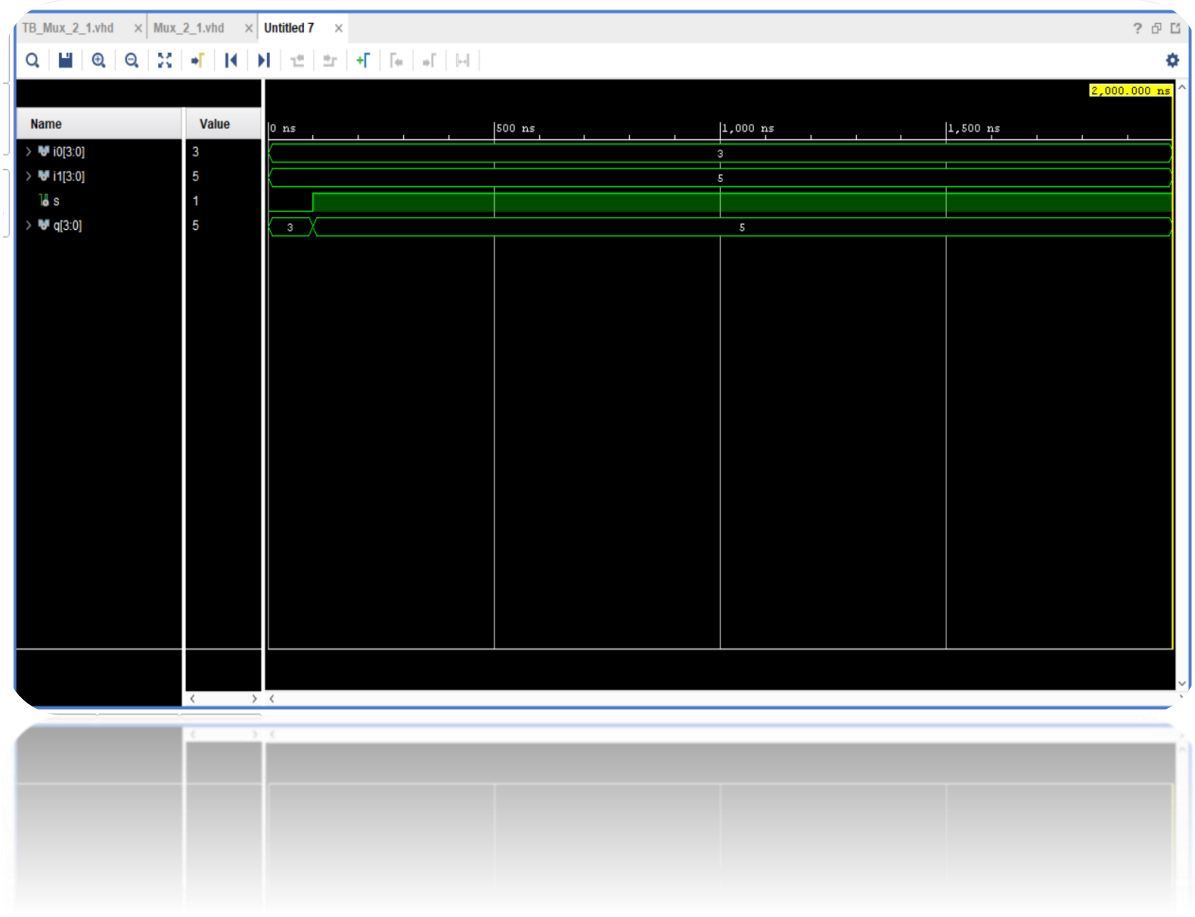
Design Diagram



Test Bench File

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity TB_Mux_2_1 is
-- Port ( );
end TB_Mux_2_1;
architecture Behavioral of TB_Mux_2_1 is
component Mux_2_1
      Port ( i0 : in STD_LOGIC_VECTOR (3 downto 0);
             i1 : in STD_LOGIC_VECTOR (3 downto 0);
             s : in STD_LOGIC;
             q : out STD_LOGIC_VECTOR (3 downto 0));
      end component;
signal i0 : STD_LOGIC_VECTOR (3 downto 0);
signal i1 : STD_LOGIC_VECTOR (3 downto 0);
signal s : STD_LOGIC;
signal q : STD_LOGIC_VECTOR (3 downto 0);
begin
UUT : Mux_2_1
port map (i0,i1,s,q);
process
begin
--INDEX Number = 220135N --> 3 & 5 used
i0 <= "0011";
i1 <= "0101";
s <= '0';
wait for 100ns;
s <= '1';
wait;
end process;
end Behavioral;
```

Timing Graph



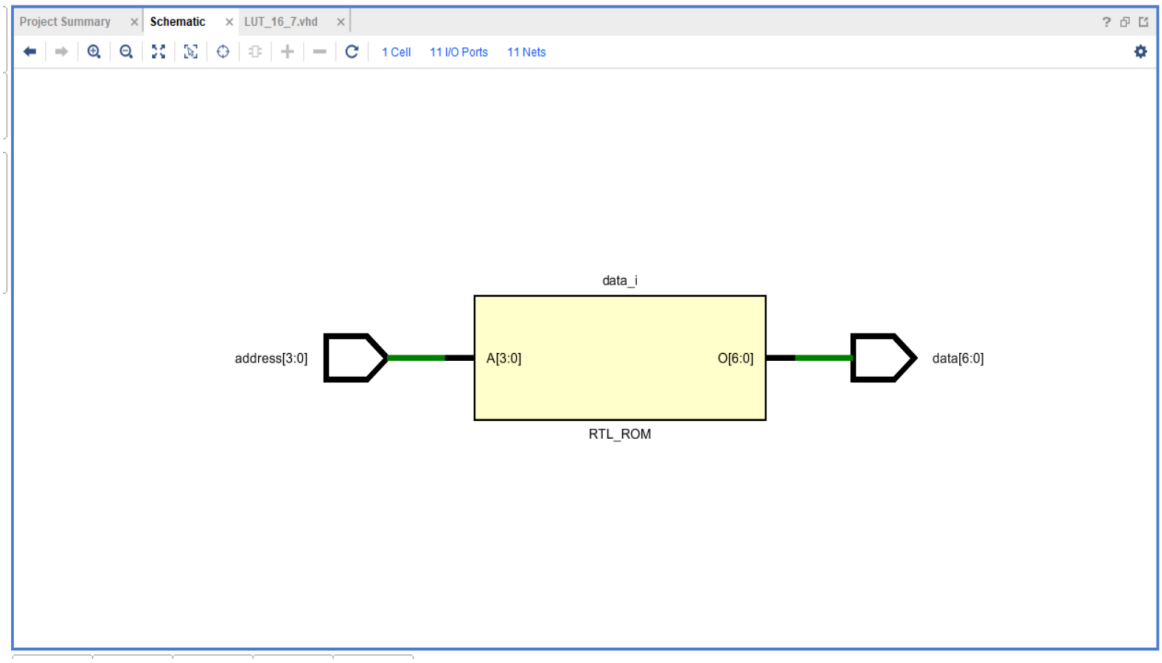
LUT for 7-Segment Display

Design Source

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
entity LUT_16_7 is
      Port ( address : in STD_LOGIC_VECTOR (3 downto 0);
             data : out STD_LOGIC_VECTOR (6 downto 0) );
end LUT_16_7;
architecture Behavioral of LUT_16_7 is
-- ROM type declaration
type rom_type is array (0 to 15) of std_logic_vector(6 downto 0);
-- ROM initialization
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 -- Output data based on the address
data <= sevenSegment_ROM(to_integer(unsigned(address)));
end Behavioral;
```

Design Diagram



Design Source

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Microprocessor is
      Port ( Clk : in STD_LOGIC;
            Reset : in STD_LOGIC := '0'; -- Resets clock, program counter
            and register bank
            L : out STD_LOGIC_VECTOR (3 downto 0); -- LED output of
            register 7
            C_Flag : out STD_LOGIC; -- Carry flag
            Z_Flag : out STD_LOGIC; -- Zero flag
            N_Flag : out STD_LOGIC; -- Negetive flag
            P_Flag : out STD_LOGIC; -- Parity flag (Odd parity detector)
            Seg_7 : out STD_LOGIC_VECTOR (6 downto 0); -- 7 Segment output
            of register 7
            Anode : out STD_LOGIC_VECTOR (3 downto 0)); -- Anode for 7
            segment display
end Microprocessor;
architecture Behavioral of Microprocessor is
-- Slow Clock
component Slow_Clk
      Port ( clkIn : in STD_LOGIC;
            en : in STD_LOGIC;
            reset : in STD_LOGIC;
            clkOut : out STD_LOGIC;
            clkOutBar : out STD_LOGIC );
end component;
-- Instruction Decoder
component Inst_Decoder
      Port ( inst : in STD_LOGIC_VECTOR (0 to 12); -- Instruction
            clk : in STD_LOGIC;
            regChk : in STD_LOGIC_VECTOR (3 downto 0); -- Check register
            value for JZR
            regSelA : out STD_LOGIC_VECTOR (2 downto 0); -- To select
            register to load into MUX A
            regSelB : out STD_LOGIC_VECTOR (2 downto 0); -- To select
            register to load into MUX B
            imdVal : out STD_LOGIC_VECTOR (3 downto 0); -- Immediate value
            regEn : out STD_LOGIC_VECTOR (2 downto 0); -- Enable register
            for write
            loadSel : out STD_LOGIC; -- Choose between Imd value or Add/Sub
            Unit result
            addSubSel : out STD_LOGIC; -- Add Sub selector clkEn : out
            STD_LOGIC; -- Enable/Disable clock
            jmp : out STD_LOGIC; -- Jump flag
            jmpAddress : out STD_LOGIC_VECTOR (2 downto 0)); -- Address to
            jump
end component;
```

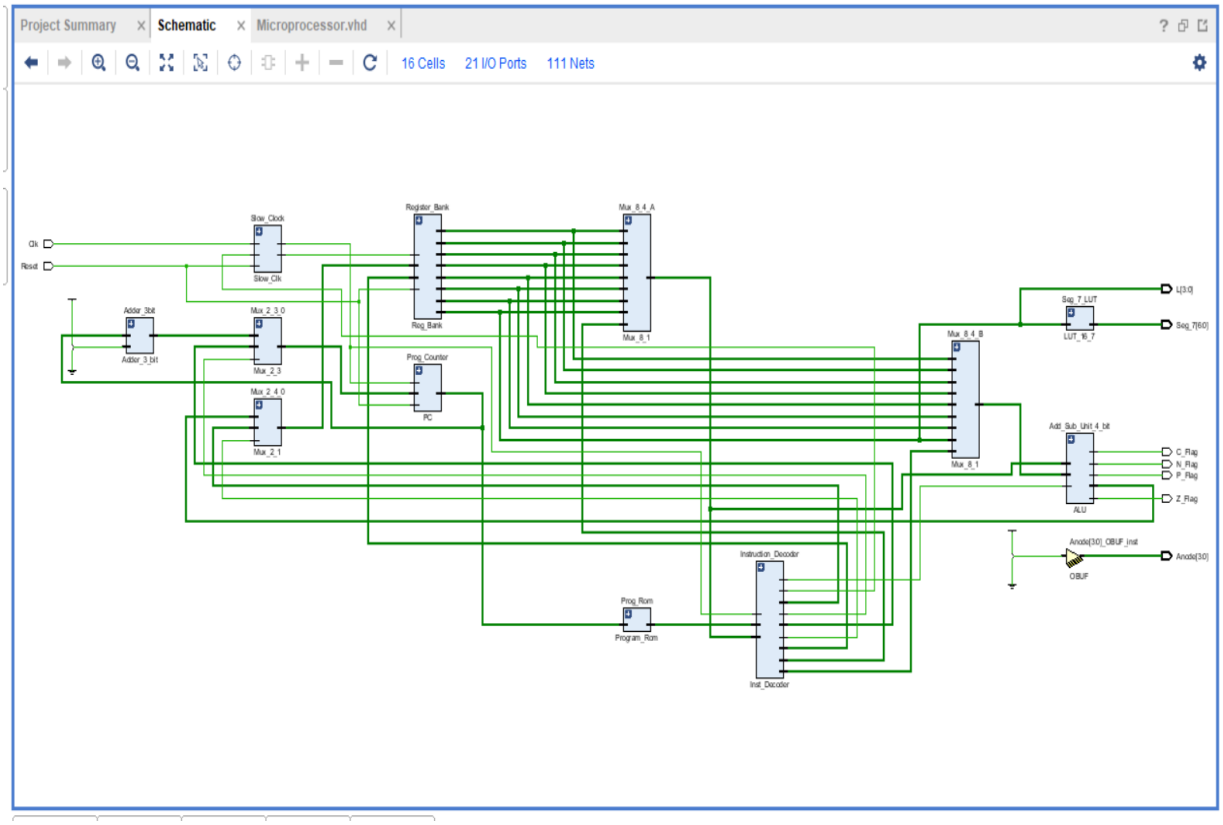
```
-- Program Rom
component Program_Rom
      Port ( romIn : in STD_LOGIC_VECTOR (2 downto 0);
             romout : out STD_LOGIC_VECTOR (0 to 12));
end component;
-- Program Counter
component PC
      Port ( pcIn : in STD_LOGIC_VECTOR (2 downto 0);
             clk : in STD_LOGIC;
             res : in STD_LOGIC;
             pcOut : out STD_LOGIC_VECTOR (2 downto 0));
end component;
 -- Register Bank
component Reg_Bank
      Port ( regIn : in STD_LOGIC_VECTOR (3 downto 0);
             regSel : in STD_LOGIC_VECTOR (2 downto 0);
             clk : in STD_LOGIC;
             reset : in STD_LOGIC;
             r0 : out STD_LOGIC_VECTOR (3 downto 0);
             r1 : out STD_LOGIC_VECTOR (3 downto 0);
             r2 : out STD_LOGIC_VECTOR (3 downto 0);
             r3 : out STD_LOGIC_VECTOR (3 downto 0);
             r4 : out STD_LOGIC_VECTOR (3 downto 0);
             r5 : out STD_LOGIC_VECTOR (3 downto 0);
             r6 : out STD_LOGIC_VECTOR (3 downto 0);
             r7 : out STD_LOGIC_VECTOR (3 downto 0));
end component;
-- 4 bit Add/Sub Unit
component ALU
      Port ( a : in STD_LOGIC_VECTOR (3 downto 0);
      b : in STD_LOGIC_VECTOR (3 downto 0);
      sel : in STD_LOGIC; -- Add/Sub Selector
      s : out STD_LOGIC_VECTOR (3 downto 0);
      cOut : out STD_LOGIC; -- Carry flag
      zOut : out STD_LOGIC; -- Zero flag
      nOut : out STD_LOGIC; -- Negetive flag
      pOut : out STD_LOGIC); -- Parity flag (Odd parity detector)
end component;
-- 3 bit Adder
component Adder_3_bit
      Port ( a : in STD_LOGIC_VECTOR (2 downto 0);
             b : in STD_LOGIC_VECTOR (2 downto 0);
             s : out STD_LOGIC_VECTOR (2 downto 0));
end component;
---- 2-way 3-bit Mux
component Mux_2_3
      Port ( i0 : in STD_LOGIC_VECTOR (2 downto 0);
             i1 : in STD_LOGIC_VECTOR (2 downto 0);
             s : in STD_LOGIC;
             q : out STD_LOGIC_VECTOR (2 downto 0));
end component;
```

```
--Mux with tri state buffer declaration
-- 2-way 4-bit Mux
component Mux_2_1
      Port ( i0 : in STD_LOGIC_VECTOR (3 downto 0);
             i1 : in STD_LOGIC_VECTOR (3 downto 0);
             s : in STD_LOGIC;
             q : out STD_LOGIC_VECTOR (3 downto 0));
end component;
-- 8-way 4-bit Mux
component Mux_8_1
      Port ( r0 : in STD_LOGIC_VECTOR (3 downto 0);
             r1 : in STD_LOGIC_VECTOR (3 downto 0);
             r2 : in STD_LOGIC_VECTOR (3 downto 0);
             r3 : in STD_LOGIC_VECTOR (3 downto 0);
             r4 : in STD_LOGIC_VECTOR (3 downto 0);
             r5 : in STD_LOGIC_VECTOR (3 downto 0);
             r6 : in STD_LOGIC_VECTOR (3 downto 0);
             r7 : in STD_LOGIC_VECTOR (3 downto 0);
             s : in STD_LOGIC_VECTOR (2 downto 0);
             q : out STD_LOGIC_VECTOR (3 downto 0));
end component;
component LUT_16_7
      Port ( address : in STD_LOGIC_VECTOR (3 downto 0);
             data : out STD_LOGIC_VECTOR (6 downto 0));
end component;
type DATA_BUS is array (0 to 7) of std_logic_vector(3 downto 0);
signal Clk_slow, Clk_slow_bar, Clk_En : STD_LOGIC; -- Internal clock
signal LD, Sub, Jmp : STD_LOGIC;
signal Sel_A, Sel_B, Reg_En, Address, Jmp_Address, Prg_Address, Mem_Sel :
STD_LOGIC_VECTOR (2 downto 0);
signal A, B, S, D, M, AddressNew, Jmp_AddressNew, Prg_AddressNew :
STD_LOGIC_VECTOR (3 downto 0);
signal I : STD_LOGIC_VECTOR (0 to 12);
signal R : DATA_BUS;
begin
-- Slow Clock
Slow_Clock : Slow_Clk
      PORT MAP ( clkIn => Clk,
                 en => Clk_En,
                 reset => Reset,
                 clkOut => Clk_slow,
                 clkOutBar => Clk_slow_bar);
-- Instruction Decoder
Instruction_Decoder : Inst_Decoder
      PORT MAP ( inst => I,
                 clk => Clk_slow,
                 regChk => A,
                 regSelA => Sel_A,
                 regSelB => Sel_B,
                 imdVal => M,
                 regEn => Reg_En,
                 loadSel => LD,
                 addSubSel => Sub,
```

```
                clkEn => Clk_En,
                jmp => Jmp,
                jmpAddress => Jmp_Address);
-- Program Rom
Prog_Rom : Program_Rom
      PORT MAP ( romIn => Mem_Sel,
                romOut => I );
-- Program Counter
Prog_Counter : PC
      PORT MAP ( pcIn => Address,
            clk => Clk_slow,
            res => Reset,
            pcOut => Mem_Sel );
-- Register Bank
Register_Bank : Reg_Bank
      PORT MAP ( regIn => D,
            regSel => Reg_En,
            clk => Clk_slow_bar,
            reset => Reset,
            r0 => R(0),
            r1 => R(1),
            r2 => R(2),
            r3 => R(3),
            r4 => R(4),
            r5 => R(5),
            r6 => R(6),
            r7 => R(7) );
-- 4 bit Add/Sub Unit
Add_Sub_Unit_4_bit : ALU
      PORT MAP ( a => A,
            b => B,
            sel => Sub,
            S => S,
            cOut => C_Flag,
            zOut => Z_Flag,
            nOut => N_Flag,
            pOut => P_Flag );
-- 3 bit Adder
Adder_3bit : Adder_3_bit
      PORT MAP ( a => Mem_Sel,
            b => "001",
            s => Prg_Address );
-- 2-way 3-bit Mux
Mux_2_3_0 : Mux_2_3
      PORT MAP ( i0 => Prg_Address,
            i1 => Jmp_Address,
            s => Jmp,
            q => Address );
-- 2-way 4-bit Mux
Mux_2_4_0 : Mux_2_1
      PORT MAP ( i0 => S,
                i1 => M,
                s => LD,
```

```
                q => D );
-- 8-way 4-bit Mux
Mux_8_4_A : Mux_8_1
      PORT MAP ( r0 => R(0),
                r1 => R(1),
                r2 => R(2),
                r3 => R(3),
                r4 => R(4),
                r5 => R(5),
                r6 => R(6),
                r7 => R(7),
                s => Sel_A,
                q => A );
-- 8-way 4-bit Mux
Mux_8_4_B : Mux_8_1
      PORT MAP ( r0 => R(0),
                r1 => R(1),
                r2 => R(2),
                r3 => R(3),
                r4 => R(4),
                r5 => R(5),
                r6 => R(6),
                r7 => R(7),
                s => Sel_B,
                q => B );
-- 7-Segment Display
Seg_7_LUT : LUT_16_7
PORT MAP ( address => R(7),
          data => Seg_7 );
L <= R(7); -- Map value in register 7 to LEDs
Anode <= "1110"; -- Enable only first display in 7 segement display
end Behavioral;
```

Design Diagram

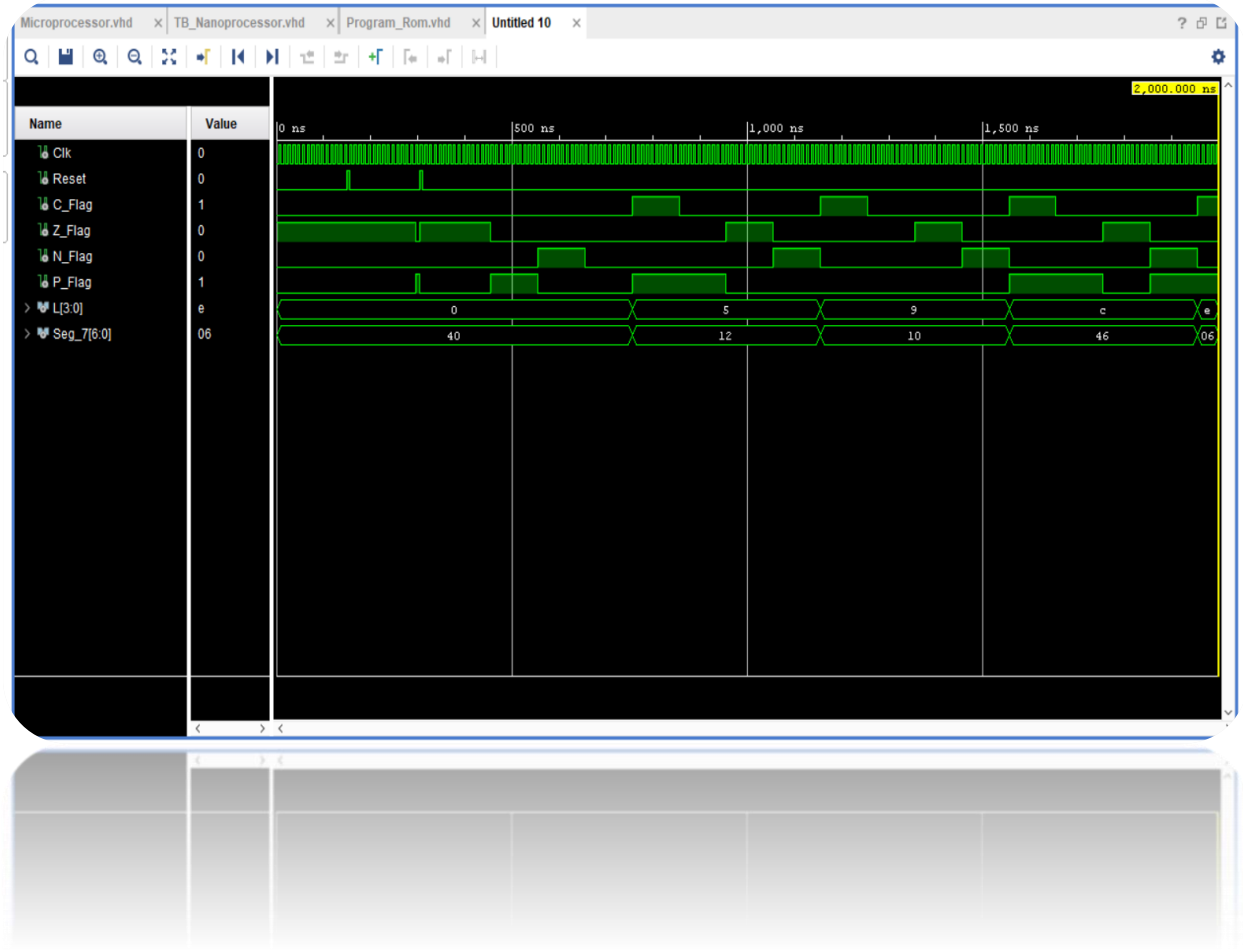


Test Bench File

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity TB_Microprocessor is
-- Port ( );
end TB_Microprocessor;
architecture Behavioral of TB_Microprocessor is
component Microprocessor
      Port ( Clk : in STD_LOGIC;
             Reset : in STD_LOGIC;
             L : out STD_LOGIC_VECTOR (3 downto 0);
             C_Flag : out STD_LOGIC; -- Carry flag
             Z_Flag : out STD_LOGIC; -- Zero flag
             N_Flag : out STD_LOGIC; -- Negetive flag
             P_Flag : out STD_LOGIC; -- Parity flag (Odd parity detector)
             Seg_7 : out STD_LOGIC_VECTOR (6 downto 0);
             Anode : out STD_LOGIC_VECTOR (3 downto 0) );
end component;
```

```
signal Clk : STD_LOGIC := '0';
signal Reset, C_Flag, Z_Flag , N_Flag, P_Flag : STD_LOGIC := '0';
signal L : STD_LOGIC_VECTOR (3 downto 0);
signal Seg_7 : STD_LOGIC_VECTOR (6 downto 0);
begin
UUT: Microprocessor
      PORT MAP( Clk => Clk,
            Reset => Reset,
            L => L,
            C_Flag => C_Flag,
            Z_Flag => Z_Flag,
            N_Flag => N_Flag,
            P_Flag => P_Flag,
            Seg_7 => Seg_7 );
process
begin
wait for 5ns;
Clk <= NOT(Clk);
end process;
process
begin
wait for 150ns;
Reset <= '1';
wait for 5ns;
Reset <= '0';
wait for 150ns;
Reset <= '1';
wait for 5ns;
Reset <= '0';
wait;
end process;
end Behavioral;
```

Timing Graph



Optimization Strategies

- Using 13-bit instructions with 3 bits dedicated to the opcode allows for a more efficient use of resources and potentially improves performance in several ways:

1. **Compact Instruction Set**: With 3 bits for the opcode, we can accommodate 8 different instructions. This means we can have a concise set of instructions that cover a wide range of operations, reducing the overall size of the instruction set compared to larger opcode sizes.
2. **Increased Instruction Variety**: Having 8 instructions available allows for a more diverse range of operations that can be directly executed by the processor. This can reduce the need for complex instruction sequences or multiple instructions to perform common tasks, streamlining the execution of programs.
3. **Improved Instruction Fetch Efficiency**: With a smaller opcode size, the instruction fetch stage of the processor pipeline becomes more efficient. Fetching shorter instructions requires less data transfer, which can lead to faster execution times.
4. **Simpler Instruction Decoding**: With fewer bits dedicated to the opcode, the instruction decoding logic becomes simpler and potentially faster, leading to reduced complexity in the processor's control unit and shorter instruction decode times.

- The use of generic multiplexers (muxs) can offer several benefits for resource optimization:

1. **Flexibility**: Generic muxs allow to create multiplexing structures that can be easily customized and reused throughout the design. By parameterizing the multiplexer entity with generic parameters such as the number of inputs and select lines, we can create muxs of various sizes to suit different requirements within the processor design.
2. **Reduced Resource Usage**: Using generic muxs allows to instantiate multiplexers of the exact size needed for each specific task. This avoids over-provisioning of resources, leading to efficient utilization of FPGA resources such as lookup tables (LUTs) and routing resources. It also prevents the wastage of unused inputs or select lines in larger multiplexers.
3. **Improved Performance**: Generic muxs can be optimized for performance by selecting the most appropriate implementation strategy based on factors such as input size, select line width, and critical path requirements. This optimization can result in faster operation and reduced propagation delay compared to using fixed-size multiplexers or implementing custom multiplexing logic for each use case.
4. **Ease of Maintenance and Debugging**: By encapsulating multiplexing functionality within generic mux components, we can promote modular design practices. This makes the design easier to understand, maintain, and debug since multiplexing logic is abstracted into reusable modules with well-defined interfaces.
5. **Portability**: Generic muxs can be easily ported to different FPGA platforms or synthesized for different target technologies without requiring significant modifications to the HDL code. This portability enhances the flexibility of the design and facilitates scalability for future iterations or upgrades of the nano processor.

6. **Enhanced Design Abstraction**: Using generic muxs promotes a higher level of design abstraction by separating the multiplexing functionality from the rest of the design. This abstraction simplifies the overall design process, allowing to focus on higher-level aspects of the processor architecture without getting bogged down in low-level implementation details.

- We have implemented multiplexers in our nano processor design using tri state buffers.

  This can contribute to resource optimization in many ways.

  1. **Reduced Logic Complexity**: Tri-state buffers can simplify the implementation of multiplexers by directly selecting one of several input signals to propagate to the output. This simplicity in logic design leads to reduced resource usage in terms of FPGA resources such as lookup tables (LUTs) and routing resources.
  2. **Efficient Resource Utilization**: Tri-state buffers can be configured to effectively disconnect unused input signals from the multiplexer output, preventing unnecessary power consumption and reducing the overall resource footprint of the design.
  3. **Dynamic Signal Routing**: Tri-state buffers enable dynamic signal routing within the multiplexer, allowing for flexible selection of input signals based on control signals or processor state. This dynamic routing capability can enhance the versatility of the nano processor design without significantly increasing resource usage.
  4. **Minimized Signal Contention**: Tri-state buffers mitigate signal contention issues that may arise when multiple input signals attempt to drive the same output line simultaneously. By enabling only one input signal at a time, tri-state buffers ensure proper signal isolation and prevent potential conflicts, leading to more reliable and robust operation of the multiplexer.
  5. **Improved Signal Integrity**: Tri-state buffers can enhance signal integrity by reducing signal reflections and noise propagation within the multiplexer circuitry. By effectively isolating inactive input signals, tri-state buffers help maintain signal quality and integrity.

- Synchronizing the instruction decoder allows for better optimization of critical paths within the processor, leading to improved performance. By coordinating the timing of operations, we can minimize latency and maximize throughput in the processor.
- Operating the register bank on the complement of the clock signal can help in reducing dynamic power consumption. By toggling these components only, when necessary, rather than on every clock cycle, overall power consumption can be minimized.
- Certain components of the nano processor have specific timing requirements or performance constraints that can be better met by operating them on the complement of the clock signal (eg-register bank). By tailoring the timing of these components independently, we have optimized their performance without

impacting the overall operation of the processor which can lead to improved overall system performance and efficiency.

- Use of data busses instead of single wires plays a major role in the optimization of the nano processor design.

1.Reduced Wiring Complexity: Data buses streamline the interconnection of various components within the processor, replacing multiple individual signal lines with a single bus. This reduces the overall complexity of the processor's wiring layout, leading to simpler routing and potentially lower routing congestion on the FPGA.

2.Improved Scalability: Data buses facilitate the expansion and scalability of the processor design by providing a standardized interface for data transfer between components. Adding new functional units or peripheral modules to the processor architecture becomes more straightforward since they can connect to existing data buses without requiring extensive modifications to the overall design.

3.Enhanced Performance: Data buses enable simultaneous transmission of multiple data elements across the processor, improving overall data throughput and performance. By leveraging parallelism inherent in data bus architectures, the processor can efficiently transfer large volumes of data between different functional units or memory modules, leading to faster execution of instructions and tasks.

4.Optimized Resource Utilization: Data buses help optimize resource utilization within the processor by reducing the number of dedicated signal lines required for inter-component communication. This conserves valuable FPGA resources such as routing resources and I/O pins, allowing for more efficient use of available resources and potentially enabling the integration of additional processor features or enhancements.

5.Simplified Control Logic: Using data buses simplifies the design of control logic within the processor, as it provides a unified interface for data transfer operations. This simplification reduces the complexity of the processor's control unit, leading to shorter critical paths and potentially improving overall clock frequency and performance.

6.Support for Modular Design: Data buses promote modular design practices by decoupling individual components within the processor architecture and providing a standardized communication interface between them. This modular approach enhances design flexibility, ease of integration, and maintenance, allowing designers to develop and test processor modules independently before integrating them into the larger system.

- Incorporating a reset option for various components such as the Program Counter, Instruction Decoder, and Register Bank in our nano processor design can contribute to optimization in several ways:

1. **Initialization**: Resetting these components ensures that they start in a known state whenever the processor is powered on or initialized. This initialization process eliminates any residual or undefined values, ensuring reliable and predictable

behavior from the start. It helps prevent issues such as undefined behavior or unintended operation due to uninitialized states.

2. **Garbage Collection**: Resetting components allows for the efficient removal of garbage or stale data that may accumulate during the processor's operation. For example, resetting the Program Counter clears any previous instruction addresses, ensuring that the processor starts executing instructions from the designated entry point. Similarly, resetting the Register Bank clears any stale data stored in registers, preventing interference with subsequent operations.
3. **Improved Stability**: Resetting key components helps maintain the stability and integrity of the processor's operation over time. By periodically resetting critical components, we can mitigate the accumulation of errors or drift that may occur during prolonged operation, ensuring consistent and reliable performance over extended periods.
4. **Performance Optimization**: Resetting components can lead to performance optimization by restoring them to a clean, optimal state. For example, resetting the Instruction Decoder ensures that it is ready to correctly interpret incoming instructions without any lingering effects from previous operations. This can reduce latency and improve overall instruction execution time, contributing to enhanced performance of the nano processor.
5. **Error Recovery**: Reset options provide a mechanism for error recovery in the event of unexpected faults or malfunctions. If the processor encounters an error condition or enters an inconsistent state, resetting affected components can help restore normal operation and recover from the error without requiring a complete system restart.

Constraint File

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

set_property PACKAGE_PIN U18 [get_ports Reset]
    set_property IOSTANDARD LVCMOS33 [get_ports Reset]


## LEDs
set_property PACKAGE_PIN U16 [get_ports {L[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {L[0]}]
set_property PACKAGE_PIN E19 [get_ports {L[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {L[1]}]
set_property PACKAGE_PIN U19 [get_ports {L[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {L[2]}]
set_property PACKAGE_PIN V19 [get_ports {L[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {L[3]}]
set_property PACKAGE_PIN P3 [get_ports {C_Flag}]
    set_property IOSTANDARD LVCMOS33 [get_ports {C_Flag}]
set_property PACKAGE_PIN N3 [get_ports {Z_Flag}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Z_Flag}]
set_property PACKAGE_PIN P1 [get_ports {N_Flag}]
    set_property IOSTANDARD LVCMOS33 [get_ports {N_Flag}]
set_property PACKAGE_PIN L1 [get_ports {P_Flag}]
    set_property IOSTANDARD LVCMOS33 [get_ports {P_Flag}]
```

```
##7 segment display
set_property PACKAGE_PIN W7 [get_ports {Seg_7[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Seg_7[0]}]
set_property PACKAGE_PIN W6 [get_ports {Seg_7[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Seg_7[1]}]
set_property PACKAGE_PIN U8 [get_ports {Seg_7[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Seg_7[2]}]
set_property PACKAGE_PIN V8 [get_ports {Seg_7[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Seg_7[3]}]
set_property PACKAGE_PIN U5 [get_ports {Seg_7[4]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Seg_7[4]}]
set_property PACKAGE_PIN V5 [get_ports {Seg_7[5]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Seg_7[5]}]
set_property PACKAGE_PIN U7 [get_ports {Seg_7[6]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {Seg_7[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]}]
```

Resource Utilization

Report of Coimponents Usage

| Tool Version : Vivado v.2018.1 (win64) Build 2188600 Wed Apr 4 18:40:38 MDT 2018

| Date : Fri May 3 21:08:33 2024

| Host : DESKTOP-FVHNC28 running 64-bit major release (build 9200)

| Command : report_utilization -file Microprocessor_utilization_synth.rpt -pb Microprocessor_utilization_synth.pb

| Design : Microprocessor

| Device : 7a35tcpg236-1

| Design State : Synthesized

Utilization Design Information

- **Table of Contents**

1.Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs	195	0	20800	0.94
LUT as Logic	195	0	20800	0.94
LUT as Memory	0	0	9600	0.00
Slice Registers	80	0	41600	0.19
Register as Flip Flop	78	0	41600	0.19
Register as Latch	2	0	41600	<0.01
F7 Muxes	0	0	16300	0.00
F8 Muxes	0	0	8150	0.00

* Warning! The Final LUT count, after physical optimizations and full implementation, is typically lower. Run opt_design after synthesis, if not already completed, for a more realistic count.

1.1 Summary of Registers by Type

Total	Clock enable	Synchronous	Asynchronous
0	-	-	-
0	-	-	Set
0	-	-	Reset
0	-	Set	-
0	-	Reset	-
0	Yes	-	-
2	Yes	-	Set
36	Yes	-	Reset
0	Yes	Set	-
42	Yes	Reset	-

2. Memory

Site Type	Used	Fixed	Available	Util%
Block RAM Tile	0	0	50	0.00
RAMB36/FIFO*	0	0	50	0.00
RAMB18	0	0	100	0.00

Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate only one FIFO36E1 or one FIFO18E1. However, if a FIFO18E1 occupies a Block RAM Tile, that tile can still accommodate a RAMB18E1

3. DSP

Site Type	Used	Fixed	Available	Util%
DSPs	0	0	90	0.00

4.IO and GT Specific

Site Type	Used	Fixed	Available	Util%
Bonded IOB	21	0	106	19.81
Bonded IPADs	0	0	10	0.00
Bonded OPADs	0	0	4	0.00
PHY_CONTROL	0	0	5	0.00
PHASER_REF	0	0	5	0.00
OUT_FIFO	0	0	20	0.00
IN_FIFO	0	0	20	0.00
IDELAYCTRL	0	0	5	0.00
IBUFDS	0	0	104	0.00
GTPE2_CHANNEL	0	0	2	0.00
PHASER_OUT/PHASER_OUT_PHY	0	0	20	0.00
PHASER_IN/PHASER_IN_PHY	0	0	20	0.00
IDELAYE2/IDELAYE2_FINEDELAY	0	0	250	0.00
IBUFDS_GTE2	0	0	2	0.00
ILOGIC	0	0	106	0.00
OLOGIC	0	0	106	0.00

5. Clocking

Site Type	Used	Fixed	Available	Util%
BUFGCTRL	1	0	32	3.13
BUFIO	0	0	20	0.00
MMCME2_ADV	0	0	5	0.00
PLLE2_ADV	0	0	5	0.00
BUFMRCE	0	0	10	0.00
BUFHCE	0	0	72	0.00
BUFR	0	0	20	0.00

6. Specific Feature

Site Type	Used	Fixed	Available	Util%
BSCANE2	0	0	4	0.00
CAPTUREE2	0	0	1	0.00
DNA_PORT	0	0	1	0.00
EFUSE_USR	0	0	1	0.00
FRAME_ECCE2	0	0	1	0.00
CAPE2	0	0	2	0.00
PCIE_2_1	0	0	1	0.00
STARTUPE2	0	0	1	0.00
XADC	0	0	1	0.00

7. Primitives

Ref Name	Used	Functional Category
LUT2	132	LUT
LUT4	70	LUT
FDRE	42	Flop & Latch
FDCE	34	Flop & Latch
LUT3	20	LUT
OBUF	19	IO
LUT5	14	LUT
LUT6	10	LUT
CARRY4	8	CarryLogic
LUT1	5	LUT
LDCE	2	Flop & Latch
IBUF	2	IO
FDPE	2	Flop & Latch
BUFG	1	Clock

8. Black Boxes

+----------+------+

| Ref Name | Used |

+----------+------+

9. Instantiated Netlists

+----------+------+

| Ref Name | Used |

+----------+------+

Individual Contributions

1. **Dissanayake.D.M.A.K (220135N):**

Dissanayake focused on Majority of key components of the project. He took charge of developing the Register Bank, which is crucial for storing temporary data during computations. Additionally, they designed the 4-bit adder/subtractor, a fundamental arithmetic unit essential for mathematical operations within the processor. Furthermore, Dissanayake contributed to the project by developing the Look-Up Table (LUT) for the 7-segment display, enabling the processor to output data in a human-readable format. Also he took the responsibility of microprocessor. He also took responsibility for creating simulation source files, allowing for thorough testing and debugging of the designed components.

2. **Ekanayake.L.A.D (220155B)**:

Ekanayake played a significant role in the project by focusing on critical components such as the Program Counter, which keeps track of the execution sequence within the processor. They also designed the Mux 2-way 3-bit and Mux 2-way 4-bit, essential multiplexers responsible for selecting between multiple data inputs. Additionally, Ekanayake contributed to the project by providing simulation source files, aiding in the verification and validation of the designed components.

3. **Jayasinghe.U.H.N (220264H):**

Jayasinghe's contributions to the project were pivotal in several areas. They were responsible for designing the 3-bit adder, a crucial component for performing addition operations within the processor. Additionally, Jayasinghe implemented the slow clock, ensuring the synchronization and timing accuracy of the processor's operations. They also developed the Program ROM, which stores the instructions that the processor executes. Furthermore, Jayasinghe took the initiative to compile and present the project's findings in the form of a comprehensive lab report, documenting the team's work and results.

4. **Lihinikaduarachchi.L.A.G.H (220361D):**

Lihinikaduarachchi made substantial contributions to the project by focusing on key components critical to its functionality. They designed the Mux 8-way 4-bit, an essential multiplexer responsible for selecting one of eight input data sources. Additionally, Lihinikaduarachchi developed the Instruction Decoder, which interprets the instructions fetched from memory and directs the processor's operations accordingly. Furthermore, they actively participated in compiling and presenting the project's findings through the creation of a detailed lab report, documenting the team's work processes and outcomes effectively.

Conclusion

In conclusion, this lab provided us an opportunity to gain hands on experience in microprocessor design and development.

Through the process of designing a 4-bit nanoprocessor capable of executing a set of fundamental instructions, we gained a comprehensive understanding of various essential components and their interconnections within a processor architecture. By successfully completing this lab, we achieved several key learning objectives:

Firstly, we mastered the design and implementation of a 4-bit arithmetic unit capable of performing addition and subtraction operations on signed integers using 2's complement representation. This involved adapting and extending existing components such as the 4-bit Ripple Carry Adder (RCA) from previous labs.

Secondly, we improved our skills in instruction decoding, enabling us to activate the necessary components within the processor based on the instructions provided.

Moreover, we developed essential modules such as k-way b-bit multiplexers, Program Counter (PC) with reset functionality, Register Bank with preset values and reset capability, Program ROM for storing assembly programs, buses for simplified interconnection, and instruction decoder for activating relevant components.

Furthermore, by adhering to design principles and leveraging VHDL techniques, we optimized the efficiency and performance of our nanoprocessor. We strategically utilized resources, such as multiplexers and flip-flops, to achieve a balance between functionality, complexity, and resource utilization.

In summary, this lab provided a hands-on experience that not only deepened our understanding of microprocessor design but also equipped us with practical skills and techniques essential for tackling real-world challenges in digital system design and implementation. Through experimentation, simulation, and verification, we laid the foundation for further exploration in microprocessor designing.

References

https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://domipheus.com/blog/designing-a-cpu-in-vhdl-part-1-rationale-tools-method/&ved=2ahUKEwjM06iimPOFAxX_4TgGHVA9Bxg4ChAWegQICxAB&usg=AOvVaw0kyMPBa2ZQoVjLRXl_YTr4

https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://www.redhat.com/sysadmin/cpu-components-functionality&ved=2ahUKEwjFlJDYmPOFAxUx1jgGHZbCDa8QFnoECBYQAQ&usg=AOvVaw2SbDMo_aFO2L5woKljqlbd

__THE END__