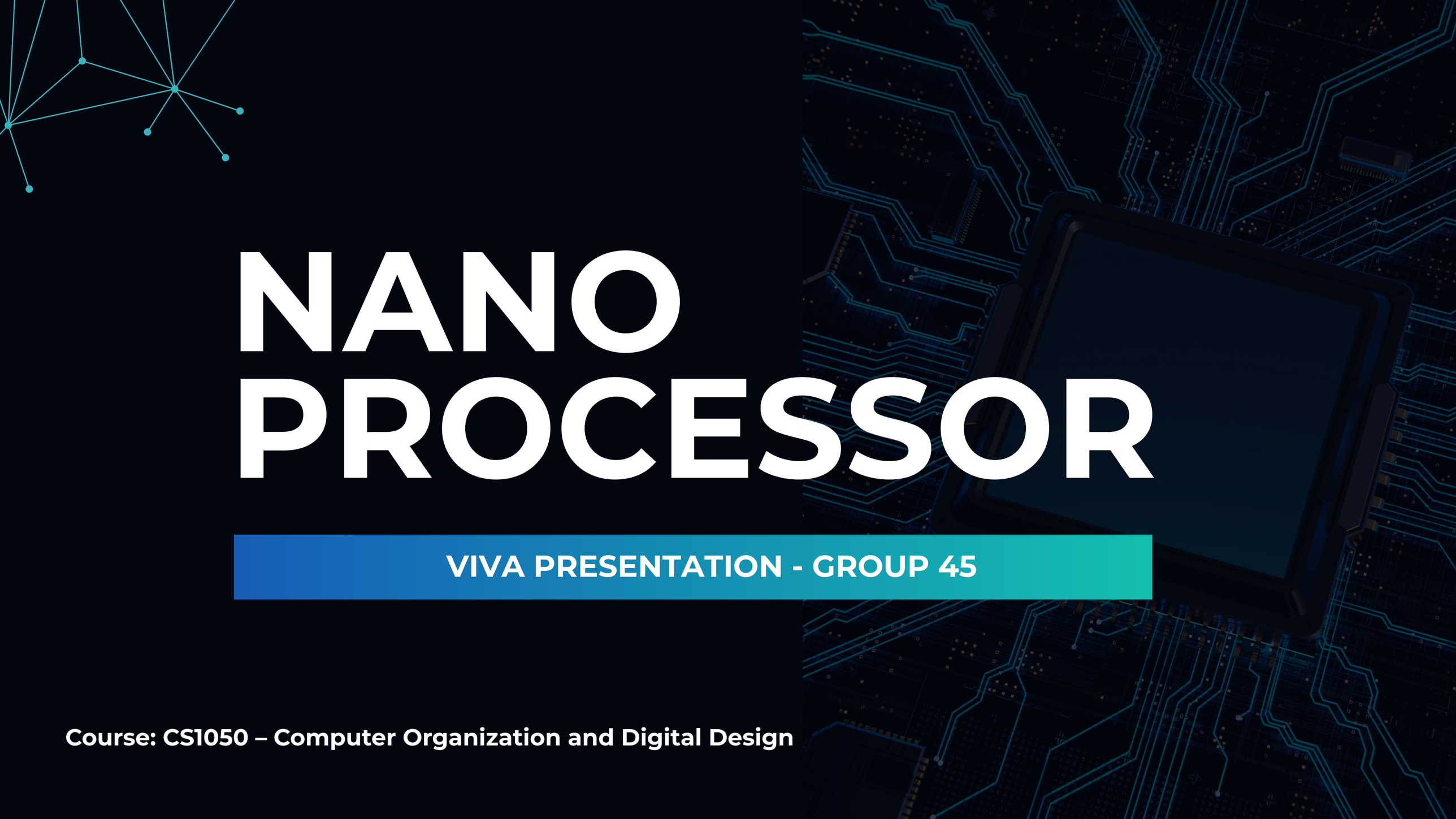


NANO PROCESSOR

VIVA PRESENTATION - GROUP 45

Course: CS1050 – Computer Organization and Digital Design

INTRODUCTION

Objective

To design a 4-bit nano-processor capable of executing a simple instruction set using VHDL and simulate its functionality

Why?

- Helps understand CPU architecture and instruction execution
- Bridges theory with hardware-level digital design
- Uses practical skills: VHDL, simulation tools, schematics

Software - Vivado (2018 version)
Language - VHDL
Tested on - Basys3 Board

INTRODUCTION

Key Features

- To design a 4-bit nano-processor capable of executing a simple instruction set using VHDL and simulate its functionality.
- Includes a custom instruction decoder and program ROM.
- Output every calculation through outputs mapped to LEDs of the BASYS 3 FPGA board.

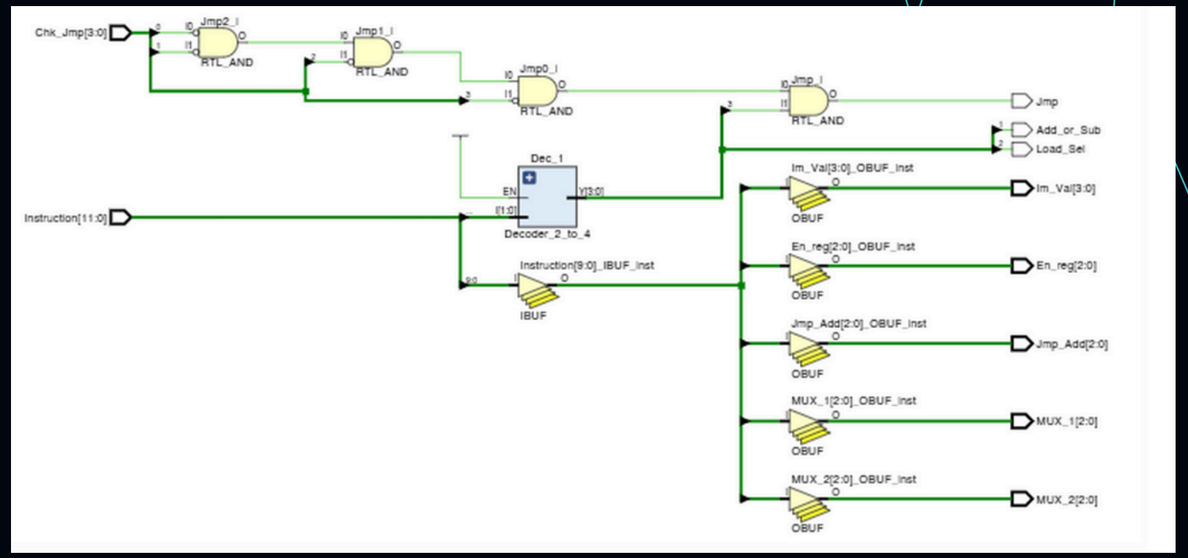
BASIC PROCESSOR INSTRUCTION SET

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

INSTRUCTION DECODER

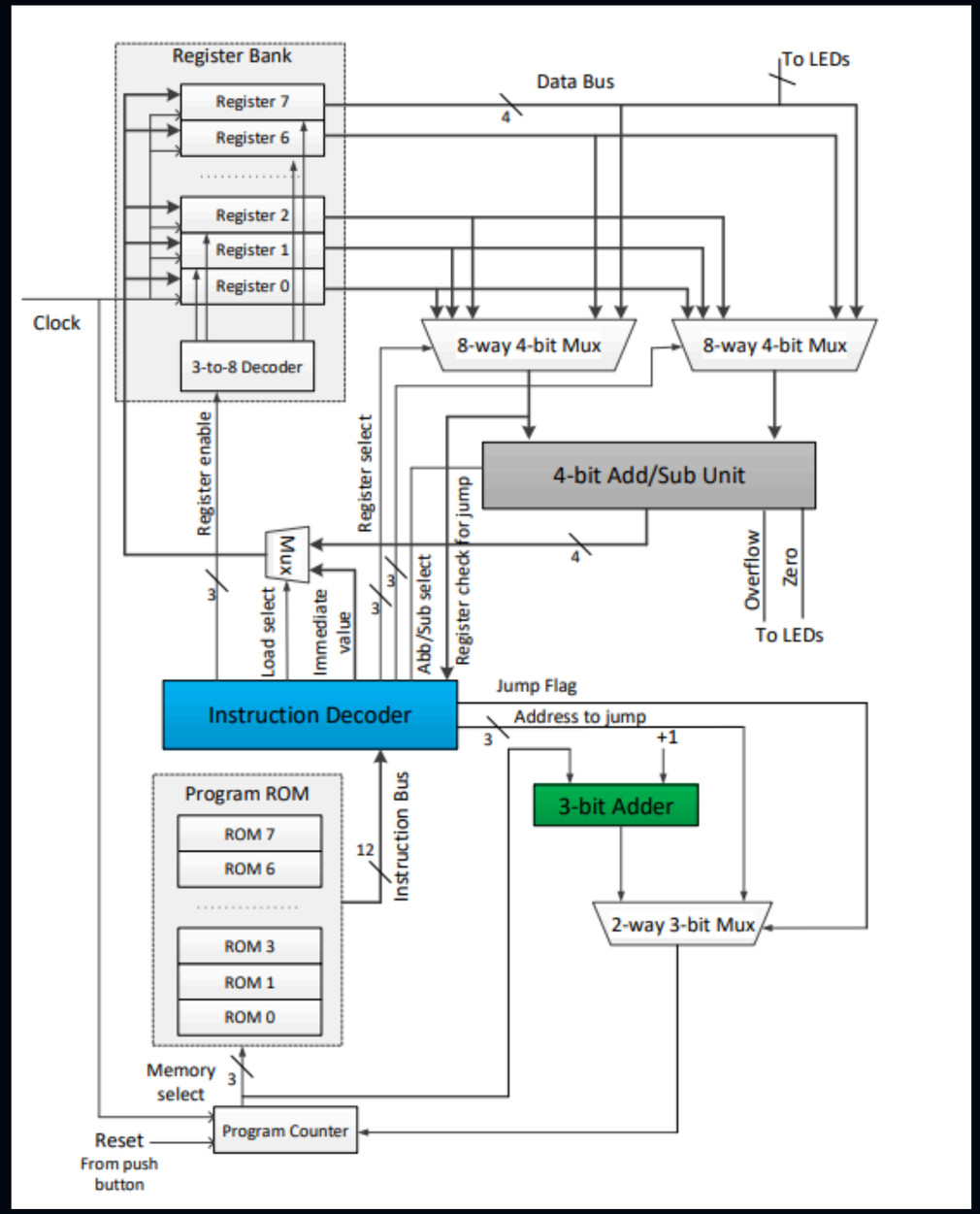
Key Features

- Decodes the 12 bit instructions as inputs/commands for the different components connected.
- Checks the jump function and passes it onto the 3bit multiplexer.
- Sends the instruction to the add/sub unit to define it's operation.



INSTRUCTION DECODER

10	MOVI R,d	1 0 R R R 0 0 0 d d d d (Store the values represented in d d d d in the register shown in R R R)
00	ADD Ra, Rb	0 0 Ra Ra Ra Rb Rb Rb 0 0 0 0 (Add the values stored in (Ra Ra Ra) and (Rb Rb Rb) and store it back to register Ra)
01	NEG R	0 1 R R R 0 0 0 0 0 0 0 (Subtract the value stored in register (R R R) from 0 to make the value negative, and store it back in (R R R)
11	JZR R,d	1 1 R R R 0 0 0 0 d d d (If (R R R) = 0, program counter should call ROM (d d d), else program counter should increase it’s value by 1)

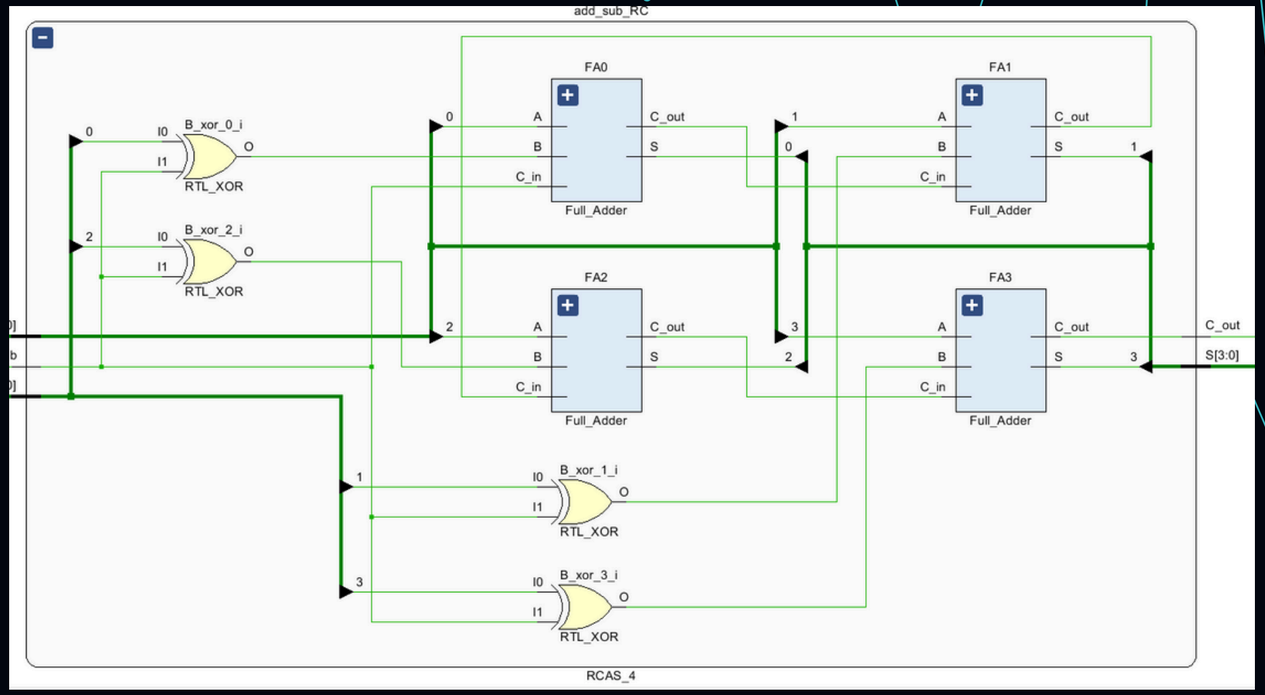


HIGH LEVEL DESIGN

4 BIT ADDER/SUBSTRACTOR

Key Features

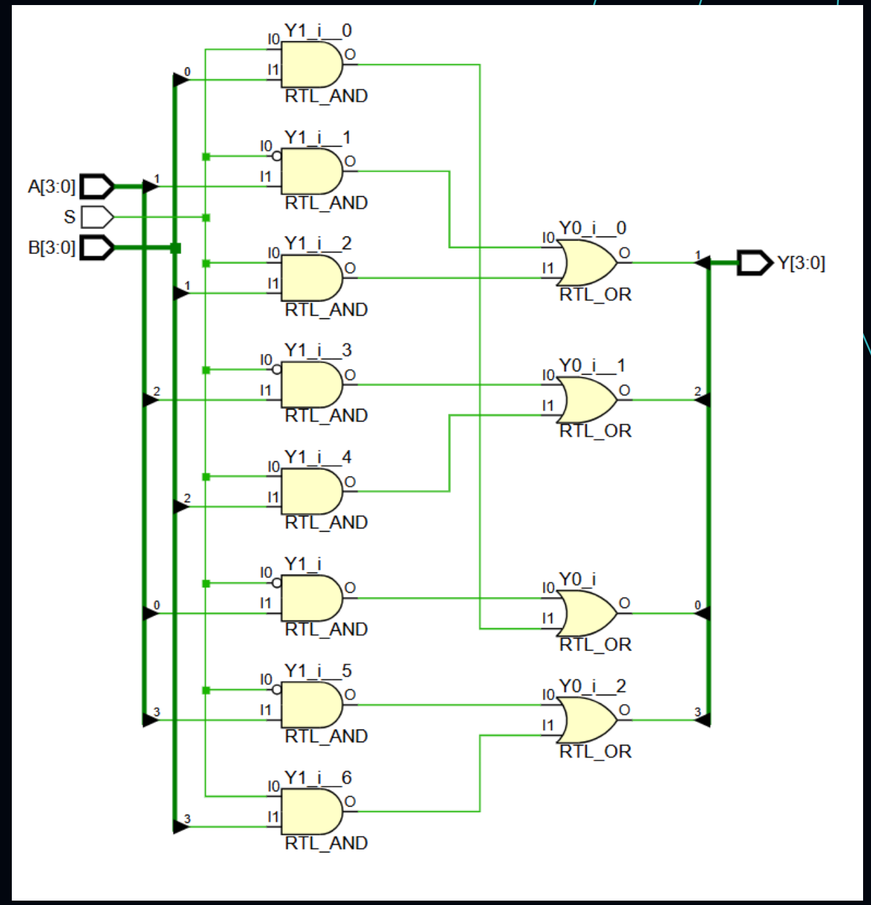
- Made of 4 full adders (which in turn are made of two half adders each)
- It has a control line that determines whether the operation is addition or subtraction.
- It has a 4-bit output line (S) that represents the result of the operation.
- It has a carry-out line (C_out) that indicates whether the result of the operation has overflowed (i.e., exceeded 4 bits).



2 WAY 4 BIT MULTIPLEXER

Key Features

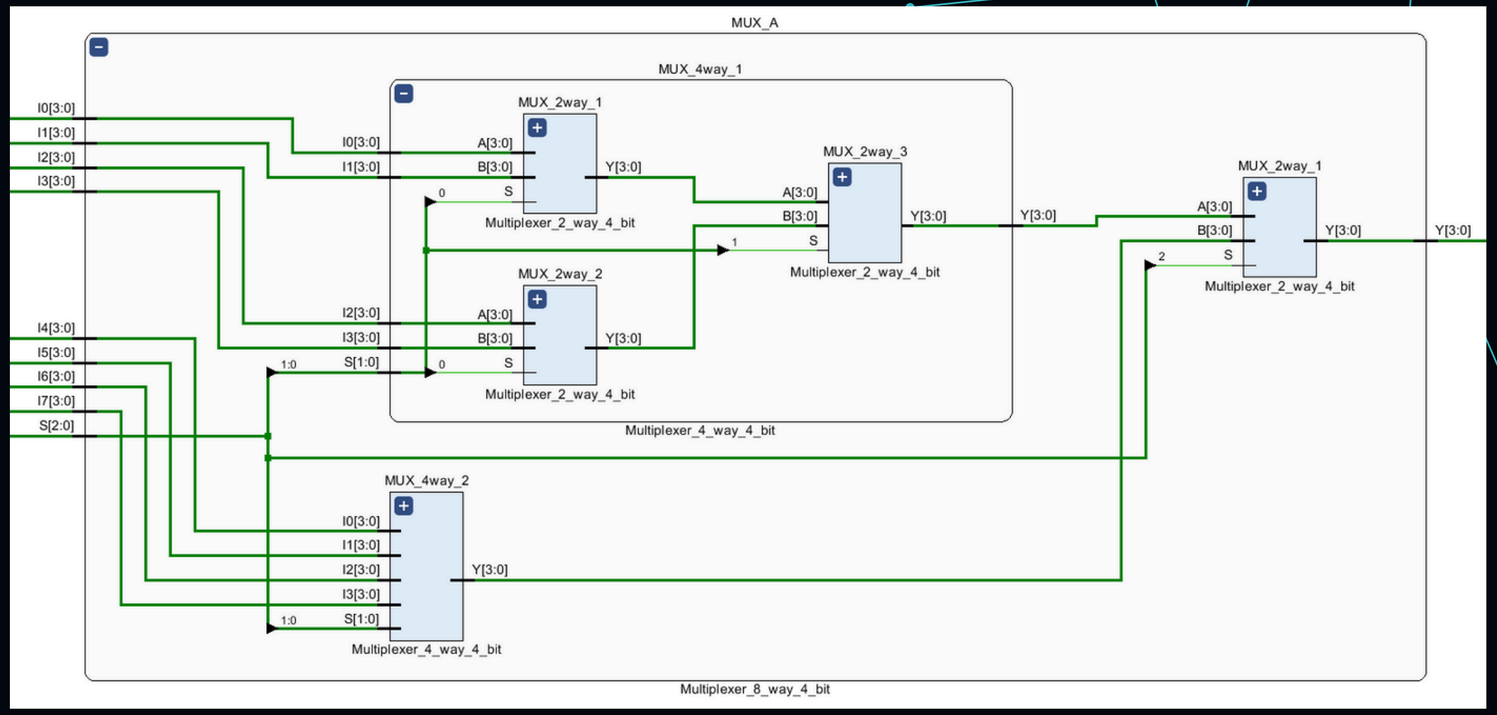
- It has 2 input lines (I0 and I1) that select which of the two 4-bit input groups to pass through to the output.
- It has 1 select line (S) that determines which of the two 4-bit input groups to select.
- It has 4 output lines (O0, O1, O2, O3) that carry the selected 4-bit input.
- Made of AND gates and OR gates



8 WAY 4 BIT MULTIPLEXER

Key Features

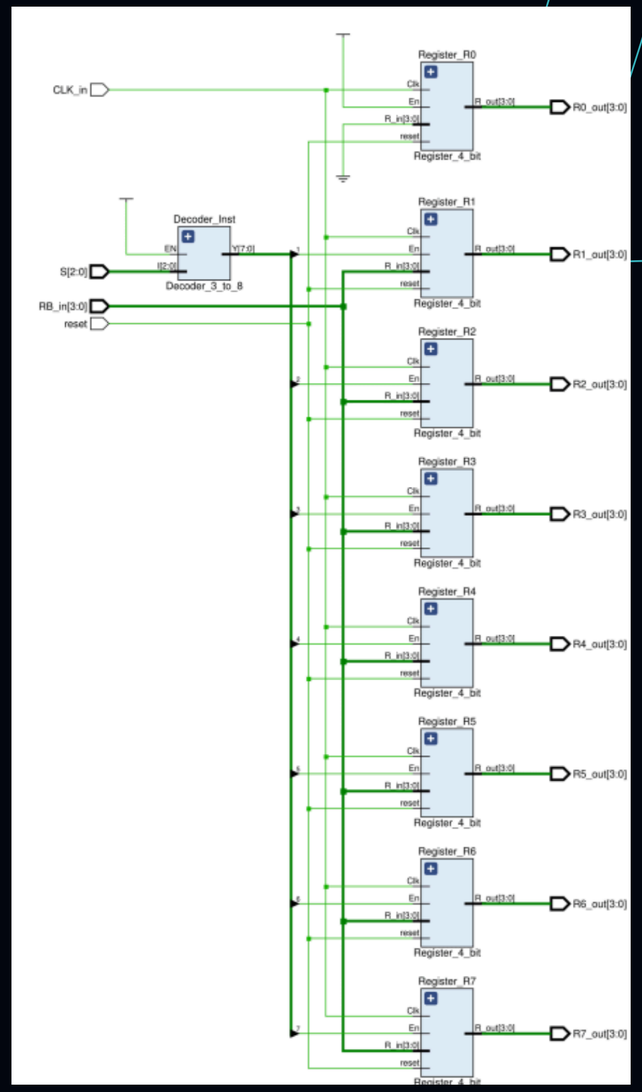
- It has select lines that determine which of the 8 inputs to select, using a 3 bit binary number
- t has 4 output lines (Y0, Y1, Y2, Y3) that carry the selected 4-bit input.
- Built using multiple 2 Way 4 Bit MUXs



REGISTER BANK

Introduction

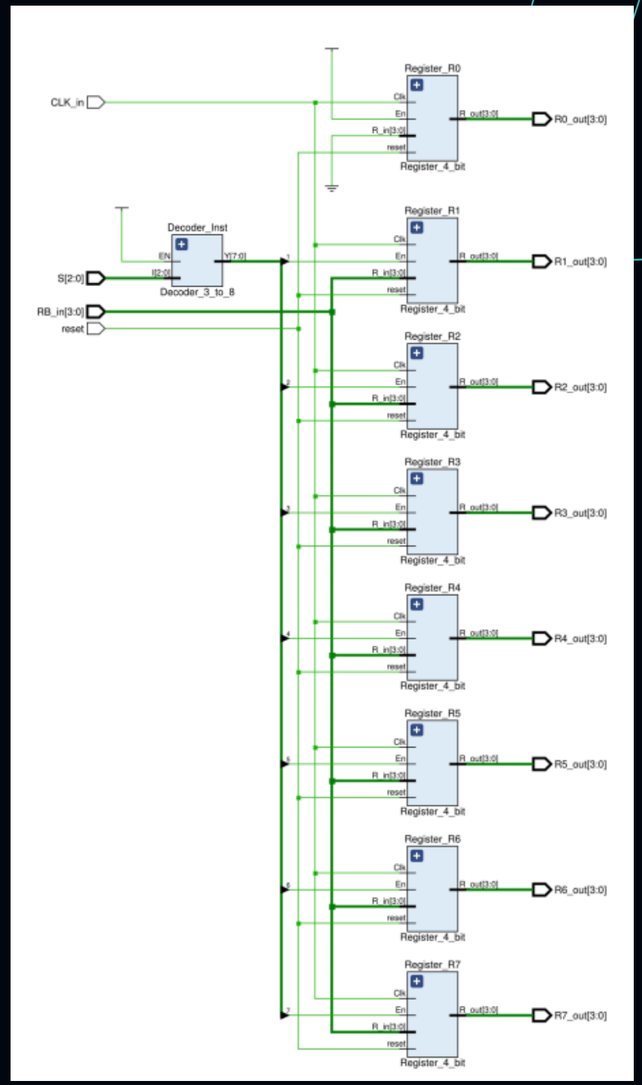
- Small, fast memory unit implement by severel registers.
- Eight 4 - bit registers, labeled R0 through R7.
- Used to store intermediate values during program execution.
- One special register : R0, always hold value 0000.



REGISTER BANK

Internal Design

- The Register Bank uses a 3-to-8 decoder to select which register to write data to.
- And each register is built using a component called Register_4_bit.
- Clock tick and enable is high, correspondig register stores the 4 - bit value

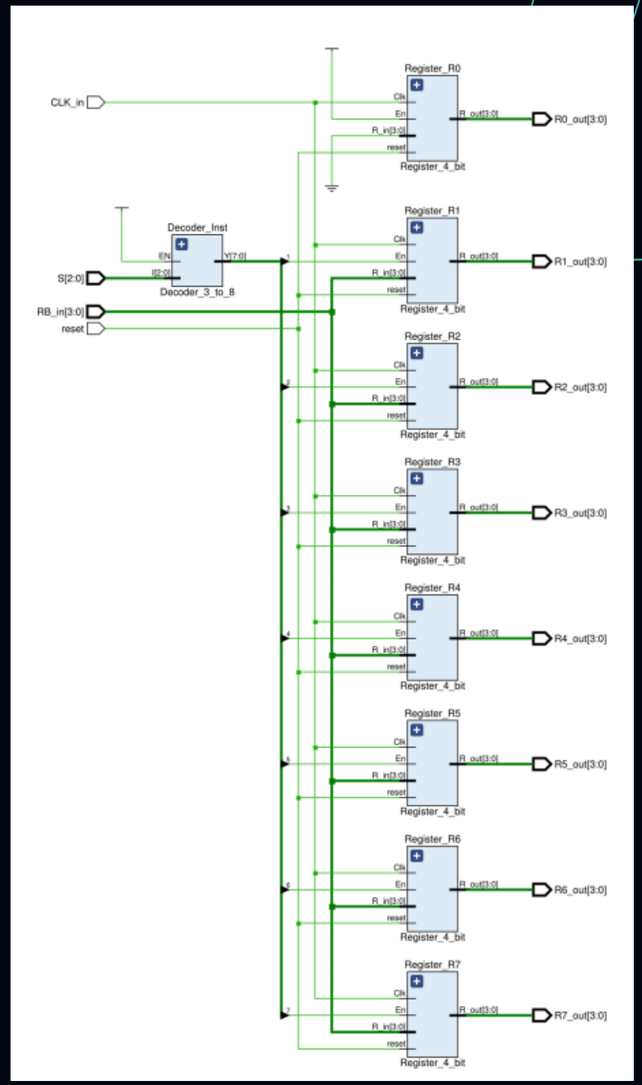


REGISTER BANK

Simulation & Output

In our simulation:

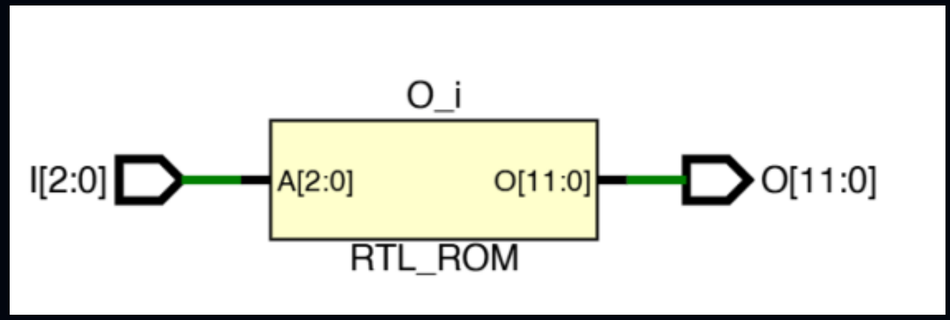
- We reset all registers to zero.
- Then we wrote values like 1010 to R1, 1100 to R2, and 1111 to R7.
- After each write operation, we checked the outputs to confirm the correct data was stored.



PROGRAM ROM

Introduction

- Program ROM is where we store the machine instructions that the nano-processor will execute.
- Each instruction is 12 bits long, stored 8 such instructions.
- ROM is addressed using a 3-bit input, which usually comes from the program counter.
- 3 bit - OPCODE
  3 - 5 bits - Register 1
  6 - 11 bits - Immidiate val



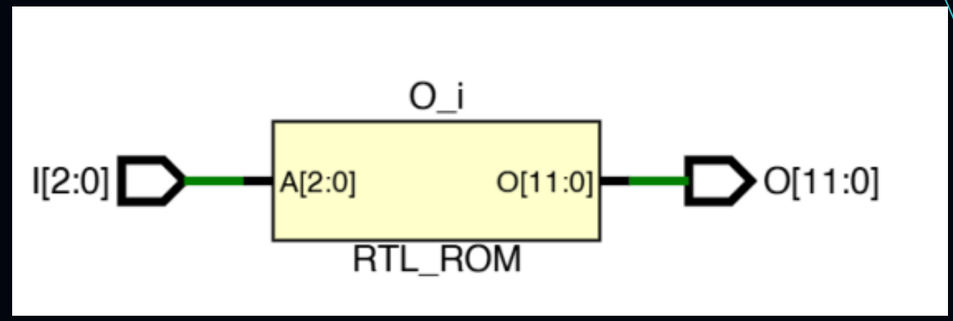
PROGRAM ROM

VHDL Design

- we implemented the ROM using an array of std_logic_vector.

- Here’s a small example:
  "100010000010", -- MOVI R1, 2
  "101110000011", -- MOVI R7, 3
  "100100000001", -- MOVI R2, 1
  "010100000000", -- NEG R2
  ...

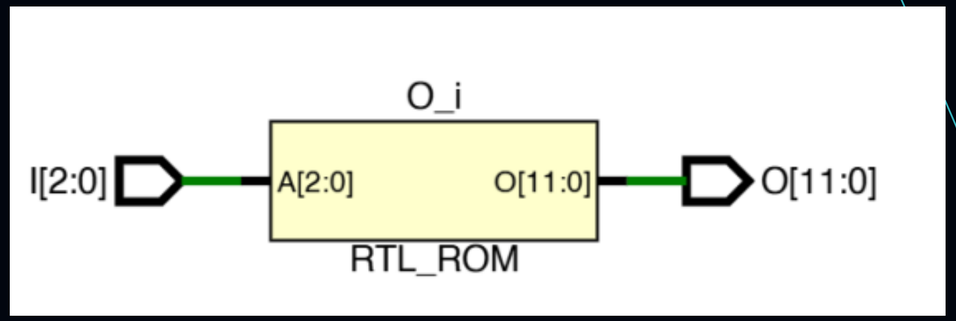
- Where I is the 3-bit address input, and O is the 12-bit output instruction.



PROGRAM ROM

Simulation & Example

- We simulated the ROM by feeding all 8 possible addresses from 000 to 111.
- Example:
  Address 000 returned 100010000010 – that's MOVI R1, 2.
- ROM is addressed using a 3-bit input, which usually comes from the program counter.

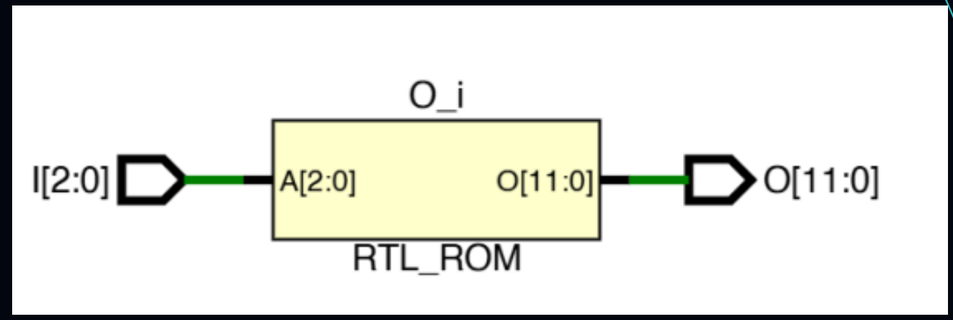


PROGRAM ROM

Conclusion

- The Register Bank gives our nano-processor 8 storage locations for temporary data. It uses a decoder and clocked logic for controlled updates.

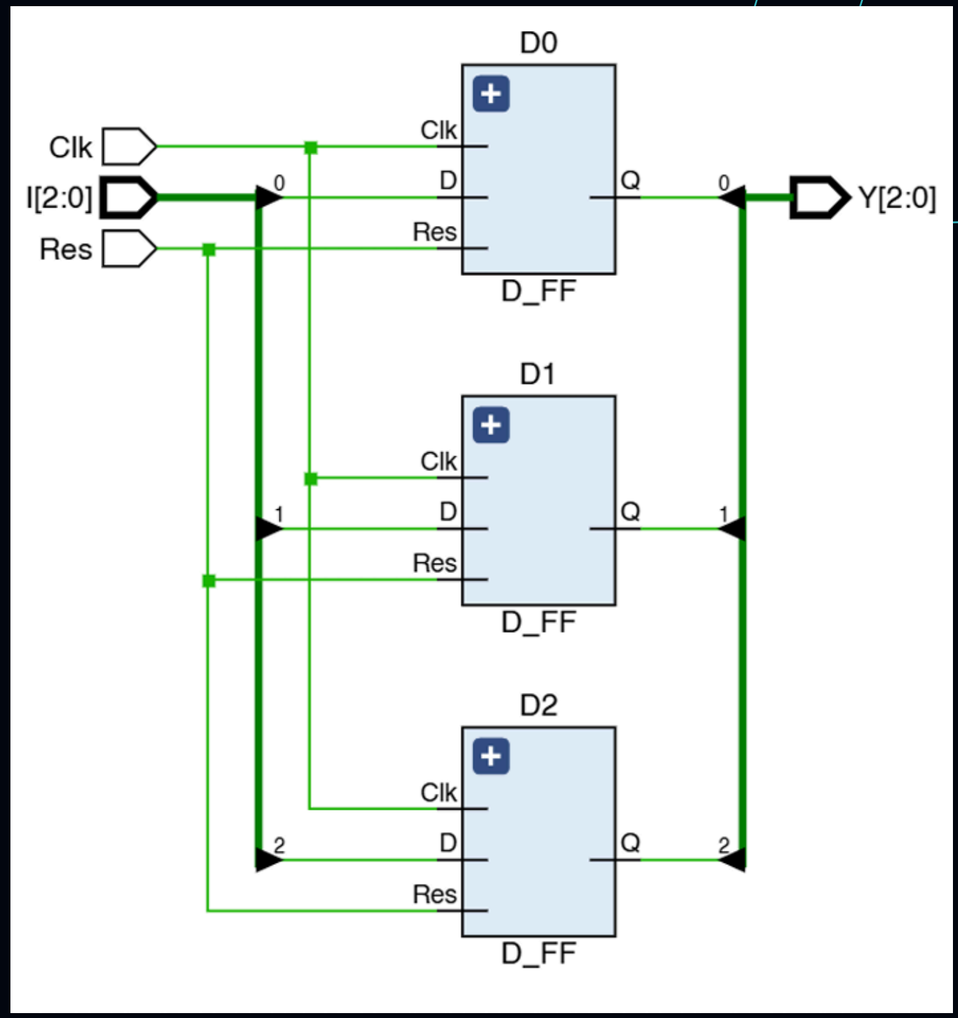
- The Program ROM provides the instruction stream that drives the processor’s behavior. Each instruction is 12 bits and is fetched using a 3-bit address



PROGRAM COUNTER

Introduction to the Program Counter

- A 3-bit register that holds the address of the next instruction to execute.
- Directs the processor to the correct instruction in Program ROM.
- After each instruction, PC is typically incremented to point to the next instruction.
- In case of jump instructions, PC can be overridden with a new address.



PROGRAM COUNTER

Internal Design and VHDL Code

- Built using three D Flip-Flops, one for each bit of the counter.
- Takes input 3 - bit input new instruction address.
- Controlled by clock (Clk) and reset (Res) signals.
- Output – Current address value (used as input to Program ROM).

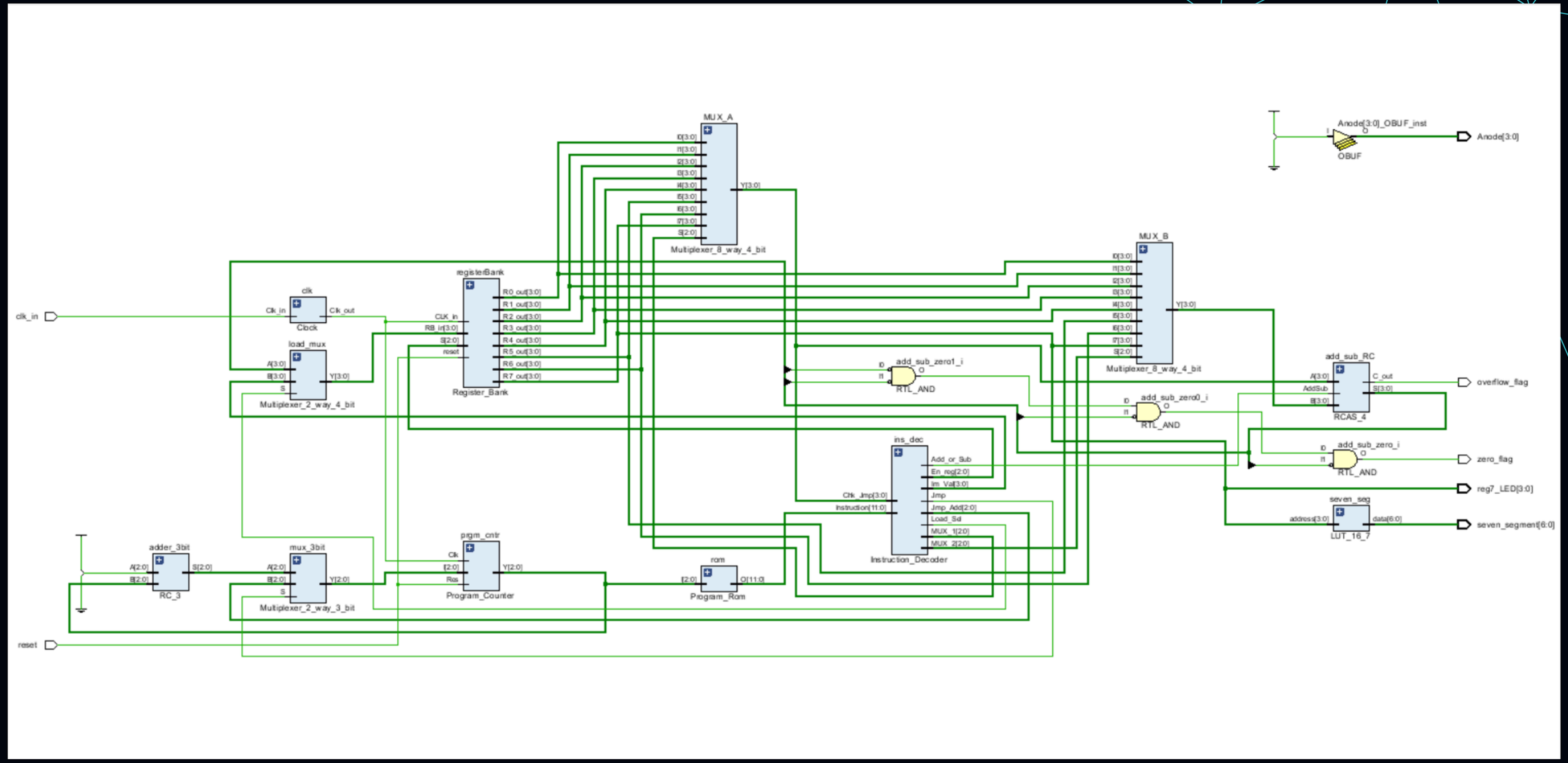
ROLE OF PC IN NANO-PROCESSOR OPERATION

Why the Program Counter is Crucial ?

Without PC: processor would not know what to execute next.

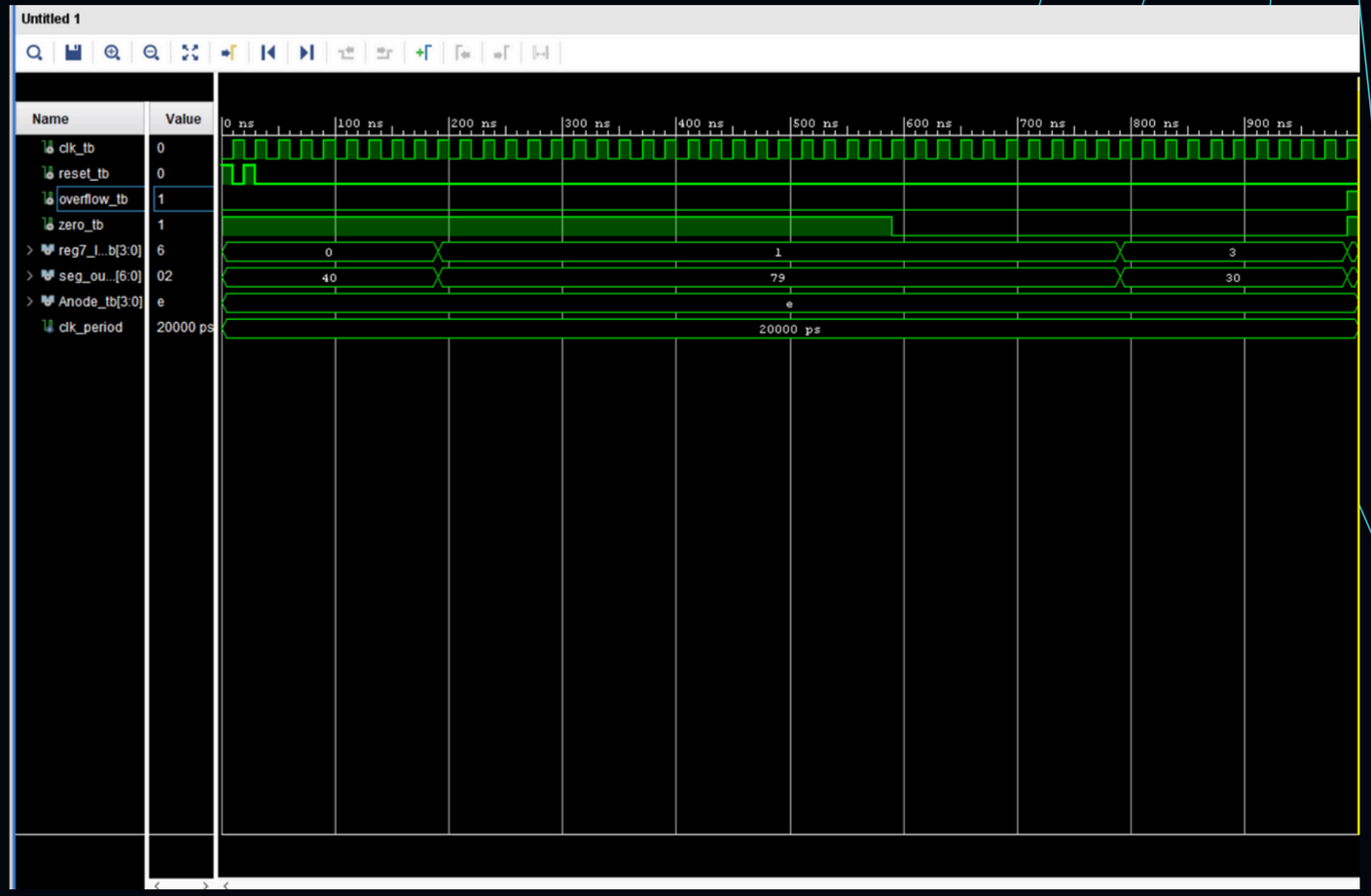
Works closely with: Program ROM (instruction fetching),Instruction Decoder (execution logic).

INTEGRATION



SIMULATION

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


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

architecture Behavioral of Program_Rom is

    type rom_type is array (0 to 6) of std_logic_vector (11 downto 0);
        signal program_ROM : rom_type := (
            "101110000001", --Movi R7,1
            "101100000010", --MOVI R6,2
            "101010000011", --MOVI R5,3
            "001111100000", --ADD R7, R6
            "001111010000", --ADD R7, R5
            "110010000110", --JZR R1, 7
             "110000000100"  --JZR R0, 5
            );

begin

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

end Behavioral;
```



IMPROVED NANO PROCESSOR

INSTRUCTION DECODER

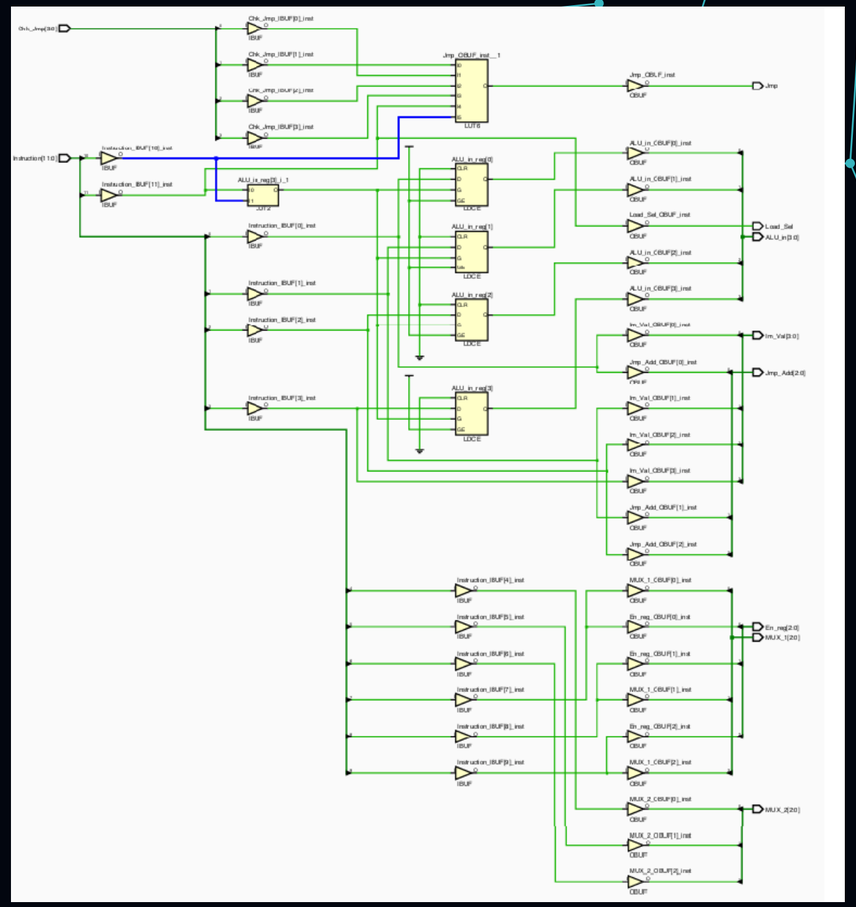
10	-	10 RRR 000 dddd Move the value dddd to register RRR
00	0000 - add	00 RaRaRa RbRbRb 0000 Add values in Ra, Rb and store in Ra
00	0001 - subtract	00 RaRaRa RbRbRb 0001 Subtract values in Ra, Rb and store in Ra
00	1000 - negate	1 1 R R R 0 0 0 0 d d d (If (R R R) = 0, program counter should call ROM (d d d), else program counter should increase it's value by 1)
11	-	00 RaRaRa 000 1000 Jump to the given instruction if check jump is = "0000"

INSTRUCTION DECODER

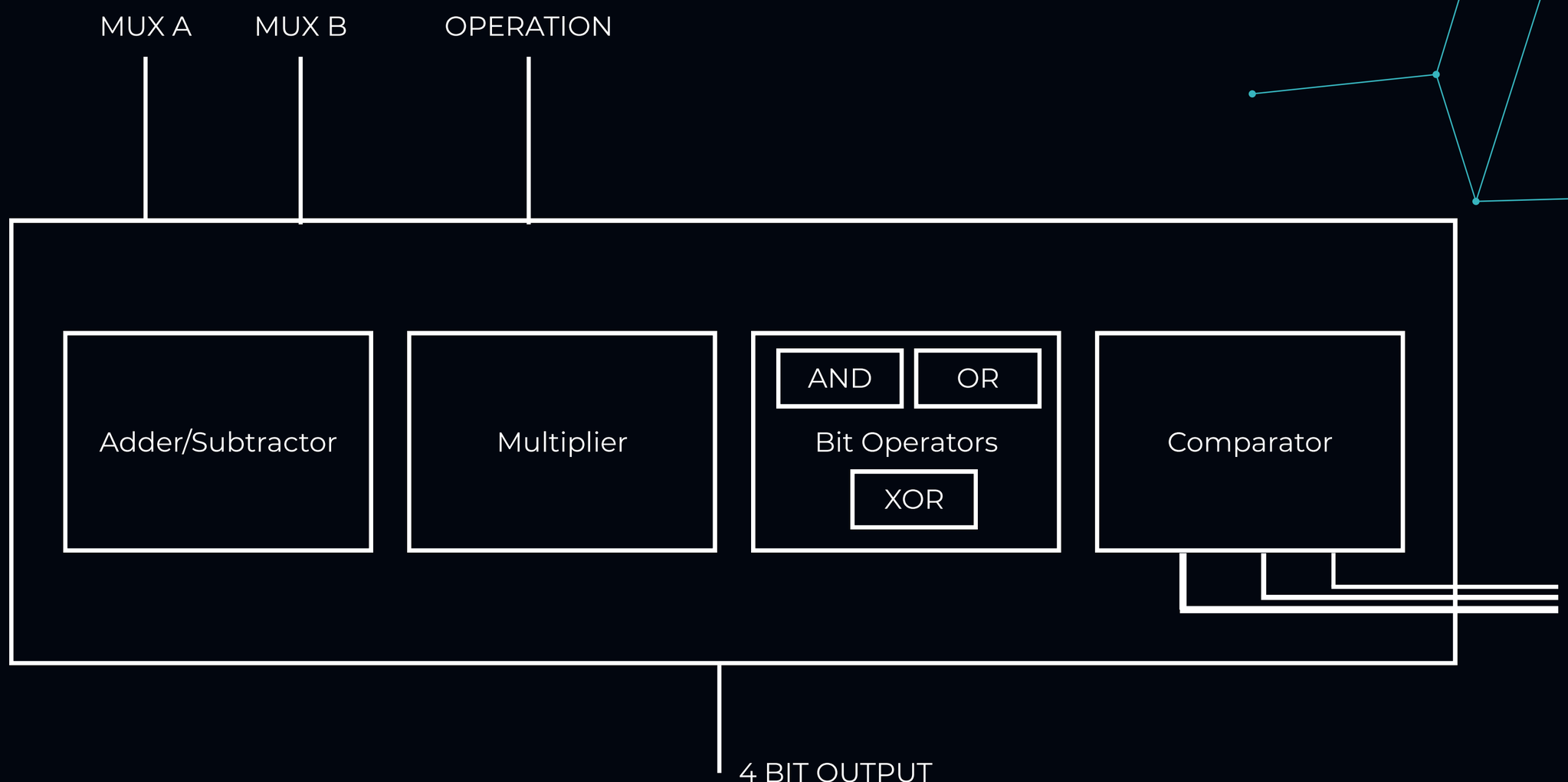
00	0010 - AND	00 RaRaRa 000 1000 Negate the value in Ra
00	0011 - OR	00 RaRaRa RbRbRb 0011 Bit operation OR to the values in Ra, Rb
00	0100 - XOR	00 RaRaRa RbRbRb 0100 Bit operation XOR to the values in Ra, Rb
00	0101 - multiply	00 RaRaRa RbRbRb 0101 Multiply Values in Ra, Rb and store in Ra
00	0111 - compare	00 RaRaRa RbRbRb 0111 Compare the value in Ra to Rb and provide an out signal saying whether it is less than, greater than or equal.

INSTRUCTION DECODER

- Breaks the 12 bit instruction to various components, each giving a component an input/command.
- The original instruction format was slightly changed in the improved nano processor.
- Gives instructions to the ALU to define the component which is used in the current operation.



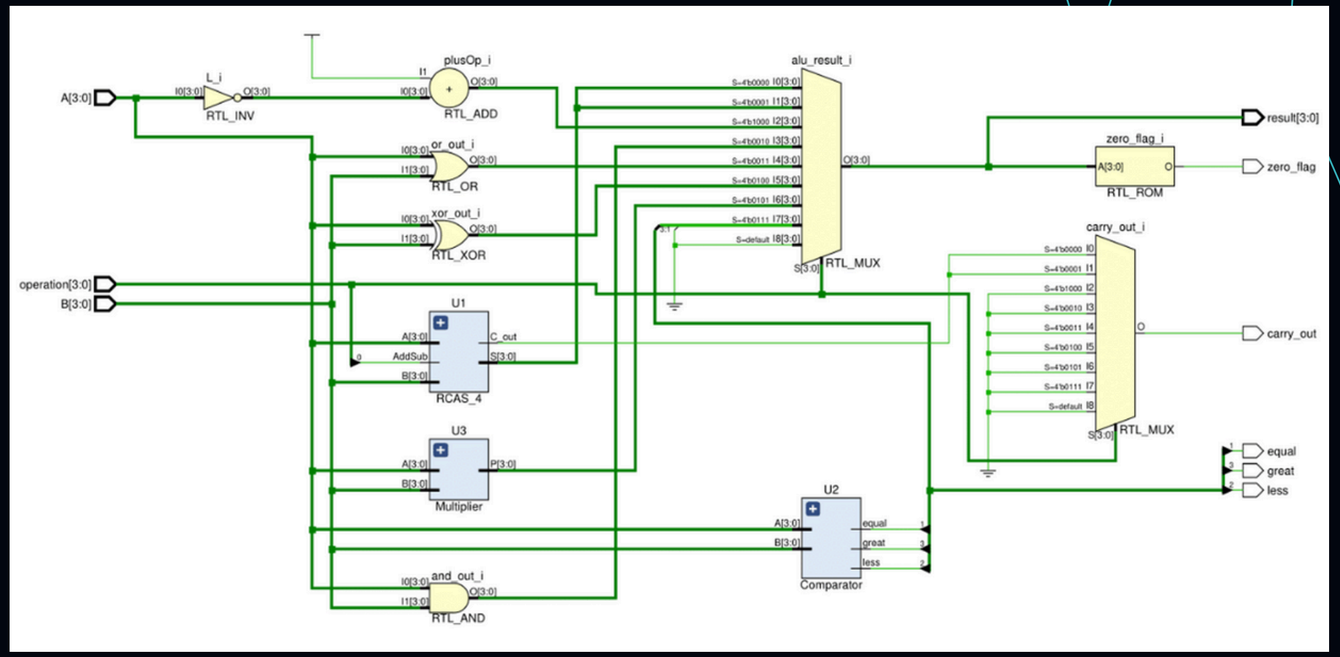
ALU


MUX A
MUX B
OPERATION
Adder/Subtractor
Multiplier
AND
OR
Bit Operators
XOR
Comparator
4 BIT OUTPUT

ALU

- Has an extra input compared to the basic nano processor. This input decides which component to use for that instruction.

- The ALU has 1, 4 bit out releasing the results of the operations, and 3 LED coming directly from the comparator.



ALU

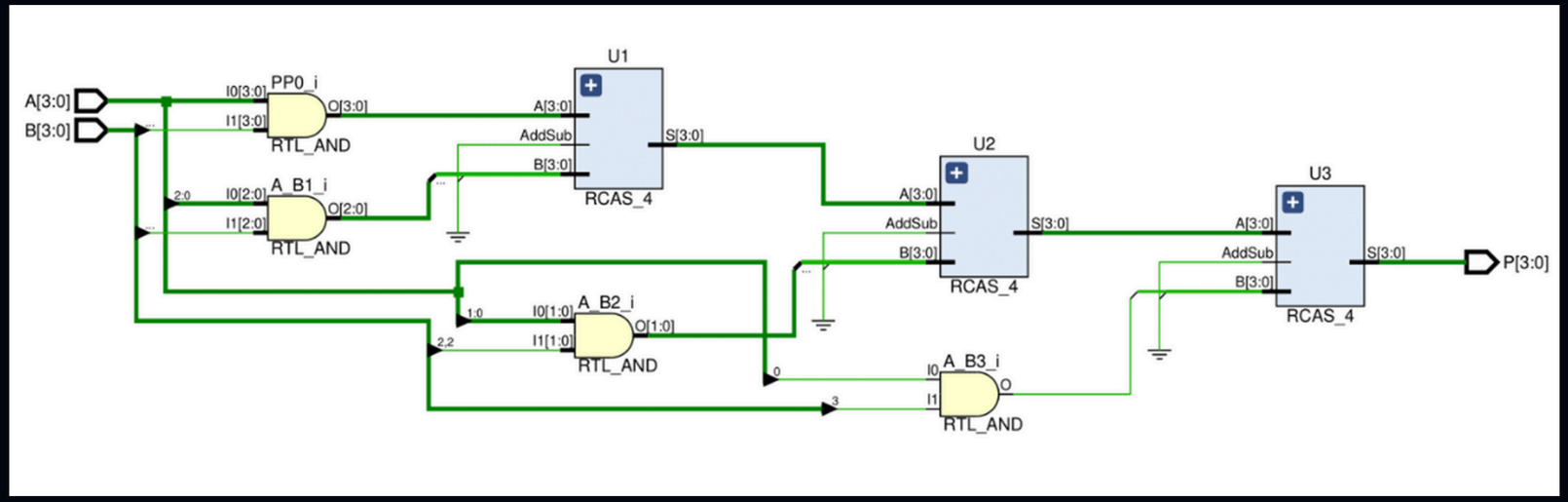
```vhdl
-- ALU Operation logic
process (operation, sum_out, and_out, or_out, xor_out, neg_out, cmp_eq, cmp_lt, cmp_gt, mul_out, carry_tmp)
begin
    case operation is
        when "0000" =>  -- ADD
            alu_result <= sum_out;
            carry_out <= carry_tmp;
        when "0001" =>  -- SUB
            alu_result <= sum_out;
            carry_out <= carry_tmp;
        when "1000" =>  -- NEG
            alu_result <= neg_out;
            carry_out <= '0';
        when "0010" =>  -- AND
            alu_result <= and_out;
            carry_out <= '0';
        when "0011" =>  -- OR
            alu_result <= or_out;
            carry_out <= '0';
        when "0100" =>  -- XOR
            alu_result <= xor_out;
            carry_out <= '0';
        when "0101" =>  -- MUL
            alu_result <= mul_out;
            carry_out <= '0';
        when "0111" =>  -- CMP: encoded
            alu_result <= cmp_gt & cmp_lt & cmp_eq & '0';  -- Optional visual encoding
            carry_out <= '0';
        when others =>
            alu_result <= (others => '0');
            carry_out <= '0';
    end case;
end process;
```

```vhdl
-- Instantiate adder/subtractor
U1: RCAS_4 port map (
    A      => A,
    B      => B,
    AddSub => operation(0),  -- 0 for ADD, 1 for SUB
    S      => sum_out,
    C_out  => carry_tmp
);

-- Bitwise logic
and_out <= A and B;
or_out  <= A or B;
xor_out <= A xor B;

-- Negation (2's complement of A)
neg_out <= std_logic_vector(unsigned(not A) + 1);

-- Comparator
U2: Comparator port map (
    A     => A,
    B     => B,
    equal => cmp_eq,
    less  => cmp_lt,
    great => cmp_gt
);

-- Multiplier
U3: Multiplier port map (
    A => A,
    B => B,
    P => mul_out
);
```

ALU VHDL CODE

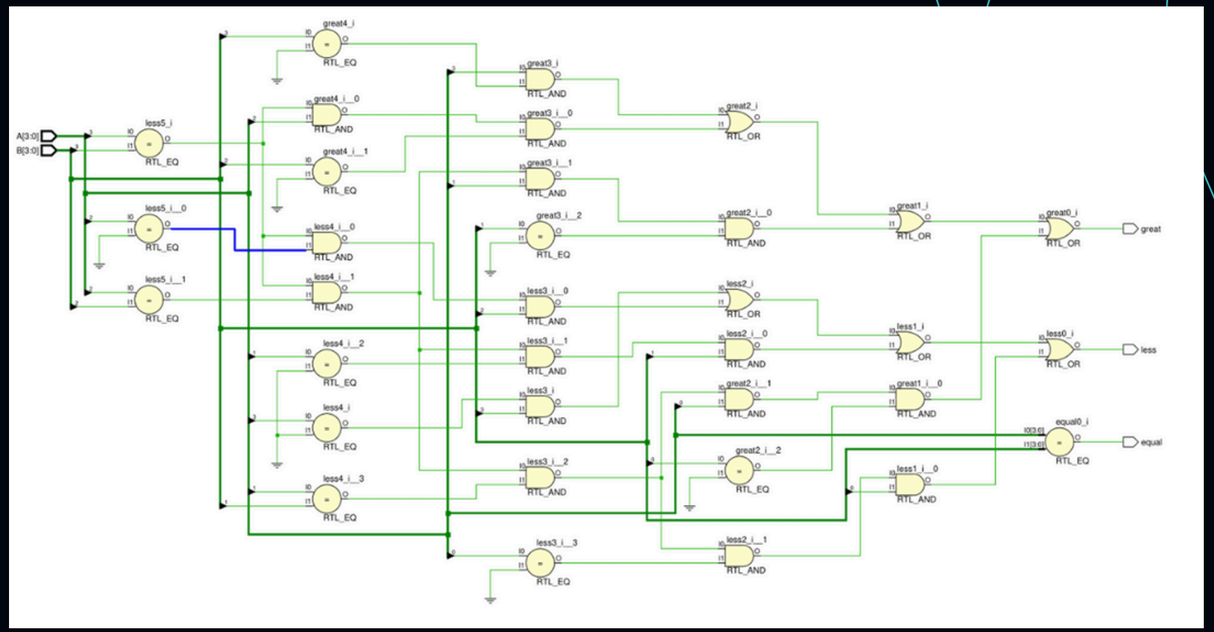
4 BIT MULTIPLIER

- Multiplies two 4-bit binary numbers to produce the last 4-bits of the result.
- Ripple carry adders are integrated to the component to execute addition within the component to output the correct result.
- A sub component of the ALU.

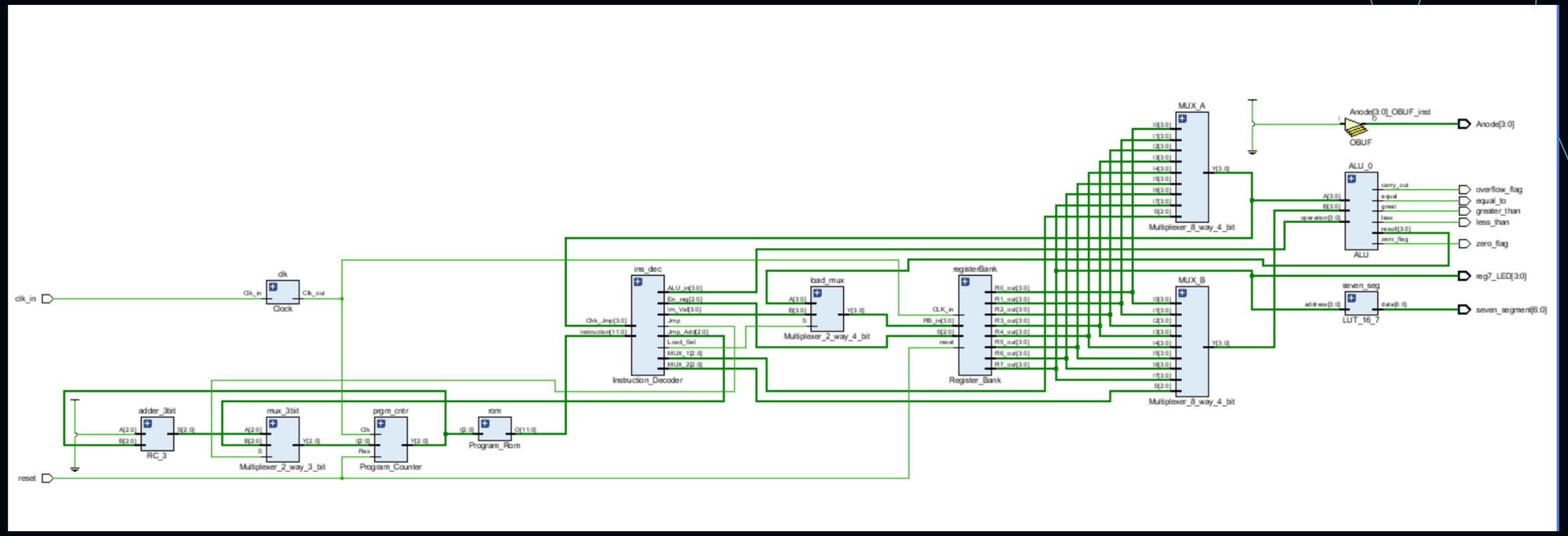


COMPARATOR

- Compares the 4 bit inputs to check whether they are greater than, less than or equal to each other.
- Checks the 4 bit inputs bitwise, from the MSB towards the LSB.
- They contain 3 direct outputs from the nano processor, mapped to 3 LED bulbs of the BASYS 3 FPGA board.
- A sub component of the ALU.

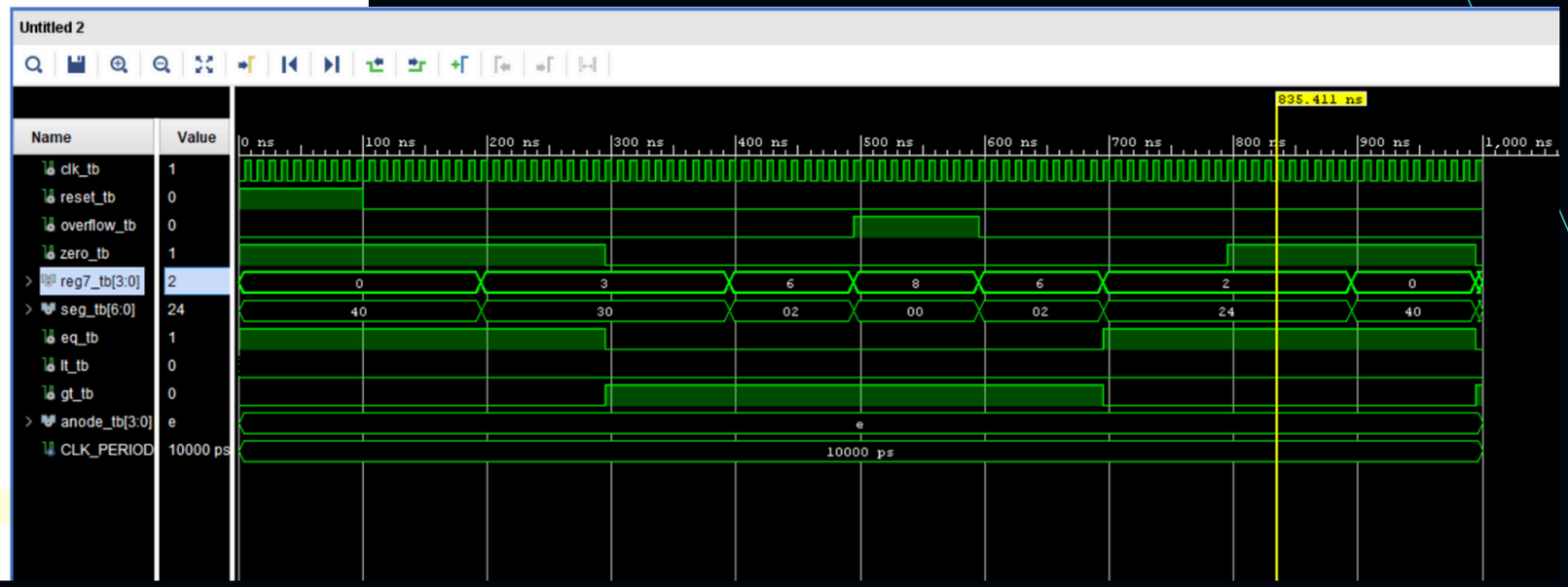


INTEGRATION



SIMULATION

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

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

architecture Behavioral of Program_Rom is
    type rom_type is array (0 to 8) of std_logic_vector (11 downto 0);
    signal program_ROM : rom_type := (
           0  => "101110000011",   -- MOVI R7, 3
           1  => "100100000010",   -- MOVI R2, 2
           2  => "001110100101",   -- MUL  R7, R2
           3  => "001110100000",   -- ADD  R7, R2
           4  => "001110100001",   -- SUB  R7, R2
           5  => "001110100010",   -- AND  R7, R2
           6  => "001110100011",   -- OR   R7, R2
           7  => "001110100100",   -- XOR  R7, R2
           8  => "001110100111"    -- CMP  R7, R2
    );

begin
    O <= program_ROM(to_integer(unsigned(I)));
end Behavioral;
```



GROUP MEMBERS

230010L - ABEYSUNDARA K.N.B

230601B - SENEVIRATNE N.S

230361L - H.M.PASINDU LAKMAL

230678N - PANKAJA WAIDYASEKARA

THANK YOU