Advanced Digital Logic
ENGR – UH 2310

Lab 4

Group 1
Spring, 2025

This report is entirely our own work, and we have kept a copy for our own records. We are aware of the University's policies on cheating, plagiarism, and the resulting consequences of their breach.

Submitted by:

Name	**Participation**	**Net ID**
Damiane Kapanadze	Task-1,Task-3, Task-4, Lab Report	dk4770
Seoyoon Jung	Task-1,Task-3, Task-4, Lab Report	sj4260
Sipan Hovsepian	Test Bench, Task-1, Task-2, Task-3, Lab Report	sh7437

Contents

1 Task 1

1.1 VHDL code for decoder

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


use work.common.all;


entity Decoder is
    port (  instruction_in : in STD_LOGIC_VECTOR (15 downto 0);


        opcode_out : out opcode_type;


        Rd_addr_out : out STD_LOGIC_VECTOR (2 downto 0);
        Rs1_addr_out : out STD_LOGIC_VECTOR (2 downto 0);
        Rs2_addr_out : out STD_LOGIC_VECTOR (2 downto 0);


        immediate_out : out STD_LOGIC_VECTOR (13 downto 0)
         );
end Decoder;


architecture Behavioral of Decoder is


--TODO add signals as needed
signal opcode_internal : opcode_type;
signal Rd_addr_internal : STD_LOGIC_VECTOR (2 downto 0);
signal Rs1_addr_internal : STD_LOGIC_VECTOR (2 downto 0);
signal Rs2_addr_internal : STD_LOGIC_VECTOR (2 downto 0);
signal tail_internal : STD_LOGIC_VECTOR (2 downto 0);




begin
    process (instruction_in, opcode_internal, Rd_addr_internal, Rs1_addr_internal, Rs2_addr_internal, tail_internal)
        begin
        --TODO implement extraction of remaining parts of the instruction


        --Extract opcode
        opcode_internal <= std_logic_vector_to_opcode_type( instruction_in(15 downto 12) );
        Rd_addr_internal <= instruction_in(11 downto 9);
        Rs1_addr_internal <= instruction_in(8 downto 6);
        Rs2_addr_internal <= instruction_in(5 downto 3);
        tail_internal <= instruction_in(2 downto 0);



    --Assign outputs
        opcode_out <= opcode_internal;
        Rd_addr_out <= Rd_addr_internal;
        Rs1_addr_out <= Rs1_addr_internal;
        Rs2_addr_out <= Rs2_addr_internal;




    --TODO derive immediate value, depending on opcode_internal
        case opcode_internal is
            when OP_ANDI =>
                immediate_out <="11111111" & Rs2_addr_internal & tail_internal;

            when OP_ORI | OP_XORI =>
                immediate_out <="00000000" & Rs2_addr_internal & tail_internal;

            when OP_ADDI | OP_SUBI =>
                if Rs2_addr_internal(2) = '0' then
                    immediate_out <="00000000" & Rs2_addr_internal & tail_internal;
                else
                    immediate_out <="11111111" & Rs2_addr_internal & tail_internal;
                end if;

            when OP_BLT  | OP_BE | OP_JMP=>
                if Rd_addr_internal(2) = '0' then
                    immediate_out <="00000000" & Rd_addr_internal & tail_internal;
                else
                    immediate_out <="11111111" & Rd_addr_internal & tail_internal;
                end if;

            when OP_SLL | OP_SRL =>
                immediate_out <="00000000000" & tail_internal;

```

```vhdl
                    when others => immediate_out <= "XXXXXXXXXXXXXX";

                end case;
        end process;

end Behavioral;
```

Listing 1: VHDL code for decoder

1.2 VHDL code for controller

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

use work.common.all;

entity Controller is
    port (  opcode : in opcode_type;

        operand_1 : out STD_LOGIC_VECTOR (13 downto 0);
        operand_2 : out STD_LOGIC_VECTOR (13 downto 0);

        result : in STD_LOGIC_VECTOR (13 downto 0);

        curr_PC : in STD_LOGIC_VECTOR (6 downto 0);

        new_PC : out STD_LOGIC_VECTOR (6 downto 0);
        PC_we : out STD_LOGIC;
        PC_incr : out STD_LOGIC;

        Rs1_data : in STD_LOGIC_VECTOR (13 downto 0);
        Rs2_data : in STD_LOGIC_VECTOR (13 downto 0);
        immediate : in STD_LOGIC_VECTOR (13 downto 0);

        Rd_we : out STD_LOGIC;
        Rd_data : out STD_LOGIC_VECTOR (13 downto 0)
         );
end Controller;

architecture Behavioral of Controller is

begin

control : process (opcode, Rs1_data, Rs2_data, result, immediate,
    curr_PC)
begin
    -- default assignments, can be overwritten below
    operand_1 <= Rs1_data;
    operand_2 <= Rs2_data;

    Rd_we <= '0';
    Rd_data <= result;

    PC_we <= '0';
    new_PC <= (6 downto 0 => 'X');

    PC_incr <= '0';

    -- regular operations with Rs1, Rs2, Rd
    -- TODO consider remaining cases
    if ((opcode = OP_AND) or (opcode = OP_OR) or (opcode = OP_XOR
       ) or (opcode = OP_ADD) or (opcode = OP_SUB)) then

        operand_1 <= Rs1_data;
        operand_2 <= Rs2_data;

        Rd_we <= '1';
        PC_incr <= '1';

    -- TODO implement remaining cases
    elsif ((opcode = OP_ANDI) or (opcode = OP_ORI) or (opcode =
       OP_XORI) or (opcode = OP_ADDI) or (opcode = OP_SUBI) or (
       opcode = OP_SLL) or (opcode = OP_SRL)) then

        operand_1 <= Rs1_data;
        operand_2 <= immediate;

        Rd_we <= '1';
        PC_incr <= '1';

    elsif ((opcode = OP_BLT)) then
        if (signed(Rs1_data) < signed(Rs2_data)) then
            operand_1 <= "0000000" & Curr_PC;
            operand_2 <= immediate;
            PC_we <= '1';
            new_PC <= result (6 downto 0);
        else
            PC_incr <= '1';
        end if;

    elsif ((opcode = OP_BE)) then
        if (signed(Rs1_data) = signed(Rs2_data)) then
            operand_1 <= "0000000" & Curr_PC;
            operand_2 <= immediate;
            PC_we <= '1';
            new_PC <= result (6 downto 0);
        else
            PC_incr <= '1';
        end if;

    elsif ((opcode = OP_JMP)) then
```

```
        operand_1 <= "0000000" & Curr_PC;
        operand_2 <= immediate;
        PC_we <= '1';
        new_PC <= result (6 downto 0);


    -- only OP_HALT should remain
    else
        operand_1 <= (13 downto 0 => 'X');
        operand_2 <= (13 downto 0 => 'X');

        Rd_data <= (13 downto 0 => 'X');
    end if;
end process;

end Behavioral;
```

Listing 2: VHDL code for Controller

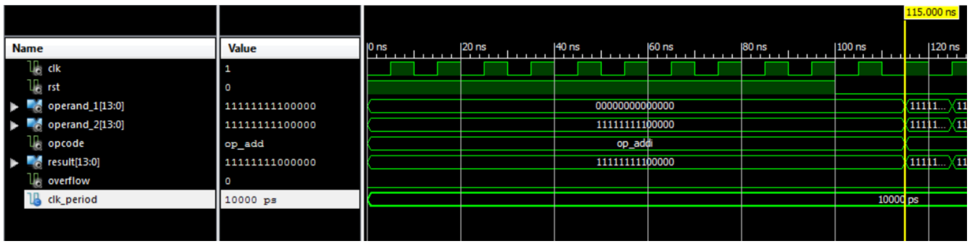
1.3 VHDL code for ALU unit

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


use work.common.all;


entity ALU is
    port (  operand_1 : in STD_LOGIC_VECTOR (13 downto 0);
        operand_2 : in STD_LOGIC_VECTOR (13 downto 0);


        opcode : in opcode_type;


        result : out STD_LOGIC_VECTOR (13 downto 0);
        overflow : out STD_LOGIC


         );
end ALU;


architecture Behavioral of ALU is


signal result_internal: STD_LOGIC_VECTOR (13 downto 0);


```

```vhdl
begin


result <= result_internal;


calculate : process (operand_1, operand_2, opcode)
begin
    -- signed ADD operations
    -- TODO consider remaining cases
    if ((opcode = OP_ADD) or (opcode = OP_ADDI) ) then
        result_internal <= std_logic_vector( signed(operand_1) +
           signed(operand_2) );


    -- SLL operation
    elsif (opcode = OP_SLL) then
        result_internal <= std_logic_vector( shift_left(unsigned(
           operand_1), to_integer(unsigned(operand_2))) );


    -- TODO implement remaining operations
    elsif ((opcode = OP_AND) or (opcode = OP_ANDI) ) then
        result_internal <= std_logic_vector(operand_1 and
           operand_2);

    elsif ((opcode = OP_OR) or (opcode = OP_ORI) ) then
        result_internal <= std_logic_vector(operand_1 or
           operand_2);

    elsif ((opcode = OP_XOR) or (opcode = OP_XORI) ) then
        result_internal <= std_logic_vector(operand_1 xor
           operand_2);

    elsif (opcode = OP_SRL) then
        result_internal <= std_logic_vector( shift_right(unsigned
           (operand_1), to_integer(unsigned(operand_2))) );

    elsif (opcode = OP_SLL) then
        result_internal <= std_logic_vector( shift_left(unsigned(
           operand_1), to_integer(unsigned(operand_2))) );

    elsif ((opcode = OP_SUB) or (opcode = OP_SUBI) ) then
        result_internal <= std_logic_vector( signed(operand_1) -
           signed(operand_2) );

    elsif ((opcode = OP_BLT) or (opcode = OP_BE) or (opcode =
       OP_JMP)) then
        result_internal <= std_logic_vector( signed(operand_1) +
           signed(operand_2) );

    -- only OP_HALT should remain
    else
        result_internal <= (13 downto 0 => 'X');


    end if;


end process;


-- TODO implement detection of overflow for all signed arithmetic
    operations
ofl : process (operand_1, operand_2, result_internal, opcode)
begin
    if opcode = OP_ADD or opcode = OP_ADDI then
        if (signed(operand_1) > 0  and signed(operand_2) > 0 and
           signed(result_internal) < 0) or (signed(operand_1) < 0
             and signed(operand_2) < 0 and signed(result_internal
           ) > 0) then
            overflow <= '1';
        else
            overflow <= '0';
        end if;

    elsif opcode = OP_SUB or opcode = OP_SUBI then
        if (signed(operand_1) < 0  and signed(operand_2) > 0 and
           signed(result_internal) > 0) or (signed(operand_1) > 0
             and signed(operand_2) < 0 and signed(result_internal
           ) < 0) then
            overflow <= '1';
        else
            overflow <= '0';
        end if;

    else
        overflow <= '0';
    end if;
end process;


end Behavioral;
```

Listing 3: VHDL code for ALU unit

1.4 Report on clock cycles per instruction



Name	Value
clk	1
rst	0
operand_1[13:0]	11111111100000
operand_2[13:0]	11111111100000
opcode	op_add
result[13:0]	11111111000000
overflow	0
clk_period	10000 ps

Figure 1: Clock cycles per instruction

As seen from Figure 1, the addition instruction takes **11** clock cycles to execute.

1.5 VHDL code FPGA top-level module

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

use work.common.all;

entity top_processor_FPGA is
    port ( next_instr : in STD_LOGIC;
        clk : in STD_LOGIC;
        rst : in STD_LOGIC;

        operand_1 : out STD_LOGIC_VECTOR (13 downto 0);
        operand_2 : out STD_LOGIC_VECTOR (13 downto 0);

        opcode : out opcode_type;

        result : out STD_LOGIC_VECTOR (13 downto 0);

        overflow : out STD_LOGIC;

        seg_bits : out STD_LOGIC_VECTOR(0 to 7);
        seg_an : out STD_LOGIC_VECTOR(3 downto 0)
    );
end top_processor_FPGA;

architecture Behavioral of top_processor_FPGA is

-- component declarations
    component Display_Controller
        port (
            clk         : in  STD_LOGIC;
            rst         : in  STD_LOGIC;
            opcode      : in  opcode_type;
            operand_1   : in  STD_LOGIC_VECTOR (13 downto 0);
            operand_2   : in  STD_LOGIC_VECTOR (13 downto 0);
```

```vhdl
                result      : in  STD_LOGIC_VECTOR (13 downto 0);
                overflow    : in  STD_LOGIC;
                seg_bits    : out STD_LOGIC_VECTOR (0 to 7);
                seg_an      : out STD_LOGIC_VECTOR (3 downto 0)
            );
        end component;

component Instructions_ROM
    port (  address_in : in STD_LOGIC_VECTOR (6 downto 0);
        data_out : out STD_LOGIC_VECTOR (15 downto 0)
         );
end component;

component PC
    port (  clk : in STD_LOGIC;
        rst: in STD_LOGIC;

        PC_in : in STD_LOGIC_VECTOR (6 downto 0);
        PC_out : out STD_LOGIC_VECTOR (6 downto 0);

        PC_we : in STD_LOGIC;
        PC_incr : in STD_LOGIC
         );
end component;

component Registers
    port (  clk : in STD_LOGIC;
        rst: in STD_LOGIC;

        Rs1_addr_in : in STD_LOGIC_VECTOR (2 downto 0);
        Rs1_data_out : out STD_LOGIC_VECTOR (13 downto 0);

        Rs2_addr_in : in STD_LOGIC_VECTOR (2 downto 0);
        Rs2_data_out : out STD_LOGIC_VECTOR (13 downto 0);

        Rd_addr_in : in STD_LOGIC_VECTOR (2 downto 0);
        Rd_data_in : in STD_LOGIC_VECTOR (13 downto 0);
        Rd_we : in STD_LOGIC
         );
end component;

component Decoder
    port (  instruction_in : in STD_LOGIC_VECTOR (15 downto 0);

        opcode_out : out opcode_type;

        Rd_addr_out : out STD_LOGIC_VECTOR (2 downto 0);
        Rs1_addr_out : out STD_LOGIC_VECTOR (2 downto 0);
        Rs2_addr_out : out STD_LOGIC_VECTOR (2 downto 0);

        immediate_out : out STD_LOGIC_VECTOR (13 downto 0)
        );
end component;

component Controller
    port (  opcode : in opcode_type;

        operand_1 : out STD_LOGIC_VECTOR (13 downto 0);
        operand_2 : out STD_LOGIC_VECTOR (13 downto 0);

        result : in STD_LOGIC_VECTOR (13 downto 0);

        curr_PC : in STD_LOGIC_VECTOR (6 downto 0);

        new_PC : out STD_LOGIC_VECTOR (6 downto 0);
        PC_we : out STD_LOGIC;
        PC_incr : out STD_LOGIC;

        Rs1_data : in STD_LOGIC_VECTOR (13 downto 0);
        Rs2_data : in STD_LOGIC_VECTOR (13 downto 0);
        immediate : in STD_LOGIC_VECTOR (13 downto 0);

        Rd_we : out STD_LOGIC;
        Rd_data : out STD_LOGIC_VECTOR (13 downto 0)
         );
end component;

component ALU
    port (  operand_1 : in STD_LOGIC_VECTOR (13 downto 0);
        operand_2 : in STD_LOGIC_VECTOR (13 downto 0);

        opcode : in opcode_type;

        result : out STD_LOGIC_VECTOR (13 downto 0);
        overflow : out STD_LOGIC
         );
end component;

--
-- internal signals
--

-- instructions
signal curr_PC : STD_LOGIC_VECTOR (6 downto 0);
signal instruction : STD_LOGIC_VECTOR (15 downto 0);

signal new_PC : STD_LOGIC_VECTOR (6 downto 0);
signal PC_we : STD_LOGIC;
signal PC_incr : STD_LOGIC;

-- decoder and controller
signal opcode_internal : opcode_type;

signal Rd_addr : STD_LOGIC_VECTOR (2 downto 0);
signal Rs1_addr : STD_LOGIC_VECTOR (2 downto 0);
signal Rs2_addr : STD_LOGIC_VECTOR (2 downto 0);
signal immediate : STD_LOGIC_VECTOR (13 downto 0);

-- registers
signal Rd_data : STD_LOGIC_VECTOR (13 downto 0);
signal Rs1_data : STD_LOGIC_VECTOR (13 downto 0);
signal Rs2_data : STD_LOGIC_VECTOR (13 downto 0);
signal Rd_we : STD_LOGIC;

-- ALU
signal operand_1_internal : STD_LOGIC_VECTOR (13 downto 0);
signal operand_2_internal : STD_LOGIC_VECTOR (13 downto 0);
signal result_internal : STD_LOGIC_VECTOR (13 downto 0);

--DISP
signal overflow_internal : STD_LOGIC;

begin

overflow <= overflow_internal;

-- simple wiring of global signals
operand_1 <= operand_1_internal;
operand_2 <= operand_2_internal;
result <= result_internal;
opcode <= opcode_internal;

--
-- component instances with port maps; nothing else is allowed
   for this top-level module
--

Instructions_ROM_inst : Instructions_ROM
    port map (curr_PC, instruction);

Decoder_inst : Decoder
    port map (instruction, opcode_internal, Rd_addr, Rs1_addr,
       Rs2_addr, immediate);

Controller_inst : Controller
    port map (opcode_internal, operand_1_internal,
       operand_2_internal, result_internal, curr_PC, new_PC,
       PC_we, PC_incr, Rs1_data, Rs2_data, immediate, Rd_we,
       Rd_data);

PC_inst : PC
    port map (next_instr, rst, new_PC, curr_PC, PC_we, PC_incr);

```

```vhdl
Registers_inst : Registers
    port map (next_instr, rst, Rs1_addr, Rs1_data, Rs2_addr,
       Rs2_data, Rd_addr, Rd_data, Rd_we);

ALU_inst : ALU
    port map (operand_1_internal, operand_2_internal,
       opcode_internal, result_internal, overflow_internal);


display_i : Display_Controller
        port map (
            clk        => clk,
            rst        => rst,
            opcode     => opcode_internal,
            operand_1  => operand_1_internal,
            operand_2  => operand_2_internal,
            result     => result_internal,
            overflow   => overflow_internal,
            seg_bits   => seg_bits,
            seg_an     => seg_an
        );

end Behavioral;
```

Listing 4: VHDL code for FPGA top-level module

1.6 Floorplan PDF/screenshot

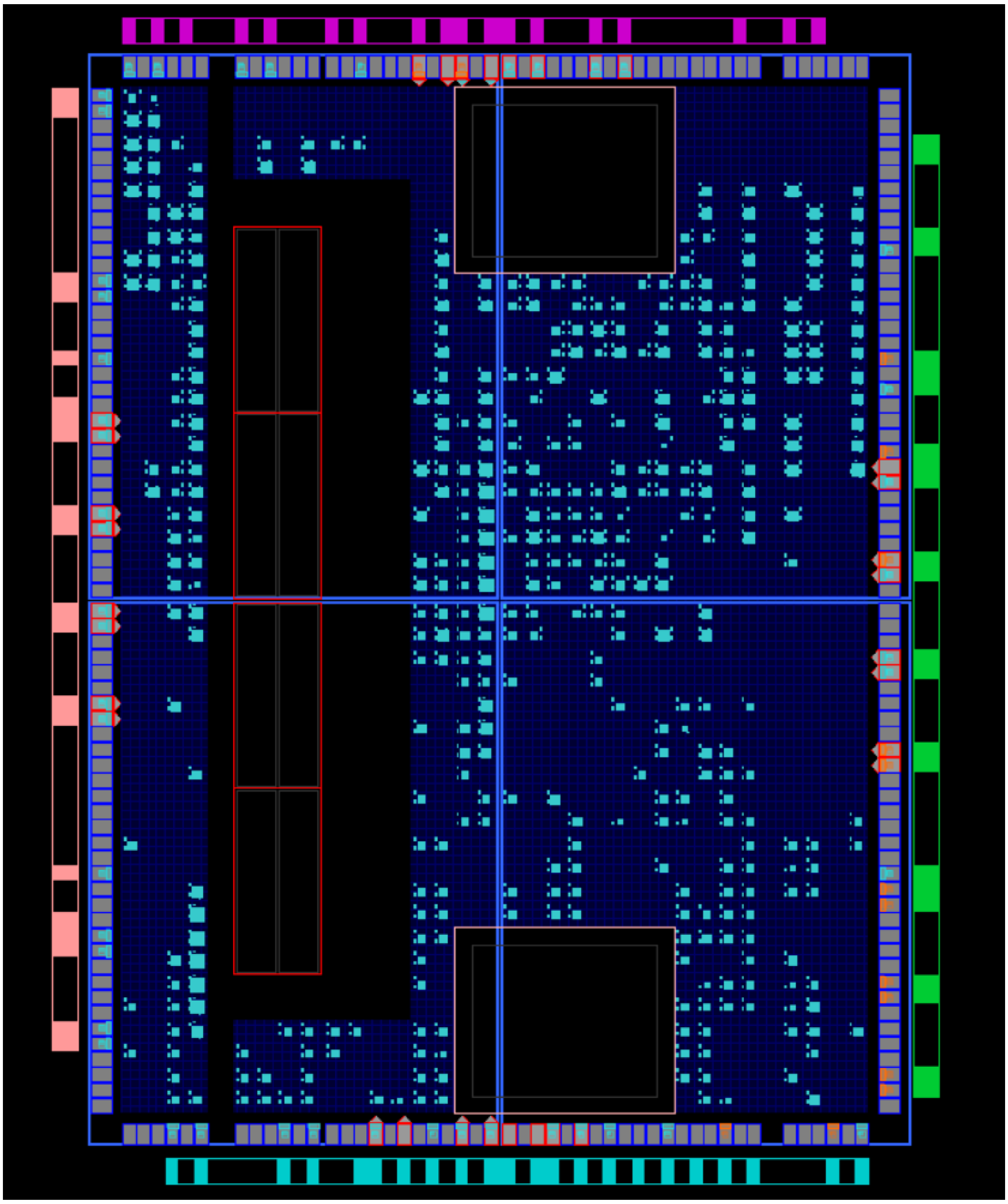
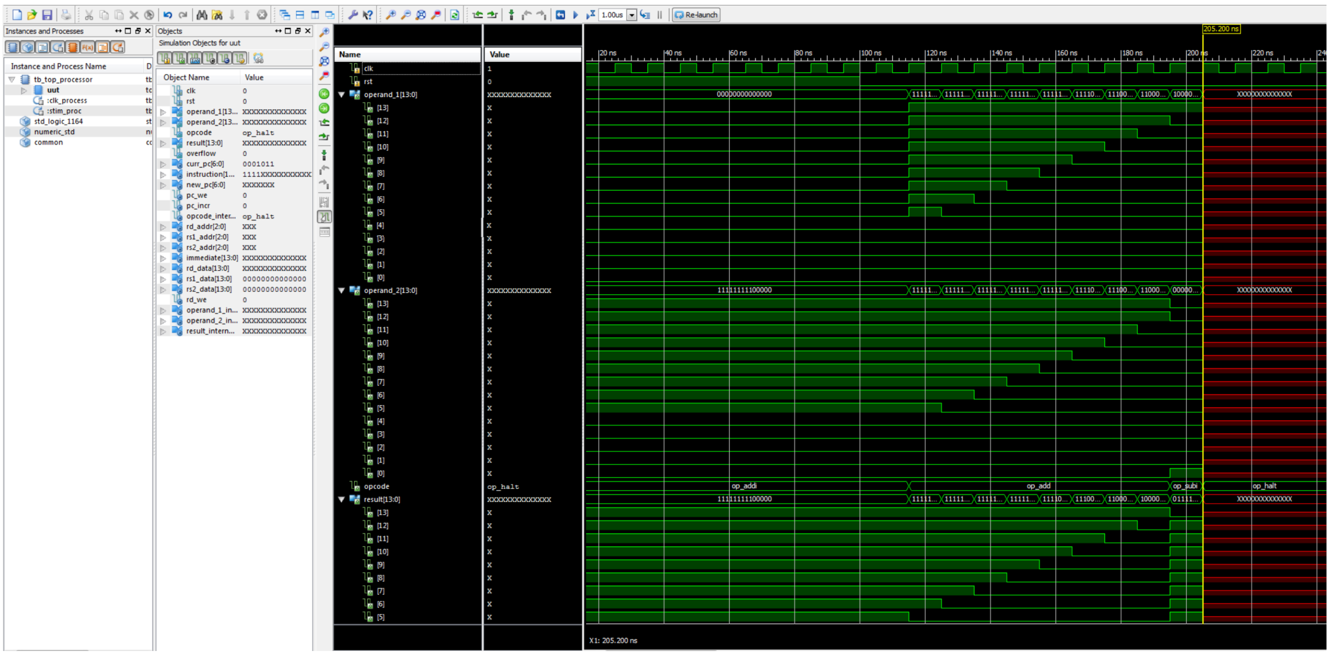


Figure 2: Floorplan

2 Task 2

2.1 Behavioral simulation snapshot


Instances and Processes

Instance and Process Name

- tb_top_processor
  - uut
  - :clk_process
  - :stim_proc
- std_logic_1164
- numeric_std
- common

Objects

Simulation Objects for uut

Object Name	Value
clk	0
rst	0
operand_1[13...	XXXXXXXXXXXXXX
operand_2[13...	XXXXXXXXXXXXXX
opcode	op_halt
result[13:0]	XXXXXXXXXXXXXX
overflow	0
curr_pc[6:0]	0001011
instruction[1...	1111XXXXXXXXXX
new_pc[6:0]	XXXXXXX
pc_we	0
pc_incr	0
opcode_inter...	op_halt
rd_addr[2:0]	XXX
rs1_addr[2:0]	XXX
rs2_addr[2:0]	XXX
immediate[13:0]	XXXXXXXXXXXXXX
rd_data[13:0]	XXXXXXXXXXXXXX
rs1_data[13:0]	00000000000000
rs2_data[13:0]	00000000000000
rd_we	0
operand_1_in...	XXXXXXXXXXXXXX
operand_2_in...	XXXXXXXXXXXXXX
result_intern...	XXXXXXXXXXXXXX

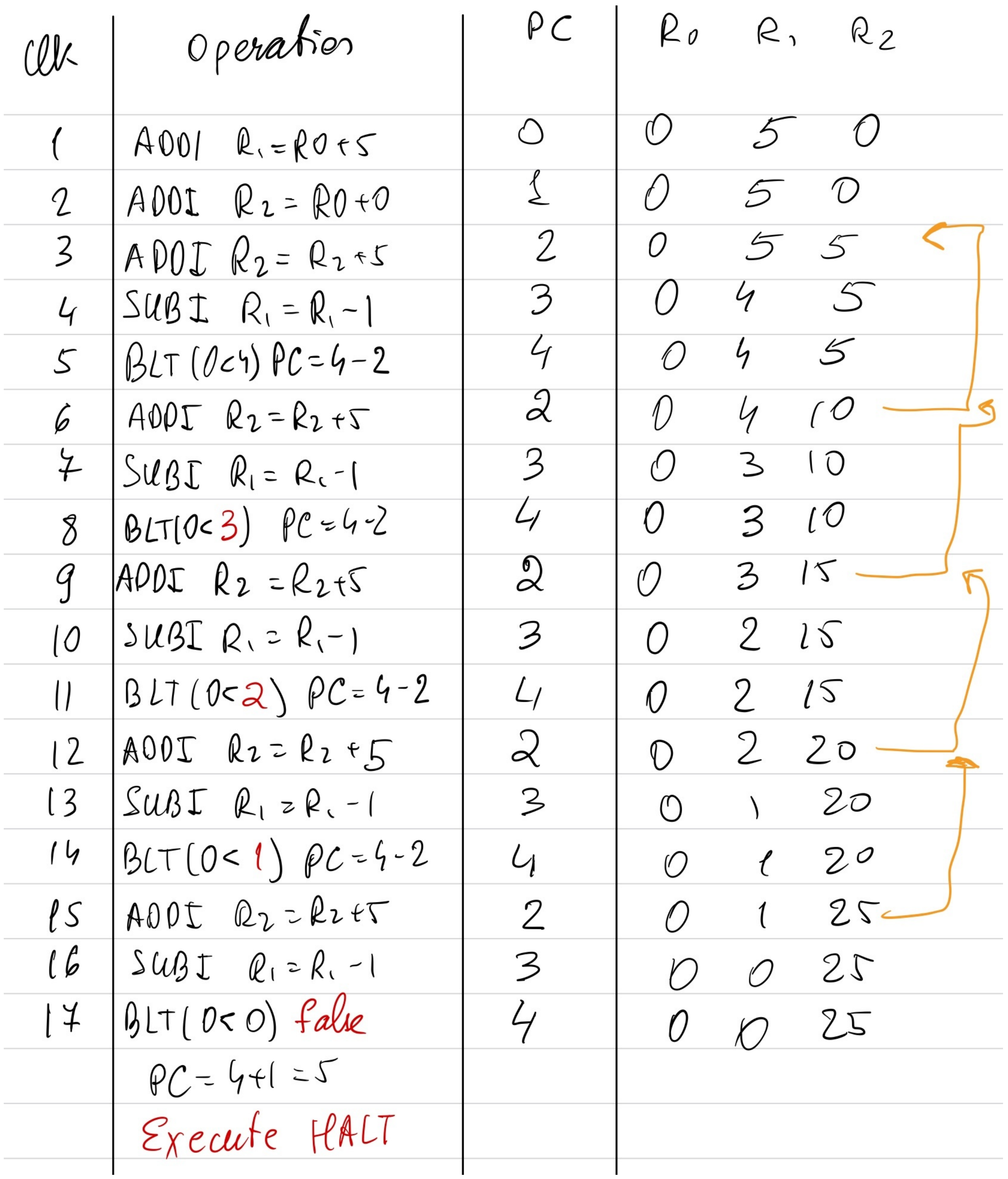
Figure 3: Behavioral simulation snapshot

2.2 Video

Click to watch video demonstration

3 Task 3

3.1 Program behavior, intermediate results, final results



Clk	Operation	PC	R_0	R_1	R_2
1	ADDI $R_1 = R0 + 5$	0	0	5	0
2	ADDI $R_2 = R0 + 0$	1	0	5	0
3	ADDI $R_2 = R_2 + 5$	2	0	5	5
4	SUBI $R_1 = R_1 - 1$	3	0	4	5
5	BLT $(0<4)$ $PC = 4-2$	4	0	4	5
6	ADDI $R_2 = R_2 + 5$	2	0	4	10
7	SUBI $R_1 = R_1 - 1$	3	0	3	10
8	BLT$(0<3)$ $PC = 4-2$	4	0	3	10
9	ADDI $R_2 = R_2 + 5$	2	0	3	15
10	SUBI $R_1 = R_1 - 1$	3	0	2	15
11	BLT $(0<2)$ $PC = 4-2$	4	0	2	15
12	ADDI $R_2 = R_2 + 5$	2	0	2	20
13	SUBI $R_1 = R_1 - 1$	3	0	1	20
14	BLT$(0<1)$ $PC = 4-2$	4	0	1	20
15	ADDI $R_2 = R_2 + 5$	2	0	1	25
16	SUBI $R_1 = R_1 - 1$	3	0	0	25
17	BLT$(0<0)$ false	4	0	0	25
	$PC = 4+1 = 5$				
	Execute HALT				

Figure 4: Handwritten description of program behavior with each clock cycle

3.2 Video

Click to watch video demonstration

4 Task 4

4.1 Program idea and description

We implemented a triangular number generator. Triangular numbers are numbers that can be represented by a pattern of dots arranged in an equilateral triangle with the same number of dots on each side.

$$T_n = \sum_{k=1}^{n} k$$

For an input value n, it is the summation of integer numbers up to n, thus implemented by an iterative addition process.

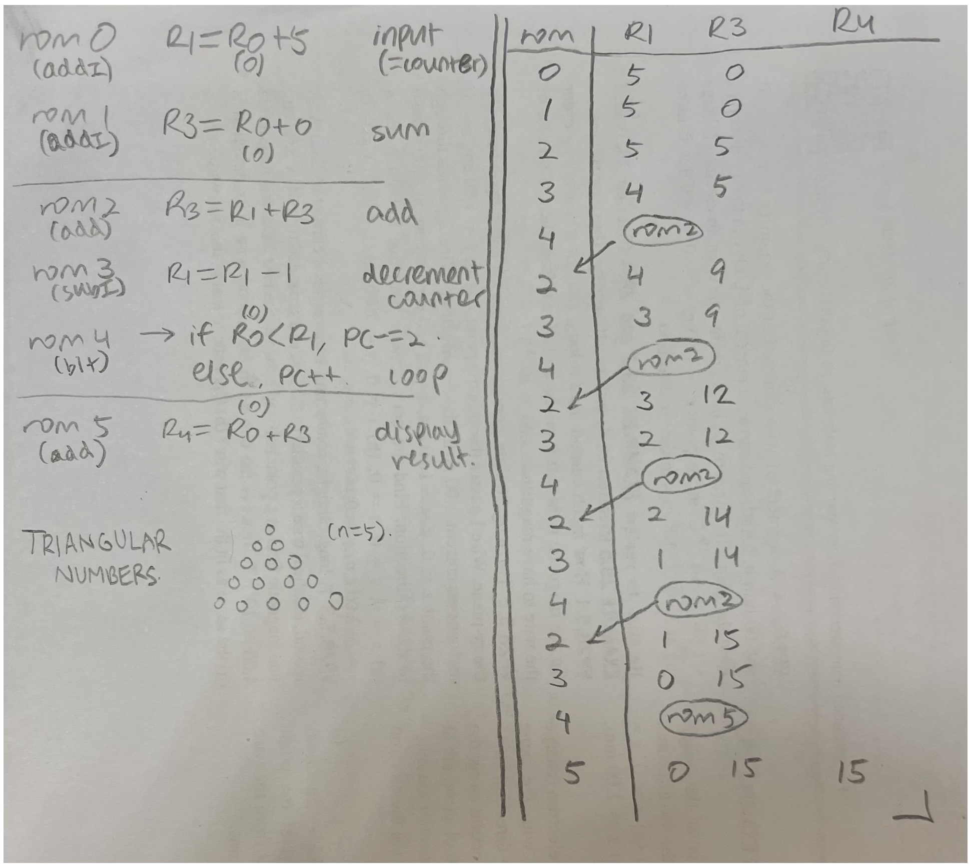
1. Initialization of inputs: R1 is the input number, also used as a decrementing counter that traverses the summing process. R3 stores the sum, initialized as 0 (using R0).
2. Actual computation: The algorithm adds the counter value to the current sum, updating the sum and in turn decrementing the counter. Once the counter reaches 0, it exits the summation loop.
3. R4 is used as a storage register of the final result upon exit of the loop.

4.1.1 Task 4 Implementation

```
-- >>> TASK 4 program: Triangular number generator
    -- ==Initialize==
    -- 0 : ADDI  R1, R0, +5 : 1001 001 000 000 101
    rom(0) <= opcode_type_to_std_logic_vector(OP_ADDI) & b"001" &
        b"000" & b"000_101";
    -- 1 : ADDI  R3, R0, 0 : 1001 011 000 000 000
    rom(1) <= opcode_type_to_std_logic_vector(OP_ADDI) & b"011" &
        b"000" & b"000_000";
    -- ==Iterative adding==
    -- 2 : ADD  R3, R1, R3 : 1000 011 001 011 000
    rom(2) <= opcode_type_to_std_logic_vector(OP_ADD) & b"011" &
       b"001" & b"011" & b"000";
    -- 3 : SUBI  R1, R1, 1 : 1011 001 001 000 001
    -- check if R1 has reached the end of iteration
    rom(3) <= opcode_type_to_std_logic_vector(OP_SUBI) & b"001" &
        b"001" & b"000_001";
    -- 4 : BLT   R1, R0, -2 : 1100 111 000 001 110
    -- (loop back to address 2 while R1<0)
    rom(4) <= opcode_type_to_std_logic_vector(OP_BLT)  & b"111" &
        b"000" & b"001_110";
    -- 5 : ADD  R4, R0, R3 : 1000 100 000 011 000
    -- store final result into a new register
    rom(5) <= opcode_type_to_std_logic_vector(OP_ADD) & b"100" &
       b"000" & b"011_000";

-- <<< TASK 4 program ends
```

Listing 5: VHDL code for FPGA top-level module

4.2 Program behavior, intermediate results, final results



rom 0 (addI)	R1=R0+5 (0)	input (=counter)
rom 1 (addI)	R3=R0+0 (0)	sum
rom 2 (add)	R3=R1+R3	add
rom 3 (subI)	R1=R1−1	decrement counter
rom 4 (blt)	→ if R0<R1 (0), PC−=2. else, PC++	loop
rom 5 (add)	R4=R0+R3 (0)	display result.

TRIANGULAR NUMBERS. (n=5).

rom	R1	R3	R4
0	5	0	
1	5	0	
2	5	5	
3	4	5	
4	rom2		
2	4	9	
3	3	9	
4	rom2		
2	3	12	
3	2	12	
4	rom2		
2	2	14	
3	1	14	
4	rom2		
2	1	15	
3	0	15	
4	rom5		
5	0	15	15

Figure 5: Program behavior of Task 4: Triangular number generator

5 Video

Click to watch video demonstration

6 Appendix

6.1 Self-drawn circuit diagram

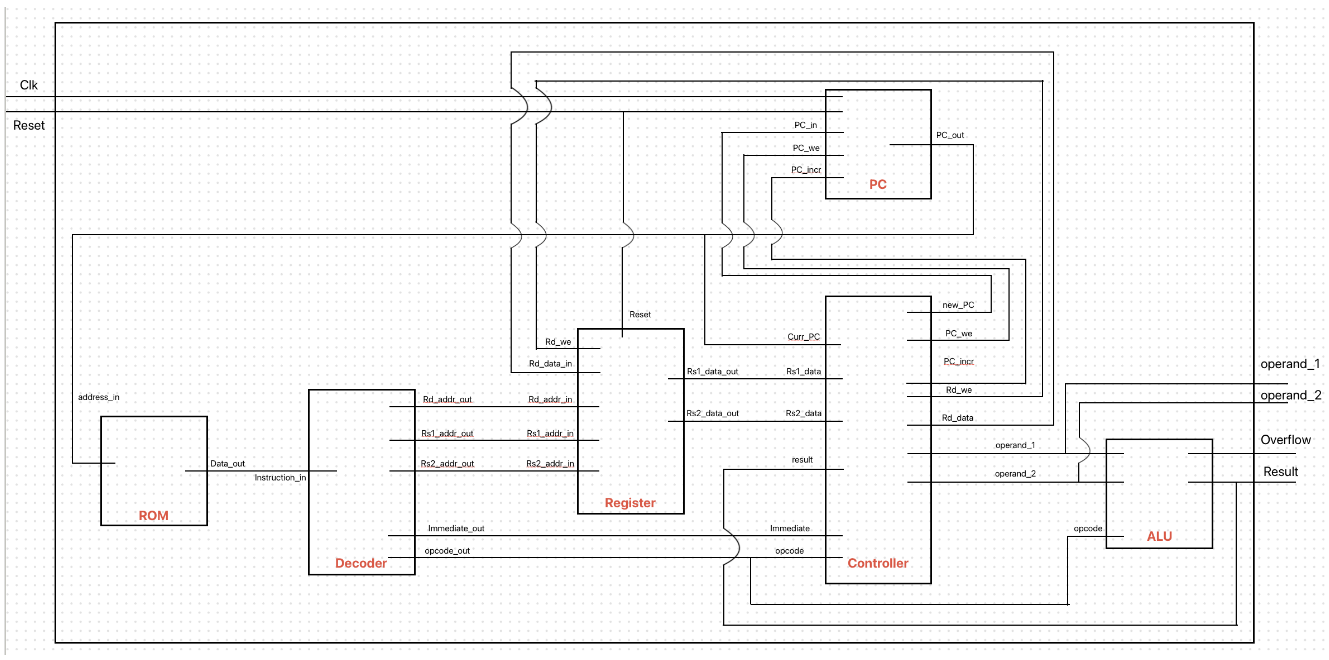


Figure 6: A self-drawn circuit diagram of the microprocessor