

جامعة نيويورك أبوظبي



NYU ABU DHABI

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 . . . . .	2
1.1	VHDL code for decoder . . . . .	2
1.2	VHDL code for controller . . . . .	4
1.3	VHDL code for ALU unit . . . . .	6
1.4	Report on clock cycles per instruction . . . . .	9
1.5	VHDL code FPGA top-level module . . . . .	9
1.6	Floorplan PDF/screenshot . . . . .	14
2	Task 2 . . . . .	15
2.1	Behavioral simulation snapshot . . . . .	15
2.2	Video . . . . .	15
3	Task 3 . . . . .	16
3.1	Program behavior, intermediate results, final results . . . . .	16
3.2	Video . . . . .	16
4	Task 4 . . . . .	17
4.1	Program idea and description . . . . .	17
4.2	Program behavior, intermediate results, final results . . . . .	18
5	Video . . . . .	18
6	Appendix . . . . .	19
6.1	Self-drawn circuit diagram . . . . .	19

# 1 Task 1

## 1.1 VHDL code for decoder

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5
6 use work.common.all;
7
8
9 entity Decoder is
10     port ( instruction_in : in STD_LOGIC_VECTOR (15 downto 0);
11
12
13         opcode_out : out opcode_type;
14
15
16         Rd_addr_out : out STD_LOGIC_VECTOR (2 downto 0);
17         Rs1_addr_out : out STD_LOGIC_VECTOR (2 downto 0);
18         Rs2_addr_out : out STD_LOGIC_VECTOR (2 downto 0);
19
20
21         immediate_out : out STD_LOGIC_VECTOR (13 downto 0)
22     );
23 end Decoder;
24
25
26 architecture Behavioral of Decoder is
27
28
29 --TODO add signals as needed
30 signal opcode_internal : opcode_type;
31 signal Rd_addr_internal : STD_LOGIC_VECTOR (2 downto 0);
32 signal Rs1_addr_internal : STD_LOGIC_VECTOR (2 downto 0);
33 signal Rs2_addr_internal : STD_LOGIC_VECTOR (2 downto 0);
34 signal tail_internal : STD_LOGIC_VECTOR (2 downto 0);
35
36
37
38
39 begin
40     process (instruction_in, opcode_internal, Rd_addr_internal,
41             Rs1_addr_internal, Rs2_addr_internal, tail_internal)
42     begin
43         --TODO implement extraction of remaining parts of the
44             instruction
45
46
47         --Extract opcode
```

```

46         opcode_internal <= std_logic_vector_to_opcode_type(
47             instruction_in(15 downto 12) );
48         Rd_addr_internal <= instruction_in(11 downto 9);
49         Rs1_addr_internal <= instruction_in(8 downto 6);
50         Rs2_addr_internal <= instruction_in(5 downto 3);
51         tail_internal <= instruction_in(2 downto 0);
52
53
54     --Assign outputs
55     opcode_out <= opcode_internal;
56     Rd_addr_out <= Rd_addr_internal;
57     Rs1_addr_out <= Rs1_addr_internal;
58     Rs2_addr_out <= Rs2_addr_internal;
59
60
61
62
63     --TODO derive immediate value, depending on
        opcode_internal
64     case opcode_internal is
65         when OP_ANDI =>
66             immediate_out <= "11111111" &
67                 Rs2_addr_internal & tail_internal;
68
69         when OP_ORI | OP_XORI =>
70             immediate_out <= "00000000" &
71                 Rs2_addr_internal & tail_internal;
72
73         when OP_ADDI | OP_SUBI =>
74             if Rs2_addr_internal(2) = '0' then
75                 immediate_out <= "00000000" &
76                     Rs2_addr_internal & tail_internal;
77             else
78                 immediate_out <= "11111111" &
79                     Rs2_addr_internal & tail_internal;
80             end if;
81
82         when OP_BLT | OP_BE | OP_JMP=>
83             if Rd_addr_internal(2) = '0' then
84                 immediate_out <= "00000000" &
85                     Rd_addr_internal & tail_internal;
86             else
87                 immediate_out <= "11111111" &
88                     Rd_addr_internal & tail_internal;
89             end if;
90
91         when OP_SLL | OP_SRL =>
92             immediate_out <= "0000000000" & tail_internal
93             ;
94
95
96
97
98
99

```

```

88         when others => immediate_out <= "XXXXXXXXXXXXXXXX";
89
90     end case;
91 end process;
92
93 end Behavioral;

```

Listing 1: VHDL code for decoder

## 1.2 VHDL code for controller

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  use work.common.all;
6
7  entity Controller is
8      port ( opcode : in opcode_type;
9
10         operand_1 : out STD_LOGIC_VECTOR (13 downto 0);
11         operand_2 : out STD_LOGIC_VECTOR (13 downto 0);
12
13         result : in STD_LOGIC_VECTOR (13 downto 0);
14
15         curr_PC : in STD_LOGIC_VECTOR (6 downto 0);
16
17         new_PC : out STD_LOGIC_VECTOR (6 downto 0);
18         PC_we : out STD_LOGIC;
19         PC_incr : out STD_LOGIC;
20
21         Rs1_data : in STD_LOGIC_VECTOR (13 downto 0);
22         Rs2_data : in STD_LOGIC_VECTOR (13 downto 0);
23         immediate : in STD_LOGIC_VECTOR (13 downto 0);
24
25         Rd_we : out STD_LOGIC;
26         Rd_data : out STD_LOGIC_VECTOR (13 downto 0)
27     );
28 end Controller;
29
30 architecture Behavioral of Controller is
31
32 begin
33
34 control : process (opcode, Rs1_data, Rs2_data, result, immediate,
35                 curr_PC)
36 begin
37     -- default assignments, can be overwritten below
38     operand_1 <= Rs1_data;
39     operand_2 <= Rs2_data;

```

```

40 Rd_we <= '0';
41 Rd_data <= result;
42
43 PC_we <= '0';
44 new_PC <= (6 downto 0 => 'X');
45
46 PC_incr <= '0';
47
48 -- regular operations with Rs1, Rs2, Rd
49 -- TODO consider remaining cases
50 if ((opcode = OP_AND) or (opcode = OP_OR) or (opcode = OP_XOR
    ) or (opcode = OP_ADD) or (opcode = OP_SUB)) then
51
52     operand_1 <= Rs1_data;
53     operand_2 <= Rs2_data;
54
55     Rd_we <= '1';
56     PC_incr <= '1';
57
58 -- TODO implement remaining cases
59 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
60
61     operand_1 <= Rs1_data;
62     operand_2 <= immediate;
63
64     Rd_we <= '1';
65     PC_incr <= '1';
66
67 elsif ((opcode = OP_BLT)) then
68     if (signed(Rs1_data) < signed(Rs2_data)) then
69         operand_1 <= "0000000" & Curr_PC;
70         operand_2 <= immediate;
71         PC_we <= '1';
72         new_PC <= result (6 downto 0);
73     else
74         PC_incr <= '1';
75     end if;
76
77 elsif ((opcode = OP_BE)) then
78     if (signed(Rs1_data) = signed(Rs2_data)) then
79         operand_1 <= "0000000" & Curr_PC;
80         operand_2 <= immediate;
81         PC_we <= '1';
82         new_PC <= result (6 downto 0);
83     else
84         PC_incr <= '1';
85     end if;
86
87 elsif ((opcode = OP_JMP)) then

```

```

88     operand_1 <= "0000000" & Curr_PC;
89     operand_2 <= immediate;
90     PC_we <= '1';
91     new_PC <= result (6 downto 0);
92
93
94     -- only OP_HALT should remain
95     else
96         operand_1 <= (13 downto 0 => 'X');
97         operand_2 <= (13 downto 0 => 'X');
98
99         Rd_data <= (13 downto 0 => 'X');
100     end if;
101 end process;
102
103 end Behavioral;

```

Listing 2: VHDL code for Controller

### 1.3 VHDL code for ALU unit

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5
6  use work.common.all;
7
8
9  entity ALU is
10     port ( operand_1 : in STD_LOGIC_VECTOR (13 downto 0);
11           operand_2 : in STD_LOGIC_VECTOR (13 downto 0);
12
13
14           opcode : in opcode_type;
15
16
17           result : out STD_LOGIC_VECTOR (13 downto 0);
18           overflow : out STD_LOGIC
19
20
21           );
22 end ALU;
23
24
25 architecture Behavioral of ALU is
26
27
28     signal result_internal: STD_LOGIC_VECTOR (13 downto 0);
29
30

```

```

31 begin
32
33
34 result <= result_internal;
35
36
37 calculate : process (operand_1, operand_2, opcode)
38 begin
39     -- signed ADD operations
40     -- TODO consider remaining cases
41     if ((opcode = OP_ADD) or (opcode = OP_ADDI) ) then
42         result_internal <= std_logic_vector( signed(operand_1) +
43             signed(operand_2) );
44
45     -- SLL operation
46     elsif (opcode = OP_SLL) then
47         result_internal <= std_logic_vector( shift_left(unsigned(
48             operand_1), to_integer(unsigned(operand_2))) );
49
50     -- TODO implement remaining operations
51     elsif ((opcode = OP_AND) or (opcode = OP_ANDI) ) then
52         result_internal <= std_logic_vector(operand_1 and
53             operand_2);
54
55     elsif ((opcode = OP_OR) or (opcode = OP_ORI) ) then
56         result_internal <= std_logic_vector(operand_1 or
57             operand_2);
58
59     elsif ((opcode = OP_XOR) or (opcode = OP_XORI) ) then
60         result_internal <= std_logic_vector(operand_1 xor
61             operand_2);
62
63     elsif (opcode = OP_SRL) then
64         result_internal <= std_logic_vector( shift_right(unsigned
65             (operand_1), to_integer(unsigned(operand_2))) );
66
67     elsif (opcode = OP_SLL) then
68         result_internal <= std_logic_vector( shift_left(unsigned(
69             operand_1), to_integer(unsigned(operand_2))) );
70
71     elsif ((opcode = OP_SUB) or (opcode = OP_SUBI) ) then
72         result_internal <= std_logic_vector( signed(operand_1) -
73             signed(operand_2) );
74
75     elsif ((opcode = OP_BLT) or (opcode = OP_BE) or (opcode =
76         OP_JMP)) then
77         result_internal <= std_logic_vector( signed(operand_1) +
78             signed(operand_2) );
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99

```



```

72     -- only OP_HALT should remain
73     else
74         result_internal <= (13 downto 0 => 'X');
75
76
77     end if;
78
79
80 end process;
81
82
83 -- TODO implement detection of overflow for all signed arithmetic
84   operations
85 ofl : process (operand_1, operand_2, result_internal, opcode)
86 begin
87     if opcode = OP_ADD or opcode = OP_ADDI then
88         if (signed(operand_1) > 0 and signed(operand_2) > 0 and
89             signed(result_internal) < 0) or (signed(operand_1) < 0
90             and signed(operand_2) < 0 and signed(result_internal
91             ) > 0) then
92             overflow <= '1';
93         else
94             overflow <= '0';
95         end if;
96
97     elsif opcode = OP_SUB or opcode = OP_SUBI then
98         if (signed(operand_1) < 0 and signed(operand_2) > 0 and
99             signed(result_internal) > 0) or (signed(operand_1) > 0
100             and signed(operand_2) < 0 and signed(result_internal
101             ) < 0) then
102             overflow <= '1';
103         else
104             overflow <= '0';
105         end if;
106     else
107         overflow <= '0';
108     end if;
109 end process;
110
111 end Behavioral;

```

Listing 3: VHDL code for ALU unit

## 1.4 Report on clock cycles per instruction

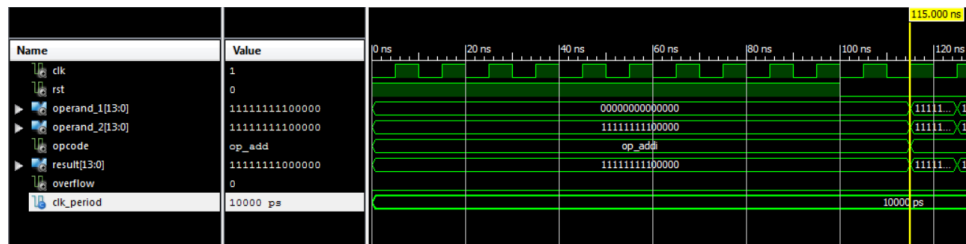


Figure 1: Clock cycles per instruction

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

## 1.5 VHDL code FPGA top-level module

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 use work.common.all;
5
6 entity top_processor_FPGA is
7     port ( next_instr : in STD_LOGIC;
8           clk : in STD_LOGIC;
9           rst : in STD_LOGIC;
10
11           operand_1 : out STD_LOGIC_VECTOR (13 downto 0);
12           operand_2 : out STD_LOGIC_VECTOR (13 downto 0);
13
14           opcode : out opcode_type;
15
16           result : out STD_LOGIC_VECTOR (13 downto 0);
17
18           overflow : out STD_LOGIC;
19
20           seg_bits : out STD_LOGIC_VECTOR(0 to 7);
21           seg_an : out STD_LOGIC_VECTOR(3 downto 0)
22     );
23 end top_processor_FPGA;
24
25 architecture Behavioral of top_processor_FPGA is
26
27 -- component declarations
28     component Display_Controller
29         port (
30             clk : in STD_LOGIC;
31             rst : in STD_LOGIC;
32             opcode : in opcode_type;
33             operand_1 : in STD_LOGIC_VECTOR (13 downto 0);
34             operand_2 : in STD_LOGIC_VECTOR (13 downto 0);

```

```

35         result      : in  STD_LOGIC_VECTOR (13 downto 0);
36         overflow     : in  STD_LOGIC;
37         seg_bits     : out STD_LOGIC_VECTOR (0 to 7);
38         seg_an       : out STD_LOGIC_VECTOR (3 downto 0)
39     );
40 end component;
41
42 component Instructions_ROM
43     port ( address_in : in  STD_LOGIC_VECTOR (6 downto 0);
44           data_out  : out STD_LOGIC_VECTOR (15 downto 0)
45           );
46 end component;
47
48 component PC
49     port ( clk : in  STD_LOGIC;
50           rst: in  STD_LOGIC;
51
52           PC_in : in  STD_LOGIC_VECTOR (6 downto 0);
53           PC_out : out STD_LOGIC_VECTOR (6 downto 0);
54
55           PC_we : in  STD_LOGIC;
56           PC_incr : in  STD_LOGIC
57           );
58 end component;
59
60 component Registers
61     port ( clk : in  STD_LOGIC;
62           rst: in  STD_LOGIC;
63
64           Rs1_addr_in : in  STD_LOGIC_VECTOR (2 downto 0);
65           Rs1_data_out : out STD_LOGIC_VECTOR (13 downto 0);
66
67           Rs2_addr_in : in  STD_LOGIC_VECTOR (2 downto 0);
68           Rs2_data_out : out STD_LOGIC_VECTOR (13 downto 0);
69
70           Rd_addr_in : in  STD_LOGIC_VECTOR (2 downto 0);
71           Rd_data_in  : in  STD_LOGIC_VECTOR (13 downto 0);
72           Rd_we       : in  STD_LOGIC
73           );
74 end component;
75
76 component Decoder
77     port ( instruction_in : in  STD_LOGIC_VECTOR (15 downto 0);
78
79           opcode_out : out opcode_type;
80
81           Rd_addr_out : out STD_LOGIC_VECTOR (2 downto 0);
82           Rs1_addr_out : out STD_LOGIC_VECTOR (2 downto 0);
83           Rs2_addr_out : out STD_LOGIC_VECTOR (2 downto 0);
84
85           immediate_out : out STD_LOGIC_VECTOR (13 downto 0)

```

```

86         );
87 end component;
88
89 component Controller
90     port ( opcode : in opcode_type;
91
92         operand_1 : out STD_LOGIC_VECTOR (13 downto 0);
93         operand_2 : out STD_LOGIC_VECTOR (13 downto 0);
94
95         result : in STD_LOGIC_VECTOR (13 downto 0);
96
97         curr_PC : in STD_LOGIC_VECTOR (6 downto 0);
98
99         new_PC : out STD_LOGIC_VECTOR (6 downto 0);
100        PC_we : out STD_LOGIC;
101        PC_incr : out STD_LOGIC;
102
103        Rs1_data : in STD_LOGIC_VECTOR (13 downto 0);
104        Rs2_data : in STD_LOGIC_VECTOR (13 downto 0);
105        immediate : in STD_LOGIC_VECTOR (13 downto 0);
106
107        Rd_we : out STD_LOGIC;
108        Rd_data : out STD_LOGIC_VECTOR (13 downto 0)
109    );
110 end component;
111
112 component ALU
113     port ( operand_1 : in STD_LOGIC_VECTOR (13 downto 0);
114         operand_2 : in STD_LOGIC_VECTOR (13 downto 0);
115
116         opcode : in opcode_type;
117
118         result : out STD_LOGIC_VECTOR (13 downto 0);
119         overflow : out STD_LOGIC
120     );
121 end component;
122
123 --
124 -- internal signals
125 --
126
127 -- instructions
128 signal curr_PC : STD_LOGIC_VECTOR (6 downto 0);
129 signal instruction : STD_LOGIC_VECTOR (15 downto 0);
130
131 signal new_PC : STD_LOGIC_VECTOR (6 downto 0);
132 signal PC_we : STD_LOGIC;
133 signal PC_incr : STD_LOGIC;
134
135 -- decoder and controller
136 signal opcode_internal : opcode_type;

```

```

137
138 signal Rd_addr : STD_LOGIC_VECTOR (2 downto 0);
139 signal Rs1_addr : STD_LOGIC_VECTOR (2 downto 0);
140 signal Rs2_addr : STD_LOGIC_VECTOR (2 downto 0);
141 signal immediate : STD_LOGIC_VECTOR (13 downto 0);
142
143 -- registers
144 signal Rd_data : STD_LOGIC_VECTOR (13 downto 0);
145 signal Rs1_data : STD_LOGIC_VECTOR (13 downto 0);
146 signal Rs2_data : STD_LOGIC_VECTOR (13 downto 0);
147 signal Rd_we : STD_LOGIC;
148
149 -- ALU
150 signal operand_1_internal : STD_LOGIC_VECTOR (13 downto 0);
151 signal operand_2_internal : STD_LOGIC_VECTOR (13 downto 0);
152 signal result_internal : STD_LOGIC_VECTOR (13 downto 0);
153
154 --DISP
155 signal overflow_internal : STD_LOGIC;
156
157 begin
158
159 overflow <= overflow_internal;
160
161 -- simple wiring of global signals
162 operand_1 <= operand_1_internal;
163 operand_2 <= operand_2_internal;
164 result <= result_internal;
165 opcode <= opcode_internal;
166
167 --
168 -- component instances with port maps; nothing else is allowed
169 -- for this top-level module
170
171 Instructions_ROM_inst : Instructions_ROM
172     port map (curr_PC, instruction);
173
174 Decoder_inst : Decoder
175     port map (instruction, opcode_internal, Rd_addr, Rs1_addr,
176             Rs2_addr, immediate);
177
178 Controller_inst : Controller
179     port map (opcode_internal, operand_1_internal,
180             operand_2_internal, result_internal, curr_PC, new_PC,
181             PC_we, PC_incr, Rs1_data, Rs2_data, immediate, Rd_we,
182             Rd_data);
183
184 PC_inst : PC
185     port map (next_instr, rst, new_PC, curr_PC, PC_we, PC_incr);
186

```

```

183 Registers_inst : Registers
184     port map (next_instr, rst, Rs1_addr, Rs1_data, Rs2_addr,
185               Rs2_data, Rd_addr, Rd_data, Rd_we);
186
186 ALU_inst : ALU
187     port map (operand_1_internal, operand_2_internal,
188               opcode_internal, result_internal, overflow_internal);
189
190 display_i : Display_Controller
191     port map (
192         clk      => clk,
193         rst      => rst,
194         opcode   => opcode_internal,
195         operand_1 => operand_1_internal,
196         operand_2 => operand_2_internal,
197         result   => result_internal,
198         overflow  => overflow_internal,
199         seg_bits => seg_bits,
200         seg_an   => seg_an
201     );
202
203 end Behavioral;

```

Listing 4: VHDL code for FPGA top-level module

## 1.6 Floorplan PDF/screenshot

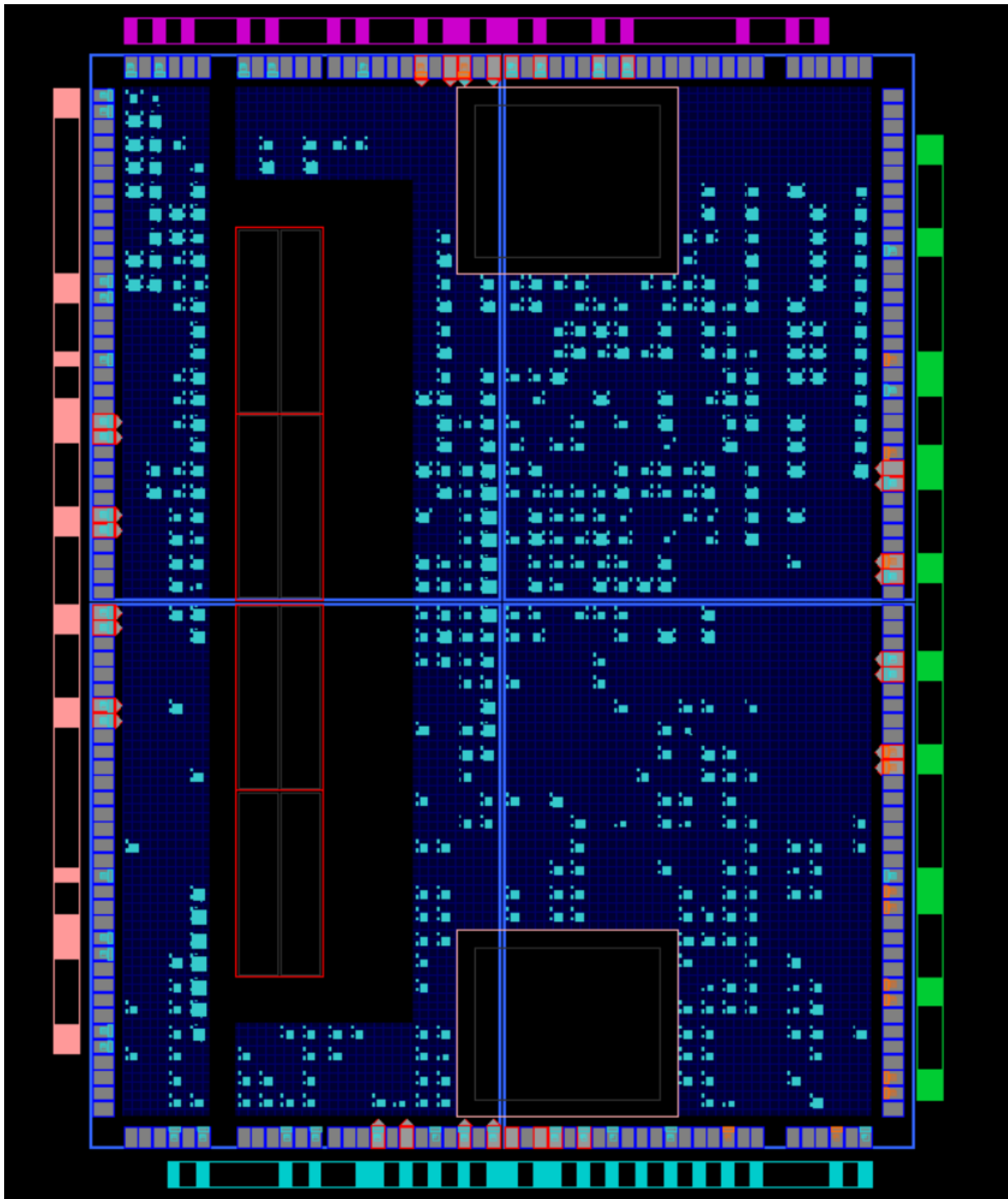


Figure 2: Floorplan

## 2 Task 2

## 2.1 Behavioral simulation snapshot



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 <sub>0</sub>	R <sub>1</sub>	R <sub>2</sub>
1	ADDI R <sub>1</sub> = R <sub>0</sub> + 5	0	0	5	0
2	ADDI R <sub>2</sub> = R <sub>0</sub> + 0	1	0	5	0
3	ADDI R <sub>2</sub> = R <sub>2</sub> + 5	2	0	5	5
4	SUBI R <sub>1</sub> = R <sub>1</sub> - 1	3	0	4	5
5	BLT (0 < 4) PC = 4 - 2	4	0	4	5
6	ADDI R <sub>2</sub> = R <sub>2</sub> + 5	2	0	4	10
7	SUBI R <sub>1</sub> = R <sub>1</sub> - 1	3	0	3	10
8	BLT (0 < 3) PC = 4 - 2	4	0	3	10
9	ADDI R <sub>2</sub> = R <sub>2</sub> + 5	2	0	3	15
10	SUBI R <sub>1</sub> = R <sub>1</sub> - 1	3	0	2	15
11	BLT (0 < 2) PC = 4 - 2	4	0	2	15
12	ADDI R <sub>2</sub> = R <sub>2</sub> + 5	2	0	2	20
13	SUBI R <sub>1</sub> = R <sub>1</sub> - 1	3	0	1	20
14	BLT (0 < 1) PC = 4 - 2	4	0	1	20
15	ADDI R <sub>2</sub> = R <sub>2</sub> + 5	2	0	1	25
16	SUBI R <sub>1</sub> = R <sub>1</sub> - 1	3	0	0	25
17	BLT (0 < 0) <i>false</i>	4	0	0	25
	PC = 4 + 1 = 5				
	<i>Execute HALT</i>				

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

```
1 -- >>> TASK 4 program: Triangular number generator
2 -- ==Initialize==
3 -- 0 : ADDI R1, R0, +5 : 1001 001 000 000 101
4 rom(0) <= opcode_type_to_std_logic_vector(OP_ADDI) & b"001" &
   b"000" & b"000_101";
5 -- 1 : ADDI R3, R0, 0 : 1001 011 000 000 000
6 rom(1) <= opcode_type_to_std_logic_vector(OP_ADDI) & b"011" &
   b"000" & b"000_000";
7 -- ==Iterative adding==
8 -- 2 : ADD R3, R1, R3 : 1000 011 001 011 000
9 rom(2) <= opcode_type_to_std_logic_vector(OP_ADD) & b"011" &
   b"001" & b"011" & b"000";
10 -- 3 : SUBI R1, R1, 1 : 1011 001 001 000 001
11 -- check if R1 has reached the end of iteration
12 rom(3) <= opcode_type_to_std_logic_vector(OP_SUBI) & b"001" &
   b"001" & b"000_001";
13 -- 4 : BLT R1, R0, -2 : 1100 111 000 001 110
14 -- (loop back to address 2 while R1<0)
15 rom(4) <= opcode_type_to_std_logic_vector(OP_BLT) & b"111" &
   b"000" & b"001_110";
16 -- 5 : ADD R4, R0, R3 : 1000 100 000 011 000
17 -- store final result into a new register
18 rom(5) <= opcode_type_to_std_logic_vector(OP_ADD) & b"100" &
   b"000" & b"011_000";
19
20 -- <<< TASK 4 program ends
```

Listing 5: VHDL code for FPGA top-level module

## 4.2 Program behavior, intermediate results, final results

rom 0 (addi)	$R_1 = R_0 + 5$ (0)	input (=counter)	rom	R1	R3	R4
rom 1 (addi)	$R_3 = R_0 + 0$ (0)	sum	0	5	0	
rom 2 (add)	$R_3 = R_1 + R_3$	add	1	5	0	
rom 3 (subi)	$R_1 = R_1 - 1$	decrement counter	2	5	5	
rom 4 (bit)	$\rightarrow$ if $R_0 < R_1$ , $PC--2$ else, $PC++$ loop		3	4	5	
rom 5 (add)	$R_4 = R_0 + R_3$	display result.	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

TRIANGULAR NUMBERS. (n=5).

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

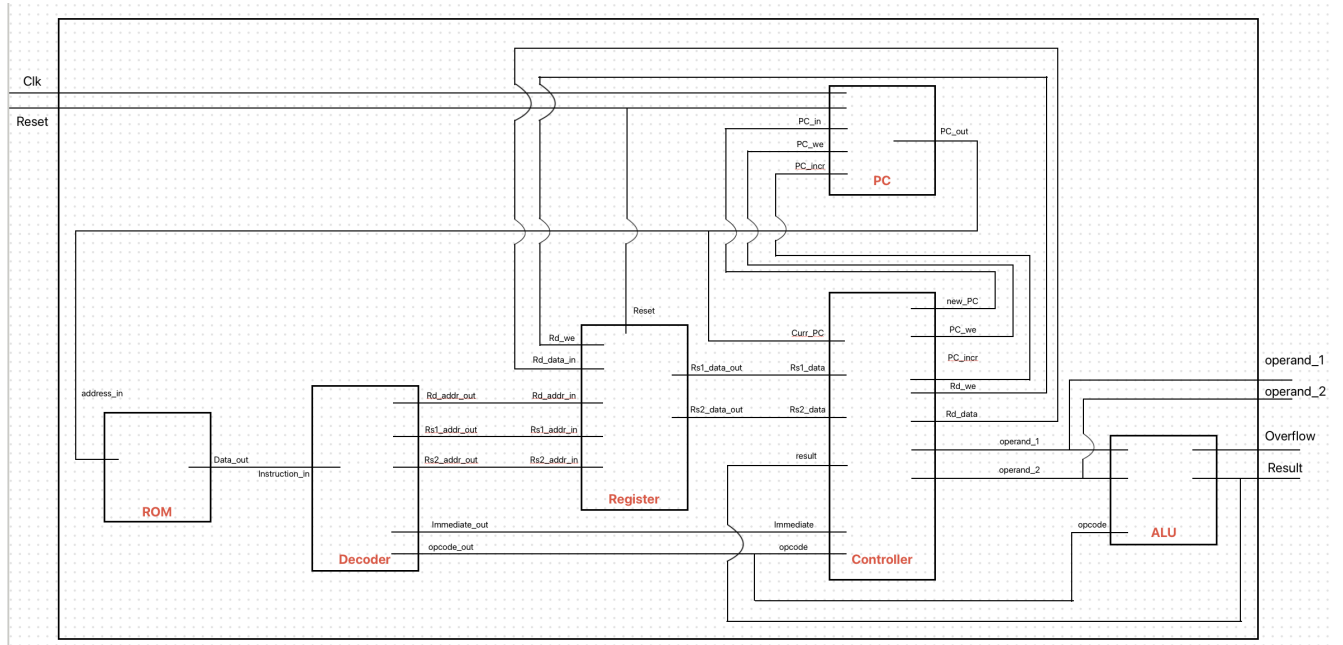


Figure 6: A self-drawn circuit diagram of the microprocessor