Digital System Design with Hardware Description Language

HW2

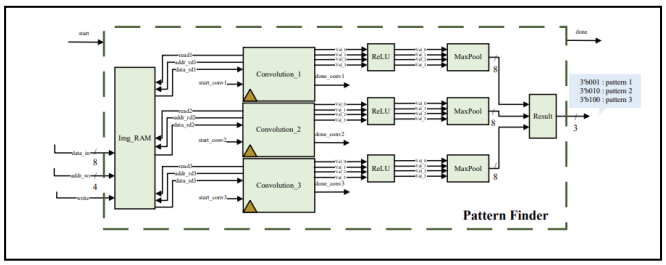
MohammadJavad Rabiei Kashanaki
electrical and computer engineering department
university of Tehran
SID : 810102145

***Abstract* — In this project we are going to implement a pattern finder circuit which is constructed from three convolutional block. Each convolutional block searches for a certain pattern which is given to it as a kernel. Other moduels are added to this to find a pattern in a given image.**

Keywords* — *CNN, convolution, pattern finder, kernel

PATTERN FINDER

As it is shown in the home work, the pattern finder circuit is seen in figure below.

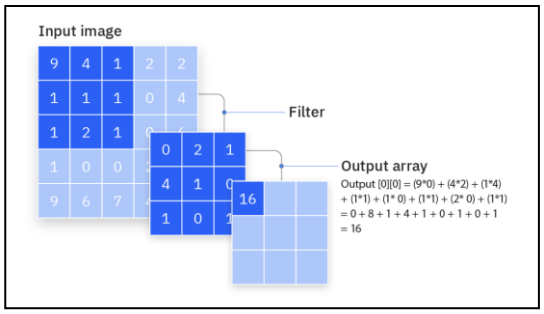


As it can be seen, the circuit is generated from different parts. We are going to see the functionality of each part and VHDL implementation of each one will be provided in the next part.

I. CONVOLUTION

A convolutional block is a building block used in a convolutional neural network (CNN) for image recognition. It is made up of one or more convolutional layers, that are used to extract features from the input image.

The operation of the convolution section can be observed in the diagram below.



Additionally, since this section only includes one multiplier and one adder, the circuit will operate sequentially. Fist we are going to see needed elements for the datapath and see the VHDL implementation of each part. To this end, we will further examine the data path and controller associated with the convolution circuit, review how the utilized sections function for implementing this circuit, and look at their VHDL code.

A. *Data path elements*

1) *register*

As we know, registers are used to store data. So the VHDL implementation can be seen in figure below.

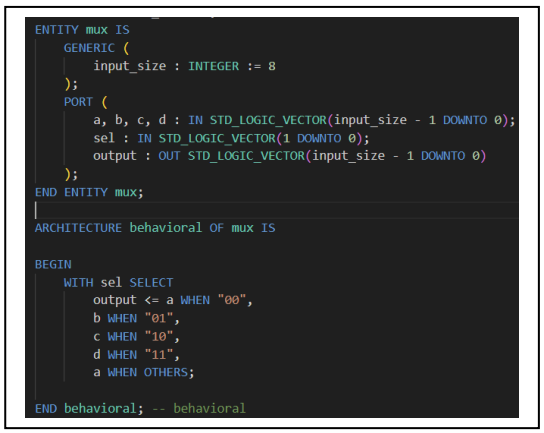
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY reg IS
    GENERIC (register_size : INTEGER := 16);
    PORT (
        clk, rst, en : IN STD_LOGIC;
        d : IN STD_LOGIC_VECTOR(register_size - 1 DOWNTO 0);
        q : OUT STD_LOGIC_VECTOR(register_size - 1 DOWNTO 0));
END ENTITY reg;

ARCHITECTURE behavioral OF reg IS
BEGIN
    PROCESS (clk)
    BEGIN
        IF (clk = '1' AND clk'event and rst = '1') THEN
            q <= (OTHERS => '0');
        ELSIF (clk = '1' AND clk'event and en = '1') THEN
            q <= d;
        END IF;

    END PROCESS;
END behavioral; -- behavioral
```

The implementation is designed such that the register size can be varied using generics, as illustrated. There is multiple instances from this register in datapath that will be discussed in future.

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

2) Mux

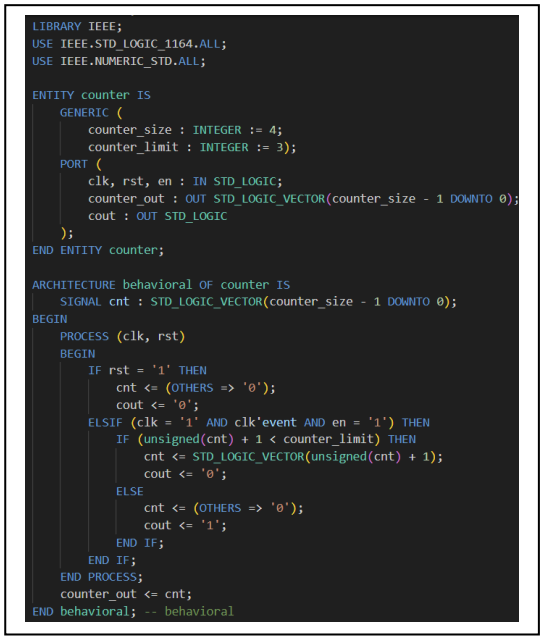
This unit is used to choose between multiple input based on select signal. VHDL implementation can be seen below.



```
ENTITY mux IS
    GENERIC (
        input_size : INTEGER := 8
    );
    PORT (
        a, b, c, d : IN STD_LOGIC_VECTOR(input_size - 1 DOWNTO 0);
        sel : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        output : OUT STD_LOGIC_VECTOR(input_size - 1 DOWNTO 0)
    );
END ENTITY mux;

ARCHITECTURE behavioral OF mux IS

BEGIN
    WITH sel SELECT
        output <= a WHEN "00",
        b WHEN "01",
        c WHEN "10",
        d WHEN "11",
        a WHEN OTHERS;

END behavioral; -- behavioral
```

3) counter

This unit is a counter that start counting on posedge of clk whenever enable signal is one.

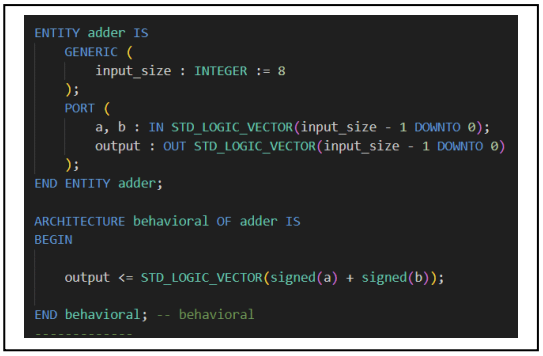


```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY counter IS
    GENERIC (
        counter_size : INTEGER := 4;
        counter_limit : INTEGER := 3);
    PORT (
        clk, rst, en : IN STD_LOGIC;
        counter_out : OUT STD_LOGIC_VECTOR(counter_size - 1 DOWNTO 0);
        cout : OUT STD_LOGIC
    );
END ENTITY counter;

ARCHITECTURE behavioral OF counter IS
    SIGNAL cnt : STD_LOGIC_VECTOR(counter_size - 1 DOWNTO 0);
BEGIN
    PROCESS (clk, rst)
    BEGIN
        IF rst = '1' THEN
            cnt <= (OTHERS => '0');
            cout <= '0';
        ELSIF (clk = '1' AND clk'event AND en = '1') THEN
            IF (unsigned(cnt) + 1 < counter_limit) THEN
                cnt <= STD_LOGIC_VECTOR(unsigned(cnt) + 1);
                cout <= '0';
            ELSE
                cnt <= (OTHERS => '0');
                cout <= '1';
            END IF;
        END IF;
    END PROCESS;
    counter_out <= cnt;
END behavioral; -- behavioral
```

The counter limit which is passed as a generic signal to this counter shows that after the counter reaching that limit the counter will be zero and the cout signal sees a positive pulse on it.

4) adder

this part is use for adding to signals. Code implementation of this part can be seen in figure below.

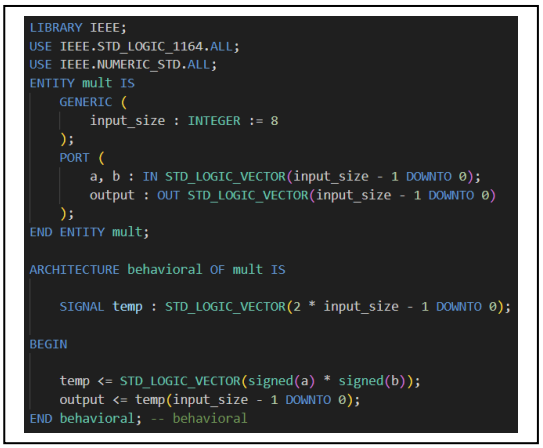


```
ENTITY adder IS
    GENERIC (
        input_size : INTEGER := 8
    );
    PORT (
        a, b : IN STD_LOGIC_VECTOR(input_size - 1 DOWNTO 0);
        output : OUT STD_LOGIC_VECTOR(input_size - 1 DOWNTO 0)
    );
END ENTITY adder;

ARCHITECTURE behavioral OF adder IS
BEGIN

    output <= STD_LOGIC_VECTOR(signed(a) + signed(b));

END behavioral; -- behavioral
```

5) multiplier

This part will calculate the multiply of two input signals.

VHDL implementation can be seen in figure below.

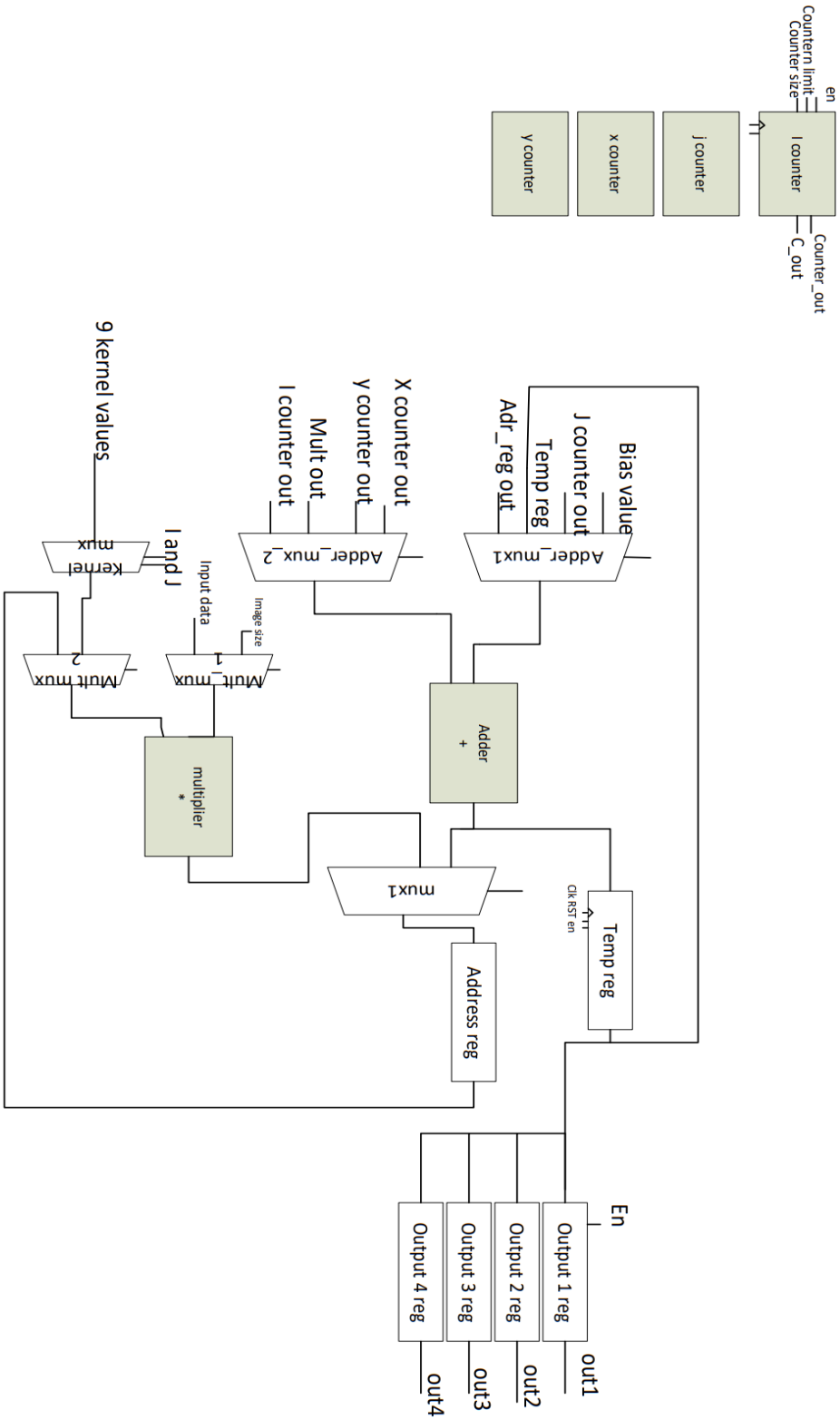


```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
ENTITY mult IS
    GENERIC (
        input_size : INTEGER := 8
    );
    PORT (
        a, b : IN STD_LOGIC_VECTOR(input_size - 1 DOWNTO 0);
        output : OUT STD_LOGIC_VECTOR(input_size - 1 DOWNTO 0)
    );
END ENTITY mult;

ARCHITECTURE behavioral OF mult IS

    SIGNAL temp : STD_LOGIC_VECTOR(2 * input_size - 1 DOWNTO 0);

BEGIN

    temp <= STD_LOGIC_VECTOR(signed(a) * signed(b));
    output <= temp(input_size - 1 DOWNTO 0);
END behavioral; -- behavioral
```

As it can be seen, the output is also 8 bit, we assume the answer is lower than 255.

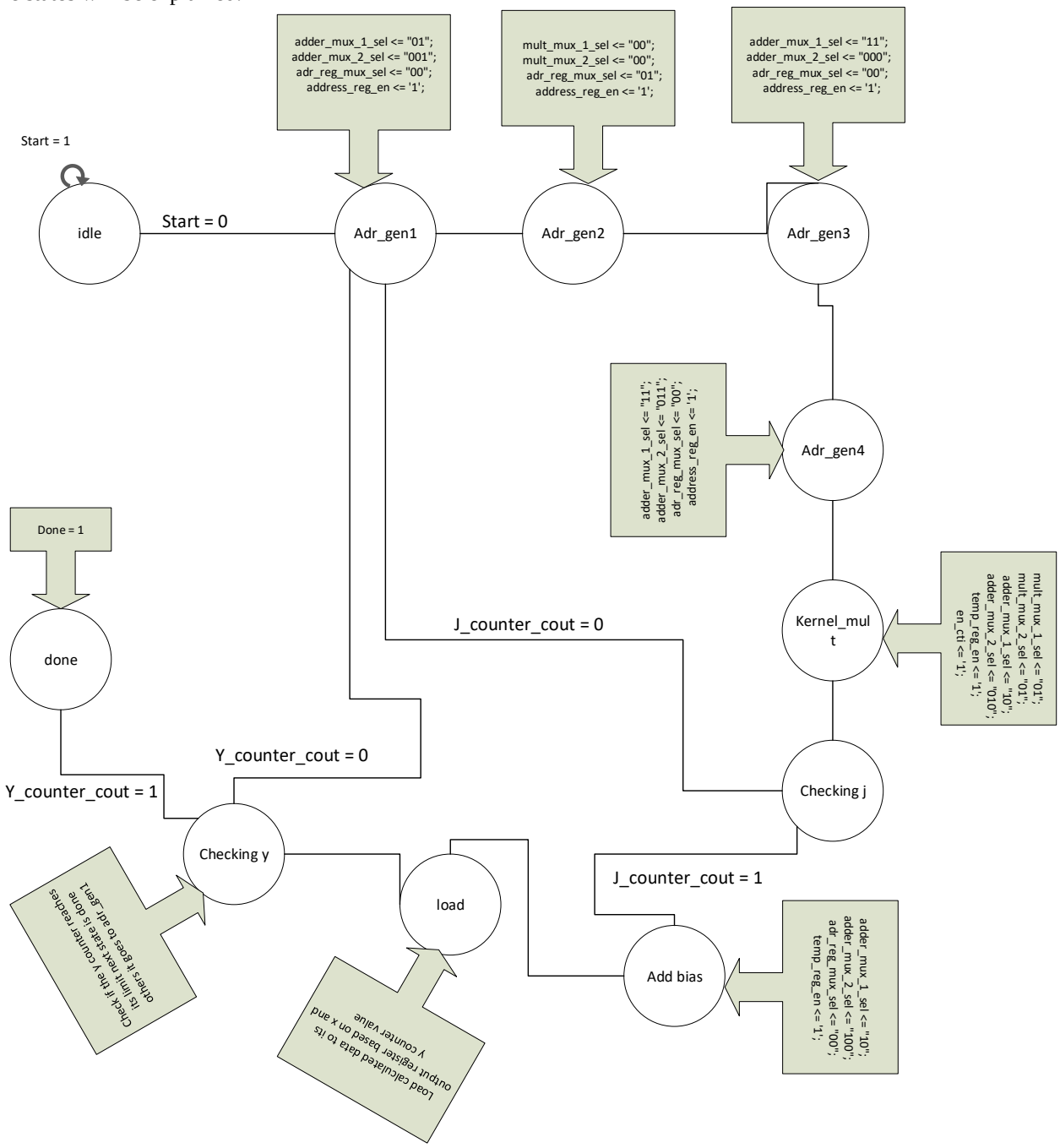
B. datapath

The data path of convolution circuit can be seen in figure below.



C. Controller

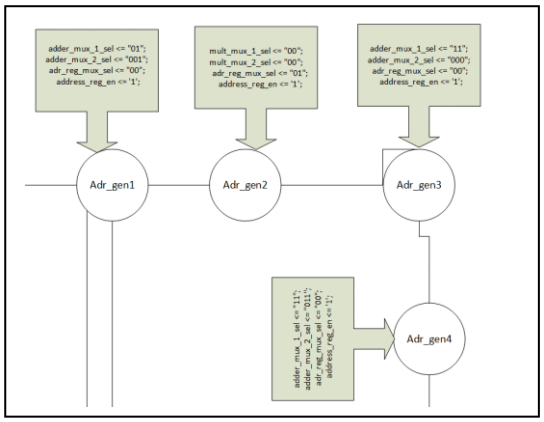
The controller can be seen in figure below. In the following each of the states will be explained.



1) Idle

In this state, the convolution waits for a positive pulse on start to start the calculation. To do so, it stays in the idle state whenever the start is equal to one. When the start signal changes from one to zero it will go to the next state.

2) Address generate states



In this four states, the address of the ram will be calculated. As we know, to calculate the convolution, each cell of the kernel must be multiplied by all the selected 3x3 cells from our 4x4 image. For this purpose, based on i, j and the starting cell x, y, we need to determine the address of the desired pixel in the RAM. Given that we have only one multiplier and one adder, this operation is performed over three cycles.

- In the first state, y+j will be stored in adr_reg
- Then adr_reg will be multiplied with image size
- After that address reg will be added to x
- Eventually adr_reg will be added with i

3) Kernel mult state

In this state, temp register will be updated with kernel_value multiplied with input data which comes from ram based on the address and it will be added to temp register value.

4) Checking j

In this state, we will check if the kernel table observed completely, if so, the next state will be add bias. Otherwise the calculated value doesn't have its final value.

5) Add bias

In this state, the calculated value will be added with bias value and it will be loaded to temp register.

6) Load

The calculated value will be loaded to the respective output register.

7) Checking y

In this state we will check if the image is observed completely and base on that it will chose the next state.

8) Done

In this state, done signal will see a positive pulse to understand that the convolution part process completed.

D. Convolution top module

After completing datapath and controller, we will connect them and generate convolution module which can be seen below.

```
datapath : ENTITY work.convolution_datapath(modular)
    GENERIC MAP(
        bias_value, image_size, data_width, kernel_1, kernel_2, kernel_3,
        kernel_4, kernel_5, kernel_6, kernel_7, kernel_8, kernel_9
    )
    PORT MAP(
        clk, rst, en_cti, en_ctj, en_ctx, en_cty, temp_reg_en, address_reg_en, out1_reg_en, out2_reg_en, out3_reg_en, out4_reg_en,
        data_in,
        data_out1, data_out2, data_out3, data_out4,
        counter_i_cout, counter_j_cout, counter_x_cout, counter_y_cout,
        adder_mux_1_sel, adder_mux_2_sel, adr_reg_mux_sel, mult_mux_1_sel, mult_mux_2_sel,
        counter_x_out, counter_y_out, counter_i_out, counter_j_out, address_out, rst_temp
    );
controller : ENTITY work.convolution_controller(behavioral)
    PORT MAP(
        clk, rst, start, en_cti, en_ctj, en_ctx, en_cty, temp_reg_en, address_reg_en, out1_reg_en, out2_reg_en, out3_reg_en, out4_reg_en,
        counter_i_cout, counter_j_cout, counter_x_cout, counter_y_cout,
        adder_mux_1_sel, adder_mux_2_sel, adr_reg_mux_sel, mult_mux_1_sel, mult_mux_2_sel,
        counter_x_out, counter_y_out, counter_i_out, counter_j_out,
        done, rst_temp
    );
D ARCHITECTURE modular;
```

To making the circuit configurable, we use generic values for kernel part, size of image, data width and bias value which can be seen below.

```
ENTITY convolution IS
    GENERIC (
        bias_value : STD_LOGIC_VECTOR(7 DOWNTO 0) := "00000000";
        image_size : STD_LOGIC_VECTOR(7 DOWNTO 0) := "00000100";
        data_width : INTEGER := 8;
        kernel_1 : STD_LOGIC_VECTOR(7 DOWNTO 0) := (OTHERS => '0');
        kernel_2 : STD_LOGIC_VECTOR(7 DOWNTO 0) := (OTHERS => '0');
        kernel_3 : STD_LOGIC_VECTOR(7 DOWNTO 0) := (OTHERS => '0');
        kernel_4 : STD_LOGIC_VECTOR(7 DOWNTO 0) := (OTHERS => '0');
        kernel_5 : STD_LOGIC_VECTOR(7 DOWNTO 0) := (OTHERS => '0');
        kernel_6 : STD_LOGIC_VECTOR(7 DOWNTO 0) := (OTHERS => '0');
        kernel_7 : STD_LOGIC_VECTOR(7 DOWNTO 0) := (OTHERS => '0');
        kernel_8 : STD_LOGIC_VECTOR(7 DOWNTO 0) := (OTHERS => '0');
        kernel_9 : STD_LOGIC_VECTOR(7 DOWNTO 0) := (OTHERS => '0')
    );
    PORT (
        SIGNAL clk, rst, start : IN STD_LOGIC;
        SIGNAL data_in : IN STD_LOGIC_VECTOR(data_width - 1 DOWNTO 0);
        SIGNAL data_out1, data_out2, data_out3, data_out4 : OUT STD_LOGIC_VECTOR(data_width - 1 DOWNTO 0);
        SIGNAL done : OUT STD_LOGIC;
        SIGNAL address_out : OUT STD_LOGIC_VECTOR(data_width - 1 DOWNTO 0)
    );
END ENTITY convolution;
```

E. Image Ram

This part will be used to store pixel of the image. The code implementation can be seen in figure below.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_TEXTIO.ALL;
USE STD.TEXTIO.ALL;
USE IEEE.std_logic_arith.ALL;

ENTITY ram IS
    GENERIC (
        DATA_WIDTH : INTEGER := 8;
        number_of_rows : INTEGER := 16);
    PORT (
        rst : IN STD_LOGIC;
        data_in : IN STD_LOGIC_VECTOR(DATA_WIDTH - 1 DOWNTO 0);
        address_in_wr, address_in_read : IN STD_LOGIC_VECTOR(data_width-1 DOWNTO 0);
        write_en, read_en : IN STD_LOGIC;
        data_out : OUT STD_LOGIC_VECTOR(DATA_WIDTH - 1 DOWNTO 0));
END ENTITY ram;

ARCHITECTURE behavioral OF ram IS
    TYPE memory_array IS ARRAY (0 TO number_of_rows - 1) OF STD_LOGIC_VECTOR(DATA_WI
    SIGNAL mem : memory_array := (OTHERS => (OTHERS => '0'));
    FILE input_file : TEXT OPEN READ_MODE IS "input_paterns/input1.txt"; -- Open the
    SIGNAL init_done : STD_LOGIC := '0';
BEGIN
    PROCESS (rst, write_en)
        VARIABLE line : LINE;
        VARIABLE text_data : STD_LOGIC_VECTOR(1 TO DATA_WIDTH);
        VARIABLE i : INTEGER := 0;
    BEGIN
        IF init_done = '0' THEN
            WHILE i < 16 LOOP
                readline(input_file, line);
                read(line, text_data);
                mem(i) <= (text_data);
                i := i + 1;
            END LOOP;
            file_close(input_file);
            init_done <= '1';

        ELSIf write_en = '1' and write_en'event then
            mem(conv_integer(unsigned(address_in_wr))) <= data_in;
        END IF;

    END PROCESS;
    data_out <= mem(conv_integer(unsigned(address_in_read)));
END ARCHITECTURE behavioral;
```

As it can be seen the data width can be change based on generic value.

F. Relu

This part will check its input and assign the negative input with zero. The VHDL code can be seen in the next figure.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY relu IS
    GENERIC (
        data_width : INTEGER := 8
    );
    PORT (
        a, b, c, d : IN STD_LOGIC_VECTOR(data_width - 1 DOWNTO 0);
        d1, d2, d3, d4 : OUT STD_LOGIC_VECTOR(data_width - 1 DOWNTO 0)
    );
END ENTITY relu;

ARCHITECTURE behavioral OF relu IS
BEGIN
    PROCESS (a, b, c, d)
    BEGIN

        IF signed(a) < 0 THEN
            d1 <= (OTHERS => '0');
        ELSE
            d1 <= a;
        END IF;

        IF signed(b) < 0 THEN
            d2 <= (OTHERS => '0');
        ELSE
            d2 <= b;
        END IF;

        IF signed(c) < 0 THEN
            d3 <= (OTHERS => '0');
        ELSE
            d3 <= c;
        END IF;

        IF signed(d) < 0 THEN
            d4 <= (OTHERS => '0');
        ELSE
            d4 <= d;
        END IF;

    END PROCESS;
END behavioral; -- behavioral
```

G. *Maxpool*

This module, will show the max value between its inputs on the output. VHDL code can be seen in figure below.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY maxpool IS
    GENERIC (
        data_width : INTEGER := 8
    );
    PORT (
        a, b, c, d : IN STD_LOGIC_VECTOR(data_width - 1 DOWNTO 0);
        output : OUT STD_LOGIC_VECTOR(data_width - 1 DOWNTO 0)
    );
END ENTITY maxpool;

ARCHITECTURE behavioral OF maxpool IS
BEGIN
    PROCESS (a,b,c,d)
        VARIABLE max_var : STD_LOGIC_VECTOR(data_width - 1 DOWNTO 0);
    BEGIN
        max_var := a;

        IF b > max_var THEN
            max_var := b;
        END IF;

        IF c > max_var THEN
            max_var := c;
        END IF;

        IF d > max_var THEN
            max_var := d;
        END IF;
        output <= max_var;
    END PROCESS;
END behavioral; -- behavioral
```

H. *Result module*

This module, will check which of the input is bigger and it will shows as one hot on its output.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY resualt IS
    GENERIC (
        data_width : INTEGER := 8
    );
    PORT (
        a, b, c : IN STD_LOGIC_VECTOR(data_width - 1 DOWNTO 0);
        output : OUT STD_LOGIC_VECTOR(2 DOWNTO 0)
    );
END ENTITY resualt;

ARCHITECTURE behavioral OF resualt IS
BEGIN
    PROCESS (a, b, c)
    BEGIN
        output <= "000";

        IF a > b and a > c THEN
            output <= "001";
        END IF;

        IF b > a and b > c THEN
            output <= "010";
        END IF;

        IF c > b and c > a THEN
            output <= "100";
        END IF;

    END PROCESS;
END behavioral; -- behavioral
```

II. TOP MODULE PATTERN FINDER CIRCUIT

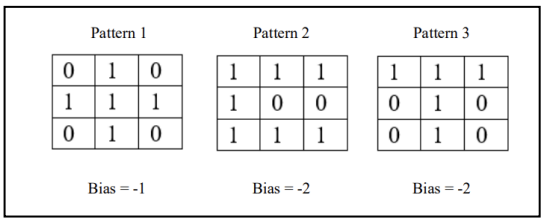
To use three different pattern for three convolution part of the circuit, as it is said in home work, we use a generic value and a generate statement to instantiate modules and the circuit can be configurable.

The configurable architecture of pattern finder circuit can be seen in figure below.

```
ARCHITECTURE configurable OF patter_finder IS
    TYPE maxpool_out IS ARRAY (0 TO number_of_conv - 1) OF STD_LOGIC_VECTOR(DATA_WIDTH - 1 DOWNTO 0);
    SIGNAL maxpools_out : maxpool_out;
BEGIN
    ram1 : ENTITY work.ram(behavioral)
        GENERIC MAP(8, 16)
        PORT MAP(
            rst,
            data_in,
            address_in_wr, address_out,
            write_en, read_en,
            ram_data_out
        );

    gen_circuit : FOR i IN 0 TO number_of_conv - 1 GENERATE
    BEGIN
        conv_i : ENTITY work.convolution(modular)
        GENERIC MAP(
            bias_arrays(i), image_size, 8, kernels_array(i, 0), kernels_array(i, 1), kernels_array(i,
            kernels_array(i, 3), kernels_array(i, 4), kernels_array(i, 5), kernels_array(i, 6), kerne
        )
        PORT MAP(
            clk, rst, start, ram_data_out, convs_ouput(i, 0), convs_ouput(i, 1), convs_ouput(i, 2), c
        );
        relui : ENTITY work.relu(behavioral)
            GENERIC MAP(8)
            PORT MAP(
                convs_ouput(i, 0), convs_ouput(i, 1), convs_ouput(i, 2), convs_ouput(i, 3),
                relus_out(i, 0), relus_out(i, 1), relus_out(i, 2), relus_out(i, 3)
            );
        maxi : ENTITY work.maxpool(behavioral)
            GENERIC MAP(8)
            PORT MAP(
                relus_out(i, 0), relus_out(i, 1), relus_out(i, 2), relus_out(i, 3), maxpools_out(i)
            );

    END GENERATE;

    res2 : ENTITY work.resualt(behavioral)
        GENERIC MAP(8)
        PORT MAP(
            maxpools_out(0), maxpools_out(1), maxpools_out(2), output_pattern
        );

END configurable; -- behavioral
```

III. RESUALTS

As its is said in the home work, the kernel and bias values should be like the figure below.



To configure the circuit with given values, we will configure the pattern finder with generic values which can be seen in below.

```
p_finder : entity work.patter_finder(configurable)
    GENERIC MAP(8, 3, 3,
    ((
    "00000000", "00000001", "00000000",
    "00000001", "00000001", "00000001",
    "00000000", "00000001", "00000000"
    ), (
    "00000001", "00000001", "00000001",
    "00000001", "00000000", "00000000",
    "00000001", "00000001", "00000001"
    ), (
    "00000001", "00000001", "00000001",
    "00000000", "00000001", "00000000",
    "00000000", "00000001", "00000000"
    )),
    ("11111111","11111110","11111110"),
    "00000100")
    PORT MAP(
        start, clk, rst, write_ram,
        data_in, address_in_wr, done,
        output_pattern
    );
```

As it is said in the home work, we will check the circuit for 6 different inputs.
We can change the input by changing RAM loading file as it can be seen below.

```
ARCHITECTURE behavioral OF ram IS
    TYPE memory_array IS ARRAY (0 TO number_of_rows - 1) OF STD_LOGIC_VECTOR(DATA_WIDTH - 1 DOWNTO 0);
    SIGNAL mem : memory_array := (OTHERS => (OTHERS => '0'));
    FILE input_file : TEXT OPEN READ_MODE IS "input_paterns/input4.txt"; -- Open the text file for re
    SIGNAL init_done : STD_LOGIC := '0';
BEGIN
```

A. *input1*

we will load the shown image in the ram and we will check the output

0	0	0	0
1	1	1	1
1	0	0	1
1	1	1	1

the output of the pattern finder circuit can be seen in the next figure.

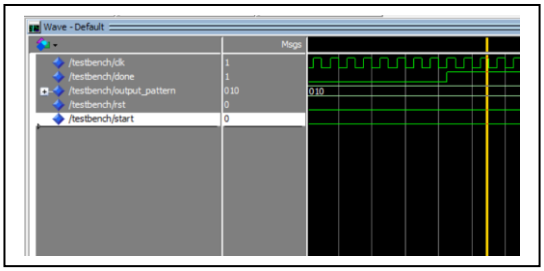
As it can be seen after done signal changes to 1 we can see that the second pattern is found in image.
Output pattern is equal to 010 and it's correctly found the second pattern in the loaded image.

B. *input 2*

we will load the shown image in the ram and we will check the output

1	1	1	0
1	0	0	0
1	1	1	0
0	0	0	1

the output of the pattern finder circuit can be seen in figure below.



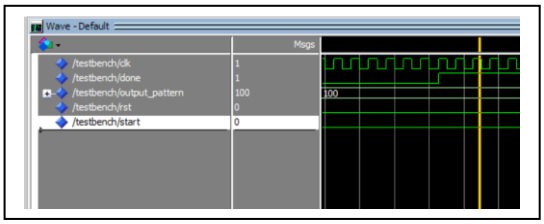
Output pattern is equal to 010 and it's correctly found the second pattern in the loaded image.

C. input 3

We will load the shown image in the ram and we will check the output

0	1	1	1
0	0	1	0
0	0	1	0
0	0	1	0

the output of the pattern finder circuit can be seen in figure below.



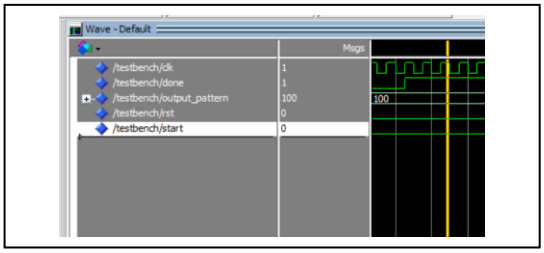
Output pattern is equal to 100 and it's correctly found the third pattern in the loaded image

D. input 4

we will load the shown image in the ram and we will check the output

0	1	1	1
0	0	1	0
0	0	1	0
0	0	0	0

the output of the pattern finder circuit can be seen in figure below.



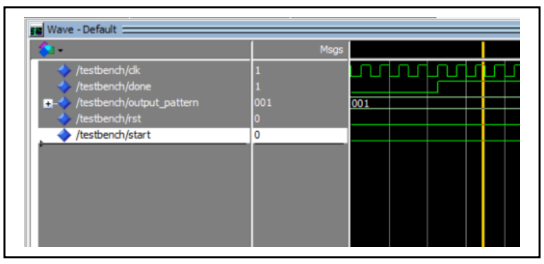
Output pattern is equal to 100 and it's correctly found the third pattern in the loaded image

E. input 5

we will load the shown image in the ram and we will check the output

0	0	1	0
0	1	0	1
0	0	1	0
0	0	0	0

the output of the pattern finder circuit can be seen in figure below.



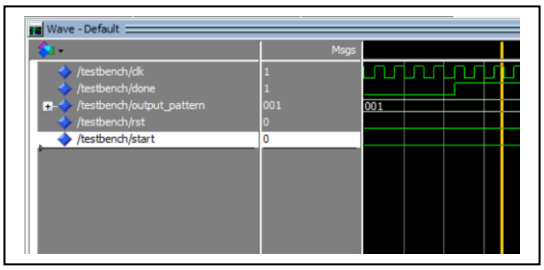
Output pattern is equal to 001 and it's correctly found the first pattern in the loaded image

F. input 6

we will load the shown image in the ram and we will check the output

0	0	0	1
0	1	0	0
1	1	1	1
0	1	0	0

the output of the pattern finder circuit can be seen in figure below.



Output pattern is equal to 001 and it's correctly found the first pattern in the loaded image

All codes can be found in project folder/ codes.
Reports and data path and controller can be found in main folder.
Also all codes and commits can be found in this link