**Εθνικό Μετσόβιο Πολυτεχνείο**
**Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών**

Έτος: 2024- 2025

# *Ψηφιακά VLSI*

**Ομάδα 45**
*Παπαδόπουλος Χαράλαμπος* (ΑΜ: 03120199)
*Νικόλαος Παπακωνσταντόπουλος* (ΑΜ: 03120069)

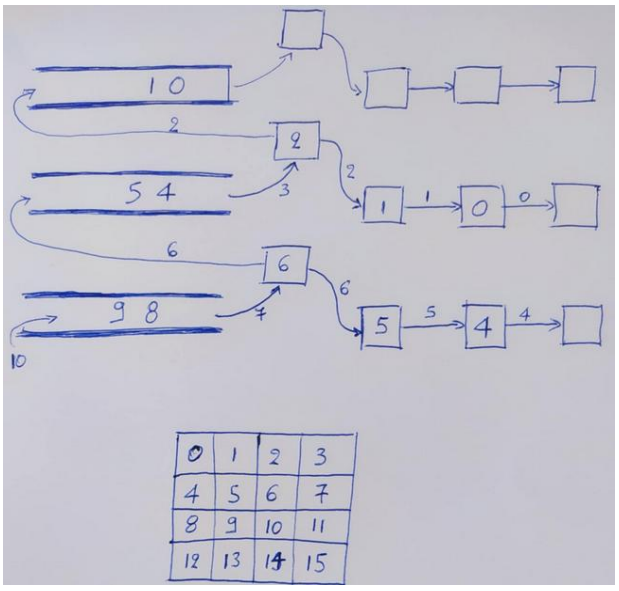
# Υλοποίηση Debayering φίλτρου

## Υλοποίηση

Σε αυτή την άσκηση μας ζητήθηκε να υοποιήσουμε ένα debayering φίλτρο που να δέχεται ως είσοδο μια εικόνα (N x N pixels - 8 bit) και κάποια σήματα ελέγχου και να παράγεται ως έξοδος η φιλτραρισμένη εικόνα N x N (ως έξοδοι R, G, B των 8 bit) με μια καθυστέρηση από την είσοδο του πρώτου pixel. Τα βασικά entities / components που αποτελούν το debayering είναι:

- **calculator**: Η λογική του calculator είναι σχετικά απλή. Βασιζόμενοι στο τελευταίο bit του μετρητή της σειράς και στο τελευταίο bit του μετρητή στήλης διαπιστώνουμε σε ποιο σημείο του 3x3 grid είμαστε. Έπειτα, βασιζόμενοι στο μωσαϊκό που μας δόθηκε, υπολογίζουμε τα κατάλληλα χρώματα.

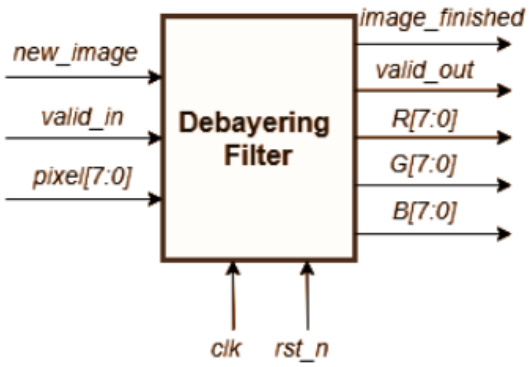
- **serial_to_parallel:** Χρησιμοποιεί 3 FIFOs, 9 (3x3 flip flops για τον σχηματισμό της γειτονιάς του pixel) και άλλα 3 flip flops για την αποθήκευση του στοιχείου που γίνεται pop από τη FIFO (ώστε να δοθεί αφενός στην επόμενη FIFO και αφετέρου στο flip flop της πρώτης στήλης του grid).
  Όσο εισέρχεται η πρώτη σειρά pixels, μπαίνουν τα pixels στην κάτω (1η) FIFO της οποίας το άκρο ανοίγει μόλις μπει και το τελευταίο της σειράς ώστε τα pixels να προωθηθούν στην από πάνω (2η) FIFO και στα D Flip flops του grid.
  Όσο εισέρχεται η δεύτερη σειρά pixels, ανοίγει το άκρο εγγραφής της 2ης FIFO και μπαίνουν σε αυτή τα pixels που ήταν στην 1η FIFO. Επίσης, μπαίνουν τα νέα pixels στην κάτω (1η) FIFO της οποίας το άκρο είναι ανοιχτό τώρα και μόλις μπει και το τελευταίο της σειράς ανοίγει και το άκρο ανάγνωσης.
  Ο σχηματισμός της γειτονιάς του πρώτου pixel αργεί για 2N + 2 κύκλους (αυτός ο αριθμός τίθεται ως INITIAL_OVERHEAD στο **debayering_real** ) και από τότε και μετά, για N*N κύκλους σχηματίζονται οι γειτονιές όλων των pixels
  Λαμβάνεται μέριμνα ώστε στις παραβλέπονται τα out_of_bounds flip-flops σε περίπτωση pixel που είναι στο περιθώριο του N*N grid. Δίνεται εικόνα εκτέλεσης για το 10ο-11ο κύκλο ενός 4*4 grid



**debayering_real**: Είναι η «καρδιά» της υλοποίησης μας. Πρακτικά, ορίζει ένα FSM δύο καταστάσεων:

i) Έρχεται νέα εικόνα και κάνουμε reset
ii) Έχουμε αρχίσει να λαμβάνουμε pixel τα οποία «προωθούμε» στο serial to parallel και στον calculator.

Επιπλέον, περιλαμβάνει έναν μετρητή κύκλων, ώστε μετά από μία συγκεκριμένη καθυστέρηση (latency) 2*N+2 κύκλων να θέσει το valid_out = 1, συμβολίζοντας έτσι πως έχουμε αρχίσει τους υπολογισμούς των χρωμάτων.
Τέλος, όταν ξεπεράσουμε τον συνολικό αριθμό κύκλων που απαιτούνται για τον υπολογισμό μίας εικόνας (συνολικά 2*N+2 + N*N) θέτει το σύστημα σε λειτουργία αδράνειας.




Το **debayering_wrapper.vhd** ορίζει την κύρια οντότητα η οποία στη συνέχεια δοκιμάζεται με ένα testbench που διαβάζει ένα δοσμένο αρχείο εισόδου. Το **debayering_wrapper.vhd** προκαλεί μια καθυστέρηση των εξωτερικών σημάτων valid_in και pixel ώστε το FSM που έχει σχεδιαστεί με διαφορετικό τρόπο στην οντότητα **debayering_real** να αντιλαμβάνεται ότι αν στον 1ο κύκλο έρχεται το new_image, τότε στον 2ο κύκλο έρχεται valid_in = 1 και είναι πλέον έγκυρα τα pixel που έρχονται ως είσοδος (1ο pixel…). Αυτό μας διευκόλυνε στο να κρατήσουμε ίδια και απλή την υλοποίηση των καταστάσεων του FSM όταν έρχεται νέα εικόνα. Επίσης, το **debayering_wrapper.vhd** κάνει διακριτό το image_finished κρατώντας σε έναν καταχωρητή την προηγούμενη τιμή του και συγκρίνοντας με αυτήν (και αυτό μας βόλεψε καθώς στο **debayering_real** , όταν τελειώνει η εικόνα απλώς κάνουμε 1 το image_finished και δεν το μηδενίζουμε).

## calculator.vhd

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


entity calculator is
    generic (
        constant NUM_BITS: natural := 8; -- number of bits for input
vectors.
        constant N: natural := 8

    );
    port (
        clk, rst: in std_logic;
```

```
        p00, p01, p02: in std_logic_vector(NUM_BITS-1 downto 0);
        p10 , p11, p12: in std_logic_vector(NUM_BITS-1 downto 0);
        p20, p21, p22: in std_logic_vector(NUM_BITS-1 downto 0);
        begin_calc: in std_logic;
        cycles_count:                                          in
std_logic_vector(integer(ceil(log2(real(N*N+  2*N+2))))-1  downto
0);
        row_count,                  col_count:                 in
std_logic_vector(integer(ceil(log2(real(N))))-1 downto 0);
        R, G, B: out std_logic_vector(NUM_BITS-1 downto 0)
    );
end calculator;

architecture Behavioral of calculator is
    signal  R_temp,  G_temp,  B_temp:  std_logic_vector(NUM_BITS-1
downto 0) := (others => '0');
begin


calculate: process (clk, rst)
begin
    if rst = '0' then
        R_temp <= (others => '0');
        G_temp <= (others => '0');
        B_temp <= (others => '0');
    end if;
    if rising_edge(clk) and begin_calc = '1' then
        -- checking what color we output...
        if row_count(0) = '0' and col_count(0) = '0' then -- green
            B_temp <= std_logic_vector(resize((unsigned("0" & p10)
+ unsigned("0" & p12))/2, NUM_BITS));
            G_temp <= p11;
            R_temp <= std_logic_vector(resize((unsigned("0" & p01)
+ unsigned("0" & p21))/2, NUM_BITS));
        elsif row_count(0) = '1' and col_count(0) = '0' then -- red
            R_temp <= p11;
            G_temp <= std_logic_vector(resize((unsigned("00"&p01) +
unsigned("00"&p12)  +  unsigned("00"&p21)  +  unsigned("00"&p10))/4,
NUM_BITS));
            B_temp <= std_logic_vector(resize((unsigned("00"&p00) +
unsigned("00"&p02)  +  unsigned("00"&p20)  +  unsigned("00"&p22))/4,
NUM_BITS));
        elsif row_count(0) = '0' and col_count(0) = '1' then -- blue
            B_temp <= p11;
```

```
            G_temp <= std_logic_vector(resize((unsigned("00"&p01) +
unsigned("00"&p12) + unsigned("00"&p21) + unsigned("00"&p10))/4,
NUM_BITS));
            R_temp <= std_logic_vector(resize((unsigned("00"&p00) +
unsigned("00"&p02) + unsigned("00"&p20) + unsigned("00"&p22))/4,
NUM_BITS));
        else
            R_temp <= std_logic_vector(resize((unsigned("0"&p10) +
unsigned("0"&p12))/2, NUM_BITS));
            G_temp <= p11;
            B_temp <= std_logic_vector(resize((unsigned("0"&p01) +
unsigned("0"&p21))/2, NUM_BITS));
        end if;
    end if;
end process;

R <= R_temp;
G <= G_temp;
B <= B_temp;
end Behavioral;
```

## serial_to_parallel.vhd

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

-- This file contains 3 fifos along with 9 registers...
entity serial_to_parallel is
    generic(
        constant FIFO_DEPTH: natural := 1024;
        constant NUM_BITS: natural := 8;
        constant N: natural := 512
    );
    port (
        clk, rst: in std_logic;
        pixel: in std_logic_vector(NUM_BITS-1 downto 0);
        valid_in: in std_logic;
        cycles_count:                                          in
std_logic_vector(integer(ceil(log2(real(N*N+  2*N+2))))-1  downto
0);
        row_count,                  col_count:                 in
std_logic_vector(integer(ceil(log2(real(N))))-1 downto 0);
        p00, p01, p02: out std_logic_vector(NUM_BITS-1 downto 0);
```

```
        p10, p11, p12: out std_logic_vector(NUM_BITS-1 downto 0);
        p20, p21, p22: out std_logic_vector(NUM_BITS-1 downto 0)
    );
end serial_to_parallel;

architecture Behavioral of serial_to_parallel is
-- copied from the fifo template file...
COMPONENT fifo_generator_0
 PORT (
    clk : IN STD_LOGIC;
    srst : IN STD_LOGIC;
    din : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    wr_en : IN STD_LOGIC;
    rd_en : IN STD_LOGIC;
    dout : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
    full : OUT STD_LOGIC;
    empty : OUT STD_LOGIC
 );
END COMPONENT;

-- d flip flops
signal dff00, dff01, dff02: std_logic_vector(NUM_BITS-1 downto
0) := (others => '0');
signal dff10, dff11, dff12: std_logic_vector(NUM_BITS-1 downto
0) := (others => '0');
signal dff20, dff21, dff22: std_logic_vector(NUM_BITS-1 downto
0) := (others => '0');

-- full/empty for fifos
signal full_1, full_2, full_3: std_logic;
signal empty_1, empty_2, empty_3: std_logic;

-- write enable/read enabled/ valid for fifos
signal we_1, we_2, we_3: std_logic := '0';
signal re_1, re_2, re_3: std_logic := '0';

-- data count/fifo outputs for fifos
signal d_cnt_1, d_cnt_2, d_cnt_3: std_logic_vector(9 downto 0);
signal f_out_1, f_out_2, f_out_3: std_logic_vector(NUM_BITS-1
downto 0); -- FIFO outputs

signal f1_to_f2, f2_to_f3, f3_to_dff: std_logic_vector(NUM_BITS-1
downto 0) := (others => '0');
signal reset_reverse: std_logic;
begin
```

```
reset_reverse <= not rst;

f1: fifo_generator_0 port map(
   clk => clk,
   srst => reset_reverse,
   din => pixel,
   wr_en => we_1,
   rd_en => re_1,
   dout => f1_to_f2,
   full => full_1,
   empty => empty_1
);

f2: fifo_generator_0 port map(
   clk => clk,
   srst => reset_reverse,
   din => f1_to_f2,
   wr_en => we_2,
   rd_en => re_2,
   dout => f2_to_f3,
   full => full_2,
   empty => empty_2
);

f3: fifo_generator_0 port map(
   clk => clk,
   srst => reset_reverse,
   din => f2_to_f3,
   wr_en => we_3,
   rd_en => re_3,
   dout => f3_to_dff,
   full => full_3,
   empty => empty_3
);

process (clk, rst)
begin

   -- reset dffs
   if rst = '0' then
       dff00 <= (others => '0'); dff01 <= (others => '0'); dff02
<= (others => '0');
       dff10 <= (others => '0'); dff11 <= (others => '0'); dff12
<= (others => '0');
       dff20 <= (others => '0'); dff21 <= (others => '0'); dff22
<= (others => '0');
```

```
        we_1 <= '0'; we_2 <= '0'; we_3 <= '0';
    elsif rising_edge(clk) then
        -- Only if we have new data!
        we_1 <= '1' and valid_in;

        if unsigned(cycles_count) >= N-1 then
            if valid_in = '1' or unsigned(cycles_count) >= N*N then
                re_1 <= '1';
            else re_1 <= '0';
            end if;
        end if;
        if unsigned(cycles_count) >= 2*N-1 then
            if valid_in = '1' or unsigned(cycles_count) >= N*N then
                re_2 <= '1';
            else re_2 <= '0';
            end if;
        end if;
        if unsigned(cycles_count) >= 3*N-1 then
            if valid_in = '1' or unsigned(cycles_count) >= N*N then
                re_3 <= '1';
            else re_3 <= '0';
            end if;
        end if;
        if unsigned(cycles_count) >= N then
            if valid_in = '1' or unsigned(cycles_count) >= N*N then
                we_2 <= '1';
            else we_2 <= '0';
            end if;
        end if;
        if unsigned(cycles_count) >= 2*N then
            if valid_in = '1' or unsigned(cycles_count) >= N*N then
                we_3 <= '1';
            else we_3 <= '0';
            end if;
        end if;

        if valid_in = '1' or (unsigned(cycles_count) >= N*N+1) then
            dff02 <= dff01; dff01 <= dff00;
            dff12 <= dff11; dff11 <= dff10;
            dff22 <= dff21; dff21 <= dff20;

            dff00 <= f1_to_f2;
            dff10 <= f2_to_f3;
            dff20 <= f3_to_dff;
        end if;
    end if;
```

```
end process;

-- easier to understand. Image looking at a 4x4 box.
-- the image pixels are loaded in the dffs in reversed.
p01 <= dff21 when unsigned(row_count) /= 0 else (others => '0');
p11 <= dff11;
p21 <= dff01 when unsigned(row_count) /= N-1  else (others => '0');

p00 <= dff22 when unsigned(col_count) /= 0 and unsigned(row_count)
/= 0 else (others => '0');
p10 <= dff12 when unsigned(col_count) /= 0 else (others => '0');
p20 <= dff02 when unsigned(col_count) /= 0 and unsigned(row_count)
/= N-1 else (others => '0');

p02    <=    dff20    when    unsigned(col_count)    /=    N-1    and
unsigned(row_count) /= 0 else (others => '0');
p12 <= dff10 when unsigned(col_count) /= N-1 else (others => '0');
p22    <=    dff00    when    unsigned(col_count)    /=    N-1    and
unsigned(row_count) /= N-1 else (others => '0');

end Behavioral;
```

## debayering.vhd

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

-- FSM STATES
-- state 0: New picture comes
-- state 1: New pixel comes
-- state 2: Picture done

entity debayering_real is
   generic(
        constant NUM_BITS: natural := 8; -- rgb values are between
0-255
        constant N: natural := 8
   );
   port(
        clk, rst, valid_in, new_image: in std_logic;
        pixel: in std_logic_vector(NUM_BITS-1 downto 0);
        image_finished, valid_out: out std_logic;
        R, G, B: out std_logic_vector(NUM_BITS-1 downto 0)
```

```
   );
end debayering_real;

architecture Behavioral of debayering_real is
signal   p00,   p01,   p02,   p10,   p11,   p12,   p20,   p21,   p22:
std_logic_vector(NUM_BITS-1 downto 0);

component serial_to_parallel is
generic (
       constant N: natural := 8);
port(
   clk, rst: in std_logic;
   pixel: in std_logic_vector(NUM_BITS-1 downto 0);
   valid_in: in std_logic;
   row_count,                    col_count:                    in
std_logic_vector(integer(ceil(log2(real(N))))-1 downto 0);
   cycles_count:  in  std_logic_vector(integer(ceil(log2(real(N*N+
2*N+2))))-1 downto 0);
   p00, p01, p02: out std_logic_vector(NUM_BITS-1 downto 0);
   p10, p11, p12: out std_logic_vector(NUM_BITS-1 downto 0);
   p20, p21, p22: out std_logic_vector(NUM_BITS-1 downto 0)
);
end component;
component calculator is
generic (
       constant N: natural := 8
);
port(
   clk, rst: in std_logic;
   p00, p01, p02: in std_logic_vector(NUM_BITS-1 downto 0);
   p10 , p11, p12: in std_logic_vector(NUM_BITS-1 downto 0);
   p20, p21, p22: in std_logic_vector(NUM_BITS-1 downto 0);
   begin_calc: in std_logic;
   row_count,                    col_count:                    in
std_logic_vector(integer(ceil(log2(real(N))))-1 downto 0);
   cycles_count:  in  std_logic_vector(integer(ceil(log2(real(N*N+
2*N+2))))-1 downto 0);
   R, G, B: out std_logic_vector(NUM_BITS-1 downto 0)
);
end component;


-- pixel counter. We will receive a total of N*N pixels in total!
signal  cycles_count:  std_logic_vector(integer(ceil(log2(real(N*N+
2*N+2))))-1 downto 0);
```

```
signal  col_count:  std_logic_vector(integer(ceil(log2(real(N))))-1
downto 0) := (others => '0');
signal  row_count:  std_logic_vector(integer(ceil(log2(real(N))))-1
downto 0) := (others => '0');
signal R_exit, G_exit, B_exit: std_logic_vector(NUM_BITS-1 downto
0) := (others => '0'); -- calculated RGB values.
signal begin_calc: std_logic := '0'; -- can we begin calculating???
-- previous values of row/col. For synchronization (see below)
signal               row_count_prev,               col_count_prev:
std_logic_vector(integer(ceil(log2(real(N))))-1   downto   0)   :=
(others => '0');

-- how many cycle in order to do first calculation. PEIRAMATIKO
constant INITIAL_OVERHEAD: natural := 2*N+2;
constant TOTAL_OPERATION_COST: natural := INITIAL_OVERHEAD + N*N;

-- boolean indicating if new_image='1' has come in a previous cycle
(so we are calculating pixels and not stalling)
signal new_image_received: std_logic := '0';
begin


-- serial-to-parallel and calculator need to see the same column/row
in order to properly synchronize
-- otherwise you lose 3 extra hours of sleep
row_col_prev: process(clk, rst)
begin
   if rst = '0' then
       row_count_prev <= (others => '0');
       col_count_prev <= (others => '0');
   elsif rising_edge(clk) then
       row_count_prev <= row_count;
       col_count_prev <= col_count;
   end if;
end process;

-- Serial to Parallel connections
stp: serial_to_parallel generic map (N => N) port map(
   clk => clk, rst => rst, pixel => pixel, valid_in => valid_in,
   cycles_count => std_logic_vector(cycles_count),
   row_count => row_count_prev, col_count => col_count_prev,
   p00 => p00, p01 => p01,p02 => p02, p10 => p10,
   p11 => p11, p12 => p12,p20 => p20, p21 => p21, p22 => p22
);

-- Calculator connections
```

```
calc: calculator generic map (N => N) port map(
   clk => clk, rst => rst,
   p00 => p00, p01 => p01,p02 => p02, p10 => p10, p11 => p11, p12
=> p12,p20 => p20, p21 => p21, p22 => p22,
   cycles_count => std_logic_vector(cycles_count),
   begin_calc => begin_calc,
   row_count => row_count_prev, col_count => col_count_prev,
   R => R_exit, G => G_exit, B => B_exit
);

fsm: process(clk, rst)
begin
   if rst = '0' then
       image_finished <= '0';
       valid_out <= '0';
       cycles_count <= (others => '0');
       begin_calc <= '0';
       col_count <= (others => '0');
       row_count <= (others => '0');
       new_image_received <= '0';
   elsif rising_edge(clk) then
       if new_image = '1' then -- FSM state 1
           -- if new image comes then we must make sure we reset
the state
           -- (a new image could come before the previous finished
because the user desided to change it)
           new_image_received <= '1';
           image_finished <= '0';

           cycles_count <= (others => '0');
           col_count <= (others => '0');
           row_count <= (others => '0');
           begin_calc <= '0';
       elsif new_image = '0' and new_image_received = '1' then --
FSM state 2
       -- we have started receiving pixels
       -- we need to keep record of the column/row

           cycles_count <= std_logic_vector(unsigned(cycles_count)
+ 1);
           if unsigned(cycles_count) >= INITIAL_OVERHEAD + 1 then
               valid_out <= '1';
           end if;
           if unsigned(cycles_count) >= INITIAL_OVERHEAD then
               begin_calc <= '1';
```

```
                if unsigned(col_count) = N - 1 then
                    col_count <= (others => '0');
                    row_count                                   <=
std_logic_vector(unsigned(row_count) + 1);
                else
                    col_count                                   <=
std_logic_vector(unsigned(col_count) + 1);
                end if;
            end if;
        end if;

        if unsigned(cycles_count) >= TOTAL_OPERATION_COST then
            row_count <= (others => '0');
            col_count <= (others => '0');
            cycles_count <= (others => '0');
            image_finished <= '1';
            new_image_received <= '0';
            begin_calc <= '0';

        end if;
    end if;

end process;

R <= R_exit;
G <= G_exit;
B <= B_exit;


end Behavioral;
```

## debayering_wrapper.vhd

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

entity debayering is
    generic(
        constant NUM_BITS: natural := 8; -- rgb values are between
0-255
        constant N: natural := 8
    );
    port(
```

```
        clk, rst, valid_in, new_image: in std_logic;
        pixel: in std_logic_vector(NUM_BITS-1 downto 0);
        image_finished, valid_out: out std_logic;
        R, G, B: out std_logic_vector(NUM_BITS-1 downto 0)
    );
end debayering;

architecture Behavioral of debayering is
    signal valid_in_1 : std_logic := '0';
        signal pixel_1 : std_logic_vector(NUM_BITS-1 downto 0) :=
(others => '0');

    signal valid_out_1 : std_logic;
    signal image_finished_1 : std_logic;

    signal valid_out_prev : std_logic := '0';
    signal image_finished_prev : std_logic := '0';
begin
    debayer: entity work.debayering_real generic map (NUM_BITS =>
NUM_BITS, N =>N) port map (
                                clk => clk,
                                rst => rst,
                                valid_in => valid_in_1,
                                new_image => new_image,
                                pixel => pixel_1,
                                image_finished => image_finished_1,
                                valid_out => valid_out_1,
                                R => R,
                                G => G,
                                B => B
                            );

    delay: process(clk, rst)
    begin
        if rst = '0' then
            valid_in_1 <= '0';
            pixel_1 <= (others => '0');
            image_finished_prev <= '0';

        elsif rising_edge(clk) then
            valid_in_1 <= valid_in;
            pixel_1 <= pixel;
            image_finished_prev <= image_finished_1;
        end if;
    end process;
```

```
    valid_out <= valid_out_1;
    image_finished <= '1' when image_finished_prev = '0' and
image_finished_1 = '1' else '0';
end architecture;
```

## debayering_tb.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use ieee.math_real.all;
use std.textio.all; -- Import the textio package for file I/O
operations

entity debayering_tb is
end debayering_tb;

architecture testbench of debayering_tb is
    constant NUM_BITS : natural := 8;
    constant N : natural := 32;
    constant CLK_PERIOD : time := 10 ns;  -- Clock period
    signal clk : std_logic := '0';  -- Clock signal
    signal rst : std_logic := '0';  -- Reset signal
    signal valid_in, new_image : std_logic := '0';   -- Control
signals
    signal pixel : std_logic_vector(NUM_BITS-1 downto 0);  -- Pixel
input
    signal image_finished, valid_out : std_logic;  -- Output signals
    signal R, G, B : std_logic_vector(NUM_BITS-1 downto 0);  -- RGB
outputs

    file pixel_file : TEXT; -- Declare the file handle
    shared variable pixel_line : LINE; -- Line buffer
    shared variable pixel_value : integer; -- Variable to hold pixel
value
    file rgb_file : TEXT; -- Declare the file handle for RGB values
    shared variable rgb_line : LINE; -- Line buffer for writing RGB
values
begin
    -- Instantiate the DUT
    DUT : entity work.debayering
        generic map (
            NUM_BITS => NUM_BITS,
            N => N
```

```
        )
        port map (
            clk => clk,
            rst => rst,
            valid_in => valid_in,
            new_image => new_image,
            pixel => pixel,
            image_finished => image_finished,
            valid_out => valid_out,
            R => R,
            G => G,
            B => B
        );

    -- Stimulus process
    stim_proc : process
    begin
        -- Open the file for reading
        file_open(pixel_file,   "/home/nikolaospapa3/Documents/ECE-
NTUA/dvlsi/dvlsi-ntua/ex6/scripts/bayer_matrix.txt", READ_MODE);
        file_open(rgb_file,     "/home/nikolaospapa3/Documents/ECE-
NTUA/dvlsi/dvlsi-ntua/ex6/scripts/vivado_output.txt",
WRITE_MODE);
        -- Reset DUT
        rst <= '0';  -- Assert reset
        wait for CLK_PERIOD;
        rst <= '1';  -- Deassert reset
        new_image <= '1';
        valid_in <= '1';
        wait for CLK_PERIOD;
        -- Read pixel values from file
        while (not image_finished) = '1' loop
            if not endfile(pixel_file) then
                readline(pixel_file, pixel_line); -- Read a line
from the file
                read(pixel_line, pixel_value); -- Read an integer
value from the line
                pixel <= std_logic_vector(to_unsigned(pixel_value,
NUM_BITS)); -- Convert to std_logic_vector

                -- Set control signals
                new_image <= '0';
                valid_in <= '1';  -- Assert valid_in
            end if;

            wait for CLK_PERIOD/2;  -- Wait for half a clock period
```

```
            valid_in <= '0';  -- Deassert valid_in
            wait for CLK_PERIOD / 2;  -- Wait for half a clock period

            if valid_out = '1' then
                write(rgb_line,
integer'image(to_integer(unsigned(R)))      &      ",      "      &
integer'image(to_integer(unsigned(G)))      &      ",      "      &
integer'image(to_integer(unsigned(B)))); -- Write RGB values to the
line
                writeline(rgb_file, rgb_line); -- Write the line to
the file
            end if;
        end loop;

        -- Close the file
        file_close(pixel_file);
        file_close(rgb_file);


        wait;  -- Wait forever
    end process;

    clk <= not clk after CLK_PERIOD/2;
end testbench;
```