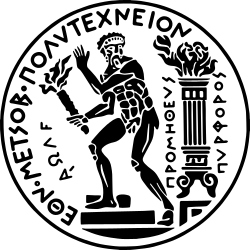
**Εθνικό Μετσόβιο Πολυτεχνείο** 

**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

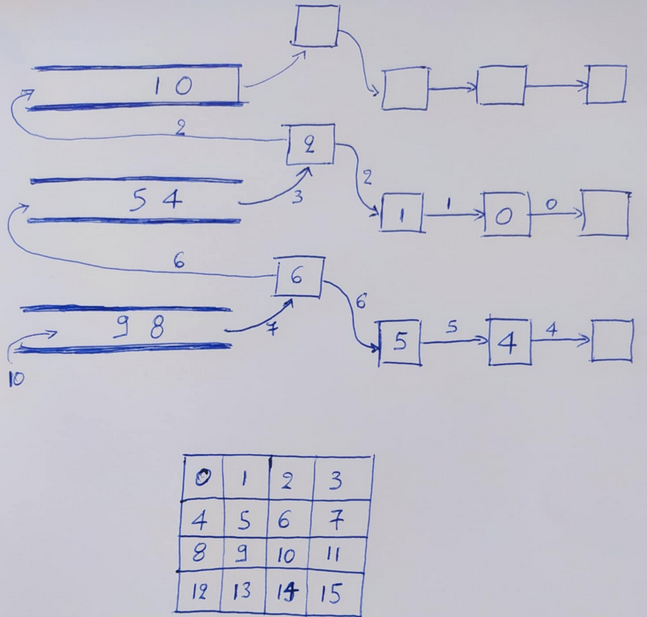
Σε αυτή την άσκηση μας ζητήθηκε να υοποιήσουμε ένα debayering φίλτρο που να δέχεται ως είσοδο μια εικόνα (N x N pixels - 8 bit) και κάποια σήματα ελέγχου και να παράγεται ως έξοδος η φιλτραρισμένη εικόνα N x N (ως έξοδοι R, G, B των 8 bit) με μια καθυστέρηση από την είσοδο του πρώτου pixel. Τα βασικά 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 10o-11o κύκλο ενός 4\*4 grid

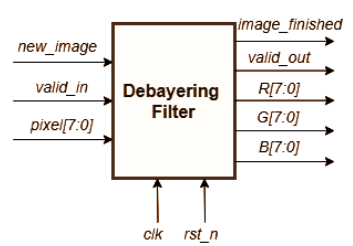


**debayering\_real**: Είναι η «καρδιά» της υλοποίησης μας. Πρακτικά, ορίζει ένα FSM δύο καταστάσεων:

1. Έρχεται νέα εικόνα και κάνουμε reset
2. 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\*Ν+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.



To **debayering\_wrapper.vhd** ορίζει την κύρια οντότητα η οποία στη συνέχεια δοκιμάζεται με ένα testbench που διαβάζει ένα δοσμένο αρχείο εισόδου. To **debayering\_wrapper.vhd** προκαλεί μια καθυστέρηση των εξωτερικών σημάτων valid\_in και pixel ώστε το FSM που έχει σχεδιαστεί με διαφορετικό τρόπο στην όντότητα **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;