Sommario

[0. Project specifications 1](#_Toc106577281)

[1. Introduction 1](#_Toc106577282)

[1.1 Field of applications and examples 2](#_Toc106577283)

[1.2 Algorithm Description 2](#_Toc106577284)

[1.3 Possible Architectures 2](#_Toc106577285)

[2. Architecture Description 3](#_Toc106577286)

[2.1 Counter 3](#_Toc106577287)

[2.2 Subtractor 4](#_Toc106577288)

[2.3 Accumulator 4](#_Toc106577289)

[2.4 General SAD architecture 4](#_Toc106577290)

[3. VHDL code 4](#_Toc106577291)

# 0. Project specifications

1) Design a synchronous digital circuit that implement the SAD calculation, defined as the sum of pixel values absolute differences between two blocks of monochromatic images A and B. Consider 2 blocks of size 16 pixel x 16 pixel, each pixel has an integer value between 0 and 255 represented in 8 bit. The inputs are the clock, the reset, the enable and PA and PB signals that will be provided by an external source, the 256 pixel values of A and B are provided sequentially in different clock cycles. The output is a SAD signal in 16 bits and a data\_valid signal of 1 bit. On reset SAD=0 and data\_valid=0. Data\_valid is set at the end of the SAD calculation. If enable=0 the circuit keeps its state independently from the input values.

2) Define a VHDL parametrized version of point 1 for the case of NxN image blocks, with N power of 2. Note: The input and output signals are the same such that with a SAD ouput of n bits:

The timing constraints must be changed because there are pixel couples.

# 1. Introduction

The sum of absolute differences is a measure of similarity between image blocks, it consists in calculating the absolute differences between each pixel value of the two images and summing them to obtain a similarity metric. The higher the value the lower the similarity between the 2 blocks.

Immagine che contiene testo, orologio

Descrizione generata automaticamente

Figure 1: SAD computation of two R × C matrices **A** and **B**

## 1.1 Field of applications and examples

SAD is used for different purposes like object recognition, video compression, disparity maps and others.

For example, in the case of **object detection**, the environment light changing, and other factors can lead to misinterpretation of objects. In this context sad is used as a complementary method working with other type of object detection, like edge detection, to cope with these problems and to improve the reliability of the results.

In **digital image processing** the sum of absolute differences is used to identify which part of a search image is most similar to a template image. In the following example we have the comparison of a 3x3 template and a 3x5 search image where each pixel is represented with an integer value from 0 to 9.

Immagine che contiene testo

Descrizione generata automaticamente

Figure 2: Template and Search Image pixel values

The template can fit in the left side of the search image, in the center and in the right side, shifting the block by one column for each case. Calculating the absolute differences between each pixel:

Immagine che contiene testo

Descrizione generata automaticamente

Figure 3: left, center and right absolute differences

For each case we can add the 9 absolute differences to get the SAD values, respectively: 20, 25, 17. We can state that the right side of the search image is the most similar to the template, that’s because it’s the lower value.

## 1.2 Algorithm Description

The algorithm used is the basic one and it consists in three main operations:

* Taking the pixel values couples sequentially
* Calculate the absolute difference
* Sum to the total value (SAD)

Those steps repeat until the counter threshold is reached, for example considering 16x16 blocks the total number of sums is 256. Further improvements can be made considering parallel solution (see next paragraph).

## 1.3 Possible Architectures

In “*Efficient sum of absolute difference computation on FPGAs”* (see References)the circuit exploits parallelism in the absolute difference calculation, in the image below there is the proposed architecture.

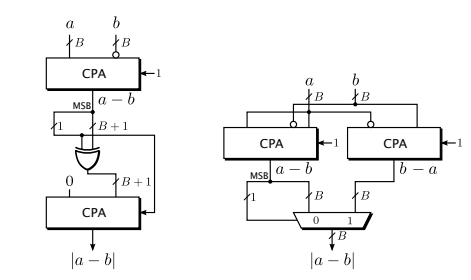


Figure 4: Sequential vs Parallel Absolute Difference Calculation circuits

In this case, instead of negating the result if negative, both and are calculated in parallel and the positive result is selected by the most significant bit. In this way the complexity is higher, but the critical path is shorter. A third solution is evaluating if and then perform one of the 2 operations, this is simpler and faster due to the carry-chain MUXes of the Xilinx architecture.

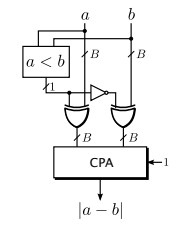


Figure 5: FPGA optimized absolute difference circuits

To improve the performances in the SAD calculation some basic SAD core units can be put in parallel like shown in *Figure 5*.

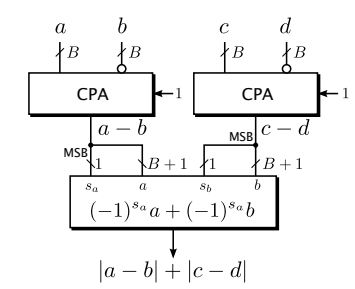


Figure 6: SAD core units in parallel

*Note:* CPA refers to carry propagate adder.

# 2. Architecture Description

Three different components have been considered to implement the SAD calculation: clock counter, subtractor, accumulator. A classical D-flip-flop implementation has been used to keep the circuit state.

## 2.1 Counter

This component is used to understand the end of the SAD calculation. When the enable signal is set it counts the clock cycles and so the number of sums performed in total, otherwise the counting is freeze. When the counting value overflows it means that the SAD calculation is finished, the overflow signal triggers the accumulator to stop summing the inputs and to set the data\_valid signal.

MAYBE SOME SCHEMA HERE OR PSEUDO CODE?

## 2.2 Subtractor

The subtractor is a combinational circuit that calculate the absolute difference of the 2 inputs, it is implemented following the logic of the parallel absolute difference presented before.

## 2.3 Accumulator

The accumulator adds the output coming from the subtractor to the total SAD value, when it receives the overflow signal from the counter it adds the last absolute difference and then set the data\_valid signal.

## 2.4 General SAD architecture

The general architecture can be summarized using the following scheme:

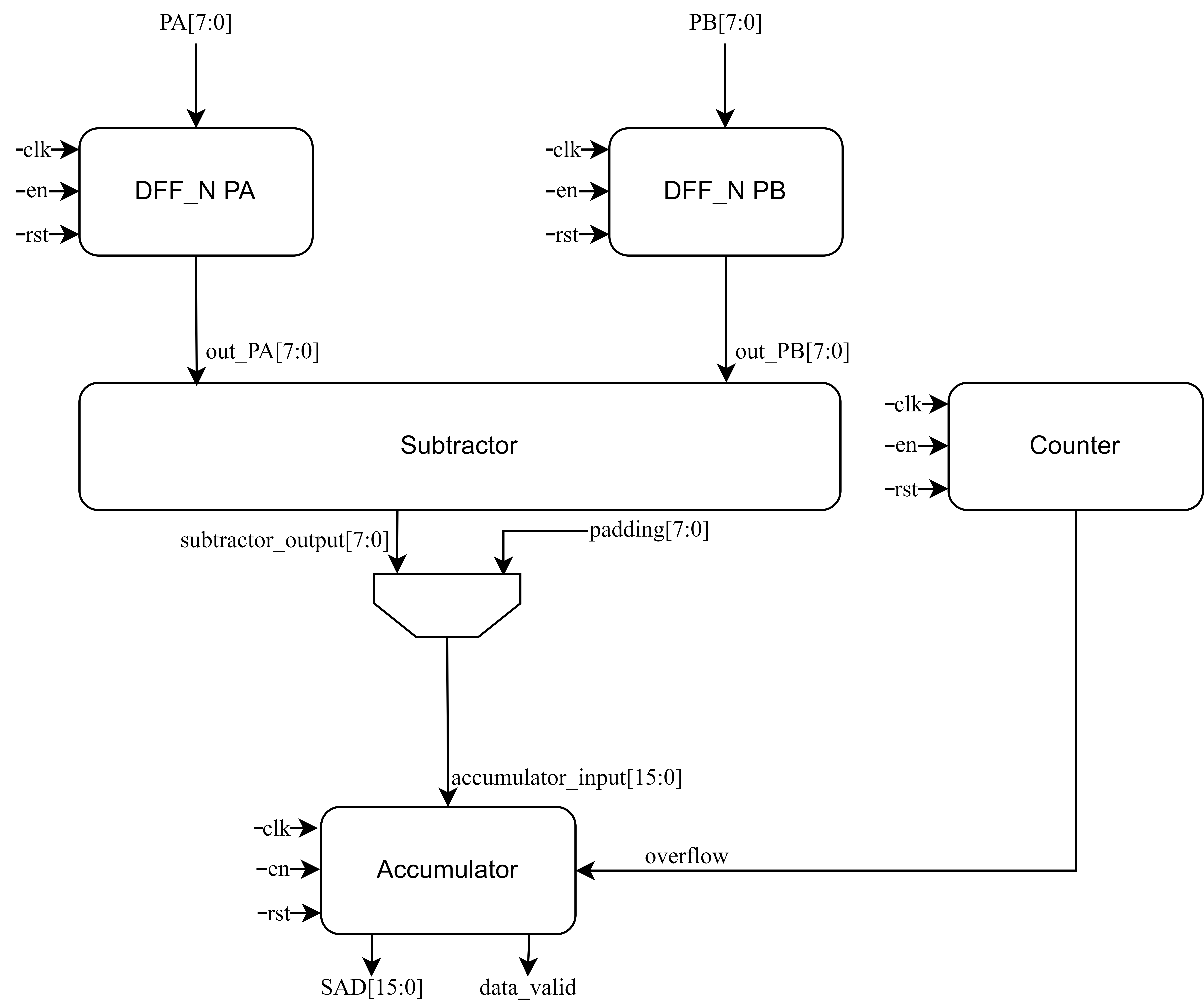


Figure 7: General SAD architecture

# 3. VHDL code

The architecture described in VHDL follows a behavioral approach and they are inspired by the solutions presented before. The developing of the project follows a bottom-up approach where starting from the implementation of the main components we put them all together to build the complete circuit.

## 3.1 DFF\_N:

This is a classical implementation of d-flip-flop with set/reset signals.

**library** ieee**;**

**use** ieee**.**std\_logic\_1164**.all;**

**entity** DFF\_N **is**

**generic(** NBit **:** positive **:=** 8**);**

**port(**

clk **:** **in** std\_logic**;**

a\_rst\_n **:** **in** std\_logic**;**

en **:** **in** std\_logic**;**

d **:** **in** std\_logic\_vector**(**NBit **-** 1 **downto** 0**);**

q **:** **out** std\_logic\_vector**(**NBit **-** 1 **downto** 0**)**

**);**

**end** DFF\_N**;**

**architecture** struct **of** DFF\_N **is**

**begin**

ddf\_n\_proc**:** **process(**clk**,** a\_rst\_n**)**

**begin**

**if(**a\_rst\_n **=** '1'**)** **then**

q **<=** **(others** **=>** '0'**);**

**elsif(rising\_edge(**clk**))** **then**

**if(**en **=** '1'**)** **then**

q **<=** d**;**

**end** **if;**

**end** **if;**

**end** **process;**

**end** struct**;**

## 3.2 Counter

The counter architecture is equipped with an output register to maintain the counting value, the latter is freeze if the enable signal is not set. When max\_val threshold is reached the counter overflows and set the corresponding signal.

**library** IEEE**;**

**use** IEEE**.**STD\_LOGIC\_1164**.ALL;**

**use** IEEE**.**STD\_LOGIC\_ARITH**.all;**

**use** IEEE**.**STD\_LOGIC\_UNSIGNED**.all;**

**entity** counter **is**

**generic** **(**

NBit **:** positive **:=** 8

**);**

**port** **(**

en**:** **in** std\_logic**;**

rst**,**clk **:** **in** std\_logic**;**

overflow**:** **out** std\_logic**;**

o**:** **out** std\_logic\_vector**(** NBit**-**1 **downto** 0 **)**

**);**

**end** counter**;**

-- when en = '1' it counts the clock cycles, otherwise the counting is freezed

-- set overflow to 1 when max\_val is reached

**architecture** beh **of** counter **is**

**signal** output\_reg **:** std\_logic\_vector **(**NBit**-**1 **downto** 0**);**

**constant** max\_val **:** std\_logic\_vector **(**NBit**-**1 **downto** 0**)** **:=** **(others** **=>** '1'**);**

**begin**

counter\_output\_reg**:** **process(**clk**,** rst**)**

**begin**

**if(**rst **=** '1'**)** **then**

output\_reg **<=** **(others** **=>** '0'**);**

overflow **<=** '0'**;**

**elsif((rising\_edge(**clk**))** **AND** en**=**'1'**)** **then**

-- when max\_val is reached the counter overflows and overflow is set to '1'

**if** **(**unsigned**(**output\_reg**)** **=** unsigned**(**max\_val**))** **then**

output\_reg **<=** **(others** **=>** '0'**);**

overflow **<=** '1'**;**

**else**

output\_reg **<=** output\_reg **+** 1**;**

**end** **if;**

**end** **if;**

**end** **process** counter\_output\_reg**;**

-- mapping the output

o **<=** output\_reg**;**

**end** beh**;**

## 3.4 accumulator

The accumulator sums the inputs on clock edges if it is enabled and if the counter has not already overflowed. When it receives the overflow signal from counter, the accumulator performs the last sum and then set data\_valid to 1 to advertise that the result is ready.

**library** ieee**;**

**use** ieee**.**std\_logic\_1164**.all;**

**use** ieee**.**numeric\_std**.all;**

**use** IEEE**.**std\_logic\_unsigned**.all;**

**entity** accumulator **is**

**generic** **(**

NBit**:** positive **:=** 16

**);**

**port(**

i**:** **in** std\_logic\_vector**(**NBit**-**1 **downto** 0**);**

rst**,**clk **:** **in** std\_logic**;**

en**:** **in** std\_logic**;**

counter\_ow**:** **in** std\_logic**;**

o**:** **out** std\_logic\_vector**(** NBit**-**1 **downto** 0 **);**

data\_valid**:** **out** std\_logic

**);**

**end** accumulator**;**

-- Sum the input at each clock if en = '0', set data\_valid to 1 when counter overflows

**architecture** beh **of** accumulator **is**

**signal** output\_oreg **:** std\_logic\_vector **(**NBit**-**1 **downto** 0**);**

**signal** data\_valid\_oreg **:** std\_logic**;**

**begin**

accumulator\_p**:** **process(**rst**,** clk**)**

**begin**

**if(**rst **=** '1'**)** **then**

output\_oreg **<=** **(others** **=>** '0'**);**

data\_valid\_oreg **<=** '0'**;**

**elsif(rising\_edge(**clk**)** **and** en**=**'1' **and** data\_valid\_oreg**=**'0'**)** **then**

output\_oreg **<=** std\_logic\_vector**(**unsigned**(**output\_oreg**)** **+** unsigned**(**i**));**

-- check on counter overflow value to prevent the restart of summing

-- when overflow

**if(**counter\_ow **=** '1'**)** **then**

data\_valid\_oreg **<=** '1'**;**

**end** **if;**

**end** **if;**

**end** **process** accumulator\_p**;**

-- Mapping the output

data\_valid **<=** data\_valid\_oreg**;**

o **<=** output\_oreg**;**

**end** beh**;**

Descrivi il tuo algoritmo e avvalora la tua tesi del behavioral model, per esempio puoi dire che è più semplice da scrivere e rende il codice più leggibile. Una volta fatto ciò prosegui con la descrizione del tuo algoritmo utilizzando anche dello pseudo code, a seguire una architettura fisica possibile.

Afferma anche che hai usato il behavioral model per definire la logica sequenziale, mentre RTL per descrivere quella combinatoria.

Infine dici che farai riferimenti a ciò che ha trovato il synthetizer quando fa il design.

I riferimenti a possibili implementazioni lo metti in fondo.

The architecture described in VHDL follows a behavioral approach and they are inspired by the solutions presented before.