SAD calculation of two monochrome images

Francesco Urbani

University of Pisa

PSM, Final Project Report
Spring 2018

Submitted June 1, 2018

Contents

List of Figures

0 Project specifications

Design a synchronous digital system that calculates the SAD value, defined as the sum of the magnitude of the differences of each pixel of two (monochrome) images, image A and image B. Each pixel's value is stored in `N`-bits and each image has `px` pixels per side.

The circuit has a *clock*, a *reset* and an *enable* signal as inputs together with the signals *PA* and *PB* representing one pixel of each image flowing into the system from the outside, sequentially. The circuit has a `M`-bits output signal called *SAD* where:

$$2^{\mathtt{M}} \geq (2^{\mathtt{N}} - 1) \cdot \mathtt{px}^2 \tag{1}$$

and a *DATA_VALID* signal which is set when the SAD calculation is completed.

When the reset is active, *SAD* and *DATA_VALID* are 0. If the enable signal is zero the system does not change its current state.

1 Introduction

In digital image processing, the sum of absolute differences (SAD) is a measure of the similarity between image blocks. It is calculated by taking the absolute difference between each pixel in the original block and the corresponding pixel in the block being used for comparison. These differences are summed to create a simple metric of block similarity, the L1 norm of the difference image or Manhattan distance between two image blocks.

The sum of absolute differences may be used for a variety of purposes, such as object recognition, the generation of disparity maps for stereo images, and motion estimation for video compression.

The sum of absolute differences provides a simple way to automate the searching for objects inside an image, but may be unreliable due to the effects of contextual factors such as changes in lighting, color, viewing direction, size, or shape. The SAD may be used in conjunction with other object recognition methods, such as edge detection, to improve the reliability of results.

1.1 State of the art architectures

- Implementation of SAD Algorithm with Folded Tree Architecture using VHDL
  (http://www.internationaljournalssrg.org/IJECE/2015/Volume2-Issue7/IJECEV2I7P107.pdf)

- A Highly Parallel SAD Architecture for Motion Estimation in HEVC Encoder
  http://unitec.researchbank.ac.nz/bitstream/handle/10652/3813/PID3380667.pdf

2 Algorithm description

The two images are fed into the system by means of two signals, called *PA* and *PB*, whose size is `N`, i.e. the number of bits representing the pixel grade on a scale from 0 (i.e. 100% black) to $2^N - 1$, typically 255 (i.e. 100% white).

Being the system synchronous, it works on the clock rising-edge.

The other two inputs are the *reset* (active high) and the *enable*, that prevents the SAD value from being updated when it's on its low state.

The two outputs are *SAD* and *DATA_VALID*.

SAD represent the actual information we are interested in, and it's defined as the sum of the absolute differences of each pixel of the two images. Therefore it's value is updated at each clock-cycle and it is only frozen whenever the *enable* is low or the calculation is completed.

DATA_VALID goes high when the SAD calculation is finished and it stays high until a reset occurs.

3 Architecture

The following is a basic architecture representation of the algorithm just described, with the wires labeled according to the specific case
`N`=8, `px`=16 (⇒ `SAD_bits`=16),
but can be extended to any number of bits, later in the code.

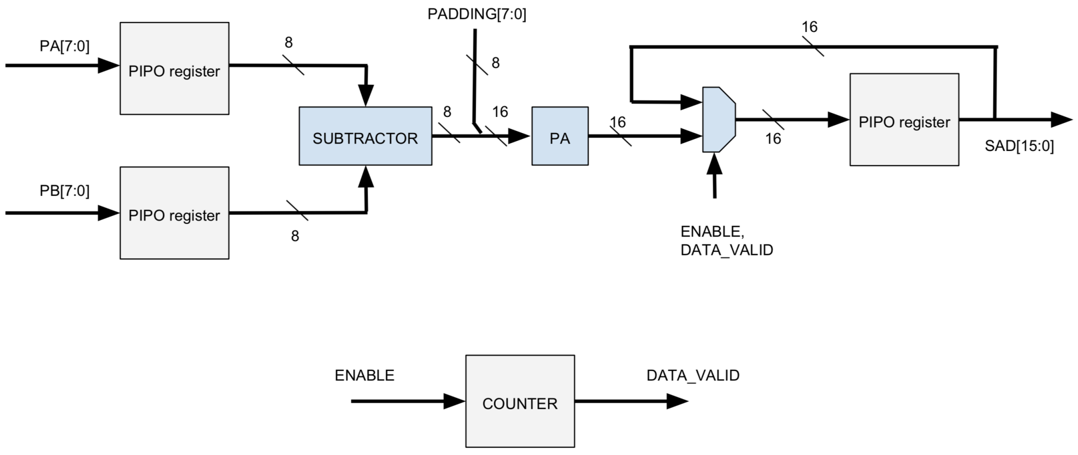


Figure 1: Architecture diagram

3.1 Design choices

- the signal *padding* has been necessary to "tune" the upstream signal out of the subtractor to the downstream signal. It's type is `std_logic_vector [SAD_bits-N-1 downto 0]` and it is `x"00"` in the default configuration.

- PA is a phase accumulator;

- the multiplexer following the PA has the function to freeze the value of the output SAD when it must not be changed, that's why the control signal of the MUX depends upon *enable* and *DATA_VALID*, more specifically it is `S <= EN nand (not RESET)`, i.e. it lets the signal flow ahead only when *enable* is `1` and *DATA_VALID* is `0`;

- the counter counts up to $\texttt{px}^2+3$, which means that the whole image has been processed. It then sets its output to 1 until the system is newly reset.

  This is because 3 clock-cycles is the propagation of *PA* and *PB* towards the system's output *SAD*. Putting such displacement lets the system stay in sync and not erroneously set *DATA_VALID* to 1 before the actual calculation is completed.

- since when *enable* is not active the system is required to maintain its state, i.e. being transparent to the input signals *PA* and *PB*, the input reset signal into the "first stage" PIPO registers, is also driven by the *enable* signal, more specifically:

  `RST_IN <= EN nand (not RST_global).`

4 VHDL code

4.1 Source structure

A structural approach has been followed. After writing the `.vhd` files describing all the components that I would have been using in the top-level one, I've simply wired them up together to create the final design, which is `SAD.vhd`, see Appendix A for reference.

The only two sub-components described with a *behavioral* approach have been `counter.vhd` and `subtractor.vhd`.

For additional explanation about this section, see the comments along the code (`./src/*` and `./tb/*`).

4.2 Test benches

In order to write a meaningful test-bench that would actually test a real-life situation – meaning that the two images are most likely different from each other pixel by pixel – writing a script to automate this process turned out to be almost mandatory.

That's right what `./scripts/tb_SAD_generator.py` does: it takes (interactively) as input the number of bits representing each pixel value (`N`) and the number of pixels/side, i.e. the square root of the total number of pixels composing each image (`px`), and it calculates beforehand the minimum number of bits necessary to store the SAD value w/o information loss, according to the following expression:

$$\lceil \log_2((2^{\mathtt{N}} - 1) \cdot \mathtt{px}^2) \rceil \tag{2}$$

and asks the user whether he wants to increase that number, called `SAD_bits`.

Secondly, the script generates two square matrices with random integer elements in the range $[0, 2^N)$, it calculates a third matrix whose elements are the magnitude of the difference of the former two matrices' elements and eventually it calculates the overall sum of it's elements, i.e. the expected SAD value.

Eventually it generates/overwrites the test bench file `./tb/tb_SAD.vhd` coherently to the model so that comparing the model to the waveform result is a trivial task.

A second test bench has also been developed first, for a quick debug: two values of PA and PB enters the system without ever changing – say PA=0; PB=1 – and checking SAD value to be equal to $\mathtt{px}^2$ at the end of the calculation is a preliminary way to check whether the system's working correctly.

4.3 Compiling instructions

I've used the open-source compiler GHDL to compile the VHDL code in conjunction with the waveform viewer Scansion (macOS) which is an alternative to GTKWave (cross-platform).

Once organized the folder like this:

.

```
├── .work
├── scripts
├── src
├── tb
├── vivado
└── waveforms
```

move into `.work/` and from your terminal type:

```
ghdl -i ../src/*
ghdl -i ../tb/*
```

to import all the `.vhd` files into the working directory. Then:

```
ghdl -m sad
```

to compile the top-level component. That will compile all the components instantiated into the top-level component as well. Then type:

```
ghdl -m tb_sad
```

to compile the test bench of the top-level component. And finally:

```
ghdl -r tb_sad --vcd=../waveforms/tb_SAD.vcd
```

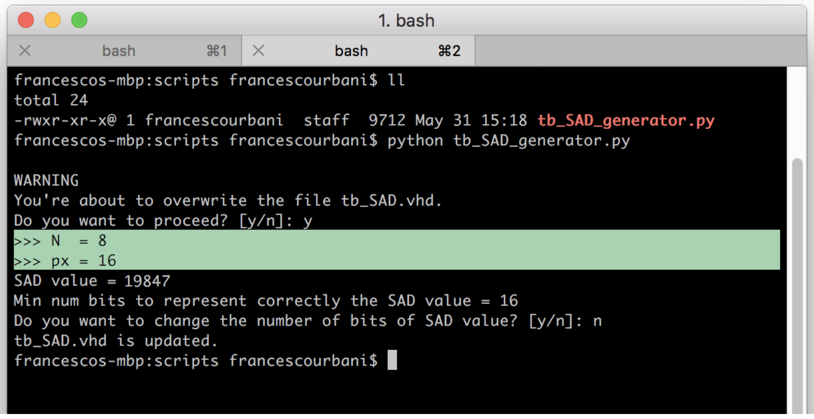
to create the waveform file `tb_SAD.vcd` which can be afterwards opened with one of those two above mentioned waveform viewers.

5 Simulations

5.1 Simulation 1

Inputs: N=8; px=16. Number of bits to correctly store SAD=16. This is the default case given by the specification requirements.

This version is the one that will be later implemented.



```
francescos-mbp:scripts francescourbani$ ll
total 24
-rwxr-xr-x@ 1 francescourbani  staff  9712 May 31 15:18 tb_SAD_generator.py
francescos-mbp:scripts francescourbani$ python tb_SAD_generator.py

WARNING
You're about to overwrite the file tb_SAD.vhd.
Do you want to proceed? [y/n]: y
>>> N  = 8
>>> px = 16
SAD value = 19847
Min num bits to represent correctly the SAD value = 16
Do you want to change the number of bits of SAD value? [y/n]: n
tb_SAD.vhd is updated.
francescos-mbp:scripts francescourbani$
```

Figure 2: Test bench generation, first simulation

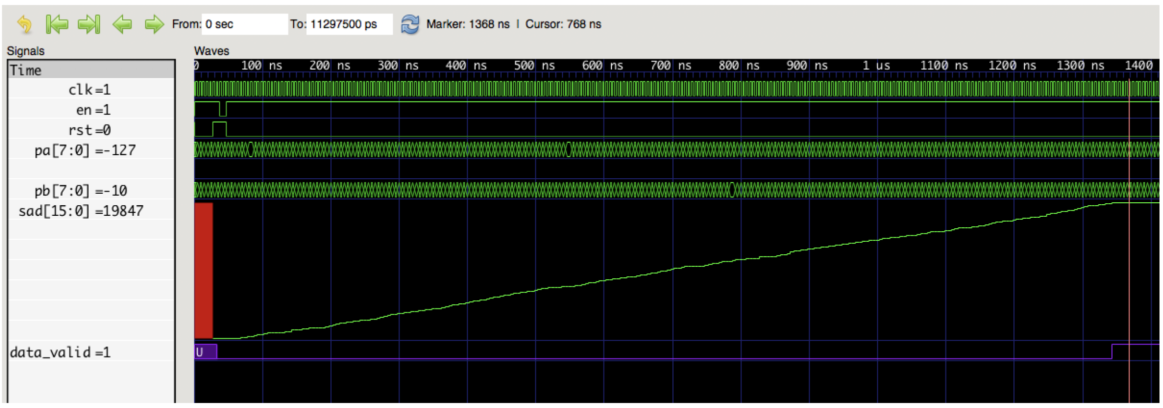


Figure 3: Waveform result, first simulation.

We can see from fig. 3 that when `data_valid` is set, `sad[15:0]` corresponds to the expected value given by the script.

We can see from fig. 4 that when `en` is 0, the SAD value does not increase, nor creates a discontinuity when the system resumes its operation, meaning that the state of the system does not change, as requested by the specs. This

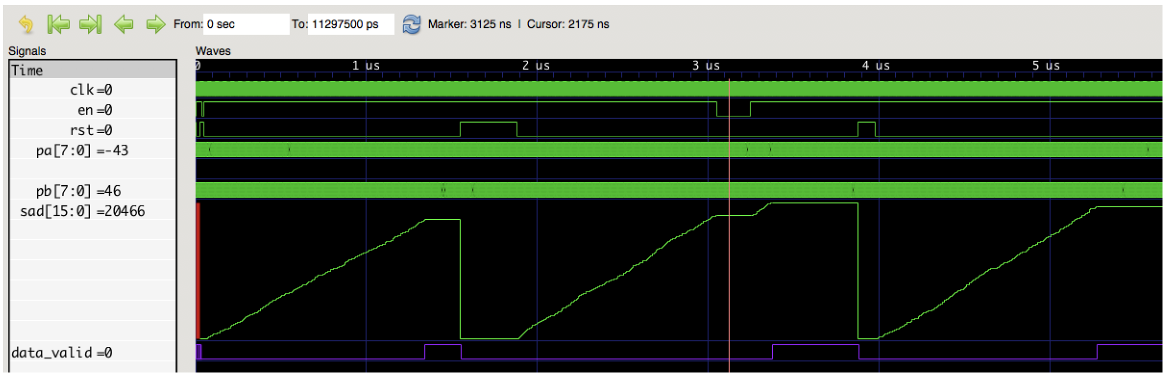


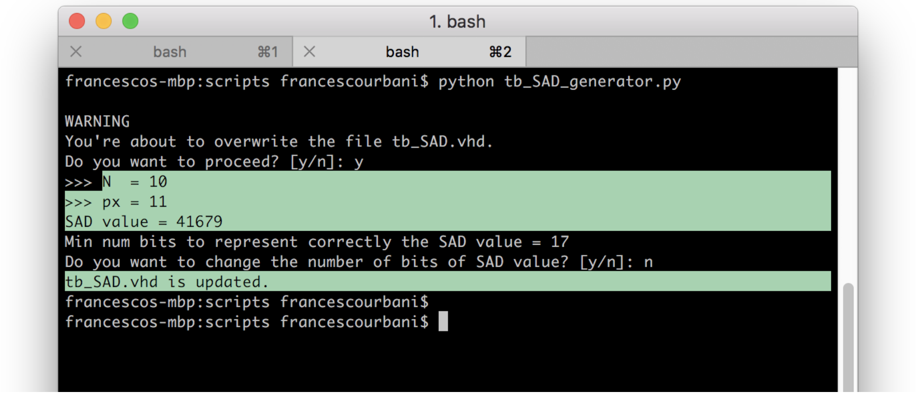
Figure 4: Waveform result, first simulation, unzoomed

is due to the fact that the *enable* signal in its low state also resets the two input PIPO registers, preventing them from processing their inputs.

If this was not specifically planned, even though the *SAD* output would be frozen, the system would still be internally processing the inputs coming in, and creating a sort of discontinuity of the output signal *SAD* when the system would resume to it's normal operational state.

5.2 Simulation 2

Inputs: `N`=10; `px`=11. Number of bits to correctly store `SAD`=17.



```
francescos-mbp:scripts francescourbani$ python tb_SAD_generator.py

WARNING
You're about to overwrite the file tb_SAD.vhd.
Do you want to proceed? [y/n]: y
>>> N  = 10
>>> px = 11
SAD value = 41679
Min num bits to represent correctly the SAD value = 17
Do you want to change the number of bits of SAD value? [y/n]: n
tb_SAD.vhd is updated.
francescos-mbp:scripts francescourbani$
francescos-mbp:scripts francescourbani$
```

Figure 5: Test bench generation, second simulation

Again, we can see (fig. 5) that changing the input parameters leads to a re-adaptation the number of bits to correctly store *SAD*, which turns out to be 17 with these two input values.

The results match.

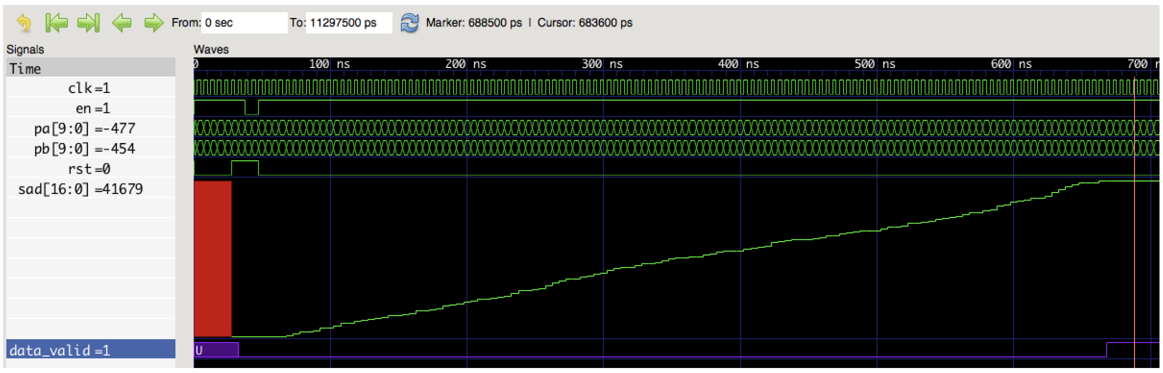


Figure 6: Waveform result, second simulation.

5.3 Simulation 3

Inputs: `N`=8; `px`=16.

This is actually the dummy simulation made beforehand to check some preliminary results.

We can see that leaving the two inputs `pa[7:0]` and `pb[7:0]` unchanged and fixed, respectively, at `1` and `0`, leads to a SAD value, once complete, of `256` which is actually the total number of pixels given as input, (`px`^{2}).

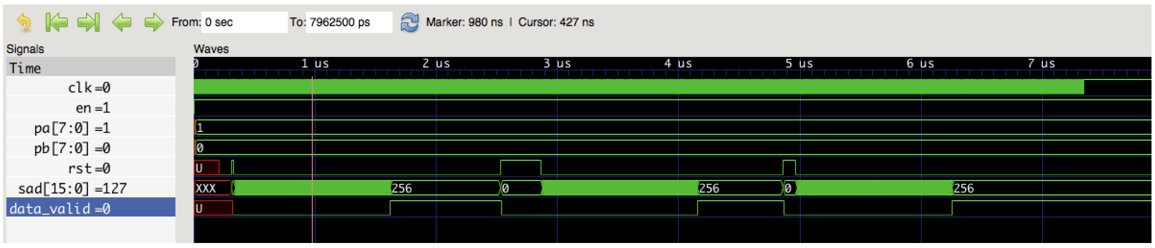


Figure 7: Waveform result, third simulation.

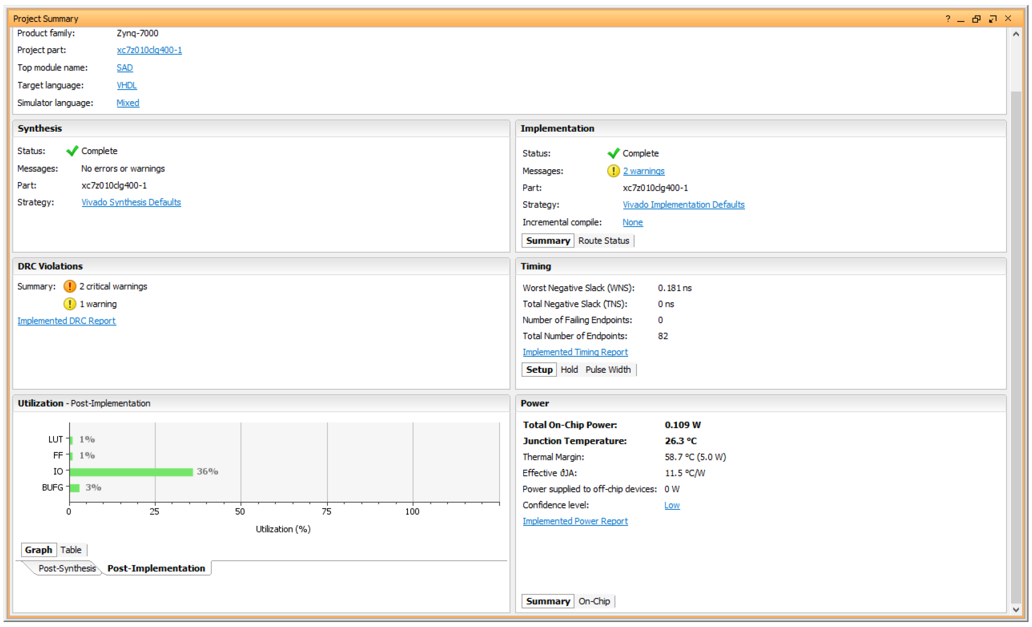
6 Synthesis and implementation

I will now walk through the synthesis and implementation of my design onto the Xilinx Zynq-7000 `xc7z010clg400-1` FPGA, through it's proprietary HDL design suite Vivado.

What matter most at this point of the design is evaluating the maximum clock speed that the system is able to reach reliably, along with the critical path, the power consumption and the resources utilization.

After setting a clock period of 6 ns (i.e. 167 MHz) as time constraint for the synthesis and implementation engines, we can now analyze the results given by the tool.

This is the compact *Project Summary* that Vivado provides after the synthesis and implementation are successfully executed, (*timing constraints are met*).


Project Summary

Product family:	Zynq-7000
Project part:	xc7z010clg400-1
Top module name:	SAD
Target language:	VHDL
Simulator language:	Mixed

Synthesis

Status:	✓ Complete
Messages:	No errors or warnings
Part:	xc7z010clg400-1
Strategy:	Vivado Synthesis Defaults

Implementation

Status:	✓ Complete
Messages:	2 warnings
Part:	xc7z010clg400-1
Strategy:	Vivado Implementation Defaults
Incremental compile:	None

DRC Violations

Summary: 2 critical warnings
1 warning
Implemented DRC Report

Timing

Worst Negative Slack (WNS):	0.181 ns
Total Negative Slack (TNS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	82

Implemented Timing Report

Utilization - Post-Implementation

	Utilization (%)
LUT	1%
FF	1%
IO	36%
BUFG	3%

Power

Total On-Chip Power:	**0.109 W**
Junction Temperature:	**26.3 °C**
Thermal Margin:	58.7 °C (5.0 W)
Effective ϑJA:	11.5 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Implemented Power Report

Figure 8: Vivado Project Summary

In order to gain even more detailed information and have a deeper understanding of what's going on, some Tcl commands have been issued to the Vivado Tcl console:

- `report_utilization -file rpt_utilization.rpt` to generate a resources utilization report and store it into a file;
- `report_timing_summary -file rpt_timing_summary.rpt` to generate timing report and store it into a file;
- `report_power -file rpt_power.rpt` to generate a power consumption report and store it into a file;
- `report_route_status -file rpt_route_status.rpt`
- `report_clock_interaction -file rpt_clk_interaction.rpt`
- `report_clock_networks -file rpt_clk_networks.rpt`

and got all these files automatically saved into the folder `./vivado/SAD` to be opened and read via a text editor.

6.1 Utilization report

What's worth highlighting about the content of the file `rpt_utilization.rpt` is:

```
1. Slice Logic
--------------

+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs              |   56 |     0 |     17600 |  0.32 |
|   LUT as Logic          |   56 |     0 |     17600 |  0.32 |
|   LUT as Memory         |    0 |     0 |      6000 |  0.00 |
| Slice Registers         |   83 |     0 |     35200 |  0.24 |
|   Register as Flip Flop |   83 |     0 |     35200 |  0.24 |
|   Register as Latch     |    0 |     0 |     35200 |  0.00 |
| F7 Muxes                |    0 |     0 |      8800 |  0.00 |
| F8 Muxes                |    0 |     0 |      4400 |  0.00 |
+-------------------------+------+-------+-----------+-------+
2. Slice Logic Distribution
---------------------------

+-------------------------------------------+------+-------+-----------+-------+
|                 Site Type                 | Used | Fixed | Available | Util% |
+-------------------------------------------+------+-------+-----------+-------+
| Slice                                     |   35 |     0 |      4400 |  0.80 |
|   SLICEL                                  |   24 |     0 |           |       |
|   SLICEM                                  |   11 |     0 |           |       |
| LUT as Logic                              |   56 |     0 |     17600 |  0.32 |
|   using O5 output only                    |    0 |       |           |       |
|   using O6 output only                    |   26 |       |           |       |
|   using O5 and O6                         |   30 |       |           |       |
| LUT as Memory                             |    0 |     0 |      6000 |  0.00 |
|   LUT as Distributed RAM                  |    0 |     0 |           |       |
|   LUT as Shift Register                   |    0 |     0 |           |       |
| LUT Flip Flop Pairs                       |   29 |     0 |     17600 |  0.16 |
|   fully used LUT-FF pairs                 |   15 |       |           |       |
|   LUT-FF pairs with one unused LUT output |   14 |       |           |       |
|   LUT-FF pairs with one unused Flip Flop  |   14 |       |           |       |
| Unique Control Sets                       |    4 |       |           |       |
+-------------------------------------------+------+-------+-----------+-------+
```

which shows that the device is mostly unutilized, as we can also see from the *device view* (fig 9).

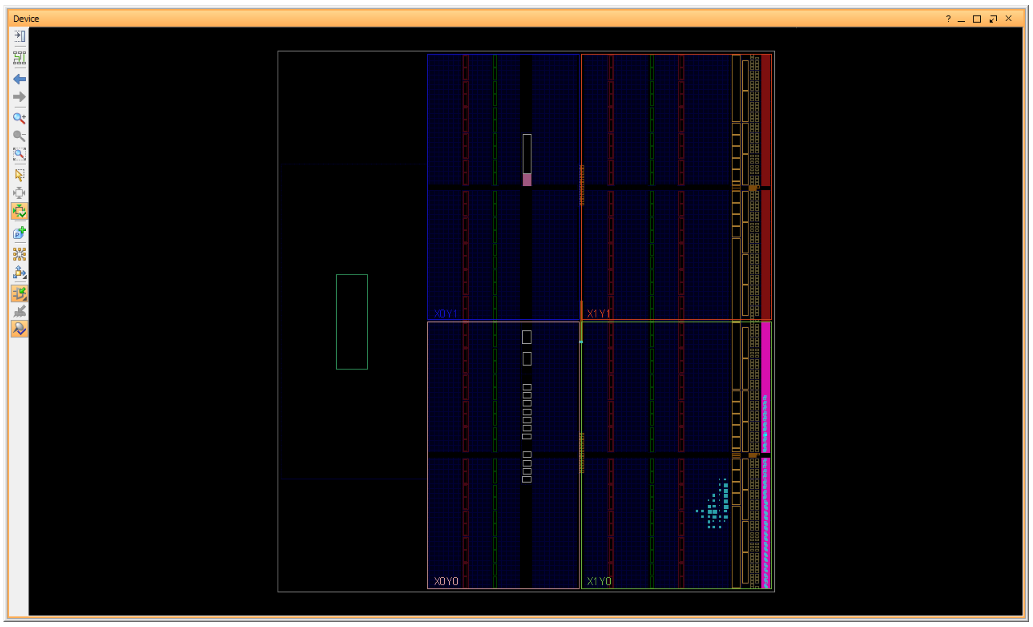


Figure 9: Vivado, device view

6.2 Timing summary report

Let's now open `rpt_timing_summary.rpt`

```
Max Delay Paths
--------------------------------------------------------------------------------
Slack (MET) :              0.181ns  (required time - arrival time)
  Source:                  reg_PB/reg_gen[3].i_dff/q_reg/C
                             (rising edge-triggered cell FDCE clocked by clk_6ns
  Destination:             phaseacc/i_reg/reg_gen[13].i_dff/q_reg/D
                             (rising edge-triggered cell FDCE clocked by clk_6ns
  Path Group:              clk_6ns
  Path Type:               Setup (Max at Slow Process Corner)
  Requirement:             6.000ns  (clk_6ns rise@6.000ns - clk_6ns rise@0.000ns)
  Data Path Delay:         5.755ns  (logic 2.498ns (43.403%)  route 3.257ns (56.5
  Logic Levels:            8  (CARRY4=2 LUT3=1 LUT4=2 LUT5=2 LUT6=1)
  Clock Path Skew:         -0.060ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    4.455ns = ( 10.455 - 6.000 )
    Source Clock Delay      (SCD):    4.970ns
    Clock Pessimism Removal (CPR):    0.455ns
  Clock Uncertainty:        0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter      (TSJ):    0.071ns
    Total Input Jitter       (TIJ):    0.000ns
    Discrete Jitter           (DJ):    0.000ns
    Phase Error               (PE):    0.000ns
--------------------------------------------------------------------------------
```

The key part of these lines are those red-highlighted.

The requirement was 6 ns and the slack 0.181 ns (note that it's positive), which means that we could have asked for a clock period 0.181 ns shorter and it would still be OK. So it could have been (6–0.181) ns = 5.819 ns, which is 172 MHz.

The answer to what's the maximum frequency is then 172 MHz, but this value could change if the constraints change.

The Data Path Delay tells us something about what made this worst path slow, how much delay went on logic, and how much on the (estimated) route delays. So does the detailed delay report that follows.

Note that this is the Max Delay Paths section, there is also a Min Delay Paths section which is useful for spotting hold time violation, and has no effect on the maximum frequency.

6.3 Power consumption report

The following piece from the file `rpt_power.rpt` reports who needs most power on the chip, differentiate between static and dynamic and estimate whether a heat sink is required for the application or not.

```
1. Summary
----------

+--------------------------+-------+
| Total On-Chip Power (W)  | 0.109 |
| Dynamic (W)              | 0.007 |
| Device Static (W)        | 0.102 |
| Effective TJA (C/W)      | 11.5  |
| Max Ambient (C)          | 83.7  |
| Junction Temperature (C) | 26.3  |
| Confidence Level         | Low   |
| Setting File             | ---   |
| Simulation Activity File | ---   |
| Design Nets Matched      | NA    |
+--------------------------+-------+


1.1 On-Chip Components
----------------------

+----------------+-----------+----------+-----------+-----------------+
| On-Chip        | Power (W) | Used     | Available | Utilization (%) |
+----------------+-----------+----------+-----------+-----------------+
| Clocks         |     0.001 |        3 |       --- |             --- |
| Slice Logic    |    <0.001 |      219 |       --- |             --- |
|   LUT as Logic |    <0.001 |       60 |     17600 |            0.34 |
|   Register     |    <0.001 |       83 |     35200 |            0.24 |
|   CARRY4       |    <0.001 |       11 |      4400 |            0.25 |
|   Others       |     0.000 |       39 |       --- |             --- |
| Signals        |    <0.001 |      167 |       --- |             --- |
| I/O            |     0.004 |       36 |       100 |           36.00 |
| Static Power   |     0.102 |          |           |                 |
| Total          |     0.109 |          |           |                 |
+----------------+-----------+----------+-----------+-----------------+
```

Appendices

A Source code

This is the source code for the top-level component `SAD.vhd`. This is also stored into the folder `./src` together with the other components used. Head over that folder for a mode detailed reference.

```
------------------------------------------------------------------
-- Title       : SAD
-- Project     : Final project: SAD Calculation
------------------------------------------------------------------
-- File        : SAD.vhd
-- Language    : VHDL
-- Author(s)   : Francesco Urbani
-- Company     :
-- Created     : Fri May 18 16:23:33 CEST 2018
------------------------------------------------------------------
-- Description : Actual SAD calculator (top level)
------------------------------------------------------------------
-- Update      :
------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;



entity SAD is
        generic (
                Npixel      :       positive := 16;
                                            -- total # of pixels of
                    the image
                Nbit        :       positive := 8;
                                             -- # bits needed to
                    represent the value of each pixel
                SAD_bits    :       positive := 16
                                             -- # of bits needed to
                    represent the output
        );
        port (
                CLK         : in    std_logic;
                                                  -- CLK, active on
                    rising edge
                RST         : in  std_logic;
                                                    -- RST, active high
                PA          : in  std_logic_vector(Nbit-1 downto
                   0);      -- input pixel value image A
                PB          : in  std_logic_vector(Nbit-1 downto
                   0);      -- input pixel value image B
                EN          : in  std_logic;
                                                    -- enable input

                SAD         : out std_logic_vector(SAD_bits-1
                   downto 0); -- ouput SAD value
                DATA_VALID  : out std_logic
                                                    -- specifies whether
                   the output SAD is valid or not
        );
end entity SAD;


architecture struct of SAD is

        -- DECLARING COMPONENTS NEEDED.

        component PIPOreg is
                generic (N : positive); -- N BITS PIPO REGISTER
                port (
                        clk      : in  std_logic;
                        rst      : in  std_logic;
                        d        : in  std_logic_vector(N-1
                           downto 0);
                        q        : out std_logic_vector(N-1
                           downto 0)
                );
        end component;

        component PhaseAccumulator is
                generic (N : positive );
                port(
                        clk     : in  std_logic;
                        rst     : in  std_logic;
                        pa_in   : in  std_logic_vector(N-1 downto
                            0);
                        pa_out  : out std_logic_vector(N-1 downto
                            0)
                );
        end component;


        component counter is
                generic ( overflow_val : natural );
                port (
                        count_puls    : in std_logic;
                        count_enable  : in std_logic;
                        rst           : in std_logic;
                        tc            : out std_logic
                );
        end component;

        component subtractor is
                generic (Nbit : positive);
                port (
                        a : in  std_logic_vector(Nbit-1 downto
                           0);
                        b : in  std_logic_vector(Nbit-1 downto
                           0);
                        s : out std_logic_vector(Nbit-1 downto
                           0)
                );
        end component;


        component mux2to1 is
                generic (Nbit : in positive );
                port (
                        i0    : in  std_logic_vector(Nbit-1
                           downto 0);
                        i1    : in  std_logic_vector(Nbit-1
                           downto 0);
                        s     : in  std_logic;
                        f     : out std_logic_vector(Nbit-1
                           downto 0)
                ) ;
        end component ;


        -- intermediate signals
        signal padding                : std_logic_vector(SAD_bits
            -Nbit-1 downto 0);
                                      -- turns the signal out off
                                         the subractor to a number
                                         of bits
                                      -- coherent with the
                                         downstream signal

        signal PA_to_sub_nbit         : std_logic_vector(Nbit-1
            downto 0);
                                      -- connection from the out of
                                           the reg on the PA side to
                                           the subtractor
```

```
signal PB_to_sub_nbit          : std_logic_vector(Nbit-1
    downto 0);
                             -- connection from the out of
                                the reg on the PB side to
                                the subtractor

signal sub_out_nbits           : std_logic_vector(Nbit-1
    downto 0);
                             -- signal out of the
                                subtractor (Nbit)

signal pa_in                   : std_logic_vector(SAD_bits
    -1 downto 0);
                             -- input to the phase
                                accumulator

signal pa_out                  : std_logic_vector(SAD_bits
    -1 downto 0);
                             -- phase accumulator output
                                wire

signal mux_to_reg_out_wire     : std_logic_vector(SAD_bits
    -1 downto 0);
                             -- connection from the
                                multiplex to the output
                                PIPO register

signal rst_input_registers     : std_logic;
                             -- reset input signal to the
                                two heading PIPO registers

signal sad_wire                : std_logic_vector(SAD_bits
    -1 downto 0);
                             -- output-register output (i.
                                e. the actual SAD signal)

signal tc_wire                 : std_logic; -- output of
    the counter
signal hold_wire               : std_logic; -- input to
    the MUX control signal.
                                            -- it also
                                               coincides
                                               with
                                               DATA_VALID

constant counter_of_value      : positive := Npixel+3;
                                 -- counter overflow value
                                    , "+3" takes into
```

```
                                                    account
                                                 -- the delay caused by
                                                    the upper chain , i.e.
                                                 -- to get from PA to SAD.


begin

        rst_input_registers <= EN nand (not RST);
        hold_wire           <= EN nand (not tc_wire);


        reg_PA  : PIPOreg
                generic map(Nbit)
                port map (CLK, rst_input_registers , PA,
                    PA_to_sub_nbit);

        reg_PB  : PIPOreg
                generic map(Nbit)
                port map (CLK, rst_input_registers , PB,
                    PB_to_sub_nbit);


        -- generating some zeros to make the padding.
        gen_padding : for i in 0 to SAD_bits-Nbit-1 generate
                padding(i) <= '0';
        end generate;


        -- merging padding and sub_out_nbits to make a new
            signal called pa_in
        -- which enters the phase accumlator
        pa_in <= padding & sub_out_nbits;

        -- subtractor instance
        sub: subtractor
                generic map(Nbit)
                port map(PA_to_sub_nbit , PB_to_sub_nbit ,
                    sub_out_nbits);

        phaseacc: PhaseAccumulator
                generic map(SAD_bits)
                port map(CLK, RST, pa_in , pa_out);


        -- multiplexer on the output register side.
        -- Needed to freeze the SAD signal when ENABLE is 0 and
            when the SAD computation is completed.
        mux1 : mux2to1
```

```
                generic map(SAD_bits)
                port map(pa_out, sad_wire, hold_wire,
                    mux_to_reg_out_wire);

        reg_out: PIPOreg
                generic map(SAD_bits)
                port map (CLK, RST, mux_to_reg_out_wire,
                    sad_wire);


        -- counter instance
        -- it counts up to counter_of_value, after that it sets
            its output to 1,
        -- hence the DATA_VALID signal of the system is high and
             the SAD value is frozen.
        cnt: counter
                generic map(counter_of_value)
                port map(CLK, EN, RST, tc_wire);


        -- connection of the intermediate signals to the output
            of the SAD.
        SAD        <= sad_wire;
        DATA_VALID <= tc_wire;


end architecture;
```