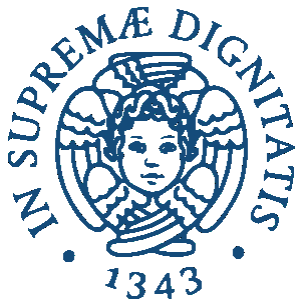


Università di Pisa

Master Of Science in Computer Engineering

Electronics and Communication Systems

Calculation of f_0 coefficient

Francesco De Lucchini

Academic Year 2024/2025

Contents

1 Introduction

1.1 Problem description

In an image elaboration system, it is necessary to calculate the f_0 coefficient, defined as:

$$f_0 = \alpha \cdot y(i-1, j) + (1-\alpha) \cdot y(i, j)$$

where:

- $y(i, j) \in [0, 255]$ are the pixels of an image, represented by the matrix y, which is stored in a ROM
- $\alpha \in (0, 1)$ is a parameter chosen by the user

It is required to design a digital circuit that implements such operation under the following constraints:

- α is represented in 7.3 fixed-point notation
- Only one pixel per cycle can be read from the ROM

The interface of the circuit to design is the one illustrated in Figure 1.

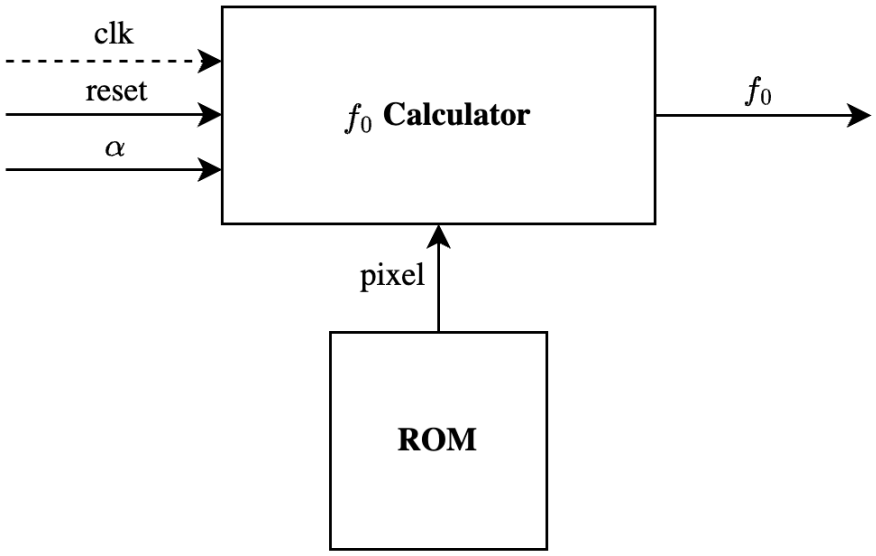


Figure 1: Interface of the circuit to design

1.2 Possible applications

It is obvious that **the circuit does some kind of digital image processing**. In particular, one can observe that, since $\alpha \in (0, 1)$ and $y(i, j) \in [0, 255]$, then, $f_0 \in [0, 255]$, therefore, after rounding or truncating the fractional part, every f_0 coefficient can be interpreted as a pixel.

Moreover, since the circuit described above basically outputs an f_0 coefficient for every pixel in the ROM, then **the output of the circuit can also be interpreted as an image**.

Specifically, since every f_0 coefficient is the α-weighted average of two vertically adjacent pixels of the input image, then **the output image will appear smoother**, i.e., the (vertical) transitions from pixels with high intensities to pixels with low intensities will be less noticeable.

1.3 Possible architectures

Before starting to implement the circuit, three main issues (which all lead to slightly different architectures) must be addressed.

The first issue that has to be addressed is **what to do when computing the f_0 coefficients for the first row of pixels** (i.e., when $i = 0$). This is because f_0 depends on $y(i-1, j)$, which, in the former case, is not clearly defined. Possible solutions are:

- **Assume a constant value**: $y(-1, j) = k \; \forall \, j$, where k can be any integer $\in [0, 255]$ (a sensible choice could be 0, 127 or 255)
- **Repeat**: $y(-1, j) = y(0, j) \; \forall \, j$
- **Wrap around**: $y(-1, j) = y(N-1, j) \; \forall \, j$, where N is the number of rows of the image
- **Ignore**: simply start computing the coefficients from the second row

Ignoring the first row is probably the "safest" option, nevertheless, since, for reasons that will be clear later on, it leads to an easier implementation, $y(-1, j)$ will be assumed to be $0 \; \forall \, j$.

The second issue to tackle is **how to actually compute f_0**. Since the calculations are fairly easy (two multiplications and a sum), it is best to just use **combinational logic** (standard multipliers and adders). If the calculations were more complex, a **look-up-table** might have been considered.

The third and final issue regards the fact that **each f_0 coefficient requires two different pixels, but the ROM only allows a single one to be fetched each clock cycle**. This can be addressed in two ways:

- Make a **slow, but resource efficient, circuit**, which takes two clock cycles to compute each f_0 coefficient, but uses little-to-no registers
- Make a **fast, but resource hungry, circuit**, which takes one clock cycle to compute each f_0 coefficient, but requires to cache the last M pixels read from the ROM, where M is the number of columns of the image

Since there are no strict constraints on memory utilization, the second option is chosen.

2 Implementation

The digital circuit, implemented in **VHDL**, is illustrated via the block diagram in Figure 2. In particular:

- The numbers between square brackets are the numbers of bits needed to represent the signals. When not present, the signals are only one bit big. Note that, for simplicity, these numbers are set to what was written in the specifications. In the code, everything is implemented using `GENERICS`, allowing for maximum flexibility
- The rectangles with a bar in the middle are D-Flip-Flops (DFFs)
- The little blocks with ones/zeroes inside mean "constant logic value of one/zero". As as consequence, in this case, the output registers are always enabled
- The `_n` suffix means that the signal is active low (only used for the reset signal)

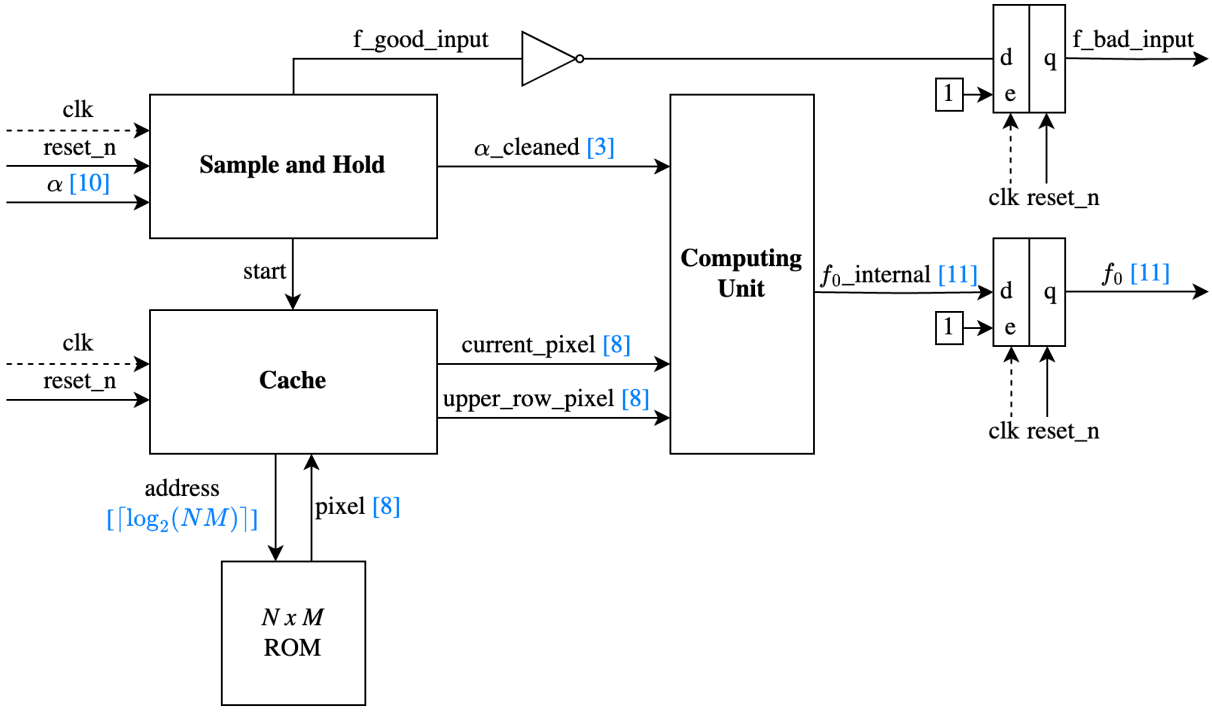


Figure 2: Block diagram of the implemented circuit

The first thing to observe is that α is given using 7.3 fixed-point notation, however, since $\alpha \in (0, 1)$, **the 7 bits for the integer part are not really necessary**. That's why the α given as an input to the *Computing Unit* is represented using only 3 bits.

Moreover, as observed in Section 1.2, given that $\alpha \in (0, 1)$ and $y(i, j) \in [0, 255]$, then $f_0 \in [0, 255]$. Since α is represented using 3 bits for the fractional part, then **the output** f_0

can be represented in 8.3 fixed-point notation (i.e., 11 bits total).

Furthermore, note that **the image inside the ROM does not have to be a square matrix**. In Figure 2, the number of rows and columns of the image are called N and M, respectively.

Finally, with respect to the circuit interface shown in Figure 1, note the addition of the output signal `address`, needed to index the memory cells in the ROM, and the output signal `f_bad_input`, needed to signal to the user when the f_0 coefficients start being valid.

2.1 Basic components

Before introducing the three main blocks of the circuit (*Sample and Hold*, *Cache* and *Computing Unit*), the basic components used to construct them are briefly presented.

2.1.1 Full adder

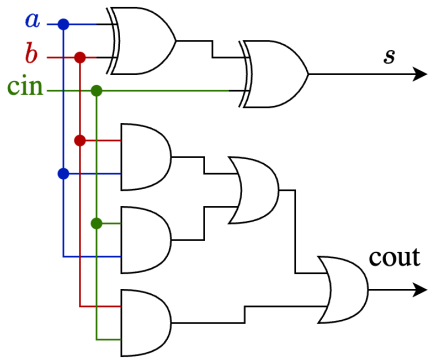


Figure 3: Circuit diagram of a generic *Full Adder*

2.1.2 Ripple Carry Adder

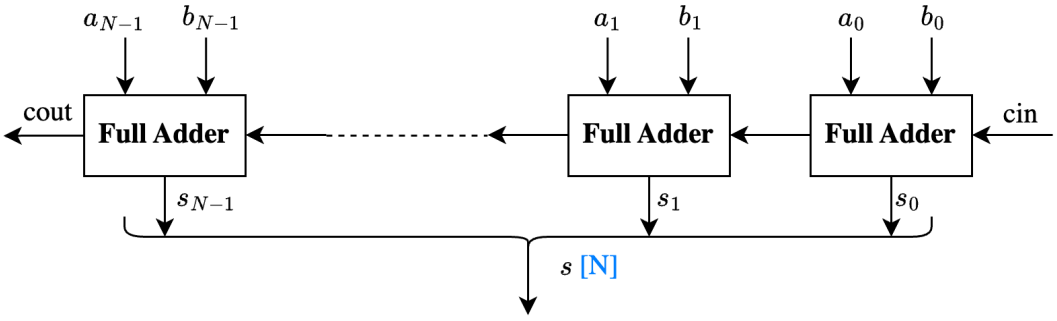


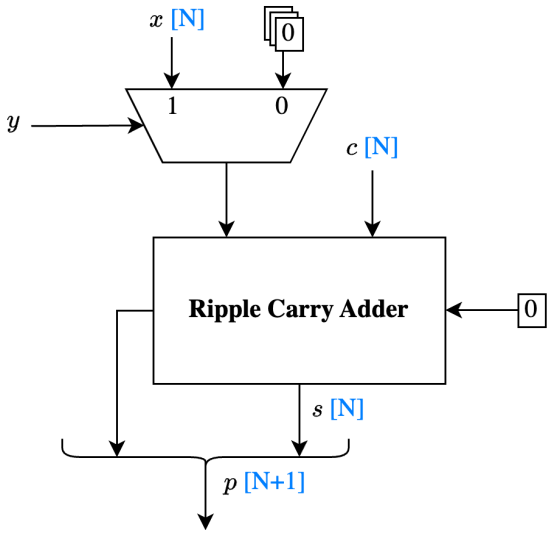
Figure 4: Block diagram of a generic *Ripple Carry Adder*

2.1.3 Nx1 Multiplier

Performs $p = x \cdot y + c$, where:

- x is a N bit natural
- y is a 1 bit digit
- c is a N bit natural
- p, consequently, is a $N + 1$ bit natural

Basically, if $y = 0$ then $p = c$, else $p = x + c$.

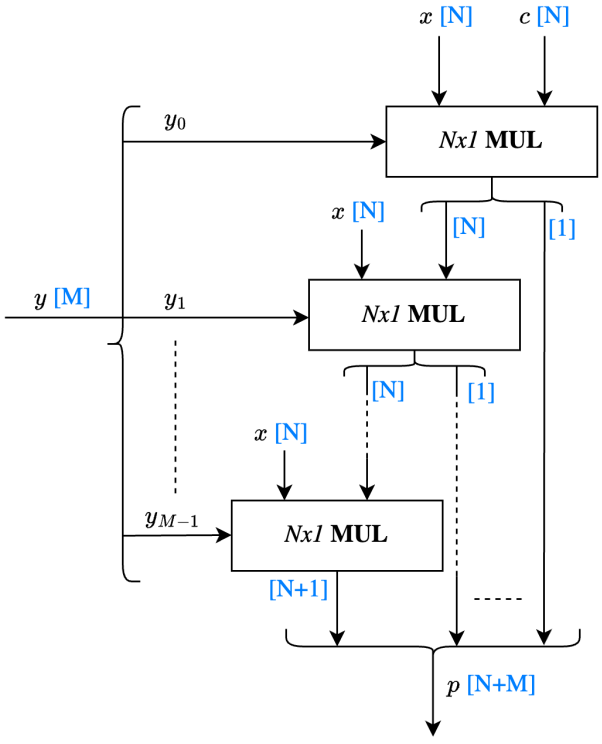


2.1.4 NxM Multiplier

Performs $p = x \cdot y + c$, where:

- x is a N bit natural
- y is a M bit natural
- c is a N bit natural
- p, consequently, is a $N + M$ bit natural

This is achieved by employing a cascade of M Nx1 multipliers.



2.1.5 D-Flip-Flop

A N-bit async-reset-active-low D-flip-flop with enabler, composed by a parallel of N async-reset-active-low D-Flip-Flops.

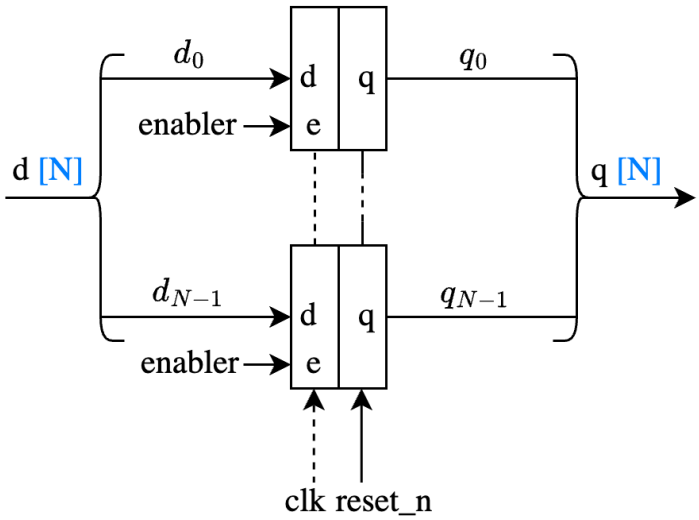


Figure 5: Circuit diagram of a generic *D-Flip-Flop*

2.1.6 Shift Register

An M-stage shift register is composed by a cascade of M D-Flip-Flops, where M is the number of rising edges of the clock after which the input d appears at the output q.

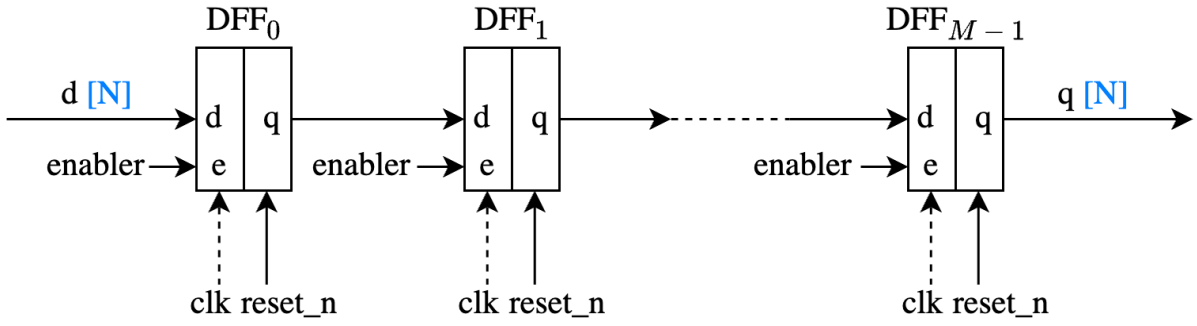


Figure 6: Block diagram of a generic *Shift Register*

2.1.7 Counter

If enabled (and not resetted), at each clock cycle performs `count += increment`. If the sum overflows, the counter simply starts again from 0.

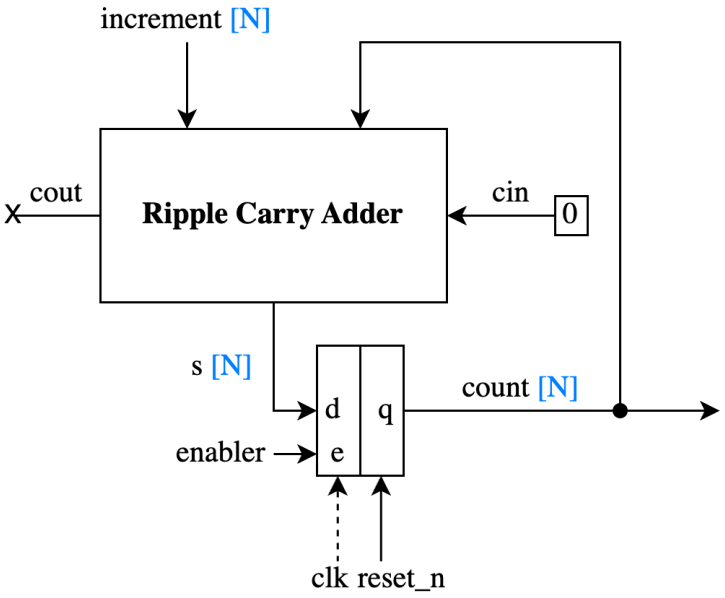


Figure 7: Block diagram of a generic *Counter*

2.2 Sample and Hold

As soon as the reset phase is over, and α follows the specifications, α is sampled and held in a register until the next reset phase. This way, **the circuit is sensible to the variations of α only before the computations start**.

Indeed, the `start` output signal is used by the *Cache* to know when to start fetching pixels, while **the `f_good_input` signal is used by the user to know when the f_0 coefficients start being valid**.

Another thing to notice is how the circuit checks that α follows the specifications (recall that it must be between 0 and 1, extremes excluded). This is achieved by checking that:

1. The integer part of α is composed by all `0`s (7-input NOR gate)
2. The fractional part of α is composed by at least one `1` (3-input OR gate)

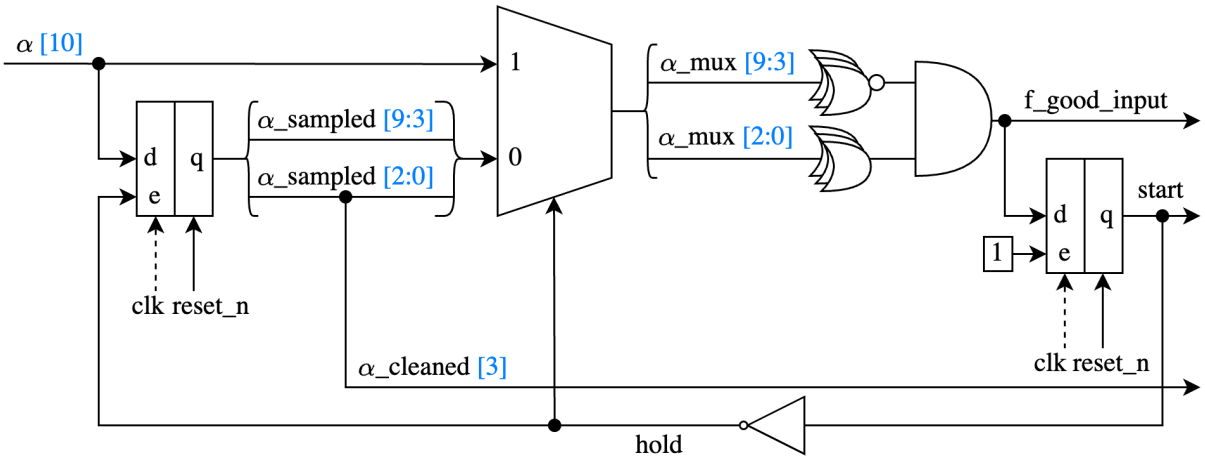


Figure 8: Block diagram of the *Sample and Hold*

2.3 Cache

As discussed in 1.3, in order to compute every f_0 coefficient in one clock cycle, the last M pixels used must be cached. This is achieved by using an **M-stage shift register**. This way, the current pixel is the one currently coming from ROM, and the upper row pixel is just the output of the shift register.

The address of the current pixel is simply the output of a counter with `1`-bit increments. The counter starts when α follows the specifications and stops when the maximum addressable pixel ($NM - 1$) is reached (N and M are the number of rows and columns of the image in the ROM, respectively).

Finally, recall that, in Section 1.3, it was stated that: "*Since, for reasons that will be clear later on, it leads to an easier implementation, $y(-1, j)$ will be assumed to be $0\ \forall\, j$*". This is because the flip-flops inside of the shift registers all contain zeroes by default, hence, assuming $y(-1, j) = 0\,\forall\, j$ does not require to make any extra changes to the shift register.

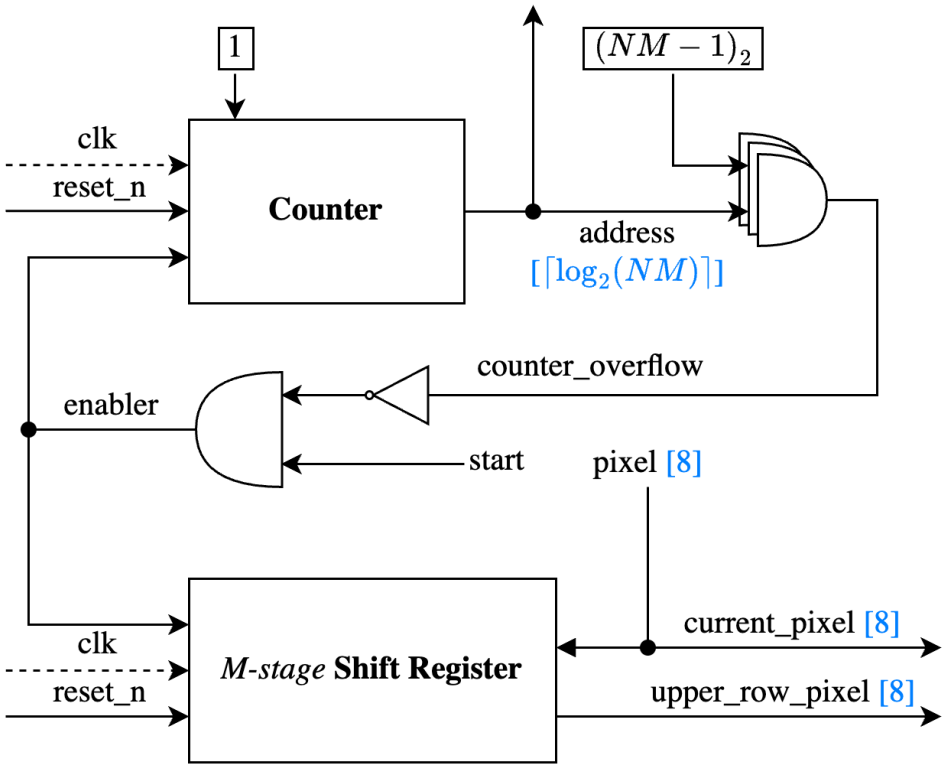


Figure 9: Circuit diagram of the *Cache*

2.4 Computing Unit

The *Computing Unit* is a **pure combinational logic network** that, given α, the current pixel $y(i,j)$, and the pixel on the upper row $y(i-1,j)$, computes $f_0 = \alpha \cdot y(i-1,j) + (1-\alpha) \cdot y(i,j)$.

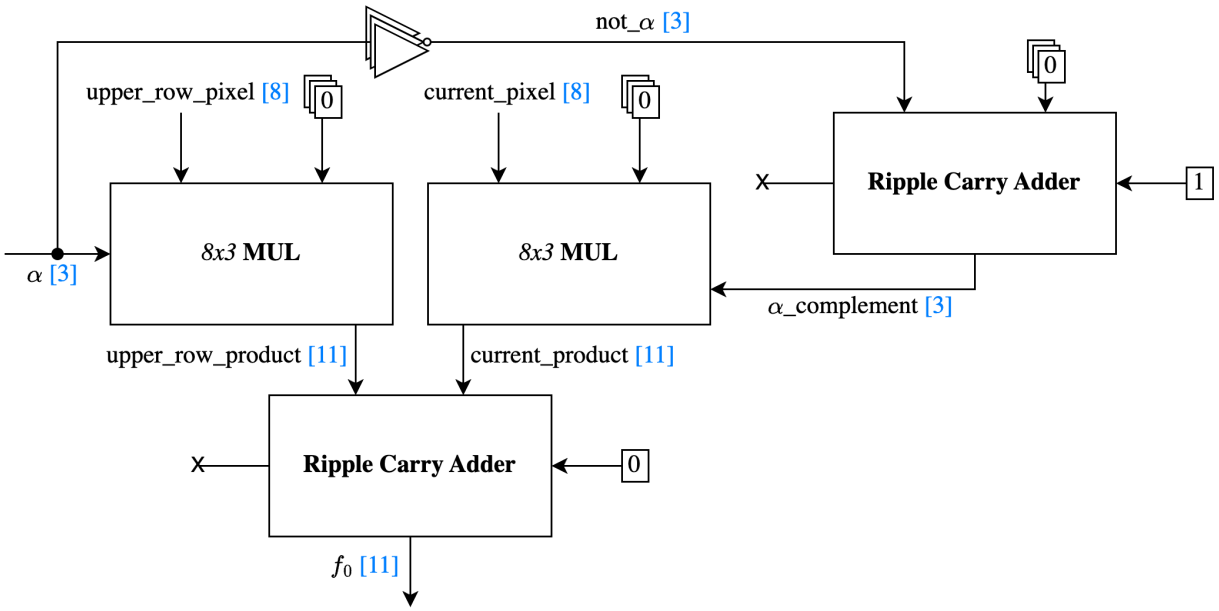


Figure 10: Block diagram of the *Computing Unit*

The only thing to notice is that $1-\alpha$, called `alpha_complement`, can simply be computed as $(\texttt{not}\ \alpha) + 1$. For example, using 3 bits to represent the fractional part of α, you get the following table, which proves that the desired output, $(1-\alpha)_{10}$, is indeed equal to $((\texttt{not}\ \alpha)+1)_{10}$:

$(\alpha)_{10}$	$(\alpha)_2$	$(\texttt{not}\ \alpha)_2$	$((\texttt{not}\ \alpha)+1)_2$	$((\texttt{not}\ \alpha)+1)_{10}$	$(1-\alpha)_{10}$
0.125	001	110	111	0.875	0.875
0.250	010	101	110	0.750	0.750
0.375	011	100	101	0.625	0.625
0.500	100	011	100	0.500	0.500
0.625	101	010	011	0.325	0.325
0.750	110	001	010	0.250	0.250
0.875	111	000	001	0.125	0.125

3 Verification

The verification of the components is carried out by simulating them using `ModelSim-Intel©` `FPGAs Standard Edition 2020.1` and analyzing the waveforms of their input and output ports.

3.1 Verification of the basic components

- **Ripple Carry Adder**: Since the circuit is particularly simple, it is feasible to test it exhaustively by using every possible combination of the input signals. In particular, the test shown in Figure 11 is relative to a 2-bit ripple carry adder. From the waveform, it is clear that indeed $s =\mid a + b + \text{cin} \mid_{2^2}$ and $\text{cout} = 1$ when $a + b + \text{cin} > 2^2 - 1$, therefore, the component behaves as expected

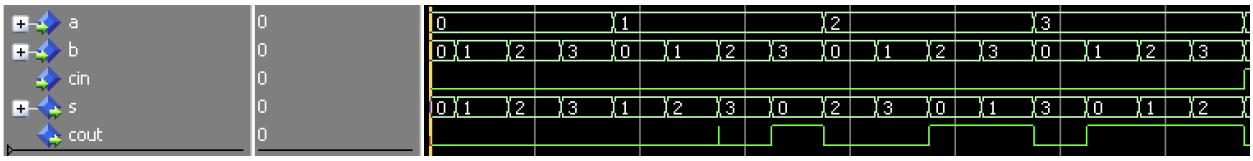


Figure 11: Verification of the *Ripple Carry Adder*

- **Multiplier**: Again, the circuit is particularly simple, hence, it can be verified exhaustively by using every possible combination of the input signals. Specifically, the test shown in Figure 12 is relative to a 2x2 multiplier. By looking at the waveform, it is clear that indeed $p = x \cdot y + c$, therefore, the component behaves as expected

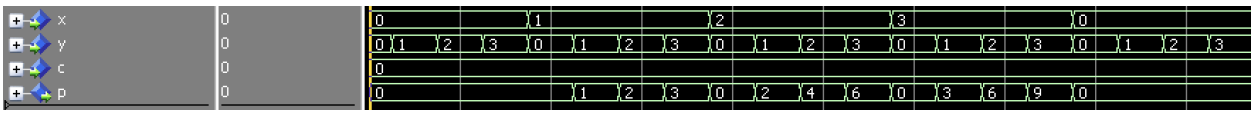


Figure 12: Verification of the *Multiplier*

- **D-Flip-Flop**: When the flip-flop is enabled and the reset signal is high, the output q samples the input d at each rising edge of the clock. Moreover, whenever the reset signal is low, the output immediately goes to 0. Finally, the output is insensible to variations of the input when the enabler signal is low. Consequently, since the component is an asynchronous reset-active-low flip-flop, it behaves as expected

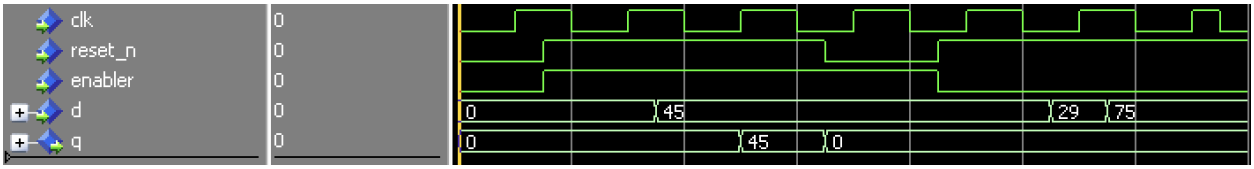


Figure 13: Verification of the *D-Flip-Flop*

- **Shift Register**: When the register is enabled and the reset signal is high, the output q should repeat the input d that was present $M-1$ rising edges of the clock in the past. In particular, the test shown in Figure 14 is relative to a 4-stage shift register. From the waveform, it is clear that indeed the output q repeats the same exact sequence of inputs shifted by 3 clock periods, therefore, the component behaves as expected

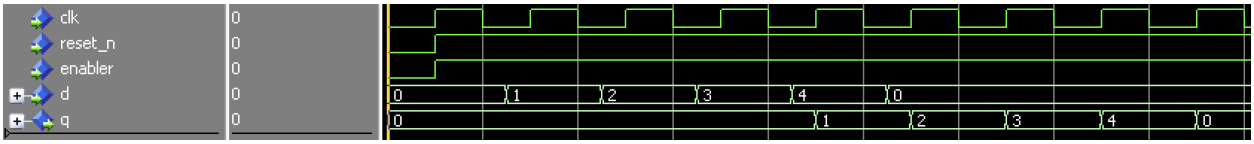


Figure 14: Verification of the *Shift Register*

- **Counter**: When the counter is enabled and the reset signal is high, at each rising edge of the clock the output of the circuit should increment by `increment`. From the waveform, it can be observed that the counter behaves exactly as stated before. Moreover, when the reset signal is low, the output immediately goes to 0. Therefore, the component behaves as expected

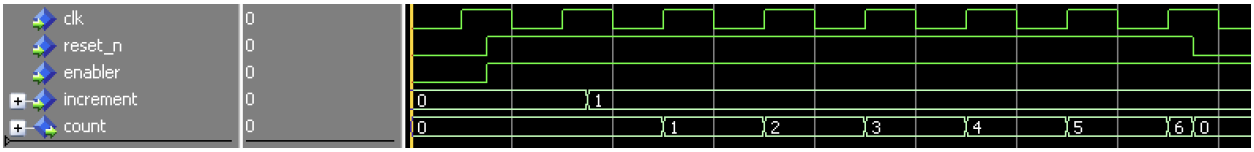


Figure 15: Verification of the *Counter*

3.2 Verification of the Sample and Hold

If α changes while the reset signal is still low, it should have no effect. When the reset phase is over, if α is out of specifications (in this case, $= 0$ or ≥ 8), the bad input flag should stay up. As long as the `start` signal is down, at each rising edge of the clock `alpha_cleaned` should sample the 3 lowest bits of α. At the first rising edge of the clock that α meets the specifications, the `start` signal should go up, when it does, the circuit should not be sensible anymore to variations of α. At the reset, the output signals should go back to their default values (i.e., `start = 0`, `f_bad_input = 1`, `alpha_cleaned = 0`). From the waveform, it is clear that the module behaves exactly as described before.

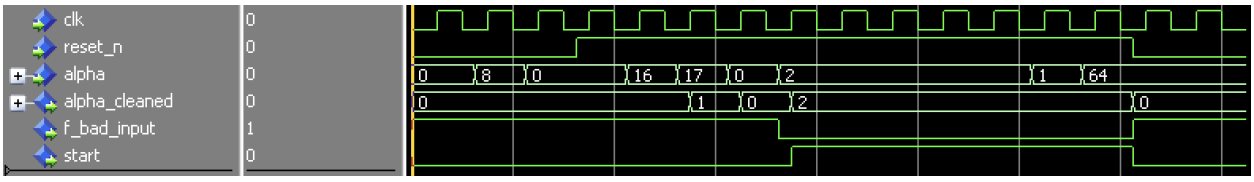


Figure 16: Verification of the *Sample And Hold*

3.3 Verification of the Cache

Consider the 2x3 test ROM $y = \{212, 170, 127; 85, 42, 0\}$. Excluding the first pixel, which is fetched by default because the reset value of the `address` signal is 0, the cache should only start fetching pixels the first rising edge of the clock after which the `start` signal goes high, and should stop fetching them after reaching the highest addressable one (5, in this case).

Moreover, the `current_pixel` output should always mirror the `pixel` input. Instead, the `upper_row_pixel` signal should be 0 until the `address` signal is equal to or lower than $M - 1$, then, it should always be equal to the pixel at address: `address` $- M$ (in this case, $M = 3$).

Furthermore, resetting the cache should overwrite the `address` and the `upper_row_pixel` signals with zeroes. Consequently, the `current_pixel` signal should get the value of the first cell of the ROM.

By looking at the waveform, the module appears to behave as expected.

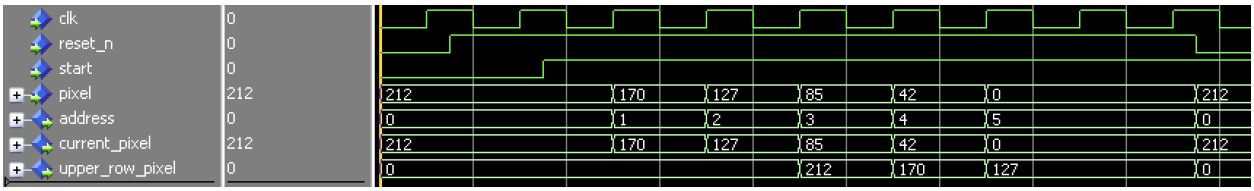


Figure 17: Verification of the *Cache*

3.4 Verification of the Computing Unit

To verify the behavior of the computing unit, the outputs of the circuit are compared with the ones manually computed in the table below:

`current_pixel`	$1 - \alpha$	`upper_row_pixel`	α	f_0	$f_0 \cdot 2^3$
200	0.750	100	0.250	175.00	1400
200	0.500	100	0.500	150.00	1200
0	0.750	255	0.250	63.75	510
100	0.750	100	0.250	100.00	800
255	0.125	255	0.875	255.00	2040
0	0.750	0	0.250	0.00	0
179	0.675	51	0.325	128.45	1048

By looking at the waveform in Figure 18, it is clear that the module behaves as expected.

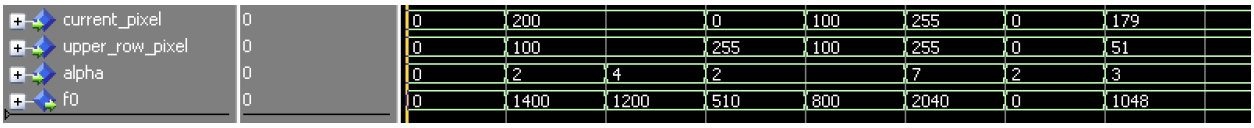


Figure 18: Verification of the *Computing Unit*

3.5 Verification of the main circuit

As long as either the reset signal or the `f_bad_input` signal is low, the circuit should be insensible to the variations of α. When the reset phase ends, if α does not meet the specifications, at the next rising edge of the clock the bad input flag should go up. Moreover, at the very next rising edge of the clock after α meets the specifications, the bad input flag should go down. This signals the user that, starting from the successive rising edge of the clock, the f_0 coefficients will be valid. With respect to these observations, by looking at the waveform shown in Figure 19, the circuit behaves as expected.

Considering the same 2x3 test ROM introduced in Section 3.3 $y = \{212, 170, 127; 85, 42, 0\}$, from the waveform shown in Figure 19 it is also clear that the cache correctly fetches the pixels from the ROM. Furthermore, when the reset signal gets low, the system immediately resets, as expected.

Finally, the values of the f_0 coefficients and the cached pixels must also be correct, indeed:

`current_pixel`	$1 - \alpha$	`upper_row_pixel`	α	f_0	$f_0 \cdot 2^3$
212	0.750	0	0.250	159.00	1272
170	0.750	0	0.250	127.5	1020
127	0.750	0	0.250	95.25	762
85	0.750	212	0.250	116.75	934
42	0.750	170	0.250	74.00	592
0	0.750	127	0.250	31.75	254

Therefore, the circuit seems to work as expected.

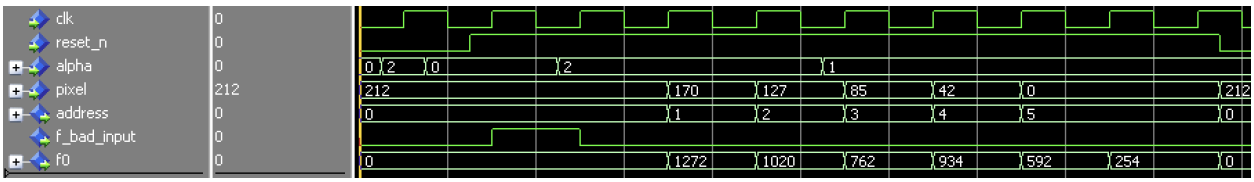


Figure 19: Verification of the main circuit

4 Interpretation of the output

As discussed in Section 1.2, $f_0 \in [0, 255]$, therefore, it can be interpreted as a pixel, and, consequently, the collection of all the f_0 coefficients produced by the circuit can be interpreted as an image. It may be interesting to **actually visualize the effect that the circuit has on various input images**. This can be achieved by simulating the circuit, just like it was done in Section 3, exporting the waveform of the f_0 output port, and, with a scripting language (i.e., python), parsing the pixels and constructing the output image.

4.1 Grayscale

Since the pixels are represented using 8 bits, the easiest thing to do is to interpret them as different intensities of gray, 0 being completely black and 255 being completely white.

Consider a 8x32 ROM containing all the possible 8-bit pixels (i.e., $y = \{0, 1, ...255\}$), interpreted in Figure 20 as a grayscale image (the black border is, of course, not part of the image).

Figure 20: 8x32 ROM containing all the possible 8-bit pixels, interpreted as a grayscale image

Then, the output of the circuit is the following:

Figure 21: Output of the circuit when the input is the ROM in Figure 20 and $\alpha = 0.125$

Figure 22: Output of the circuit when the input is the ROM in Figure 20 and $\alpha = 0.875$

Recall that each pixel of the output image is composed by $(1 - \alpha)\,\%$ of the original pixel and $\alpha\,\%$ of the pixel on the upper row (0 if it is the first row, which explains the weird behavior in the first row of the output images). Therefore, when α is low (Figure 21), the image is more similar to the original one, instead, when α is high (Figure 22), the pixel on the upper row is weighted more, hence, the output image looks like a 1-row shifted version of the input one.

As another example, consider a 128x128 ROM constructed by using a grayscale version of the logo of the University Of Pisa, shown in Figure 23. As expected, with $\alpha = 0.500$, the output image, shown in Figure 24, looks like a blurred version of the input one.

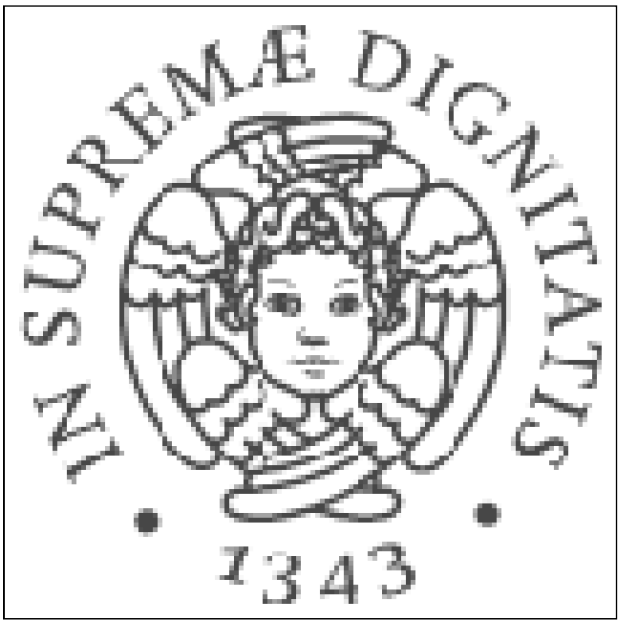


Figure 23: 128x128 grayscale version of the logo of the University Of Pisa

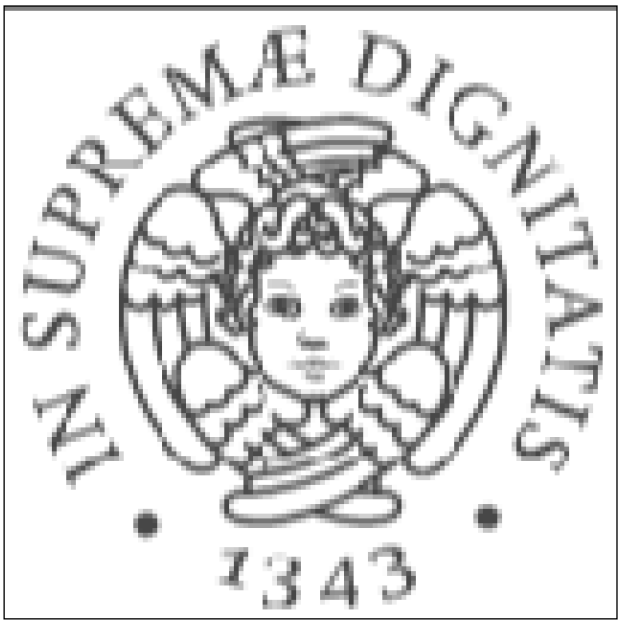


Figure 24: Output of the circuit when the input is the ROM in Figure 23 and $\alpha = 0.500$

4.2 3-3-2 bitmap

Another possible way of interpreting 8-bit pixels, is by using the so called 3-3-2 bitmap, i.e., 3 bits for `RED`, 3 bits for `GREEN` and 2 bits for `BLUE` (`RRRGGGBB`).

By considering all the possible combinations of 8 bits, the following palette is obtained:

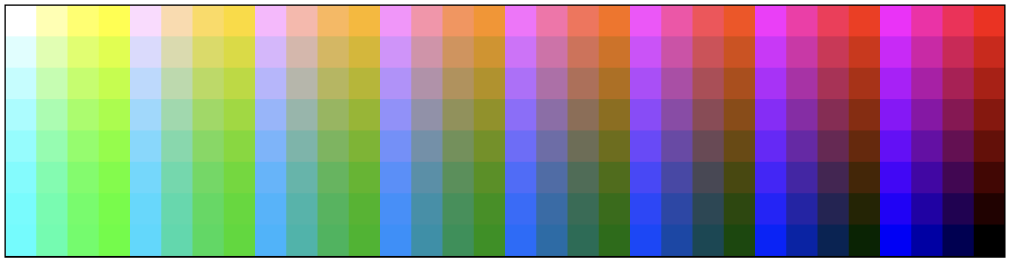


Figure 25: Complete 3-3-2 bitmap palette

As an example, applying this palette to the image in Figure 26 yields the results shown in Figure 27 (note the artifacts in the background).

Figure 26: Original, 24 bit-per-pixel (8 per channel), RGB image

Figure 27: 3-3-2 bitmap version of the image in Figure 26

Just for fun, if the image in Figure 27 is fed to the circuit, the output is the one shown in Figure 28. The output pixels are not clearly predictable, this is because the circuit does not blend each pixel in a channel-wise way, i.e., it does not blend the three bits of the `RED` channel of the current pixel with the three bits of the `RED` channel of the pixel on the upper row, and so on, but considers the pixels as a whole.

Figure 28: Output of the circuit when the input is the ROM in Figure 27 and $\alpha = 0.500$

5 Synthesis

The synthesis of the circuit was performed using `Vivado 2024.1`, targeting the `FPGA` module of a `Zynq-7000` development board (in particular, the `xc7z010clg400-1`). Since Vivado only evaluates *Register-to-Register* paths, the inputs of the circuit were wrapped with registers (the outputs are already held by internal registers). The resulting circuit is the one shown in Figure 29.

Furthermore, the synthesis was run using a 16x16 ROM, 8 bits-per-pixel (bpp) and 7.3 fixed point notation for α, with the constraint of an 125 MHz clock with 50 % duty cycle (that's the clock available by default on the FPGA side of the board) and using `out-of-context` mode (i.e., the `I/O` ports of the circuit are not mapped to the physical pins of the board).

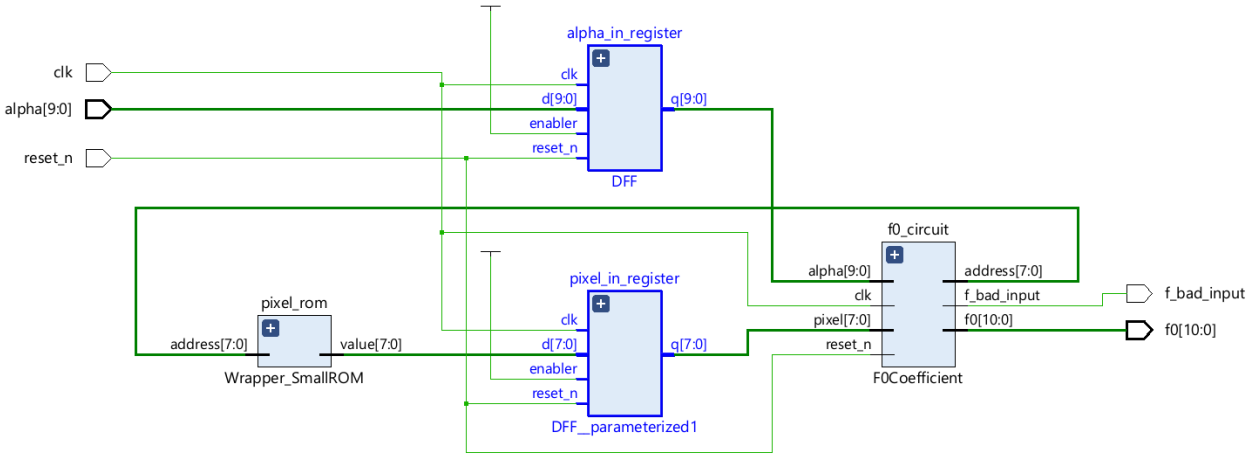


Figure 29: Schematic of the circuit wrapper. The highlighted blocks are the input registers

Before illustrating the obtained results, the warning messages generated by Vivado are briefly explained:

- Synthesis: `[Synth 8-7080] Parallel synthesis criteria is not met` - This warning indicates that the design is not large enough to benefit from parallel synthesis, therefore, it can be safely ignored
- Implementation: a bunch of warnings, all due to the `out-of-context` mode, basically, the timings are not perfectly accurate because the paths from and to the `I/O` pins of the board are not considered
- DRC Violations: `The PS7 cell must be used in this Zynq design in order to enable correct default configuration` - That cell is used by the board's ARM processor to fetch configuration settings. Since, in this project, only the programmable logic (FPGA) part of the board is used, this warning can be safely ignored too

The results of the synthesis (and implementation), are illustrated in the following sections.

5.1 Timings

With respect to the timings reported in Figure 30, the minimum allowed clock period is $T_{min} = T_{clk} - WNS = 8 - 1.012 = 6.988$ ns, consequently, the maximum operating frequency is $f_{max} = 1/T_{min} = 143.102$ MHz, just slightly above the default one (125 MHz)

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	1,012 ns	Worst Hold Slack (WHS):	0,167 ns	Worst Pulse Width Slack (WPWS):	3,020 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	150	Total Number of Endpoints:	150	Total Number of Endpoints:	96

All user specified timing constraints are met.

Figure 30: Timings obtained with a 16x16 ROM, 8 bpp and 7.3 fixed point α

Moreover, since **the critical path always goes through the computing unit**, one can observe that:

- **A bigger ROM does not lead to worse timings**, as shown in Figure 31

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	1,145 ns	Worst Hold Slack (WHS):	0,141 ns	Worst Pulse Width Slack (WPWS):	3,020 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	550	Total Number of Endpoints:	550	Total Number of Endpoints:	296

All user specified timing constraints are met.

Figure 31: Timings obtained with a 128x128 ROM, 8 bpp and 7.3 fixed point α

- **The higher is the number of bits-per-pixel**, the bigger are the adders in the computing unit, hence, the more is the waiting time due to the propagation of the carries, and **the lower is the worst negative slack**. Probably, implementing a carry-lookahead adder instead of a ripple-carry one would lead to better performances. As a proof for this reasoning, with a lower number of bpp, for example, 4 instead of 8, the worst negative slack does increase, as shown in Figure 32

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	3,546 ns	Worst Hold Slack (WHS):	0,163 ns	Worst Pulse Width Slack (WPWS):	3,020 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	110	Total Number of Endpoints:	110	Total Number of Endpoints:	72

All user specified timing constraints are met.

Figure 32: Timings obtained with a 16x16 ROM, 4 bpp and 7.3 fixed point α

- **The higher is the precision of** α, the bigger are the multipliers, hence, the more are the levels of logic to go through, and the **lower is the worst negative slack**. For example, representing α using 7.6 fixed point notation leads to the timings shown in Figure 33. If this becomes an issue, different multiplier architectures should be considered, perhaps even one purely based on look-up-tables (pre-compute a table of $\alpha \cdot$`pixel` products)

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-0,601 ns	Worst Hold Slack (WHS):	0,143 ns	Worst Pulse Width Slack (WPWS):	3,020 ns
Total Negative Slack (TNS):	-1,296 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	3	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	168	Total Number of Endpoints:	168	Total Number of Endpoints:	110

Timing constraints are not met.

Figure 33: Timings obtained with a 16x16 ROM, 8 bpp and 7.6 fixed point α

5.2 Power consumption

Since the design is simple, it makes sense that, as shown in Figure 34, the power is mostly static (i.e., due to the leakage of the transistors)

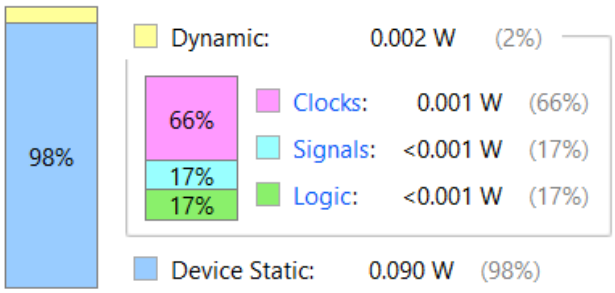


Figure 34: Power consumption obtained with a 16x16 ROM, 8 bpp and 7.3 fixed point α

5.3 Resource utilization

First of all, by looking at Figure 35, one can observe that, other than standard `LUT`s and `FF`s, Vivado uses the so called `LUTRAM`s. In other words, it uses the function generators

(LUTs) of SLICEMs (a particular type of slice) as small memory elements, which are also called "**distributed RAM**". This is due to the presence of the shift register in the cache, as implementing it using LUTRAMs is more efficient than just using flip-flops (it would require more of them, and the LUTs in the same slice of the FFs would not be used).

Resource	Utilization	Available	Utilization %
LUT	93	17600	0.53
LUTRAM	8	6000	0.13
FF	88	35200	0.25

Figure 35: Resource utilization obtained with a 16x16 ROM, 8 bpp and 7.3 fixed point α

When using the same 16x16 ROM, interpreted as a 256x1 image, the shift register becomes a single register, and the LUTRAMs are no longer used, as shown in Figure 36.

Resource	Utilization	Available	Utilization %
LUT	80	17600	0.45
FF	65	35200	0.18

Figure 36: Resource utilization obtained with a 256x1 ROM, 8 bpp and 7.3 fixed point α

In this simple case, one can also check that the number of flip-flops used by Vivado corresponds to the ones instantiated in the VHDL code:

- 10 FFs for the input register of α
- 8 FFs for the input register of the `pixel` signal
- 11 FFs for the output register of f_0
- 1 FF for the output register of the `bad_input_flag` signal
- 1 FF, internal to the Sample and Hold module, for the `start` signal
- 10 FFs, internal to the Sample and Hold module, needed to hold α
- 8 FFs, internal to the cache, for the cached pixels (since there only is one column, the pixel on the upper row is always the last pixel, hence only one register is needed)
- 8 FFs, internal to the cache, used by the counter to keep track of the current address

For a total of 57 flip-flops ... which is different from what Vivado reported (65). This is due to an optimization called "**register replication**": when needed, Vivado can replicate a register to better route the signals on the physical board. In this case, the replicated

register is the one inside the counter of the cache module (8 FFs, hence, a total of $57+8=65$).

This optimization kind of makes sense because the register, as shown in Figure 37, is used by both the internal logic of the counter and by the ROM (as the address input), which may have been placed physically "far" from the counter.

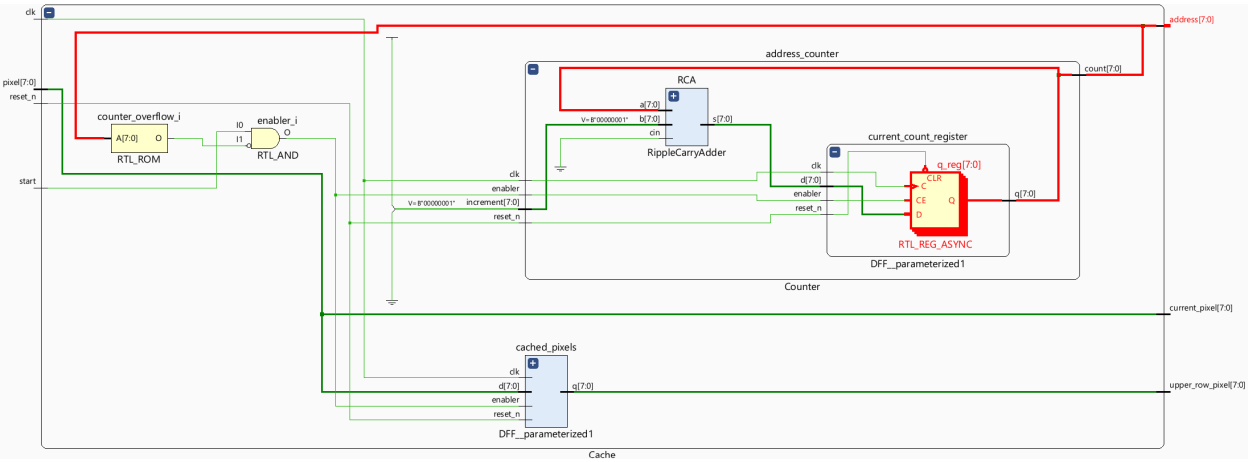


Figure 37: Outgoing paths from the replicated register

As a proof for this, if, instead, a register is manually added before the address output of the cache, Vivado no longer replicates the one inside the counter, and the total number of FFs remains 65.

6 Conclusions

In this project, I implemented a digital circuit capable of performing a fundamental image-processing task: combining adjacent pixels with different weights.

The work began with an exploration of potential applications and architectural options for the circuit. Then, the circuit was designed and implemented using VHDL, employing a bottom-up approach, i.e., creating basic building blocks and progressively integrating them to construct a more complex circuit.

Once the design was completed, the circuit was thoughtfully validated by simulating and analyzing the waveform of the input and output ports of every component. Moreover, both to check its correctness and to gain a deeper understanding of its behavior, the circuit was also tested using sample images.

Once validated, the circuit was synthesized using Vivado, and its timing, power consumption, and resource utilization statistics were analyzed, also considering slightly different variations of the circuit's parameters (e.g., ROM size, α precision, number of bits per pixel, ...).

Generally speaking, the critical path consistently runs through the computing unit, hence:

- Increasing the size of the ROM does not lead to worse timings
- Implementing a carry-lookahead adder would improve the timings, epecially in circuits with a higher number of bits per pixel
- If greater precision for the α parameter is required, alternative multiplier architectures should be considered. This could include designs based on look-up tables that store precomputed $\alpha \cdot$`pixel` products

Finally, in its current state, the circuit's performance is limited by the complexity of the computing unit. As a result, even a faster ROM would not lead to a significant improvements in the overall performance, as the circuit can not operate at a much higher frequency than the one it is currently using. Instead, if better performances are required, one should focus on improving the design of the computing unit (in primis, by employing carry look-ahead adders), or, if possible, one could split the image across multiple ROMs, parallelizing the operations.