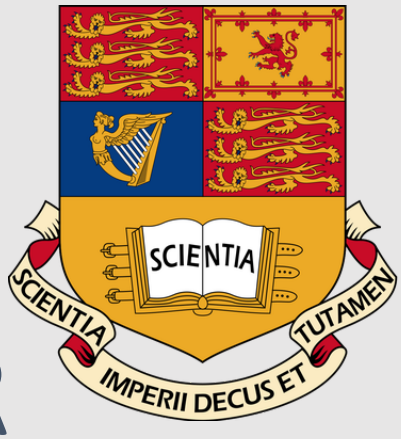


Dr Bouganis,

FPGA FLAP YOUR HANDS

Imperial College London, EIE12019

1st year project

TABLE OF CONTENTS

ABSTRACT

Real time video processing is an interesting problem to be approached via a hardware implementation. The purpose of this project is to use a hardware system, a [1] Field Programmable Gate Array (FPGA)[1], to implement a game with user interaction. By adding a camera video stream as an input and the HDMI monitor to output the player, it will be possible for the system to perceive and react to the user's inputs. The expected result for such a project is a smooth-running game with a user input being a movement. Such result would improve and enhance the user's experience when compared to a screen or board game where they might not be as involved. Whilst the hardware implementation was feasible, the choices to narrow down such a project and optimise it relied mostly on trials and testing. `

INTRODUCTION

A - FLAPPY BIRD: INITIAL GAME DESCTIPTION

Flappy Bird is a classical side-scroller game, developed by the programmer and Vietnamese video game artist Dong Nguyen for iOS. The player controls a bird who attempts to fly between columns of green pipes without hitting them.

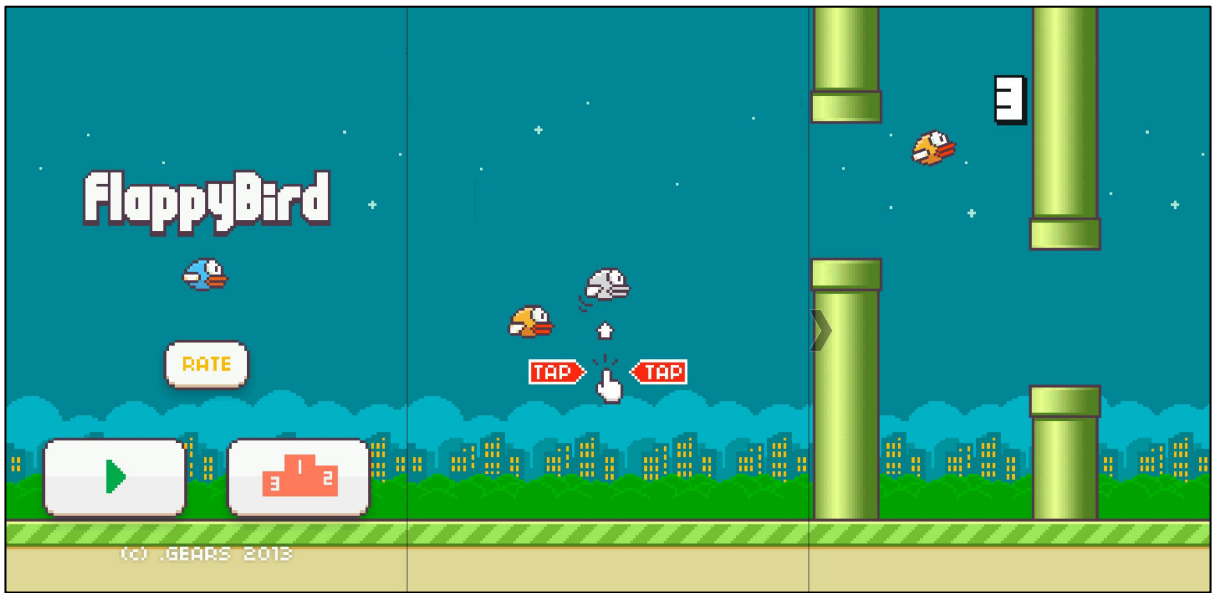


Figure 1 - Screenshot of the game Flappy Bird by Dong Nguyen

Since the background moves at a constant speed, the only move available to the player is to change the vertical position of the bird by jumping. It was the most downloaded free game in the App Store at the end of January 2014 with 50 million downloads[2]. This game rapidly became so addicting that its creator decided to remove it from online stores.

[1] Field Programmable Gate Array (FPGA): Gate-array where the logic network can be programmed into the device after its manufacture. Description on part II, B.

[2] Source: BBC.

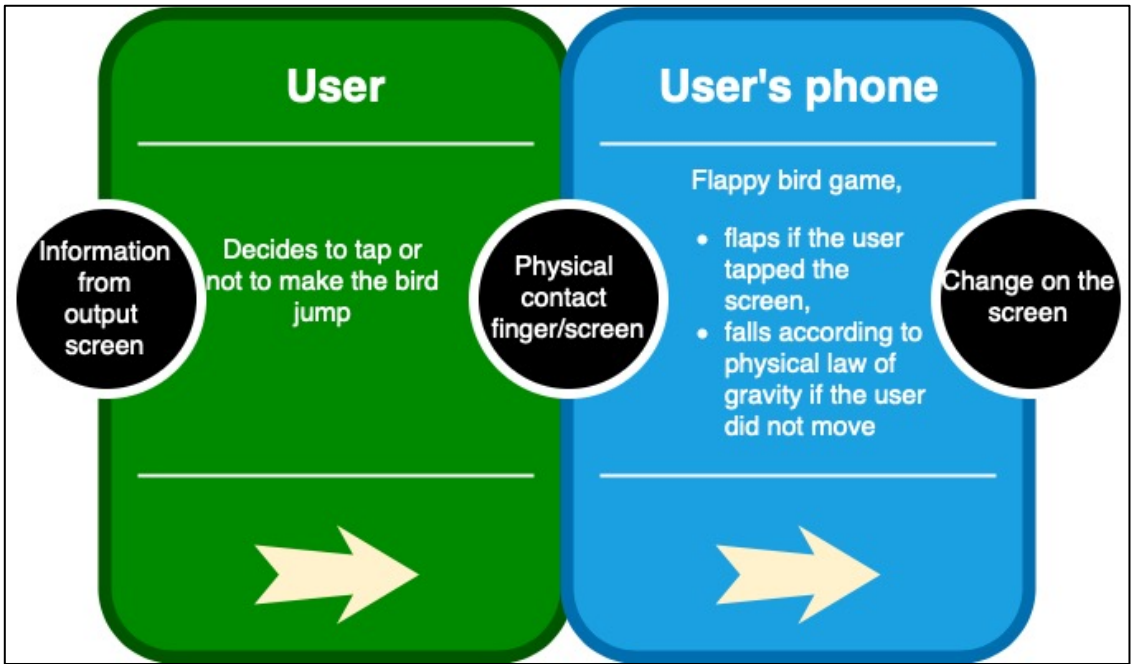


Diagram 1 - Flappy Bird Information Flow

B - VIDEO INPUT FLAPPY BIRD: UPDATED DESCRIPTION

The expected output of the project developed in this report is to implement the game flappy bird as described above, on a Field Programmable Gate Array (FPGA).
The main purpose of this design is to allow the user to interact with the game by *flapping* their hands in the air, in front of the monitor. In this report, a flap will be defined as the arm moving from being perpendicular to the body to arms lying along the torso.

Figure 2 - Man "Flapping" his arms

This specific move should result in making the bird jump a certain height, according to the game. The user's move would be recorded and stored through an input video; each frame will be transferred to the PYNQ[3] board.

[3] Pynq is an open-source project from Xilinx to make the design of embedded system process easier.

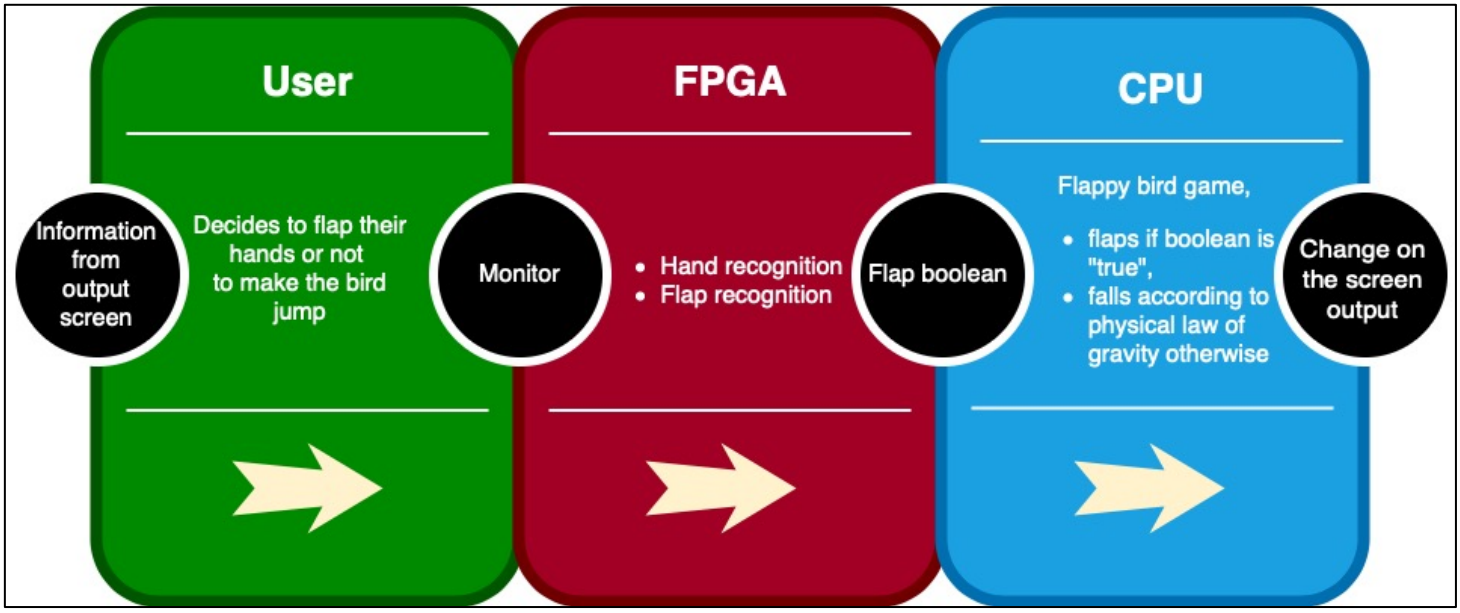


Diagram 2 - Flap Your Hands Information Flow

C - OBJECTIVE OF THE WORK

The idea behind such work is to distance the users from the phone and create a more ludic, interactive and realistic game which recognise the user's features. In order to get efficient results, it has been chosen that the user will wear a bracelet to ease the recognition of movements.

D – BACKGROUND THEORY

Since there is no online document nor any book that refer to anyone attempting to develop such a game on a FPGA, there is no possibility to makeup a background theory based on previous attempts to design this project.

PROJECT MANAGEMENT

The elaboration of this project involved the use of several skills, such as planning, coding in python, C/C++, using the provided tools of Vivado and Vivado HLS, or having an overall understanding of how to use the PYNQ board.

Once the primary ideas for the project were set, a preliminary allocation of work split the work between the team members. This allowed to provide an overview of the project's theoretical evolution over months. This table being very general, its layout left out freedom of organisation since it could contain uncertainties, changes due to unexpected constraints or errors in the design process.

PRELIMINARY ALLOCATION OF WORK

Table 1 - Monthly allocation of work

	February	March	April	May	June
Jaafar	Labs & Python testing	"Detect flap' code	Python flappy bird	Python flappy bird display	Oral presentation and end of the project
Maëlle	Labs & Python testing	Labs & report	Result analysis	Report writing	
Victor	Labs	Labs & report	High level synthesis	'Detect flap' skin color	

A GANTT chart was also created in order to give an overview of the different phases of the work and how it could be allocated. Instead of using a strategy based on weekly tasks, it was decided to set deadlines depending on how long each task was expected to last. Such technique allows to visualise both the project's milestones (phases), structure and each team member's contribution to the work. Because some tasks of the project depend on others to be done, this chart also illustrate how each phase contributes to the goals of the project as a whole. Whilst the first GANTT chart[4] was done keeping in mind that some parts were expected to change, this one is more realistic since it has been developed from the various unexpected constraints faced during the project. It was decided to prioritise the creation of a more 'basic' game, easier to code and to synthesise, so it could then be improved regarding the amount of time left and team's ideas.

The chosen strategy for the allocation of work was based on each team member's strengths. One member will mainly code, another one can do the planning and preparation researches and the last member will be the most comfortable using Vivado. Even though this strategy was set, the fact that every team member will take part in every step of the project must be outlined. Indeed, one of the goals of this project is that every member is comfortable with every phase of the project.

[4] GANTT Chart from the previous report, cf Flap your hands, April 2019

GANTT chart (updated) – example date of the chart progress state : 7/05/2019

Table 2 - Update GANTT Chart

Project Start: Mon, 2/11/2019

TASK	ASSIGNED TO	PROGRESS	START	END
Phase 1 - Set up of the project		100%		
Generating ideas	Victor, Maëlle, Jaafar	100%	2/11/19	3/11/19
Keeping track of the labs	Victor, Maëlle, Jaafar	100%	2/11/19	3/11/19
Phase 2 - Detect input movement using openCV				
Skin color detection	Maëlle	100%	3/11/19	3/26/19
Average personalised object position	Jaafar	100%	3/11/19	3/26/19
Phase 3 FPGA - implementation				
Program " detect flap"	Jaafar	100%	3/26/19	4/5/19
High level synthesis	Victor, Maëlle, Jaafar	100%	4/5/19	4/20/19
HLS program for skin color detection	Victor	60%	5/7/19	5/14/19
High level synthesis	Victor	10%	5/14/19	5/21/19
Phase 4 - Interpretation				
Printing in 3D a green bracelet	Victor	50%	5/7/19	5/7/19
Analysing the results, writing the report	Maëlle	60%	3/26/19	5/19/19
Oral presentation	Maëlle	10%	5/19/19	6/8/19
Phase 5 - CPU Operations				
Python Flappy Bird Game	Jaafar	100%	3/26/19	4/5/19
FPGA Feed Coordination	Victor	60%	4/5/19	4/20/19
Screen Output (Display)	Victor, Maëlle, Jaafar	100%	4/5/19	5/7/19

PROJECT DESCRIPTION

I. GENERAL DESCRIPTION

The goal of this project is not to exactly reproduce the game, and all its particularities, but to create a game based on Nguyen's idea, that is smooth and comfortable to play for the users. The following constraints have therefore been set to develop the game to the Flappy Bird created by Dong Nguyen.

1. The display of the game - background, bird, pipes and score counting - will remain the same.
2. The height of the jump, size of the screen output, number of pipes and speed of the background can be modified in order to make the game possible to play. It will take longer to the user to flap his or her hand than to tap on a screen – this has to be taken in consideration whilst developing this project.

The main technical goal of this project is therefore to identify hand-flapping motion realized by the user and responsively update corresponding actions on the game running in real-time on the output monitor.

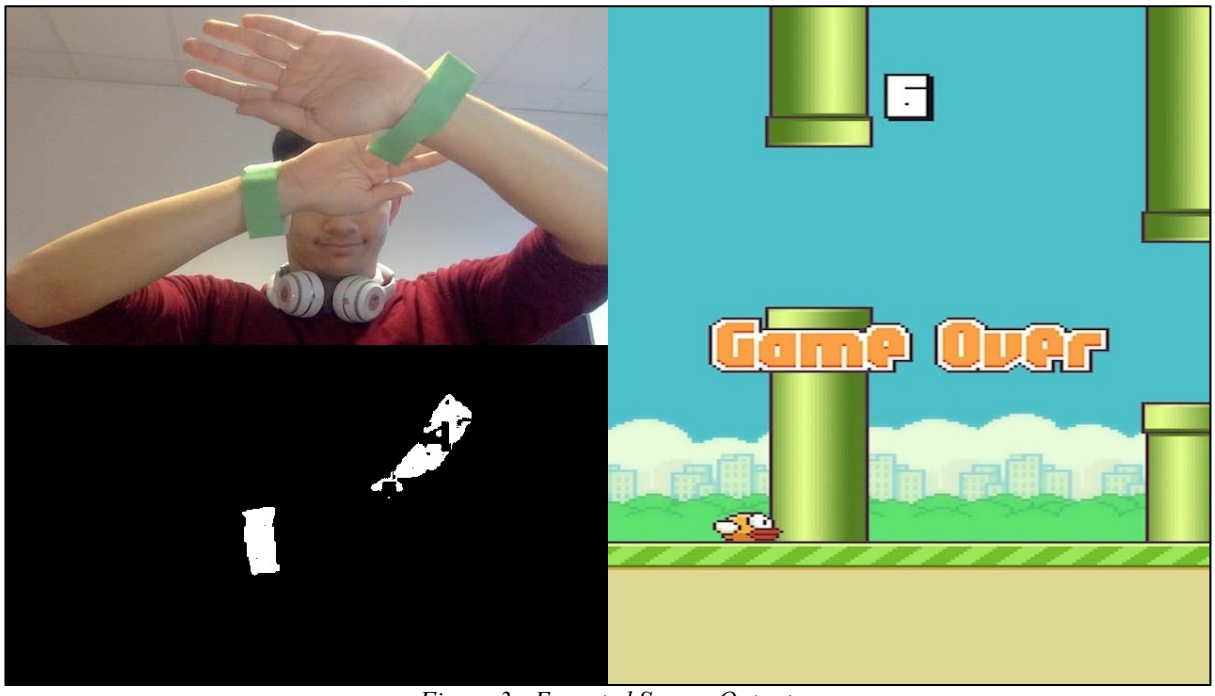


Figure 3 - Expected Screen Output

The expected display – output on screen will not *only* be the flappy bird game like in the conventional phone screens, but a combination of the hand detection, and the game. This will allow the users to see themselves on the screen as well as making sure that the flap detection is efficient. It will then be possible to adapt the threshold for this detection depending on the output screen.

1. LIMITATIONS

It is possible to translate the global constraints from above, combining them with technical constraints set by Imperial College to create the following table of constraints.

Table 3 - Constraints for the project

Function	Appreciation criteria	Flexibility
Main function: Bird from the game jumps when the user flaps their hands	Speed response between user input and game output	±0.2 seconds
Constraint 1: Detecting the user's arms	Noise – number of pixels which are not the user's arms but detected as such.	Undefined – depends on the program's efficiency
Constraint 2: Detecting a "flap" from the user	Correlation between the output Boolean "flap" and the user flapping.	None
Constraint 3: Working performance	Functionality, smooth operations	Time for operations to process not perceived by human.
Constraint 4: Complexity of the project	- Ability to work with different users, - Detection independent from background / environment	None
Constraint 5: Budget and material	Using an HLS tool and the board PYNQ-Z1 from diligent – Xilinx Field Programmable Array (FPGA), SD card and software provided, minimum budget	± 30 £
Constraint 6: Displaying the output	- Flappy bird game response time - Flap detector video output in black and white response time	±0.2 seconds

2. UPDATED DECISIONS

In the previous report about this project (cf Flap your hands, April 2019), the initial proposition was to create two blocks for the flap detection. The following graph sums up this initial proposition.

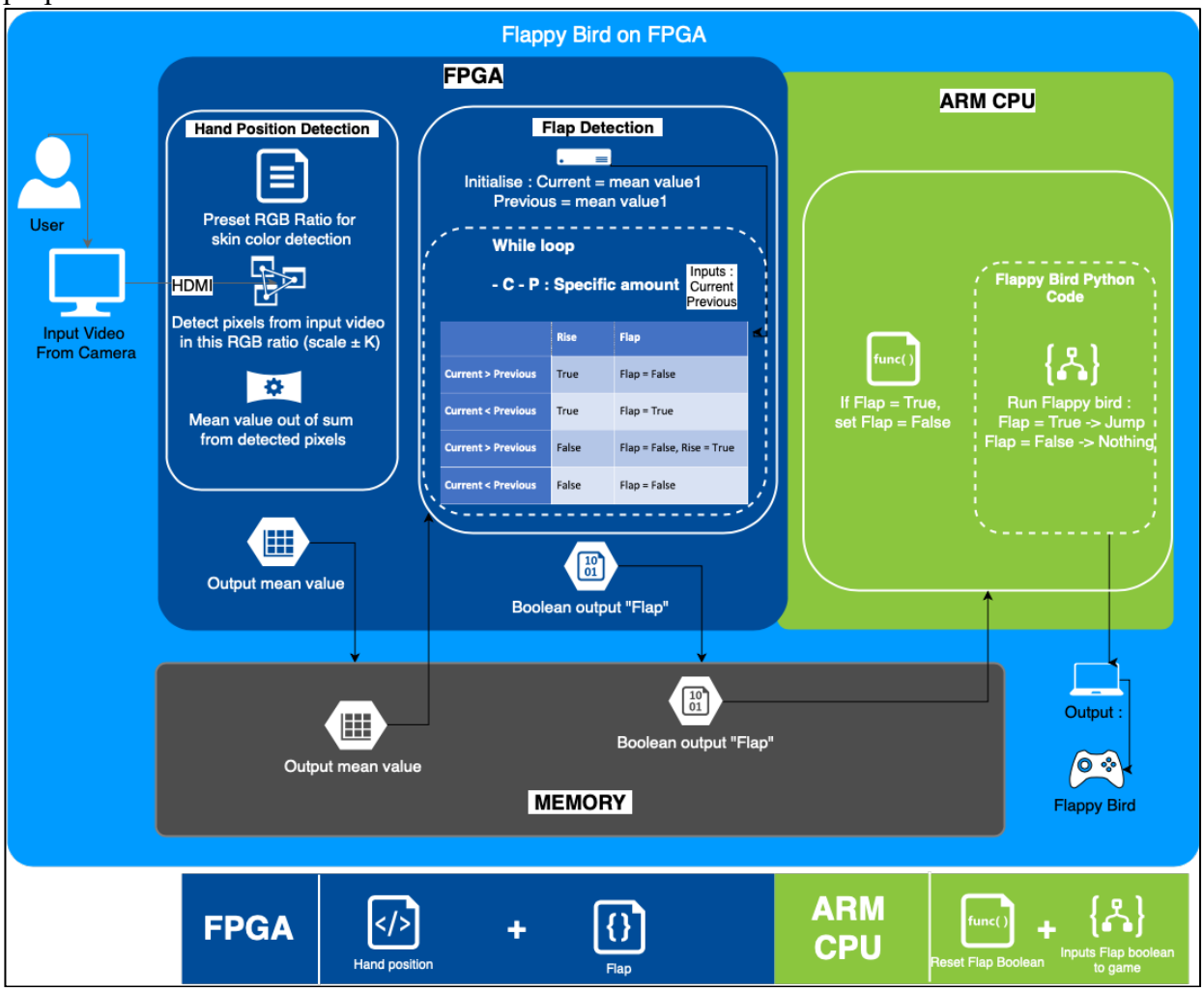


Diagram 3 - Past Logical Diagram of the project

In this report, the decision has been made to combine the 'hand position detection' and 'flap detection' blocks in order to generate only one block of hardware. The reasons behind this choice are the following:

1. One function will be quicker to write then two functions that depend on each other.
2. By combining the two functions, there is no need to store the mean value of the pixels and transfer it to the flap detection, which will reduce the time to compute the Boolean "flap". On the diagram, this correspond to suppressing the 'memory' box.
3. There are less risks of errors between block diagrams due to clock timing or connecting the blocks.
4. This will reduce the time to generate the bitstream. Writing two functions would double the amount of time due to all the operation to run the HLS synthesis, RTL verification, IP and bitstream creation.

It was also discussed rather the Flappy Bird game should run on CPU or FPGA. Due to time requirements, it was chosen to keep the game running on the CPU as a base idea.

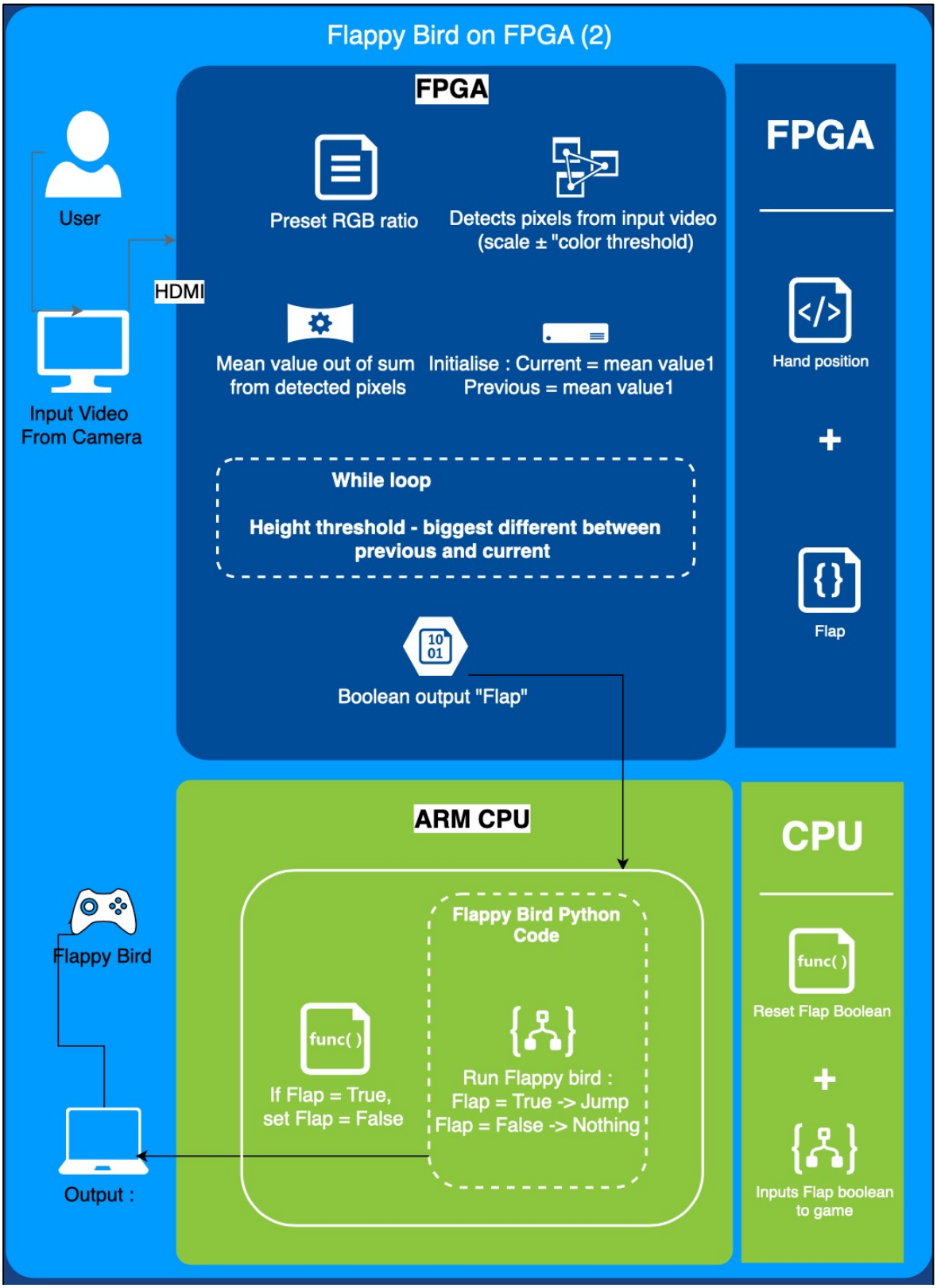


Diagram 4 - Current Logical Diagram for the project

The functionality of each block described on this diagram can be found in part II, High level description.
The comparison of these two diagrams enhances the four points above. By combining the two blocks running in the FPGA, less variables are used between blocks, and the functions are simplified. The part of the programs which are written on the same line can run at the same time, reducing the throughput.[5]
The exchange of information within the blocks is also simplified and clearer, as the following diagrams can show.

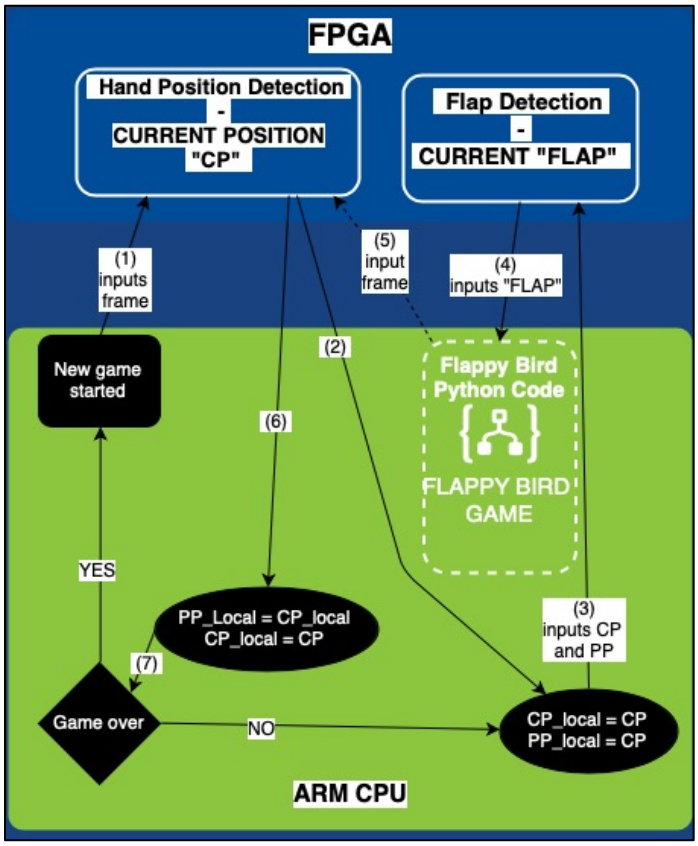


Diagram 5 - Past Information flow between FPGA and CPU

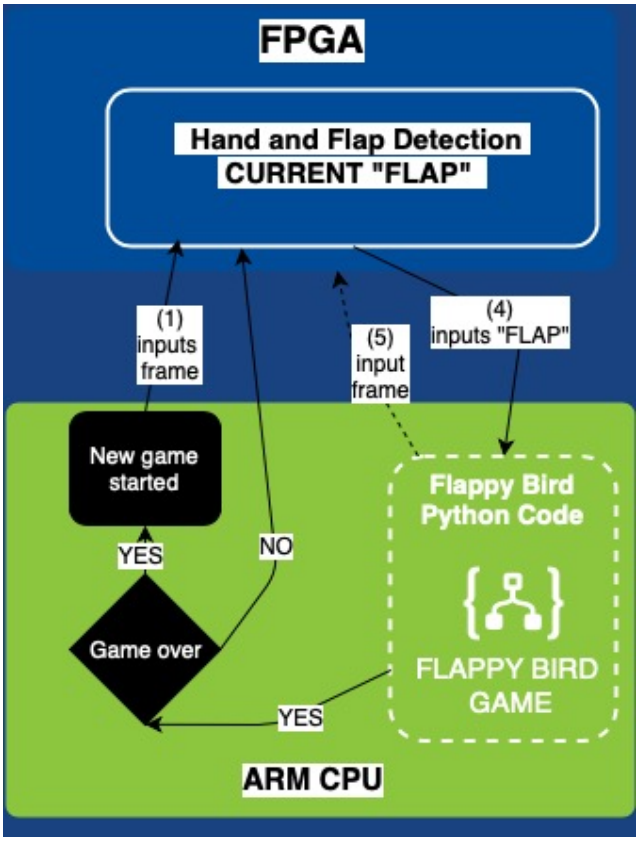


Diagram 6 - Current Information flow between FPGA and CPU

The comparison between these two diagrams suggests the idea of a simplified logical flow of information. Furthermore, the game over can be generated independently inside the python game (on CPU), without any need for input variables from the user's arm position.

The layout of the display has also changed. Instead of showing the output for "rise", "flap" and the code on the right-hand side, the screen will be split in 3. The left-hand side will compare the input video with the black and white video output, whilst the game will be played on the right-hand side.

[5] Throughput: measure of how many units of information can be processed in a given amount of time.

II. HIGH LEVEL DESCRIPTION

Before generating the hardware for the game to run smoothly, it is primary to write the software first and make sure it is optimised.

1. WRITING THE CODE

A) PYTHON SIMULATION

The final block will be written in C++, in order to be synthesized in Vivado. Some specific decisions about the "flap detection" block which had to be made beforehand needed some testing.

Writing a code in python allowed a first easy use of the OpenCV library[6]. In order to get a first hands on experience of the project. This library saves the time of writing the full colour or detection program. The code can be found in the appendix.
The process requires different steps:

- Initialize skin colours with a box
- Apply the range on every pixel in the frame and colour the pixel black or white

The idea behind the recognition on this program, is to use a ratio between Red, Green and Blue (RGB ratio), in order for the recognition to be possible independently from the change of lighting or shadows.

- Input: video stream
- Output: mean value of the skin colour, black and white video – skin in white, background in white.

The thresholds are the output of the get_means functions

The purpose of the little program was to try different colors and set a threshold for the recognition of the flap.

[6] OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products.

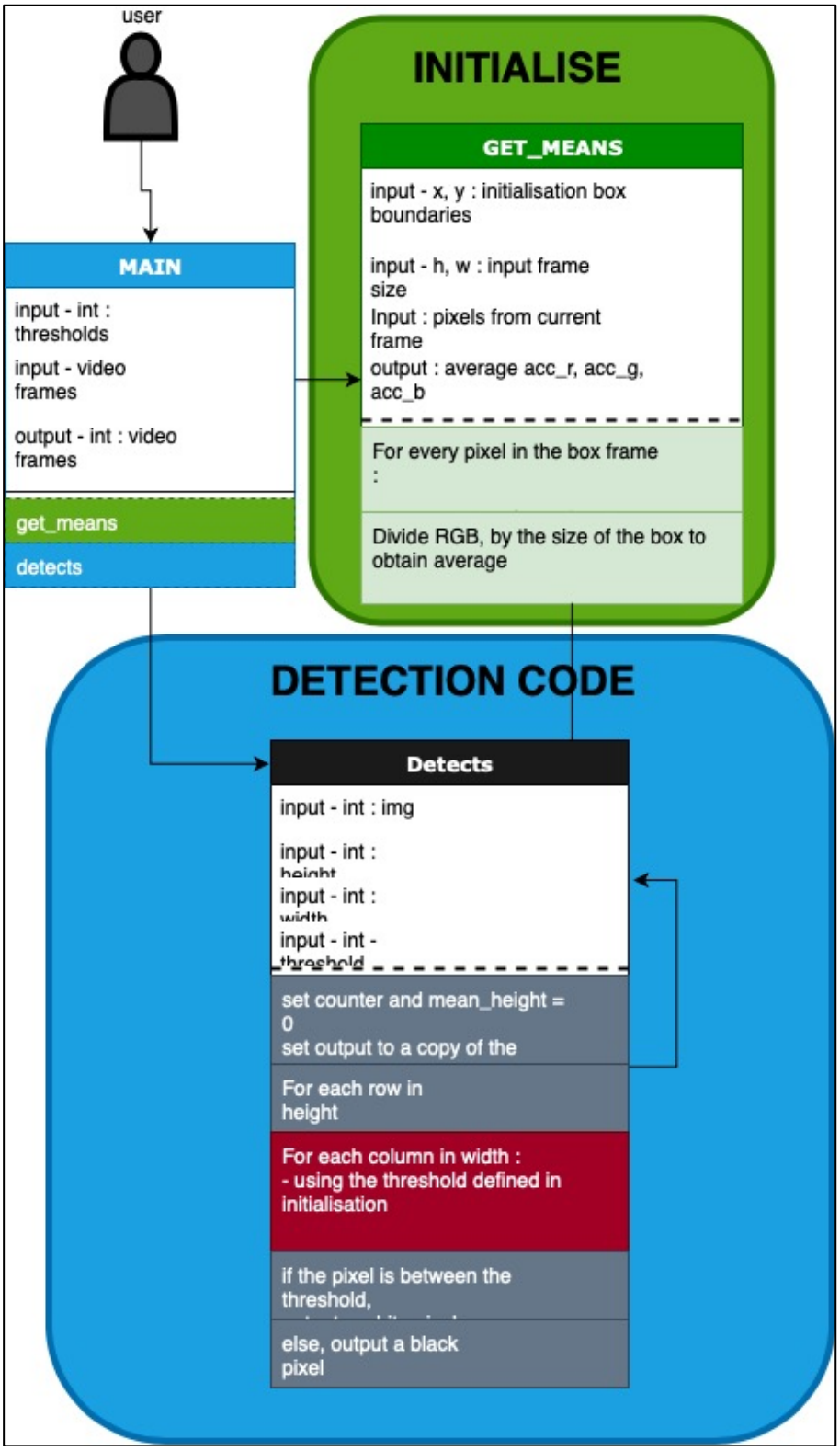


Diagram 7 - Colour recognition code summary

(i) SKIN COLOUR TEST

By setting the parameters on skin colour – using a hand for initialisation – the image below depicts the amount of noise.
The noise on this image is due to the fact the program uses RGB ratio. Skin colour being often close to pink or brown, the amount of red in the ratio is very high, whilst the green and blue are low. Most of walls, desks, tables and chairs are conventionally red, light beige or brown – which increases the chances of having noise when using a threshold that is too high.

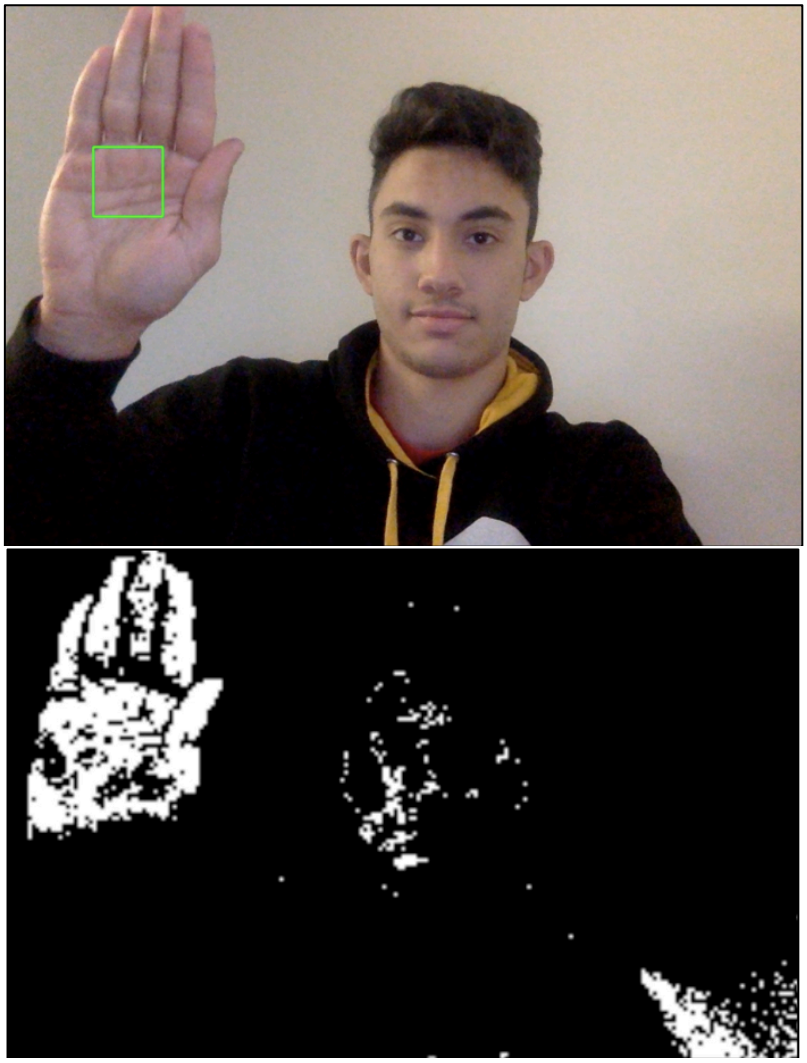


Figure 4 - Results of RGB Hand Recognition

Table 4 - Comparison of results and expectations for hand recognition

Element of the picture	Hand	Face	Body	Background
Expected	White	White	Black	Black
Output	90% White	20% White	Black	90% Black

The output image is satisfying yet can be improved. On the output picture, the hand is outputted in white pixel, but not completely. The face is also detected, as well as part of the background. Using different skin colour tones improved the recognition. The next step for more improvement would be to use a non-skin colour and compare to the result above.

(ii) GREEN COLOUR TEST

On the other hand, using a non-common colour, such as green, surprisingly improves the colour detection. The choice of this colour was made based on a RGB scale because Red and Blue are common colours in the world[7].

Table 5 - Comparison of results with expectation for green recognition

Element of the picture	Bracelets	Arms & Face	Body	Background
Expected	White	Black	Black	Black
Output	White	Black	Black	Black

On the picture below, only the green scanned bracelet used for the initialisation is detected, there is almost no noise. The contour and filling of the bracelet is almost ideal – all green pixels from the input video are outputted as white pixels. The background is not detected and does not generate any noise.

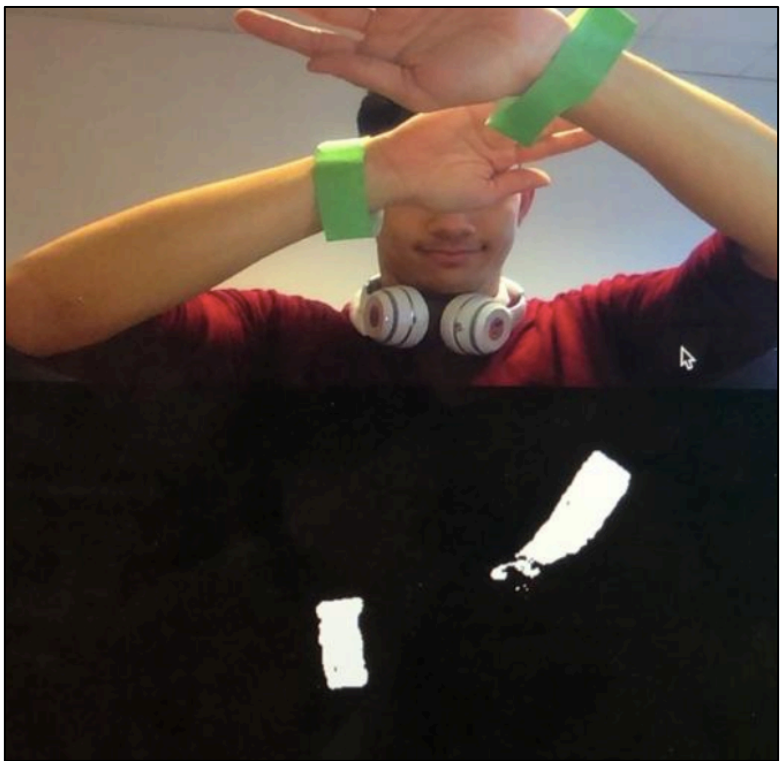


Figure 5 – Results of RGB Green recognition using Python

From these results, it was then possible to conclude that an object with a specific colour will preferably be used in order to improve the detection of the flap movement. Using an RGB ratio being not ideal for skin recognition, the first version of the project will be to implement flappy bird for a user wearing green bracelets:

- easing the process of movement recognition,
- allowing to pre-set the RGB ratio. The user will not have to initialise the game with his skin colour.

To go further, it will be possible to implement skin recognition with another ratio such as HLS and compare it with RGB ratio for a green bracelet.

[7] Source : Quora

B) C SIMULATION

The python code enhanced the idea that it is possible to use RGB ratio to detect a particular colour from the input video. The amount of noise was too high when the skin colour was initialised depending on the user. This first test therefore led to conclude it is more efficient to use green bracelets.

- The function detects a colour.
- It has been decided to put the sensitivity threshold as an input in order to be modified from the python code without any need to synthesise the whole function again.
- The flap detection logic is included in the function. It is however split between the hand position detection – similar to the python code above – and a flap detection – Boolean from the hand position.

The whole function can be found in the appendix. The main variables similar to the ones used in the Python test code.

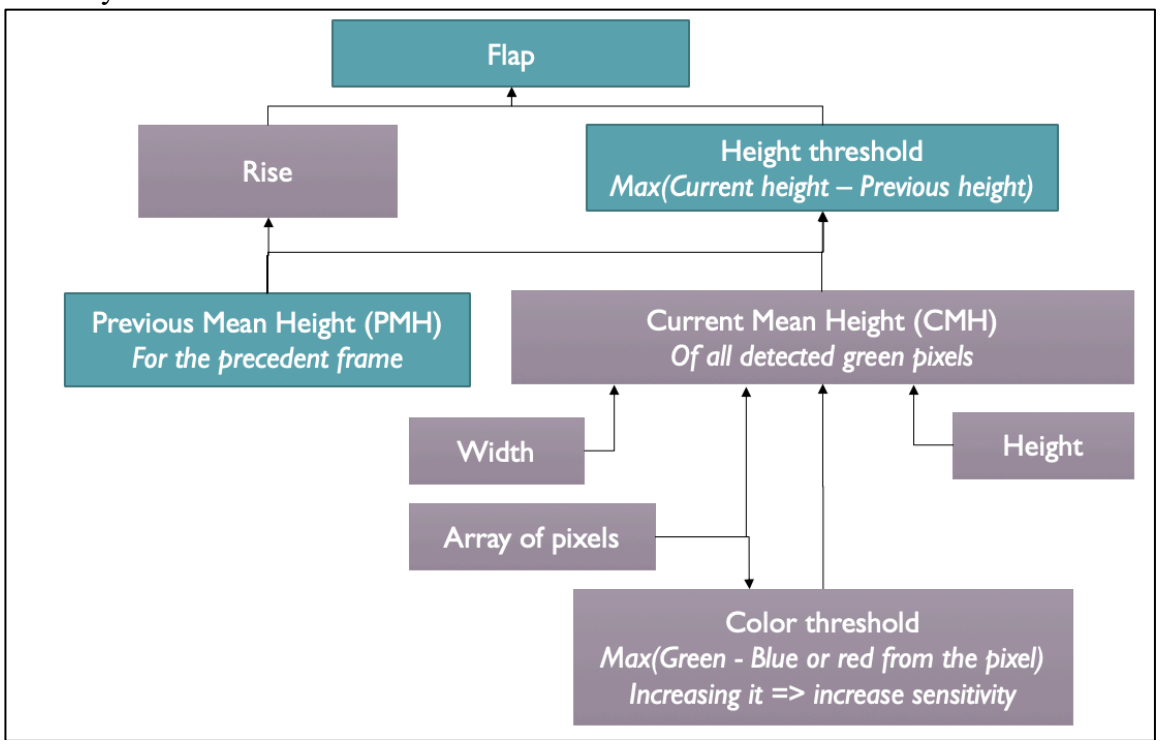


Diagram 8 - Variable Dependance in "flap" function

The diagram sums up the needed input and output for the Hand position detection (in gray) and the Flap detection (in light blue). The following table resumes the input and output needed for each function

Table 6 - 'Flap' functions inputs and outputs

Operation	Inputs	Outputs
Hand position detection	Array of pixels Height Width	Array of pixels modified Mean height (current position)
Flap detection	Current position Previous position Rise state	Flap state

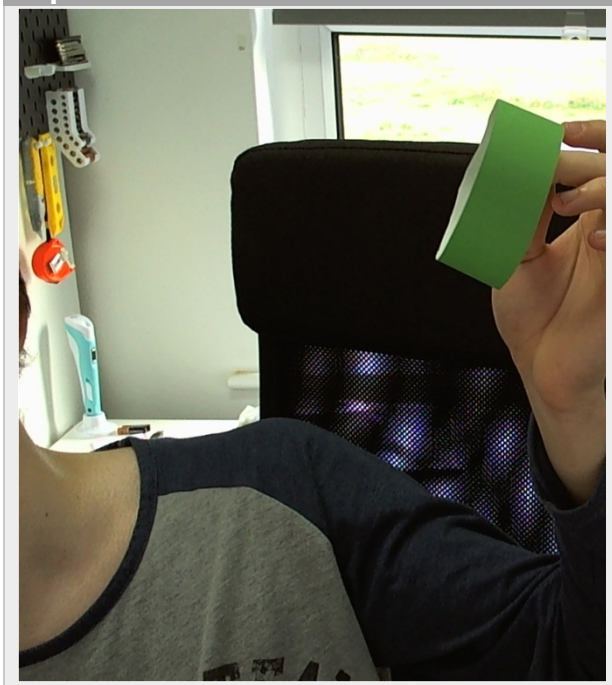
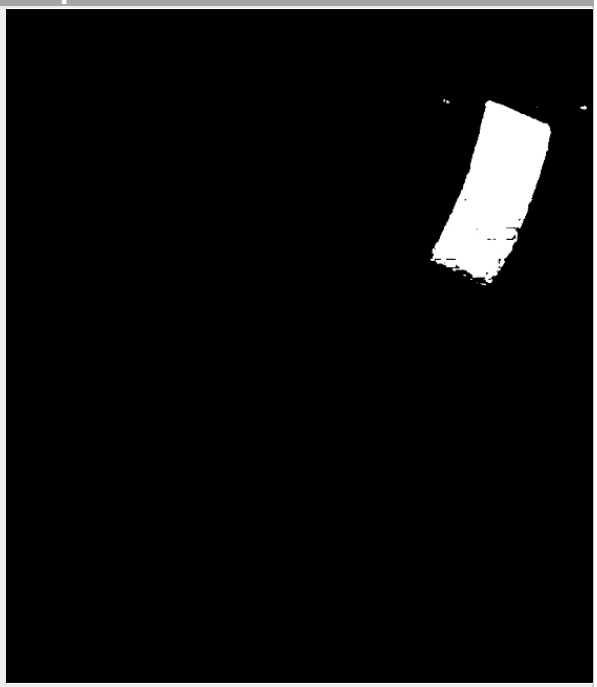
Table 7 - Logic Table for "rise" and "flap" boolean outputs

Inputs		Outputs
CMH ≥ PMH	Rise = True	Flap = False
CMH < PMH	Rise = True	Flap = True, Rise = False
CMH ≥ PMH	Rise = False	Flap = False, Rise = True
CMH < PMH	Rise = False	Flap = False

The results of this function when running the C Simulation are similar to the Python test. The colour recognition is particularly satisfying since there is no noise from the background.

GREEN COLOR DETECTION

Table 8 - Input/Output comparison for C code Green recognition

Input	Output
	

This function was optimised regarding its future efficiency as a block of hardware.

1. Since the HDMI in frame has the camera reduced to the upper quarter of the screen, this is the only area that has to be considered when calculating the mean height.
2. While scanning this area, it is more efficient to save the black and white output image on the FPGA memory. It will then be retrieved when reaching the corresponding region in the output.
3. Using only one for loop instead of several ones allows a maximised unroll.
4. Reducing the resolution of the image also helps the design process time, optimising the game reactivity.

SKIN COLOR DETECTION

Since the initial idea involve skin detection, it might be interesting to compare the solution above with skin detection. Such solution would require 2 IP blocks.
one that runs once for every player for the system to calibrate to their skin colour,
and another that is the similar as the flap detector above.

Since the test of skin color detection using python OpenCV library in RGB was not conclusive, it is preferable to try using HSV. This would imply dealing with the mean Hue, and brightness values and saturation thresholds in order to decide if a pixel is a skin colour.

CALIBRATION

The calibration can be made by calibrating the expected value. This will be the mean value taken from an array of pixel, inputted as a square. The user can put their hand on this square, press a key and calibrate for the skin. This whole square should be filled with the skin when calibrating.

The input would be just the image, and the calibration square dimension in pixels. The output is the mean hue, mean brightness value and mean saturation. The thresholds of hue, lightness and saturation can be set manually in order to compare efficiency depending on how much noise is generated in each case:

(i) Creating new coefficients for RGB

$$R' = \frac{R}{255}; G' = \frac{G}{255}; B' = \frac{B}{255}$$

$$C_{max} = \max(R', G', B')\,;\; C_{min} = \min(R', G', B')$$

$$\Delta = C_{max} - C_{min}$$

(ii) Hue Calculation

$$H = \begin{cases} 0^\circ\,, & \Delta = 0 \\ 60^\circ * \left(\frac{G' - B'}{\Delta} * mod[6]\right), & C_{max} = R' \\ 60^\circ * \left(\frac{B' - R'}{\Delta} + 2\right), & C_{max} = G' \\ 60^\circ * \left(\frac{R' - G'}{\Delta} + 4\right), & C_{max} = B' \end{cases}$$

(iii) Saturation calculation

$$f(x) = \begin{cases} 0, & \Delta = 0 \\ \frac{\Delta}{1 - |2L - 1|}, & \Delta < 0 \; or \; \Delta > 0 \end{cases}$$

An issue raised by such formulas is that it implies the use of floating points, which would slow down the overall function. In order to use less resources and run a faster design, it is possible to use integers. This will not imply to change all of the formulas used. The order of the operation must however be controlled in order to prevent rounding which can affect the result.

The final result will therefore only be an approximation, which can be acceptable for the application needed – skin recognition and not displaying of a new output. Not using floating points has an expected effect of massively increasing performance.

Figure 6 - Input image for skin detection using Vivado HLS

Figure 7 - Output image for skin detection using Vivado HLS

The results above are from the Vivado high-level synthesis. The output image noise is not reduced in comparison to an RGB colour detection. Using a green colour is therefore still the best and safest solution for an optimised "flap" detection.

2. CREATING BLOCKS OF HARDWARE

The following first paragraphs describe the path to generate hardware from a C++ code using Vivado and Vivado HLS. The ones below are how this knowledge was applied for the implementation of Flappy Bird.

A) HIGH LEVEL AND LOGICAL SYNTHESIS

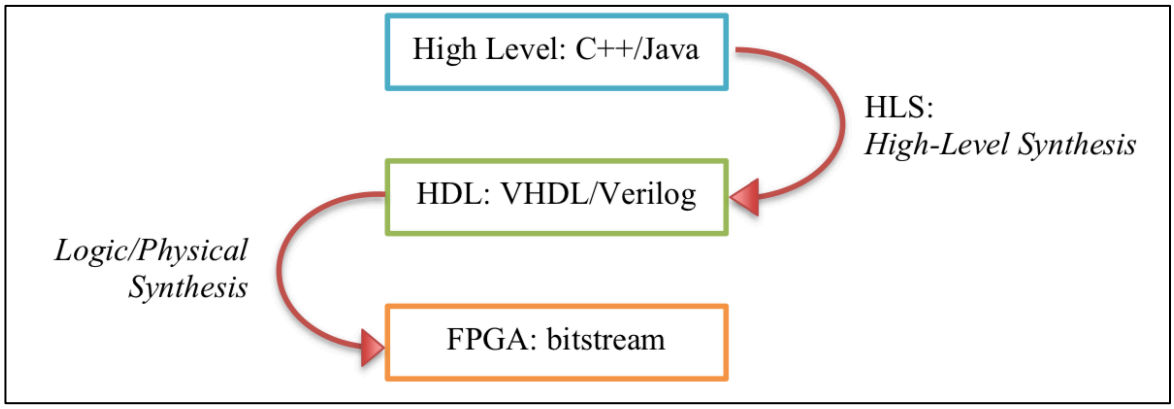


Diagram 9 - Language levels and types of synthesis

The purpose of using Vivado HLS and Vivado is to break down a high-level language into logic blocks in order for it to be stored as hardware. Overall, the process involves:

1. A HIGH-LEVEL SYNTHESIS (using Vivado HLS) to transform the high-level language - here C++ - into Verilog or VHDL (VHSIC[8] Hardware Description Language). This is a hardware description language used to describe digital systems such as FPGA and integrated circuits.
2. A LOGIC SYNTHESIS (using Vivado), from the Hardware Description Language to a bitstream that can be sent on the FPGA to access the hardware.

The following diagram displays the link between the C code and the Register Transfer Level (RTL) code. This code is an intermediate language between the high-level description (behavioural) and purely logic (Structural). A RTL "wrapper" for the HDL language allows to use the testbenches from the C code to make sure the generated code by Vivado HLS performs the correct directions.

The following diagrams give a visual representation of this process.

[8] VHSIC : Very High Speed Integrated Circuit

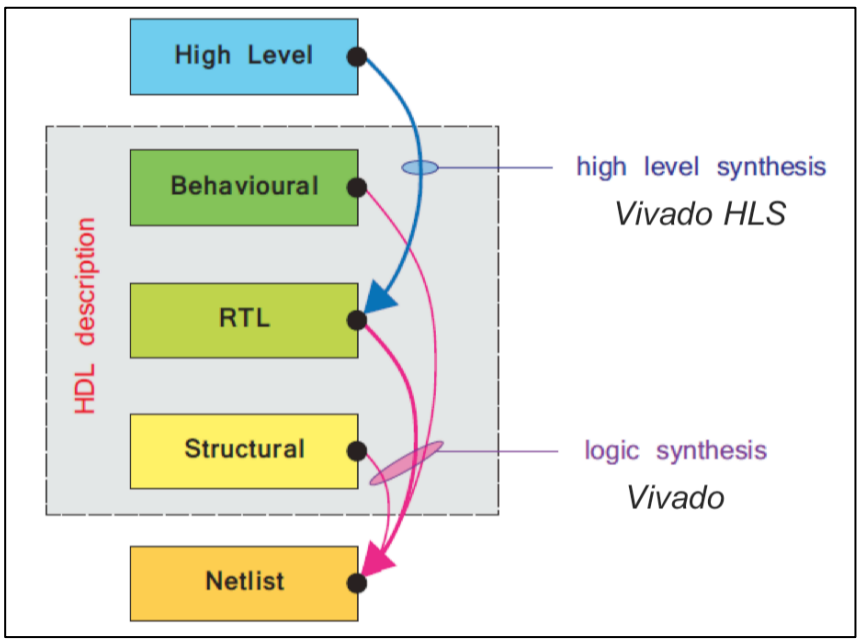


Diagram 10 - Language Levels and tools to translate one from another

Once the RTL[9] code is created, it is therefore possible to use the testbench to evaluate the implementation cost in hardware of the C code. The "design iteration" is an important part of the process where the goal is to optimise the high-level synthesis code to decrease the amount of hardware used and increase the throughput. In order to do so, some techniques that can be used are:

1. Changing the clock frequency to perform more operations in a single clock cycle,
2. Pipelining the loop or the data flow. For example, loop unrolling can be used to perform the different iterations of the same loop in the same clock cycle,
3. Change the custom precision of the variables – length of variable used. For example, a signed 32 bits arithmetic number needs more clock cycles to be computed than an unsigned 16 bits number.
4. Several operations on loop can also be made such as merging them if they don't need each other's data to be processed.
5. Since it is very costly in time to access memory, the choice of memory type is crucial. Changing the array partition can optimize the throughput of the program.

[9] RTL : Register Transfer Level Language

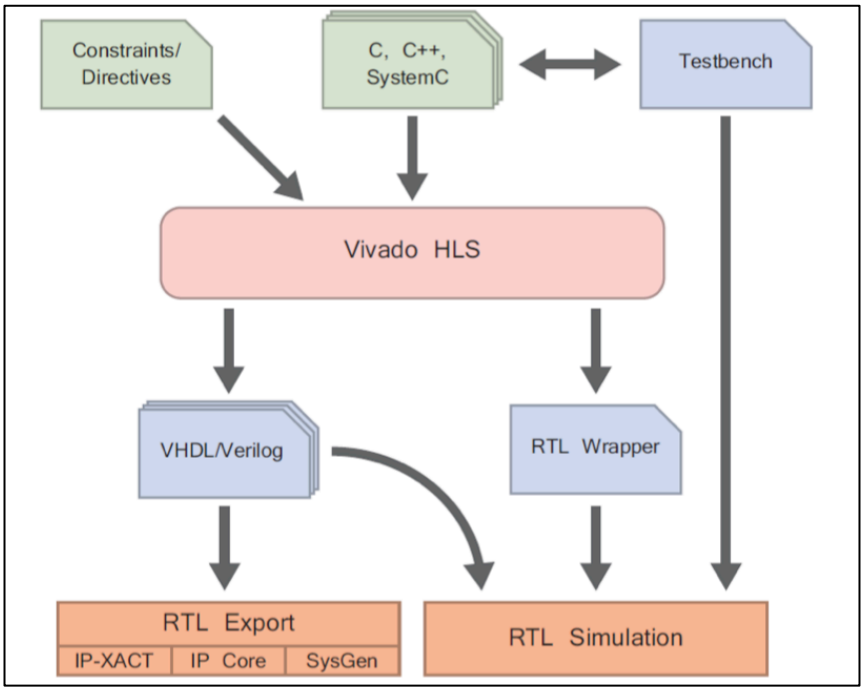


Diagram 11 - Vivado HLS structure

B) AVAILABLE MATERIAL – PYNQ BOARD

One of the constraints of the design is for it to be performed using a HLS tool and the PYNQ-Z1 board from Diligent. Among other things, this board contains:

- A 32 bit processor core,
- A Xiling 7 series FPGA,
- On-board memories,
- Video and audio I/O,
- USB,
- Ethernet,
- HDMI.

It uses Python for both programming the processors and interfacing with the overlays.

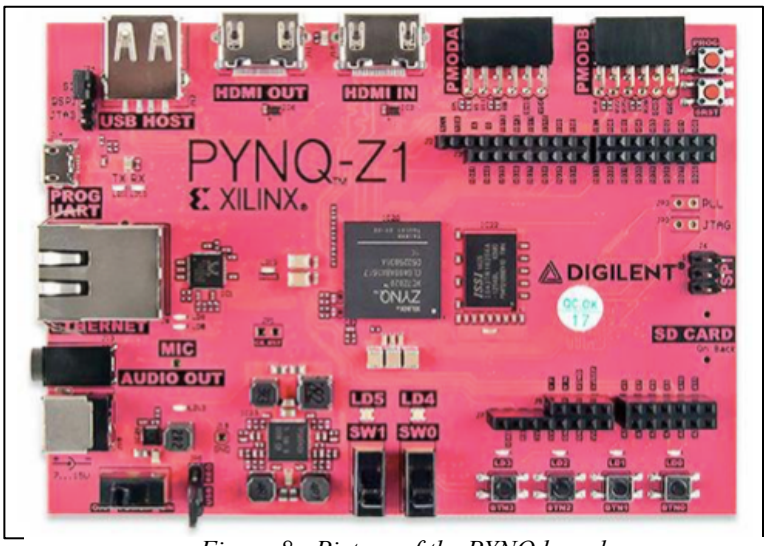


Figure 8 - Picture of the PYNQ board

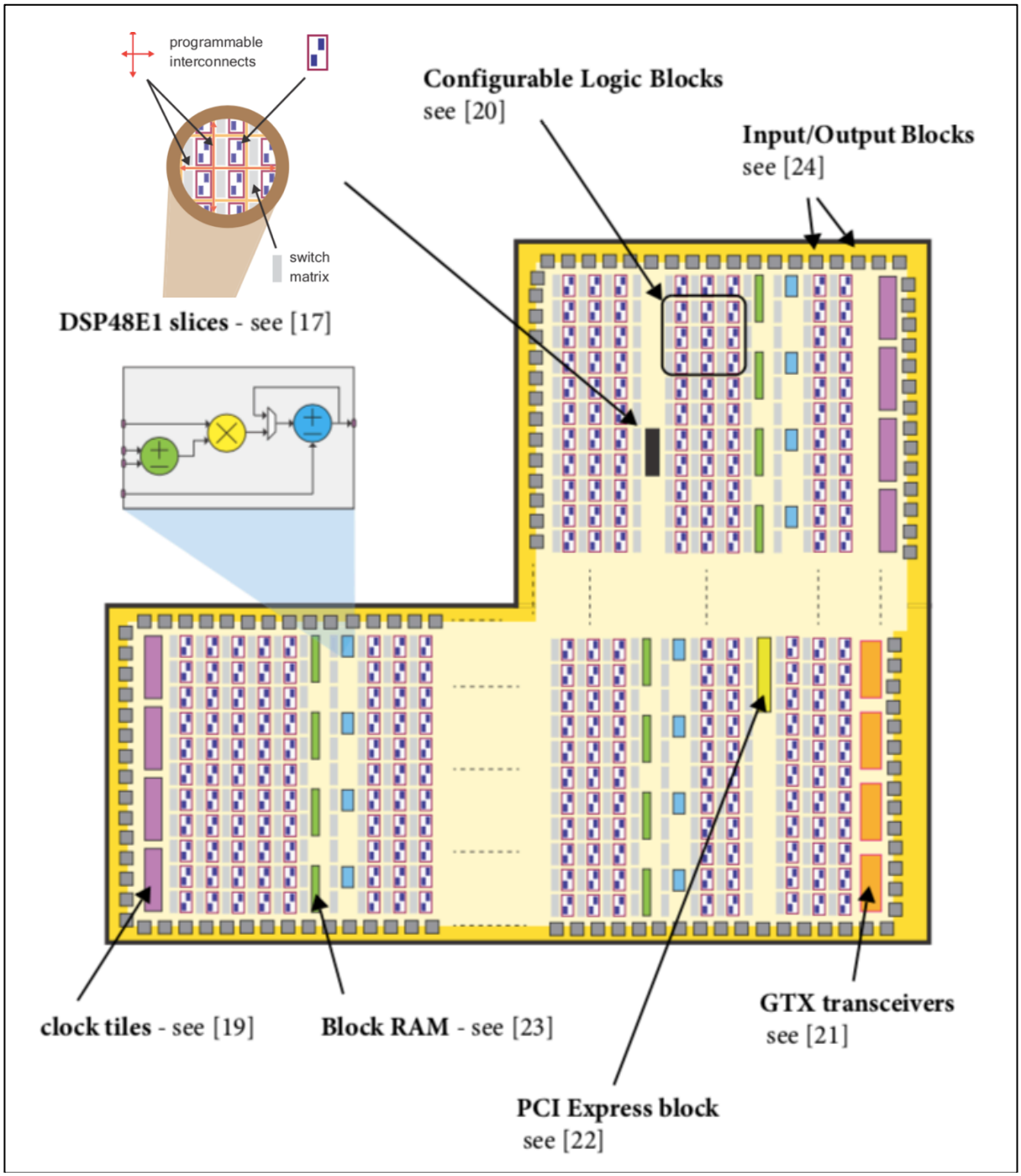


Figure 9 - Programmable Logic (PL) board structure

The Programmable Logic (PL) board structure is made up of logic circuits called overlays.[10] Each Configurable Logic Block (CLB) groups several logic elements. It contains two logic slices.
In the case of this project, every slice is composed of 4 lookup tables, 8 Flip-Flops, and DSP.

[10] Overlays : "overlays are analogous to software libraries. A software engineer can select the overlay that best matches their application. The overlay can be accessed through an application programming interface (API)."

- Flip flops (FF) are the most basic resource, they implement a 1-bit register
- Lookup Tables (LUT) are conventionally used to implement arithmetic operators of a short worldlength. LUT "are most suitable for arithmetic operators with short worldlengths."[11] They can be used and combined to implement several lengths of
    - Read Only Memories (ROMs)
    - Random Access Memories (RAMs)
    - Shift registers
- Input / Output Blocks (IOBs) handle a 1-bit input or output signal to interface between logic resource and the pads connected to the external circuit
- Block RAM have a storage size of 36Kb, but it can be divided into smaller blocks or combined to create bigger ones.
- DSPs are also used for high speed arithmetic.

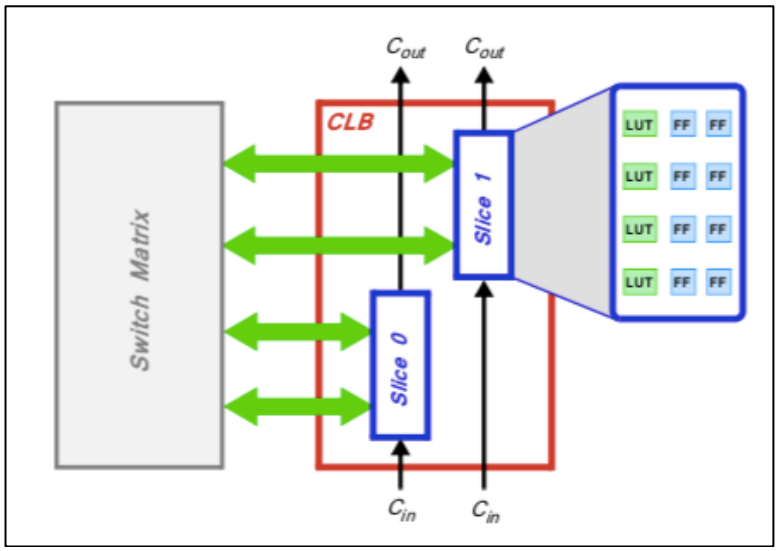


Figure 10 - Composition of a Configurable Logic Blocks (CLB)

To sum up, the process from a high level language to a bitstream that can be sent to the PL, using the provided tools of vivado hls and vivado is the following:

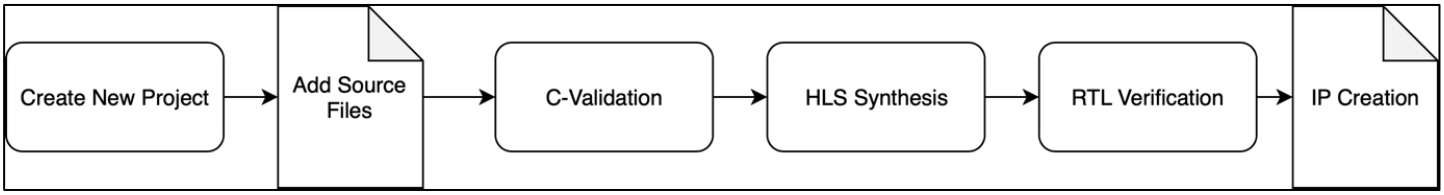


Diagram 12 - How to : high Level Synthesis

[11] Source : The Zynq_book_ebook

III. RESULTS AND DISCUSSION

The first High Level Synthesis was tested between for an output picture of resolution 1080p, with resolution 1920x1080. This choice was made for picture quality, since a 1080p is a Full HD picture. The idea behind this choice is to give a high-level picture on the output screen.

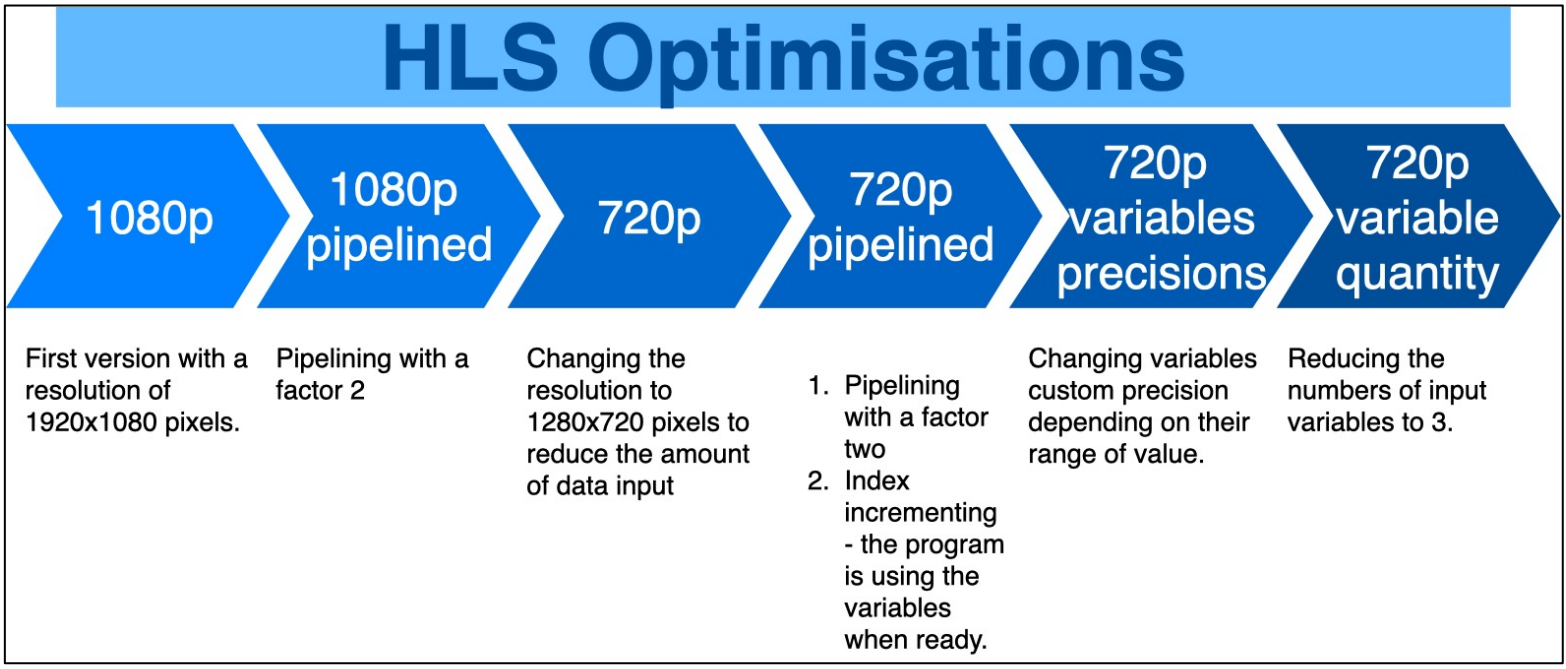


Figure 11 – HLS Optimisation timeframe

1. HLS SYNTHESIS - OPTIMISING THE 1080 P LOOP WITH NO IMAGE INPUT

In this project, optimisation will focus on decreasing latency[12] and throughput with little focus about the amount of hardware used. Since one of the constraints is too design a smooth game with an immediate response to the user's input, the main goal will be to optimise latency. The following tables compare the results after C-Simulation for several optimisation techniques on the main "for loop" as the ones named on Part II, 2, A : pipelining with a factor 2, unrolling, flattening and merging.

[12] Latency is the time interval between initiating a process and receiving the result – time for the function to operate in clock cycle.

COMPARATIVE TABLES AFTER HIGH LEVEL SYNTHESIS - GREEN DETECTION 1080P (NO INPUT IMAGE)

1. Timing (ns)

Table 9 - Comparative timings after high level synthesis - Green detection 1080p

Clock		nonOpt	pipeline	unroll	flatten	merge
ap_clk	Target	10.00	10.00	10.00	10.00	10.00
	Estimated	8.750	8.750	8.750	8.750	8.750

2. Latency (clock cycles)

Table 10 - Comparative latency after high level synthesis - Green detection 1080p

•		nonOpt	pipeline	unroll	flatten	merge
Latency	min	11404838	1036853	23068838	11404838	11404838
	max	11404838	1036853	23068838	11404838	11404838

3. Utilisation estimates

Table 11 - Comparative utilisation estimates after high level synthesis - Green detection 1080p

	nonOpt	pipeline	unroll	flatten	merge
BRAM_18K	2	2	2	2	2
DSP48E	0	0	2	0	0
FF	1723	2002	2521	1723	1723
LUT	2199	2294	3209	2199	2199

The main optimisations affect the ‘for loop’.

1. The results above suggest that flattening or merging the single for loop has no added value since the latency and estimated use of hardware are not affected.
2. In addition, unrolling the loop, surprisingly, makes the design run slower. This might be due to the unroll factor. If the unroll factor is superior to the minimum number of iterations of loops, the design will slow down.
3. Finally, pipelining the operations, improves the latency of the design as expected. The latency decreases from 11404838 without optimisation to 1036853 when using a random pipeline. This first optimisation factor is therefore of

$$11404838/1036853 \cong 11$$

From these results, it is possible to conclude that the best optimisation is **PIPELINING**.

The reason for such different results becomes obvious when looking at what happens for every clock cycle. The table below shows that pipelining does not affect clock cycle behaviour before clock cycle 10. While the unoptimized version performs 6 operations, the pipelined performs 14 operations in only one clock cycle. Similarly, the non-optimised version performs only one operation during clock cycle 16 and 17. These operations are combined in the 11th clock cycle when using pipelining.

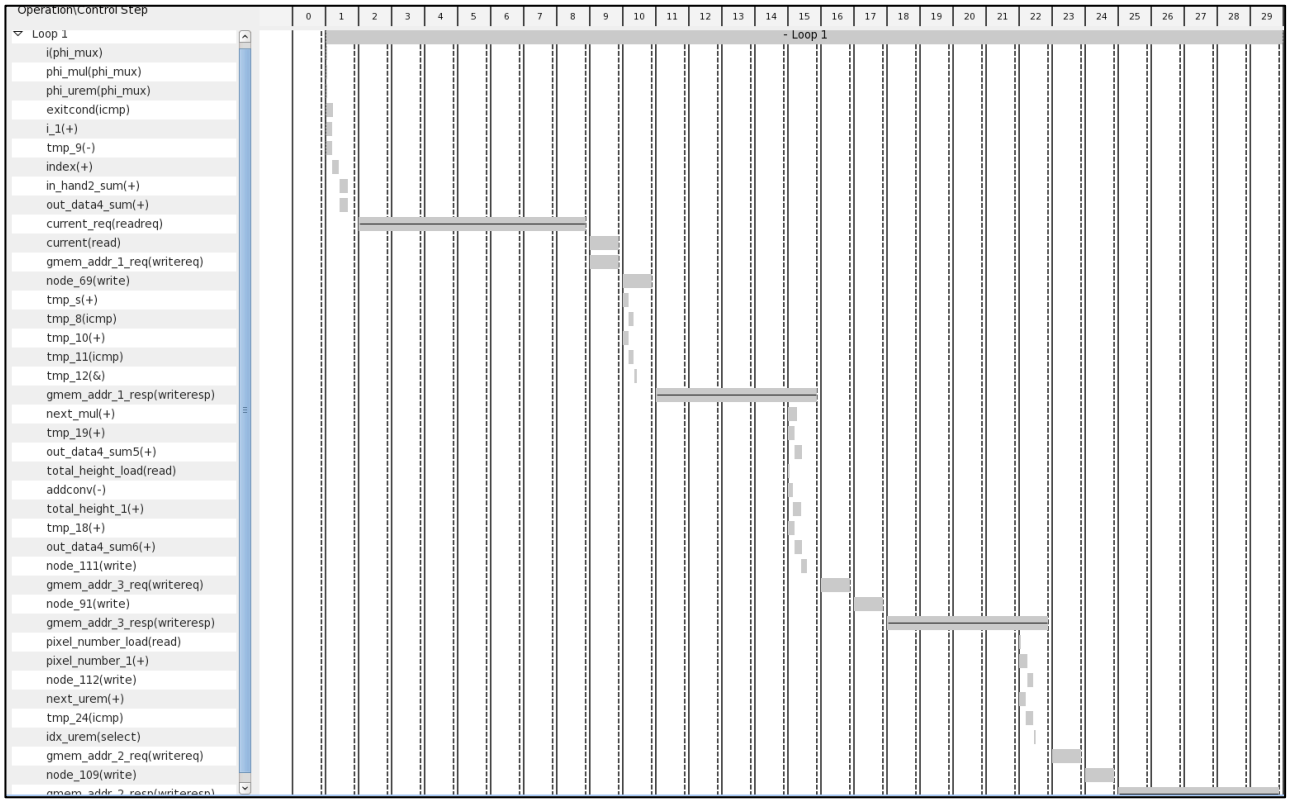


Figure 12 - Operations per cycles (Not optimised loop)

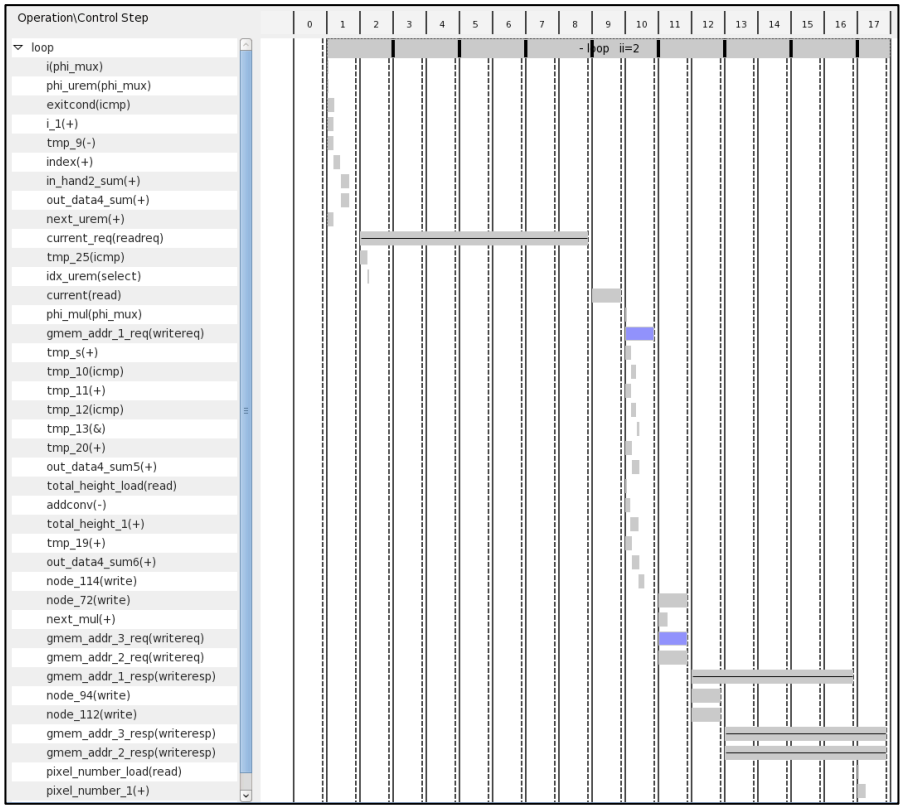


Figure 13 - Operations per Cycles (Pipelined loop, initialisation interval = 2)

The next step is to optimise the pipelining parameters: flushing, initiation interval, and loop rewinding.

Table 12 - Comparative table of different initialisation intervals for optimisation

Initiation Interval (Target)	Achieved	Latency
1	2	1 036 853
2	2	1 036 853
4	4	2 073 651
10	10	5 184 006

It seems that using loop rewinding, increasing the initiation interval will slow down the design – increase latency. On the other hand, flushing does affect latency.

From part II, 2, A, other optimisations can be deduced.

1. For example, the default use of this loop was 32 bits variables. By changing the custom variable precision to unsigned 8 bits integers, the throughput has been improved.
2. Similarly, memory access is costly in terms of number of cycles. Using arrays where possible can be an optimisation to keep in mind when writing an optimised version of this function.
3. The output of this function is a HD picture with resolution 1920x1080 frame. By reducing the resolution to 1280x720 (720p), it may be possible to improve the function even further.

2. HLS SYNTHESIS - OPTIMISING THE OUTPUT 720P WITH IMAGE INPUT

Since the output of this function is a picture with a defined frame, changing the output resolution will optimise the throughput. Since a full HD might not be needed for the output video, it has been decided that a HD output (720p) could be sufficient quality for the black and white picture. Less pixels will have to be outputted, reducing the amount of cycles needed for the function to perform the scan of the input picture.
In order to get an idea of what the output would like, it was decided to change the initial code and add an input image. The 1080p code did not provide an output image for testing. For this case, the colour recognition will provide a single image output.

The picture below outputs in HD, the quality is efficient enough to play the game (on the right-hand side) and check the colour recognition (left hand side). The input is on the quarter side, output recognition on lower left quarter. The game can be played on the full right-hand side. There is therefore no need for a full HD output. This result expects to improve the final throughput.

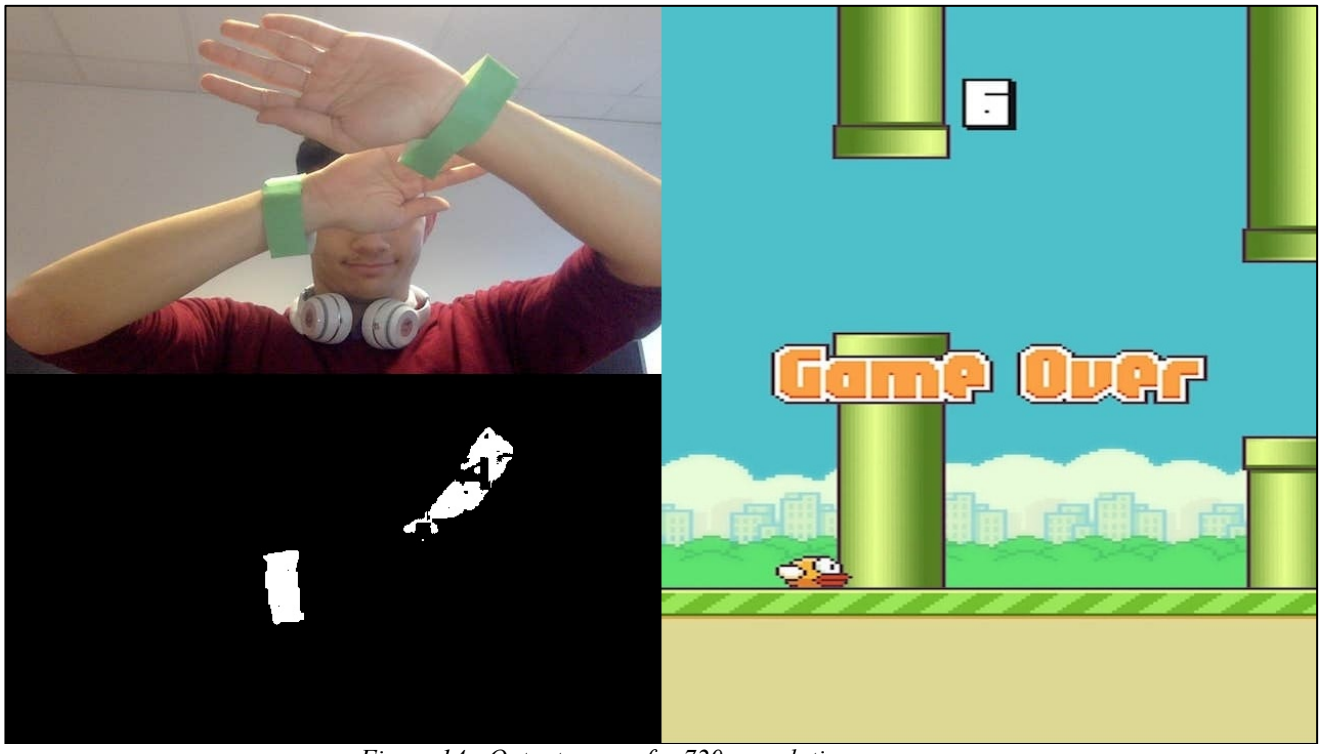


Figure 14 - Output screen for 720p resolution

The C code and testbench are the same as for a 1080p resolution. Running the high-level synthesis gives an estimation of hardware components usage and number of cycles needed.

- Similarly, to the 1080p synthesis, a function with pipelining offers a more stable solution than a non-optimised loop in terms of latency and resources.
- It is expected to get a lower latency for both the not optimised and pipelined versions of the 720p synthesis in comparison to the 1080p resolution synthesis.

The goal of this synthesis is to reduce the latency in order for the function to be performed in less clock cycles. The output video will be smoother and the game more reactive to the user's inputs movements.

COMPARATIVE TABLES AFTER HIGH LEVEL SYNTHESIS - GREEN DETECTION 720P

- **Timing (ns)**

Table 13 - Comparative timings after high level synthesis - Green detection 720p

		No input image			With a I/O image		
Clock		**nonOpt**	**pipeline**	**unroll**	**nonOpt**	**pipelined**	**unroll**
ap_clk	Target	10.00	10.00	10.00	10.00	10.00	10.00
	Estimated	8.750	8.750	8.750	8.750	8.750	8.750

- **Latency (clock cycles)**

Table 14 - Comparative latency after high level synthesis - Green detection 720p

		No input image			With a I/O image		
		nonOpt	**pipeline**	**unroll**	**nonOpt**	**pipelined**	**unroll**
Latency	min	921636	921651	16635422	1843238	1843253	1843238
	max	10137616	921651	25851422	20275238	1843253	20275238

- **Utilisation estimates**

Table 15 - Comparative utilisation estimates after high level synthesis - Green detection 720p

	No input image			With a I/O image		
	nonOpt	**pipeline**	**unroll**	**nonOpt**	**pipelined**	**unroll**
BRAM_18K	2	2	2	2	2	2
DSP48E	0	0	10	1	1	1
FF	1709	2052	9155	2046	2387	2046
LUT	2162	2299	10943	2762	2859	2762

OPTIMISATION 1: 1280X720 / 1920X1080

A first comparison of performance estimates between the two resolutions confirms the expected results.

- Just like when optimising the 1080p code, the timing does not change when the resolution is changed.
- Both non-optimised and pipelined latency decrease by almost 10 when changing the resolution.

  - The minimum non-optimised latency decreases from **11 404 838** clock cycles to **921 636** clock cycles.
  - The minimum pipelined latency changes from **10 36 853** to **921 651** clock cycles.
- The utilisation estimates numbers can seem surprising since the amount of BRAMS, Flip Flops and Lookup Tables is more important for a lower resolution with image input than a high one with no image input. These results can be explained by the function loop, which takes an input image for a low resolution. The full HD code did not take an input image, which is why there is no need to store in memory. Using more memory to store the image input and output will increase the use of hardware.

Overall, decreasing the resolution from full HD (1080p) to HD (720p) increases the latency – which is the main goal of the design optimisation. The result will be a smoother game and smoother video output.
The context of the game will give a utilisation of hardware more similar to the result with an input image, since each frame will be processed. Using more hardware is not an issue for the project design as long as it respects the amount on the Programmable Logic (PL) block. Since the purpose of optimisation is to find the balance between a short latency and a reasonable utilisation estimates, the results given by the 720p High-Level synthesis seem satisfying.

OPTIMISATION 2: NON-OPTIMISED / PIPELINED

Table 16 - Comparative latency table for 720 pixels

		No input image			**With a I/O image**			**Optimised code**
		nonOpt	**pipeline**	**unroll**	**nonOpt**	**pipelined**	**unroll**	**pipelined**
Latency	min	921 636	921 651	16 635 422	1 843 238	1 843 253	1 843 238	1 617 215
	max	10 137 616	921 651	25 851 422	20 275 238	1 843 253	20 275 238	1 617 215

The written code was optimised (column on the right), by reducing the number of inputs. The latency given after high level synthesis is **1617215,** which is about 200 000 clock cycles less than a pipelined loop with input image. The latency is not the only factor influenced by this optimisation. By reducing the number of variables input to

1. In_data
2. Height
3. Color_threshold

The outputted black and white video will have a higher rate of frames per seconds (FPS). The reason behind this is that there will be less addresses calls.
The comparison with the column of optimised code therefore leads to conclude that the overall latency will not be influenced a lot by reducing input variables. However, it is expected that the display of the output will be smoother because it will reduce addresses calls.

By comparing the results for the 720p with image input code, it is possible to conclude that pipelining gives optimal results. Several trials led to conclude that the best initialisation interval is 2. The results above compare a non-optimised and pipelined loop.

1. The maximum latency is 200 000 clock cycles less for the pipelined version. It can be concluded that pipelining fixes the latency to a single amount of clock cycle. Because the minimum case is only one special image, the average latency will be lower for a pipelined version.

2. On the other hand, the minimum latency is higher for a non-optimised solution compared to a pipelined one, **18 43 238 >1 843 253**. The reason behind this result is the structure of the for loop. If the data goes through the if loop, it will take longer than the else (only one condition). For a full black picture, the entire data will pass through the else loop, leading to an "optimised" latency.

```
loop:for(int i = 0; i < 460799; i++){

            if(i%1280 < 640){

                    unsigned int current = in_hand[i];

                    out_data[i] = current;

                    unsigned char in_r = current & 0xFF;
                    unsigned char in_g = (current >> 8) & 0xFF;
                    unsigned char in_b = (current >> 16) & 0xFF;

                    if((in_g > (in_r + colour_threshold)) & (in_g > (in_b + colour_threshold))){

                            total_height += (720-i/1280); //i starts from the top, so the pixel
                            pixel_number ++;

                            out_data[i+460799] = 0xFFFFFF;
                    }

                    else{
                            out_data[i+460799] = 0;
                    }
            }


    }
```

Figure 15 - Screenshot of the colour detection loop

Comparing the operations per clock cycle for this optimised loop provides results in correlation with the explanations from above. This cannot be compared to the previous figure since the function shown is only determined for the pipelined loop.

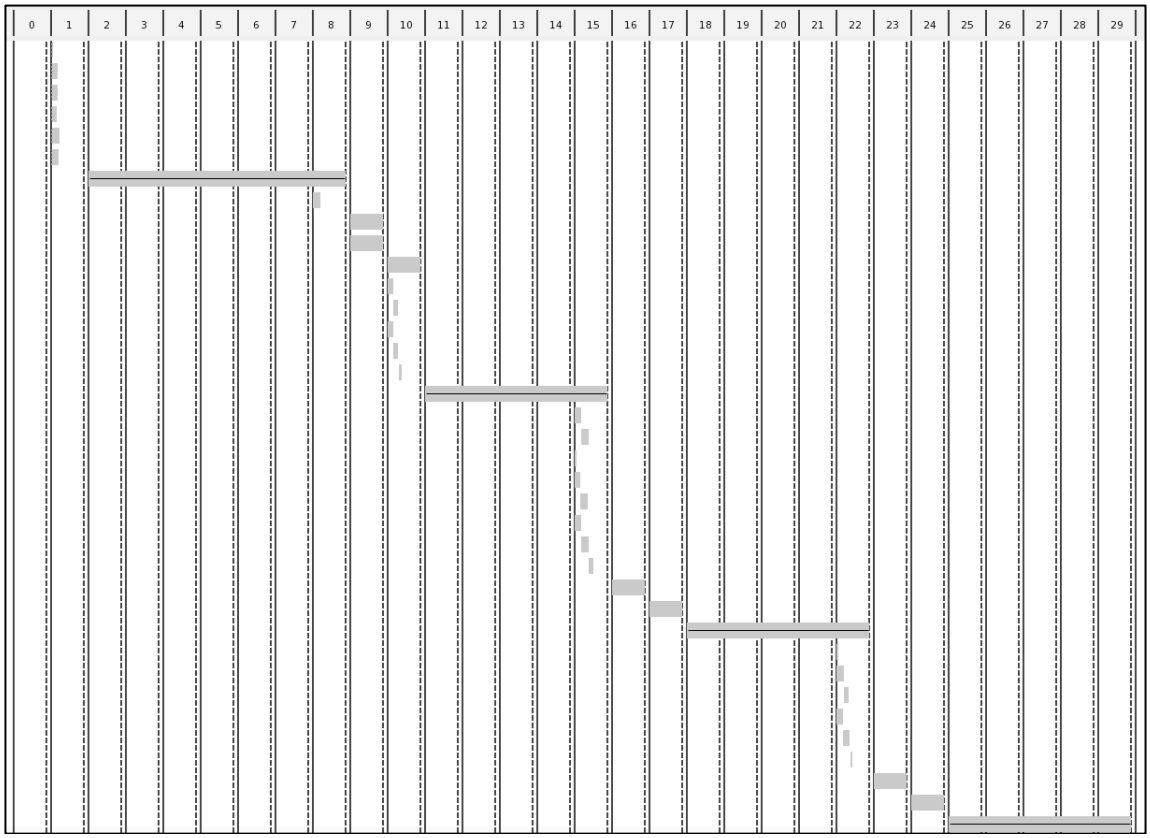


Figure 16 - Operation per clock cycle 720p 'for' loop (non-optimised)

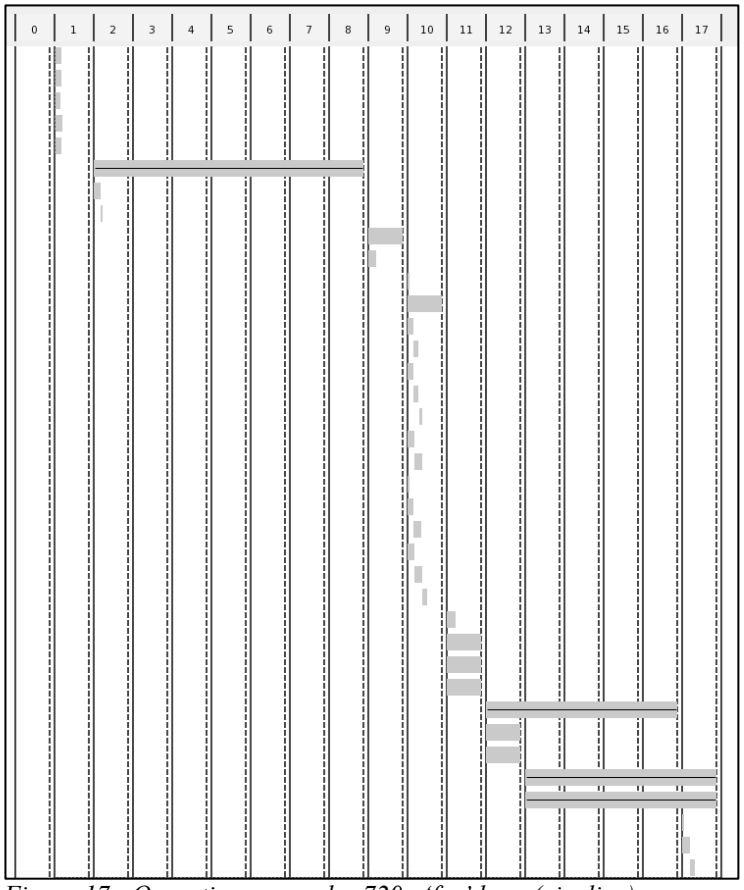


Figure 17 - Operation per cycles 720p 'for' loop (pipeline)

OPTIMISATION 3: SETTING PRECISION OF VARIABLES

The function heading for flap detector is the following:

```
void flap_detector(volatile uint32_t* in_hand, volatile uint32_t* out_data, uint16_t*
mean_height, bool* flap, bool* rise, uint8_t colour_threshold, uint8_t height_threshold){
```

Each variable precision depends on the values it can take.

1. Color_threshold is in the interval [0;255], since $255 < 2^8$, only 8 bits are required.
2. Height_threshold represents the minimum height difference to detect a flap. Since this value is particularly small, the amount of pixel will also be less than 2^8.

By reducing the precision of each variable, the overall latency is improved because less bits will have to go through the loops.

3. CREATING THE BLOCKS

Once the high-level synthesis is done, it is possible to create blocks using Vivado. The "Connection Automation Panel" allows the users to control the block connections. After the first synthesis, there were several errors to fix.

1920X1080 PIXELS

1. Even though the function was optimised as one loop only, the output block connections were checked and changed. By comparing the C function and block picture, it was possible to link each loop input and output.
2. For each block, some errors could be fixed by changing the clock source.

This first synthesis, for gave the following hardware utilisation.

Table 17 - 1080p logical synthesis : use of hardware

Total Power	Failed Routes	LUT	FF	BRAMs	URAM	DSP
		0	**0**	**0.00**	**0**	**0**
2.339	0	35266	51136	64.00	0	18

There might seem to be an important amount of LUTs and FF used to perform this single function. When compared to the utilisation estimates from the high-level synthesis, the amount of LUT and FF increase by 10. Whilst the utilisation estimates the amount of hardware for only the flap detector block, the synthesis generates the amount of LUT, FF, BRAMs and DSP for the whole design.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
flap_detector_0	s_axi_AXILiteS	Reg	0x83C2_0000	64K	0x83C2_FFFF

Table 18 - 1080p logical synthesis : offset address

Overall, this first synthesis is satisfying since the amount of hardware used respect the PL layout – it does not require more hardware than available – and the total power of 2.339***W?*** can be handled by the board.

1290X720 PIXELS

For a lower resolution, it is expected that less hardware will be used. When running the synthesis, the following table displays the use of hardware.

Table 19 - 720p logical synthesis : use of hardware

Total Power	Failed Routes	LUT	FF	BRAMs	URAM	DSP
		0	**0**	**0.00**	**0**	**0**
2.339	0	35054	50883	64.00	0	18

When comparing to the previous resolution, the following conclusions can be drawn:

1. The amount of LUT and FF decrease proportionally to the resolution. Using a 720pixels resolution saves 211 Lookup Tables and **253** Flip Flops. This is due to the fact less pixels are stored; the array of pixel is smaller.
2. Both programs use as many Read Only Memories (BRAMS) à and DSPs. Since it has to perform the same operations, the number of DSP will not change. Furthermore, the size of a BRAM is big enough so that a change from full HD to HD for such a small input video will only impact the internal use of the BRAM.
3. The total power is the same

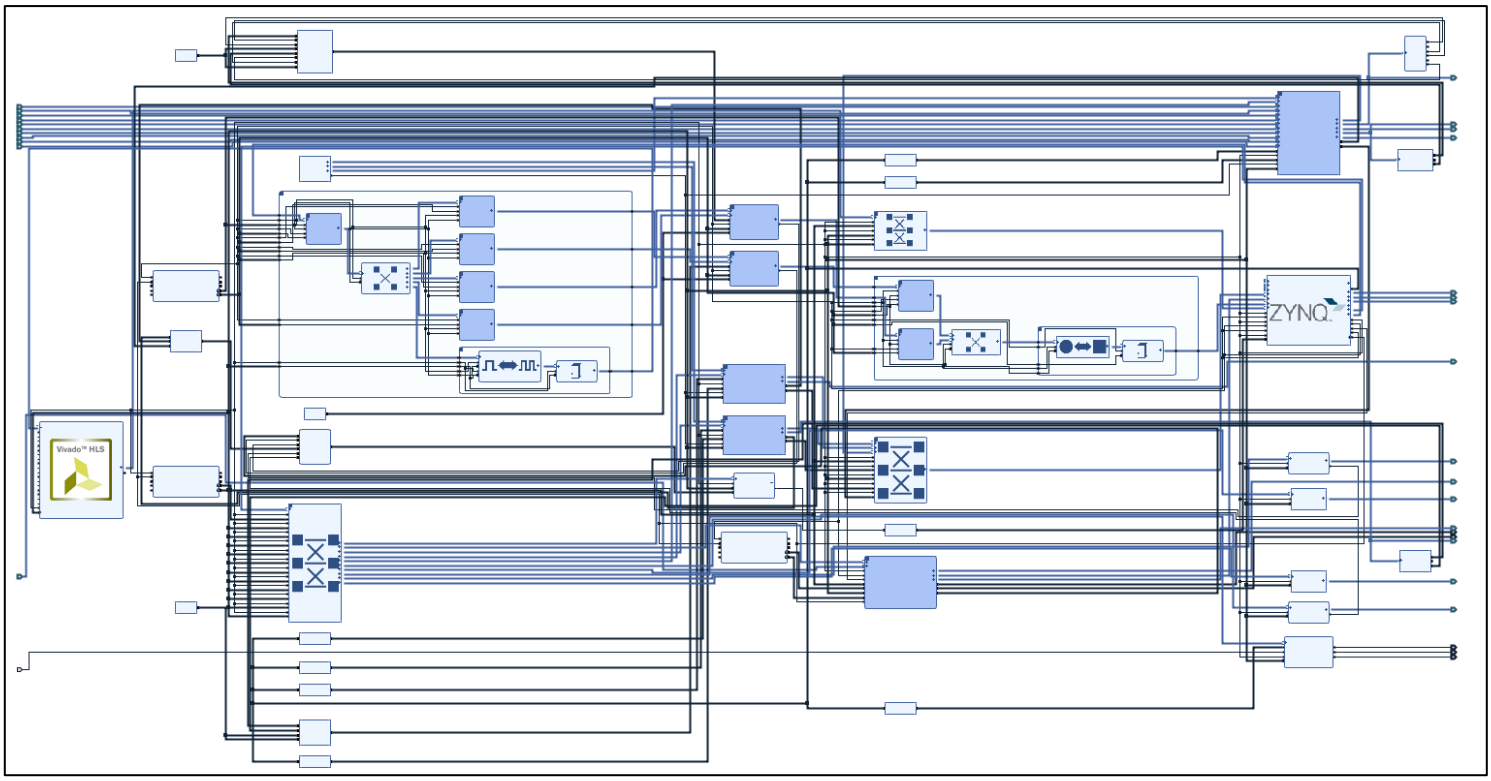


Diagram 13 - 720p logical synthesis : blocks connection

The specific addresses for each variable can be found in the appendix.

IV. DISPLAYING THE OUTPUT

1. FLAPPY BIRD IN PYTHON – USING OPENCV2

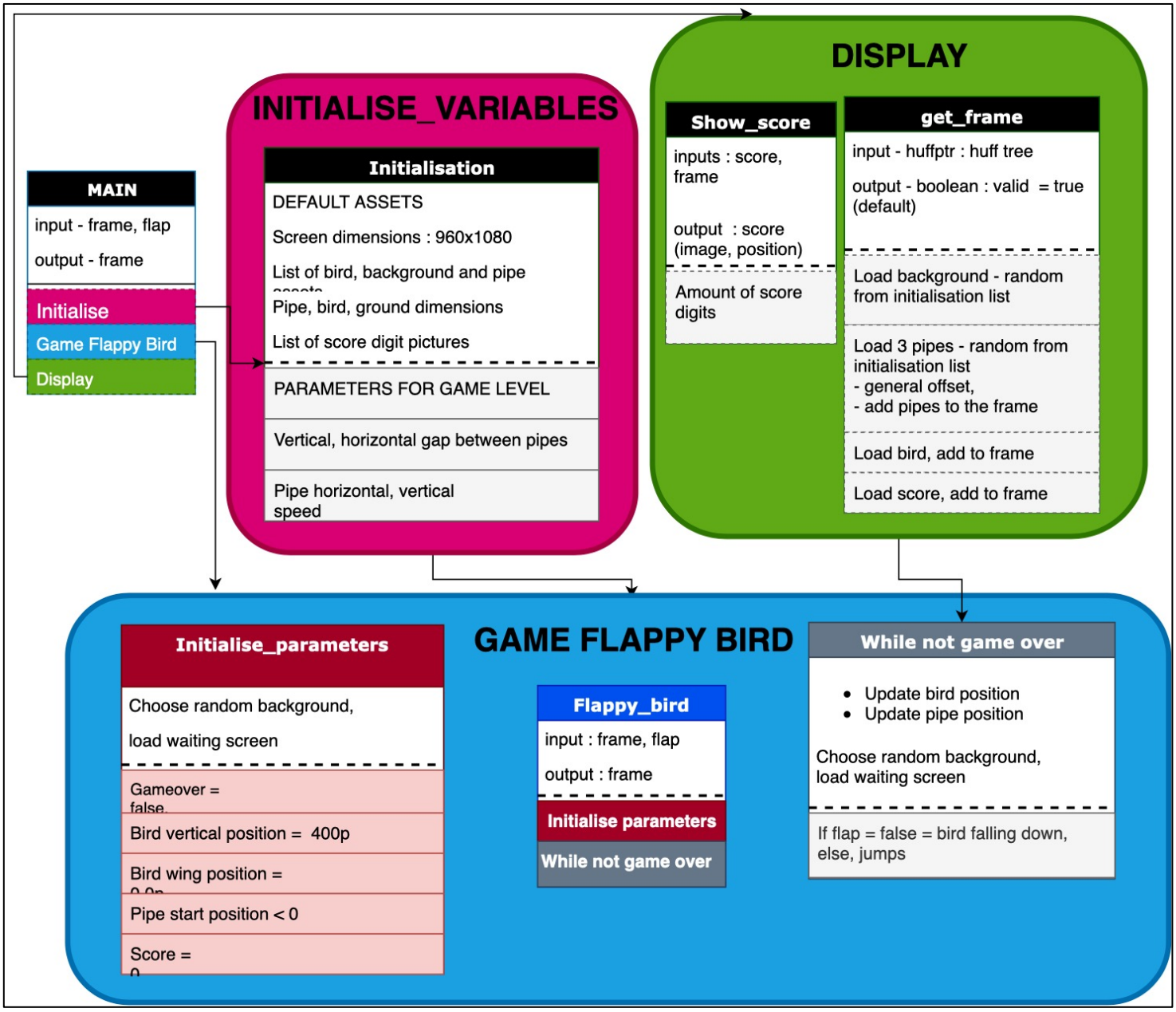


Diagram 14 - Python game code

In order to mimic the pipes 'moving', each clock cycle updates the position of the pipe. A third "virtual" pipe with a negative position appears to the right-hand side of the screen when one of the pipes reaches the screen left side.
Some parameters are set so they can change the game level. A narrow space between the pipes or a higher speed will make the game more difficult.
//The display of the bird, pipe and background is randomly generated from the input library of images.

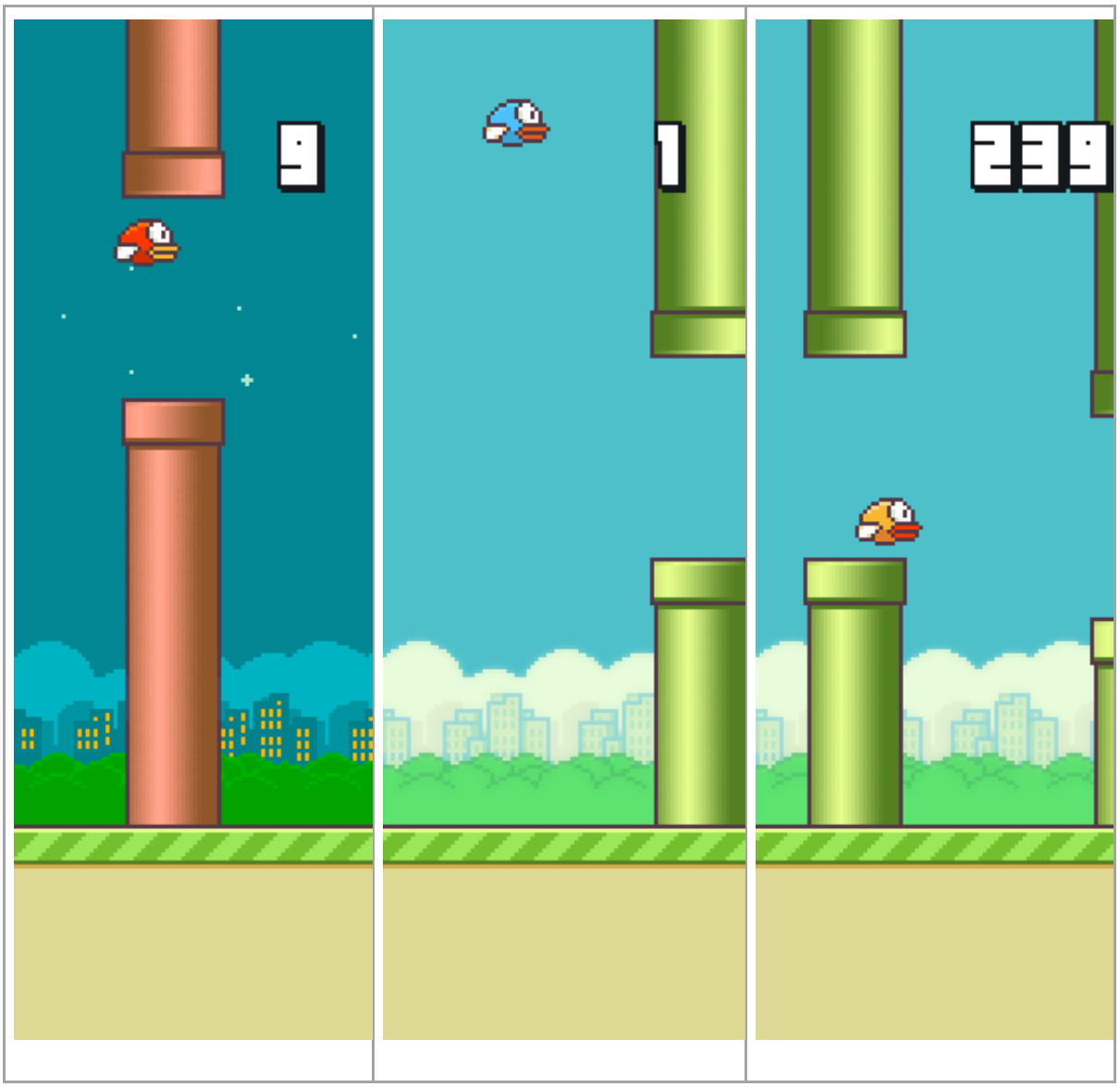


Figure 18 - Flappy Bird game output

To sum up, this code has the following specificities:

- Set parameters for display, which are taken randomly to generate different displays.
- Variable parameters that can be changed to level up the game,
- 1 “virtual” pipe that has a negative position, from which the modulo is taken in order to give randomise pipe appearing heights
- A main function to detect the game over.

The display part of the game could be done either on the FPGA or the CPU. In each case, each pipe has to be drawn pixel by pixel, as well as the shadows, colour changes and the bird. The main goal here is for the game to react in ‘real time’ with the user’s inputs moves.

Either:

1. The game will be outputted from the FPGA, but the CPU controls the different game positions. Since the FPGA already calculates the flap, the code will have to be re-written to avoid delay. The "flap" value will still be computed first but the game variables such as bird height, pipe horizontal position or game over, will also have to be updated at the same time.

Table 20 - Operations per frame for flappy bird game running on the FPGA

Operation	Frame n°1	Frame n°2	Frame n°2
User	Flaps hands	Waits	Flaps hands
FPGA	Default game values Flap = true	Update game values Flap = False	Default game values Flap = true
Game outputting	Bird falling	Bird flaps	Bird falling

2. The game can be run on the CPU, using OpenCV to draw the output. Since this will create 2 different functions, there will not be any delay. The only issue raised by this technique is the possible latency due to the use of software to display instead of the field programmable array, which is supposed to be faster.

Table 21 - Operations per frame for flappy bird game running on the CPU

Operation	Frame n°1	Frame n°2	Frame n°2
User	Flaps hands	Waits	Flaps hands
FPGA	Flap = true	Flap = False	Flap = true
CPU	Runs game Jump update for variables	Runs game Default update for variables	Runs game Jump update for variables
Game outputting	Bird flaps	Bird falling	Bird flaps

After several tests, the following comparison led to the second choice:

- Drawing 2 rectangles using OpenCV reduced the frame rate by 1.1fps[13] in comparison to using the FPGA.
- Calling 10 extra parameters with OpenCV reduced the frame rate by 0.ll fps.

It therefore seems that running the game on the CPU is fast enough to make the output seem like real time.

Because of the efficiency and simplicity of using OpenCV, the game will run on the CPU. Furthermore, this will avoid a delay in the game between the user "flapping" their hand and the bird's flap.
In order to make the game intuitive, it will be possible to delay the output flap so the bird's flap and the user lowering his hand are synchronised.

[13] FPS : Frame per second, // find proper definition

VISUAL COMPARISONS OF THE OPERATIONS PER FRAME EVOLUTION

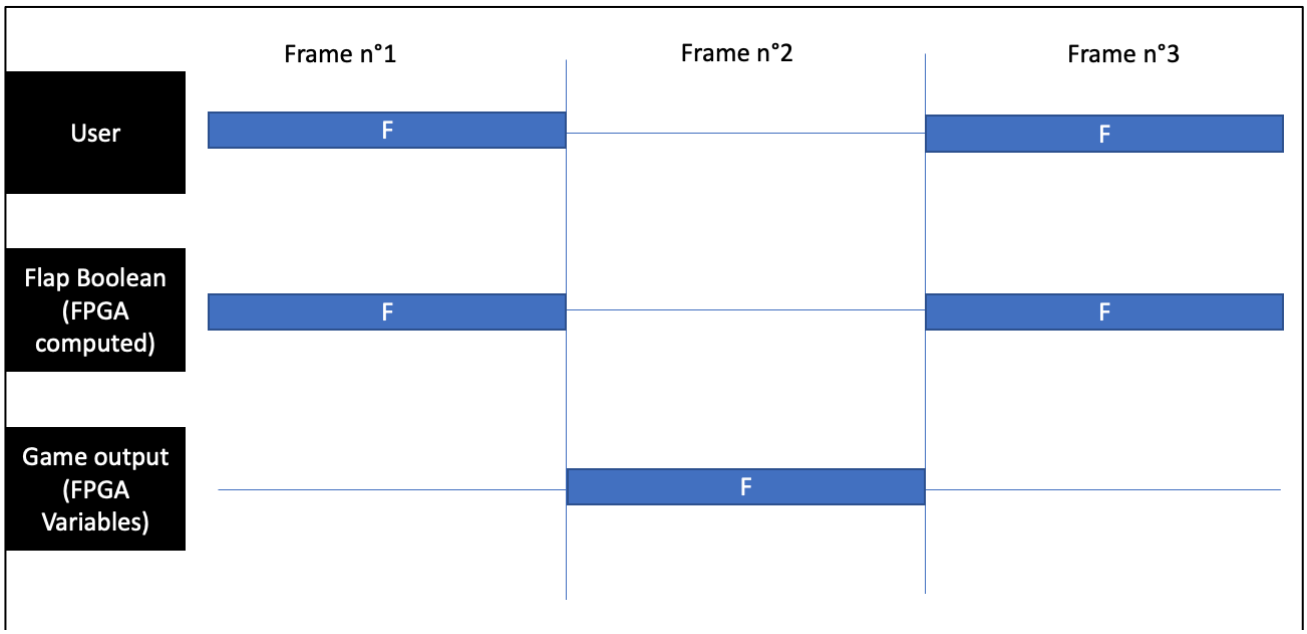


Diagram 15 - Operations per frame for flappy bird game running on the FPGA

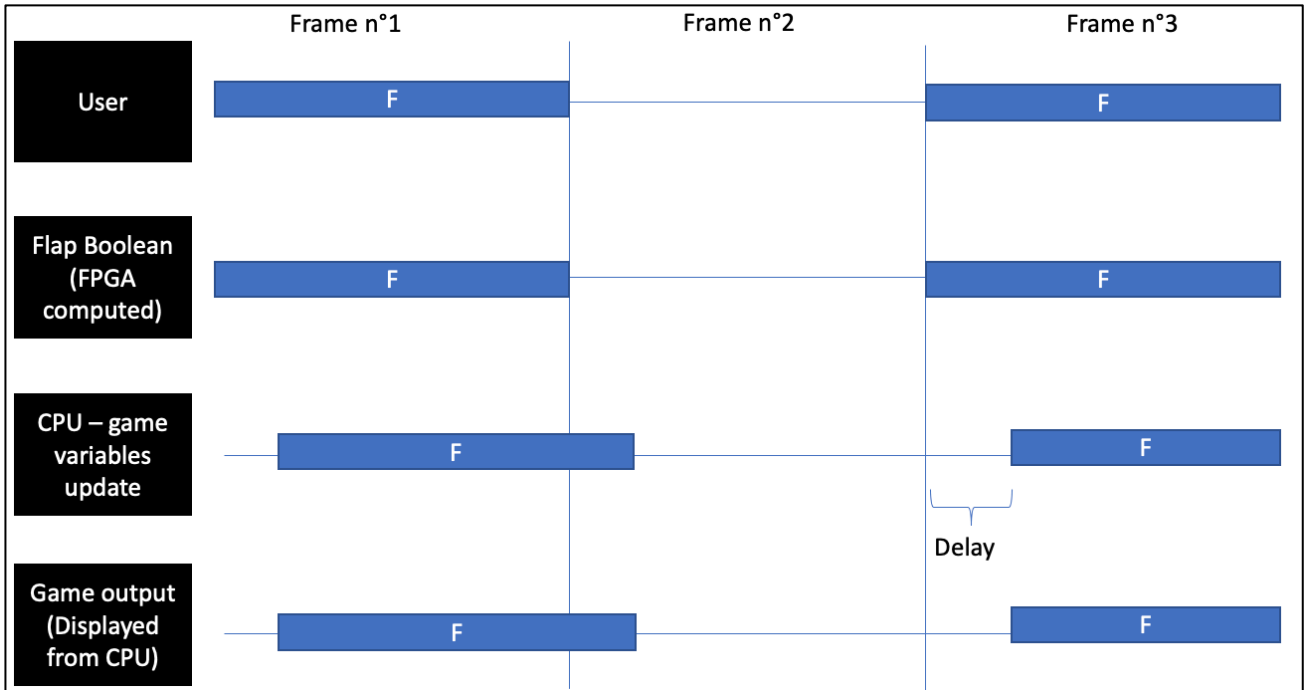


Diagram 16 - Operations per frame for flappy bird game running on the CPU

As the diagram above depicts, there might be a delay due to the use of the CPU to run the game, which might lead to a slower frame.

2. DISPLAYING ON SCREEN

The main goal when displaying the output is to make it as user friendly as possible. In order the achieve this constraint, the screen design was studied using skills from User-Centred Information System courses[14]. Showing the camera input as well as the detection output allows the users to familiarise with the environment and understand for themselves the feedback of their different movements on the detection.
By not including the camera results, the following ambiguities could have been happened:

- The user has no tool to make sure if they are inside the input video frame
- The user cannot directly link the black and white output from the detection image with their movements.

Furthermore, several input video games such as Just Dance[15] give on the top left corner the output of the detected body.

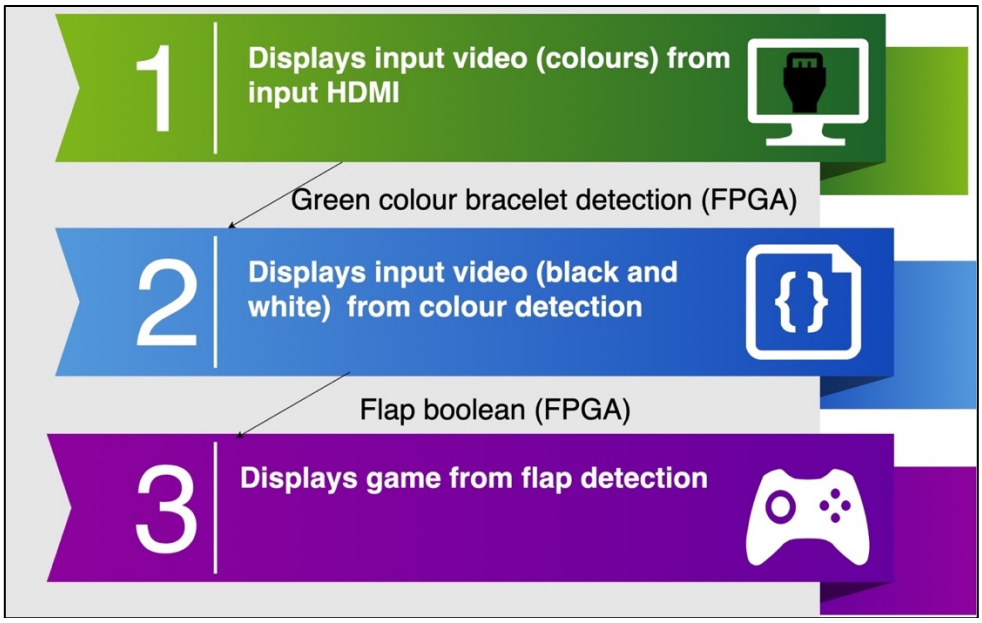


Figure 19 - Displaying the output on screen

In order to ease the process of displaying the output, it was chosen to scan the input video from on the left corner.

- This technique avoids having to scan the whole array from the input HDMI.
- Since the output black and white with the detected colour will have the same size as the input with colours, there is no need to resize the array of pixels, which saves times and improve latency.
- This leave the right-hand half for the game display. The game has constant parameters, it is therefore easier to resize it from a mobile phone size (1024x728) to half of a computer screen.

The diagram above sums up the process for the display of the output game

[14] Source: Spence R, 2006, Information visualization: design for interaction, ISBN: 9780132065504

[15] Game for xbox 360, *add reference*

FUTURE WORK

For further work, several ideas could not be implemented in the time span of the project. These features, once added, will lead to a user friendly and smoother game.

INTERFACE

It is possible to add a main interface where users can select levels of difficulties, personalise the output screen. Since the Python game already has a set of different birds, background and pipes to choose from, the users would be able to define their own game design.
Furthermore, adding an interface would allow adding different game levels. Such implementation could be done through the selection of several python files which could display different background at different speed.
Rising the speed of the background on the screen output in order to make the same level more difficult.

SOUND EFFECTS

Since the goal of this project is to create a video input flappy bird, one way to make the game more similar to the commercial one would be to add sound effects.
This can be done by adding an Arduino sound output connected with the CPU. The python game code could therefore control sound effect at the same time as the game is outputted on the screen.

CHANGING THE THEME TO LONDON

One of the ideas was to change the pipes from the game to pictures of big ben or of subway and create a London Background in order to make this game unique. Since the display relies on a library of pictures, the possibilities are infinite. It was also discussed to do a “Imperial College” display with a Bird jumping over queen towers with a Background of Sherfield Building with a score count next to the Imperial blazon.

CONCLUSION

To conclude, this project has led to conclude that a movement recognition using a FPGA with an input video from a HDMI stream can be optimised by using colour recognition.
After several trials and tests, it has been proved that the green colour is one of the optimal in comparison to different shades of skin colour.
Furthermore, for the colour recognition, using RGB detection is easier, less costly in operations and more effective than other, more recent pixel combinations such as HLV.

The purpose of this project was to implement a real time flappy bird game with user interaction. It was met by several optimisations, such as the decision to lower the resolution of the output screen or scanning the input video without resizing its black and white scanned input.

Another goal of this project was to make it understandable by the user. The specific layout of the output with the video outputting the colour recognition allows them to get an insight of how their movements are detected as well as getting an insight of how the game works.

The entire detection and flap Boolean variable output run on the FPGA, which allows the black and white video output to have 19.3 Frames per seconds.

To sum up, the design as a whole was optimised at different level in order to get a short latency as well as getting a high value of frame per seconds and avoid delay.

1. Code level:
    a. Combining in one function green detection and flap detection,
    b. Reducing the number of variables,
    c. Using RGB for colour detection over HLS.
2. High-level synthesis level:
    a. Reducing the resolution from 1080p to 720p,
    b. Pipelining the for loop with factor 2,
    c. Setting the variable precisions depending on the range of values they can take.
3. Block synthesis level:
    a. Fixing the clocks?
4. Design decisions level:
    a. Running the flappy bird game on the CPU over the FPGA to improve fps and avoid delay,
    b. Output the colour video and green detection with the same size to avoid to have to resize each frame.

ADD FINAL PICTURE OF OUTPUT

APPENDIX

LINK TO GITHUB (WITH ALL THE CODES)

https://github.com/JaafarRammal/FPGA_Year1_Project.git

2. FIGURES

1. REFERENCES

(1) Xilinx, Vivado Design Suite Tutorial , *High-Level Synthesis UG871,* v2018.2, June 6, 2018
(2) Xilinx, Pynq Productivity for Zynq (Documentation), Release 2.2, Sep 19, 2018
(3) Allen B. Downey, Think Python: How to Think Like a Computer Scientist, 2nd edition, 2015
(4) Peter Prinz, Ulla Kirch-Prinz, C Pocket Reference: C Syntax and Fundamentals, 2002
(5) https://www.bbc.co.uk/news/technology-26114364
(6) https://opencv.org/about/
(7) https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf
(8) https://www.xilinx.com/products/silicon-devices/fpga.html
(9) http://www.pynq.io
(10) https://www.quora.com/
(11) https://www.lifewire.com/

3. DEFINITIONS

FPGA

A field-programmable gate array (FPGA) is an integrated circuit that can be programmed in the field after manufacture. FPGAs are similar in principle to, but have vastly wider potential application than, programmable read-only memory chips.

Open cv

OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products. Being a BSD-licensed product, OpenCV makes it easy for businesses to utilize and modify the code.

The library has more than 2500 optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. These algorithms can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, track moving objects, extract 3D models of objects, produce 3D point clouds from stereo cameras, stitch images together to produce a high resolution image of an entire scene, find similar images from an image database, remove red eyes from images taken using flash, follow eye movements, recognize scenery and establish markers to overlay it with augmented reality, etc. OpenCV has more than 47 thousand people of user community and estimated number of downloads exceeding 18 million. The library is used extensively in companies, research groups and by governmental bodies.

PYNQ board

PYNQ is an open-source project from Xilinx that makes it easy to design embedded systems with Zynq Systems on Chips (SoCs). Using the Python language and libraries, designers can exploit the benefits of programmable logic and microprocessors in Zynq to build more capable and exciting embedded systems.

RTL (register transfer language)

In digital circuit design, register transfer level (RTL) is a design abstraction which models a synchronous digital circuit interms of the flow of digital signals (data) between hardware registers, and the logical operations performed on those signals.
Register-transfer-level abstraction is used in hardware description languages (HDLs) like Verilog and VHDL and to create high-level representations of a circuit, from which lower-level representations and ultimately actual wiring can be derived. Design atthe RTL level is t ypical practice in modern digital design.

Latency

The time interval between initiating a query, transmission, or process, and receiving or detecting the results, oftengiven as an average value over a large number of events.

Fps

Frames per second (FPS) is a unit that measures display device performance. It consists of the number of complete scans of the display screen that occur each second. This is the number of times the image on the screen is refreshed each second, or the rate at which an imaging device produces unique sequential images called frames.

4. SOME ANNOTATED CODES (FOR FURTHER UNDERSTANDING)

PYTHON DRAFT CODE FOR SKIN COLOR DETECTION

The initialization code:

```
#####################################
######
def get_means(img, x, y, h, w):

acc_r = 0
acc_g = 0
acc_b = 0
min_r = 255
max_r = 0
min_g = 255
max_g = 0
min_b = 255
max_b = 0
for i in range(h):
for j in range(w):
r = img[y+i, x+j][0]
g = img[y+i, x+j][1]
b = img[y+i, x+j][2]
acc_r = acc_r + r
acc_b = acc_b + b
acc_g = acc_g + g
if r>max_r:
max_r = r
if r<min_r:
min_r = r
if g>max_g:
max_g = g
if g<min_g:
min_g = g
if b>max_b:
max_b = b
if b<min_b:
min_b = b
c = (h*w)
acc_r = acc_r / c
acc_g = acc_g / c
acc_b = acc_b / c
return acc_b, acc_g, acc_r, max_b-min_b,
max_g-min_g, max_r-min_r
#####################################
######
```

In the code, H and W are the frame size. X and Y are the initialization box boundaries

The detection code:

```
#####################################
######

def transfrom_image(img, height, width,
thresholds):

counter = 0
mean_height = 0
output = img.copy()
for row in range(height):
for col in range(width):
acc_b, acc_g, acc_r, t_b, t_g, t_r =
thresholds
r, g, b = img[col, row]
if abs(r-int(acc_r)) > t_r/6 or abs(g-
int(acc_g)) > t_g/6 or abs(b-int(acc_b)) >
t_b/6:
#if 100*g<r*60 or 100*g>r*90 or
100*b<r*50 or 100*b>r*90:
output[col, row] = 0,0,0
else:
output[col, row] = 255,255,255
mean_height = mean_height + row
counter = counter + 1
mean_height = mean_height /
max(counter,1)
return output, mean_height

#####################################
######
```

5. ADDRESSES FROM LOGICAL SYNTHESIS

1080p

```
//------------------------Address Info-------------------
// 0x00 : Control signals
//        bit 0  - ap_start (Read/Write/COH)
//        bit 1  - ap_done (Read/COR)
//        bit 2  - ap_idle (Read)
//        bit 3  - ap_ready (Read)
//        bit 7  - auto_restart (Read/Write)
//        others - reserved
// 0x04 : Global Interrupt Enable Register
//        bit 0  - Global Interrupt Enable (Read/Write)
//        others - reserved
// 0x08 : IP Interrupt Enable Register (Read/Write)
//        bit 0  - Channel 0 (ap_done)
//        bit 1  - Channel 1 (ap_ready)
//        others - reserved
// 0x0c : IP Interrupt Status Register (Read/TOW)
//        bit 0  - Channel 0 (ap_done)
//        bit 1  - Channel 1 (ap_ready)
//        others - reserved
// 0x10 : Data signal of in_data
//        bit 31~0 - in_data[31:0] (Read/Write)
// 0x14 : reserved
// 0x18 : Data signal of out_data
//        bit 31~0 - out_data[31:0] (Read/Write)
// 0x1c : reserved
// 0x20 : Data signal of w
//        bit 31~0 - w[31:0] (Read/Write)
// 0x24 : reserved
// 0x28 : Data signal of h
//        bit 31~0 - h[31:0] (Read/Write)
// 0x2c : reserved
// 0x30 : Data signal of mean_height_i
//        bit 31~0 - mean_height_i[31:0] (Read/Write)
// 0x34 : reserved
// 0x38 : Data signal of mean_height_o
//        bit 31~0 - mean_height_o[31:0] (Read)
// 0x3c : Control signal of mean_height_o
//        bit 0  - mean_height_o_ap_vld (Read/COR)
//        others - reserved
// 0x40 : Data signal of flap
//        bit 0  - flap[0] (Read)
//        others - reserved
// 0x44 : Control signal of flap
//        bit 0  - flap_ap_vld (Read/COR)
//        others - reserved
// 0x48 : Data signal of rise_i
//        bit 0  - rise_i[0] (Read/Write)
//        others - reserved
// 0x4c : reserved
// 0x50 : Data signal of rise_o
//        bit 0  - rise_o[0] (Read)
//        others - reserved
// 0x54 : Control signal of rise_o
//        bit 0  - rise_o_ap_vld (Read/COR)
//        others - reserved
// 0x58 : Data signal of colour_threshold
//        bit 31~0 - colour_threshold[31:0] (Read/Write)
// 0x5c : reserved
// 0x60 : Data signal of height_threshold
//        bit 31~0 - height_threshold[31:0] (Read/Write)
// 0x64 : reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)
```

720p

```
//------------------------Address Info-------------------
// 0x00 : Control signals
//        bit 0  - ap_start (Read/Write/COH)
//        bit 1  - ap_done (Read/COR)
//        bit 2  - ap_idle (Read)
//        bit 3  - ap_ready (Read)
//        bit 7  - auto_restart (Read/Write)
//        others - reserved
// 0x04 : Global Interrupt Enable Register
//        bit 0  - Global Interrupt Enable (Read/Write)
//        others - reserved
// 0x08 : IP Interrupt Enable Register (Read/Write)
//        bit 0  - Channel 0 (ap_done)
//        bit 1  - Channel 1 (ap_ready)
//        others - reserved
// 0x0c : IP Interrupt Status Register (Read/TOW)
//        bit 0  - Channel 0 (ap_done)
//        bit 1  - Channel 1 (ap_ready)
//        others - reserved
// 0x10 : Data signal of in_hand
//        bit 31~0 - in_hand[31:0] (Read/Write)
// 0x14 : reserved
// 0x18 : Data signal of out_data
//        bit 31~0 - out_data[31:0] (Read/Write)
// 0x1c : reserved
// 0x20 : Data signal of mean_height_i
//        bit 15~0 - mean_height_i[15:0] (Read/Write)
//        others   - reserved
// 0x24 : reserved
// 0x28 : Data signal of mean_height_o
//        bit 15~0 - mean_height_o[15:0] (Read)
//        others   - reserved
// 0x2c : Control signal of mean_height_o
//        bit 0  - mean_height_o_ap_vld (Read/COR)
//        others - reserved
// 0x30 : Data signal of flap
//        bit 0  - flap[0] (Read)
//        others - reserved
// 0x34 : Control signal of flap
//        bit 0  - flap_ap_vld (Read/COR)
//        others - reserved
// 0x38 : Data signal of rise_i
//        bit 0  - rise_i[0] (Read/Write)
//        others - reserved

// 0x3c : reserved
// 0x40 : Data signal of rise_o
//        bit 0  - rise_o[0] (Read)
//        others - reserved
// 0x44 : Control signal of rise_o
//        bit 0  - rise_o_ap_vld (Read/COR)
//        others - reserved
// 0x48 : Data signal of colour_threshold
//        bit 7~0 - colour_threshold[7:0] (Read/Write)
//        others  - reserved
// 0x4c : reserved
// 0x50 : Data signal of height_threshold
//        bit 7~0 - height_threshold[7:0] (Read/Write)
//        others  - reserved
// 0x54 : reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)
```