DEGGENDORF INSTITUTE OF TECHNOLOGY

Master of Applied Computer Science

Embedded Acceleration

Summer Term 2021

Hardware Accelerated Steganography

Group 17

Vipin Koshy Thomas
Majd Victor Hafiri

Date 05-07-2021

Contents

Overview and objectives

Steganography is a method of hiding secret information in digital files such as images, audio files etc. In this project , we implemented an FPGA based solution to hide a secret word in images by changing some pixel values at some positions without being detected by human eyes. Moreover, it can also reveal the hidden word in an encoded image.

The goal of our project is to perform the steganography technique faster using the FPGA and the CPU together rather than the Software only solution. We have also implemented a Software Only solution which is used to compare the execution time and speed ups between both.

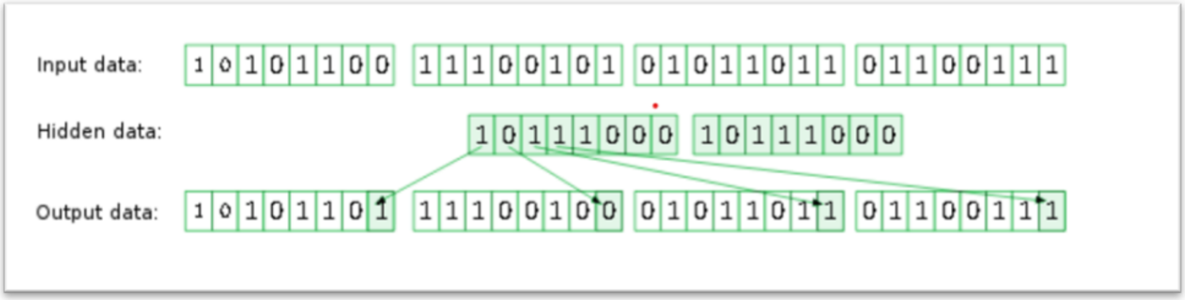
LSB Manipulation Algorithm

Images are made up of tiny pixels where each pixel consists of 3 values which are Red, Green and Blue colors. Any changes on RGB values will make the image look different, Unless these changes to the pixel values are very small. In Steganography, hiding a secret data in the pixels of an image , is all about changing the Least significant bit of Red or Green or Blue value of a Pixel, and changing the LSB of the value won’t be visible to Human eyes and The image will look untouched. However, It can be detected using Software and Hardware solutions.

We have created a Least significant bit (LSB) manipulation algorithm for our project which changes and calculates the LSB of RGB values for the selected pixels (position).

How it works

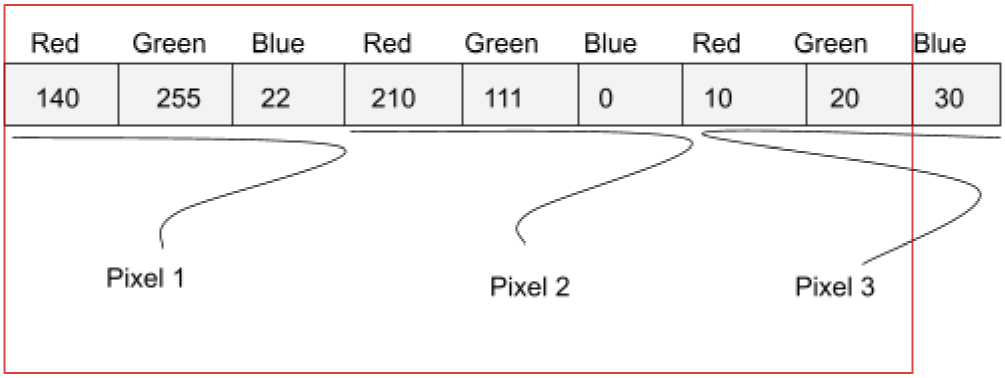
For concealing the word in pixel values , the algorithm takes the decimal value and strips it to integer numbers of two digits. Each 2 digit value represents a character of the word. For every 2 digit value , the algorithm converts this number to a binary value of 8 bit and for each bit , the algorithm checks one value of the passed data (Pixel RGB values) if it's even or odd. If the bit is 1 and data is even ,it changes the LSB of the value to 1 , and if the bit is 0 and the data is odd , it changes the LSB of the value to 0 , otherwise it remains the same.



Input data:	10101100	11100101	01011011	01100111
Hidden data:		10111000	10111000	
Output data:	10101101	11100100	01011011	01100111

Ref: https://www.irjet.net/archives/V3/i12/IRJET-V3I1247.pdf

the structure of data of pixels looks like the following:



Red	Green	Blue	Red	Green	Blue	Red	Green	Blue
140	255	22	210	111	0	10	20	30

8 values = 8 bits = 1 character

This process continues for the other characters until it reaches the end of the data stream and the last character in the secret word.

For decoding and revealing the secret word , the algorithm checks the Least significant bit of the data stream and finds the binary value for every 8 data packets. When this process is done , it writes the decimal value for the character to be used by the software side.

Steganography IP

The hardware accelerated solution focuses on the LSB manipulation on the input image constrained by the positions provided through any key exchange algorithms like Diffie Hellman(not explained here, out of scope) using FPGA. We have used Vitis HLS to process the C++ pixel function to produce solutions for Vivado.

We are using HLS streams to send and receive image data using send channel and receive channel. Other than that AXI Lite bus is used to set the values of Registers, which will be used as the controlling parameters on the process. We are not using No block-level I/O protocol and using the reference pointer to write the decoded value to the registers.

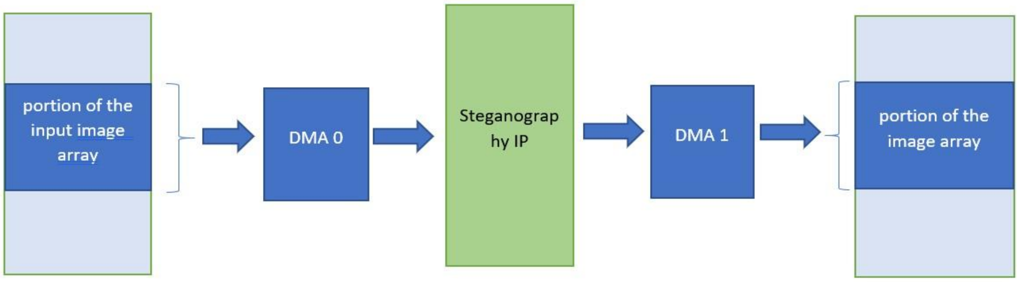


Figure 1

Two Direct Memory Access blocks were used in the project. DMA0 is used for input and DMA1 is used for the output(Figure 1), which are connected to High performance port HP0 and HP1 respectively through an AXI Interconnect(Figure 2).

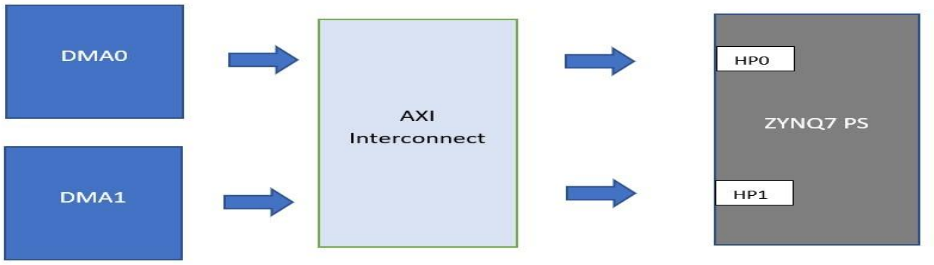
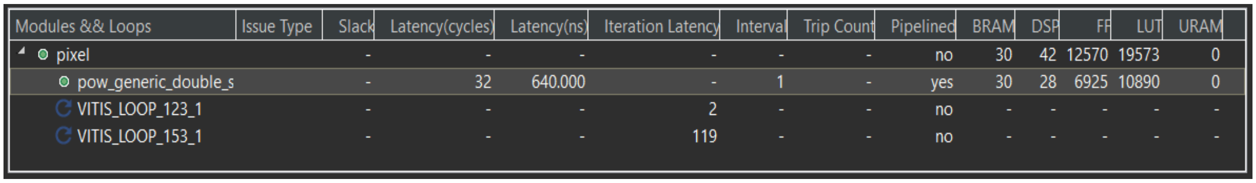


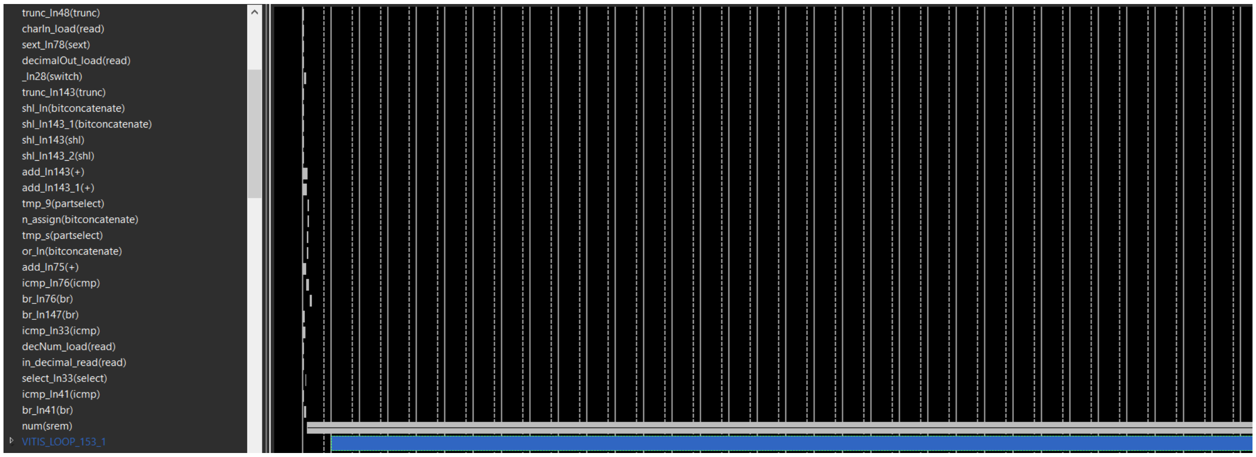
Figure 2

Schedule Viewer

In the CPP code some functions are used to convert integer to Binary and Binary to Integer. The functions use loop which are unrolled and using #pragma HLS unroll which will help the FPGA to run the unrolled loop which can be run in parallel.

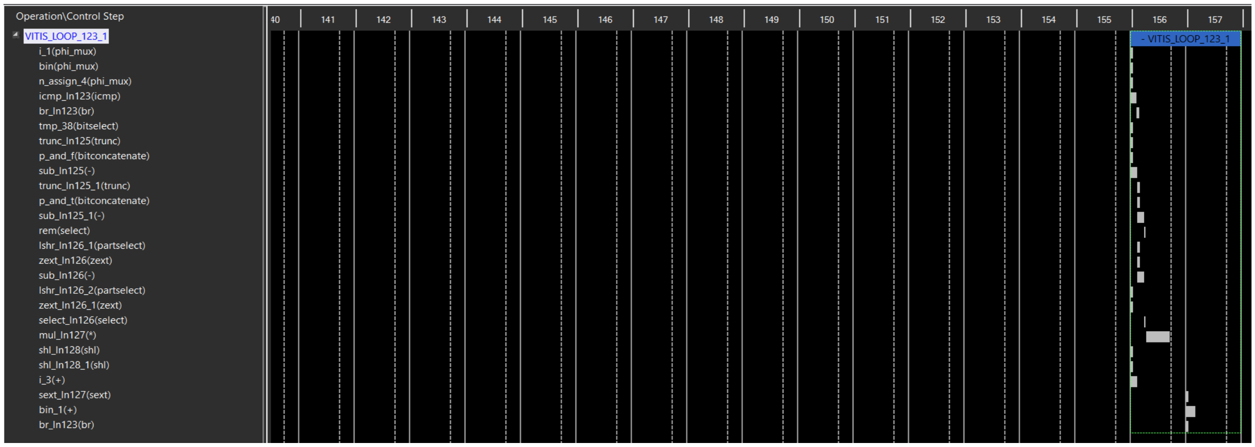


Modules && Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
pixel		-	-	-	-	-	-	no	30	42	12570	19573	0
pow_generic_double_s		-	32	640.000	-	1	-	yes	30	28	6925	10890	0
VITIS_LOOP_123_1		-	-	-	2	-	-	no	-	-	-	-	-
VITIS_LOOP_153_1		-	-	-	119	-	-	no	-	-	-	-	-

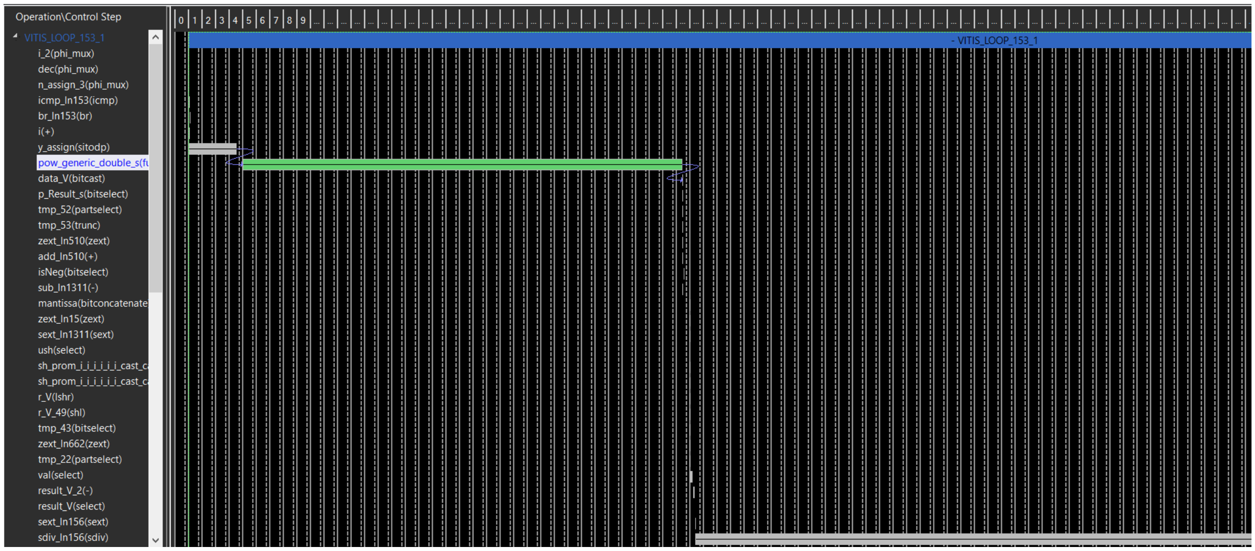


```
trunc_ln48(trunc)
charIn_load(read)
sext_ln78(sext)
decimalOut_load(read)
_ln28(switch)
trunc_ln143(trunc)
shl_ln(bitconcatenate)
shl_ln143_1(bitconcatenate)
shl_ln143(shl)
shl_ln143_2(shl)
add_ln143(+)
add_ln143_1(+)
tmp_9(partselect)
n_assign(bitconcatenate)
tmp_s(partselect)
or_ln(bitconcatenate)
add_ln75(+)
icmp_ln76(icmp)
br_ln76(br)
br_ln147(br)
icmp_ln33(icmp)
decNum_load(read)
in_decimal_read(read)
select_ln33(select)
icmp_ln41(icmp)
br_ln41(br)
num(srem)
VITIS_LOOP_153_1
```

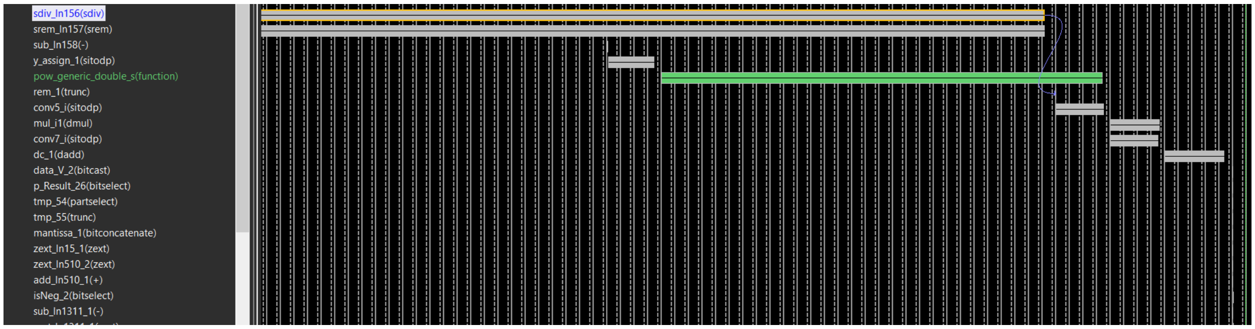
Read operation and first loop



Schedule Viewer for integer to binary conversion-function loop

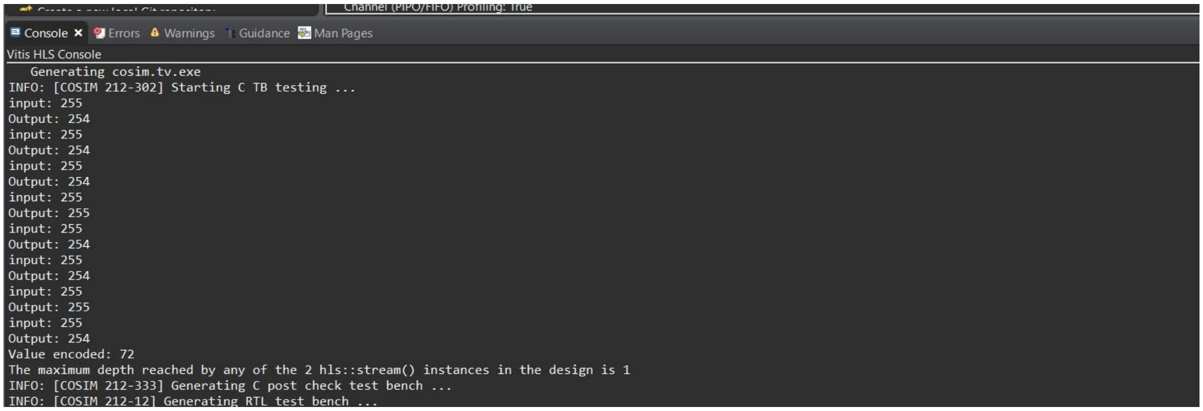


Schedule Viewer for binary to integer conversion-function loop



Test benches

Debugging and co simulation options have been used to check the correctness of the pixel function and improve it, out of those can be found below.



```
   Generating cosim.tv.exe
INFO: [COSIM 212-302] Starting C TB testing ...
input: 255
Output: 254
input: 255
Output: 254
input: 255
Output: 254
input: 255
Output: 255
input: 255
Output: 254
input: 255
Output: 254
input: 255
Output: 255
input: 255
Output: 254
Value encoded: 72
The maximum depth reached by any of the 2 hls::stream() instances in the design is 1
INFO: [COSIM 212-333] Generating C post check test bench ...
INFO: [COSIM 212-12] Generating RTL test bench ...
```

Explanation:

Value to Encode: 72
Binary value of 72: 01001000
Stream Count = 8(stream size)
Test Input array : [255,255, 255, 255, 255, 255, 255, 255]

Encoded Output array is generated as follows

1) 0100100 | 0(LSB) => 255(11111111) => 254(11111110)(updated value with LSB =0)

2) 010010 | 0(LSB) => 255(11111111) => 254(11111110)(updated value with LSB =0)

3) 01001 | 0(LSB) => 255(11111111) => 254(11111110)(updated value with LSB =0)

4) 0100 | 1(LSB) => 255(11111111) => 255(11111111)(updated value with LSB =1)

5) 010 | 0(LSB) => 255(11111111) => 254(11111110)(updated value with LSB =0)

6) 01 | 0(LSB) => 255(11111111) => 254(11111110)(updated value with LSB =0)

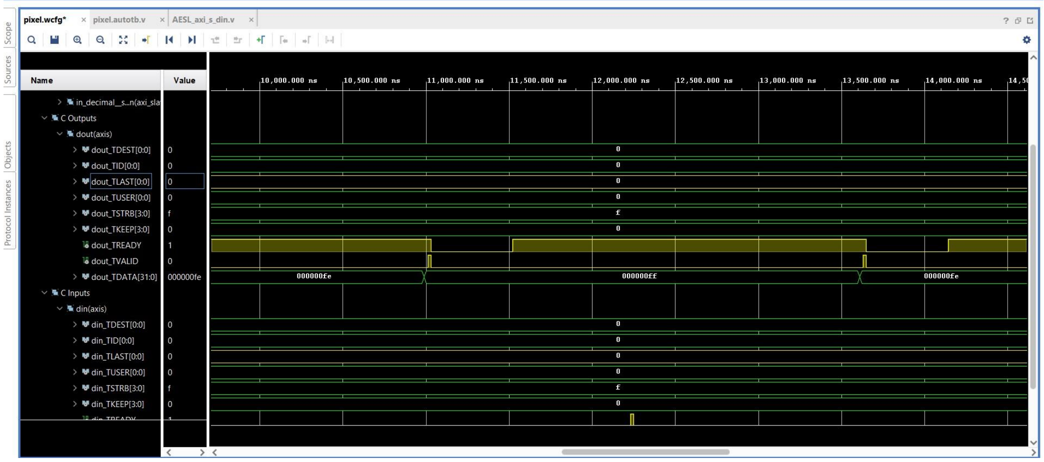
7) 0 | 1(LSB) => 255(11111111) => 255(11111111)(updated value with LSB =1)

8) | 0(LSB) => 255(11111111) => 254(11111110)(updated value with LSB =0)

So the output array will be [254,254,254,255,254,254,255,254] which is the same as the CSIM output.

Co-Simulation outputs show the encoded value and output array. We have used main.cpp to call the pixel function, which is given in the Test bench directory of the git repo along with all other necessary files.

Wave Debug diagram on Vivado:



Name	Value
in_decimal_s...n(axi_sla	
C Outputs	
dout(axis)	
dout_TDEST[0:0]	0
dout_TID[0:0]	0
dout_TLAST[0:0]	0
dout_TUSER[0:0]	0
dout_TSTRB[3:0]	f
dout_TKEEP[3:0]	0
dout_TREADY	1
dout_TVALID	0
dout_TDATA[31:0]	000000fe
C Inputs	
din(axis)	
din_TDEST[0:0]	0
din_TID[0:0]	0
din_TLAST[0:0]	0
din_TUSER[0:0]	0
din_TSTRB[3:0]	f
din_TKEEP[3:0]	0
din_TREADY	1

Wave Debug diagrams can be used to check the internal signals and data flow with respect to clock cycles, which can be used to improve the circuit.

Block diagram

https://mygit.th-deg.de/vt16684/embedded-acceleration/-/blob/master/Steganography%20IP%20Block%20Diagram.pdf

Jupyter Notebook

https://mygit.th-deg.de/vt16684/embedded-acceleration/-/blob/master/Pixel_Image_Steganography_1_.ipynb

Timing and speedup comparison

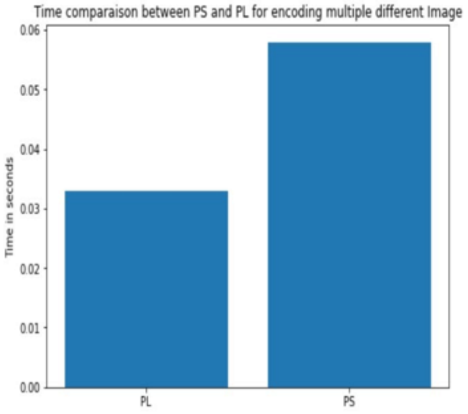
We were testing the IP with different clock frequencies namely 50,100,150,200 MHz, for the final IP synthesis we have used a clock frequency of 150MHz. During the Timing analysis we have found out time taking for encoding and decoding the secret word is indirectly proportional to the number of characters in the word, i.e. embedded accelerated solution performs better when number of characters increases.

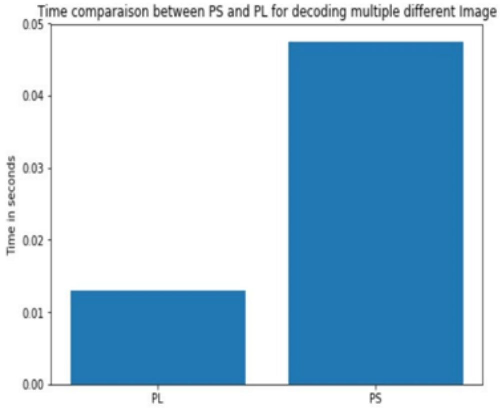
So we can say:

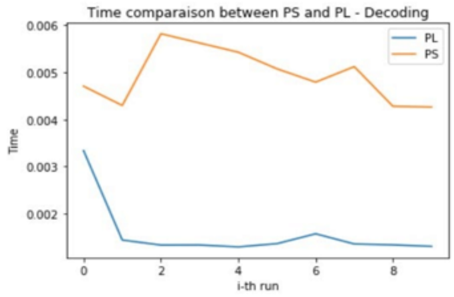
Embedded accelerated solution performance
compared to software only solution $\propto$ number of characters

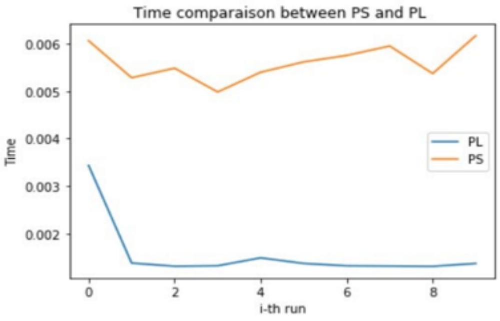
The speedups charts were giving 1.17 to more than 7 times speedups for hardware accelerated solutions. During our analysis we also found out decoding was taking less time than encoding since it was not manipulating the bits, but just reading the input data. For initial image transfer IP was taking almost double/triple time to encode/decode the values, It might be because of the “DMA to Memory” mapping, we can confirm that by checking the times for the multi image encoding / decoding which was almost same time for each transfer (more time compared to the first case since allocating and freeing up the arrays). In the Pixel function there two loops which is dealing with binary - decimal conversions, we have used #HLS pragma unroll to run the independent variables updates to make it faster along with that we are using POW() function which is also running parallely, since POW is a resource hungry function, the concurrent unrolling the of the function make the entire process way faster than sequential software only solutions.

Time Comparison for 4 Character word

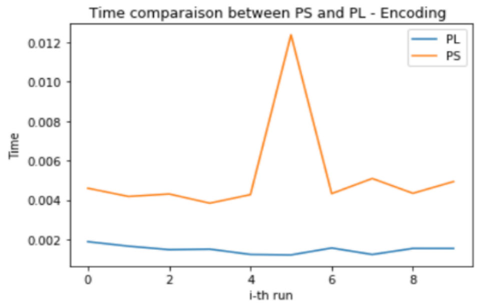


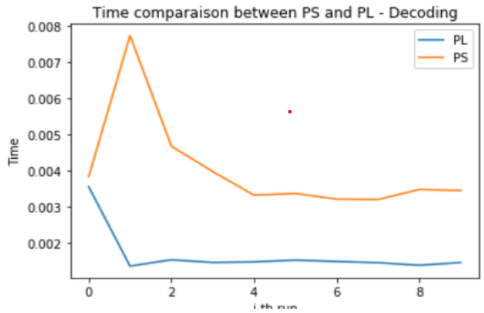


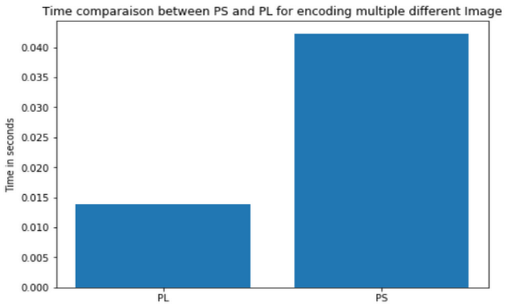


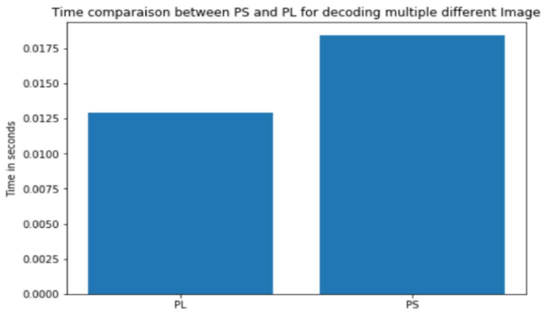


Time Comparison for 3 Character word

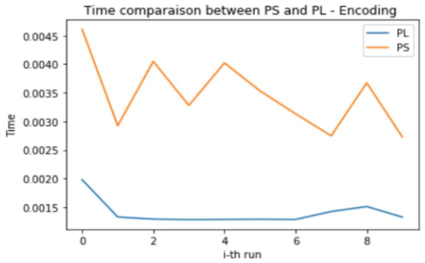


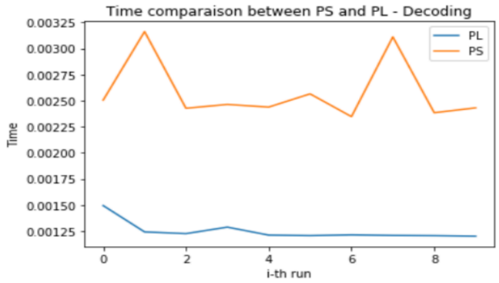


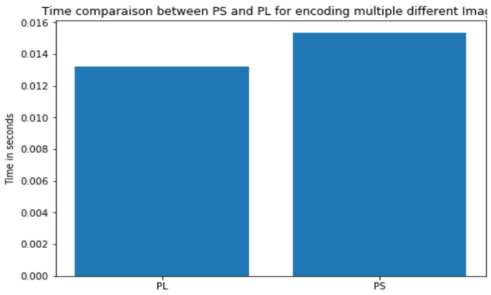


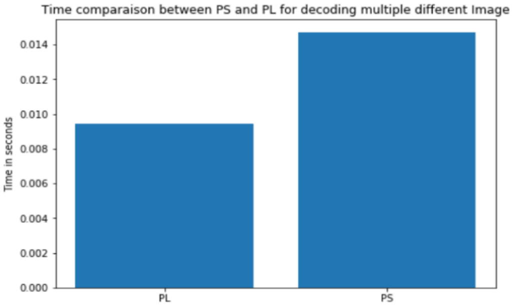


Time Comparison for 2 Character word

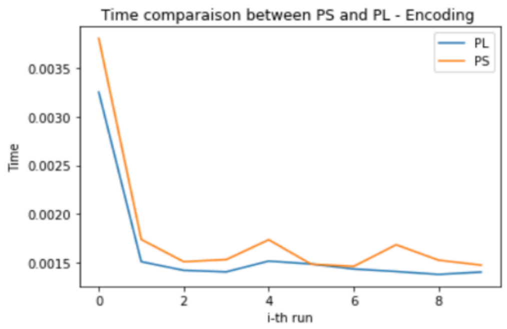


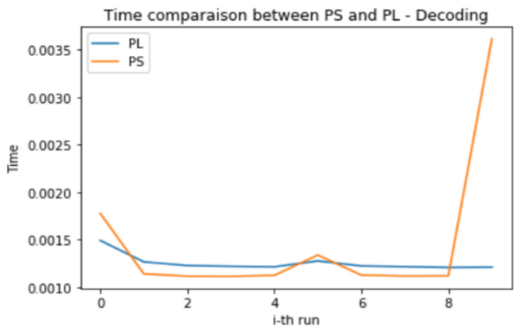


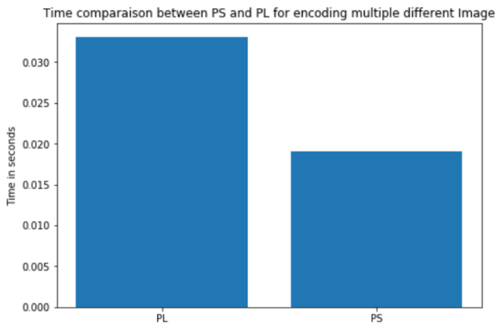


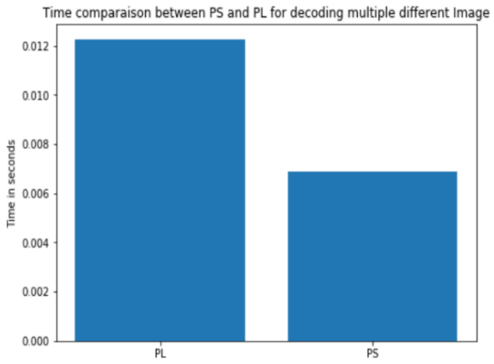


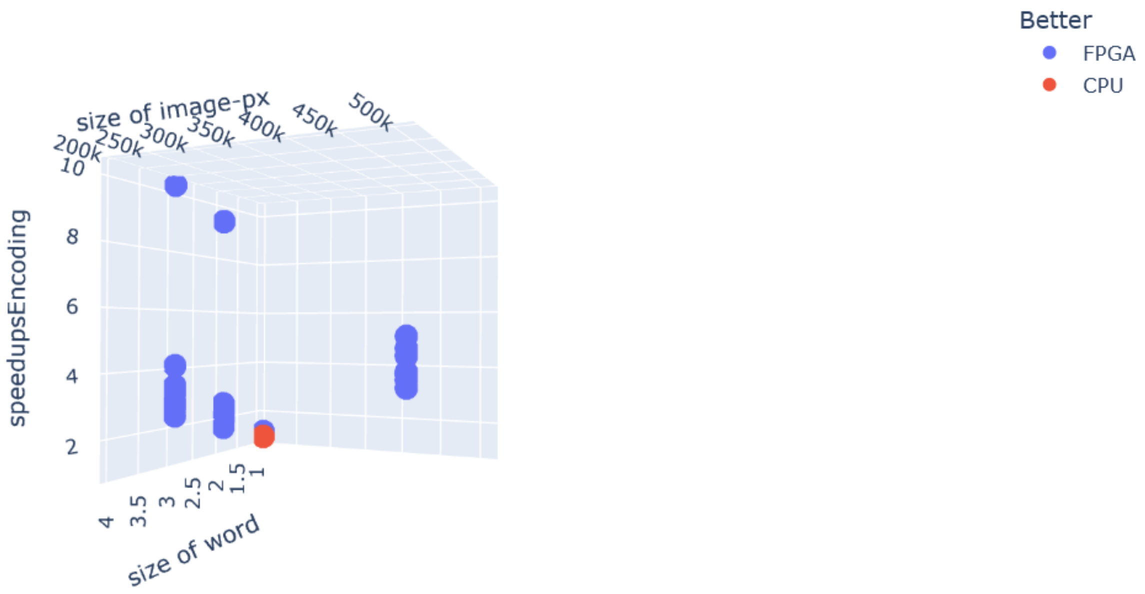
Time Comparison for 1 Character word

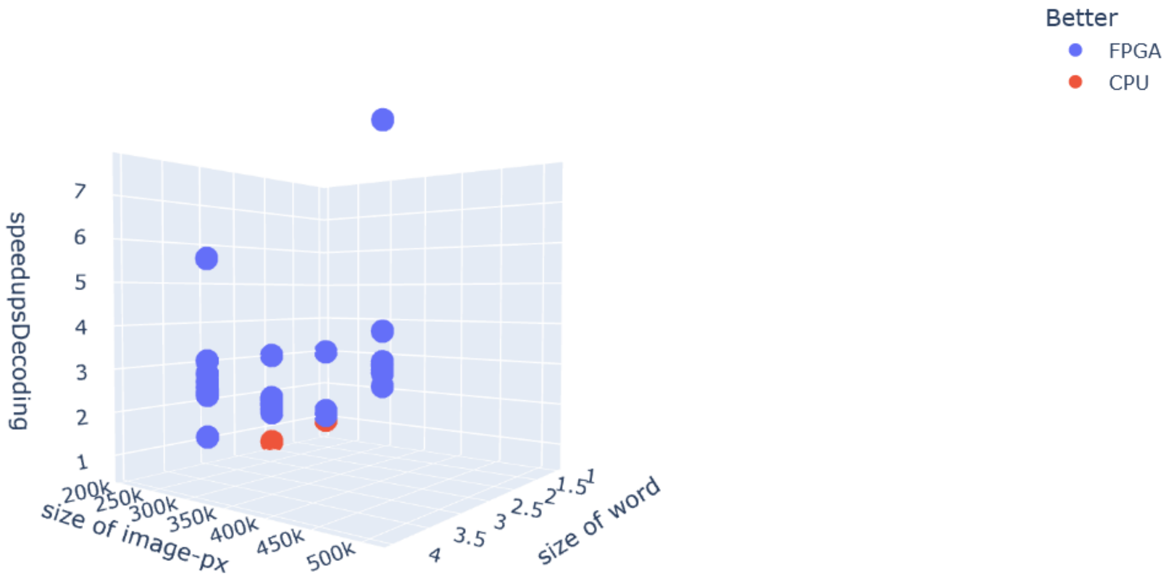


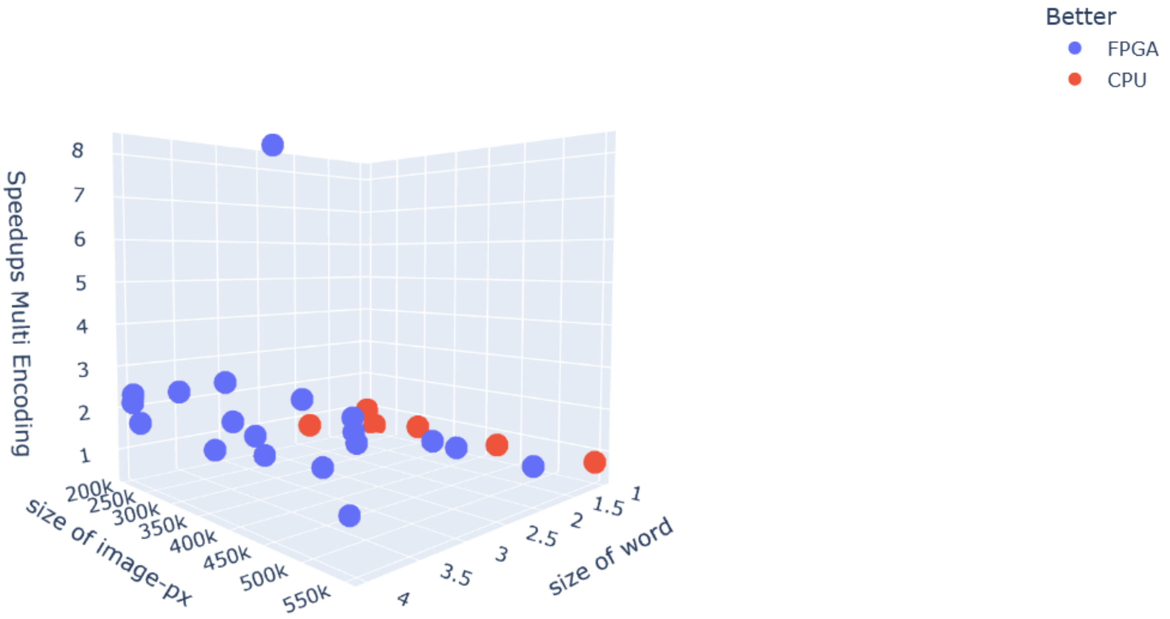


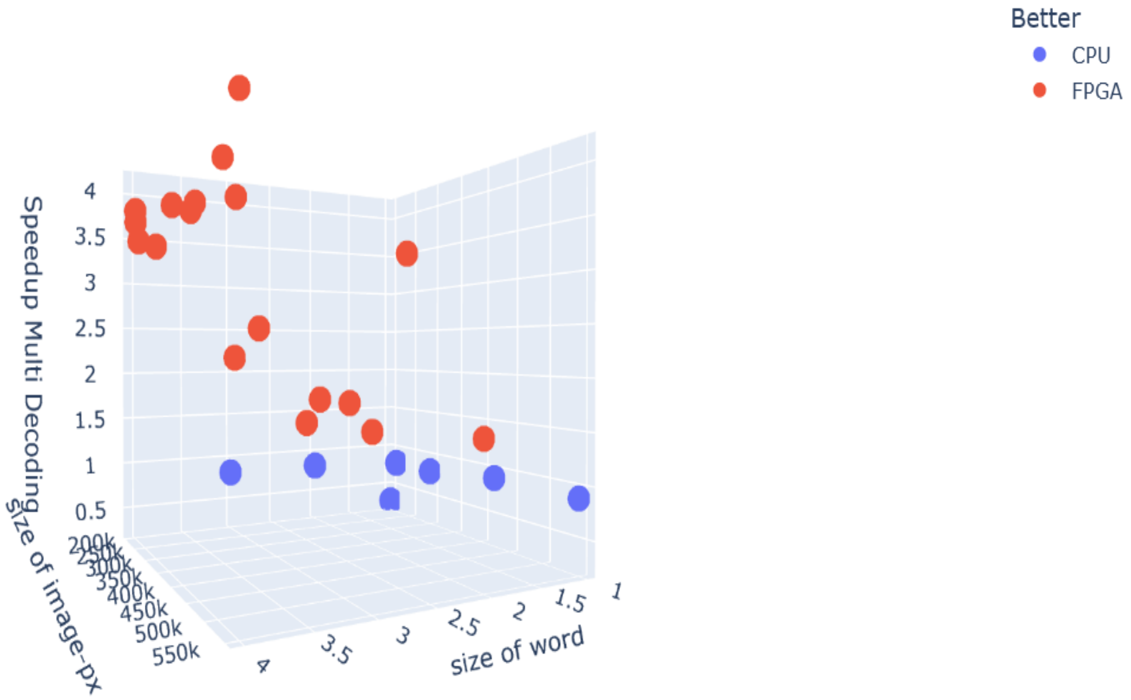







Better
● FPGA
● CPU
speedupsDecoding
1
2
3
4
5
6
7
size of image-px
200k
250k
300k
350k
400k
450k
500k
size of word
1
1.5
2
2.5
3
3.5
4


Better
● FPGA
● CPU
Speedups Multi Encoding
1
2
3
4
5
6
7
8
size of image-px
200k
250k
300k
350k
400k
450k
500k
550k
size of word
1
1.5
2
2.5
3
3.5
4



Detailed 3d Speedup plots can be found in [jupyter notebook]

Issues and Solutions

We encountered the following issues and errors while working on the project:

- ERROR: [COSIM 212-345] Cosim only supports the following 'ap_ctrl_none' designs: (1) combinational designs; (2) pipelined design with II of 1; (3) designs with array streaming or hls_stream or AXI4 stream ports.

    - https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/cosimulationinvitishls.html
    - [Note] Using the ap_ctrl_none mode might prevent the design from being verified using the C/RTL co-simulation feature.

- HLS stream input and output for Test bench main.cpp

    - Created input and output from the ap_axis data structure

- %matplotlib widget not working for 3d interactive arrays

    - We Have tried to solve the issue by enabling widgets and installing ipympl, but were not successful so we solved the issue by using plotly lib

- RuntimeError: Failed to allocate Memory! on PYNQ board

    - As a workaround we have freed up all the unused buffers and shutdown the unused jupyter notebooks

Future Improvement Scope

The project can be improved in many ways, some suggestions are given below,

1. Character pointers or character vectors for storing the secret data to embed over combined ascii values as a number. As of now we can use only ascii values of 4 characters(MAX) as the secret data.

2. Can use memory map over HLS stream, since in this project there is not much use of it, since we have a single IP block connected to ZYNQ7. We were using the HLS stream to get familiar with it.

3. Control the HLS stream with loop and limit it to stream_count
   And use #pragma HLS PIPELINE II=1 for optimization.

4. Can use HLS Pragmas for optimization such as
   #pragma HLS ARRAY_PARTITION for character arrays.

Conclusion

To conclude , FPGAs are powerful resources when it comes to data processing due to the fact that the data is processed in parallel. Even if we have not used Hardware acceleration to its full potential, we have noticed in the time comparison, the execution time for FPGA gets better and better with the number of secret word characters.

Embedded accelerated solution performance
compared to software only solution ∝ number of characters

References

1. **HLS Pragmas**

2. **Optimization Techniques**

3. **HLS stream**

4. **PYNQ Introduction — Python productivity for Zynq (Pynq)**