CS 577: Project Report

Project Number :	17
Group Number:	19
Name of the top modules:	Crypto_sign
Link for GitHub Repo:	https://github.com/PrachiS24/project_17_optimized.git

Group Members	Roll Numbers
Ira Bisht	194101019
Prachi Shrivastava	194101037
Rashika Sharma	194101040
Sakshi Sharma	194101042

Date: 15.05.2020

INTRODUCTION

The project aimed at understanding high level synthesis(HLS), and the steps involved in the process. It also comprises of learning the use of the software suite Vivado, for the same. This report consists of explanations and results of the several steps involved in performing optimization on the code provided, with the help of the Vivado software.

HLS is an automated design process that compiles a high level description of a design into a RTL implementation keeping in mind the specific constraints. HLS input is an untimed data flow and is specified in languages like C, C++ etc. The output of the HLS may include RTL implementation, Analysis feedback etc.. HLS is used to identify and extract parallelism and optimize the code according to various optimization goals. Major benefit of HLS is that it allows designing at a high level of abstraction.

Vivado is a software tool for synthesis and analysis of hardware designs. We can perform HLS using this software tool. Vivado high level synthesis can compile C programs directly into RTL with the need for doing it manually.

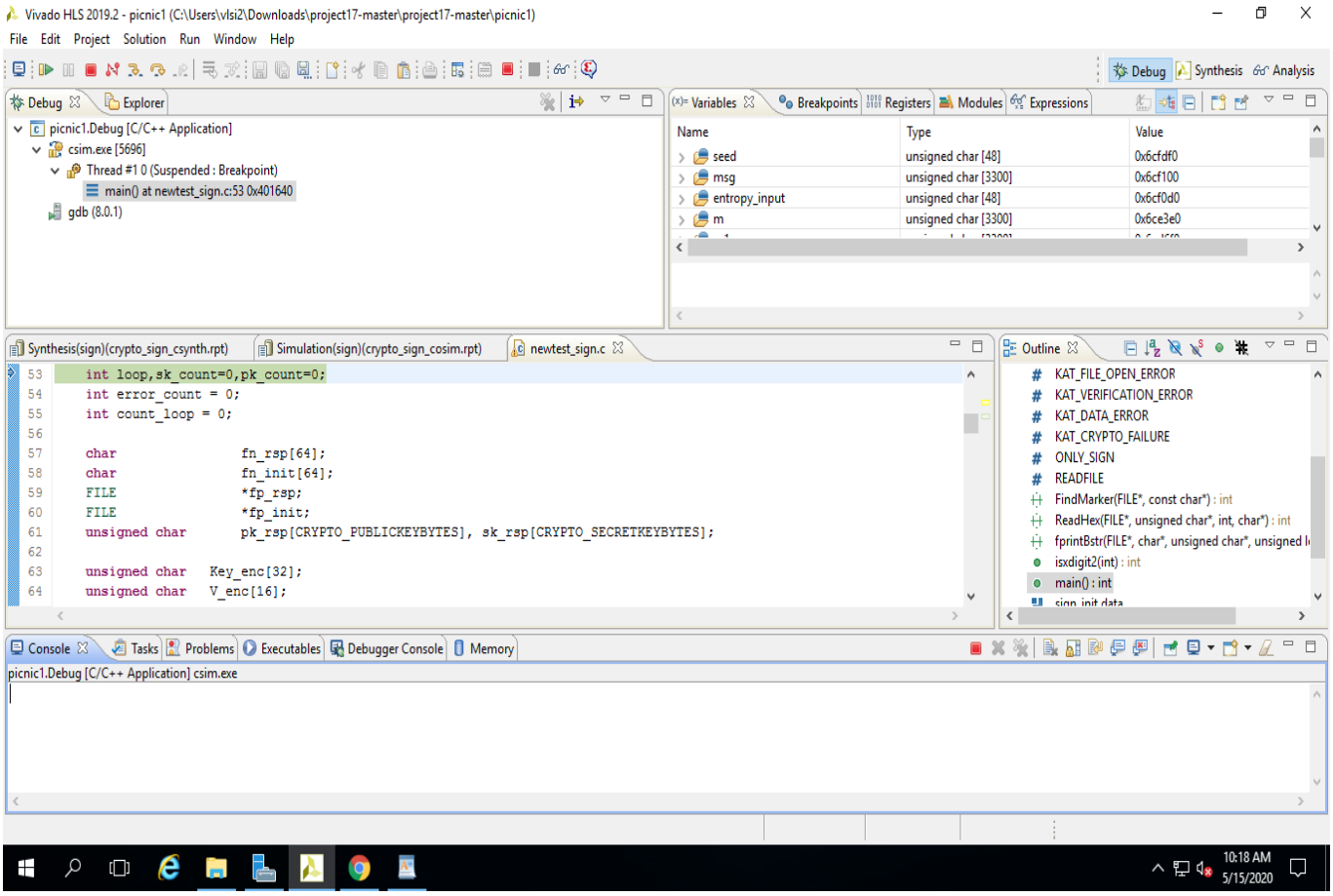
The major steps in the project includes importing the code (“picnic” , in our case) from github to the Vivado IDE and perform simulation ,synthesis and RTL simulation on the code. Initially, area utilization for the project was greater than 100% (103%). We have applied resource and area optimization techniques using various directives in VIVADO. Area and latency utilization were significantly reduced after using various directives.

We get reports after each of these operations using which we perform optimizations in order to make our code more efficient and faster. The report consists of these reports and the results of these operations in a tabular form. We identify the areas where we can perform optimizations for getting more efficient results. These optimizations and its details are also enlisted in phase two of the results.

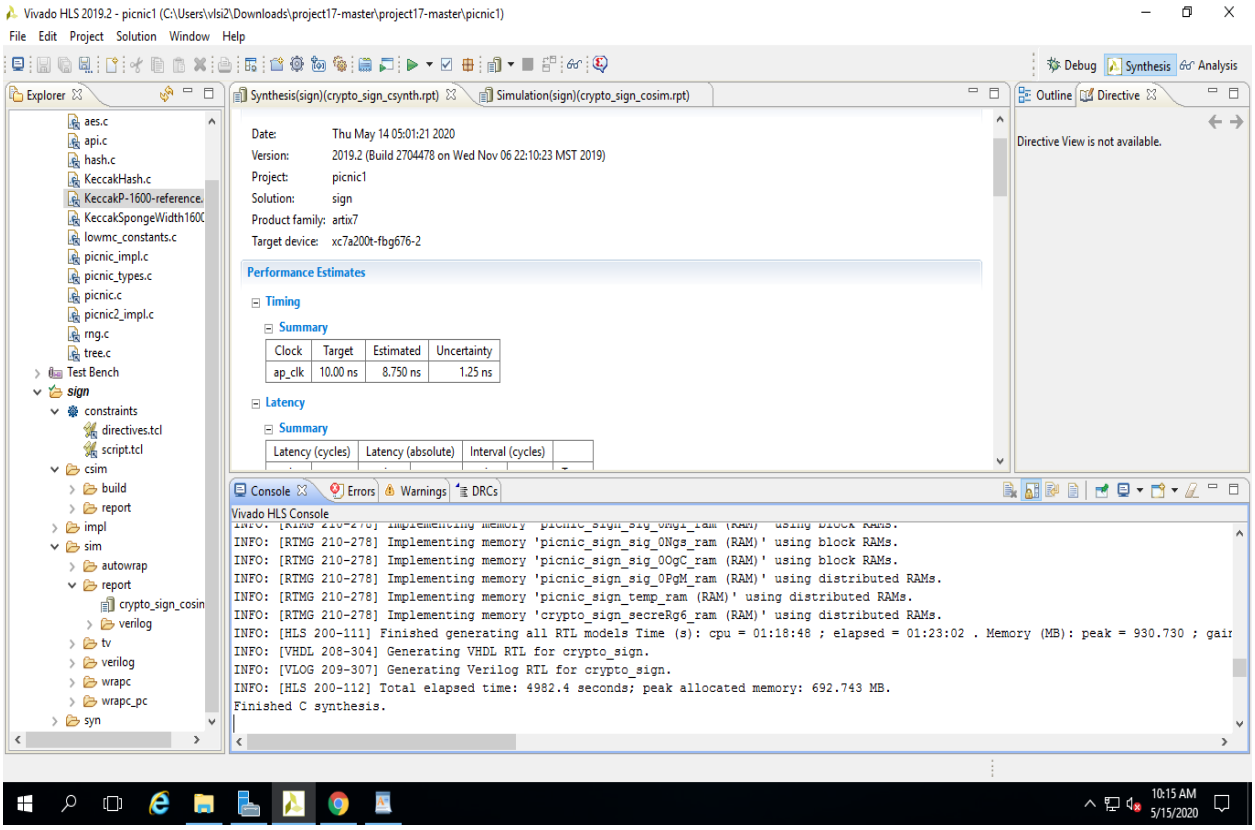
PHASE-1

- Running the algorithm

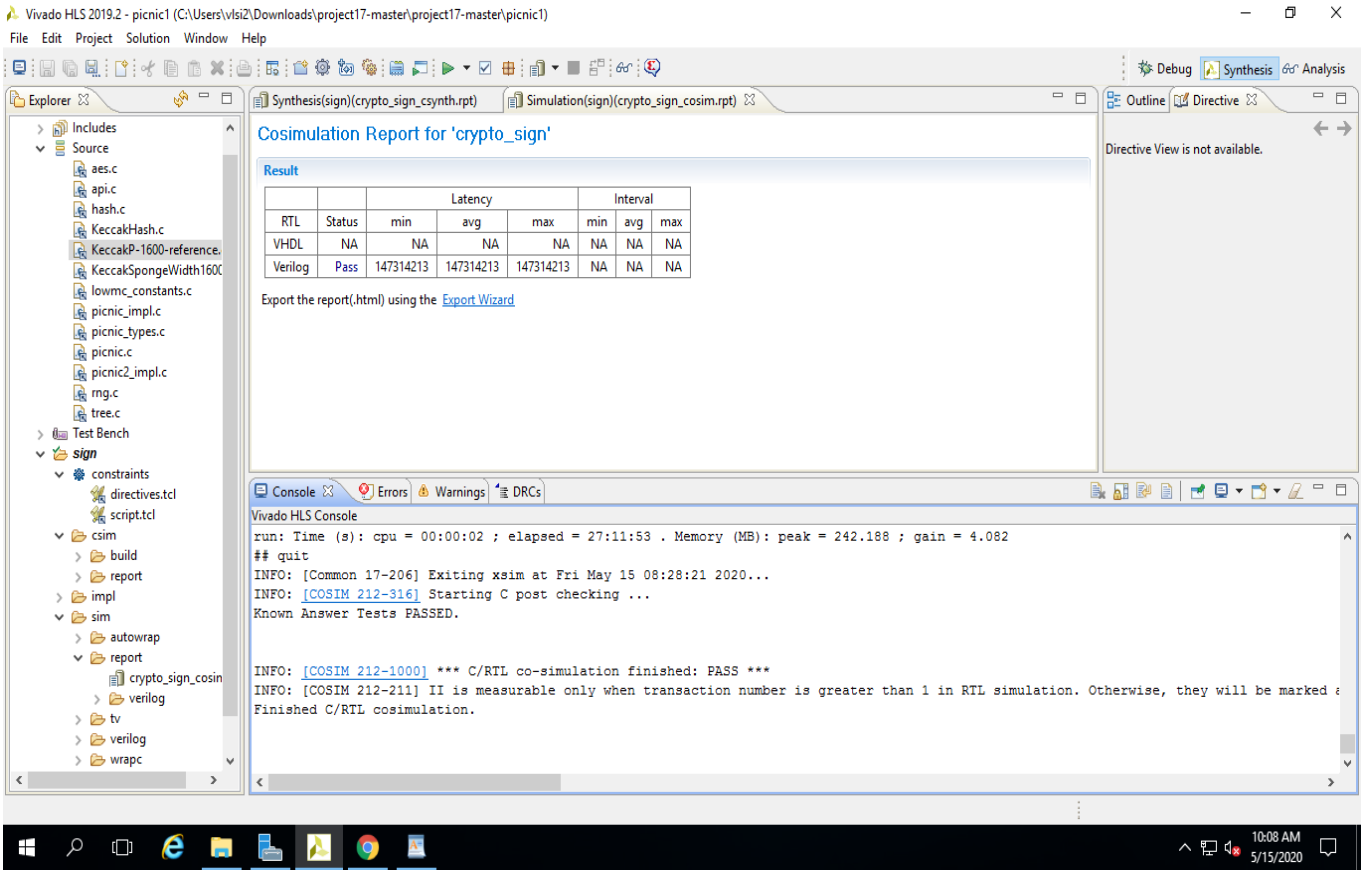
1.1 Simulation screenshot



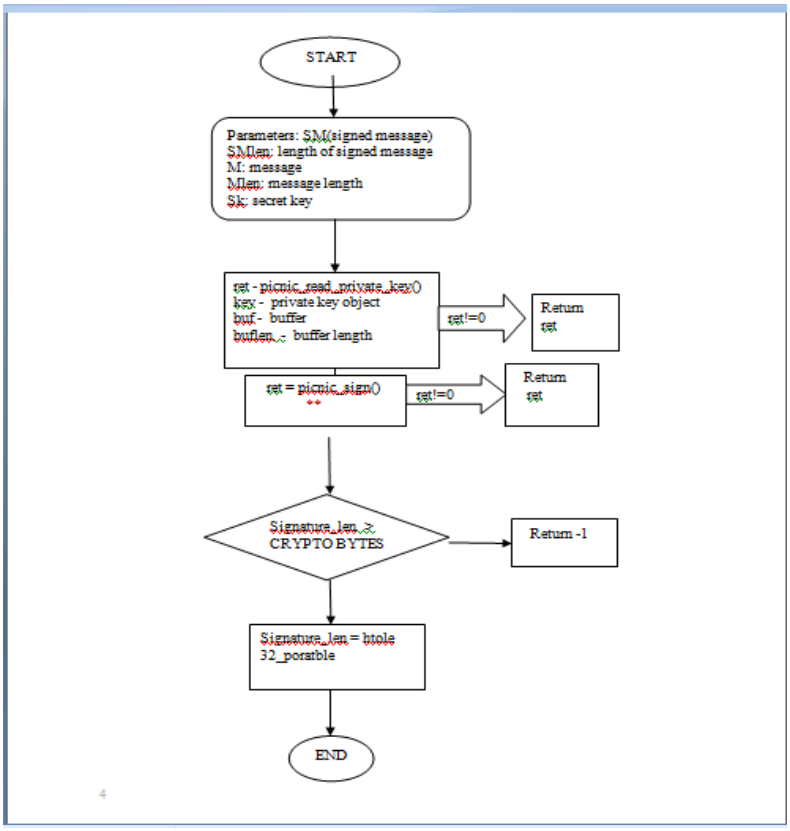
1.2 Synthesis screenshot

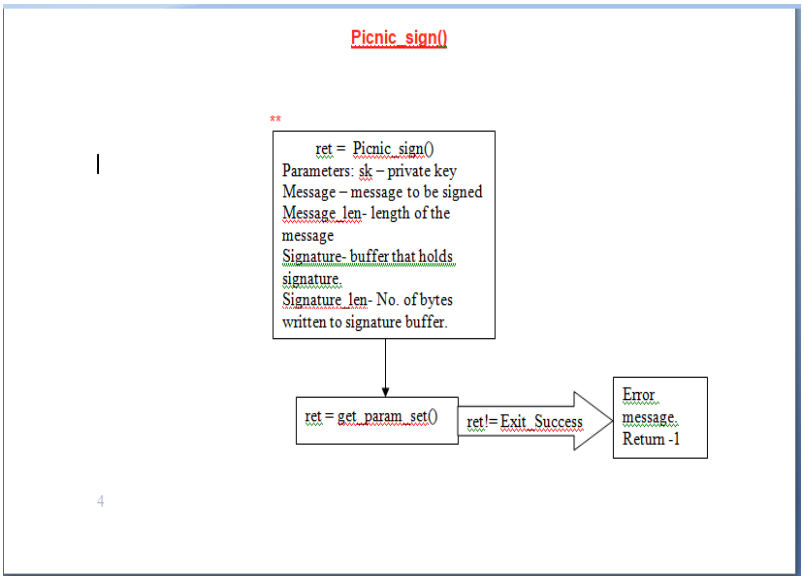


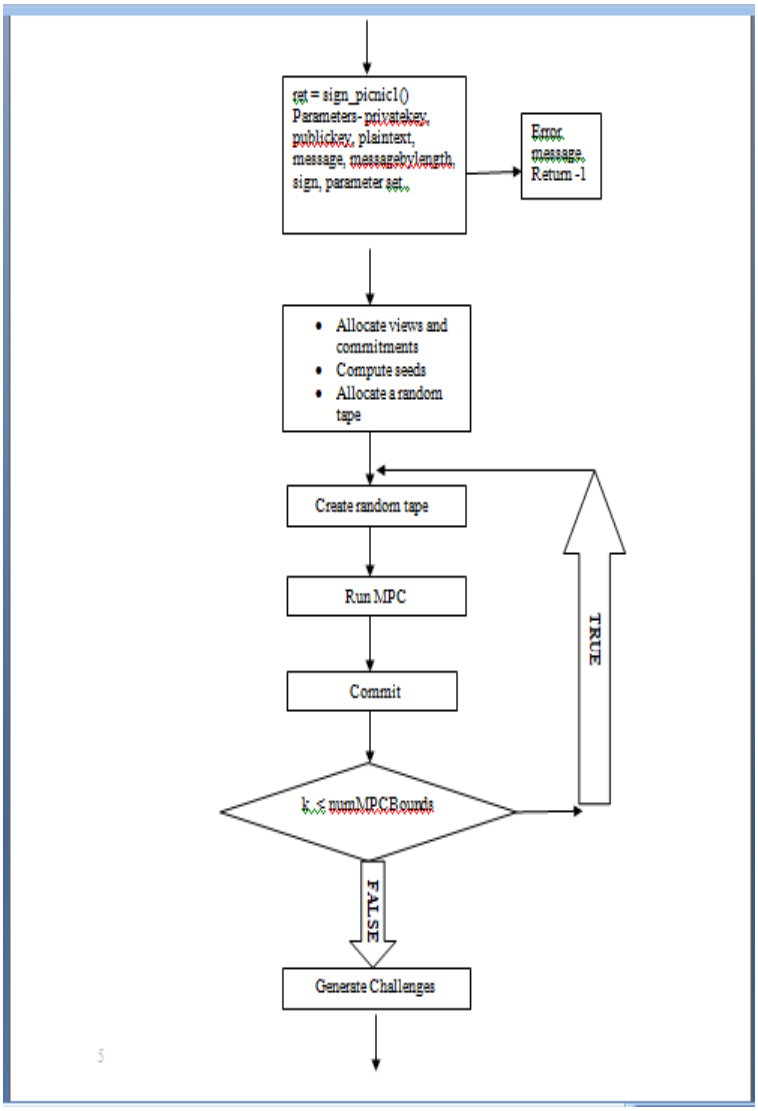
1.3 C/RTL co-simulation screenshot

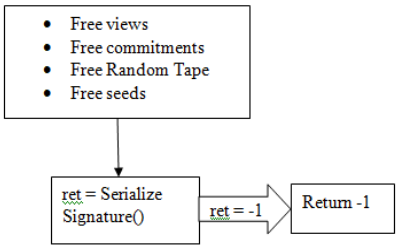


- Flowchart (Give the flowchart of the function used. Describe basic understanding of the algorithm)






ret = sign_picnic1()
Parameters- privatekey, publickey, plaintext, message, messagebylength, sign, parameter set,
Error message. Return -1
Allocate views and commitments
Compute seeds
Allocate a random tape
Create random tape
Run MPC
Commit
k < numMPCBounds
TRUE
FALSE
Generate Challenges


Free views
Free commitments
Free Random Tape
Free seeds
ret = Serialize Signature()
ret = -1
Return -1

The crypto_sign function is the top function. It basically adds a signature to the beginning of the message m by making use of the secret key sk. The picnic_write_private_key serialisers the private key. It returns the number of bytes written. The picnic_sign function is the signature function. Given the key-pair, it signs a message with it

picnic_sign calls a number of functions to achieve this. Some major functions among then are get_param_set, sign_picnic1 which actually does most of the tasks such as allocating views and commitments, computing seeds, allocation random tape. createRandomTape, runMPC, commit , these functions are a part of a loop.

For each iteration compute random tapes, simulate the MPC protocol to compute the LowMC which consists of a number of XOR and matrix multiplication operations, and finally we store the shares in the views. And then we form commitments.

Next we compute challenges with the function H3, which is a hash function. It hashes the output shares, commitments, public key and the message. Lastly SerializeSignature() serialises the signature into a byte array, encoding the signature. Signature length must be at most CRYPTOBYTES .If greater than that, the function returns -1. Else signature length is set by calling the function htole32_portable.

- Result

FPGA Part	Name of Top Module	FF	LUT	BRAM	DSP	Latency	II
xc78200tfbg76-2	crypto_sign	46135(17%)	104855(77%)	468(64%)	5(~0%)	95602574	

3.1 Explain the result

Initially, the area utilization of the project was 103% for LUTs. We used directives for area utilization which resulted in reducing area utilization of LUTs from 103% to 77%. Also, FF area utilization decreased from 18% to 17% when resource optimization was applied along with latency optimization. Initially, latency was 147314213 which was reduced to 95602574 using latency optimization techniques.

3.2 Problems and its solution

Major concern of the project was area utilization. It was greater than 100%. That is, the circuit was unable to fit in the chip. LUTs were demanding a huge portion of the chip (more than available). To remove this, problem, we used various directives and techniques for area utilization. The main directive that reduced area utilization significantly was 'INLINE' directive. Using INLINE with functions that were being called multiple times, the area utilization of LUTs and FFs reduced significantly to 77% and 17% respectively. Also, after using function INLINE, synthesizing and RTL co-simulation processes became much faster which initially took hours to complete.

After area utilization, the latency increased to 129961224. To reduce this, latency optimizations were done using different directives such as PIPELINE , LOOP UNROLL and ARRAY_PARTITION. Small loops were unrolled to reduce the loop overheads. The nested loops which were frequently used were pipelined. So the final latency became 95602574.

PHASE-2

The target FPGA board is **Artix-7 board**

Benchmark	Type (Area /Latency)	Resource Utilization				Latency		Major Optimizations
		LUT	FF	DSP	B RAM	No of Clock cycle/latency	Clock period	
Baseline	B	102%	18%	~0%	57%	147314213	8.750	
Optimization11	Area	65%	11%	~0%	60%	129961224	8.750	Used Inline directives with most frequently called function like HashUpdate, HashFinal etc.
Optimization 2	Latency	77%	17%	~0%	64%	95602574	8.750	AREA+LATENCY Used pipline and loop unroll directives.

Explanation-

Optimization 1- Area Optimization-

- Initially, area utilization was greater than 100%. That is, the circuit was unable to fit in the chip.
- LUTs were demanding a huge portion of the chip (more than available).
- We have used ‘INLINE’ directive for function inlining.
- . Using INLINE with functions that were being called multiple times, the area utilization of LUTs and FFs reduced significantly to 60% and 11% respectively
- Also, after using function INLINE, synthesizing and RTL co-simulation processes became much faster which initially took hours to complete.

Optimization 2- Area+Latency Optimization-

- After the area optimization, latency optimization has to be done which was initially very high.
- Pipeline directive was used in inner most loop in nested loops to remove the sequential execution responsible for high latency.
- Small loop were unrolled to remove the loop overhead of condition checking, incrementing etc.
- Finally array_partition was also used as RAM has limited ports to read and write the data.