

# CS577:Project Report

Project Number :	19
Group Number:	01
Name of the top modules:	ntt
Link for GitHub Repo:	<a href="https://github.com/m-shivam/CS577VLSI_grp01_proj19">github.com/m-shivam/ CS577VLSI_grp01_proj19</a>

<b>Group Members</b>	<b>Roll Numbers</b>
Shivam Maurya	194101045
Mayank Sharma	194101030
Vinay Kumar Gupta	194101056
Rupak Gupta	194101041

## A. Introduction:

General Information:

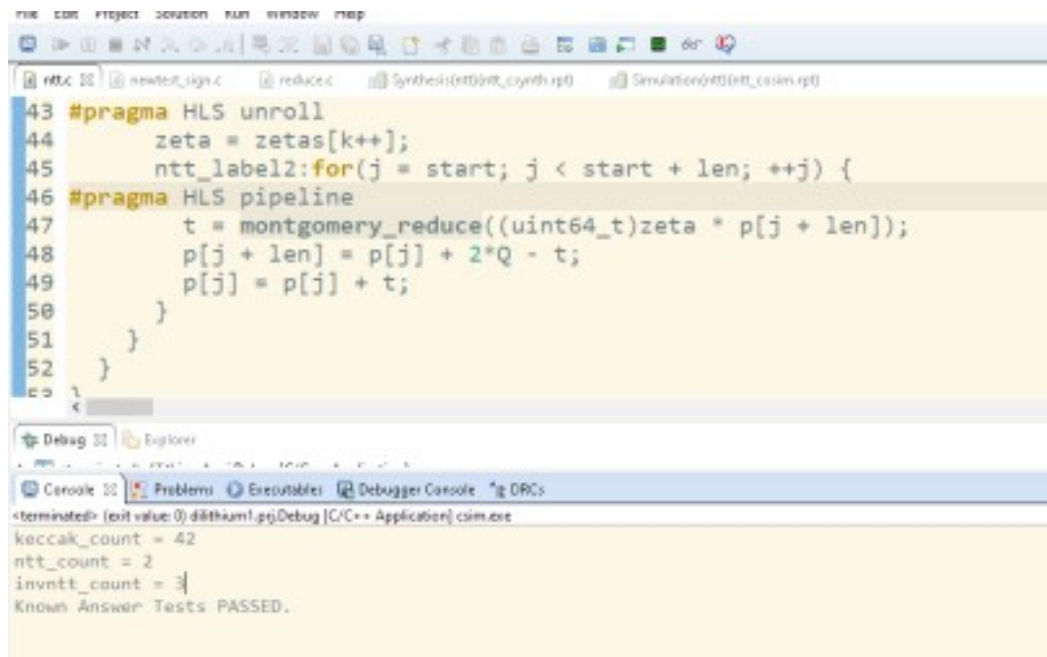
**Project:** dilithium1.prj  
**Solution:** ntt  
**Product family:** artix7  
**Target device:** xc7a200t-fbg676-2

The following steps are followed in order:

1. Run C Simulation
2. Synthesize The Design
3. Run Co-Simulation

Running the algorithm and presenting screenshots of Simulation, Synthesis, and Co-simulation for both non-optimized and optimized versions.

### 1. Simulation screenshot



```
43 #pragma HLS unroll
44     zeta = zetas[k++];
45     ntt_label2:for(j = start; j < start + len; ++j) {
46 #pragma HLS pipeline
47         t = montgomery_reduce((uint64_t)zeta * p[j + len]);
48         p[j + len] = p[j] + 2*Q - t;
49         p[j] = p[j] + t;
50     }
51 }
52 }
53 }
```

```
<terminated> [exit value: 0] dilithium1.prj.Debug [C/C++ Application] csim.exe
keccak_count = 42
ntt_count = 2
invntt_count = 3
Known Answer Tests PASSED.
```

## 2. Synthesis screenshot

### Performance Estimates:

*Timing Summary:*

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	8.363 ns	1.25 ns

*Non\_optimized*

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	8.635 ns	1.25 ns

*Optimized*

### Utilization Estimates:

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	2	0	797	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	1	-	0	0	-
Multiplexer	-	-	-	146	-
Register	-	-	536	-	-
Total	1	2	536	943	0
Available	730	740	269200	134600	0
Utilization (%)	~0	~0	~0	~0	0

*Non\_optimized*

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	256	0	51060	-
FIFO	-	-	-	-	-
Instance	-	-	0	435	-
Memory	1	-	0	0	-
Multiplexer	-	-	-	13702	-
Register	0	-	46364	5728	-
Total	1	256	46364	70925	0
Available	730	740	269200	134600	0
Utilization (%)	~0	34	17	52	0

*Optimized*

### 3. C/RTL co-simulation screenshot

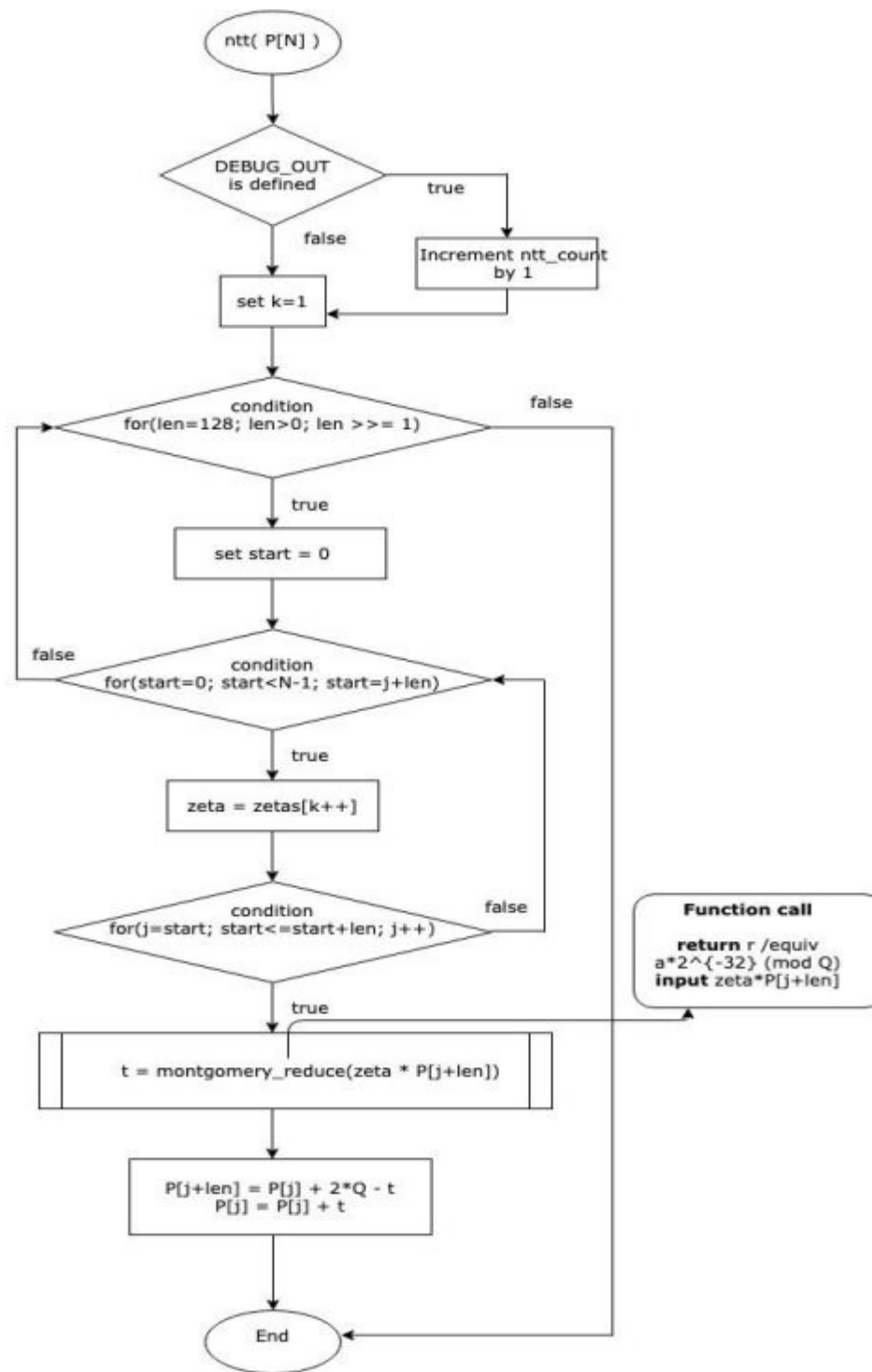
		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	8974	8974	8974	8975	8975	8975

*Non-Optimized*

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	3620	3620	3620	3621	3621	3621

*Optimized*

## B.1. Flowchart



## B.2. Understanding Of Algorithm (ntt):

“*ntt*” function is implemented in triple nested for-loops. The outer for-loop runs eight times and the inner loops iterate through the elements of the given array *p* while doing operations on those elements. The array *p* likely to represent the polynomial and the elements of *p* represents the coefficients of this polynomial. In *ntt* algorithm value of *N* is constant and which is 256 so the polynomial has 256 coefficients and we can change the value of *N* based on the implementation of Algorithm.

The array **Zetas** is the array of predefined values. In this algorithm `Montgomery_reduce()` function is used instead of the normal modulo operation to reduce the number of clock cycles. The outer loop runs from length 128 to 0. In each step, length becomes half so the outer loop runs total 8 number of times. We count from 7 to 0 for the levels.

- **First Iteration of Outer most loop:**

We can see that the First Iteration only loops through the second nested for-loop once because the condition,  $start < N$  is not met anymore after one iteration. Thus for this level, the second nested for-loop has not to be taken into account and we only have to look at the third nested for-loop. The First Iteration of the NTT can be thought of as follows. Array *p* is used to compute the new version of array *p* after the for-loop, There are two new values *p*[0] and *p*[128] is generated which is again assigned to it.

$$t = \text{Montgomery\_reduce}(\text{zeta} * p[128]) \text{ and}$$
$$p'[128] = p[0] + 2 * Q - t$$
$$p'[0] = p[0] + t$$

This happens pairwise for every *p* [*j*] and *p* [*j* + 128] for *j* in range 0 to 127. And we see that that second operation happens between *p* [1] and *p* [129]. The idea is to take *p*[128:255] (*p* high) to compute *t*. Then we take *p* [0:127] (*p* low) to compute *p'* [0:127] and *p'* [128:255]. Every iteration multiplies *p* high with an element of the array *zetas* to compute *p'*. The first iteration uses *zetas* [1], the second packet uses *zetas* [2], and so on.

- **Second Iteration of Outer loop:**

This Iteration differs from the First Iteration because it must use the second nested for-loop. We now iterate twice through this for-loop, thus we have two parts of operation. The first part of operations includes pairwise operations for every  $p[j]$  and  $p[j+64]$  for  $j \in [0,63]$ . The operations in the second part happen pairwise for every  $p[j]$  and  $p[j+64]$  for  $j \in [64,191]$ .

- **Third to Eighth Iterations:**

As we can see, there is a pattern across the Iterations and in the Algorithm, the butterfly-like operations occur at a narrower range for a higher level Iterations. The whole First Iteration happens in one bundle of butterfly operations. With the first operation being  $p[0] * p[128]$ . The second Iteration happens in two bundles of butterfly-like operations. The first operations for the bundle are:  $p[0] * p[64]$  and  $p[128] * p[192]$ . The next Iteration, the third one, has four bundles of butterfly-like operations with the first operations for each bundle being:

- $p[0] * p[32]$
- $p[64] * p[96]$
- $p[128] * p[160]$
- $p[192] * p[224]$

The fourth Iteration has eight bundles of butterfly-like operations. The first operations of each bundle are:

- $p[0] * p[16]$
- $p[32] * p[48]$
- $p[64] * p[80]$
- $p[96] * p[112]$
- $p[128] * p[144]$
- $p[160] * p[176]$
- $p[192] * p[208]$
- $p[224] * p[240]$

We can see the pattern; the number of bundles is doubled by each level. And the addresses for the very first butterfly-like operation for a level is computed by  $p[0] * p[1 \ll \text{level}]$  where level starts from the first Iteration to the last (level = 7 to 0). For the first operations of the remaining bundles, we increase the addresses of p with  $2(1 \ll \text{level})$  for each level. Thus for the second bundle it would be  $p[0 + 2(1 \ll \text{level}) * p[(1 \ll \text{level}) + 2(1 \ll \text{level})]$ . With this in mind, we can compute the first operations of each packet of butterfly-like operations for the remaining levels.

## C. Final Result

Latency is reduced from 8974 to 3620 with resources trade-off.

FPGA Part	Name of Top Module	FF	LUT	BRAM	DSP	Latency	II
Artix-7	ntt	46364	70925	1	256	3620	-

## E. Optimization Logic:

In top function "ntt", there are three nested loops, viz. loop1, loop2, loop3 (from outer to inner) and the input is an integer array of size 256.

Loop1 is running constant(7) times. Loop2 and Loop3 are running for variable times. Further Loop3 has one function call in it.

In the implemented optimization strategy, the following pragmas are used:

- ✓ pragma HLS pipeline
- ✓ pragma HLS unroll
- ✓ pragma HLS array\_partition
- ✓ pragma HLS inline



Since the outermost loop is having constant iterations, loop unrolling is done on loop1 and loop2 to enhance parallelism. Full unrolling of loops is done because of very short iteration length of loops.

The problem arose because of unrolling is restricted access of input array in a single clock cycle. For that purpose array is partitioned into 32 subarrays(each of size=8) with <block> type partitioning.

Inline pragma is used in the function montgomery\_reduce(), called in the inner-most loop. It reduces the level of hierarchy in RTL, leading to more optimal RTL.

In the innermost variable length loop, pipelining is done with unspecified Initiation Interval. Vivado itself optimizes the II, based on data dependencies. However, observing the log file while synthesis, it is observed that final II varies between 1 to 6.