An Investigation of Optimization of Area and Latency for Dilithium2

Anirudh Phukan	Pradnesh Prasad Kalkar	Mesharya M Choudhary	Kilaparthi Heemanth	Saket Kumar Singh
190101104	190101103	190101053	200101056	190101081
B.Tech. CSE	*B.Tech. CSE*	*B.Tech. CSE*	*B.Tech. CSE*	*B.Tech. CSE*
IIT, Guwahati	*IIT Guwahati*	*IIT Guwahati*	*IIT Guwahati*	*IIT Guwahati*
Guwahati, India	Guwahati, India	Guwahati, India	Guwahati, India	Guwahati, India
aphukan@iitg.ac.in	p.pradnesh@iitg.ac.in	m.mesharya@iitg.ac.in	k.heemanth@iitg.ac.in	saketkumar@iitg.ac.in

I. Introduction

The objective of this assignment is to minimize latency and various parameters corresponding to area i.e. LUT, FF and BRAMs for the **crypto_sign top function of Dilithium2 project** for the **FPGA board Artix 7 xc7a200tfbg676-2**. In the previous assignment we worked to successfully run synthesis and cosimulation on the project and our work here is the extension over that by using additional pragmas which are supported by the Vivado/Vitis HLS tool. All executions of code have been done for Vivado HLS version - 2019.2. The report is organised as follows:-

- **Introduction**
- **Methodology** - The description of the pragmas used for optimizing area and latency as well as an incremental description of changes made for each version.
- **Results and Analysis** - This section compiles the various results and contains a brief analysis of observations drawn.
- **Conclusion**

II. Methodology

A. Area Optimization

We have utilized the following pragmas to reduce area utilization (values obtained after **synthesis**).

1) **pragma HLS inline** - The HLS compiler is instructed to insert a function or loop inside the function or loop that it is calling. This may decrease the quantity of instructions and memory accesses needed to run the program, enhancing performance and lowering power consumption. Performance can be increased by using this instruction as it can reduce the need for register and memory accesses.
2) **pragma HLS array_partition** - By partitioning an array into smaller subsets, each subset can be stored in a separate memory location or implemented in a different hardware module, which can reduce the amount of logic required to access and operate on the array. In addition to improving area efficiency, array partitioning can also help to improve performance by increasing parallelism and reducing data dependencies. By partitioning the array into smaller subsets, multiple hardware modules can operate on the array in parallel, reducing the overall computation time.
3) **pragma HLS allocation instances=list limit=value type** - This directive instructs the HLS compiler to allocate a limited number of hardware resources, specifically value number of Xilinx FPGA list instances, to perform a specific operation in the design. This can be useful when implementing designs with resource constraints, such as when there are limited hardware resources available or when designing for low-power applications. The instances argument specifies the number of instances of the specified operation to be allocated, and the type argument specifies the type of operation to be performed.

In order to optimize, we took an incremental approach, either by using more types of macros or by using them in more places. As a result, we noticed various **FF**, **LUT**, and **BRAM** values that corresponded to various versions. The following is a list of the various versions and the significant adjustments made to each version, along with the resource usage:

Note: First value denotes **BRAM utilization percentage**, the second value is the **FF utilization percentage** and the last value denotes the **LUT utilization percentage**.

1) **9_14_136** - These Resource Utilization values corresponds to the code which we submitted during Phase-1 of the project. We use this latency value as the baseline upon which we improved further.
2) **8_9_92** - We used inlining in keccak_absorb and crypto_sign_signature.
3) **8_8_79** - Apart from inlines used in the previous version, We used inline in keccak_init, keccak_finalize, keccak_squeezeblocks,keccak_squeeze, shake128_absorb, shake256_init, shake256_absorb, shake256_finalize, shake256, poly_ntt, poly_challenge,polyvec_matrix_expand, dilithium_shake256_stream_init.
4) **8_6_67** - Apart from inlines used in the previous versions, we used inline in the function shake256_squeezeblocks
5) **8_19_34** - Apart from inlines used in the previous

versions, We restricted the number of xor resources to 4 (since in the function, there were at most 4 XOR operations being used in a single line) by using this command pragma HLS allocation instances=xor limit=4 operation in KeccakF1600_StatePermute function. Experiments with limit=5,8 gave similar results.

6) **8_14_35** - As a continuation to previous experiment, we also experimented with 12 xor resources in KeccakF1600_StatePermute function. This reduced FF utilization percentage significantly. Experiments with limit=16 gave similar results.
7) **8_15_38** - As a continuation to previous experiment, we also experimented with 20 xor resources in KeccakF1600_StatePermute function. This increased both FF and LUT utilization percentage by a little amount. Experiments with limit=30 gave similar results.
8) **8_11_55** - As a continuation to previous experiment, we also experimented with 100 xor resources in KeccakF1600_StatePermute function. This reduced FF utilization percentage with a trade-off over LUT utilization percentage.
9) **8_6_67** - As a continuation to previous experiment, we also experimented with 1000 xor resources in KeccakF1600_StatePermute function. This brought us back to where we had started with (i.e., just in-lined code without any restriction over xor) which indicated that 1000 is more than sufficient to execute the code.
10) **8_19_25** - Next, we also tried to restrict the NOT (to 4) and AND (to 4) resources in KeccakF1600_StatePermute function keeping XOR=4. This reduced the LUT utilization percentage significantly. Experiments with 4 XOR, 1 NOT and 1 AND gave similar results. Note that we need fewer NOT and AND gates since, the code uses at most one of them in a single line.
11) **8_16_34** - As a continuation to the previous idea, we experimented with 30 XOR, 4 NOT and 4 AND (to 4) resources in KeccakF1600_StatePermute function. This reduced FF utilization percentage with a trade-off over LUT utilization percentage. Experiments with 30 XOR, 1 AND and 1 NOT gave similar results.
12) **8_16_27** - As a continuation to the previous idea, we experimented with 12 XOR, 4 NOT and 4 AND (to 4) resources in KeccakF1600_StatePermute function. This reduced LUT utilization percentage without any compromise over any other resources. Experiments with 12 XOR, 1 AND and 1 NOT gave similar results.
13) **8_14_28** - As a continuation to the previous idea, we experimented with 12 XOR, 12 NOT and 12 AND (to 4) resources in KeccakF1600_StatePermute function. This reduced FF utilization percentage without much compromise over any other resources. Experiments with 30 XOR, 30 AND and 30 NOT gave similar results.
14) **8_12_35** - As a continuation to the previous idea, we experimented with 32 XOR, 32 NOT and 32 AND (to 4) resources in KeccakF1600_StatePermute function. This reduced FF utilization percentage with a little compromise over LUT utilization percentage.
15) **8_14_28** - Finally, to conclude this idea, we experimented with 64 XOR, 64 NOT and 64 AND (to 4) resources in KeccakF1600_StatePermute function. Note that in all of these experiments, we have maintained the NOT and AND count to the same value due to the symmetry present in usage of them in the function.
16) **6_45_59** - Apart from inlines used in the previous versions (as obtained in experiment 4), and restricting XOR, AND and NOT to 32 each in KeccakF1600_StatePermute function, we next performed array partitioning by using the command pragma HLS array_partition variable=state complete dim=1. This reduced the BRAM utilization percentage with a trade-off over LUT and FF utilization percentages.
17) **7_12_38** - Apart from inlines used in the previous versions (as obtained in experiment 4), and restricting XOR, AND and NOT to 32 each in KeccakF1600_StatePermute function, we next performed array partitioning by using the command pragma HLS array_partition variable=KeccakF_RoundConstants complete dim=1. This reduced the BRAM utilization percentage a little without much compromise over LUT and FF utilization percentages.
18) **4_44_61** - Apart from inlines used in the previous versions (as obtained in experiment 4), and restricting XOR, AND and NOT to 32 each in KeccakF1600_StatePermute function, we next performed array partitioning on both the variables state and KeccakF_RoundConstants using the previous two pragmas. This drastically reduced the BRAM utilization percentage with a little compromise over LUT and FF utilization percentages.
19) **4_36_106** - Finally, to conclude our experiments, we removed the restriction on XOR, AND and NOT in KeccakF1600_StatePermute function and array performed partitioning on both the variables state and KeccakF_RoundConstants using the previous two pragmas along with inlining (as obtained in experiment 4). This reduced the BRAM utilization percentage, however reduction in LUT utilization percentage was not that much compared to the initial code.

B. Latency Optimization

We have used the following pragmas for optimizing the average latency (after executing cosimulation on PQC-genKAT_sign.c for 2 iterations):-

1) **pragma HLS unroll** - It instructs the HLS tool to unroll the loop under consideration completely. This approach causes all the loop iterations to be executed in parallel. Thus it decreases the latency maximally (compared to all the other pragmas) for a loop whose number of iterations is known during compile time (i.e., a fixed number of iterations).

2) **pragma HLS pipeline** - It instructs the HLS tool to pipeline a loop which increases throughput of the loop by overlapping the execution of multiple iterations. We used pipelining in the loops which are variable sized because we observed from the logs generated by the tool on synthesis that it can't completely unroll variable sized loops and pipelining gave better reductions than unrolling with a specified factor.

We followed an incremental approach while optimizing by either using more types of macros or using them at more locations. As a result of this we observed multiple latency values corresponding to different versions. The different versions as well as the major changes made to obtain each version along with their **average latency** is listed as follows:-

1) **347094** - This latency value corresponds to the code which we submitted during Phase-1 of the project. We use this latency value as the baseline upon which we improved further.
2) **344635** - We pipelined the for loops in unpack_pk, unpack_sk, pack_sig in packing.c. We pipelined the loop in polyveck_pointwise_poly_montgomery in polyvec.c. We unrolled the loop in crypto_sign in sign.c with factor = 2.
3) **272150** - Apart from the changes discussed previously we pipelined the outer loop in ntt in ntt.c. We pipelined the loops in poly_reduce, poly_caddq, poly_pointwise_montgomery, poly_chknorm, polyz_pack, polyw1_pack in poly.c. We pipelined the loops in polyvecl_ntt, polyvecl_pointwise_poly_montgomery, polyveck_reduce, polyveck_caddq, polyveck_ntt, polyveck_chknorm, polyveck_pack_w1 in polyvec.c.
4) **242440** - We shifted to using unroll everywhere instead of pipeline in the previous version. Apart from this we unrolled all the nested for loops in ntt and invntt_tomont in ntt.c, we unrolled the loops in load64, KeccakF1600_StatePermute, keccak_absorb (some loops only), keccak_squeezeblocks, keccak_squeeze (some loops only) in fips202.c.
5) **226897** - We pipelined some of the variable sized loops that were earlier unrolled and pipelined/unrolled some more loops in the functions discussed previously in fips202.c. We pipelined the for loop in crypto_sign in sign.c.
6) **177555** - We unrolled the for loop in keccak_init in fips202.c. We unrolled the for loops in pack_pk, unpack_pk, pack_sk, pack_sig(some more), unpack_sig(some more). We unrolled the for loops in poly_reduce,poly_caddq, poly_freeze, poly_add, poly_decompose, poly_make_hint, poly_use_hint, poly_chknorm, poly_eta in poly.c. We unrolled the for loops in polyvec_matrix_expand, polyvecl_uniform_eta, polyvecl_reduce, polyvecl_freeze,polyvecl_add, polyvecl_ntt, polyvecl_invntt_tomont, polyvecl_pointwise_poly_montgomery, polyvecl_pointwise_acc_montgomery, polyvecl_chknorm,polyveck_uniform_eta, polyveck_freeze, polyveck_add, polyveck_sub, polyveck_shiftl, polyveck_ntt, polyveck_power2round, polyveck_make_hint, polyveck_use_hint, polyveck_pack_w1 in polyvec.c.

III. Experiments

A. Assumptions

We ran the experiments with number of iterations as 2 in PQCgenKAT_sign.c as was already given in the dilithium2 folder to obtain all the values specified in the report. We ran the c-simulation on final solutions with number of iterations as 100 to verify correctness. All executions of code have been done for Vivado HLS version - 2019.2.

B. Metrics

1) Time: For time, **average latency** was used as a metric.Artix-7 FPGA
2) Area: Since the area metric, involved BRAMs, LUTs and FFs, we researched on a few metrics that can be used and have considered the **Equivalent Slice Count (ESC) metric** to aggregate these for the sake of the assignment.
   The Equivalent Slice Count (ESC) is a metric used to estimate the resource utilization of FPGA Artix-7 devices, specifically in VLSI synthesis designs. It provides a measure of the amount of resources used by the design in terms of the number of equivalent slices utilized.
   An equivalent slice represents the maximum number of resources that can be placed in a single slice on the Artix-7 device. The ESC is calculated by counting the number of LUTs, flip-flops (FFs), and block RAMs (BRAMs) used in the design, and weighting each resource as follows:
   - Each LUT counts as 1/4 slice.
   - Each FF counts as 1/8 slice.
   - Each BRAM counts as 46 slices.

   The total number of slices is then divided by 100 to get the equivalent slice count. $(BRAM \times (\frac{33650}{730}) + FF \times (\frac{33650}{269200}) + LUT \times (\frac{33650}{134600}))_{\overline{100}}$

C. Results and Analysis

1) Time: The average latency has been reduced from 347094 to 177555. We have thus achieved an overall reduction of **49%**. Using unrolling and pipelining for bottleneck functions proved to be useful for this.

2) Area: This section shows the utilization percentage for area optimization as variation of the various experiment settings described earlier.

- BRAM: Experiments 18, 19 which use array partitioning and inlining are observed to be best to optimize BRAM (to **4%**).
- FF: Experiment 4 which uses inlining performs best to optimize FF (to **6%**).

- LUT: Experiment 10 which uses inlining and restricting XOR, AND and NOT instances to 4 each performs best to optimize LUT (to **25%**).
- ESC: Experiment 18 which uses array partioning, inlining and restricting XOR, AND and NOT instances to 32 each performs best overall according to ESC metric.

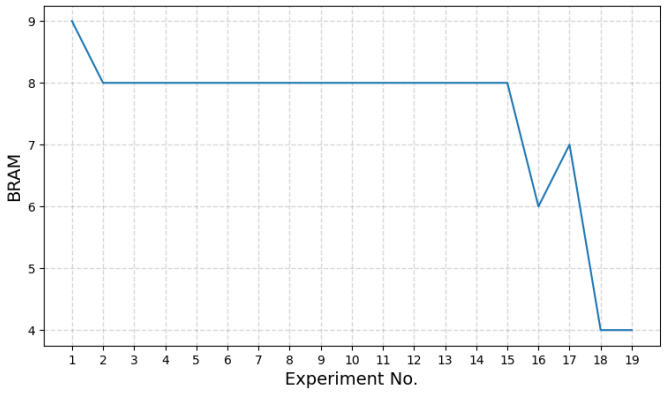

Fig. 1. BRAM Utilization percentage for various experiments

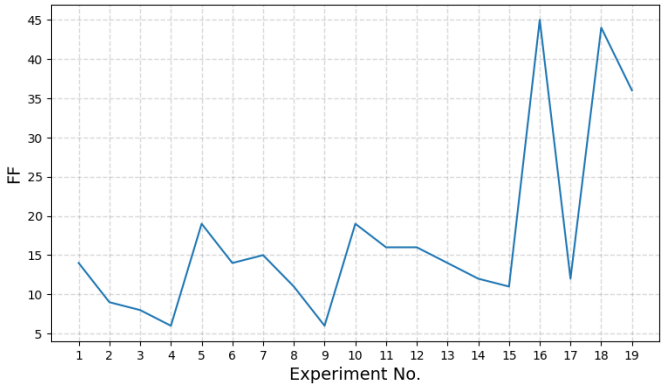

Fig. 2. FF Utilization percentage for various experiments

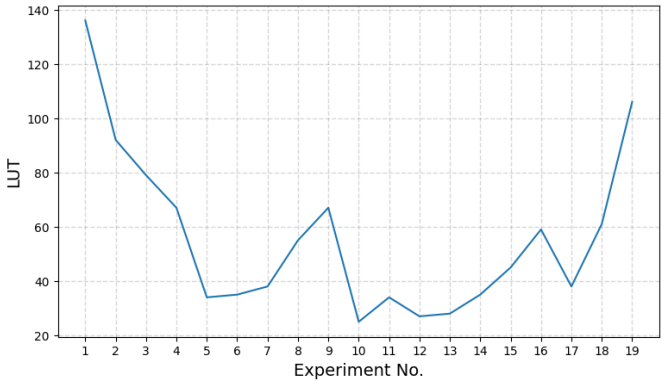

Fig. 3. LUT Utilization percentage for various experiments

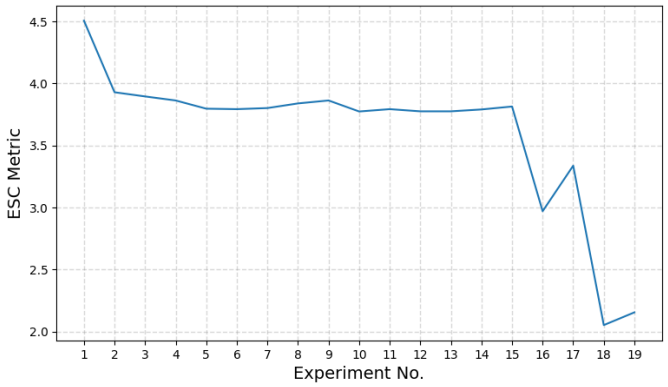

Fig. 4. ESC Metric for various experiments

IV. Conclusion

We achieved a **49**% reduction in **average latency**(after running **cosimulation** of PQCgenKAT_sign.c for 2 iterations) over the baseline. We have verified correctness of our modifications by running c-simulation for 100 iterations (We didn't do latency experiments with 100 iterations because cosimulation takes too long to run so we generated cosim report for 2 iterations and noted the values)

For area, we have achieved the following reduction percentages (which are generated as a result of synthesis). We have verified correctness for 100 iterations by running c-simulation for 100 iterations:

- **56%** reduction in BRAM utilization percentage (experiment 18, 19).
- **57%** reduction in FF utilization percentage (experiment 9).
- **82%** reduction in LUT utilization percentage (experiment 10).
- **54%** reduction in ESC metric (experiment 18).