CS 577: Project Report

Project Number :	20
Group Number:	2
Name of the top modules:	KeccakF1600_StatePermute
Link for GitHub Repo:	https://github.com/jayprakashp/VLSI-project20

Group Members	Roll Numbers
Jayprakash Patidar	194101020
Md Amir Khan	194101032
Shubham Sharma	194101047
Yash Soni	194101057

Date: 09.05.2020

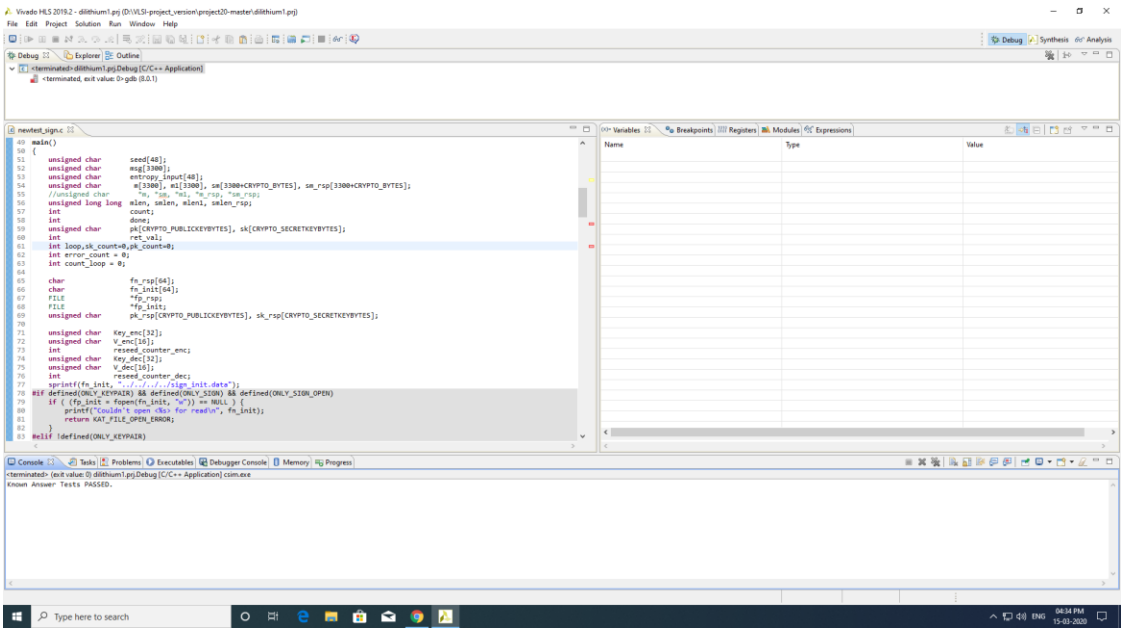
INTRODUCTION

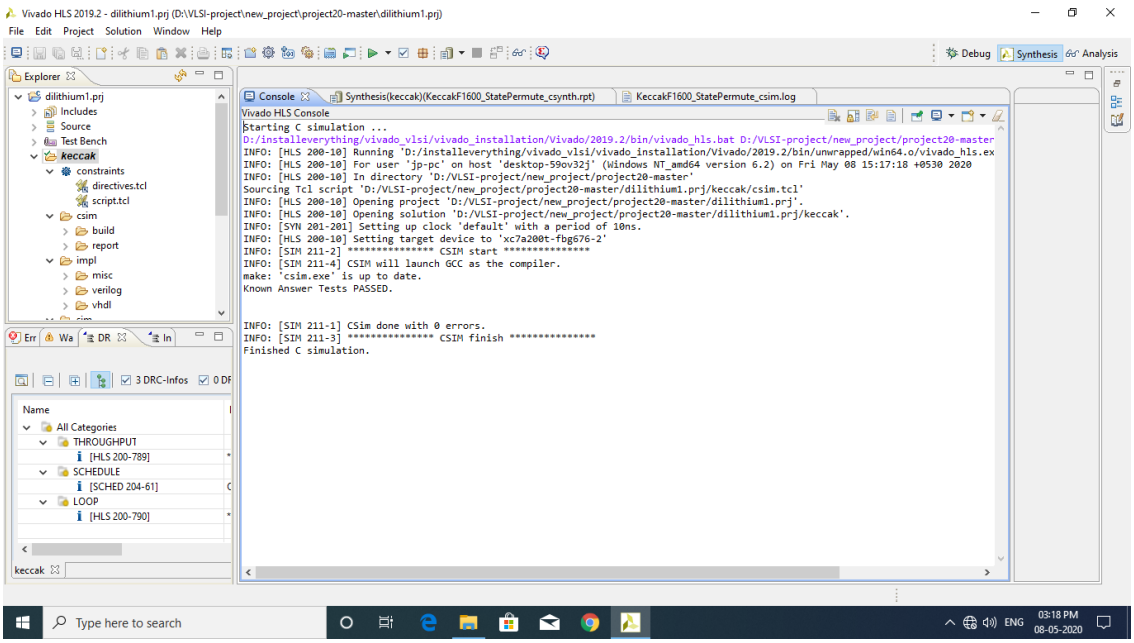
PHASE-1

1. Running the algorithm

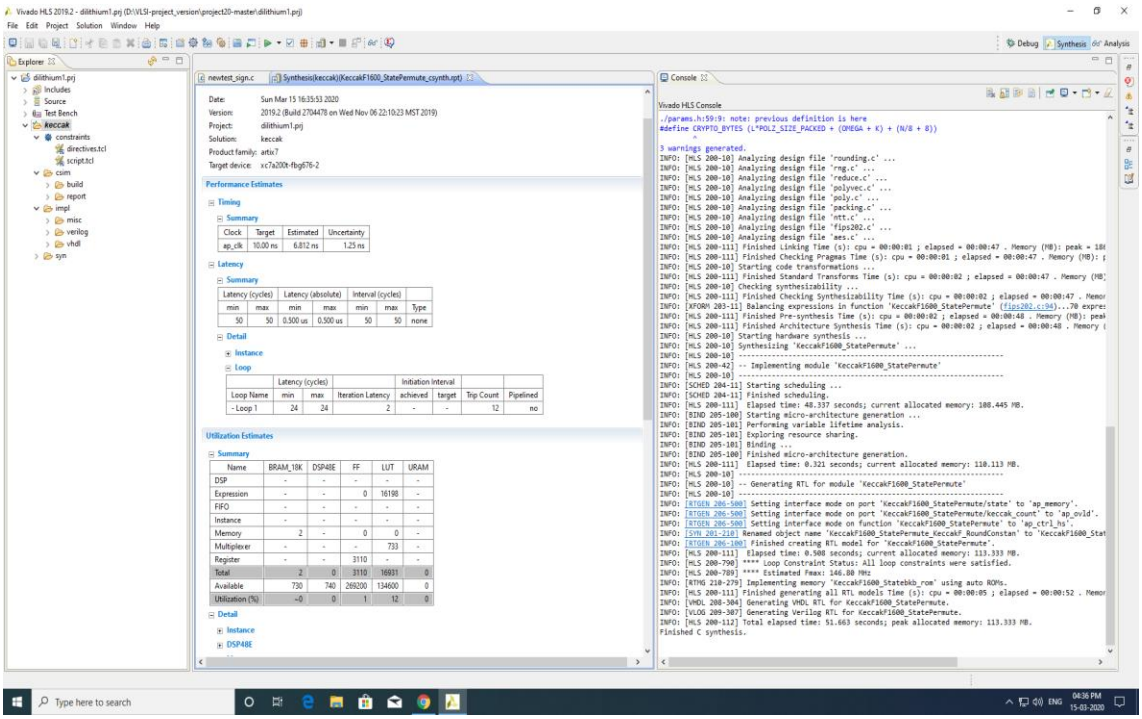
1.1 Simulation screenshot

The top module was simulated successfully. Here's the screenshot attached for the same.

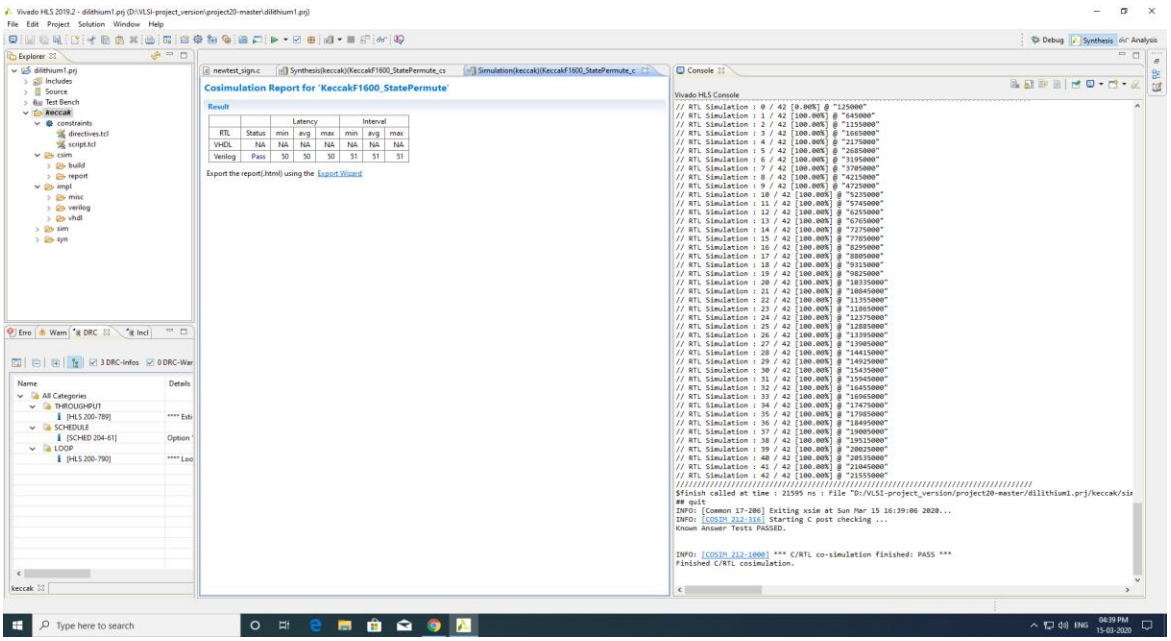




1.2 Synthesis screenshot



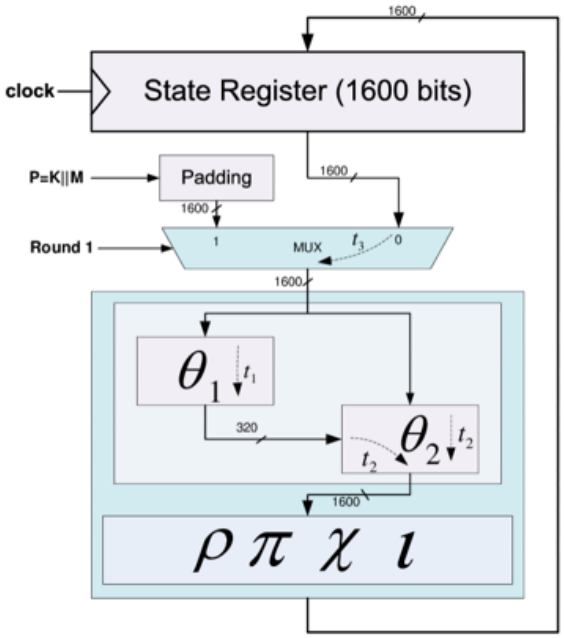
1.3 C/RTL co-simulation screenshot



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

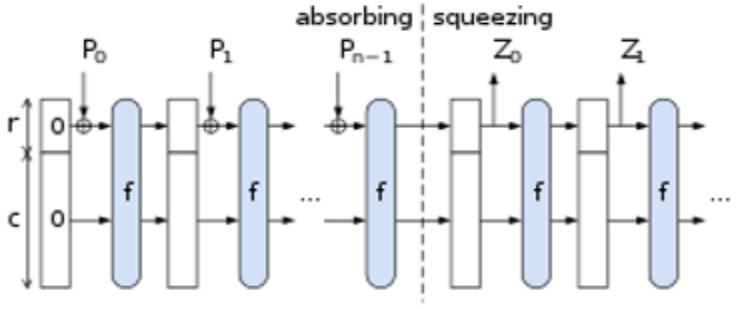
Keccak algorithm is based on an approach called as sponge construction. Sponge construction is basically a wide random function or may be called as random permutation that allows us to input any amount of data (i.e. "absorbing" in sponge terminology) and can let us output any amount of data (i.e. "squeezing"), while acting as a pseudo-random function with regard to all its previous inputs. This leads to tremendous flexibility.

The figure shows the flow of the Keccak algorithm.



Design:

SHA-3 uses sponge construction where the data is "absorbed" into the sponge and the result is "squeezed out". When the data is being absorbed in, the message blocks are XORed into a subset of state and then are transformed using a permutation function. During squeezing phase, the output blocks are read from the same subset of the state and then they are alternated using a state transform function. The state is divided into two parts. The part that is read and written is called as the "rate", while the part of the state that is untouched by the input/output is called as "capacity". The size of the security scheme is determined by the capacity as the maximum security level is half the capacity.



Given an input bit string, a padding function, a permutation function that operates on bit blocks of width, a rate and an output length, we have capacity and the sponge construction, yielding a

bit string, which works as follows:

- pad the input N by using the pad function, that results in a padded bit string P with a length divisible.
- break P into n consecutive r-bit pieces $P_0,...,P_{n-1}$
- then initialize the state S to a string of b zero bits.
- absorb the input into the state for each block P_i.
- extend P_i at the end by a string *c* of zero bits, yielding a string of length b
- XOR that string *b* with S
- apply the block permutation f to the result, yielding a new state S
- initialize Z to be the empty string
- while the length of Z is less than d
- append the first r bits of S to Z
- if Z is still less than d bits long, apply f to S, yielding a new state S
- truncate Z to d bits

Padding:

To ensure the message can be evenly divided into r-bit blocks, padding is required. SHA-3 uses the pattern 10*1 in its padding function, i.e. a 1 bit, followed by zero or more 0 bits (maximum r − 1) and a final 1 bit.

The maximum (r − 1) zero bits occurs when the last message block is r − 1 bits long. Then another block is added after the initial 1 bit, containing another (r − 1) zero bits before the final 1 bit. The two 1 bits will be added even if the length of the message is already divisible by r. In such a case, another block is added to the message, containing a 1 bit, followed by a block of (r − 2) zero bits and another 1 bit. It is necessary because a message whose length divisible by r, ending in something that looks like padding does not produce the same hash as that for the message with those bits removed.

The initial 1 bit is required so that the messages those differing only in a few additional 0 bits at the end do not end up producing the same hash. While, the position of the final 1 bit denotes the rate r that was used (multi-rate padding), which is needed for the security proof to work for different hash variants. Without it, different hash variants of the same short message would be the same up to truncation.

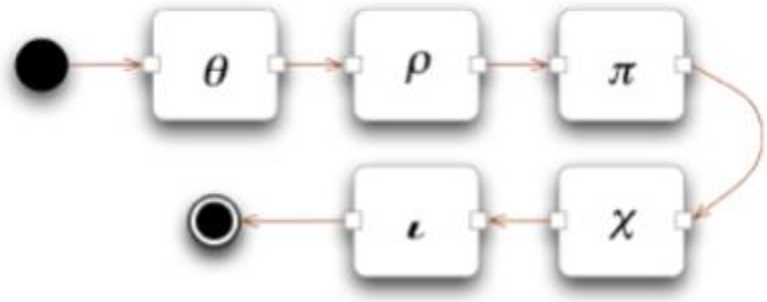
The Block Permutation:

The block transformation f, which is Keccak-f[1600] for SHA-3, is a permutation which uses AND,

X-OR and NOT operations, and it is designed for an easy implementation for both software and hardware.

The state can be considered to be a 5 × 5 × w array of bits. Let a[i][j][k] be bit (5i + j) × w + k of the input. Index arithmetic is performed modulo 5 for the first two dimensions and modulo w for the third dimension.

Keccak uses 24 permutation rounds to reduce the message text into a hash. Each round invokes five modules in succession as follows:



The theta module renders the internal state into a 5x5 array of 64-bit elements. It computes the parities of each column and combines them with an exclusive-or (XOR) operator. Then it XORs the resulting parity to each state bit as follows:

S[i][j][k] ^= parity(S[0...4][j-1][k])

^ parity(S[0...4][j+1][k-1])

where i = 0...4; j = 0...4; k = 0...63

The rho module rotates each 64-bit element by a triangular number. However, it excludes element S[0][0] from rotation. The phi module permutates the 64-bit elements. Permutation follows the fixed pattern assignment as shown below:

S[j][2*i + 3*j] = S[i][j]

The chi module adds a non-linear aspect to the permutation round. It combines the row elements using the three bitwise operators: AND, NOT, and XOR. Then it writes back the result to the state array as follows:

S[i][j][k] ^= ~S[i][j + 1][k] & S[i][j + 2][k]

The iota module breaks up any symmetry caused by any of the other modules. It does this by XORing one of the array elements to a round constant. The module has 24 round constants to

choose from. These constants are defined internally by Keccak.

3. Result

FPGA Part	Name of Top Module	FF	LUT	BRAM	DSP	Latency	II
	KeccakF1600_StatePermute	3110	16931	2	0	50	51

3.1 Explanation of result

In this project our top function is KeccakF1600_StatePermute , which permute the input states and output the permuted states ,for which algorithm is described above .

- In top function total 3110 flip-flops is used as register.
- 16931 Lookup table is used in which 733 is used for multiplexer and 16198 is used for Expressions .
- 2 BRAM is used as memory to store 24 Keccak round constants.
- No DSP(data signal processing) is used in this project.
- Latency of the top function is 50 cycles.
- Initialization interval (II) requires 51 cycles.

3.2 **Problems and its solution**

During working on the project to optimize the top module code, there were some issues that were needed to be addressed in order to optimize the code and are described briefly below:

- Test Folder was not available:

  When we ran the code for the first time, there was no 'test folder', which led us to compilation errors.

  Solution:

  We informed Chandan Sir about the issue and asked him to help us out. We received the test folder and as a result, the code was compiled successfully.

- Increase in clock period greater than target clock period:

Our aim was to reduce latency and in our top function mainly the loop was responsible for latency so we tried doing loop pipelining, but this did not bring any significant changes in the latency but it almost halved the clock period .

Now, this situation gives an idea of using loop unrolling pragma, we unroll the loop by factor=2, this results increase in clock period and small decrease in latency . Now we try for factor =3,4,6,8,12 as these all factors of 24 (24 is the loop iteration count). Now, as we increase the factor, the clock period also starts increasing and latency starts decreasing but this will lead to clock period end up being greater than target the clock period.

So we need to find a factor which unrolls the loop in such a way that latency decreases and also the clock period does not exceed the target clock period, and we achieved it at factor = 2 of unrolling the loop.

- Complete array partitioning leads to large Look-up table:

In order to optimize the code for the top module, initially we performed complete array partitioning. This led to generation of a large look up table that was not easy to handle.

Solution:

In order to keep the look up table of considerable size such that it remains easy to handle, we performed blocked array partitioning.

- Loop unroll led to increase in the required number of registers:

There are 24 rounds of iterations for the top module and each requires some registers. When loop unrolling was performed, it led to increase in the number of registers and hence increased the register count.

Solution:

The team members decided not to perform the 'loop unrolling'. This resulted in decrease of register count leading to Resource optimization on the cost of some increase in the latency.

PHASE-2

You need to update the below table with your results. The grading primarily depends on the efforts in optimizing the design. You should target (i) Resource optimization (ii) Latency Optimizations. Note that both cannot be achieved at the same time. For each optimization strategy, you need to explain why that optimization gives you performance benefits. Do not apply optimizations blindly. For each case, uploading

the synthesis results in the github directory is MUST. We shall cross check the data you entered in the table with the synthesis results. You won't get marks if the results are not uploaded in the github or below data mismatches with the actual results.

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

Benchmark	Type (Area / Latency)	Resource Utilization				Latency		Major Optimizations
		LUT	FF	DSP	BRAM	No of Clock cycle / latency	Clock period	
Baseline	Baseline	16931	3110	0	2	50	6.812ns	Baseline
Optimization 1	Area	4555	2401	0	8	2785	8.002ns	LUT,FF
Optimization 2	Latency	16950	3108	0	2	39	6.812ns	latency

Explanation for optimization-1:

- In this optimization our was aim to optimize the resources. In our top function resources used were mainly in the form of variables. There were 60 variables used in loops so we tried to reduce the use of variables to optimize the use of registers.
- This gave us the idea to use of an array in place of the variables and also input to the function is an array name- state[25] and from this array 25 variables are initialized and rest all variables are used as intermediate storage in the loop.
- So we decide to use state[25] array directly in the loop and in place of intermediate variables we used two small arrays var1[5] , var2[5] and one more array was used of same size as of input array(state[2]) i.e. state2[25] to store the output of current iteration used as input to next iteration.
- These implementations of array gives us idea of rewriting the loop body in the form of small

inner loops, this was possible only because of the array implementation .

- So we Re-write inner loop code in form of small loops in such a way that we can apply pipelining on inner loops , this will control increase of iteration latency because of array access.
- In this way we can achieved resource optimization on cost of increase in latency .
- We also tried to apply pipeline on our main outer loop to reduce the increased latency to some extent. This reduced the latency but it also led to increase in number of resources to a large number that even crossed the available number of resources and that would not fit on target FPGA board , so we did not apply pipelining on the outer main loop.
- We also thought to apply many other optimization techniques to further more optimize resource and latency like array partitioning, loop unrolling etc. But this would not significantly optimize the code because of array accessed in loop is iteration-wise not collectively so array partition not affect so much on our top function.

Explanation for optimization-2:

- In this optimization our aim is to optimize the latency and in top function loop is the main cause of latency . So we have to optimize the loop code to reduce latency. For this we have to better utilize the available resources.
- So to reduce latency we think to apply pipelining on the loop with resolving iteration dependency, this result not much optimize the latency but this reduced the clock period significantly, so now we aim for reduce the latency with some increase in clock period but also in limit with target clock period.
- We think to unroll the loop since available FF and LUT is far more than used resources. So we start unrolling the loop by factor=2,3,4,8,6,12 and also total unroll , we use these factors because they are multiple of 24 (24 is the loop trip count) , and select the best factor that help us achieving our optimization goal. As we increase the factor of unrolling we faced a problem of increased clock period that exceeds target clock period.
- After trying on all the possible factors to unroll, we see that for factor=2 gives the best optimization under limit of target clock period with total average clock cycles=39.
- In this way we optimized our top function from baseline latency 50 clock cycles to 39 clock cycles.