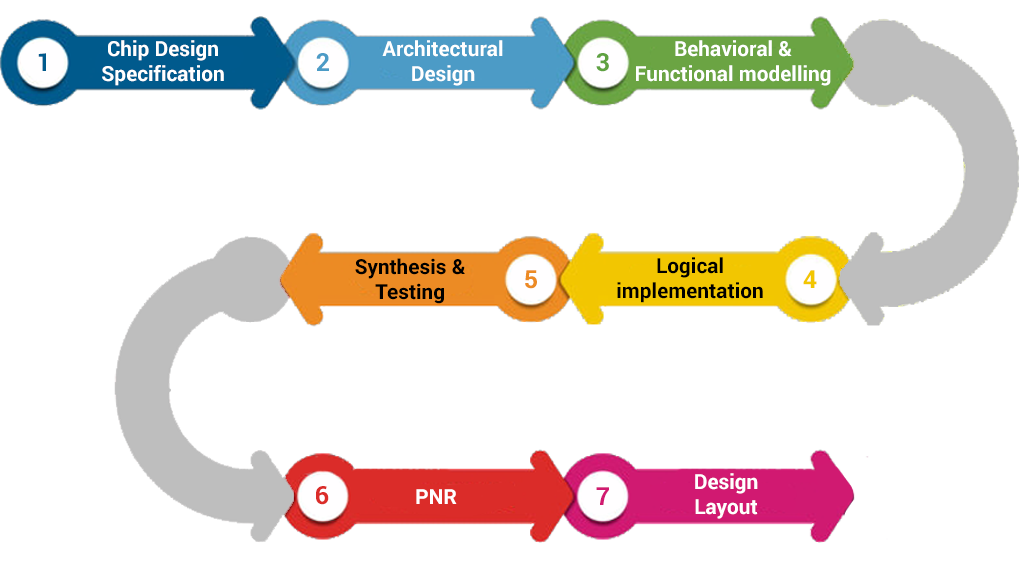
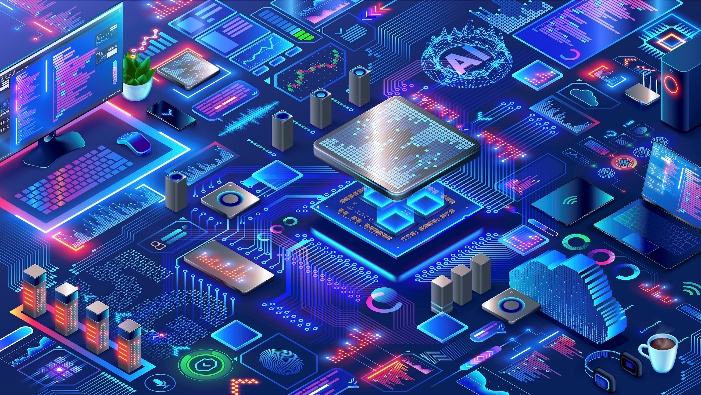
A picture containing shape

Description automatically generatedECE330 System On Chip Design

AES Encryption/Decryption System with Error Detection capabilities: From RTL to Post-PnR





Logo

Description automatically generated with low confidence



Kyritsis Spyridon

Topali Konstantina

Introduction

The goal of this course is to implement an Application Specific Integrated Circuit (ASIC), using the ASIC flow methodologies and design tools. The design tools that are being used for the testing of our design (gate-level behavioral simulation, synthesis etc.) are the Cadence© Incisive™, Synopsys© Design Compiler™, Cadence© Innovus™ and Synopsys© PrimeTime™.

In this assignment we use as the main RTL code a 128-bit encipher/decipher circuit, from GitHub (<https://github.com/secworks/aes>), author: Joachim Strömbergson).

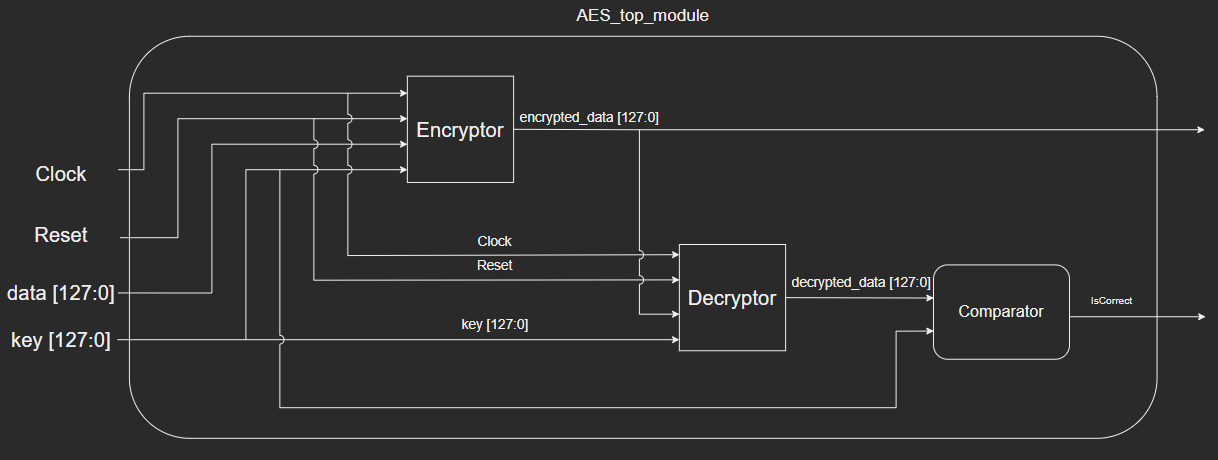
We will transform the provided core into an encryption/decryption system and will pass our new core through the ASIC design flow.

This means that, we will carry out the behavioral simulation through Incisive™, we will use Design Compiler™ for synthesis, timing, area, power analysis, we will produce the Pareto Curves for different clock-periods and we will use Cadence© Innovus™ for placement, routing and extraction of post-route PPA reports as well as .spef and .sdf files. These files will then be passed as inputs in Synopsys© PrimeTime™ tool that is used for post-PnR timing and power reports and finally Incisive™ to make sure that the post-route and place simulation is working as expected.

RTL Modification and System Ensemble

In this part, we connect the encipher and decipher modules in series and produce an extra module called aes\_compare, to check if the output of the decipher is equal to the input block that comes from the testbench.

Our goal is to verify that the encryption and decryption modules function correctly. The following block diagram, displays the functionality of our RTL code:



The encipher takes as input a 128-bit plain text data sequence, as well as the 128-bit encryption key. It encrypts the data and the encrypted data, are passed as input to decipher together with the initial key, that originates from the testbench. The 128-bit decrypted data, as well as the initial data that originating from the testbench, are then compared, in order to make sure that they are equal.

Software Model VS Behavioural Simulation

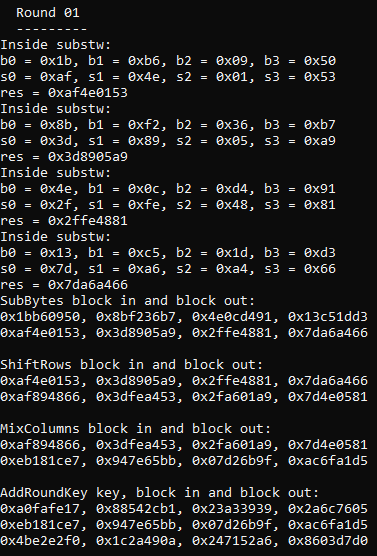
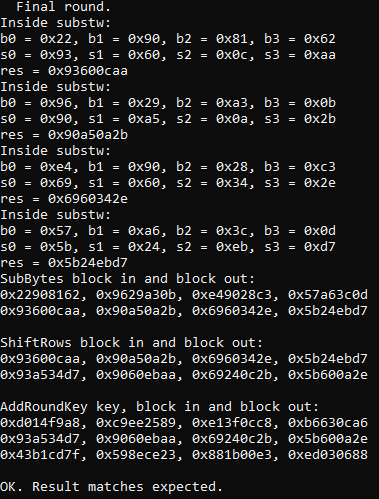
In this part we test the functionality of our circuit at algorithmic level. Firstly, we create the expected output data with the software model written in python, provided by the author. The software model produces the following results:

Text

Description automatically generatedA picture containing text, plaque

Description automatically generated

**….**



For the behavioral simulation we will be using Incisive™. Incisive™ is a suite of tools created by Cadence© Design Systems and is related to the design and verification of ASIC’s, SoC’s and FPGA’s. The following waveforms correspond to the behavioral simulation of our circuit:



We observe that the signals decipher\_output and core\_block are the same. Furthermore, the signal outputs\_comp\_result, becomes equal to one when the previous signals are equal, otherwise remains equal to zero.

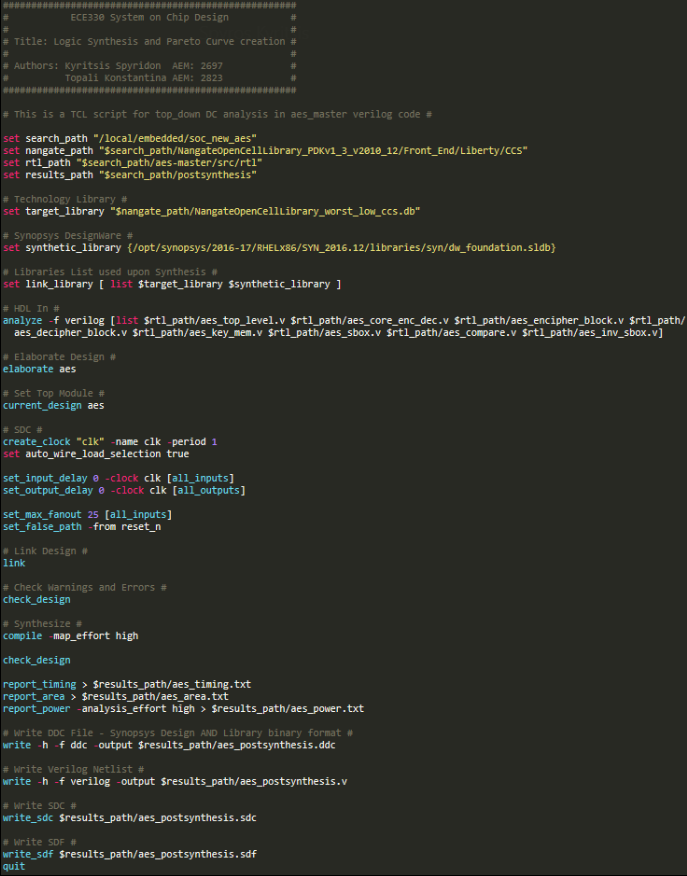
Logic Synthesis

In this chapter we synthesized our design from the RTL code to a gate-level netlist. To do this we used Synopsys© Design Compiler™. Our goal is to synthesize our design in 2 distinct ways. Firstly, by using a top-down approach and then by using a bottom-up approach. The difference in approach lies in the order in which our design’s submodules are synthesized. The different compilation order allows the Design Compiler™ to employ different optimization strategies so that it better explores the solution space.

To synthesize our design, we created TCL scripts for top-down and bottom-up.

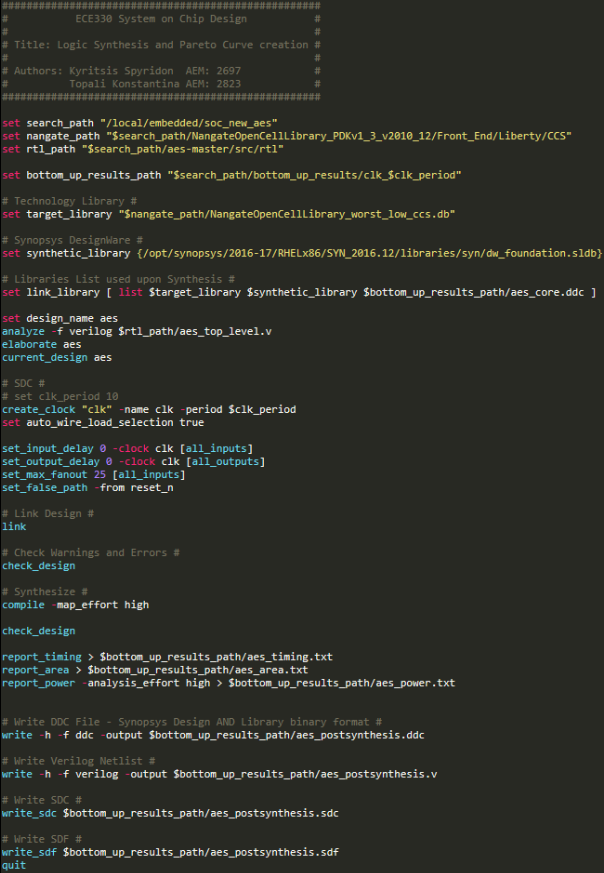
For the top-down approach, we created the following automated script using the TCL scripting language. The script performs multiple synthesis runs with the same design constraints, while increasing the clock frequency up to a point. The script collects log files regarding the timing, area and power of the synthesized design and produces as output files: the synthesized gate-level netlist, a .sdc file containing the Synopsys© Design Constraints for our design, a .sdf file which will be used to delay annotate our design in the post-synthesis simulation and a .ddc which is a design and library binary file that can later be used by Design Compiler™.

The TCL script is the one presented below.

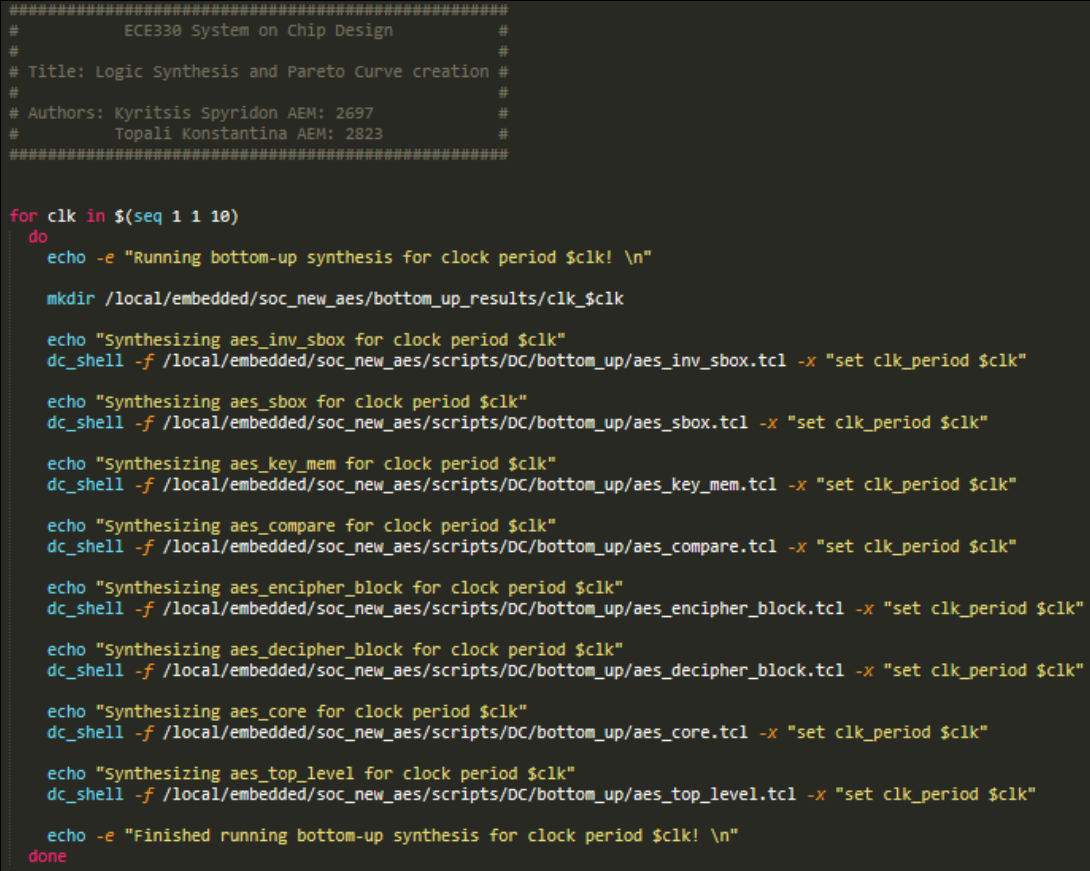


For the bottom-up approach, we created a TCL script to synthesize each unique submodule in our design. Our goal was to synthesize the design in a bottom-up manner and produce a .ddc binary file for each unique submodule. The produced .ddc files should then be linked as we go through the different hierarchy layers to enable Design Compiler™ to perform different optimizations.

In the screenshot below, we can see the bottom-up synthesis script of our aes\_top\_level module.



In order to automate all the bottom-up synthesis runs, we also created a bash script, that sweeps the clock\_period parameter starting from 1ns to 10ns, at 1ns increments. The bash script is the following.

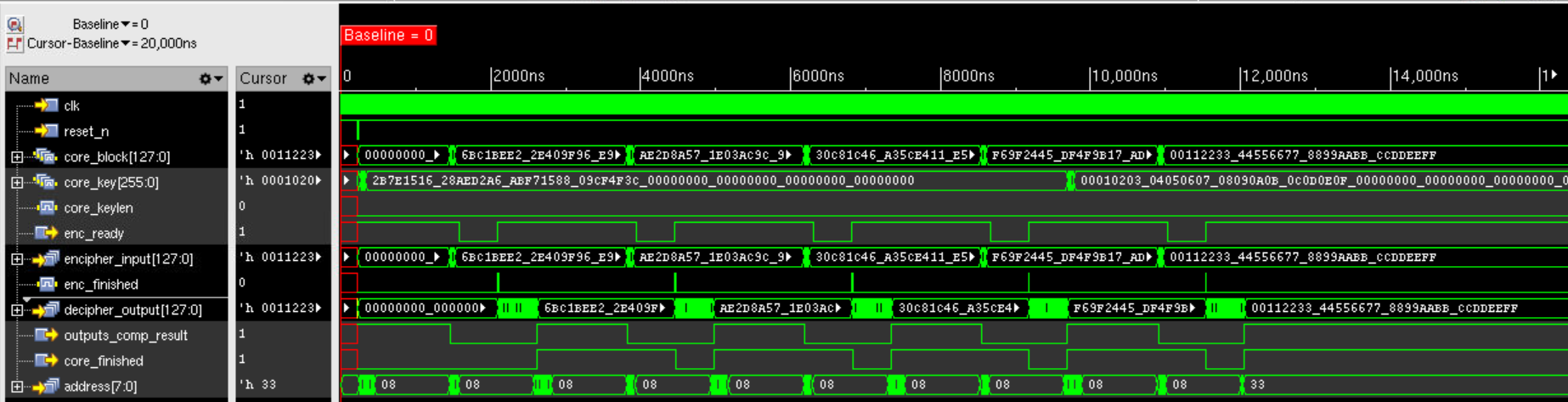


Post Logic Synthesis Simulation

For this part we had to simulate our design in the Incisive™ with and without .sdf annotation.

We procced by testing the design without sdf annotation, using a clock\_period of 5ns for the top-down and the bottom-up analysis. This came true by compiling the Verilog (.v) files from the Nangate library, postsynthesis and aes\_top\_level testbench and then elaborating and simulating the design.

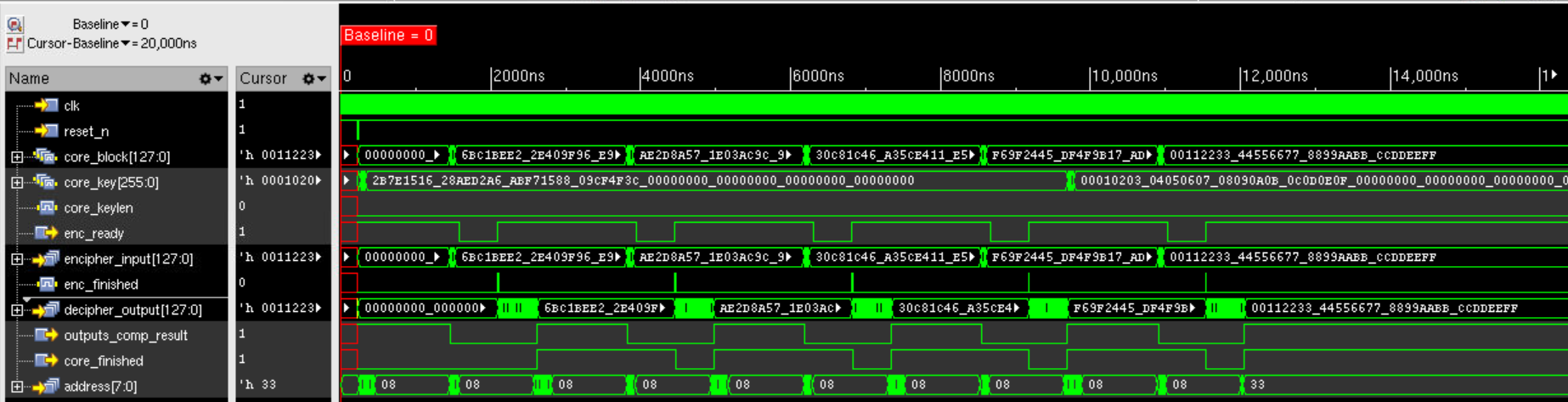
The waveform below shows the simulation from both the top-down and bottom-up synthesis without sdf annotation.



The results are clearly the same as the behavioral simulation. The design works for the post-synthesis simulation without SDF annotation.

We procced by testing the design with SDF annotation, using a clock\_period of 5ns for the top-down and the bottom-up analysis. This came true by compiling the Verilog (.v) files from the Nangate library, post-synthesis and aes\_top\_level testbench, which includes the SDF annotation Verilog command ($sdf\_annotate (path\_including\_sdf, tb\_aes.dut)) and then elaborating and simulating the design.

The waveform below shows the simulation from both the top-down and bottom-up synthesis with SDF annotation.



Pareto Curves Extraction

The extraction of the pareto curves was made by using EXCEL and the area, power and timing (PPA) results from the top-down and bottom-up synthesis. Concretely, we produced three different pareto curves for the area, power and timing (slack) as the y-axis and the clock period as the x-axis.

From the resulting pareto curves we can observe that our synthesis scripts adequately explore our circuit’s solution space. The scripts provide solutions that meet different PPA (Power Performance Area) criteria, depending on the input clock frequency we provided to Design Compiler™.

The results our analysis provided, with one exception, are the ones we were expecting, based on the given material and existing literature.

The one exception we observed lies on our area curve. The Synopsys© Design Compiler™ provides a more or less stable area for the synthesized result, for the data points in the middle of the curves, while largely deviating at the graphs endpoints. For the slowest clock period we provided, i.e 10ns, there is a large spike in area. This is most likely due to the Design Compiler™ trying to perform different optimizations to further reduce the power consumption, at the cost of area.

On the other end of the graph, i.e 1ns clock period, we observe another large spike in the area. This can be explained by looking at the slack graph. In an attempt to meet timing for an extremely fast clock period, Design Compiler™ tries to reduce the logic levels of the circuit, which is expensive in terms of area, and the Design Compiler™ is ultimately not able to meet timing.

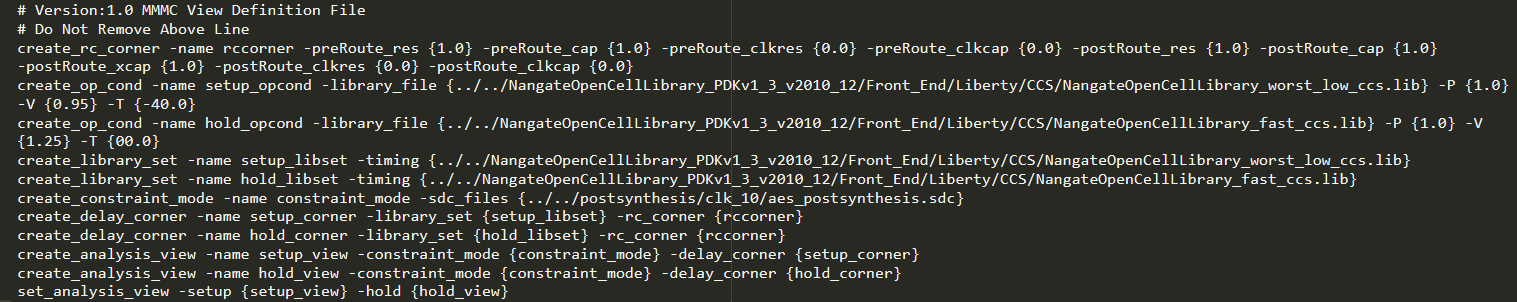
For the top-down analysis these three curves are shown below.

As in the top-down analysis, we also produced the same pareto curves for the bottom-up analysis. The bottom-up analysis gives more or less the same results as the top-down one, with one small exception. The bottom-up scripts do not produce the same area spike up we observed at the top-down script.

Our bottom-up pareto curves can be seen below.

As clock\_period is increasing slack is increasing as well, as we expected.

Back-end Physical Flow

In this part we import our design in Cadence© Innovus™. We produce the MMMC (Multi Mode Multi Corner) file that gives the analysis of the design over varied process corners. It describes the RC corners (resistances and capacitances), the operating conditions (temperature, voltage, process), the Nangate library files (worst\_low\_ccs.lib and fast\_ccs.lib) and constraints for hold and setup time and includes the .sdc file from the post-synthesis step for a target clock period of 10ns. The MMMC file is the following.

Firstly, we have to define our core area. The design is placed inside a core with core offsets 5μm and utilization 50%. The second important step we have to make is to create a power plan of our design. We used metal 1 for all horizontal power lines and metal 2 for all the vertical ones as described in the technology .lef file. For the power rings, we used the proposed width of 1.8μm. We also added vertical power stripes (also in metal 2) for our design, to help combat IR drop. The width of these stripes was set to 1μm. After that the Special Route command was issued through the GUI from Innovus™.

In the picture below we see the design state after the Power Planning phase.

A picture containing building, window blind

Description automatically generated

Following the completion of the Power Planning phase we need to place the standard cells of our design.

Below we can see the post-placement phase of the design with and without the metal layers visible. In the figure with the metal layers enabled, we observe a first trial route as performed internally by the tool during the placement process.

A screenshot of a computer

Description automatically generated with low confidence

In the figure below, where the metal layers are not visible, we can clearly see that there are clear empty spaces below the areas where the power stripes are placed, as indicated by the red ellipses in the figure. Upon further inspection, the cells that are placed under the power stripes cause a lot of DRC violations. To combat this, we introduced hard placement blockages under the regions the power stripes were placed.

Text

Description automatically generated

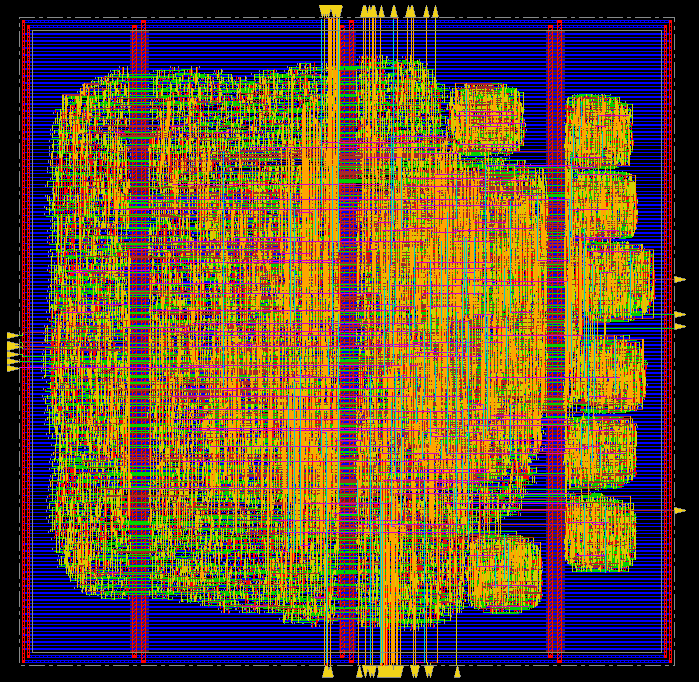
Below we can see the placement state after the blockages have been inserted and an incremental placement run has been performed along with the pre-CTS optimizations.

A picture containing text, outdoor

Description automatically generated

After placement was finished, it was time to create our clock tree. This was achieved through the Innovus™ shell.

The Clock Tree Synthesis results can be seen in the figure below.



Having concluded Clock Tree Synthesis, we shall now proceed to the Routing phase.

The design in the post-Routing phase looks like this.

A picture containing text, light, building, lit

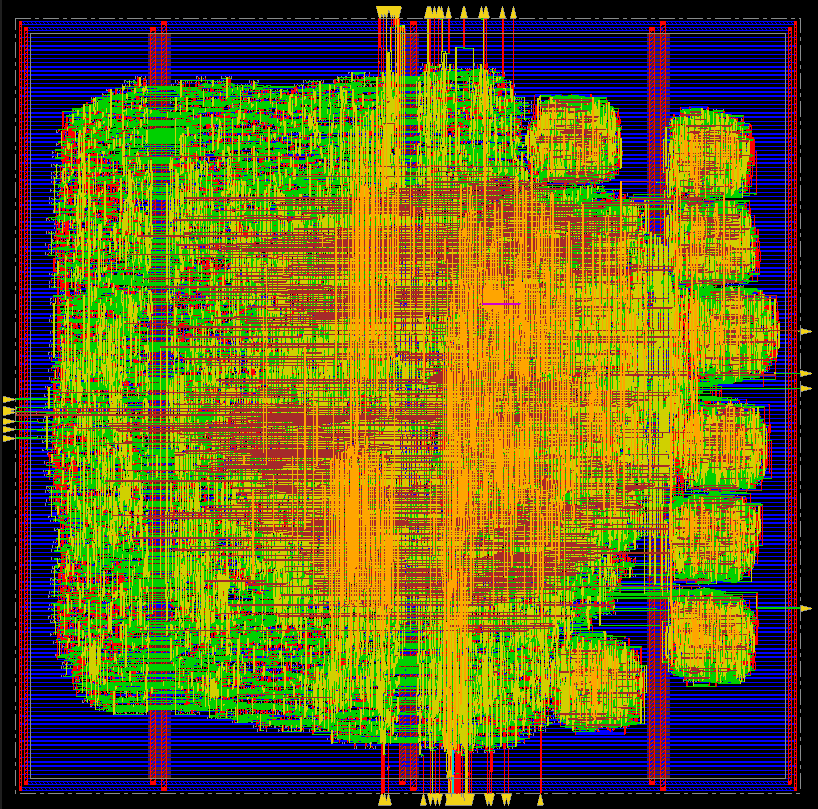
Description automatically generated

It is apparent by the white Xs on the design that our design contains DRC violations. These occurred due to power stripes not being connected with vias to their respective power line with vias across the power rails. To fix that, we simply added the missing vias.

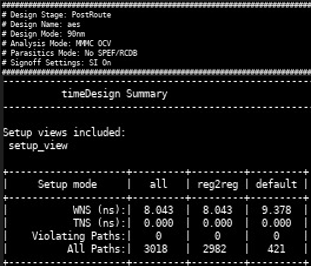
By running a timing analysis through Innovus™ we noticed that, although our design meets setup timing, it fails hold time. Therefore, an ECO optimization is needed to ensure that timing is MET.

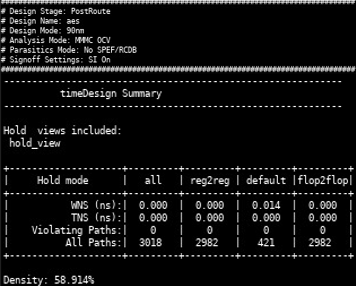
Finally, after the ECO optimization aiming to fix the hold timing violations, our final placed and routed design meets timing and is DRCs free. The post-placement and route setup time slack (WNS) is 8.043ns (MET) and the hold time slack (WNS) is 0 ns. The metals that were used range from metal 1 to metal 8.

The final placed and routed design is the following.

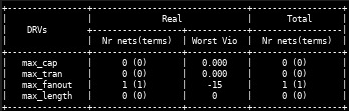


Below are the setup and hold reports that extracted from Innovus™.



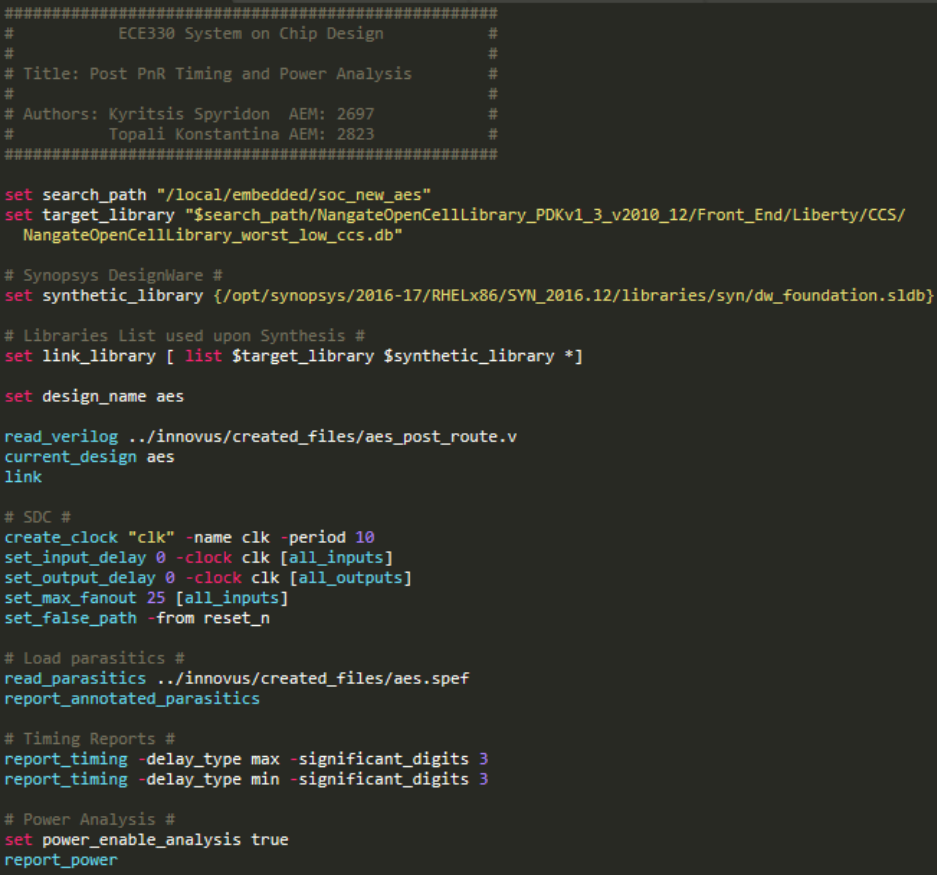


While running for DRVs report we came up with a max\_fanout DRV. It was due to the max\_fanout constraint 25 we had from the bottom up and top-down tcl scripts from DC that then produced the sdc file we imported in the MMMC file of Innovus™. This problem can be solved if we just increase the maxfanout constraint in the sdc file by 15.

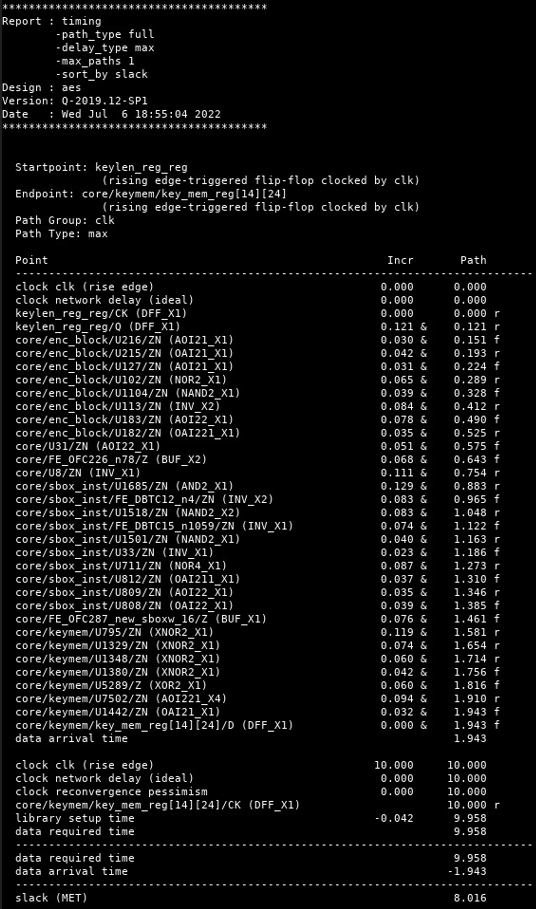


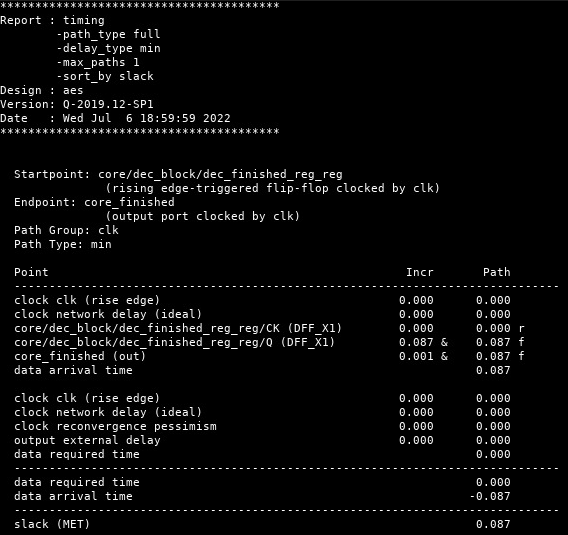
Verification of timing and power results with parasitics

The next step after extracting .sdf and .spef files from Innovus™ is to verify our timing and power using PrimeTime™ by Synopsys©. In order to run the PrimeTime™ tool, we implemented a TCL script that sets the search and Nangate paths and the target, synthetic and link libraries where we have the .db and .sldb files from Nangate and the netlist extracted from Innovus™ after placement and routing. We also provide design constraints that describe the clock period, input and output delays, max fanout and we set the reset path as a false path. After that we read the parasitics (resistance and capacitance) from the .spef file extracted from Innovus™ and we extract the setup and hold timing and power reports.



The reports that describe timing and power are the following.





Text

Description automatically generated

We observe that with clock period 10ns, the setup slack (WNS) is 8.016 ns and hold slack (WNS) is 0.087 ns and do not differ much from the timing reports of Innovus™. The total power is equal to 2.268 mW.

Post Place and Route Simulation

In the final part we need to compare our post synthesis design simulation with the post-placement and route. For this reason, we provide the post-route and placement .sdf file, the post-PnR netlist of the aes circuit, the testbench and the Nangate netlist in the Incisive™ tool to make sure that the design is properly working after placement and routing. The simulation as we expected is the same as in the post-synthesis simulation, thus the design is properly working.

A screenshot of a computer

Description automatically generated

Problems occurred

Εικόνα που περιέχει κείμενο, εσωτερικό, στιγμιότυπο οθόνης

Περιγραφή που δημιουργήθηκε αυτόματαThe following screenshots, present the main problems we were facing with our post-synthesis design:

As we can observe in the screenshot above, there is no logic between testbench and aes\_top\_level module, yet when tb\_reset\_n becomes equal to zero, reset\_n becomes equal to x.

We tried multiple different approaches to solve this issue, most notably by changing the synthesis’ script parameters. Unfortunately, all such attempts, reached a dead end and the issue remained unsolved.

We also tried running the 2018-2019 version of Design Compiler™, which is also available for the purposes of our class, to see if somehow the tool’s version is related to the unexpected behaviour we were observing. By doing this we observed the following:

Εικόνα που περιέχει κείμενο

Περιγραφή που δημιουργήθηκε αυτόματα

The 2018-2019 version of Design Compiler™ is facing a fatal error. The crash is persistent through multiple runs of the tool. Thus, we proceed our testing with the previous version of the Design Compiler™ 2016-2017.

This problem seems to have been produced from the Nangate standard cell library that we are using to synthesize our design. We, unexpectedly, managed to resolve this issue by making all flip flops synchronous from asynchronous.

After resolving the reset issue described above, we were faced with yet another problem. The post-synthesis simulation, even without the SDF annotation, still does not produce the expected results. The following waveform shows the current state of our post-synthesis simulation.

Timeline

Description automatically generated with medium confidence

The above problem solved by using compile with -map\_effort high in the top-down and bottom-up synthesis as well as deleting all the gate clocks creation from our scripts.