

Ben-Gurion University of the Negev Faculty of Engineering Science

School of Electrical and Computer Engineering Dept. of Electrical and Computer Engineering

Fourth Year Engineering Project Final Report

Project name - Digital IC Synthesis of I2C - SPI bridge block

|  |  |
| --- | --- |
| **Project number:** | **p-2024-078** |
| **Students (name & ID):** | **Maayan Dagan 205884042**  **Noam Guez 318854858** |
| **Supervisors:** | **Dr. Matan Gal-Katziri** |
| **Submitting date:** | **17/08/2024** |

**Table of content**

[abstruction 3](#_Toc173847481)

[Hebrew 3](#_Toc173847482)

[English 4](#_Toc173847483)

[Project Objectives 5](#_Toc173847484)

[Introduction 6](#_Toc173847485)

[The approach taken for the solution and the engineering design 7](#_Toc173847486)

[Step One- 7](#_Toc173847487)

[Step Two- 9](#_Toc173847488)

[Step Three- 15](#_Toc173847489)

[Step Four- 17](#_Toc173847490)

[Step Five- 19](#_Toc173847491)

[Step Six- 20](#_Toc173847492)

[Follow up for future progress 21](#_Toc173847493)

[Finishing the Logical Synthesis stage: 21](#_Toc173847494)

[Start of the Backend design process: 21](#_Toc173847495)

[How to progress in more details 23](#_Toc173847496)

[How to start working with Cadance tools: 23](#_Toc173847497)

[How to DFT: 23](#_Toc173847498)

[How to Manual Check: 24](#_Toc173847499)

[Sources 27](#_Toc173847500)

# abstruction

## Hebrew

תכנון של מעגל משולב להמרה בין פרוטוקול I2C לפרוטוקול SPI

שמות הסטודנטים: מעיין דגן, נועם גז *מייל:* [*dagan131313@gmail.com*](mailto:dagan131313@gmail.com)שם מנחה: ד"ר. מתן גל קצירי

בעולם התקשורת קיימים שני פרוטוקולים שימושיים הנקראים I2C ו.SPI- הפרוטוקול I2C משמש לתקשורת ללא מגע בצימוד השראתי, ופרוטוקול הSPI- שימושי לניהול מערך רגיסטרים, לקריאה וכתיבה. מהות הפרויקט היא לתמוך במחקרו של ד"ר מתן גל קצירי, ליצור מעגל RFIC ולהקים מתודולוגיה שימושית לצוות המנחה ליצירת רכיבי VLSI בעצמם.

מטרת הפרויקט שלנו היא בניית מתאם המתרגם בין שני רכיבים, כאשר לכל רכיב פרוטוקול תקשורת משלו, ומבצע תרגום והעברת פקודות ומידע בין אחד לשני. בניית הרכיב דרשה תכנון לוגי (בעזרת שפת Verilog) יצירת סביבת

וריפיקציה לבדיקה לוגית, סינתזה לוגית ומימוש פיזי (implementation).

לצורך מימוש הפרויקט יצרנו דיזיין לוגי אשר נכתב בהתאם לדרישות הלקוח, שבמקרה זה הוא ד"ר מתן גל קצירי. פיתחנו מתודולוגיות בעזרתן הקריאה והכתיבה בין הרכיבים תעבוד בהתאם לדרישות. לשם כתיבת הדיזיין השתמשנו בשפת ,Verilog ולשלב הווריפיקציה בשפת .SystemVerilog לשלבי הסינתזה והbackend- (שהוא שלב ה- implementation) השתמשנו בכלים של Cadence הנקראים Genus וInnovus- בהתאמה.

Synthesis

,Innovus

,Cadence, Genus

,SystemVerilog

,I2C, SPI, VLSI, Verilog

מילות מפתח:

Design

## English

Digital IC Synthesis of I2C - SPI bridge block

Student Names: Maayan Dagan, Noam Guez

*Email:* [*dagan131313@gmail.com*](mailto:dagan131313@gmail.com)

Advisor's Name: Dr. Matan Gal Katziri

In the world of communication, there are two useful protocols called I2C and SPI. The I2C protocol is used for contactless communication via inductive coupling, while the SPI protocol is useful for managing register arrays for reading and writing.

The essence of the project is to support Dr. Matan Gal Katziri's research, create an RFIC circuit, and develop a useful methodology for the advisor’s team to create their own VLSI components.

The purpose of our project is to build an adapter that translates between two components, each with its own communication protocol, translates and transfers commands and information between them. Building the component required logical design (using Verilog), creating a verification environment for logical testing, logical synthesis, and physical implementation.

For the project's implementation, we created a logical design that was written according to the client's requirements, in this case, Dr. Matan Gal Katziri. We developed methodologies that ensure reading and writing between the components work according to all the requirements. For writing the design, we used Verilog, and for the verification stage, we used SystemVerilog. For the synthesis and backend stages (which are part of the implementation), we used Cadence tools called Genus and Innovus, respectively.

At the beginning of the year, we expected to reach the end of the design writing stage and start the verification stage. We managed to progress beyond expectations and even begin the backend stage and start working with the Cadence tools.

Keywords: I2C, SPI, VLSI, Verilog, SystemVerilog, Cadence, Genus, Innovus, Synthesis Design

# Project Objectives

Practical Objective: Our goal in this project is to develop a physical adapter that translates communication between the I2C protocol and the SPI protocol. This module is desired and defined by our advisor, Dr. Matan Gal-Katziri, for his own research purposes. While such an adapter is commonly available for purchase and easy to integrate, Dr. Gal-Katziri has a broader objective beyond merely creating an adapter: establishing a methodology for the self- fabrication of logical modules by his team.

Methodological Objective: This is the more critical objective as it will significantly impact the future work of Dr. Gal-Katziri's team. We aim to investigate and independently navigate the entire process of designing, verifying, synthesizing, and implementing a physical module. Through this project, we will learn practical techniques, identify where tools are lacking, and document a follow-up and solution process. This will enable Dr. Gal-Katziri's team to create physical modules independently using the available resources in the future.

# Introduction

The primary goal of this project is to develop an adaptor for Dr. Matan Gal-Katziri's RFIC circuit. This project is significant as it allows us to create a physical module from scratch, establishing a methodology for VLSI self-implementation in Dr. Gal-Katziri’s team.

The process involves several steps: logical design, simulation verification, synthesis, and physical implementation. Each of these steps is crucial to ensure the module functions correctly and meets the required specifications.

We are using Verilog and SystemVerilog for logical design and simulation verification. Additionally, we employ tools like Cadence Genus and Innovus for the Place and Route (PNR) stage and other backend processes. These tools help us map our designs onto physical hardware accurately and efficiently.

With the resources and support from our advisor Dr. Gal-Katziri and his environment, we are progressing in our project to achieve our physical printing objective. This work enhances our skills and knowledge in VLSI design and implementation, preparing us for future projects.

# The approach taken for the solution and the engineering design

The solution stages: I2C ←→ SPI Adapter

As described in our solution proposed at the start of the year, we will now describe broadly each stage we passed this year, the difficulties we've surpassed and the solution we produced.

## Step One-

theoretical understanding of the protocols, the Verilog Coding language and building the solution program. (09/23–01/24)

The first step was the one most basic yet longest. We had to learn from scratch a coding language and the two protocols on the net with the situation we've been in.

Here is a short brief of what is Verilog, his purpose, use in our case and limits:

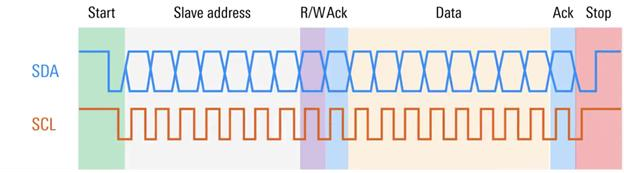
Verilog is a hardware description language (HDL) used to model and simulate electronic systems. The language has a transistor and logic gate resolution making it very useful for low level design. It always uses processes which capture synchronous or asynchronous input signals for doing serial actions. Its limits in our case is to capture patterns without synchronous clocks (wil be explained widely soon).

We will now explain in more details both communication protocols, their advantages, disadvantages and stages.

protocol I2C (Inter-Integrated Circuit) is a serial communication protocol widely used in embedded systems for connecting low-speed peripherals to microcontrollers. It has two lines:

* SDA (Serial Data Line) - this line is bidirectional because here the slave returns an Ack(acknowledge) after every byte gotten. This is the line that transmits the data and is being captured and must be stable at every rising edge of the clock.
* SCL (Serial Clock Line)- this is the clock of the slaves being generated by the master. The clock rate is normally slow in comparison to other protocols. The clock is crucial for serial synchronous communication. In this case when the clock isn't changing the slave modules don't

work. This raises the first obstacle to be surpassed: how will our adapter capture the start and stop signals without any clock.

Now to understand better the protocol process we will pass through every step in the expected communication of I2C using this diagram. This understanding is crucial for understanding our design solution:

1. Stage start communication- start signal. The i2c protocol has a special pattern that the master make for signaling the slave to start communication: the data line (SDA) fall edge without any change of the clock which stays high stable. Next the clock fall edge and the communication starts at the following rising edge of the clock.
2. Stage capture address- first byte, adapter address and read/write bit command.

The first byte to be sent is the address of the i2c slave(in our case the adapter), and the command of read or write from the master.

1. Stage Ack1- returning Ack if the address is accurate to the adapter address. If the address isnt correct returns 0 and expecting a resend (in our case). In case of several slaves connected to one i2c master the slave isnt sending anything (because the information is for a different slave). The Ack is being written on the SDA line which is bedirectional after the first byte.
2. Stage read/write byte- second byte, for now on the cases are different in cases of read and write. The master reads/writes a byte from/to the slave and later the slave/master expects to get an acknowledgement. (the two options are respectively in order).
3. Stage Ack2- second Ack in case of read and in case of write. If the byte is stable returning a positive ack.
4. Stage Burst- after the second ack has being written the slave expects to write/get (r/w) another byte of information until the master signals to the slave to end the communication.
5. Stage ending communication- the ending communication signal is also asynchronous which makes it very difficult to capture. The data line fall edge at the last rising edge of the clock. The clock stays stable at high till the end and then the data rise edge asynchronously.

As written above understanding this communication process is crucial for understanding the design solution.

SPI protocol- this protocol is much more simple. It has four lines: MOSI, MISO, SCL and CS. in case of multiple slaves it has multiple CSs (in our case it has one).

* + MOSI (master out slave in)- this is the master line to write to the slave. In our case the master is our adapter and the slave is some kind of SPI slave.
  + MISO (master in slave out)- this is the master line to read from the slave.
  + SCL- this is the serial clock. Normally SPI clock can be much faster than i2c, in our case because all the information eventually is getting written to the i2c master this is the timing bottleneck so we aren't using any faster clock.
  + CS- this is the line which signals to the slave that the master (in our case our adapter) wants to start a communication.

To sum up this part it was very important for the basics.

## Step Two-

Designing the Solution in Verilog. (01/24–04/24)

Our design objective is to translate the I2C information to SPI and back. To achieve this, we must communicate effectively in both I2C and SPI, primarily focusing on I2C since it is the master’s language. After understanding the stages of I2C communication, we need to design our solution to follow these stages. Here, we will outline the steps of communication and briefly explain how we addressed each part of our solution.

**Top Comment-** Designing in Verilog: In Verilog and most low-level design languages, the strategy for creating the solution is using “always” blocks. These coding blocks are activated whenever one of the defined signals changes, such as the clock, reset, and, in our case, SDA for start and end communication signals. This approach complicates our design due to the mixing of synchronous (read byte) and asynchronous (capture start signal) “always” blocks.

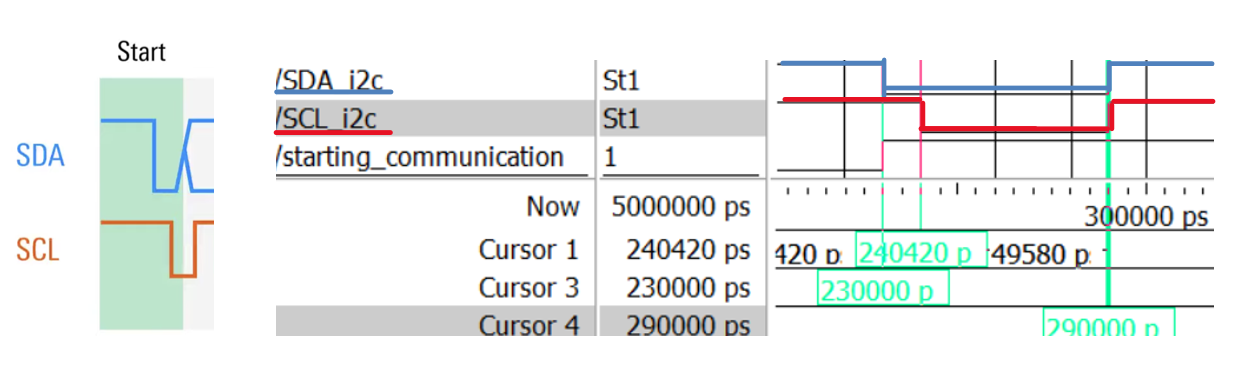
1. Stage Start Communication- This was the first process we addressed since all communication relies on capturing the start communication signal. Here, we encountered our first significant problem: capturing a signal in Verilog.

**Problem N1: Capturing a Signal-** In Verilog design, you are advised to consistently use your “always” block on either the rising edge or the falling edge of the clock, but not both. Here, we must capture the falling edge of the SDA, then the falling edge of the SCL, and immediately on the next rising edge of the clock, the first bit of data starts.

**Our solution:** It took us some time to understand that we had no other choice but to follow the protocol as defined. We cannot capture the start signal synchronously since there is no clock yet in the communication. We cannot progress the rest of the communication on the falling edge of the clock because the data is stable on the rising edge. Therefore, we must capture the signal asynchronously with the following pattern:

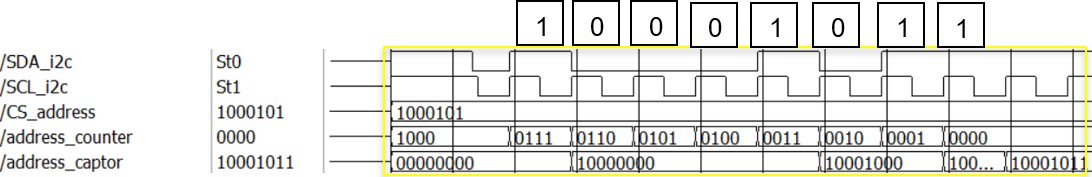
* 1. Capture the falling edge of the SDA.
  2. Capture the falling edge of the SCL.
  3. Raise our start communication signal.
  4. Capture the first data bit on the next high clock.

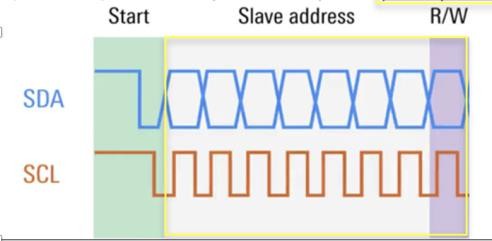
Jump from the Future: This is a snapshot of this pattern in our simulation on step 3, showing this process. This example is relevant to the following design explanations:



We can see the falling edge of the SDA and the falling edge of the SCL. Then, our start communication signal rises, and we capture the first data bit at 290ns, as defined in the I2C protocol.

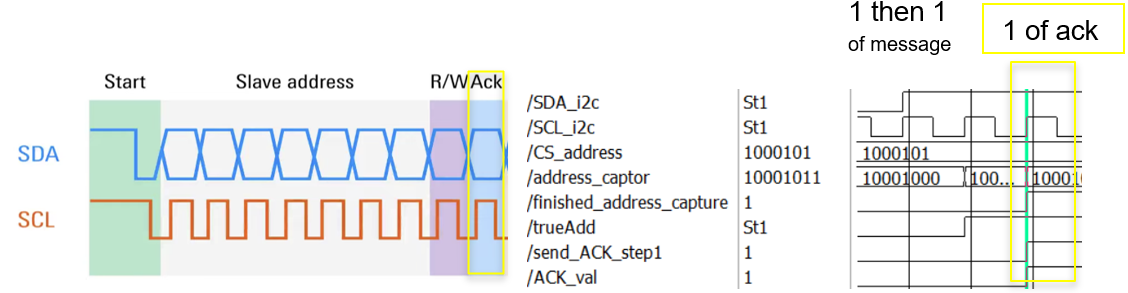
**Important Note:** Due to this scenario, our maximum clock rate is limited to ensure all module logic is stable before the next clock cycle. This is not an issue for us because of the slow rate defined in the SPI protocol.

* 1. Stage Capture Address- In this stage, we start at the first rising edge and capture the first byte of information (7 bits of address and a read/write bit). We used a synchronous always process and a register named `address\_captor` (classic misspelling).



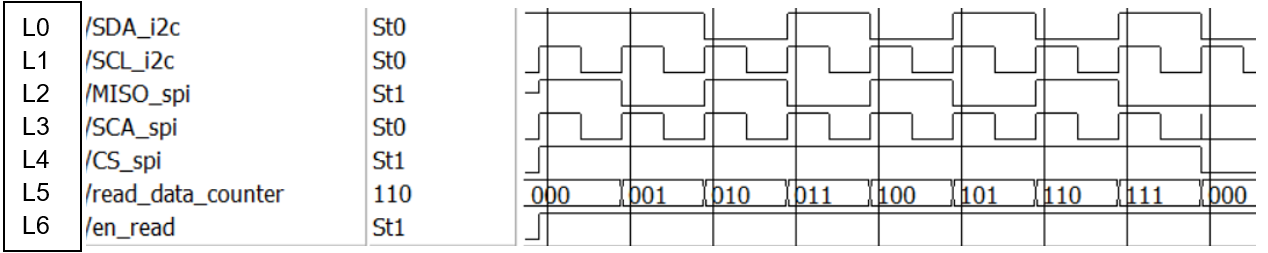
Here we are getting the byte 10001011 from the master which is captured (1 is read).

* 1. Stage Ack1- Here we check if the captured address matches the one defined by the user (provided as an input). If it does, we write a true ACK (acknowledgment) to the SDA. **Problem N2: Ack Timing-** Again, we encounter a timing limitation because we must stabilize our ACK response after validating the address and write it on the next rising edge, as defined by the I2C protocol process. This again limits our clocking rate, but as before, the SPI-defined rate is lower, so this is not an issue for us.



We can see that the true address is `1` in the previous stage, then `send\_ack` is set, and the value is `1`.

1. Stage Read/Write Byte: Now we've arrived at the essence of the communication, which is data transmission. For our module in this section, we need to do the following:
   * Raise the CS of the SPI slave.
   * Transmit the clock of the SPI from the SCL.
   * If we are in write mode, transmit the SDA to MOSI.
   * If we are in read mode, transmit the MISO to SDA.



This is the snapshot of the reading process. First, we see that our inner signal `en\_read` (L6) rises at the rising edge, then the CS (L4) rises, and the slave immediately starts to write (L2). **Problem N3: Input-Output Transmission Delay-** From our knowledge and experience with Verilog, it is not possible to transmit an input directly to an output within an always block, which creates a delay in transmitting the data from the slave (L2) to the master (L0). Since the I2C protocol defines a byte of data followed by an acknowledgment (ACK), we must transmit the data on time!

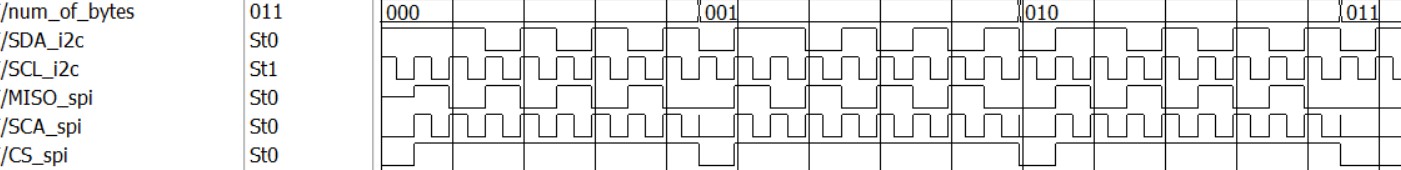
**Our Solution:** We started the communication with the slave one clock cycle earlier, when we returned an ACK (stage 3) to the master. This way, the delay does not affect the timing of the I2C communication and does not create another longest path transmission that would limit our clocking rate (as in Problem N1).

1. Stage Ack2: This stage is very similar to stage 3, with the key difference being that here, we must also:
2. Transmit High-Z to the SDA if we are reading, allowing the master to ACK, then capture it and check if it is `1`.

**Problem N4: Handling Master’s ACK=0 While Reading-** In this case, we must back up every transmitted byte in our module with an appropriate register.

**Our Solution:** After consulting with our adviser (who is also our customer, and the customer is always right!), we chose not to support a backup in the case of ACK=0.

1. Mute the SPI. Since SPI communication does not operate in a byte/ack system, we need to turn CS to `0` at this stage. If we are in burst mode, we will raise it back in the next stage.
2. Stage Burst: This stage continues to transmit data (stages 4 & 5) until the ending communication signal is received.

By addressing these issues and implementing the solutions, we have ensured that our design meets the required timing and protocol specifications for both I2C and SPI communication. Each stage has been carefully crafted to handle specific challenges, making our module robust and reliable.

Here we can see in zoom out the data transmission. Notice that we fall edge the CS at each ack (as explained in stage 5). And that the data transmitted is delayed inside the adapter but timed with the master’s I2C protocol (as explained in stage 4).

**Problem N5: how to generate bidirectional communication from serial-** the SPI protocol has the ability to send (MOSI) and to get (MISO) at the same time (bidirectional). But the I2C is serial.

**Our solution:** since the slave were suppose to translate is a register file that only writes or read but never both and since the time of communication isn’t the importance, and bidirectional communication is important mainly in speeding the transmission, our advisor- costumer suggested us to leave it this way. The bottleneck of speed up is still the I2C protocol clock rate.

1. Stage ending communication- when the master finishes the communication, he sends the ending signal.



**Problem N6: Inability to Capture the Ending Signal-** Despite our efforts, we couldn’t capture the ending signal as a distinct ending signal and differentiate it from bursting data (stage 6).

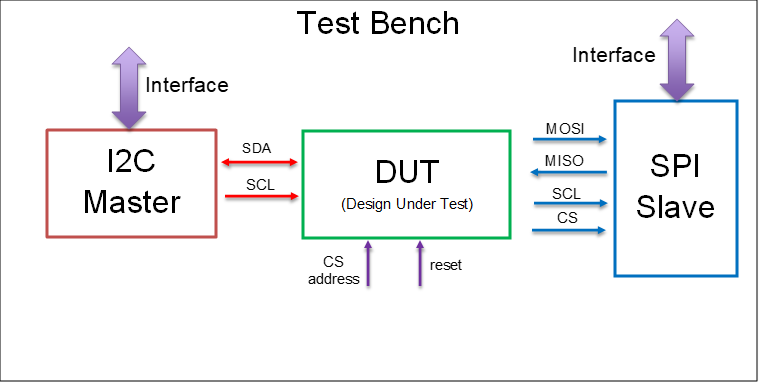
**Our Solution:** As advised by our advisor (who is also the customer), we tailored the design to his specific needs. He requires only 4 bytes of communication each time (to read or write to a register array SPI slave). Therefore, we decided to count the communication bytes and automatically end the communication after 4 bytes (1 for address and 3 for data). Following the customer's directive, this is what we implemented.

We have navigated through the stages of design, addressing each problem and providing solutions. We have also demonstrated our simulation results for each stage and the functionality of our module. Next, we will explain the overall simulation process, where we encountered unexpected issues.

## Step Three-

Verification of the Design Using SystemVerilog Simulation (03/24–06/24)

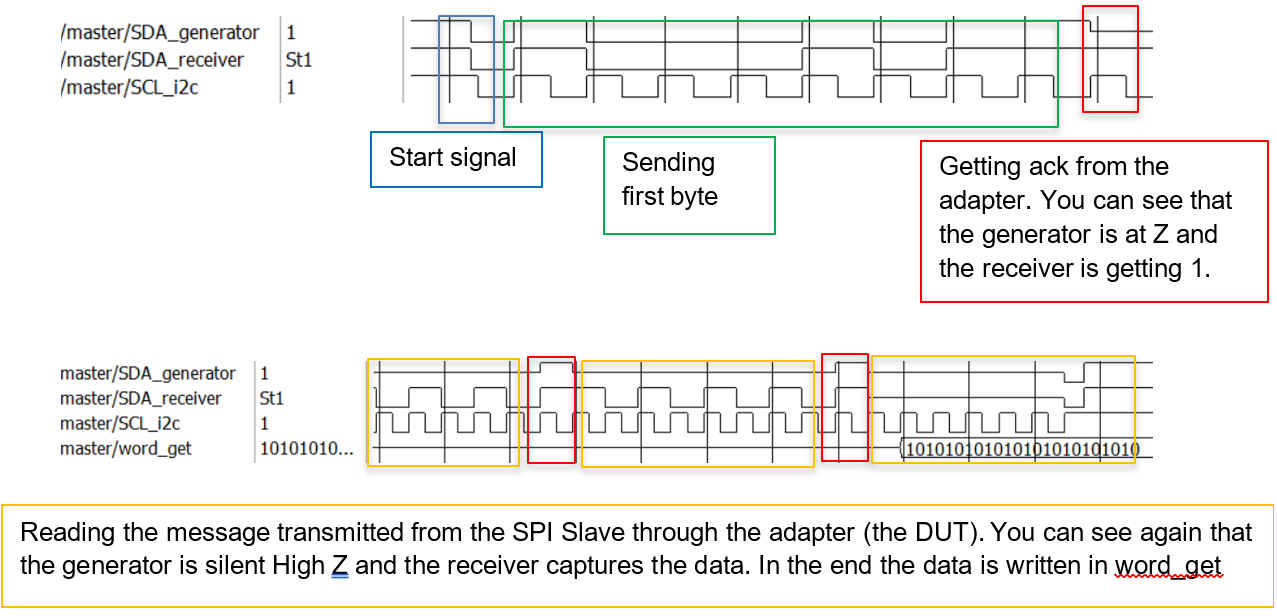
At this stage, we created a SystemVerilog environment to verify the logical operation we designed in the previous step. Here's an overview of the process and the components involved:



We will shortly explain what SystemVerilog is, what the testing modules do, how the testing process has progressed, and what, in our opinion, can be added to further validate the logical simulation check.

SystemVerilog is a hardware description and verification language that extends Verilog by adding advanced verification features, object-oriented programming, and data types. It assists in the verification process by providing robust constructs for creating testbenches, assertions, and randomized testing, enabling more efficient and thorough validation of digital designs.

I2C Master- This module is connected to the test bench (TB) through the interface module. It is the most important module of the verification process since, in our case, the master engages and leads the communication. Therefore, we started by creating it. The master must generate the start communication signal (stage 1) and the end communication signal (stage 7). It must generate a clock for the SCL and be able to write data to the SDA. It must be silent to capture every acknowledgment (ACK) sent by the adapter, and unlike the adapter, the master can resend the byte if ACK = 0 (as discussed in Problem N4 in stage 5). The significant difference here is that the adapter is a physical module, whereas the master is a simulation. Therefore, it is not necessary to add another register to the master since it is not being printed.

Creating the master took us a long time, but it helped us begin the verification of stages 1, 2, and 3 in the adapter. This laid the groundwork for making the overall system work.

Test Bench- This is the module that connects everything. It is not very complex, but it is essential, much like our nervous system. The test bench also generates the reset signal for all modules and manually connects the interface to the DUT since there is no SystemVerilog interface support in Verilog.

Interface- This special module connects the SystemVerilog modules, specifically the I2C Master and SPI Slave, to the test bench.

SPI Slave- This module simulates the communication between the adapter and the SPI Slave. It generates data for read processes and captures data for write processes. It operates only when CS is active (as explained in stage 5) and must be synchronous with the SCL. Although simpler than the I2C Master, it is crucial for verifying stages 4, 5, and 6.

We have created the verification environment; now, let's elaborate on the tests we conducted and passed:

We succeeded in all communication stages, both in reading and writing, including bursting data and communications (several communications in the same test). We haven't randomized the input or checked edge cases.

These synthesis components were sufficient for us to validate the logical simulation of our design. However, there is always room for improvement in verification. Here are some additional components that could enhance the validation process:

As discussed earlier, adding a randomizer is a good idea. Creating an SVA file to automate basic verifications and the entire verification process would also be beneficial.

Eventually, we chose to stop at this step to advance to the backend work of the project. To sum up, this step was the longest and most complicated since we had to create all the SystemVerilog files, validate all the code we wrote (both the DUT and the verification) simultaneously, and debug the DUT. Next, we progressed to creating a synthesis file and debugging the design to ensure it is synthesizable.

## Step Four-

Learning Cadence Applications for Verification, Synthesis, and Backend Progress (05/24– 07/24)

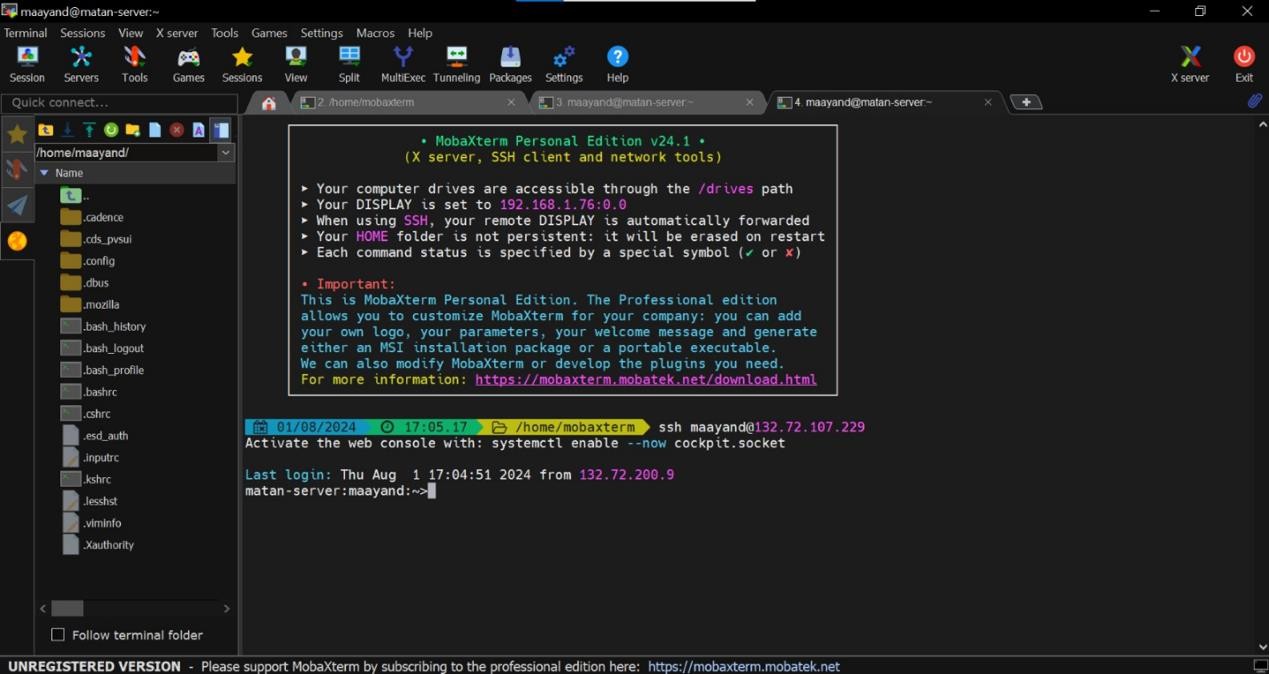
After completing verification, the next step is to physically implement the design. This involves performing synthesis (detailed in the next step) and starting the place and route (PNR) stages. These stages are accomplished using Cadence tools such as Genus and Innovus.

Initially, we focused on learning how to use these tools and the Moba workarea, an environment for engineers to work on their designs. We gathered the necessary information from Cadence workshop directories provided by Cadence representatives.

To begin, we downloaded MobaXterm and entered our user credentials as follows:

1. Download MobaXterm.
2. Enter your user credentials to access the workarea.

This initial setup allowed us to effectively use Cadence tools for the subsequent stages of the project.



To begin, we downloaded MobaXterm and logged in for the first time by entering the password provided by Cadence (or requested from Matan). After logging in, we created our workarea with the command tsmc180G, which sets up the environment for the 180 technology node, suitable for our timing and power constraints. This command creates or opens the directory

/projects/tsmc180G/users/\*user\_name\*/ws, where we work on our design.

To complete this step, we imported all the necessary LEF files for the design and saved them in the LEF directory. These LEF files provide the technology information, libraries of standard cells used in the design, and the metal layers for the signal routing in the design.

## Step Five-

Synthesis (06/24–07/24)

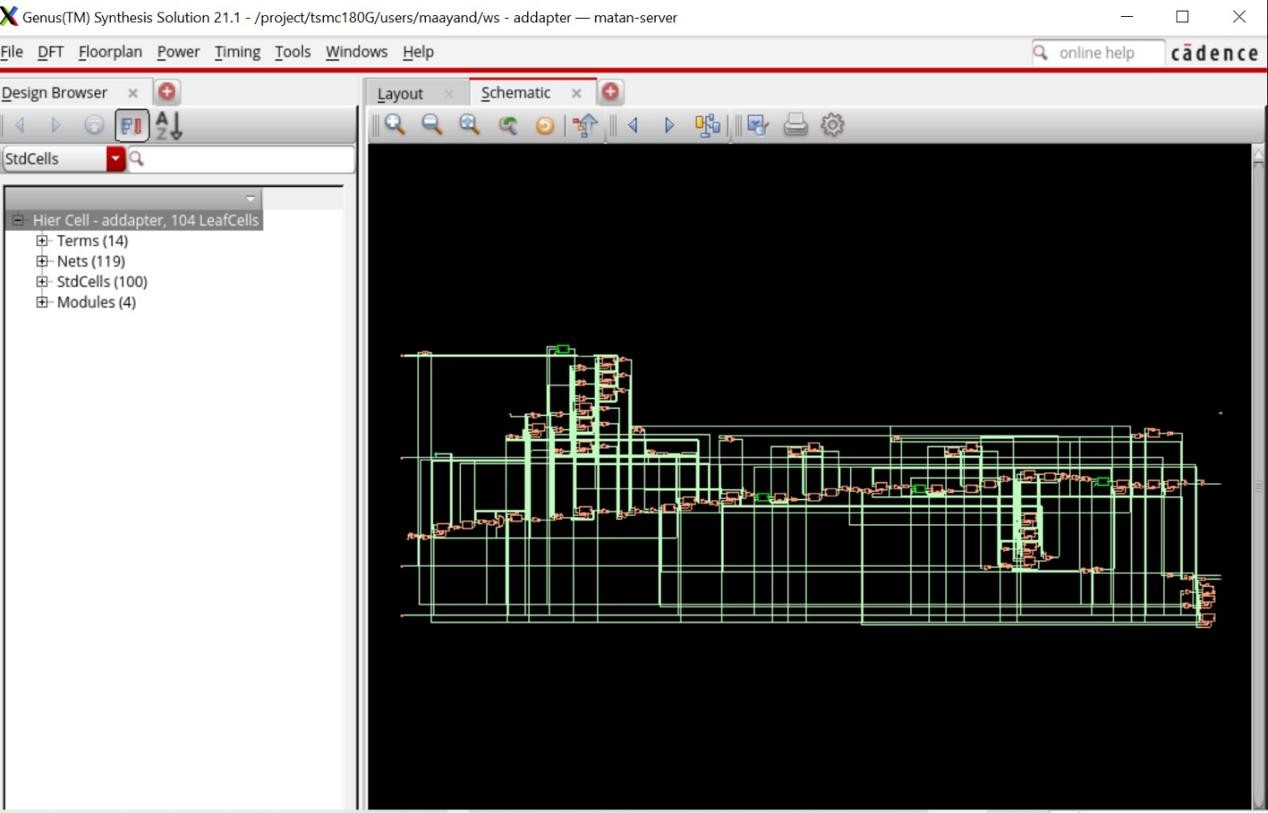
After learning the Cadence applications, we performed the synthesis. The process is straightforward: the Verilog design serves as the input, and the output is a netlist file. This file includes all the physical cells and nets (signals) in the physical design. We ran the synthesis using Cadence Genus.

During this stage, we usually define the clock parameters, including its name and frequency. However, our design does not use a fixed clock source since the I2C protocol relies on the master to generate the clock for the slave via the SCL line, similar to how the adaptor generates the clock for the SPI slave.

The relevant files are located in the directory: `/project/tsmc180G/users/maayand/ws`. The netlist file is in `output/adapter\_m.v`, and the Verilog file is in `syn/adapter.v`.

To perform the synthesis:

1. Open the Genus tool using the command: `genus`.
2. Once open, use: `source syn/syn\_run.tcl` (edit this file if needed).
3. Use the command: `gui\_show` to see your results.



This is the schematic design (to see this, you need to right click on Hier Cell > Schematic View)

## Step Six-

Checking Backend Factors for Creating the Module (07/24–08/24)

At this stage, we focused on checking backend factors to finalize the module creation. We detail the next steps and goals in the "Follow-up for Future Progress" section.

We uploaded the netlist and LEF files to verify functionality using Cadence Innovus. We wrote short scripts to load all the relevant LEF files, which contain information about the technology implemented in the design (in our case, the 180G process), libraries of standard and other cells, and the metals used in the design. These files can be found in the LEF/ directory.

# Follow up for future progress

Project Status (Briefly): So far, we have designed the module in Verilog and tested it using a SystemVerilog verification environment. Then, we created a netlist using Cadence Genus and imported all the relevant LEF files. More details can be found in the previous section about the solution stages.

All those files can be found in Matan’s working space at:

/project/tsmc180G/users/ Maayan\_Noam\_Project

Now, we will explain what we believe needs to be done for this project to reach the final stage: the creation of the physical module.

### Finishing the Logical Synthesis stage:

1. Design For Test (DFT): First, you need to create a DFT chain. This chain is created during the synthesis stage and is designed to allow testing of the final product. You can read more about DFT in the instruction folders provided by Cadence. You can check the “How To” section below.
2. Logical Equivalence (LEQ): You should use an LEQ tool to check the validity of the synthesis netlist by comparing the netlist file with the Verilog design. Currently, this tool is not available in Matan Cadence access.
3. Manual Check: you must upload the Cadence Innovus tool and import the netlist. Then you’ll verify manually that everything looks as expected, And that the created module match the expected logics that you understand from the Verilog file (sanity check for the LEQ).

**Note:** Steps 2 and 3 can take a while because the tools aren't always reliable, and the module could have undetected synthetic errors. You must complete these steps perfectly before proceeding for the project to succeed.

### Start of the Backend design process:

1. Creating a BUM File: In the BUM file, you define the shape and size of your design. We agreed with Matan that each block should be approximately 200um x 200um.

Creating a PTN File: In the PTN file, you define all the terminals and their locations in the design. Each signal entering your design should have an entrance pin, and each signal going from your design to another block should be defined here.

**Note:** Both the BUM file and the PTN file must match the files of other designs. For example, if your output pin is not in the same location as your neighbor's entrance pin (for the same signal), it won’t work.

1. Place and Route (PNR): This will be the majority of your work. Start by uploading the synthesis file and the LEF files (see "How to 3. Manual Check" below, steps 1-4). After that, upload your BUM and PTN files.

The first step is Floorplanning, which involves manually placing your memories in the design. The main steps to follow are: feedthrough, place, clock, route, and finish.

We recommend watching Adi Taman's course on YouTube for a better understanding of the process. You can find the videos at this link: [https://www.youtube.com/watch?v=GIPhBfenqMc&list=PLZU5hLL\_713x0\_AV\_rVbay0pW](https://www.youtube.com/watch?v=GIPhBfenqMc&list=PLZU5hLL_713x0_AV_rVbay0pWmED7992G) [mED7992G](https://www.youtube.com/watch?v=GIPhBfenqMc&list=PLZU5hLL_713x0_AV_rVbay0pWmED7992G)

1. Run Analysis: After completing the PNR stage, you should perform Static Timing Analysis, Power Analysis, Physical Design Verification, and IR Drop Analysis. These tests should be done using a Cadence tool and defined by the user (Matan) needs. All relevant information should be included in Adi Taman's course. This should be the last step before sending your design to the factory for implementation.

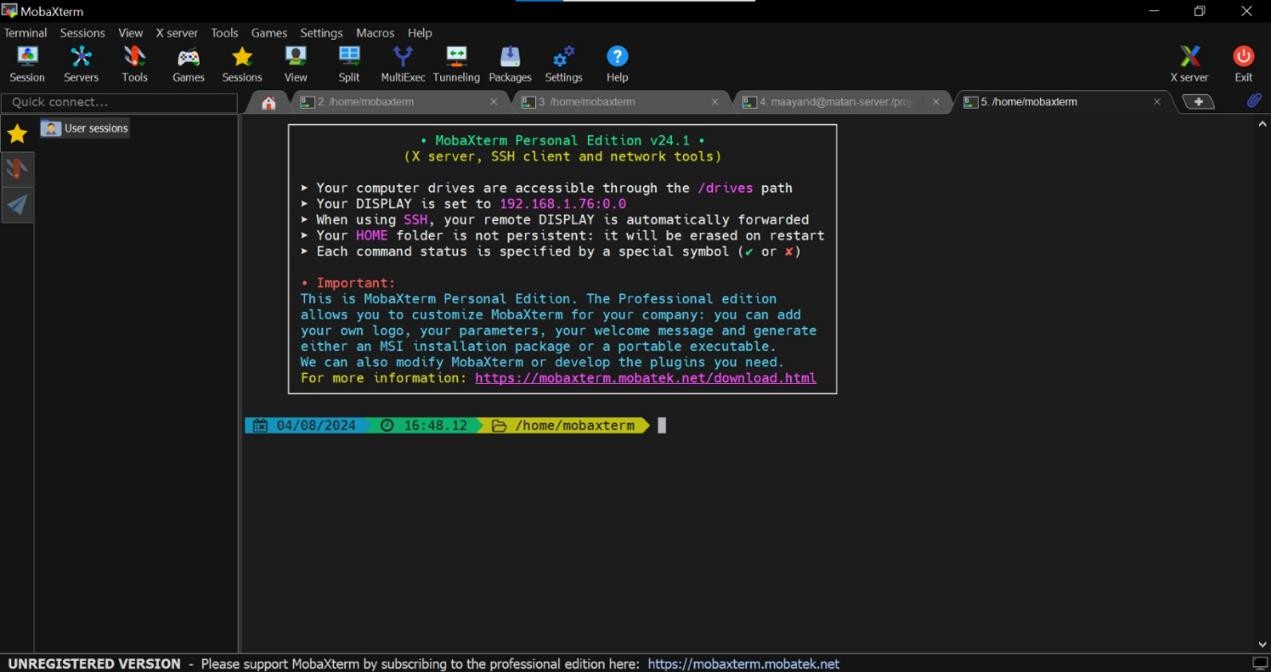
**Note:** Steps 5 and 6 are interconnected. After running the analysis, you will likely need to process the PNR again.

This concludes our explanation of the project's follow-up steps as we understand them. Next, we will provide more information on starting the work, including short "how-to" guides for the initial steps, and we will summarize our contribution to this project.

# How to progress in more details

## How to start working with Cadance tools:

1. Download Mobaxtrem and open the terminal. It should look like this:



1. Enter your user: type: ssh [#user\_name@132.72.107.229](mailto:%23user_name@132.72.107.229) (example [maayand@132.72.107.229](mailto:maayand@132.72.107.229))

then, you should enter your code (ask from Matan for one)

1. Enter your wa: just type tsmc180G (it should open a new workarea if doesn’t exist already and enter this directory).

## How to DFT:

1. you must open the Verilog file, add a scan\_clk input and a test\_mode input, which are important for the synthesis building but is unused logically because of the SCL master’s clock submission. Then you must create a mux such as:

wire inner\_clock= test\_mode scan\_clk : SCL\_i2c;

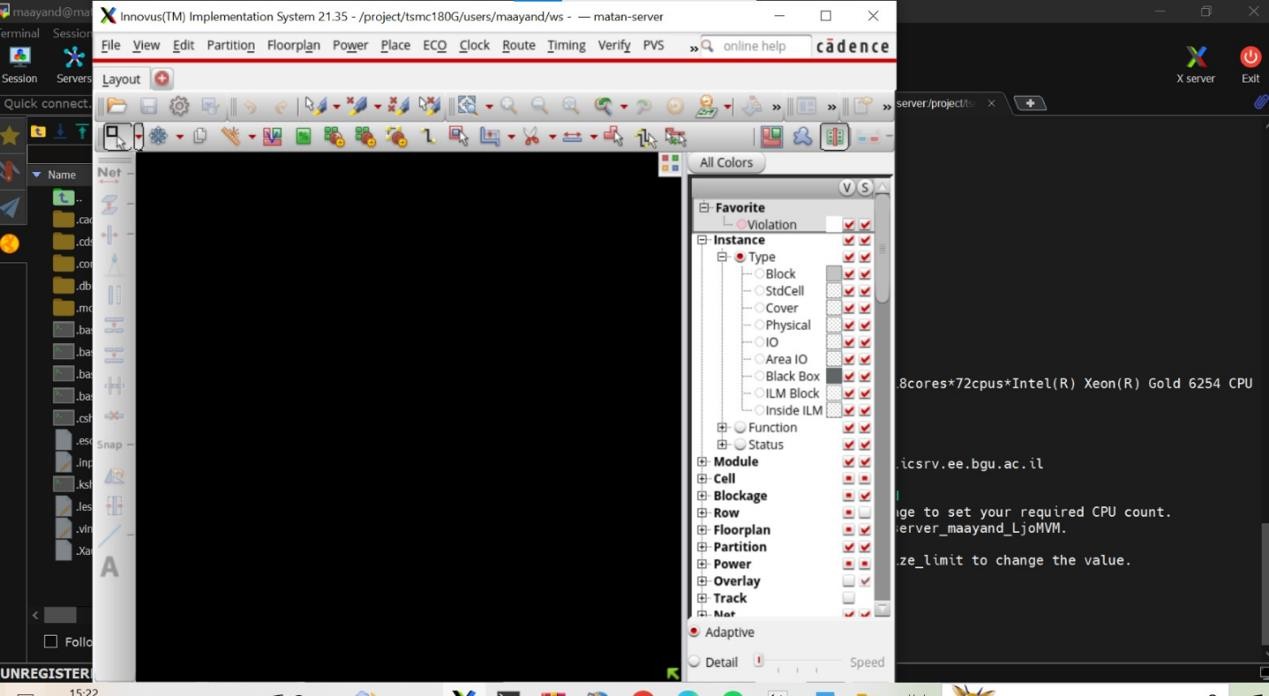
this line will assign the scan\_clk to the module and be used for the DFT construction.

1. You must add inputs scan\_en, scan\_in and output scan\_out, for the tool to function on the design and construct the DFT.
2. Use the given Genus code, we have it as a comment in the tcl file.
3. Be happy that everything got constructed by the DFT

**Note:** you must check your code again under simulative verification test bench and verify that all the logic holds.

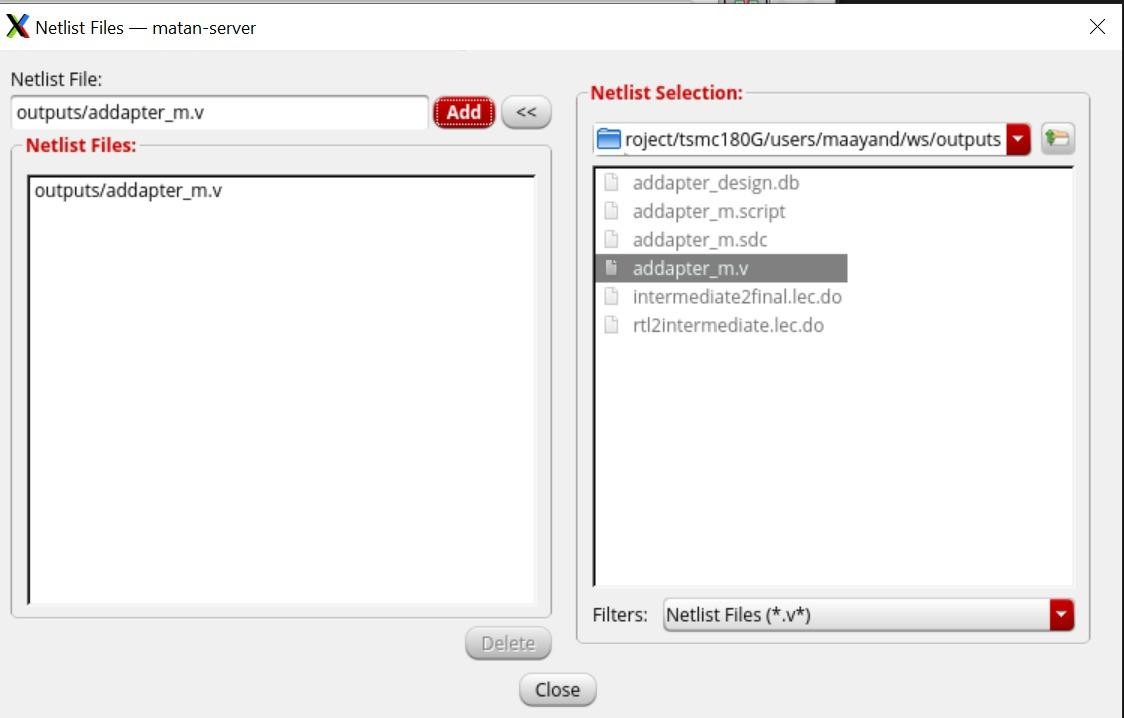
## How to Manual Check:

* 1. enter your workarea and run the command: innovus
  2. run: source scripts/load\_lef\_files (it will load all the lef files you need)
  3. open following window and click: File > Import Design



* 1. Upload the Netlist using the “ ...” in the following screens:



* 1. Go

back to the terminal and run: dbget top.insts.name. This will show you all the instances name you have in your data base (in the design)

* 1. run: dbget top.nets.name . This will give you all the nets names in your data base (on the design).

**Conclusion and Summary of the Project**

This year, we successfully completed the steps necessary to create a physical logical module from scratch. We began with a logical design using Verilog and verified its functionality in a simulative environment using SystemVerilog. We then generated a netlist with Cadence Genus, laying the groundwork for the Place and Route (PNR) Backend stage.

Through this project, we gained valuable experience with tools like Cadence Genus and Innovus and sharpened our skills in writing Verilog code and SystemVerilog verification. We thoroughly documented our progress, providing a comprehensive guide for future projects within Dr. Gal-Katziri’s team. This documentation includes a detailed description of the future process for this project as defined by our advisor.

We exceeded our objectives by establishing a clear and detailed methodology for creating a physical module and outlining the steps needed to progress this specific module to obtain the final product for use in an RFIC.

Our main recommendation for future work is to follow the provided guidelines and thoroughly understand each stage of the process to ensure successful revisions and implementations.

# Sources

The information for this project was primarily derived from guidelines and courses on creating a physical block from scratch rather than academic papers. Below are the main sources we used:

1. **Verilog Design Writing:**
   * *Digital Logic Design and Synthesis Course*
2. **Backend Implementation:**
   * *Adi Taman Course:* [YouTube Playlist](https://www.youtube.com/watch?v=GIPhBfenqMc&list=PLZU5hLL_713x0_AV_rVbay0pWmED7992G)
3. **Additional Resources:**
   * [Cadence Website](https://www.cadence.com/en_US/home.html)
   * *Cadence Workshop provided by Dr. Matan Gal Katziri*