

USB4 Logical Layer   
AI Implementation Analysis

**Submitted by:**

**Karim Samy**

**Ali Ahmed**

**Walid Salah**

**Mohamed Salah**

**Submitted to:**

**Eng. Mohamed Samy**

**Eng. Mohamed Rayan**

**Eng. Randa Aboudief**

**Eng. Ahmed Mohamed**

**Eng. Sarah Mostafa**

**Date: 31/01/2024**

Table of Contents

[REFERENCE MODEL ARCHITECTURE: 1](#_Toc166373593)

[Classes 2](#_Toc166373594)

[1. Configuration Space Class: 2](#_Toc166373595)

[2. Upper Layer Class: 2](#_Toc166373596)

[3. Electrical Layer Class: 3](#_Toc166373597)

[AI Environment: 7](#_Toc166373598)

[Env\_cfg\_mem: 8](#_Toc166373599)

[-Symbol file: 9](#_Toc166373600)

[-Generator: 10](#_Toc166373601)

[TEST: 10](#_Toc166373602)

[Scoreboard: 10](#_Toc166373603)

[Driver: 10](#_Toc166373604)

[Monitor: 10](#_Toc166373605)

[Agent: 10](#_Toc166373606)

[Comparison between AI and Hand written work 11](#_Toc166373607)

[Time Spent on the reference model using Hand Written work 12](#_Toc166373608)

[Time Spent on the reference model using AI 12](#_Toc166373609)

[Comments: 13](#_Toc166373610)

[Pros and cons of AI: 14](#_Toc166373611)

[Comparing different AI Models: 15](#_Toc166373612)

[ChatGPT 3.5 (OpenAi) 15](#_Toc166373613)

[GitHub Copilot 15](#_Toc166373614)

[Notes and examples using GitHub copilot as a coding AI tool: 16](#_Toc166373615)

[AI PROMPT Samples: 19](#_Toc166373616)

[Environment: 19](#_Toc166373617)

[1-CRC task: 19](#_Toc166373618)

[3-PRBS11 task: 21](#_Toc166373619)

[Reference model: 25](#_Toc166373620)

[WRAPPER Class: 25](#_Toc166373621)

[ELECTRICAL LAYER Class: 40](#_Toc166373622)

# REFERENCE MODEL ARCHITECTURE:

A diagram of a wrapper

Description automatically generated



The approach we used to build the reference model using Ai was to divide the model into 3 main classes, each class mimics a part of a DUT, according to the interface it uses, where we have 3 main interfaces, one with the electrical layer, one with the upper layer (transport layer) and one with the configuration spaces. These 3 classes interact with each other and with the verification environment using mailboxes, then all the classes and mailboxes are integrated together in a wrapper that we named “Reference Model Wrapper”.

## Classes

### Configuration Space Class:

This class is responsible for mimicking what we are expecting the DUT to act like through its configuration space interface, where it is responsible to do the following:

* At the beginning of the connection, it sends a read command to the configuration spaces, by raising the c\_read signal, and specifies the address the needs to be read from, which is 8’d18, then waits for the response from the configuration spaces, and if the response was not as expected (in case of any fault), read commands will be sent again until the correct data is received.
* After this step, the class waits for the lane\_disable signal to be raised, and in that case, the connection should be terminated, so the configuration space class sends a transaction to the electrical layer class and upper layer class through the mailboxes “config\_to\_elec\_disable” and “config\_to\_upper\_disable” to inform them that the connection has been disabled.

### Upper Layer Class:

This class is responsible for mimicking what we are expecting the DUT to act like through its Transport layer interface, where it is responsible to do the following:

* It receives the data from the transport layer, forwards it to the Electrical layer class where it should be sent out to its scoreboard.
* It also receives data internally from the electrical layer class and send it out to the upper layer scoreboard.

This was done by generating two threads working concurrently in each direction in order to receive and send data in both ways, Electrical to transport layer or Transport to electrical.

Since the two internal mailboxes, (upper\_to\_elec) & (elec\_to\_upper) are of different transaction type (electrical & upper layer transactions), it is mandatory to map the 8 bits of data coming from one interface to the other before forwarding the transaction to the appropriate scoreboard.

### Electrical Layer Class:

This class is responsible for mimicking what we are expecting the DUT to act like through its electrical layer interface, where it is responsible to do the following:

* Performs phase 2 of lane initialization (Router Detection).
* Performs phase 3 of lane initialization (Determining Link Attributes).
* Performs phase 4 of lane initialization (Link Training for Synchronization).
* Performs phase 5 of lane initialization (Exchanging Transport Layer Packets).

**Phase 2 (Router Detection)**:

This phase is responsible for router detection, where each router raises its sbtx, and waits for sbrx to be raised, to detect that another router is connected to this router, so during this phase, a transaction is sent to the scoreboard of the electrical layer, with the sbtx signal raised to 1.

**Phase 3 (Determining Link Attributes):**

Since this model mimics a “**Host**” Router, this phase starts by sending out the AT command to the electrical layer Scoreboard with the following transaction contents:

* Address = 78 which is the address of Register 12 in the sideband registers.
* Length = 3 to read the first 3 bytes only
* Empty data symbols since it is a Read command.

The Model then waits for an AT Command from the Generator mailbox with the same address of 78 to issue the AT response with the following Transaction Contents:

* Address = 78, Length = 3.
* Data Symbols = 24’h053303 which corresponds to saying that the model supports up to Gen4 Speed as expected by the Logical Layer itself.

For both Transactions, a CRC generator is used to create and transmit the CRC inside the same transaction.

**Phase 4 (Link Training for Synchronization):**

This phase is responsible for training, where each router sends ordered sets to the other router according to the generation speed that they agreed upon during phase 3, and the each ordered set is sent a certain amount of time, so a finite state machine (FSM) is needed inside this class to control the transition and the ordered sets to be sent, and it is according to the following for each generation speed:

* Gen 2: The model should keep sending SLOS1 until it has sent at least 2 SLOS1 and has received 2 SLOS1 in a row, then keeps sending SLOS2 until it has sent at least 2 SLOS2 and has received 2 SLOS2 in a row, then keeps sending TS1 until it has sent at least 32 TS1 and has received 2 TS1 in a row, then keeps sending TS2 until it has sent at least 16 TS2 and has received 2 TS2 in a row.
* Gen 3: The model should keep sending SLOS1 until it has sent at least 2 SLOS1 and has received 2 SLOS1 in a row, then keeps sending SLOS2 until it has sent at least 2 SLOS2 and has received 2 SLOS2 in a row, then keeps sending TS1 until it has sent at least 16 TS1 and has received 2 TS1 in a row, then keeps sending TS2 until it has sent at least 8 TS2 and has received 2 TS2 in a row.
* Gen 4: The model should keep sending TS1 until it has sent at least 16 TS1 and has received at least 1 TS1 or TS2, then keeps sending TS2 until it has sent at least 16 TS2 and has received at least 1 TS2 or TS3, then keeps sending TS3 until it has sent at least 16 TS3 and has received at least 1 TS3 or TS4, then keeps sending TS4 until it has sent at least 16 TS4.
* Additional transitions for Gen 2 and Gen 3:
  + The model should move from the state of TS1 to the state of SLOS2 if it received 2 SLOS1 Symbols in a row.
  + The model should move from state of TS2 to the state of SLOS2 if it received 2 SLOS Symbols (SLOS1 and/or SLOS2) in a row.

**Note:** The shape of SLOS1 and SLOS2 ordered sets is not the same for Gen 2 and Gen 3, and the shape of TS1 and TS2 is not the same for Gen 2, Gen 3 and Gen 4.

This phase has the following finite state machines (FSMs):

* FSM for Gen 2

A diagram of a diagram

Description automatically generated

The number of sent ordered set is the minimum number to be sent, and the number of received ordered sets should be received in a row (consecutive).

* FSM for Gen 3

A diagram of a diagram

Description automatically generated

The number of sent ordered set is the minimum number to be sent, and the number of received ordered sets should be received in a row (consecutive).

* FSM for Gen 4

A diagram of a gene

Description automatically generated

The number of sent/received ordered set is the minimum number to be sent/received.

**Phase 5 (Exchanging Transport Layer Packets):**

Just like the upper layer Class, this phase needs two threads working together but this time to receive data from the internal mailbox (upper\_to\_elec) and forwarding its contents to the electrical layer scoreboard or taking data from the electrical generator and forwarding it to the (elec\_to\_upper) internal mailbox.

# AI Environment:

A diagram of a computer

Description automatically generated***A diagram of a test

Description automatically generated***

-as shown above, the environment has the same exact blocks as the written by hand, but the Ai environment has a class called

” env\_cfg\_mem” that contains all flags and events we need for more abstraction.

A close-up of a computer code

Description automatically generatedNote: we will explain each of these classes separately and how it was generated using AI (GitHub copilot).

## Env\_cfg\_mem:

-This class acts as memory container for all the flags and events which is needed by the electrical layer for communication between components which adds an extra layer of abstraction.

-this class is located inside the environment and communicates with different classes like (Elec monitor, scoreboard, stimulus generator).

-for example, this class contains an Enum(gen\_speed), It is used to determine the clock to be used by the electrical layer monitor to sample the correct data from the Dut on monitor.

-It contains flags like “done “and “data income” which are used for blocking some threads on monitor in some phases from sequence to collect the correct data from Dut.

-the next text is sample for building this class.

**input:** I need to define class called "env\_cfg\_class" work as a memory and define on its properties called "gen\_speed" from type "GEN " and data\_income from type bit.

***A screenshot of a computer program

Description automatically generated***

## -Symbol file:

-This file contains some parameters to use on all components of the environment instead of using random numbers that don’t have clear meaning.

-this parameter is like the previous environment concept but with extra added parameters to better suit the AI environment.

The next blocks are like the previous by hand code, it does the same functionality but with different methods using Ai. We will now compare the performance of the code on each block with Ai.

## A blue rectangle with black text Description automatically generated-Generator:

-It is class to generate stimulus to golden reference model

- It also generates stimulus to the electrical agent which contains the electrical driver to generate the required sequence which is used to drive data to the Dut.

## TEST:

***-***As shown previously it is a class that contains the environment class and virtual sequence class.

## Scoreboard:

-This class works on comparing the transactions which are created by the model and monitor to check for any incorrect behavior.

## Driver:

-this class is connected to the Dut to send required electrical layer data through the interface.

## Monitor:

-monitor is used to collect data from lanes (Dut) and create the transaction and send it to the scoreboard.

## Agent:

-This electrical agent contains the electrical driver and monitor for more abstraction.

-this class is connected to the Dut’s interface and the stimulus generator.

# Comparison between AI and Hand written work

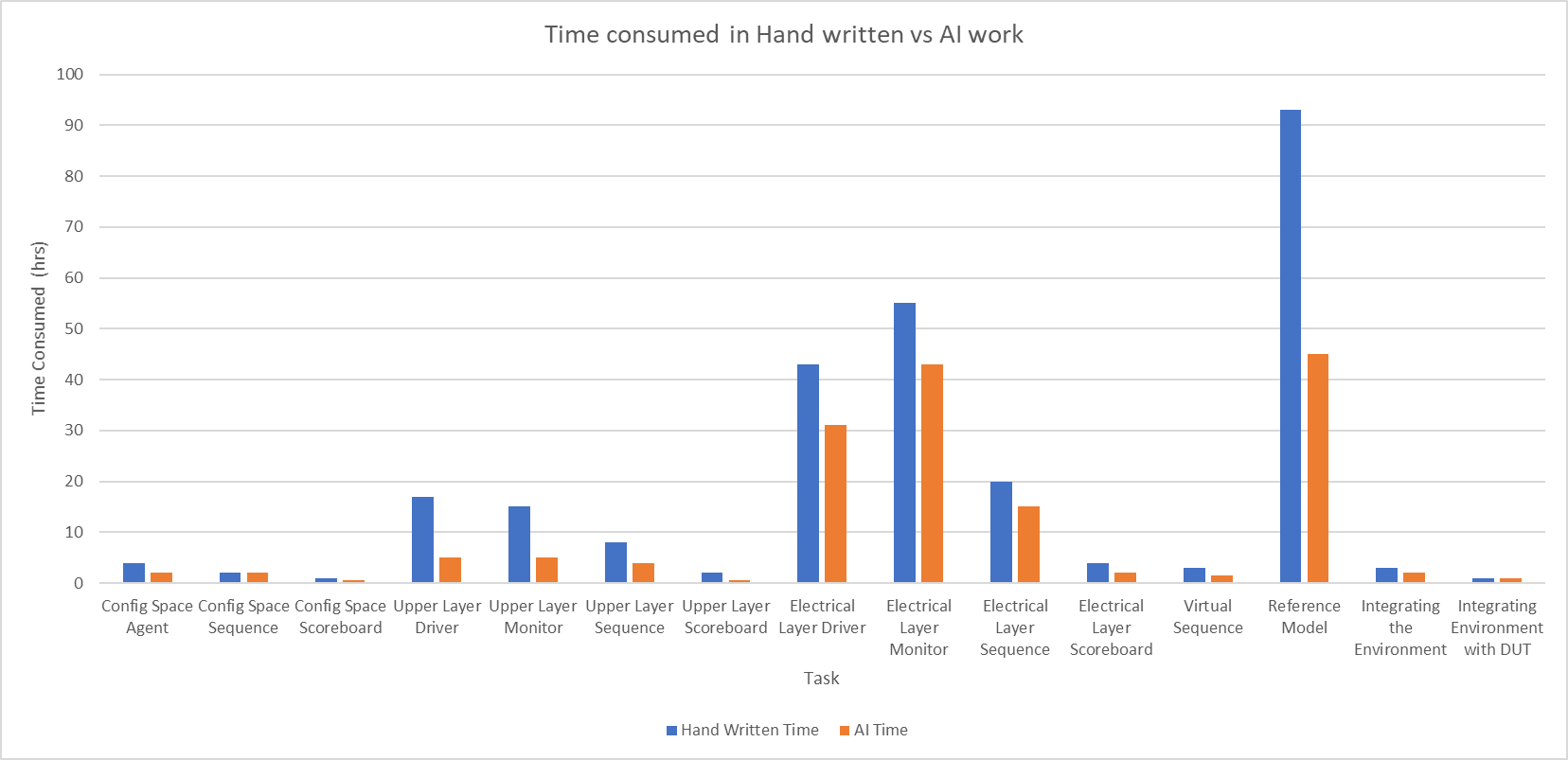
|  |  |  |  |
| --- | --- | --- | --- |
| Task | | Hand Written Time | AI Time |
| Build Verification Environment  (Implementation) | | | |
| Configuration Space’s Agent | | 4 hrs | 2 hrs |
| Configuration Space’s Sequence | | 2 hrs | 2 hrs |
| Configuration Space’s Scoreboard | | 1 hr | 0.5 hr |
| Upper Layer’s Agent | Driver | 17 hrs | 5 hrs |
| Monitor | 15 hrs | 5 hrs |
| Upper Layer’s Sequence | | 8 hrs | 4 hrs |
| Upper Layer’s Scoreboard | | 2 hrs | 0.5 hrs |
| Electrical Layer’s Agent | Driver | 43 hrs | 31hrs |
| Monitor | 55 hrs | 43hrs |
| Electrical Layer’s Sequence | | 20 hrs | 15hrs |
| Electrical Layer’s Scoreboard | | 4 hrs | 2hrs |
| All Transactions and Interfaces | | 2 hrs  (Same files were used in both) | |
| Virtual Sequence | | 3 hrs | 1.5hrs |
| Reference Model | | 93 hrs | 45 hrs |
| Build Verification Environment  (Integration) | | 3 hrs | 2 hrs |
| Build Verification Environment  (Integration with DUT) | | 1 hr | 1 hr |
| Normal Scenarios | | | |
| Basic Operation | | | |
| Gen 4 | Development | 3 hrs | 5hrs |
| Running |  |  |
| Debugging | 81 hrs | 40hrs |
| Gen 3 | Development | 5 hrs | 6hrs |
| Running |  |  |
| Debugging | 30 hrs | 10hrs |
| Gen 2 | Development | 1 hr | 1hrs |
| Running |  |  |
| Debugging | 3 hrs | 7hrs |

# Time Spent on the reference model using Hand Written work

|  |  |
| --- | --- |
| Task | Time |
| Modelling diagram | 15 hrs |
| CRC, Serializer, SB registers | 20 hrs |
| Build phases | 38 hrs |
| Debugging and connecting with environment | 20 hrs |
| Total | 93 hrs |

# Time Spent on the reference model using AI

|  |  |  |
| --- | --- | --- |
| Task | | Time |
| Reference Model Wrapper | | 2 hrs |
| Configuration Space Reference | | 2.5 hrs |
| Electrical Layer Reference | Phase 2 | 0.5 hrs |
| Phase 3 | 12 hrs |
| Phase 4 | 20 hrs |
| Phase 5 | 4 hrs |
| Upper Layer Reference | | 4 hrs |
| Total | | 45 hrs |



## Comments:

By analyzing the previous chart, it can be noticed that the time spent developing the AI implementation is considerably less than the time spent on “by-hand” implementation.

However, this can be a bit misleading to assume AI is much more efficient than writing the code by hand. That’s mainly due to the following:

* We gained a lot of experience working on the project “by hand” which helped us work faster on the AI implementation.
* A lot of the faced problems during the “by hand” implementation were easily avoided which saved a lot of time.

# Pros and cons of AI:

|  |  |
| --- | --- |
| PROS | CONS |
| It is very good in avoiding the basic errors in coding (typos, semicolons …). | Working with more than one file at the same time causes a lot of issues.   * Mixing signals together * ignoring certain parts of the code * overwriting previously generated code |
| It can understand the context of the code currently being written and help complete it. | Level of efficiency decreases drastically writing codes which describe functionality of less known constructs and complicated architectures |
| We can give the AI any error message from the simulator and let it explain and can give option to solve the problem while preserving its functionality. |  |
| AI is aware of the basic building blocks of an environment and all its components; thus, it is very efficient in writing environment templates much faster than writing them by hand.   * Class construction * mailboxes and * events…. |  |
| AI can understand the code and generate a testbench with the input specs to quickly test the functionality of any generated code. |  |
| AI is aware with the famous blocks and implement them with no mistakes (i.e. CRC). |  |
| Ai has the option to regenerate the code in deferent ways if the generated code doesn’t achieve the specification requirement. |  |

# Comparing different AI Models:

## ChatGPT 3.5 (OpenAi)

#### Pros:

* Very good at holding the context throughout the whole conversation.
* Understands the intention of the code to be implemented.

#### Cons:

* Poor implementation of the logic intended.

## GitHub Copilot

#### Pros:

* Very good in logic implementation in most of the cases
* In many cases, it can detect its own errors without needing me to point them out directly.

#### Cons:

* Sometimes while fixing part of the code, it ruins other parts.
* Slow when the code is long (while displaying the code line by line)
* Forgets the context, and sometimes it should be asked to show the whole code after every iteration, or else it will forget what it did earlier.
* Sometimes it forgets the code it has already written, and we need to write it again to continue working.

# Notes and examples using GitHub copilot as a coding AI tool:

* Using the AI to generate the whole code from the first time is impossible in most of the cases, especially complicated codes.
* It is better to use the AI to implement a specific part of the code, and in that case, it is very useful can sometimes use efficient and creative ideas that we may have not used before.
* You can you use (#) then (#file) from the nested list to choose more than one reference file and this can be useful to make the AI know the environment we are working with.

A screenshot of a computer

Description automatically generatedThis is an example of writing the code for the monitor:

**The response:**

A screenshot of a computer program

Description automatically generated

**The generated code:**

A screenshot of a computer program

Description automatically generated

• We can benefit from GitHub copilot by not only describing what we would like to build, and asking it to implement the code, but it can also help us by auto typing and suggesting the remaining of the code

We can give the AI an example of what we need it to do and this will give us more efficient results like the following example:

A screenshot of a computer

Description automatically generated

# AI PROMPT Samples:

## Environment:

the next topic is samples on the AI work for generate some tasks and small testbenches to do certain functions that save a lot of time by doing the same tasks by hand:

### 1-CRC task:

-It takes few minutes to generate the circuit compared with by hand, it takes 2 minutes exactly.

**Input:** I need crc task with inputs " input bit [7:0] data\_symb[$], output bit [15:0] CRC\_out" and

with specs Width: 16

• Poly: 8005h

• Init: FFFFh

• RefIn: True

• RefOut: True

• XorOut: 0000h

but don't flip each byte before entering it to the beginning of the circuits.

A computer screen shot of a program

Description automatically generated 2-Reverse task

**Input:** need a queue method to reverse each 8bits on the queue separately and store the value in the same queue and test this function (15 minute).

A screen shot of a computer code

Description automatically generated

## 3-PRBS11 task:

-it is one of the most Important tasks as it takes a lot of time to implement the required functionality but AI generates it much faster and saves a lot of time.

***A computer screen shot of a program

Description automatically generated***

#### Environment

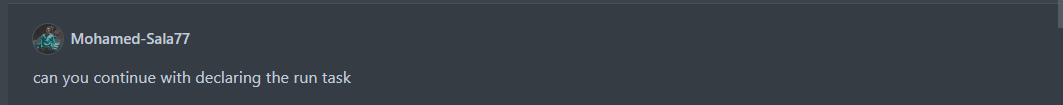
A screenshot of a computer

Description automatically generatedAdd lines in the code to declare the environment components in the Environment code file:

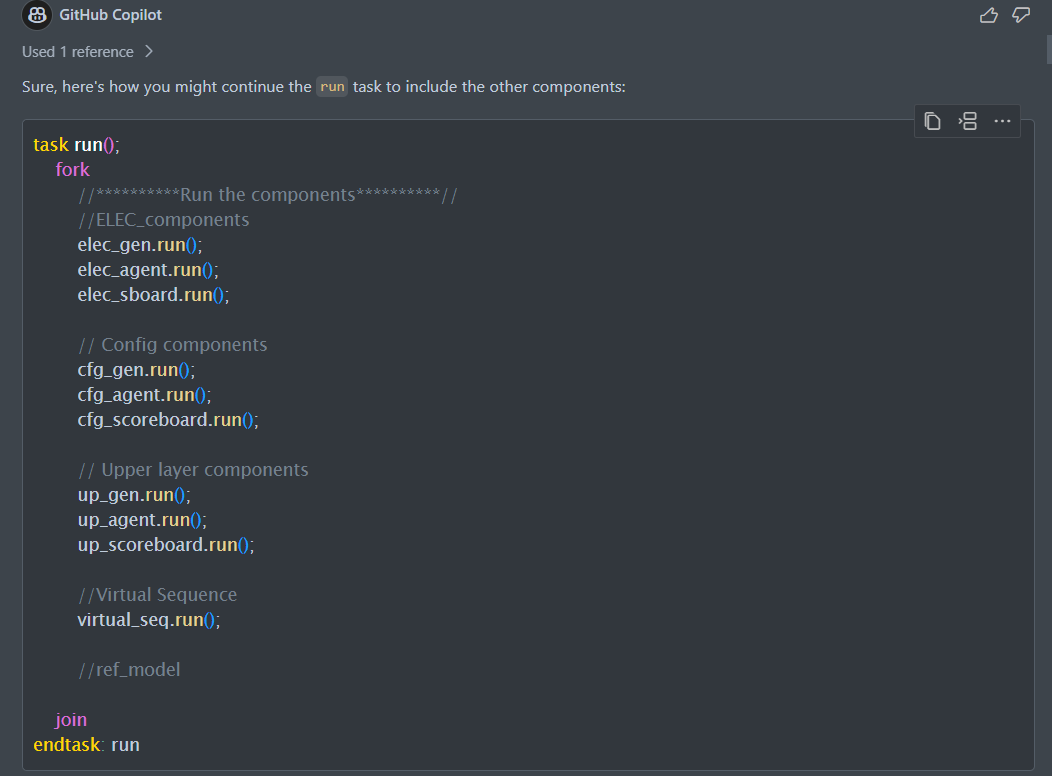
**The result**

**A screenshot of a computer program

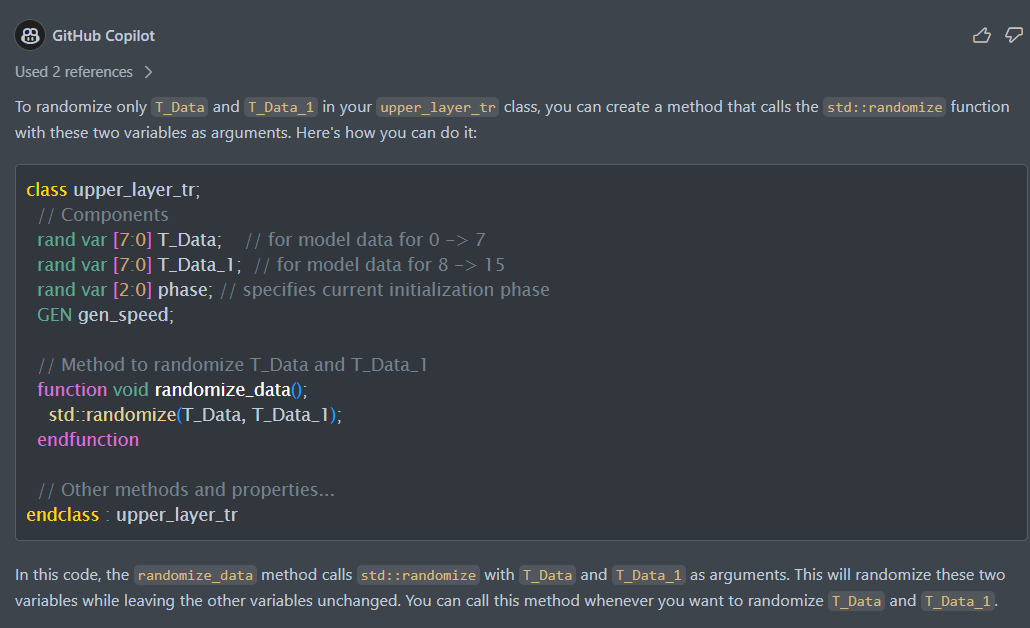
Description automatically generated**

Then AI was asked to call the buildup function of the components (which is named here “run”): 

**The result :**



**In upper files:** A screen shot of a computer

Description automatically generated  
Upper generator: randomization 

## Reference model:

### WRAPPER Class:

#### Prompt #1(Base Prompt):

\*\*consider files #file:elec\_layer\_tr.svh #file:upper\_layer\_tr.svh #file:config\_space\_pkg.sv #file:upper\_reference.sv for context\*\*

\*\*we will be creating a wrapper system verilog class that connects the main classes with each other using mailboxes and running there respective tasks in a single thread.

1- create a class named "reference\_model\_AI"

2 - inside the class do as follows:

1 - create an instance of 3 classes named "upper\_layer\_tr" , "elec\_layer\_tr" , "config\_transaction"

2 - define (upper layer) mailboxes as follows:

- name: upper\_S , only accepts data type of "upper\_layer\_tr"

- name: upper\_G, only accepts data type of "upper\_layer\_tr"

3 - define (electrical layer) mailboxes as follows:

- name: elec\_S, only accepts data type of "elec\_layer\_tr"

- name: elec\_G, only accepts data type of "elec\_layer\_tr"

4 - define (configuration space layer) mailboxes as follows:

- name: config\_S, only accepts data type of "config\_transaction"

- name: config\_G, only accepts data type of "config\_transaction"

5 - define (internal communication) mailboxes as follows:

- name: upper\_to\_elec , only accepts data type of "upper\_layer\_tr"

- name: elec\_to\_upper , only accepts data type of "elec\_layer\_tr"

3 - create the "new" function which takes input of the "upper layer" mailboxes, "electrical layer" mailboxes , "configuration space" mailboxes

4 - inside the "new" function, connect the "upper layer" mailboxes, "electrical layer" mailboxes , "configuration space" mailboxes to the function inputs as follows this.upper\_S = upper\_S (repeat for all)

5 - inside the "new" function, create a handle for the "internal communication" mailboxes as follows : upper\_to\_elec = new()

6 - inside the "new" function, create an instance of the class named "upper\_ref\_AI" and call the new function with given mailbox-inputs in the following order:

1 - upper\_S , 2- upper\_G, 3- upper\_to\_elec , 4- elec\_to\_upper

7- create a task named "run"

8 - inside the "run" task create a thread using fork join and call the upper\_ref\_AI.run

\*\*

#### Generated Code:

```systemverilog

class reference\_model\_AI;

  // Instances of classes

  upper\_layer\_tr upper\_layer\_tr\_inst;

  elec\_layer\_tr elec\_layer\_tr\_inst;

  config\_transaction config\_transaction\_inst;

  // Mailboxes

  mailbox upper\_S;

  mailbox upper\_G;

  mailbox elec\_S;

  mailbox elec\_G;

  mailbox config\_S;

  mailbox config\_G;

  mailbox upper\_to\_elec;

  mailbox elec\_to\_upper;

  // Instance of upper\_ref\_AI

  upper\_ref\_AI upper\_ref\_AI\_inst;

  // Constructor

  function new(mailbox upper\_S, mailbox upper\_G, mailbox elec\_S, mailbox elec\_G, mailbox config\_S, mailbox config\_G);

    this.upper\_S = upper\_S;

    this.upper\_G = upper\_G;

    this.elec\_S = elec\_S;

    this.elec\_G = elec\_G;

    this.config\_S = config\_S;

    this.config\_G = config\_G;

    // Create internal communication mailboxes

    this.upper\_to\_elec = new();

    this.elec\_to\_upper = new();

    // Create instance of upper\_ref\_AI

    this.upper\_ref\_AI\_inst = new(this.upper\_S, this.upper\_G, this.upper\_to\_elec, this.elec\_to\_upper);

  endfunction

  // Task run

  task run;

    fork

      upper\_ref\_AI\_inst.run();

    join

  endtask

endclass

```

This code creates a class `reference\_model\_AI` with instances of `upper\_layer\_tr`, `elec\_layer\_tr`, and `config\_transaction`. It also creates several mailboxes for different layers and internal communication. The `new` function initializes these instances and mailboxes. The `run` task starts a thread that runs the `run` method of the `upper\_ref\_AI` instance.

**-----------------------------------**

#### Prompt #2:

\*\*mailboxes should be assigned a data type :

- upper\_S / upper\_G should be of data type "upper\_layer\_tr"

- elec\_S / elec\_G should be of data type "elec\_layer\_tr"

- config\_S / config\_G should be of data type "config\_transaction"

- upper\_to\_elec should be of data type "upper\_layer\_tr"

- elec\_to\_upper should be of data type "elec\_layer\_tr"\*\*

#### Generated Code:

```systemverilog

class reference\_model\_AI;

  // Instances of classes

  upper\_layer\_tr upper\_layer\_tr\_inst;

  elec\_layer\_tr elec\_layer\_tr\_inst;

  config\_transaction config\_transaction\_inst;

  // Mailboxes

  mailbox #(upper\_layer\_tr) upper\_S;

  mailbox #(upper\_layer\_tr) upper\_G;

  mailbox #(elec\_layer\_tr) elec\_S;

  mailbox #(elec\_layer\_tr) elec\_G;

  mailbox #(config\_transaction) config\_S;

  mailbox #(config\_transaction) config\_G;

  mailbox #(upper\_layer\_tr) upper\_to\_elec;

  mailbox #(elec\_layer\_tr) elec\_to\_upper;

  // Instance of upper\_ref\_AI

  upper\_ref\_AI upper\_ref\_AI\_inst;

  // Constructor

  function new(mailbox #(upper\_layer\_tr) upper\_S, mailbox #(upper\_layer\_tr) upper\_G, mailbox #(elec\_layer\_tr) elec\_S, mailbox #(elec\_layer\_tr) elec\_G, mailbox #(config\_transaction) config\_S, mailbox #(config\_transaction) config\_G);

    this.upper\_S = upper\_S;

    this.upper\_G = upper\_G;

    this.elec\_S = elec\_S;

    this.elec\_G = elec\_G;

    this.config\_S = config\_S;

    this.config\_G = config\_G;

    // Create internal communication mailboxes

    this.upper\_to\_elec = new();

    this.elec\_to\_upper = new();

    // Create instance of upper\_ref\_AI

    this.upper\_ref\_AI\_inst = new(this.upper\_S, this.upper\_G, this.upper\_to\_elec, this.elec\_to\_upper);

  endfunction

  // Task run

  task run;

    fork

      upper\_ref\_AI\_inst.run();

    join

  endtask

endclass

```

In this code, the mailbox data types are specified using the `#(type)` syntax. The `new` function also takes these typed mailboxes as parameters.

**-----------------------------------**

#### Prompt #3:

the internal communication mailboxes and upper\_ref\_AI instances inside the "new" function should not have the "this." constructor as there is no confliction between naming of class content and function input\*\*

#### Generated Code:

```systemverilog

class reference\_model\_AI;

  // Instances of classes

  upper\_layer\_tr upper\_layer\_tr\_inst;

  elec\_layer\_tr elec\_layer\_tr\_inst;

  config\_transaction config\_transaction\_inst;

  // Mailboxes

  mailbox #(upper\_layer\_tr) upper\_S;

  mailbox #(upper\_layer\_tr) upper\_G;

  mailbox #(elec\_layer\_tr) elec\_S;

  mailbox #(elec\_layer\_tr) elec\_G;

  mailbox #(config\_transaction) config\_S;

  mailbox #(config\_transaction) config\_G;

  mailbox #(upper\_layer\_tr) upper\_to\_elec;

  mailbox #(elec\_layer\_tr) elec\_to\_upper;

  // Instance of upper\_ref\_AI

  upper\_ref\_AI upper\_ref\_AI\_inst;

  // Constructor

  function new(mailbox #(upper\_layer\_tr) upper\_S, mailbox #(upper\_layer\_tr) upper\_G, mailbox #(elec\_layer\_tr) elec\_S, mailbox #(elec\_layer\_tr) elec\_G, mailbox #(config\_transaction) config\_S, mailbox #(config\_transaction) config\_G);

    this.upper\_S = upper\_S;

    this.upper\_G = upper\_G;

    this.elec\_S = elec\_S;

    this.elec\_G = elec\_G;

    this.config\_S = config\_S;

    this.config\_G = config\_G;

    // Create internal communication mailboxes

    upper\_to\_elec = new();

    elec\_to\_upper = new();

    // Create instance of upper\_ref\_AI

    upper\_ref\_AI\_inst = new(upper\_S, upper\_G, upper\_to\_elec, elec\_to\_upper);

  endfunction

  // Task run

  task run;

    fork

      upper\_ref\_AI\_inst.run();

    join

  endtask

endclass

```

In this version of the code, `this.` is removed from the `upper\_to\_elec`, `elec\_to\_upper`, and `upper\_ref\_AI\_inst` inside the `new` function.

**-----------------------------------**

#### Prompt #4: (poor comments compared to Microsoft copilot)

\*\*add comments to the code to be more readable \*\*

#### Generated Code:

Sure, here's the code with added comments for better readability:

```systemverilog

// Define the reference\_model\_AI class

class reference\_model\_AI;

  // Declare instances of classes

  upper\_layer\_tr upper\_layer\_tr\_inst; // Instance of upper\_layer\_tr class

  elec\_layer\_tr elec\_layer\_tr\_inst; // Instance of elec\_layer\_tr class

  config\_transaction config\_transaction\_inst; // Instance of config\_transaction class

  // Declare mailboxes with specific data types

  mailbox #(upper\_layer\_tr) upper\_S; // Mailbox for upper layer (Sender)

  mailbox #(upper\_layer\_tr) upper\_G; // Mailbox for upper layer (Getter)

  mailbox #(elec\_layer\_tr) elec\_S; // Mailbox for electrical layer (Sender)

  mailbox #(elec\_layer\_tr) elec\_G; // Mailbox for electrical layer (Getter)

  mailbox #(config\_transaction) config\_S; // Mailbox for configuration space layer (Sender)

  mailbox #(config\_transaction) config\_G; // Mailbox for configuration space layer (Getter)

  mailbox #(upper\_layer\_tr) upper\_to\_elec; // Mailbox for internal communication from upper layer to electrical layer

  mailbox #(elec\_layer\_tr) elec\_to\_upper; // Mailbox for internal communication from electrical layer to upper layer

  // Declare instance of upper\_ref\_AI class

  upper\_ref\_AI upper\_ref\_AI\_inst;

  // Constructor

  function new(mailbox #(upper\_layer\_tr) upper\_S, mailbox #(upper\_layer\_tr) upper\_G, mailbox #(elec\_layer\_tr) elec\_S, mailbox #(elec\_layer\_tr) elec\_G, mailbox #(config\_transaction) config\_S, mailbox #(config\_transaction) config\_G);

    // Initialize mailboxes

    this.upper\_S = upper\_S;

    this.upper\_G = upper\_G;

    this.elec\_S = elec\_S;

    this.elec\_G = elec\_G;

    this.config\_S = config\_S;

    this.config\_G = config\_G;

    // Create internal communication mailboxes

    upper\_to\_elec = new();

    elec\_to\_upper = new();

    // Create instance of upper\_ref\_AI and pass the mailboxes as arguments

    upper\_ref\_AI\_inst = new(upper\_S, upper\_G, upper\_to\_elec, elec\_to\_upper);

  endfunction

  // Task to run the upper\_ref\_AI instance

  task run;

    fork

      // Call the run method of the upper\_ref\_AI instance

      upper\_ref\_AI\_inst.run();

    join

  endtask

endclass

```

This version of the code includes comments that explain what each part of the code does, making it easier to understand for anyone reading it.

**-----------------------------------**

#### Prompt #5:

- add the instance of the elec\_ref\_AI class

- match the mailbox inputs in the "new" function according to the appropriate order

- call the run task inside the fork join thread\*\*

#### Generated Code:

Here is the updated SystemVerilog code with the instance of `elec\_ref\_AI` class added, the mailbox inputs in the `new` function matched according to the appropriate order, and the `run` task called inside the fork join thread:

```systemverilog

class reference\_model\_AI;

  // Instances of classes

  upper\_layer\_tr upper\_layer\_tr\_inst;

  elec\_layer\_tr elec\_layer\_tr\_inst;

  config\_transaction config\_transaction\_inst;

  // Mailboxes

  mailbox #(upper\_layer\_tr) upper\_S;

  mailbox #(upper\_layer\_tr) upper\_G;

  mailbox #(elec\_layer\_tr) elec\_S;

  mailbox #(elec\_layer\_tr) elec\_G;

  mailbox #(config\_transaction) config\_S;

  mailbox #(config\_transaction) config\_G;

  mailbox #(upper\_layer\_tr) upper\_to\_elec;

  mailbox #(elec\_layer\_tr) elec\_to\_upper;

  // Instances of upper\_ref\_AI and elec\_ref\_AI

  upper\_ref\_AI upper\_ref\_AI\_inst;

  elec\_ref\_AI elec\_ref\_AI\_inst;

  // Constructor

  function new(mailbox #(upper\_layer\_tr) upper\_S, mailbox #(upper\_layer\_tr) upper\_G, mailbox #(elec\_layer\_tr) elec\_S, mailbox #(elec\_layer\_tr) elec\_G, mailbox #(config\_transaction) config\_S, mailbox #(config\_transaction) config\_G);

    this.upper\_S = upper\_S;

    this.upper\_G = upper\_G;

    this.elec\_S = elec\_S;

    this.elec\_G = elec\_G;

    this.config\_S = config\_S;

    this.config\_G = config\_G;

    // Create internal communication mailboxes

    upper\_to\_elec = new();

    elec\_to\_upper = new();

    // Create instances of upper\_ref\_AI and elec\_ref\_AI

    upper\_ref\_AI\_inst = new(upper\_S, upper\_G, upper\_to\_elec, elec\_to\_upper);

    elec\_ref\_AI\_inst = new(elec\_S, elec\_G, elec\_to\_upper, upper\_to\_elec);

  endfunction

  // Task run

  task run;

    fork

      upper\_ref\_AI\_inst.run();

      elec\_ref\_AI\_inst.run();

    join

  endtask

endclass

```

In this code, an instance of `elec\_ref\_AI` is created and assigned to `elec\_ref\_AI\_inst`. The `new` function of `elec\_ref\_AI` is then called with the appropriate mailbox inputs. The `run` task of `elec\_ref\_AI\_inst` is also called inside the fork join thread.

**-----------------------------------**

#### UPPER LAYER Class:

#### Prompt #1(Base Prompt):

\*\*consider these files throughout this session #file:upper\_layer\_tr.svh #file:elec\_layer\_tr.svh #file:config\_space\_pkg.sv as they are systemverilog classes containg the components to be used by other classes \*\*

\*\*we want to create a system verilog class that takes data from a mailbox and sends it out via another mailbox.

- create a class named upper\_ref\_AI.

- inside the class, create an instance of the two mentioned classes, (elec\_layer\_tr) & (upper\_layer\_tr)

- inside the class, instantiate 4 mailboxes as follows:

1 - name: upper\_S, only accepts data type of "elec\_layer\_tr" ,

2 - name: upper\_G, only accepts data type of "elec\_layer\_tr" ,

3 - name: upper\_to\_elec, only accepts data type of "upper\_layer\_tr",

4 - name: elec\_to\_upper, only accepts data type of "elec\_layer\_tr"

- create the "new" function that takes the 4 mailboxes as inputs

- inside the function, connect the mailboxes to the input mailboxes as follows: this.upper\_S = upper\_S (repeat for all mailboxes)

- create a task named "run"

- inside the task, create a forever loop

- inside the loop, create to threads using (join\_any)

- thread #1 (for transporting data from the upper layer to the electrical layer):

- for thread #1 do as follows:

1 - create a handle for "upper\_layer\_tr" class (where data will be saved): upper\_layer\_tr = new();

2 - if the upper\_layer\_tr.phase is equal to "5" do as follows:

- use the upper\_G mailbox to "get" data to be stored inside the "upper\_layer\_tr" handle

- next, use the upper\_to\_elec mailbox to "put" data that was stored inside the "upper\_layer\_tr" handle

- thread #2 (for transporting data from the electrical layer to the upper layer):

- for thread #2 do as follows:

1 - create a handle for "elec\_layer\_tr" class (where data will be saved): elec\_layer\_tr = new();

2 - if the elec\_layer\_tr.phase is equal to "5" do as follows:

- use the elec\_to\_upper mailbox to "get" data to be stored inside the "elec\_layer\_tr" handle

- next, assign the upper\_layer\_tr.T\_Data = elec\_layer\_tr.electrical\_to\_transport

- next, use the upper\_S mailbox to "put" data that was stored inside the "upper\_layer\_tr" handle

\*\*

#### Generated Code:

Here is the SystemVerilog class `upper\_ref\_AI` as per your requirements:

```systemverilog

class upper\_ref\_AI;

  // Instances of the classes

  elec\_layer\_tr elec\_layer\_tr\_inst;

  upper\_layer\_tr upper\_layer\_tr\_inst;

  // Mailboxes

  mailbox #(elec\_layer\_tr) upper\_S;

  mailbox #(elec\_layer\_tr) upper\_G;

  mailbox #(upper\_layer\_tr) upper\_to\_elec;

  mailbox #(elec\_layer\_tr) elec\_to\_upper;

  // Constructor

  function new(mailbox #(elec\_layer\_tr) upper\_S, mailbox #(elec\_layer\_tr) upper\_G, mailbox #(upper\_layer\_tr) upper\_to\_elec, mailbox #(elec\_layer\_tr) elec\_to\_upper);

    this.upper\_S = upper\_S;

    this.upper\_G = upper\_G;

    this.upper\_to\_elec = upper\_to\_elec;

    this.elec\_to\_upper = elec\_to\_upper;

  endfunction

  // Task run

  task run;

    fork

      // Thread 1

      begin : thread1

        forever begin

          upper\_layer\_tr upper\_layer\_tr\_handle = new();

          if (upper\_layer\_tr\_handle.phase == 5) begin

            upper\_G.get(upper\_layer\_tr\_handle);

            upper\_to\_elec.put(upper\_layer\_tr\_handle);

          end

        end

      end

      // Thread 2

      begin : thread2

        forever begin

          elec\_layer\_tr elec\_layer\_tr\_handle = new();

          if (elec\_layer\_tr\_handle.phase == 5) begin

            elec\_to\_upper.get(elec\_layer\_tr\_handle);

            upper\_layer\_tr\_inst.T\_Data = elec\_layer\_tr\_handle.electrical\_to\_transport;

            upper\_S.put(upper\_layer\_tr\_inst);

          end

        end

      end

    join\_any

  endtask

endclass

```

This class has two threads running in parallel. Thread 1 is responsible for transporting data from the upper layer to the electrical layer, and Thread 2 is responsible for transporting data from the electrical layer to the upper layer. Both threads run indefinitely due to the `forever` loop.

**-----------------------------------**

#### Prompt #2:

\*\*lets reformat the first thread,

- the handle created should be for the "upper\_layer\_tr\_inst" created before (no need for a new instance)

- the "get" operation should be before the if condition (the if condition should check the content that was just obtained via the "get" operation)

- "get" & "put" will obtain and insert data inside the "upper\_layer\_tr\_inst" ("upper\_layer\_tr\_handle" should be completely deleted)

\*\*

#### Generated Code:

Here's the reformatted first thread:

```systemverilog

// Thread 1

begin : thread1

  forever begin

    upper\_G.get(upper\_layer\_tr\_inst); // Get operation before the if condition

    if (upper\_layer\_tr\_inst.phase == 5) begin

      upper\_to\_elec.put(upper\_layer\_tr\_inst); // Put operation inside the if condition

    end

  end

end

```

In this updated version, we're using the `upper\_layer\_tr\_inst` instance that was created earlier. The `get` operation is now performed before the `if` condition, and the `put` operation is performed inside the `if` condition. The `upper\_layer\_tr\_handle` has been completely removed as per your request.

**-----------------------------------**

#### Prompt #3:

\*\*"the upper\_layer\_tr\_inst" needs a new handle at the begining of every loop to refresh its content : upper\_layer\_tr\_inst = new()\*\*

#### Generated Code:

Here's the updated first thread with a new instance of `upper\_layer\_tr\_inst` at the beginning of each loop:

```systemverilog

// Thread 1

begin : thread1

  forever begin

    upper\_layer\_tr\_inst = new(); // New instance at the beginning of each loop

    upper\_G.get(upper\_layer\_tr\_inst); // Get operation before the if condition

    if (upper\_layer\_tr\_inst.phase == 5) begin

      upper\_to\_elec.put(upper\_layer\_tr\_inst); // Put operation inside the if condition

    end

  end

end

```

#### Prompt #4:

\*\*repeat the same adjustments for the 2nd thread\*\*

#### Generated Code:

Here's the updated second thread with a new instance of `elec\_layer\_tr\_inst` at the beginning of each loop:

```systemverilog

// Thread 2

begin : thread2

  forever begin

    elec\_layer\_tr\_inst = new(); // New instance at the beginning of each loop

    elec\_to\_upper.get(elec\_layer\_tr\_inst); // Get operation before the if condition

    if (elec\_layer\_tr\_inst.phase == 5) begin

      upper\_layer\_tr\_inst.T\_Data = elec\_layer\_tr\_inst.electrical\_to\_transport;

      upper\_S.put(upper\_layer\_tr\_inst); // Put operation inside the if condition

    end

  end

end

```

In this updated version, a new instance of `elec\_layer\_tr\_inst` is created at the beginning of each loop to refresh its content. The `get` operation is performed before the `if` condition, and the `put` operation is performed inside the `if` condition.

**-----------------------------------**

### ELECTRICAL LAYER Class:

* Please note that these Prompts used to generate Phases 3 (Parameter Exchanging State) and 5(Data Transportation State) Only!

#### Prompt #1(Phase 5 Base Prompt):

consider #file:upper\_reference.sv #file:elec\_layer\_tr.svh #file:upper\_layer\_tr.svh for context

we will be creating a system verilog class that has a task performing some operations like the "upper\_ref\_AI class"

* create a class named "elec\_ref\_AI"
* class instantiates the two classes in the elec\_layer\_tr.svh , upper\_layer\_tr.svh files
* class instantiates 4 mailboxes which are:
  + name: elec\_S , can only use data type "elec\_layer\_tr"
  + name: elec\_G , can only use data type "elec\_layer\_tr"
  + name: elec\_to\_upper, can only use data type "elec\_layer\_tr"
  + name: upper\_to\_elec, can only use data type "upper\_layer\_tr"
* create a "new" function with the defined mailboxes as inputs
* inside the function, connect the mailboxes as follows : this.elec\_S = elec\_S (for all mailboxes)
* create a task named "run"
* inside the task, create a forever loop
* inside the loop, create 2 threads using fork join\_any
* for the 2nd thread, create a handle of the upper\_layer\_tr & elec\_layer\_tr (= new()) to refresh the contents each cycle
* continue with the 2nd thread, use the get function of the upper\_to\_elec mailbox to obtain data
* next, check that the upper\_layer\_tr component named phase is = 5, if true, proceed to assign the T\_Data of the "upper\_layer\_tr" to the "transport\_to\_electrical" component of the "elec\_layer\_tr"
* next, use the "put" function to send the "elec\_layer\_tr" via the elec\_S mailbox

#### Generated Code:

Here is the SystemVerilog code that implements your requirements:

class elec\_ref\_AI;

  elec\_layer\_tr elec\_layer;

  upper\_layer\_tr upper\_layer;

  mailbox #(elec\_layer\_tr) elec\_S;

  mailbox #(elec\_layer\_tr) elec\_G;

  mailbox #(elec\_layer\_tr) elec\_to\_upper;

  mailbox #(upper\_layer\_tr) upper\_to\_elec;

  function new(mailbox #(elec\_layer\_tr) elec\_S, mailbox #(elec\_layer\_tr) elec\_G, mailbox #(elec\_layer\_tr) elec\_to\_upper, mailbox #(upper\_layer\_tr) upper\_to\_elec);

    this.elec\_S = elec\_S;

    this.elec\_G = elec\_G;

    this.elec\_to\_upper = elec\_to\_upper;

    this.upper\_to\_elec = upper\_to\_elec;

  endfunction

  task run;

    forever begin

      fork : thread

        // Thread 1

        begin

        end

        // Thread 2

        begin

          elec\_layer = new();

          upper\_layer = new();

          upper\_to\_elec.get(upper\_layer);

          if (upper\_layer.phase == 5) begin

            elec\_layer.transport\_to\_electrical = upper\_layer.T\_Data;

            elec\_S.put(elec\_layer);

          end

        end

      join\_any

    end

  endtask

endclass

#### Prompt #2:

change elec\_layer and upper\_layer names to be more representative (add inst for example )

* for the first thread, create a handle for the "elec\_layer\_tr"
* "get" data via the elec\_G mailbox to be stored in "elec\_layer\_tr".
* next, create a case statement with the argument being the "phase" component of the "elec\_layer\_tr"
* the phase has 4 possibilities , (2 or 3 or 4 or 5)
* in case of phase = 5: "put" the "elec\_layer\_tr" via the "elec\_to\_upper" mailbox

#### Generated Code:

class elec\_ref\_AI;

  elec\_layer\_tr elec\_layer\_inst;

  upper\_layer\_tr upper\_layer\_inst;

  mailbox #(elec\_layer\_tr) elec\_S;

  mailbox #(elec\_layer\_tr) elec\_G;

  mailbox #(elec\_layer\_tr) elec\_to\_upper;

  mailbox #(upper\_layer\_tr) upper\_to\_elec;

  function new(mailbox #(elec\_layer\_tr) elec\_S, mailbox #(elec\_layer\_tr) elec\_G, mailbox #(elec\_layer\_tr) elec\_to\_upper, mailbox #(upper\_layer\_tr) upper\_to\_elec);

    this.elec\_S = elec\_S;

    this.elec\_G = elec\_G;

    this.elec\_to\_upper = elec\_to\_upper;

    this.upper\_to\_elec = upper\_to\_elec;

  endfunction

  task run;

    forever begin

      fork : thread

        // Thread 1

        begin

          elec\_layer\_tr elec\_layer\_handle;

          elec\_G.get(elec\_layer\_handle);

          case (elec\_layer\_handle.phase)

            5: elec\_to\_upper.put(elec\_layer\_handle);

          endcase

        end

        // Thread 2

        begin

          elec\_layer\_inst = new();

          upper\_layer\_inst = new();

          upper\_to\_elec.get(upper\_layer\_inst);

          if (upper\_layer\_inst.phase == 5) begin

            elec\_layer\_inst.transport\_to\_electrical = upper\_layer\_inst.T\_Data;

            elec\_S.put(elec\_layer\_inst);

          end

        end

      join\_any

    end

  endtask

endclass

In the first thread, a handle elec\_layer\_handle for elec\_layer\_tr is created. Data is then gotten from the elec\_G mailbox and stored in elec\_layer\_handle. A case statement is then used to check the phase of elec\_layer\_handle. If the phase is 5, elec\_layer\_handle is put into the elec\_to\_upper mailbox.

#### Prompt #3:

creating elec\_layer\_tr handle means that elec\_layer\_inst = new(), please adjust the first thread accordingly

#### Generated Code:

class elec\_ref\_AI;

  elec\_layer\_tr elec\_layer\_inst;

  upper\_layer\_tr upper\_layer\_inst;

  mailbox #(elec\_layer\_tr) elec\_S;

  mailbox #(elec\_layer\_tr) elec\_G;

  mailbox #(elec\_layer\_tr) elec\_to\_upper;

  mailbox #(upper\_layer\_tr) upper\_to\_elec;

  function new(mailbox #(elec\_layer\_tr) elec\_S, mailbox #(elec\_layer\_tr) elec\_G, mailbox #(elec\_layer\_tr) elec\_to\_upper, mailbox #(upper\_layer\_tr) upper\_to\_elec);

    this.elec\_S = elec\_S;

    this.elec\_G = elec\_G;

    this.elec\_to\_upper = elec\_to\_upper;

    this.upper\_to\_elec = upper\_to\_elec;

  endfunction

  task run;

    forever begin

      fork : thread

        // Thread 1

        begin

          elec\_layer\_inst = new();

          elec\_G.get(elec\_layer\_inst);

          case (elec\_layer\_inst.phase)

            5: elec\_to\_upper.put(elec\_layer\_inst);

          endcase

        end

        // Thread 2

        begin

          elec\_layer\_inst = new();

          upper\_layer\_inst = new();

          upper\_to\_elec.get(upper\_layer\_inst);

          if (upper\_layer\_inst.phase == 5) begin

            elec\_layer\_inst.transport\_to\_electrical = upper\_layer\_inst.T\_Data;

            elec\_S.put(elec\_layer\_inst);

          end

        end

      join\_any

    end

  endtask

endclass

In the first thread, a new instance of elec\_layer\_tr is created and assigned to elec\_layer\_inst. The elec\_G mailbox then gets data and stores it in elec\_layer\_inst. The case statement checks the phase of elec\_layer\_inst and if it's 5, elec\_layer\_inst is put into the elec\_to\_upper mailbox.

#### Prompt #4 (Phase 3 Base Prompt):

in case of phase = 3, do as follows:

- assign the following values to the "elec\_layer\_tr" components (SENDING AT COMMAND TO REQUEST DATA)

- tr\_os = tr

- transaction\_type = AT\_cmd

- address = 78

- len = 3

- read\_write = 0 (indicates read operation)

- crc\_received = 0 (to be changed later)

- "put" the contents of the elec\_layer\_inst in the "elec\_S" mailbox

- next, create a new instance to remove contents of the previous instance

- "get" elec\_layer\_inst via the "elec\_G" mailbox

- we should be waiting for transaction\_type = AT\_cmd (to respond with an AT\_rsp along with appropriate elec\_layer\_tr parameters)

- check if the contents of the elec\_layer\_tr are as follows:

- tr\_os = tr

- transaction\_type = AT\_cmd

- address = 78

- len = 3

- read\_write = 0 (indicates read operation)

- if the previous condition is true, do as follows:

- tr\_os = tr

- transaction\_type = AT\_rsp

- address = 78

- len = 3

- read\_write = 0 (indicates read operation)

- crc\_received = 0 (to be changed later)

- cmd\_rsp\_data = 24'h053303 (1st Byte idicates: both lanes are enabled, 2nd Byte indicates: GEN3 speed is supported on both lanes, 3rd byte indicates: GEN 4 speed is supported on both lanes)

- put the contents of the "elec\_layer\_tr" via the elec\_S mailbox

- create a new instance to reset the "elec\_layer\_tr" as usual

#### Generated Code:

```systemverilog

task run;

  forever begin

    fork : thread

      // Thread 1

      begin

        elec\_layer\_inst = new();

        elec\_G.get(elec\_layer\_inst);

        case (elec\_layer\_inst.phase)

          5: elec\_to\_upper.put(elec\_layer\_inst);

          3: begin

            elec\_layer\_inst.tr\_os = tr;

            elec\_layer\_inst.transaction\_type = AT\_cmd;

            elec\_layer\_inst.address = 78;

            elec\_layer\_inst.len = 3;

            elec\_layer\_inst.read\_write = 0;

            elec\_layer\_inst.crc\_received = 0;

            elec\_S.put(elec\_layer\_inst);

            elec\_layer\_inst = new();

            elec\_G.get(elec\_layer\_inst);

            if (elec\_layer\_inst.tr\_os == tr && elec\_layer\_inst.transaction\_type == AT\_cmd && elec\_layer\_inst.address == 78 && elec\_layer\_inst.len == 3 && elec\_layer\_inst.read\_write == 0) begin

              elec\_layer\_inst.tr\_os = tr;

              elec\_layer\_inst.transaction\_type = AT\_rsp;

              elec\_layer\_inst.address = 78;

              elec\_layer\_inst.len = 3;

              elec\_layer\_inst.read\_write = 0;

              elec\_layer\_inst.crc\_received = 0;

              elec\_layer\_inst.cmd\_rsp\_data = 24h'053303;

              elec\_S.put(elec\_layer\_inst);

              elec\_layer\_inst = new();

            end

          end

        endcase

      end

      // Thread 2

      begin

        elec\_layer\_inst = new();

        upper\_layer\_inst = new();

        upper\_to\_elec.get(upper\_layer\_inst);

        if (upper\_layer\_inst.phase == 5) begin

```