# **GPIO Agent Documentation**

Vaibhavi Rastogi – Intern, Samsung Semiconductor India Research (June 2020)

**Table of Contents**

1. Introduction

2. Components in an Agent

3. Requirements

4. Functionalities and Variations

5. Approach to writing a GPIO Agent

6. Testbench diagram

7. Reusability

8. Overview

9. References

Following is the EDA Playground link to the TB code: <https://www.edaplayground.com/x/sfaw>

1. **Introduction**

This project aims to construct a GPIO agent, which stands for General Purpose Input Output agent. This generic sideband agent is primarily responsible for driving certain general-purpose I/O signals to the DUT.

The signals in the testbench can be asynchronous as well as synchronous to a clock. There can be a value before reset for these signals. This will coexist alongside a primary agent. The sideband agent will not operate per any protocol but will work per the user’s requirements.

The user can configure the functionality of the agent as per their usage and the same agent can be reusable in several projects. In each project, the engineer will input their configuration in form of a configuration class. All other components of the testbench will be completely generic. User intervention is limited to configuration classes only. The user is not permitted to modify the code.

**2. Components in a GPIO Agent**

An Agent consists of Monitor, Driver, Configuration class, Sequence item, Sequence, and Sequencer. Since the rest of the testbench except for the configuration class is completely generic, its structure will be the same as in a traditional UVM testbench. The rest of the testbench contains the environment, scoreboard, test, and the DUT which is connected to the testbench through the interface, and then there’s TB Top.

**3. Requirements**

1) Interface

List of signals:

i) Clock, reset: These signals are the system’s clock and reset.

ii) Input signals: Input signals to be declared in the interface do the job of driving to the DUT (if the selected mode is mode 1)

iii) Output signals: Output signals to be declared in the interface do the job of collecting information from the DUT through the monitor and carrying out checks in the scoreboard.

iv) *synchronous* -> these signals are synchronous to a clock, meaning that they would be sampled with the help of a clock(*clk*).

v) *asynchronous* - > these signals are asynchronous to a clock, meaning that they’d be dependent on themselves for sampling. Alternatively, they would be driven whenever they themselves become high.

vi) Variable signal/bus width: Since the agent is generic, the signals to be driven have bus widths that are not hardcoded but are set depending on the user’s requirements, and hence are variable.

2) Configuration class

● A configuration class is to be created in the environment.

● Since the user is forbidden to touch the actual code, the user will be providing their inputs(i.e. signals values, bus widths and reset values) in the extended class(child class), which will be overriding the base class.

● The virtual tasks will be declared in the parent class(agent\_config class) and defined in the child class, where the user provides their input. The object will be created for the base class which will be ‘set’ using a configuration database. The tasks will be declared as virtual in the base class, and they will be defined in the child class, based on the concept of polymorphism and method overriding.

● A task get\_val would be used to take input from the user, tasks will execute in simulation time.

● There is one other bit called reset\_agent which when set to 1, sets all signals to their default value.

● Inclusion of all relevant files: There will be two files. One file will include all the component files such as test, env, agent, driver, monitor, etc and another file will include all the macros. Both of these files will then be included in the TB Top.

● The signals are declared in this fashion:

The entire bus width is used and is divided into three parts. There are signals dedicated to synchronous signal usage and asynchronous usage.

*bit [`MAX\_BUSWIDTH:`SYNCINPUT] synchronousinput;*

*bit [`SYNCINPUT-1:`SYNCOUTPUT] synchronousoutput;*

*bit [`SYNCOUTPUT-1:`ASYNCINPUT] asynchronousinput;*

*bit [`ASYNCINPUT-1:0] asynchronousoutput;*

*bit [`MAX\_SYNCINPUT:0] synchronousinput;*

*bit [`MAX\_SYNCOUTPUT:0] synchronousoutput;*

*bit [`MAX\_ASYNCINPUT:0] asynchronousinput;*

*bit [`MAX\_ASYNCOUTPUT:0] asynchronousoutput;*

● The object of the config class once set using config database in the environment, can be retrieved from the sequence item, or any other component requiring this information. The methods and variables of this class can then be used. The configuration class object will have to be set in the build phase of the env.

3) Using GPIO as Input or Output

The GPIO can be used in whichever way the user likes, and the agent can be used to drive values to the DUT (GPIO as INPUT) as well as monitor values from the DUT and comparison in the scoreboard (GPIO as OUTPUT).

4) Passing user-inputted BUS WIDTHS to the entire testbench

For accomplishing this task, we will be using macros to define bus widths as a separate file and then include that file in TB-top.

**4. Functionalities and variations**

The following points are the functionalities of the GPIO Agent.

a) Usage as Input (MODE 1):   
If the sideband agent is to be used as input, then the agent would do the job of driving signal values to the DUT. In this case, the user would know how the signals behave and thus can control the way in which the stimulus to be driven is generated. Users would know what bit of which signal is to be accessed and modified, like in the case of accessing register bits and modifying the value of a field (as an example).

Hence there needs to be a provision such that the user can change just one bit of any of the signals to be driven to the DUT, at any point in time, instead of driving all of them at once.

b) Usage as Output (MODE 2):   
If the sideband agent is to be used as output, then observed values will be taken from the monitor and checked in the scoreboard whether those values match the expected values.

c) Reset: Reset signal resets the signals to their default values when asserted. When a system starts, signals are reset automatically, to reset any garbage values. Elsewhere in the simulation also reset can be done.

**Variations:**   
  
GPIO Agent can be used for several purposes some of which include modifying SFRs (Special Function Registers), driving and monitoring sideband interrupts, reset operations or usecases at IP level where both input and output variations are possible.

**6. Approach to writing a GPIO Agent**

● The core of the GPIO Agent is the agent configuration class.

● The task(s) that will be used for obtaining the user information will be declared virtually in the *agent\_config* class. The virtual declaration is crucial so that method overriding can occur in the extended class where the extended class is assigned to the base class.

● The child class will have the definition of those tasks.

● The concept used is of overriding: To **override** a method means that given a base class(*agent\_config*) with a task, we can define a subclass (child class) that extends from that base class and then provide a new definition for the given task.

● The child class will be the only aspect of the testbench available to the user to give input to.

● List of signals required to be input from the user:

i)Bus widths for all the signals

ii)Reset values for the signals (which means the default values of the signals)  
iii) Values to be driven for all the signals

iv) MODE Selection, whether the user requires GPIO to be used as output/input.

**CONCEPT OF MAPPING**

i) The user gives input (mode 01 or 10, int 0 or 1, etc) inside *agent cfg* child class in run phase.

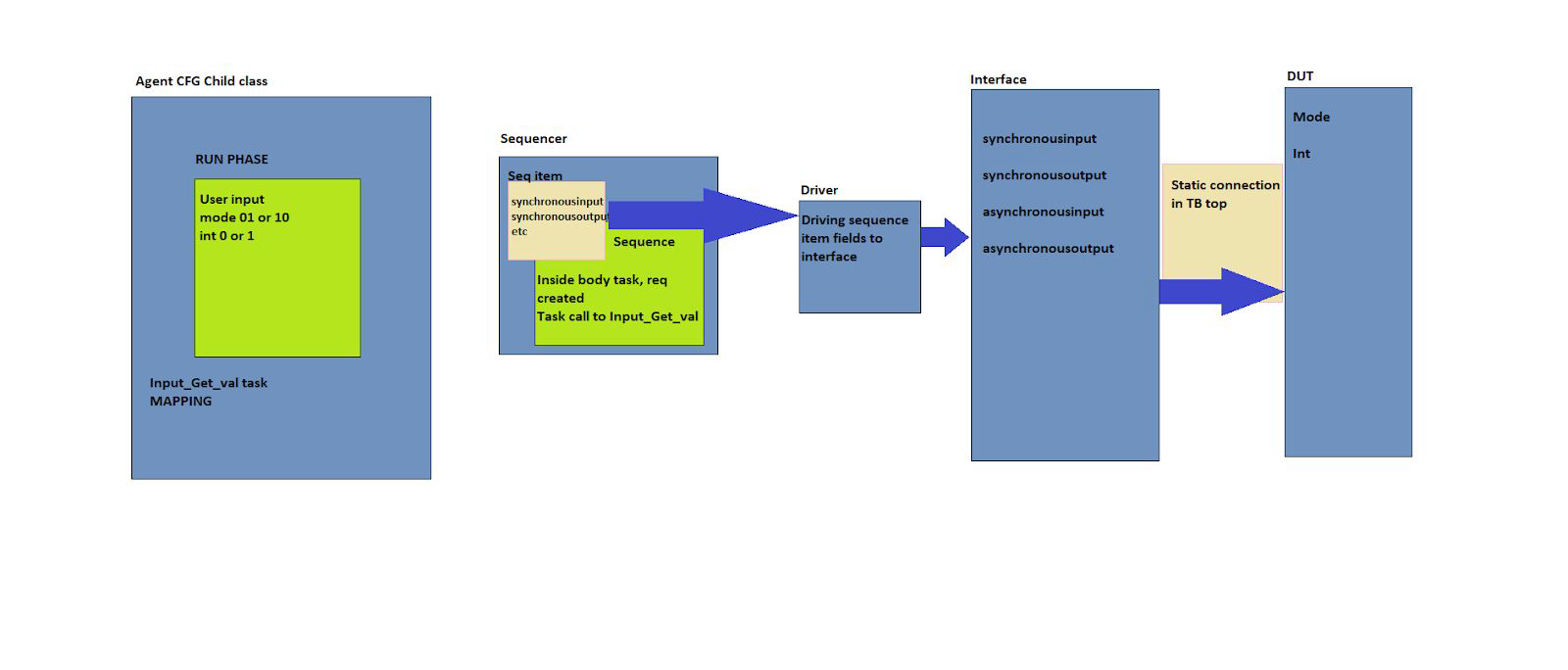
ii) In the body task in sequence, the *sequence\_item* object is created and the *get\_va*l task is called which is present in the *agent\_config* child class.

iii) In the child class, the mapping of *user\_input* variables will be done to generic testbench variables (for example mode[1:0] will be mapped to synchonousinput[1:0]).

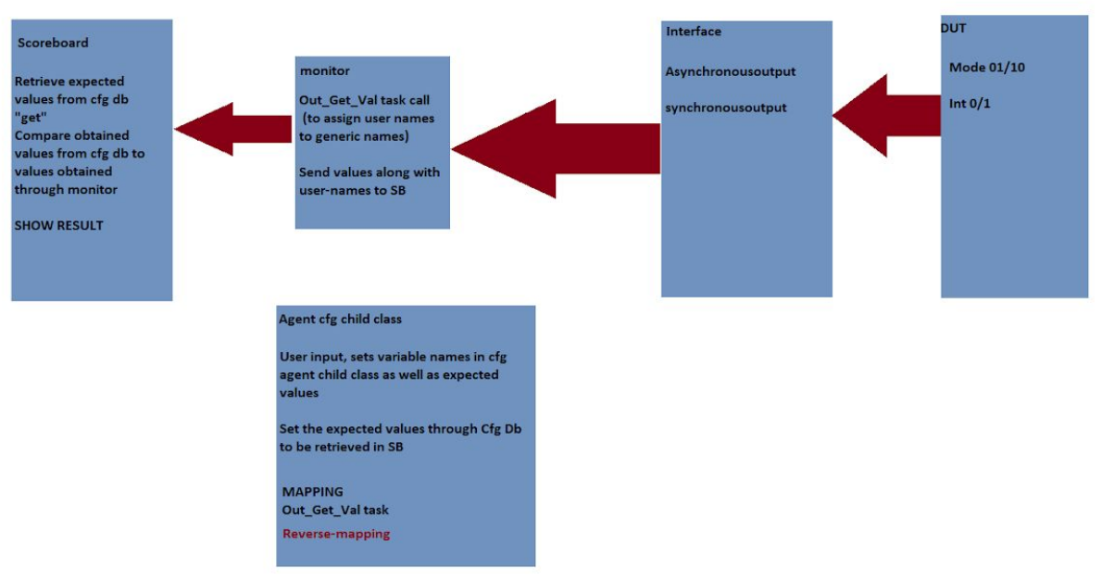
iv) Then, these values when assigned to the generic testbench variables (*synchronousinput*, *asynchronousinput*) will be driven by the driver onto the interface into the DUT.

v) The DUT will also have signals in terms of what the user entered (mode and int, for example). The respective connections will be done in TB top.

Below is a diagram indicating the same:



**For GPIO Output functionality, below is a diagram:**

****

i) the user is supposed to set the user-input names in *Cfg Agent* child class, as well as expected values for the same, are to be “Set” using Configuration Db.

ii) The user-input signal values (mode 10 /int 1, for example) will be given to the interface (where signal names are *syncoutput* and *asyncoutput*), and then to the monitor, where the monitor does task call to a function *Out\_get\_val* (present in the *Agent cfg*)

iii) In the *Out\_get\_Val* function task call, the interface and a newly created sequence item object will have to be passed as a reference.

iv) The *Out\_Get\_Val* is responsible for reverse mapping from generic

synchronous/asynchronous variable names to user-input names like mode/int.

v) After execution of the task, the newly created seq item object will have fields with the name and value of *user\_input* sequences which will be sent to the scoreboard using TLM ports.

iv) Scoreboard “Gets” the expected values through Config Db, and compares the two (comparison is valid since the expected value and obtained value will have the same width, as the width is set globally at the beginning itself and continue to remain the same)

v)The scoreboard also displays the result.

We’ll need to pass the interface in the task Out\_get\_val in order for the task to equate and reverse-map.   
  
**Concept of “Inside Constraint”**

Since there can be several IPs present for which the GPIO Agent can be utilized as input or as output, there need to be present certain constraints.

Example: For IP1, if one of the fields is Mode whose value is limited to 01 or 10, then the constraint needs to be

*constraint mode\_range { mode inside { 1,2}; }*

Similarly, for IP3, Mode values can be 000, 001, 011, 101.

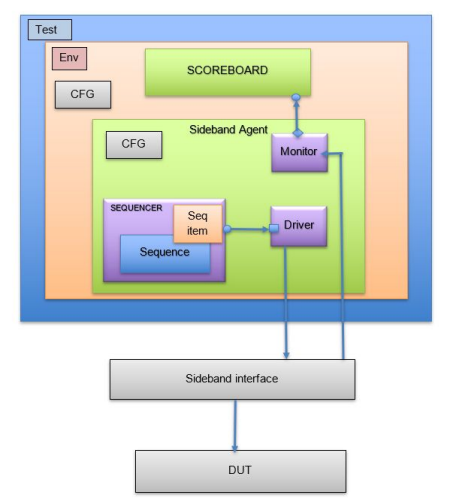
Then constraint needs to be

*constraint mode\_range { mode inside { 0,1,3,5}; }*

These constraints will be introduced inside a class extended from *sequence\_item*.

**7. Testbench diagram**

**The CFG class is inside ENV. It is parallel to agent and scoreboard. This is why it is in ENV that we “set” configuration class in config\_db.**

****

**8. Reusability**

The GPIO agent is designed keeping in mind its reusability. This means that it can be used across different projects and configured as per the user’s requirement. Because of this, all the components are to be designed as generic as possible except the *agent\_configuration* and its child class.

**9. Overview/ Conclusion**

i) GPIO Agent is generic, protocol-free, and has two modes, input, and output.

ii) Input mode drives signals to the DUT and output mode checks via the scoreboard checker that it is driven correctly. Input and Output buses need to be separate, along with synchronous and asynchronous buses.

iii) The crucial aspect of implementation is the configuration class and its child class that will be used to take user input values. Concepts of polymorphism, method overriding, and virtual methods are used here.

iv) User input Bus widths will be set through macros, reset(default) values, and simulation time values/stimulus of the signals will be set through extended agent configuration class

v) Full bus width needs to be occupied for driving or collection of information through the interface from the DUT.

vi) The GPIO agent is intended to be reusable which means that it can be configured as per requirement and used in multiple projects.

**10. References**

1. verificationguide.com/uvm/uvm-tutorial/
2. www.doulos.com/knowhow/systemverilog/uvm/uvm-verification-primer/
3. www.chipverify.com/uvm/uvm-tutorial
4. System Verilog for Verification: A Guide to Learning the Testbench Language Features, Chris Spear
5. UVM Cookbook

---------------------------------------------------------------------------------------------------------------------

**Addendum:**

**Gpio\_seq\_item -> gpio\_set\_item\_Ext -> gpio\_seq -> config\_Agent**

**Gpio\_Seq\_item basically has all the fields/members, gpio\_seq\_item\_ext will have the struct needed by the user to make some modification later, the gpio\_seq will just randomize it (or not based on the time)**

**Config agent has the mapping functions for the input and outputs , mapping from tb variables to interface variables, and vice versa. So four functions.**

**It has below 4 types of class members:**

**int size\_syncip[];**

**string name\_syncip[];**

**bit[`MAX\_SYNCOP-1:0] resetval\_syncip[string];**

**int size\_asyncip[];**

**string name\_asyncip[];**

**bit[`MAX\_ASYNCOP-1:0] resetval\_asyncip[string];**

**int size\_syncop[];**

**string name\_syncop[];**

**int size\_asyncop[];**

**string name\_asyncop[];**

**to configure sync inputs, async inputs, sync outputs and async outputs. Outputs won’t require values to be driven, only names/sizes to be driven, so value member isn’t required.**

**For inputs, resetval is required but values at a later stage are either randomized or are again provided by user.**

**Mapping function explanation:**The function `get\_val\_syncip()` is designed to map the user-input signals to interface or testbench signals for synchronous input signals. The user-input signals are contained in the associative array `val\_syncip[]` indexed by strings, and the function returns a bit array representing the mapped synchronous input signals.

Here's how this function works:

1. `totip` is declared as a bit vector. This variable will accumulate the mapped bit patterns of synchronous input signals and be returned as the result of the function.

2. `temp` is declared as an integer. It is used to keep track of the current position in the output bit vector `totip`.

3. The function then enters a loop that iterates over the `size\_syncip[]` array, which likely contains the sizes of each synchronous input signal.

4. Inside the loop, it first checks if `reset\_agent` equals to 1. If it does, it uses the reset values (`resetval\_syncip[]`) for each synchronous input signal; otherwise, it uses the user-input values (`val\_syncip[]`).

5. It then multiplies the selected value by `2\*\*temp`, effectively shifting the value `temp` number of places to the left in binary form, and adds this result to `totip`. This step is essentially gathering the input signals and packing them into a single bit vector.

6. The `$display` line (which is commented out) would display the value of `totip` at each iteration, if uncommented.

7. `temp` is then incremented by `size\_syncip[i]`, effectively moving the "cursor" for the next value to be added to `totip`.

8. Finally, `totip` is returned as the result of the function.

Essentially, this function is taking multiple user-input signals, each of which can be a different size, and packing them into a single bit vector in a certain order. The `reset\_agent` variable allows you to switch between using the user-input values and some predefined reset values.

However, it's worth noting that this function will work properly if the total of all `size\_syncip[i]` values does not exceed the size of `totip` (`MAX\_SYNCINPUT`). If the total does exceed `MAX\_SYNCINPUT`, some input signals will be lost. Also, the specific values of `MAX\_SYNCINPUT` and `MAX\_SYNCOP` would likely be defined elsewhere in the code. They set the maximum sizes for the output bit vector and the input associative array respectively.

**Output function explanation**:

The function `put\_val\_syncop()` maps synchronous testbench/interface signals to user-input signals. It takes in a bit vector (`synchronousoutput`) and a transaction item (`trans`), and updates the transaction item based on the input bit vector.

Here's a breakdown of what this function does:

1. First, it checks if the total size of synchronous outputs (obtained by summing up the `size\_syncop[]` array) exceeds the synchronous output bus width (`MAX\_SYNCOUTPUT`). If it does, a fatal error is raised using the UVM function `uvm\_fatal()`. This prevents the function from proceeding if the provided synchronous output values cannot fit within the available bus width.

2. The input bit vector `synchronousoutput` is copied to `totop`. This variable will be used to extract the value for each individual output signal.

3. The function then enters a loop that iterates over the `size\_syncop[]` array, which likely contains the sizes of each synchronous output signal.

4. Inside the loop, it performs the following operations:

- It uses the modulo operation `%` to extract the least significant bits from `totop` based on the size of the current output signal (`trans.size\_syncop[i]`).

- It assigns these bits to the corresponding user-input signal in the `val\_syncop[]` associative array of the transaction item `trans`.

- `totop` is then divided by `2\*\*trans.size\_syncop[i]`, effectively shifting it right by `trans.size\_syncop[i]` places. This step prepares `totop` for extraction of the next signal's bits.

5. Finally, the updated transaction item `trans` is returned.

In essence, this function takes a bit vector representing multiple synchronous output signals packed together, and breaks it apart into individual output signals stored in a transaction item. It uses a UVM transaction object (`gpio\_trans\_item`) to store these individual signals along with their respective sizes and names. The output signals are unpacked in a certain order and the order depends on how they were packed in the `get\_val\_syncip()` function.

This function, therefore, essentially does the reverse of what `get\_val\_syncip()` does: while `get\_val\_syncip()` packs multiple user-input signals into a single bit vector, `put\_val\_syncop()` unpacks a single bit vector into multiple output signals.

Now check driver-> monitor-> agent -> env->test->testbench

Note: Here is can be helpful to remember the demarcation between user-input signals and Interface/TB signals.

User input signals are syncip, asyncip, syncop, asyncop.

The seq\_item signals are just prefixed with a val, val\_syncip, valsyncop, etc and the name and size signals are added (name\_syncip, size\_asyncop, etc)

In the driver when we map them to the interface, we map them to interface signals that are synchronousinput, synchronousoutput, etc.

It is merely a coincidence that DUT signals are named similarly to seq\_item signals…they could have been named anything. What’s important is that in testbench\_top when we connect interface to DUT, we pass interface signals (synchronousinput, etc) to the DUT instantiation which connects interface and DUT.

**It’s important to remember that user intervention is strictly limited to agent\_config\_ext class and gpio\_seq\_item\_ext class(For changes in signal values later on)**