UVM General Purpose Input/Output Agent
- User Manual -

Author : Ivan Mokanj
Date : 2017

Table of Contents

1 Introduction......3
2 GPIO agent......5
  2.1 General......5
  2.2 Folder structure and source files......9
  2.3 Package......10
  2.4 Interface......14
  2.5 Configuration......15
3 Instructions and usage......16
  3.1 Agent instantiation......16
    3.1.1 Quick summary......16
    3.1.2 Detailed steps......17
  3.2 GPIO agent API and waveforms......21
    3.2.1 setPin task......21
    3.2.2 setWindow task......23
    3.2.3 getPin task......25
    3.2.4 getCompare task......26
  3.3 SPI protocol example......28
4 Compatibility and issues......29
  4.1 Mentor Questa......29
  4.2 Cadence IES......29
  4.3 Aldec Riviera Pro......30

1 Introduction

Almost every testbench needs a GPIO (General Purpose Input/Output) agent. The main use of this UVM VIP is:

- driving individual design under test (DUT) input pins
- reading values from desired pins connected to the GPIO interface
- reading pin values and comparing them to user specified expected values
- monitoring the interface for unknown or high impedance logic values

The GPIO agent is ideal for controlling DUT signals which are not part of a standardized protocol (SPI, AXI, Avalon-MM, etc). Some such well known signals are **`reset`**, **`done`**, **`interrupt`**, etc. A subset of these signals is almost always present in a DUT and usually don't have a specific agent used exclusively for them. This is where the GPIO agent comes into play. All these general signals which are not part of a standardized protocol should be connected to the GPIO agent interface, and controlled by the GPIO agents easy to use API (Application Programming Interface). An example of a use scenario would be asserting and de-asserting the **`reset`** signal at the beginning, or during, the simulation. Another application of the agent would be reading the status of a desired output pin, and then performing additional actions, if needed.

Signals which are part of a protocol would have their specific agents that would be responsible for driving and reading them, through their standardized interface. On the other hand, if an agent for a standardized protocol is not available the GPIO agent can be used to control these signals. It would be easy to use the GPIO agents facilities to write tasks which mimic standardized protocols. An example of using the GPIO agent to control an SPI interface is shown in chapter 3.3.

There are, effectively, three ways that the GPIO agent should be used in the UVM verification environment, and all of them depend on how the GPIO interface is connected to the DUT. These schemes are presented and described in the coming chapters. The recommended way to use the GPIO agent would be to connect all unused DUT I/O pins to the agent, or to connect all unused input pins and all DUT output pins. In both cases the unused DUT input pins can be driven, while the other input pins are controlled by a specific agent, used for that standardized interface. In the first case only the unused DUT output pins can be read by the GPIO agent, while in the second case all DUT output pins can be read by the GPIO agent. The third option of connecting all DUT I/O pins to the GPIO agent is not recommended, because this could pose a problem if the signals are already connected to some other VIP. For example, the GPIO agent should not be used to drive the AXI4 signals if there is already a AXI4 agent connected to the DUT. In this scenario the GPIO should be used, at most, to read the AXI4 agent’s signals and not drive them, or there could be undesired effects.

The detailed explanation of the GPIO agent, its features and use cases will be presented in the coming chapters.

2 GPIO agent

2.1 General

The architecture of the GPIO agent is shown in Figure 1. As it can be seen, the architecture follows the Universal Verification Methodology (UVM) recommendations for building Verification Intellectual Properties (VIP). The agent contains a sequencer-driver pair, a monitor and a configuration object. It communicates with the GPIO interface using the virtual interface construct, in order to control DUT I/O pins.

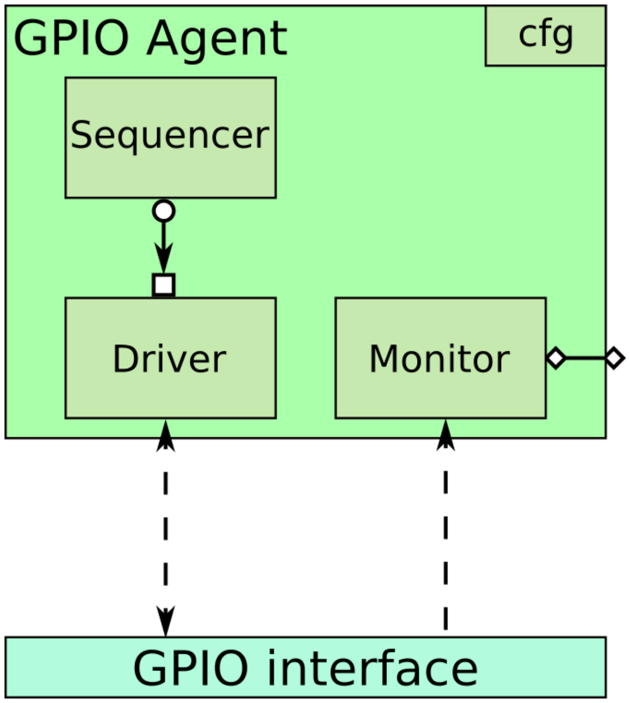


Figure 1 – GPIO agent architecture

The introductory chapter mentioned that there are three ways that the GPIO agent can be connected to the DUT. These methods are shown in Figures 2, 3 and 4. When the agent is used as depicted in Figure 1 the GPIO interface is only connected to the unused DUT I/O pins and no additional signals are needed in the top SystemVerilog testbench module, because no signals are shared between interfaces. In this mode of operation only a subset of the DUT pins can be read/written, but it is a viable and easy option to setup and use.

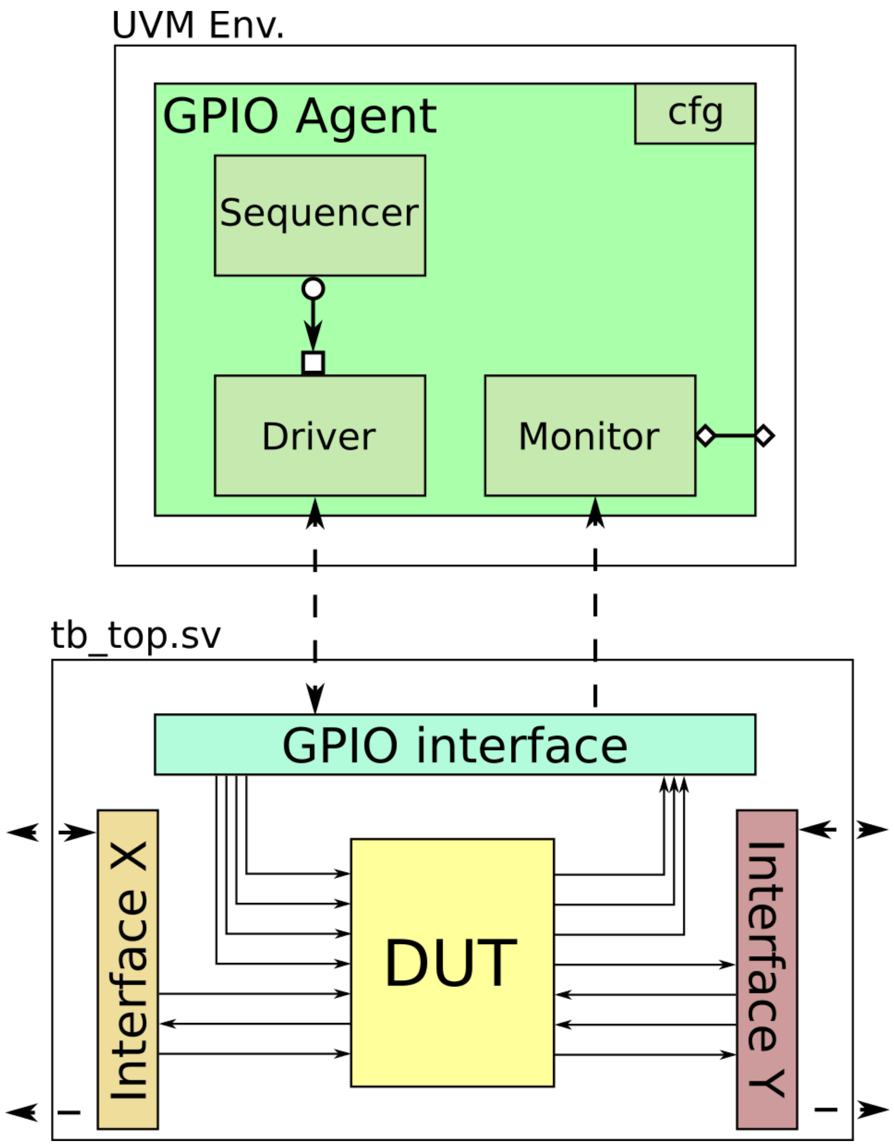


Figure 2 – GPIO agent used on unused DUT pins

The second use case assumes that the GPIO interface is connected to the unused pins and the DUT output pins that are connected to interfaces of other agents. Additional signals are needed in the top module to accomplish this, but the extra code is not that cumbersome. This, along with the method shown in Figure 1, is the recommended way of connecting the GPIO interface to the DUT, as it has a nice balance of functionality compared to the number of additional glue logic.

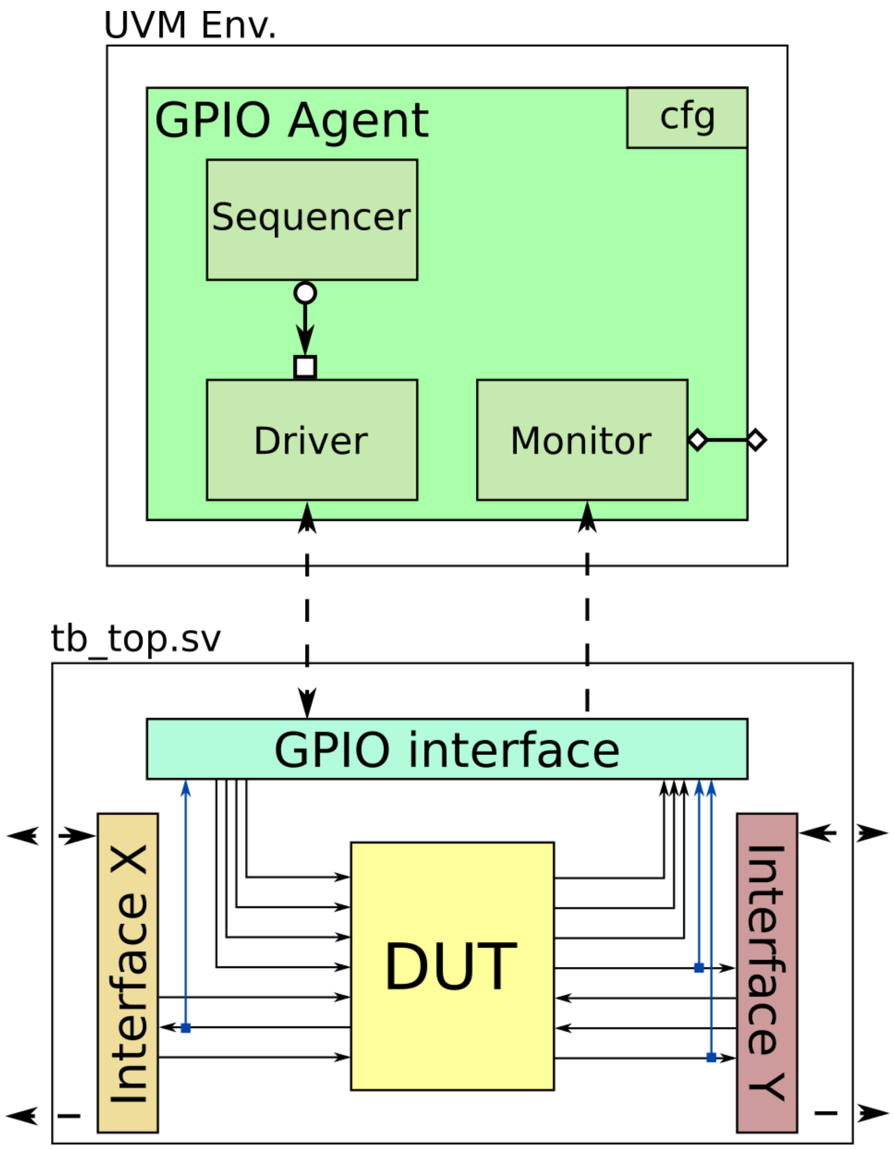


Figure 3 – GPIO agent used on a subset of DUT pins

The third way of connecting the GPIO agent and the DUT signals is shown in Figure 4 and it assumes connecting all DUT I/O signal to the GPIO interface. This method provides the most control to the user, however, it has the most glue logic that needs to be connected. Also, there is arguably little to be gained by being able to read the DUT input signals of other protocols, but this statement varies on a case by case basis, and can sometime be very desirable. All things considered, this is also a valid use of the GPIO agent.

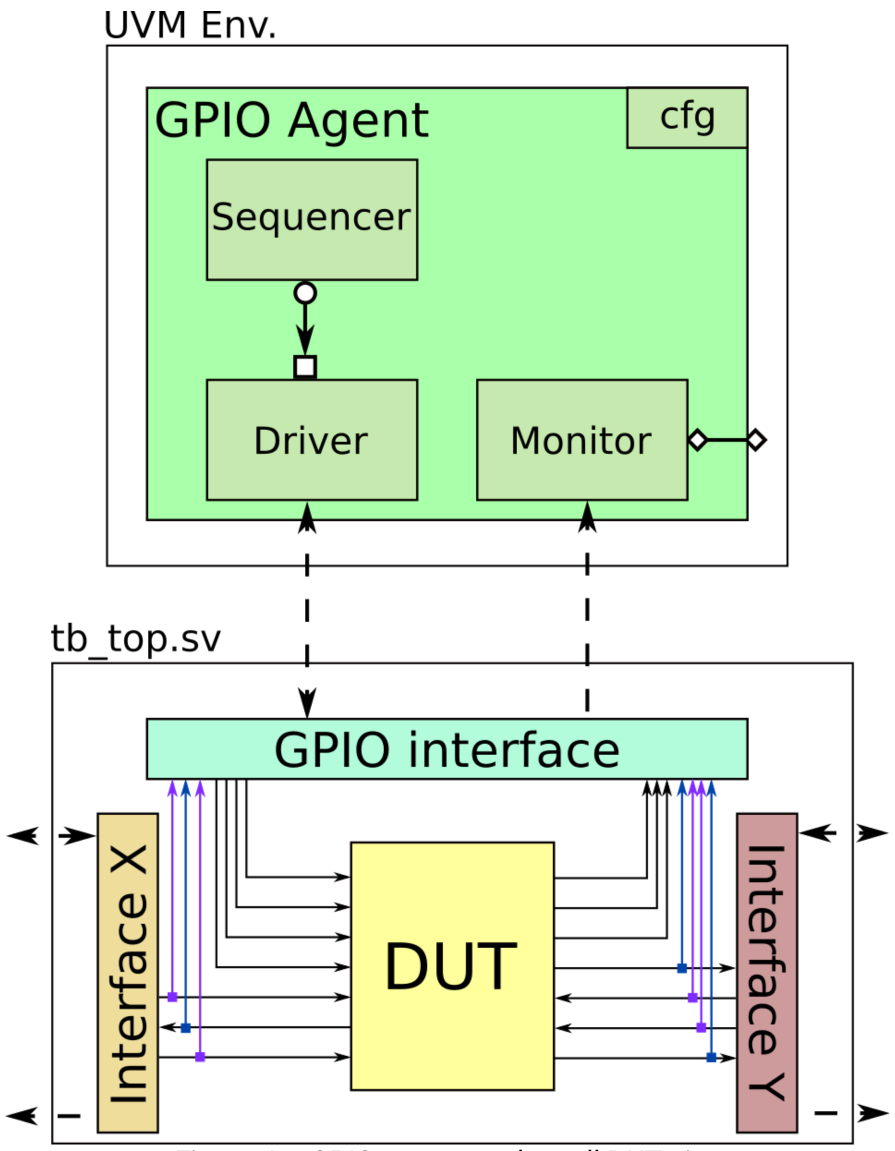
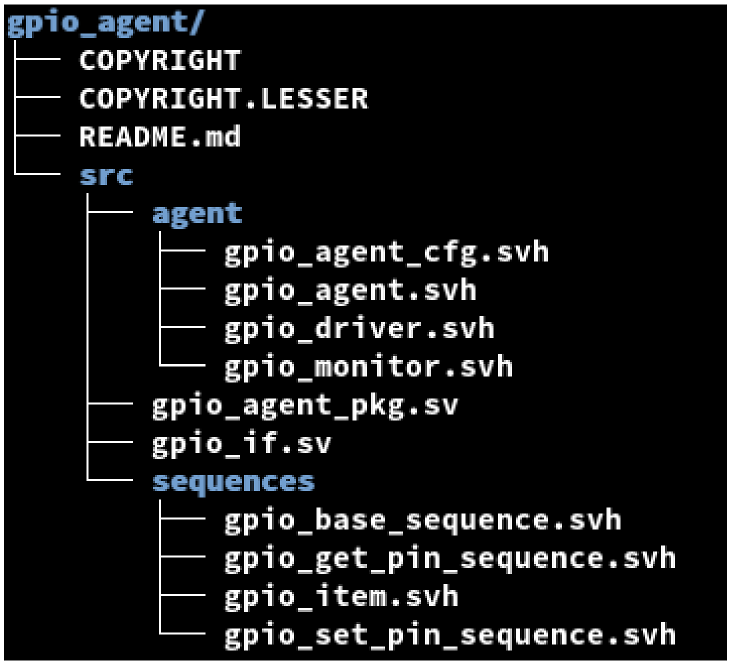


Figure 4 – GPIO agent used on all DUT pins

2.2 Folder structure and source files

The file and folder structure of the GPIO agent is shown in Figure 5, while the descriptions of the source files that make up the GPIO agent architecture are shown Table 1.



```
gpio_agent/
├── COPYRIGHT
├── COPYRIGHT.LESSER
├── README.md
└── src
    ├── agent
    │   ├── gpio_agent_cfg.svh
    │   ├── gpio_agent.svh
    │   ├── gpio_driver.svh
    │   └── gpio_monitor.svh
    ├── gpio_agent_pkg.sv
    ├── gpio_if.sv
    └── sequences
        ├── gpio_base_sequence.svh
        ├── gpio_get_pin_sequence.svh
        ├── gpio_item.svh
        └── gpio_set_pin_sequence.svh
```

Figure 5 – GPIO agent file and folder structure

Table 1 – GPIO agent source files

File name	Description
gpio_agent_if.sv	GPIO agent interface definition
gpio_agent_pkg.sv	GPIO agent package with all data types, imports, includes, variables and tasks/functions
gpio_item.svh	GPIO agent sequence item. All sequences are built using this class
gpio_base_sequence.svh	GPIO agent base sequence class. All sequences are extended from this class
gpio_set_pin_sequence.svh	GPIO agent sequence used to write user values into DUT input pins
gpio_get_pin_sequence.svh	GPIO agent sequence used to read DUT I/O pin values
gpio_agent_cfg.svh	GPIO agent configuration object
gpio_agent.svh	GPIO agent component top class definition
gpio_driver.svh	GPIO agent active component used for reading/writing DUT pins
gpio_monitor.svh	GPIO agent passive component used for recording pin’s status on each clock cycle and other functionality checks

2.3 Package

The GPIO agent package contains all information relevant to the proper use of the VIP. It contains variables, data types, user pin specifications, initial values for the DUT input signals and all functions/tasks that should be employed by the user in order to properly read/write pin values.

The package source code is divided into several sections. The most important ones for the user are the ***User*** section and the ***Functions/Tasks*** section. Only the former should be modified by the user.

The ***User section*** is used to specify the names of the I/O pins that the GPIO agent will be driving and the initial, or reset, values of the DUT input pins (GPIO agent output pins). An example of these specifications is shown in code samples 1 and 2. The reason the user should specify the pin names like this is, that, without them the use of the agent in the UVM tests would be very unclear. For example, driving a DUT signal from the testbench using the pin position would be done as shown in code snippet 3 (left). Observe the *`pin_name_o`* parameter of the *`setPin`* task. In this example the user is setting the RESET pin of the DUT synchronously to logic high. But, we are specifying the desired pin using numbers, which is hard to use and vague, as it would require remembering the number corresponding to every pin, or always looking it up prior to use. However, when the user specifies pin names, as in code snippet 3 (right), the situation becomes much clearer and someone not familiar with the code can properly decipher what is happening. The need to specify initial DUT input pin values is self explanatory.

```
// agent input pins list
typedef enum {
  READY,
  IRQ
} gpio_input_t;
```

```
// agent output pins list
typedef enum {
  RST,
  ADDR_SPACE_1,
  ADDR_SPACE_0,
  LAST
} gpio_output_t;
```

Code sample 1 – Setting the GPIO agent I/O pin list

```
// set the initial values of the GPIO output pins
logic [W_OUT-1:0] gpio_out_init = {
  1'b0, // LAST
  1'b0, // ADDR_SPACE_0
  1'b0, // ADDR_SPACE_1
  1'b1  // RST
};
```

Code sample 2 – Initial GPIO agent output pin values

```
setPin(
  ._sqcr       (env.gpio_agent.sqcr),
  ._op_type    (WR_SYNC),
  ._pin_name_o('{0}), // e.g. RESET pin
  ._wr_data    ('{1'b1})
);
```

```
setPin(
  ._sqcr       (env.gpio_agent.sqcr),
  ._op_type    (WR_SYNC),
  ._pin_name_o('{RESET}),
  ._wr_data    ('{1'b1})
);
```

Code sample 3 – Driving pins using position vs. using names

The ***Functions/Tasks*** is where all user API facilities are implemented. There are also some helper functions, but they are of little value to the end user. The user should be restricted to using the API shown in Table 1 to control the DUT.

Table 2 – List of GPIO agent user API with description

Task	Description

`function automatic GpioAgentCfg configGpioAgent(`
`  input uvm_active_passive_enum _is_active= UVM_PASSIVE,`
`  input bit                      _is_x_z_check = 1'b0`
`);`
	Function used to easily create a GPIO agent configuration object with user defined settings. Function parameters are as follows:
	1. agent can be active or passive, i.e. **UVM_ACTIVE** or **UVM_PASSIVE**
	2. enable/disable checking of logic 'X' and 'Z' values by the agent
	
	Function returns the specified GPIO agent configuration object.

`function automatic void setXZCheck(`
`  input GpioAgent _agt,`
`  input bit       _is_x_z_check`
`);`
	Function used to enable/disable checking for unknown and high impedance values on its interface, during run time.

`task automatic setPin(`
`  input  bit              _print_info    = 1'b1,`
`  input  uvm_sequencer_base _sqcr,`
`  input  op_type_t        _op_type,`
`  input  gpio_output_t _pin_name_o [],`
`  input  logic            _wr_data    [],`
`  input  bit [31:0]       _delay        = 0`
`);`
	Used to drive one or more GPIO interface output pins (DUT input pins). Task parameters are as follows:
	1. print info messages
	2. path to the GPIO agent sequencer on which the setPin task is started
	3. operation type; **should only use WR_SYNC or WR_ASYNC**
	4. array of GPIO agent output pins which will be driven
	5. array of values that the specified output pins will take
	6. delay before the new values take effect; **ignored for WR_SYNC and WR_ASYNC**

```
task automatic setPinWindow(
  input  bit           _print_info       = 1'b1,
  input  uvm_sequencer_base _sqcr,
  input  op_type_t      _op_type,
  input  gpio_output_t _pin_name_o [],
  input  logic         _wr_data_start [],
  input  logic         _wr_data_end   [],
  input  bit [31:0] _delay             = 0,
  input  bit [31:0] _duration          = 0
);
```
	Used to drive one or more GPIO interface output pins (DUT input pins), with an initial delay before the starting values are applied, and a duration after which the ending values are applied. Task parameters are as follows:
	1. print info messages
	2. path to the GPIO agent sequencer on which the setPinWindow task is started
	3. operation type; **can only use WR_WIN_SYNC or WR_WIN_ASYNC**
	4. array of GPIO agent output pins which will be driven
	5. array of values that the specified output pins will take, after the stated delay
	6. array of values that the specified output pins will take, after the stated duration
	7. delay before the start values take effect
	8. time after start values are applied, until end values take effect
	
	Synchronous delay is in clock cycles, while asynchronous is in nanoseconds.

```
task automatic getPin(
  input  bit              _print_info    = 1'b1,
  input  uvm_sequencer_base _sqcr,
  input  op_type_t        _op_type,
  input  gpio_input_t     _pin_name_i [] = {},
  input  gpio_output_t    _pin_name_o [] = {},
  output logic            _rd_data    []
);
```
	Used for reading one or more GPIO interface I/O pins (DUT I/O pins). Task parameters are as follows:
	1. print info messages
	2. path to the GPIO agent sequencer on which the getPin task is started
	3. operation type; **can only use RD_SYNC or RD_ASYNC**
	4. array of GPIO agent input pins which will be read
	5. array of GPIO agent output pins which will be read
	6. array of read values from specified pins
	
	The array that will be put in the _rd_data parameter should be specified beforehand, in the test or some other class that uses the getPin task.

`task automatic getCompare(`
`  input  bit                _print_info = 1'b1,`
`  input  uvm_sequencer_base _sqcr,`
`  input  op_type_t          _op_type,`
`  input  gpio_input_t       _pin_name_i  [] = {},`
`  input  gpio_output_t      _pin_name_o  [] = {},`
`  input  logic              _user_data_i [] = {},`
`  input  logic              _user_data_o [] = {}`
`);`
	Used for reading one or more GPIO interface I/O pins (DUT I/O pins) and comparing them to user specified values. Task parameters are as follows:
	1. print info messages
	2. path to the GPIO agent sequencer on which the getPin task is started
	3. operation type; **can only use RD_SYNC or RD_ASYNC**
	4. array of GPIO agent input pins which will be read
	5. array of GPIO agent output pins which will be read
	6. expected user input pin values
	7. expected user output pin values

These tasks form the user API for communicating with the GPIO agent. They perform the “heavy lifting” like instantiating sequences, randomization and starting them on a sequencer, while the user does not have to think about those implementation details. Instead, he can focus on writing tests and verification tasks that validate the desired functionality.

2.4 Interface

The GPIO agent interface is very simple. It consists of only two vector signals, *gpio_in* and *gpio_out*. Currently, they are fixed to a size of 1024, with each bit corresponding to an I/O pin of the DUT that the user wishes to control.

The interface also contains two clocking blocks and two modports, which are shown in Table 2. Depending on which component uses them, their direction specification changes. It is not possible to specify the slave clocking block and modport, because that would mean that the DUT signals, which are not part of any protocol would have to be known in advance, which is not always possible, or is, but only late in the project. If it is possible, then it would not be practical as the interface would have to be changed and adapted for each individual DUT.

Table 3 – Signals, clocking blocks and modports of the GPIO interface

Signals	Type	Description
`gpio_in`	`wire  [1023:0]`	GPIO agent inputs == DUT outputs
`gpio_out`	`logic [1023:0]`	GPIO agent outputs == DUT inputs
Clocking blocks		**Description**
`cb_master`		Clocking block for an active component, such as a driver; *`gpio_in`* is set as input and *`gpio_out`* is set as output.
`cb_monitor`		Clocking block for a passive component, such as a monitor; *`gpio_in`* and *`gpio_out`* are set as inputs.
Modports		**Description**
`mp_master`		Master component modport
`mp_monitor`		Monitor component modport

The procedure to connect the interface to the DUT in the top level module, will be shown in chapter 3, *Instructions and Usage*.

2.5 Configuration

The configuration of the GPIO agent is mandatory, before attempting its use. The reason for this is that the configuration contains vital information on how the agent is supposed to be built. Also, it contains additional fields, which should be set to ensure that the proper instance of the GPIO agent is being used. The configuration fields, with their descriptions, are shown in Table 4.

Table 4 – GPIO agent configuration fields

Field	Type	Description
`is_active`	`uvm_active_passive_enum`	The GPIO agent can be active or passive. If the agent is active, then it can control DUT signals. If it is passive, it can only monitor pins, pack them into transactions and send them through the agents analysis port. Possible values are: • UVM_ACTIVE (default) • UVM_PASSIVE
`_is_x_z_check`	`bit`	The GPIO agent can be setup to check for unknown or high impedance values on its interface, and report warnings if they are detected. This field enables/disables this check.
`vif`	`virtual GpioIf`	This is the virtual reference to the top testbench GPIO interface instance. It should be grabbed from the configuration database and set here, so that the agent can have access to the DUT pins.

The basic principle of use is to create the configuration object (usually using the `configGpioAgent` function) in the UVM test and set the above fields. Then, the GPIO interface needs to be grabbed from the configuration database and stored into the configuration object. A detailed example of this procedure will be presented in chapter 3.1.2.

The GPIO agent will first try to grab its configuration, identified by the string “`gpio_agent_cfg`”, from the configuration database. If it does not succeed, it will produce a fatal error. If it finds the configuration it will then check if its virtual interface field is populated. If not, the agent will again produce a fatal error. Finally, if the virtual interface has been properly set in the found configuration object, the GPIO agent can proceed to build itself and its sub-components, because it has all the necessary information.

3 Instructions and usage

This chapter will present the step-by-step instructions on how to prepare and instantiate the GPIO agent in a UVM environment, using a real-world example. Then, it will show examples of using the API from Table 2 and the corresponding waveforms.

3.1 Agent instantiation

This chapter describes how to integrate the GPIO agent in the user's UVM verification environment. The first sub-chapter will provide the short outline, which can be used as a quick reference, while the second sub-chapter will provide the step-by-step instructions.

3.1.1 Quick summary

Here is a quick overview of the steps needed to be performed in order for the GPIO agent and its facilities to become available to the user:

1. Fill in the input and output pin lists of the GPIO agent (gpio_agent_pkg.sv)
2. Specify the initial values of the GPIO output pins (gpio_agent_pkg.sv)
3. Instantiate the GPIO interface in the top SystemVerilog testbench module and connect it to the desired DUT pins
4. Put the GPIO interface object in the configuration database
5. Create the GPIO configuration object, assign the GPIO interface to the GPIO configuration object and make the configuration available to the GPIO agent by setting it in the configuration database
6. Import the GPIO agent package into the user UVM environment

3.1.2 Detailed steps

Let’s take an example design that we wish to verify with a UVM testbench, e.g, a SHA3 core shown in Figure 6. The interface of the DUT can be separated into a couple of logical groups of signals. The first group could be the standard clock and reset signals, the second the SPI interface, used for reading and writing data, and the last group could be the arbitrary control signals.

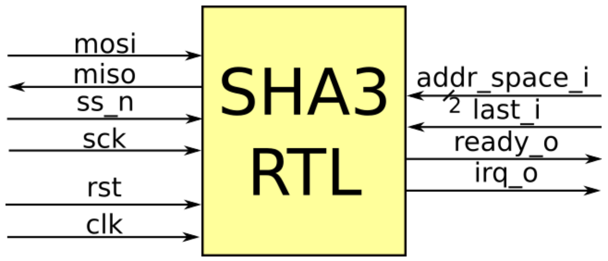


Figure 6 – Example interface of an SHA3 design

After some examination the user decides to use a clock generator to drive the **`clk`** signal, an SPI agent for the **`sck`**, **`mosi`**, **`miso`** and **`ss_n`** signals and this GPIO agent to handle the **`rst`**, **`addr_space_i`**, **`last_i`**, **`ready_o`** and **`irq_o`**. This connection setup to the DUT would correspond to the one described in chapter 2.1, and depicted in Figure 2, where only unused pins are connected to the GPIO agent interface. Figure 7 shows the structure of the verification environment after the above mentioned connections are made.

After this thought process, the signals of the DUT which will be controlled by the GPIO agent are known, and the user is ready to perform the necessary steps to include the agent in the verification environment, by using the steps shown in chapter 3.1.1. The first step would be to open the gpio_agent_pkg.sv file, and in the ***User*** section fill in the input and output pin lists of the GPIO agent according to Figure 7. This is already shown in Code sample 1.

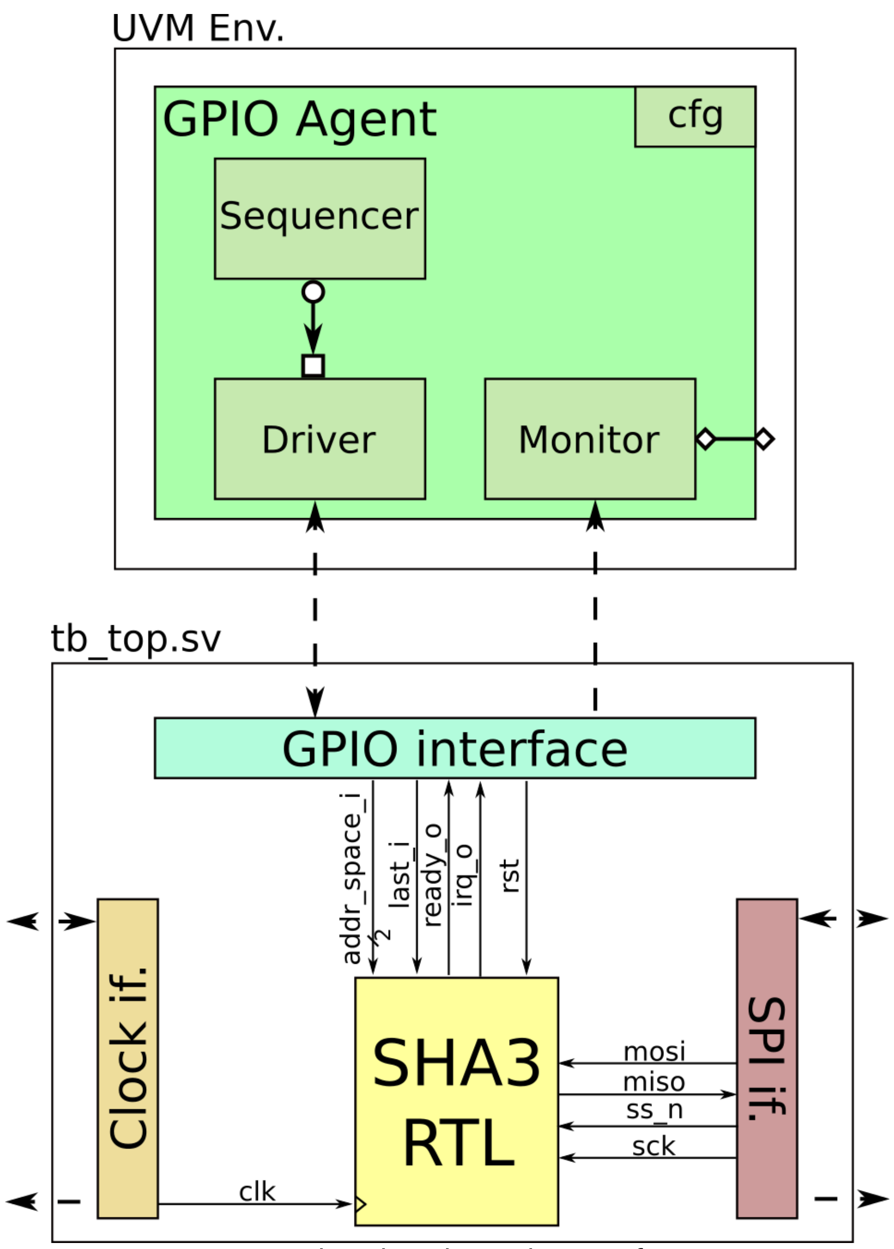


Figure 7 – SHA3 testbench and signal-to-interface partitioning

The second thing needed to be done is to specify the initial values of the GPIO output pins (DUT input pins), which are **`rst`**, **`addr_space_i`** and **`last_i`**, in this case. These values will be applied at the start of the simulation and any time after the reset is de-asserted. The way to achieve this for this example is shown in Code sample 2.

The third step would be to instantiate the GPIO interface object in the top SystemVerilog testbench module and connect all DUT pins to their respective interfaces. An example of this procedure is shown in Code sample 4.

```
// Interfaces-----------------------------------------
GpioIf gpio_if(
  .clk(clk) // any clock can be used
);

// ... additional interfaces

// DUT------------------------------------------------
sha3 dut( //synthesizable SV SPI core
  .clk          (clk),
  .rst          (gpio_if.gpio_out[RST]),
  // SPI Slave Interface
  .mosi         (spi_if.mosi),
  .sck          (spi_if.sck),
  .ss_n         (spi_if.ss_n[0]),
  .miso         (spi_if.miso),
  // Control Interface
  .addr_space_i({gpio_if.gpio_out[ADDR_SPACE_1],
                 gpio_if.gpio_out[ADDR_SPACE_0]}),
  .last_i       (gpio_if.gpio_out[LAST]),
  .ready_o      (gpio_if.gpio_in[READY]),
  .irq_o        (gpio_if.gpio_in[IRQ])
);
```

Code sample 4 – Instantiating a GPIO interface and connecting the DUT

After this, the user should put the GPIO interface object in the configuration database, so that it becomes available to the test, environment, or objects lower in the hierarchy. This procedure is shown in Code sample 5, in which the GPIO interface is made available to the currently running UVM test..

```
initial begin
  // Put the SV interface in the configuration database
  uvm_config_db #(virtual GpioIf)::set(null, "uvm_test_top",
                                       "GPIO_VIF", gpio_if);
  // ... additional interfaces
  run_test();
end
```

Code sample 5 – Making the GPIO interface available to the test

The fifth step assumes grabbing the previously set GPIO agent interface, creating the GPIO configuration object, assigning the GPIO interface to the GPIO configuration object and making the configuration available in to the GPIO agent. This procedure varies highly with the way the user has set-up his testbench and will not be described in detail here. An example of how this could be done in the UVM test is shown in Code sample 6.

```
function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  //create and configure the environment configuration object
  env_cfg.gpio_cfg = configGpioAgent(
    ._is_active   (UVM_ACTIVE),
    ._is_x_z_check(1'b0)
  );

  if (!uvm_config_db #(virtual GpioIf)::get(this, "", "GPIO_VIF",
                                            gpio_agent_cfg.vif)) begin
    `uvm_error("TST", "Couldn't get the GPIO Agent virtual interface")
  end

  uvm_config_db #(GpioAgentCfg)::set(this, "gpio_agent",
                                     "gpio_agent_cfg", gpio_agent_cfg);

  //create the environment
  gpio_agent = GpioAgent::type_id::create("gpio_agent", this);
endfunction: build_phase
```

Code sample 6 – Setting the GPIO agent configuration

The last step to be done is to import the GPIO agent package in all other packages or classes which use the GPIO agent in some way. These packages or classes are usually the ones in the higher levels of the TB hierarchy, i.e. the environment, the test and the top testbench module. Code sample 7 shows an example of importing the GPIO agent package in the environment package.

```
`ifndef _ENVIRONMENT_PKG_
`define _ENVIRONMENT_PKG_

package EnvPkg;

  // Imports
  import uvm_pkg::*;
  import GpioAgentPkg::*;

  // ... other package imports, includes,constants, tasks, etc.

endpackage: EnvPkg

`endif
```

Code sample 7 – Importing the GPIO agent package

After all the above steps have been completed the user can use the facilities that the GPIO agent provides, and which are shown in Table 2, to write tests which will control the signals that are connected to the GPIO interface. Examples of UVM tests using the GPIO agent API, with the corresponding waveforms, will be covered in chapter 3.2.

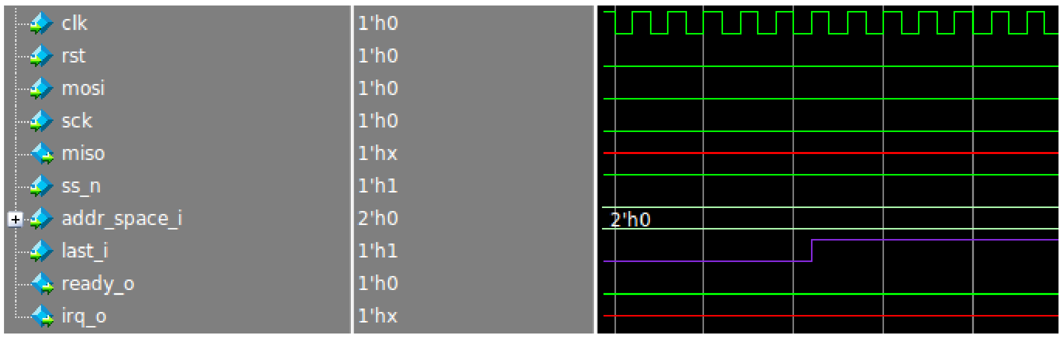
3.2 GPIO agent API and waveforms

This section expands on the previous, by showing how using the facilities provided by the GPIO agent, and shown in Table 3, can be used inside a UVM test. Each version of the task is shown with a corresponding waveform, so that the user can visualize what is happening. These tasks should be used in the UVM run phase.

3.2.1 setPin task

Synchronous single pin drive:

```
setPin(
  ._sqcr      (env.gpio_agent.sqcr),
  ._op_type   (WR_SYNC),
  ._pin_name_o('{LAST}),
  ._wr_data   ('{1'b1})
);
```



clk	1'h0
rst	1'h0
mosi	1'h0
sck	1'h0
miso	1'hx
ss_n	1'h1
addr_space_i	2'h0
last_i	1'h1
ready_o	1'h0
irq_o	1'hx

Figure 8 – Synchronous single pin drive waveform

Synchronous multi pin drive:

```
setPin(
  ._sqcr      (env.gpio_agent.sqcr),
  ._op_type   (WR_SYNC),
  ._pin_name_o('{ADDR_SPACE_1, ADDR_SPACE_0, RST, LAST}),
  ._wr_data   ('{1'b1, 1'b0, 1'b1, 1'b1})
);
```

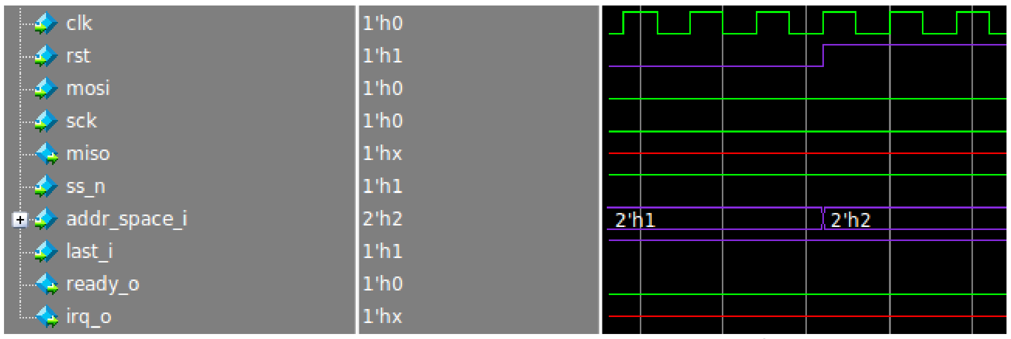


Figure 9 – Synchronous multi pin drive waveform

Asynchronous single pin drive:

```
          setPin(
            ._sqcr      (env.gpio_agent.sqcr),
            ._op_type   (WR_ASYNC),
            ._pin_name_o('{ADDR_SPACE_1}),
            ._wr_data   ('{1'b0})
          );
```

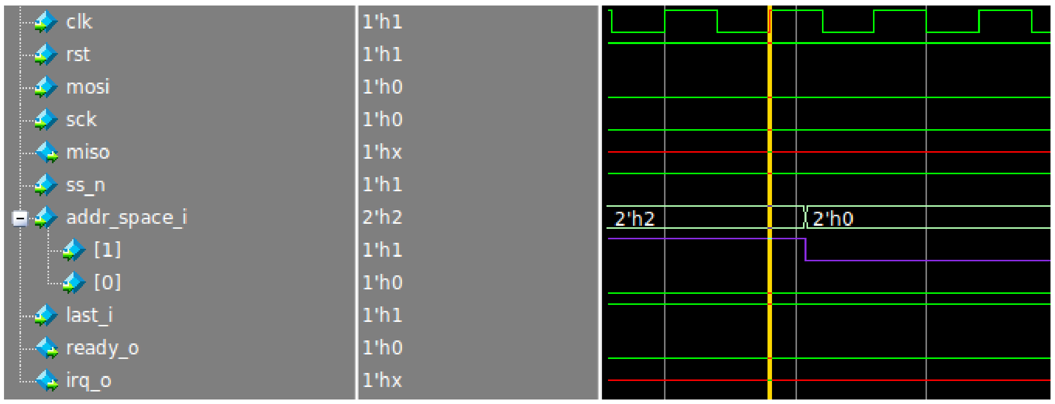


Figure 10 – Asynchronous single pin drive waveform

Asynchronous multi pin drive:

```
          setPin(
            ._sqcr      (env.gpio_agent.sqcr),
            ._op_type   (WR_ASYNC),
            ._pin_name_o('{RST, LAST, ADDR_SPACE_0}),
            ._wr_data   ('{1'b0, 1'b0, 1'b1})
          );
```

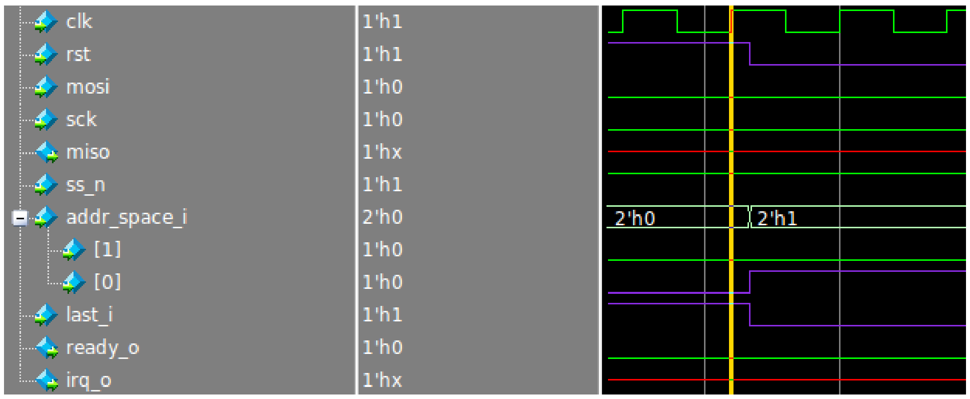


Figure 11 – Asynchronous multi pin drive waveform

3.2.2 setWindow task

Synchronous single pin window, without delay and duration specification:

When used without the delay and duration parameter, the synchronous **`setPinWindow`** task will assert the starting pin value on the next immediate rising clock edge, and assert the ending pin value on the following rising clock edge.

```
        setPinWindow(
          ._sqcr         (env.gpio_agent.sqcr),
          ._op_type      (WR_WIN_SYNC),
          ._pin_name_o   ('{LAST}),
          ._wr_data_start('{1'b1}),
          ._wr_data_end  ('{1'b0})
        );
```

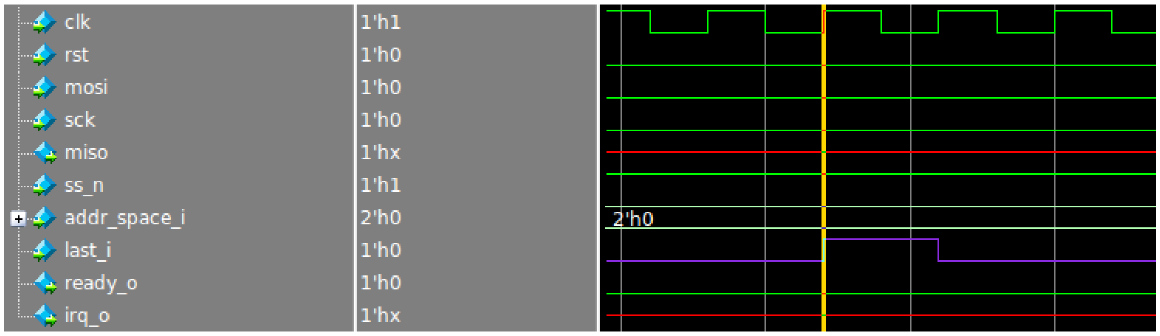


Figure 12 – Synchronous single pin window drive waveform

Synchronous multi pin window, delay and duration specified:

When the synchronous **`setPinWindow`** task is used the delay and duration fields represent the number of clock cycles before the starting values are asserted and the number of clock cycles after the first assertion that the ending values are applied.

```
        setPinWindow(
          ._sqcr          (env.gpio_agent.sqcr),
          ._op_type       (WR_WIN_SYNC),
          ._pin_name_o    ('{ADDR_SPACE_1, ADDR_SPACE_0, RST, LAST}),
          ._wr_data_start('{1'b1, 1'b0, 1'b1, 1'b1}),
          ._wr_data_end   ('{1'b0, 1'b1, 1'b0, 1'b0}),
          ._delay         (10), // clock cycles
          ._duration      (33)  // clock cycles
        );
```

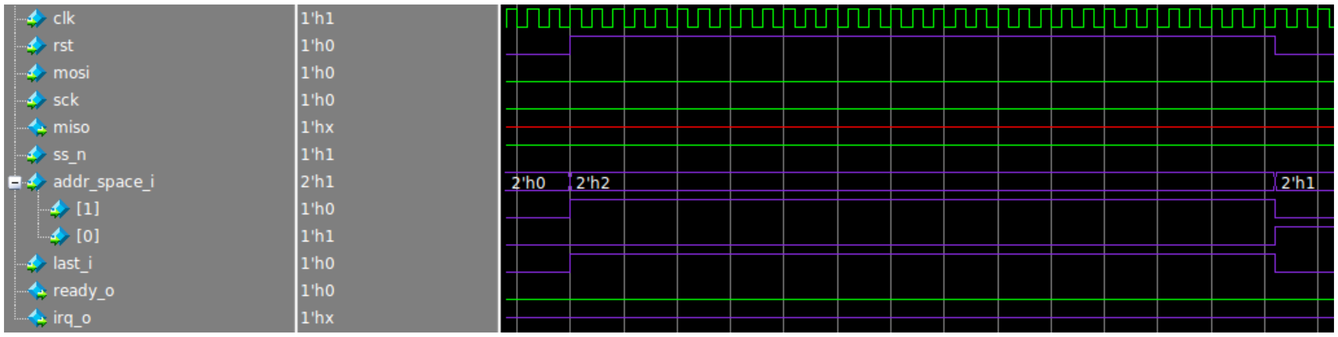


Figure 13 – Synchronous multi pin window drive waveform

Asynchronous multi pin window, duration specified:

When the asynchronous **`setPinWindow`** task is used the delay and duration fields represent the number of nanoseconds before the starting values are asserted and the number of nanoseconds after the first assertion that the ending values are applied.

```
        setPinWindow(
          ._sqcr          (env.gpio_agent.sqcr),
          ._op_type       (WR_WIN_ASYNC),
          ._pin_name_o    ('{ADDR_SPACE_1, ADDR_SPACE_0, RST, LAST}),
          ._wr_data_start('{1'b1, 1'b0, 1'b1, 1'b1}),
          ._wr_data_end   ('{1'b0, 1'b1, 1'b0, 1'b0}),
          ._duration      (617)  // nanoseconds
        );
```

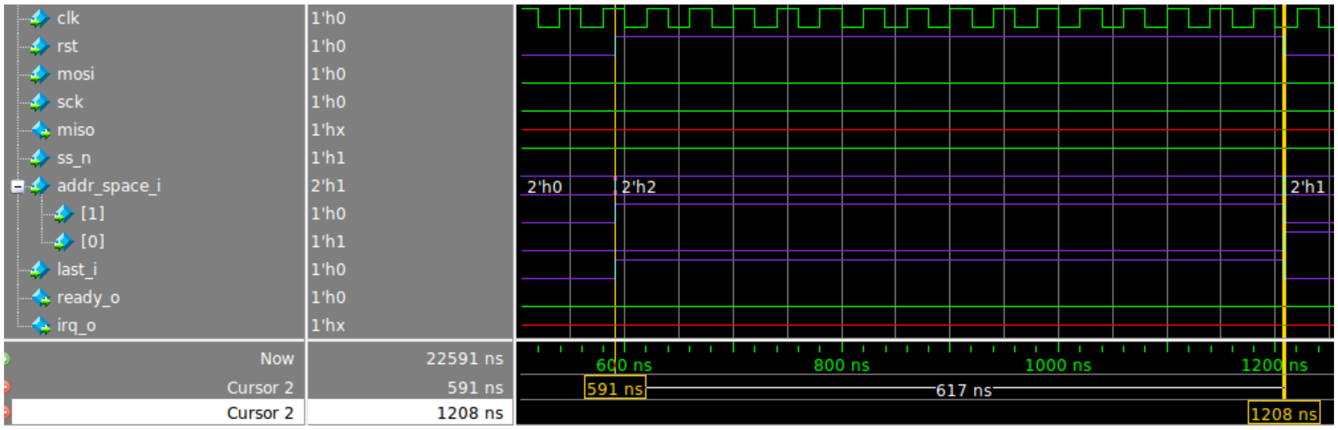


Figure 14 – Asynchronous multi pin window drive waveform

3.2.3 getPin task

Synchronous single pin:

```
    getPin(
      ._sqcr      (env.gpio_agent.sqcr),
      ._op_type   (RD_SYNC),
      ._pin_name_i('{READY}),
      ._pin_name_o('{ADDR_SPACE_0}),
      ._rd_data   (rd_data)
    );
```

```
# UVM_INFO .../gpio_agent_pkg.sv(445) @ 7580: reporter [GPIO_PKG]
# GPIO Get OP:
# ---------------------------------------------------
# OP Type     : RD_SYNC
# Pin Name(s) : READY, ADDR_SPACE_0
# Pin Num(s)  : 0, 2
# Value(s)    : 1'b0, 1'b1
```

Synchronous multi pin:

```
    getPin(
      ._sqcr      (env.gpio_agent.sqcr),
      ._op_type   (RD_SYNC),
      ._pin_name_i('{IRQ, READY}),
      ._pin_name_o('{ADDR_SPACE_1, ADDR_SPACE_0, RST, LAST}),
      ._rd_data   (rd_data)
    );
```

```
# UVM_INFO .../gpio_agent_pkg.sv(445) @ 7620: reporter [GPIO_PKG]
# GPIO Get OP:
# ---------------------------------------------------
# OP Type     : RD_SYNC
# Pin Name(s) : IRQ, READY, ADDR_SPACE_1, ADDR_SPACE_0, RST, LAST
# Pin Num(s)  : 1, 0, 1, 2, 0, 3
# Value(s)    : 1'bX, 1'b0, 1'b0, 1'b1, 1'b0, 1'b0
```

Asynchronous multi pin:

Notice that the asynchronous read operation was performed immediately, without waiting for the next rising clock edge.

```
    getPin(
      ._sqcr      (env.gpio_agent.sqcr),
      ._op_type   (RD_ASYNC),
      ._pin_name_i('{IRQ, READY}),
      ._pin_name_o('{ADDR_SPACE_1, ADDR_SPACE_0, RST, LAST}),
      ._rd_data   (rd_data)
    );
```

```
# UVM_INFO .../gpio_agent_pkg.sv(445) @ 7631: reporter [GPIO_PKG]
# GPIO Get OP:
# ----------------------------------------------------
# OP Type     : RD_ASYNC
# Pin Name(s) : IRQ, READY, ADDR_SPACE_1, ADDR_SPACE_0, RST, LAST
# Pin Num(s)  : 1, 0, 1, 2, 0, 3
# Value(s)    : 1'bX, 1'b0, 1'b0, 1'b1, 1'b0, 1'b0
```

3.2.4 getCompare task

Synchronous multi pin, no error:

```
    getCompare(
      ._sqcr       (env.gpio_agent.sqcr),
      ._op_type    (RD_SYNC),
      ._pin_name_i ('{READY}),
      ._pin_name_o ('{ADDR_SPACE_1, LAST}),
      ._user_data_i('{0}),
      ._user_data_o('{0, 0})
    );
```

```
# UVM_INFO .../gpio_agent_pkg.sv(520) @ 7700: reporter [GPIO_PKG]
# GPIO Compare OP:
# ----------------------------------------------------
# OP Type       : RD_SYNC
# Pin Name(s)   : READY, ADDR_SPACE_1, LAST
# Pin Num(s)    : 0, 1, 3
# User Value(s) : 1'b0, 1'b0, 1'b0
# Read Value(s) : 1'b0, 1'b0, 1'b0
# Status        : PASS
```

Synchronous multi pin, with error:

```
    getCompare(
      ._sqcr       (env.gpio_agent.sqcr),
      ._op_type    (RD_SYNC),
      ._pin_name_i ('{READY}),
      ._pin_name_o ('{ADDR_SPACE_1, LAST}),
      ._user_data_i('{0}),
      ._user_data_o('{0, 1})
    );
```

```
# UVM_ERROR .../gpio_agent_pkg.sv(534) @ 77800: reporter [GPIO_PKG]
# GPIO Compare OP:
# ------------------------------------------------
# OP Type        : RD_SYNC
# Pin Name(s)    : READY, ADDR_SPACE_1, LAST
# Pin Num(s)     : 0, 1, 3
# User Value(s)  : 1'b0, 1'b0, 1'b1
# Read Value(s)  : 1'b0, 1'b0, 1'b0
# Status         : FAIL
```

Asynchronous multi pin:

```
    getCompare(
      ._sqcr        (env.gpio_agent.sqcr),
      ._op_type     (RD_ASYNC),
      ._pin_name_i ('{IRQ, READY}),
      ._pin_name_o ('{ADDR_SPACE_1, LAST}),
      ._user_data_i('{1'bX, 0}),
      ._user_data_o('{0, 0})
    );
```

```
# UVM_INFO .../gpio_agent_pkg.sv(520) @ 77817: reporter [GPIO_PKG]
# GPIO Compare OP:
# ------------------------------------------------
# OP Type        : RD_ASYNC
# Pin Name(s)    : IRQ, READY, ADDR_SPACE_1, LAST
# Pin Num(s)     : 1, 0, 1, 3
# User Value(s)  : 1'bX, 1'b0, 1'b0, 1'b0
# Read Value(s)  : 1'bX, 1'b0, 1'b0, 1'b0
# Status         : PASS
```

3.3 SPI protocol example

As mentioned in the introductory chapter, the GPIO agent API can be used to form more complicated protocol API's, such as SPI, I2C, AXI, etc. This can be achieved by using the tasks mentioned in chapter , and encapsulating them in higher level tasks, which will become the new protocol. An example of achieving this for an SPI protocol is presented in this chapter.

TBD

4 Compatibility and issues

The GPIO agent has been tested and built with the tools shown in Table 5. Known issues are also shown with their resolution.

Table 5 – Tools with which the GPIO agent has been tested

Tool	Version	Status
Mentor Questa	10.5	🟩
Cadence IES	15.20-s024	🟩
Aldec Riviera Pro	2015.06	🟥

4.1 Mentor Questa

No issues found.

4.2 Cadence IES

<u>Issue 1 – Severity : minor</u>

The tested version of Cadence IES does not support invocation of built-in methods in constant expressions, so it will produce the following error:

```
file: gpio_agent_pkg.sv
parameter W_IN  = gpio_input_list.num();
                                       |
ncvlog: *E,BIMNCN (gpio_agent_pkg.sv,46|38): Invocation of built-in
          methods in constant
expressions is not yet supported. [SystemVerilog].
parameter W_OUT = gpio_output_list.num();
                                        |
ncvlog: *E,BIMNCN (gpio_agent_pkg.sv,47|39): Invocation of built-in
          methods in constant
expressions is not yet supported. [SystemVerilog].
   package worklib.GpioAgentPkg:sv
        errors: 2, warnings: 0
```

Resolution: The user has to change the following lines in the gpio_agent_pkg.sv file

```
parameter W_IN  = gpio_input_list.num();
parameter W_OUT = gpio_output_list.num();
```

, and manually set the number of input and output signals which will be controlled by the GPIO agent. For the input and output pins used in Code sample 1 the changes would look as following:

```
parameter W_IN  = 2;
parameter W_OUT = 4;
```

4.3 Aldec Riviera Pro

Issue 1 – Severity : major

The GPIO agent builds fine, but randomization fails at run time.

TBD