

Assignment

November 2025
SyoSil ApS ©

Contents

1	Introduction	3
2	Design of the Memory Arbiter	4
2.1	APB Control	5
2.2	Priority Selector	7
2.2.1	Sorting State Machine	7
2.2.2	Priority Switch Logic	8
3	SyoSil Data Transfer Protocol	10
4	Memory Arbiter Testbench	11
5	Assignment	13
5.1	A1. Development of a Verification Plan	13
5.2	A2. Integration of <i>uVC</i> s in the testbench	14
5.2.1	A2.1 Implementation of the <i>SDT uVC</i> 's driver	14
5.2.2	A2.2 Integration of <i>uVC</i> s in the testbench	14
5.2.3	A2.3 Definition of the <i>SDT uVC</i> 's configuration object	15
5.2.4	A2.3 <i>Optional</i> exercise	15
5.3	A3. Implementation of the testbench configuration object	15
5.4	A4. Implementation of test cases, including sequences and virtual sequences . .	15
5.4.1	A4.1 Random traffic test case with static priority	16
5.4.2	A4.2 Random traffic test case with dynamic priority	16
5.5	A5 (<i>Optional</i>) Implementation and integration of the Reference model	16
5.6	A6 (<i>Optional</i>) Implementation and integration of the Scoreboard component . .	16
5.7	A7 (<i>Optional</i>) Implementation of checkers for the <i>SDT</i> protocol	16
5.8	A8 (<i>Optional</i>) Implementation of a coverage collector for the <i>Memory Arbiter</i> and generation of coverage reports	17
5.8.1	A8.1 (<i>Optional</i>) Coverage scenario: A request of the three CIFs is made in parallel	18
5.9	A9 (<i>Optional</i>) Implementation of checker for the <i>Memory Arbiter</i>	18

1 Introduction

The following document will provide a description of a *Memory Arbiter* (MARB) design and a data transfer protocol, named SyoSil Data Transfer Protocol (*SDT*).

An assignment is also provided in order to create a fully functioning *PyUVM* testbench to verify the correct behavior of the *Memory Arbiter* IP provided. The assignment will be evaluated based on the correctness and completeness of the work.

2 Design of the Memory Arbiter

A *Memory Arbiter* is a device used in a shared memory system to determine which client will be allowed to access the shared memory, ensuring proper coordination and preventing conflicts.

Figure 2.1, shows a high level diagram of the provided *Memory Arbiter* IP. The IP has the following external interfaces/signals:

- Clock
- Reset (asynchronous)
- Client interface #1 (CIF1 - SDT protocol)
- Client interface #2 (CIF2 - SDT protocol)
- Client interface #3 (CIF3 - SDT protocol)
- Advanced Peripheral Bus (APB protocol)
- Memory interface (MIF - SDT protocol)

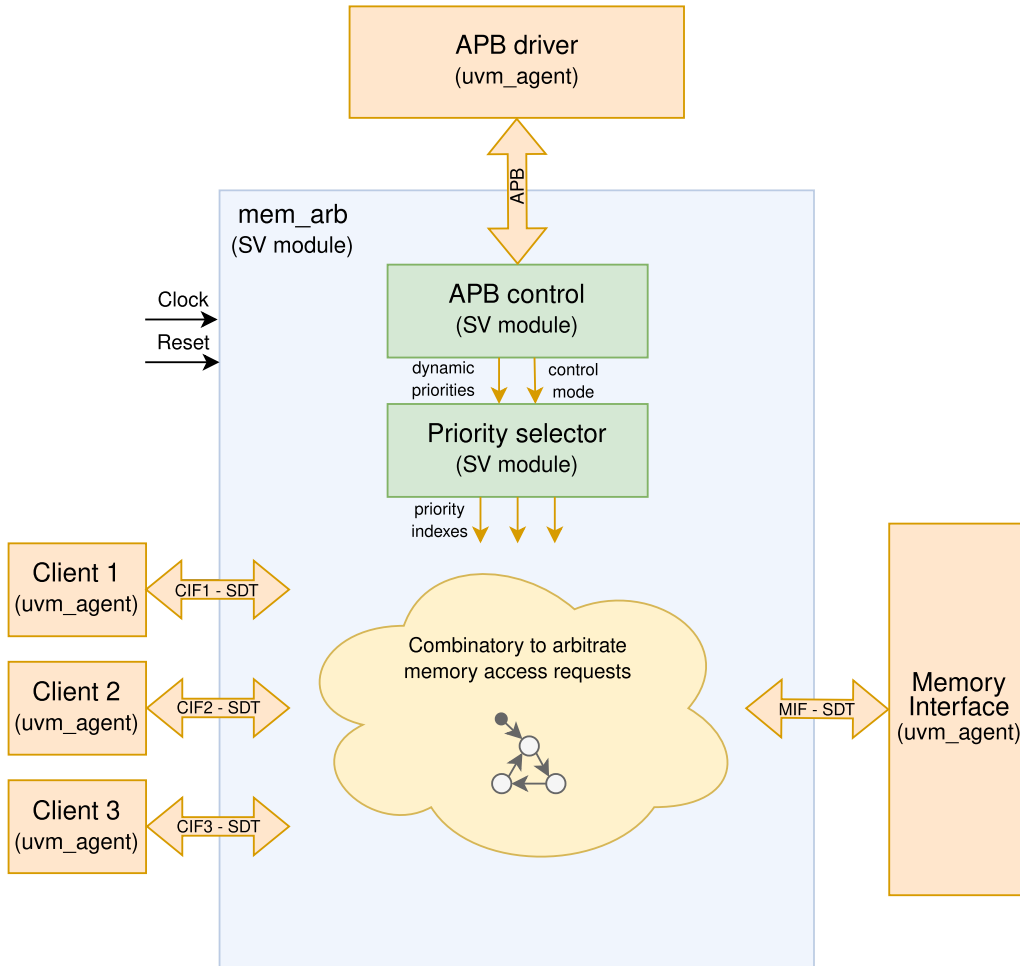


Fig. 2.1: High level diagram of the Memory Arbiter design.

As can be seen in Figure 2.1, the device uses two data transfer protocols. The SyoSil Data Transfer Protocol (*SDT*) is used by the external clients to make requests to access the memory and by the *Memory Arbiter* to access the memory. The Advanced Peripheral Bus (APB) is used to configure the device by writing its registers.

The core logic of the *Memory Arbiter* arbitrates between the three client interfaces (CIFs) to access the memory for **read** and **write** operations and always selects the one with the highest priority to be served. The IP always checks for the latest priority set. If the new value has higher priority, then the IP changes the order of the client requests as long as there is no client being served at that moment. When a request has been selected the *Memory Arbiter* drives the MIF with the request. Then, the MIF enables the **ack** signal to close the handshake.

The *Memory Arbiter* contains two registers that allows to manage the arbitration of the access from the clients: **Control Register** and **Dynamic Priority Register** and they are physically located in the submodule **APB Control**.

2.1 APB Control

This module is designed to interface with an APB driver through an APB bus and **write** and **read** registers.

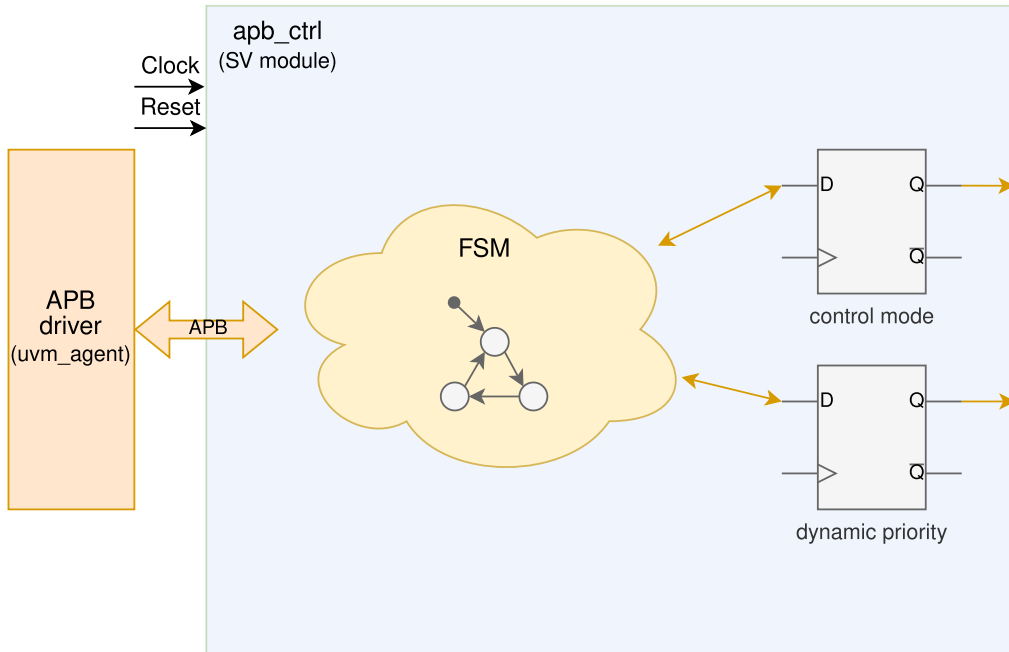


Fig. 2.2: Diagram of the APB Control module.

Fig 2.2 shows the block diagram of the APB Control module. When a **write** or **read** request arrives from the interface, the APB control module processes this request acting on the internal registers.

The APB (Advanced Peripheral Bus) is a part of the AMBA (Advanced Microcontroller Bus Architecture) protocol family, designed for low-power and low-bandwidth communication with peripheral devices. An APB slave is a peripheral device that responds to read and write operations initiated by an APB master. The APB slave performs specific functions such as reading from or writing to registers, and it communicates the status of these operations back to the master.

The APB control module contains two registers that allow the user to configure the dynamic priority of the CIFs and the arbitration mode.

The **control** register is used to manage the arbitration mode. The field **enable** is used to

enable or disable the arbitration. If the arbitration mode is enabled, then the field `mode` can be used to change the arbitration mode between `static` (`mode == 0`) or `dynamic` (`mode == 1`) prioritization of the CIFs. The `dynamic` prioritization of the CIFs is set by the `Dynamic Priority` register, that contains a field for each CIF (the field for Client 4 is unused by the DUT), where the priority value can be set.

Register	Address	Fields			
Dynamic Priority	0x04	Client 4 [31:24]	Client 3 [23:16]	Client 2 [15:8]	Client 1 [7:0]
Control mode	0x00		Unused [31:3]	Mode [2:1]	Enable [0]

Table 2.1: Structure of the APB registers.

As shown in Table 2.1, the `Dynamic priority reg` contains a field for each CIF, where the priority value can be set. Each field is 8 bits wide, allowing for 256 priority levels. The higher the value, the higher the priority the client has. The default value is set to 0. The `Dynamic priority reg` may only be changed while the *Memory Arbiter* is disabled e.g. `enable` is set to 0. After changing the value of the `Dynamic priority reg` the *Memory Arbiter* has to update its internal priority. This takes 6 clock cycles, which has to be accounted for when changing priorities.

The APB interface signals are as follows:

Signal name	Producer direction	Consumer direction	Comment
<code>conf_wr</code>	output	input	This signal indicates a write operation when asserted.
<code>conf_sel</code>	output	input	This signal selects the peripheral for the current transaction.
<code>conf_enable</code>	output	input	This signal is intended to enable the transaction. (Unused in this design).
<code>conf_addr</code>	output	input	This 32-bit signal specifies the address for the current transaction.
<code>conf_wdata</code>	output	input	This 32-bit signal carries the data to be written during a write transaction.
<code>conf_strb</code>	output	input	This 4-bit signal is the write strobe, indicating which bytes of <code>conf_wdata</code> are valid.
<code>conf_rdata</code>	input	output	This 32-bit signal carries the data read from the peripheral during a read transaction.
<code>conf_ready</code>	input	output	This signal indicates that the peripheral is ready to complete the current transaction.
<code>conf_slverr</code>	input	output	This signal indicates that the peripheral is ready to complete the current transaction.

Table 2.2: Signals of APB interface.

2.2 Priority Selector

The **Priority Selector** module is designed to select the client priorities based on the **Control Mode** and the **Dynamic Priority** signals. To this aim, it instantiates and controls a cascade of single sorter modules to sort the clients based on their priorities.

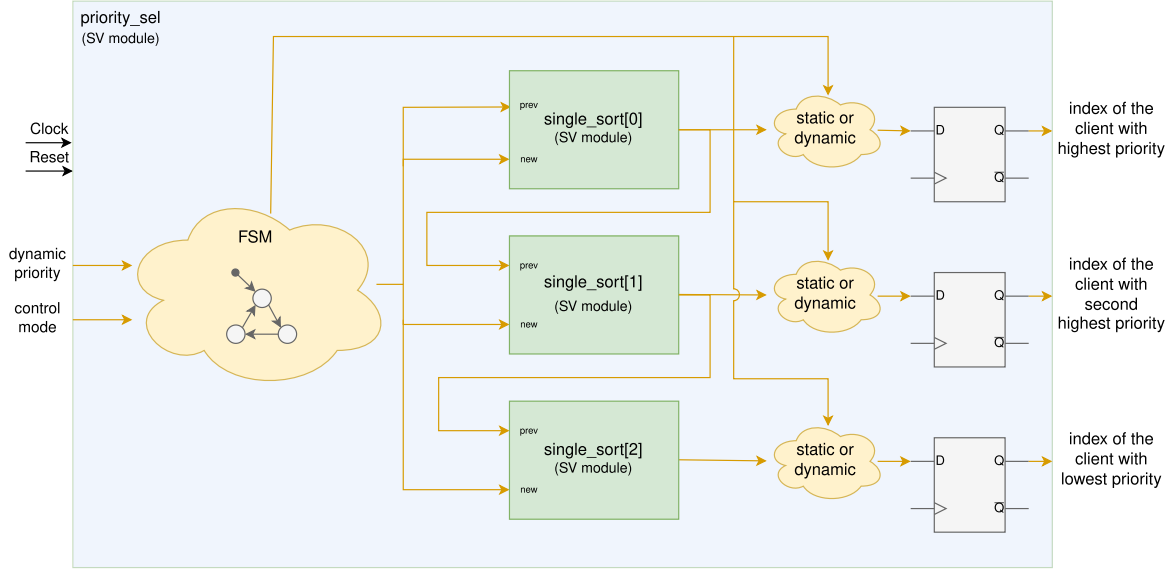


Fig. 2.3: Diagram of the Priority Selector design.

Fig 2.3 shows the architecture of the `priority_sel` module. It supports both static and dynamic priority selection based on the **Control Mode** signal. In static mode, the priority list is fixed, and the priority order is:

- CIF#1 before CIF#2
- CIF#1 before CIF#3
- CIF#2 before CIF#3

In dynamic mode, the priority list is updated based on the **Dynamic Priority** signal values when the sorting state machine is idle. The sorting algorithm takes as many clock cycles as the number of instantiated clients to complete the sorting process.

2.2.1 Sorting State Machine

As shown in Fig 2.4, the sorting state machine has three states: **SORTER_IDLE**, **SORTER_CLEAR**, and **SORTER_SORT**.

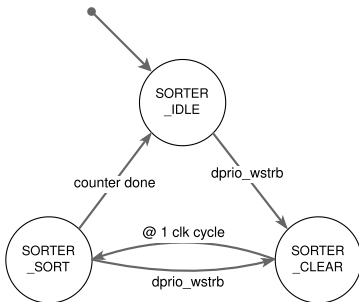


Fig. 2.4: Diagram of the Priority Selector FSM.

- In **SORTER_IDLE**, the sorters hold their values and wait for the next sorting cycle.
- In **SORTER_CLEAR**, the sorters are cleared from previous content to prepare for a new sorting operation.
- In **SORTER_SORT**, the sorting process is enabled and the clients are sorted based on their priorities. Using this state, the sorting algorithm iterates through the clients, comparing and swapping their priorities as needed until the list is fully sorted.

The state machine transitions among these states based on the control signals and the current state of the sorting process. The sorting process ensures that the module provides to the *Memory Arbiter* an array where the highest priority client index is in position 0, the second highest in position 1 and so on.

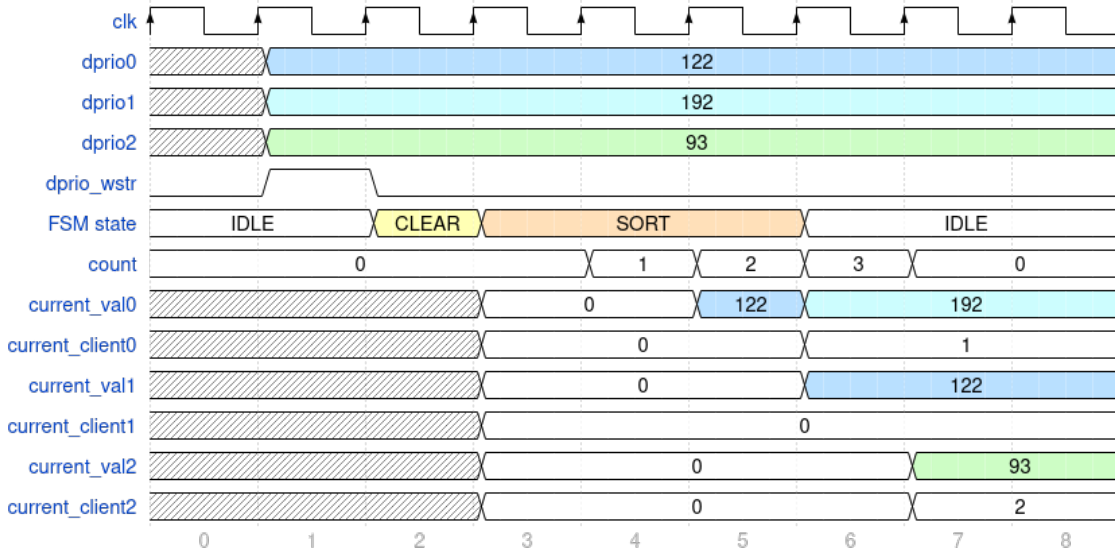


Fig. 2.5: Time evolution of sorting algorithm signals.

Fig 2.5 illustrates the time evolution of the signals involved in the sorting algorithm. The event that triggers the FSM is the pulse on the **dprio_wstrb** signal (meaning at least a new priority value has been written in the registers). The FSM clears all the flip-flops in the single sorter submodules before starting the sorting sequence. During the sorting sequence, the counter is increased until all the clients are processed. In particular, when the counter:

- is equal to 0 the reg **dprio0** is taken into account
- is equal to 1 the reg **dprio1** is taken into account
- is equal to 2 the reg **dprio2** is taken into account

At the end of the sorting algorithm, the value of **current_client** signals are respectively 1, 0 and 2 because the client with the highest priority is the one with index 1 (**dprio1 == 192**), the client with second highest priority is the one with index 0 (**dprio0 == 122**) and the client with lowest priority is the one with index 2 (**dprio2 == 93**)

2.2.2 Priority Switch Logic

The priority switch logic switches between static and dynamic priority according to the operating mode (**Control Mode**). In dynamic mode, the priority list is only updated when the sorting state machine is **SORTER_IDLE**. The priority list holds the clients with the highest priority at index 0, second highest at index 1, and so on.

The *Memory Arbiter* operation respects the following requirements:

- **DR01:** It shall be possible to reset the state of the *Memory Arbiter* by toggling the **rst** input signal;
- **DR02:** All signals shall react to the rising edge of the input **clk** signal;
- **DR03:** The architecture contains: 3 CIFs, 1 MIF, 1 APB;
- **DR04:** The priority of the clients is configurable by registers;
- **DR05:** The priority of the clients can change dynamically;
- **DR06:** The *Memory Arbiter* must be disabled to change the dynamic priority;
- **DR07:** Before re-enabling the *Memory Arbiter* after a dynamic priority change, the internal sort algorithm must be finished.
- **DR08:** The MIF should communicate with a single CIF at a time. Two (or more) CIFs cannot be ACK'ed in the same clock cycle;
- **DR09:** The MIF should accept the request of the CIF with the highest priority. If all clients have the same priority then the default priority must be respected: CIF1 before CIF2 before CIF3;
- **DR10:** The MIF should close handshake/communication with CIF by sending an acknowledge signal;
- **DR11:** The CIF and the MIF are following *SDT* protocol.

3 SyoSil Data Transfer Protocol

The SyoSil Data Transfer Protocol (*SDT*) is a traditional data transfer protocol. The protocol contains 6 signals: **rd**, **wr**, **addr**, **rd_data**, **wr_data** and **ack**. Based on the direction of the signals the protocol can be used in two variants: as a **producer** and as a **consumer**. The Table 3.3 describes the two variants.

Signal name	Producer direction	Consumer direction	Comment
rd	output	input	When asserted, the client requests a read.
wr	output	input	When asserted, the client requests a write.
addr	output	input	The read/write address. Valid when rd or wr is asserted.
rd_data	input	output	The read data. Valid when ack is asserted.
wr_data	output	input	The write data. Valid when wr is asserted
ack	input	output	Acknowledge signal. DUT acknowledges when a request has been served.

Table 3.3: SDT protocol

A timing diagram outlining the behavior of the protocol is shown in the Figure 3.1.

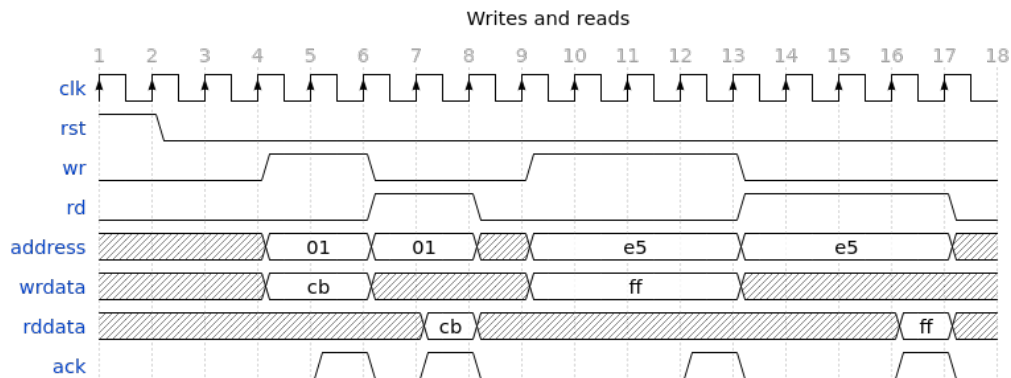


Fig. 3.1: Write and read operations using the *SDT* protocol.

The protocol has the following parameters:

- ADDR_WIDTH, the width of the address signal
- DATA_WIDTH, the width of the data signal

The SDT protocol has the following invariants:

- Asserting **rd** and **wr** at the same time is illegal
- When **rd** or **wr** is asserted, then **addr** must not be X
- When **wr** is asserted, then **wr_data** must not be X

4 Memory Arbiter Testbench

The testbench setup can be seen in Figure 4.1. All components were created by using the UVM base classes from the *PyUVM* library. The top entity is the `uvm_test` class containing a top sequence, a configuration object, the `uvm_environment`, and the interfaces for connecting to the DUT.

The environment consists of three producer *uVC*s and a single consumer *uVC*, responsible for handling the data between the testbench and the DUT through the interfaces. The three producer *uVC*s acts as clients to the memory, while the consumer *uVC* acts as a memory and handles the transactions. Additionally, the environment contains a Virtual Sequencer, a Register Model, an APB-*uVC*, a configuration object, a Reference Model handler, and a Scoreboard.

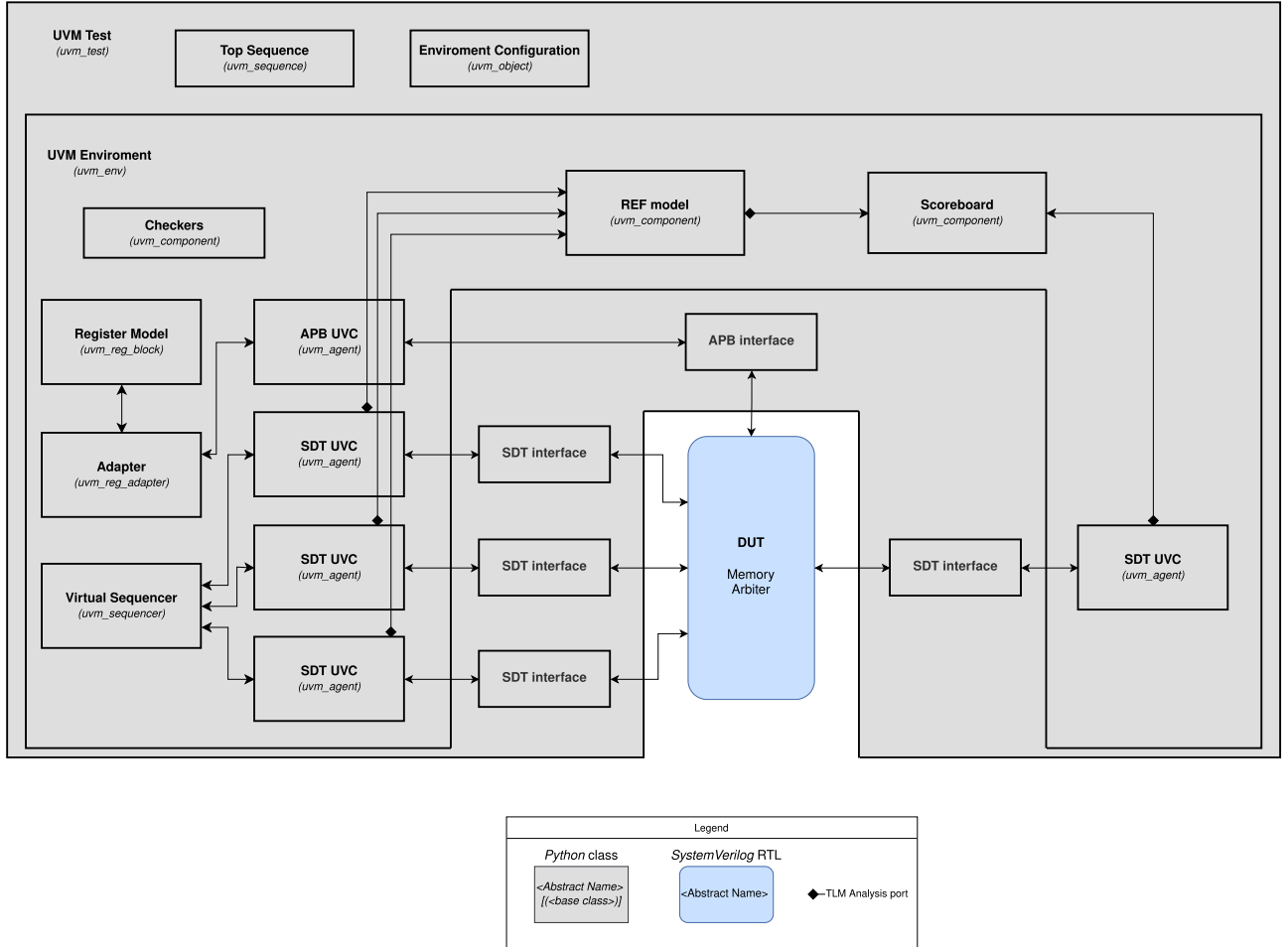


Fig. 4.1: Memory Arbiter testbench

The interfaces developed for connecting to the DUT are Python objects implemented to facilitate the connection between the *Memory Arbiter* and the testbench. The **Client IF X** is a Python object developed to facilitate the connection between the *Memory Arbiter* and the testbench, through the cocotb constructor, `cocotb.top.<signal_name>.value`. This object contains the members for all the signals available in the *SDT* protocol, operating as a typical **SystemVerilog interface** class. This interface also contains members for the **clock** and **reset** signals, which must be set when instantiated. From that point on, all inputs and outputs are probed through this interface, decoupling it from the DUT. The interfaces for the *uVC* are

instantiated in the `base test` and stored in the configuration object using the UVM `ConfigDB`, during the `build phase`. The signals are later connected, during the `connect phase` of the `base test`, storing the signals objects for the correct DUT ports in each `uVC`.

The management of the arbitration for the clients connected to the *Memory Arbiter* is dependent on writing to registers. To handle writing to registers the environment has a Register Model to keep track of the registers within the DUT. As the Register Model uses the APB protocol, an adapter was developed, to be responsible for converting APB-items to and from register items. The APB-*uVC* is responsible for writing to the registers in the DUT. Using the Register Model in the testbench facilitates `reads` and `writes` operations to the registers in the DUT, thus dynamically changing the priority of the arbiter. The provided Register Model, as shown in the Figure 4.2, contains the two registers of the *Memory Arbiter*: the `control` and the `dynamic priority` (DPrio) registers.

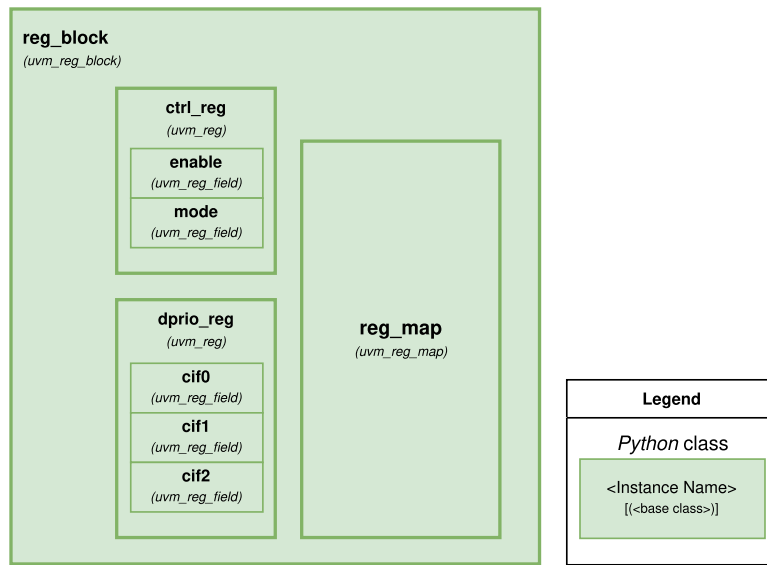


Fig. 4.2: Block diagram of the register model

The `control` register, contains two fields: `enable` and `mode`. When the `enable` field is set to 0 the module will operate without any arbitration. When the `enable` field is set to 1 the module will operate with the arbitration mode defined in the `mode` field. The `mode` field can be set to 0, to operate with the static prioritization, or set to 1, to operate with the dynamic prioritization defined in the DPrio register. The DPrio register contains a field for each CIF (CIF1, CIF2, and CIF3), where it can be set the priority value.

Register	Field	Bit size	Reset value	Access type	Description
control	-	32	-	RW	Manage arbitration operation
	enable	1	0	RW	Enable (1) or Disable (0) the arbitration operation
	mode	2	0	RW	Set prioritization mode: static (0-2-3), dynamic (1)
DPrio	-	32	-	RW	Manage dynamic prioritization
	cif0	8	0	RW	Set priority for CIF1
	cif1	8	0	RW	Set priority for CIF2
	cif2	8	0	RW	Set priority for CIF3

Table 4.4: Register Model registers

5 Assignment

This section provides a description for the assignment to create a fully working testbench to verify the *Memory Arbiter* behavior.

The assignment will cover the following tasks:

- A1. Development of a Verification Plan;
- A2. Integration of *uVCs* in the testbench;
- A3. Implementation of the testbench configuration objects;
- A4. Implementation of test cases, including sequences and virtual sequences;
- A5. Implementation and integration of the Reference model;
- A6. Implementation and integration of the Scoreboard component;
- A7. Implementation of checkers for the *SDT* protocol;
- A8. Implementation of checker for the *Memory Arbiter*;
- A9. Implementation of coverage collectors and generation of coverage reports.

The minimum requirements to complete the assignment include the steps until the implementation of the test case and the visual inspection of the waveforms to validate the results. All the other tasks can be considered as an extra. Start by implementing the mandatory tasks and, if time allows it, continue with the additional exercises.

Start by analyzing the provided files inside `<ROOT>/marb`:

- The Memory Arbiter design (the RTL code), located in `<ROOT>/marb/src/rtl`;
- The skeleton of the UVM testbench, using *PyUVM*, including a basic test case (`<ROOT>/marb/src/tb/tests/cl_marb_basic_test.py`) and a basic virtual sequence (`<ROOT>/marb/src/tb/vseqs/cl_marb_basic_seq.py`);
- The verification plan template provided in the presentation.

5.1 A1. Development of a Verification Plan

Define a verification plan based on the provided one. The plan should map the design requirements to the coverage collector information (coverage classes, covergroups or similar), and verification criteria, e.g., checker, scoreboard, assertion or reference model.

5.2 A2. Integration of *uVC*s in the testbench

A skeleton of the testbench was provided. In the source code, the following *uVC*s can be found, already developed and ready to be integrated:

- clock *uVC*, (`<ROOT>/marb/src/tb/uvc/clock`);
- reset *uVC*, (`<ROOT>/marb/src/tb/uvc/reset`);
- SDT *uVC*, (`<ROOT>/marb/src/tb/uvc/sdt`);
- APB *uVC*, (`<ROOT>/marb/src/tb/uvc/apb`).

To complete the testbench the SDT *uVC* must be finalized and integrated.

5.2.1 A2.1 Implementation of the *SDT uVC*'s driver

Locate the *SDT uVC*'s source files inside `<ROOT>/marb/src/tb/uvc/sdt/src` . Create the source file for the driver and implement the necessary code. The run phase of the driver should:

- Wait until an item is ready to be processed;
- Process the request item and generate DUT traffic;

Both producer and consumer variants should be implemented;

The implementation for both variants should reflect the *SDT* protocol's specifications;

- Inform the sequencer when the operation is complete;
- Send a response item back to the sequencer.

5.2.2 A2.2 Integration of *uVC*s in the testbench

Integrate and connect the *SDT uVC* in the testbench. The following steps can be followed for a generic integration of an *uVC*:

1. Define the configuration object for the desired *uVC* in the base test;
If needed, get the DUT parameters to pass them to the configuration object;
2. Define the required interfaces for the *uVC* and implement the required connections in the base test;
3. In the environment component instantiate the required *uVC* and pass the handler to the configuration object;
4. In the environment component implement the required connections for the agent:
 - agent's sequencer to the virtual sequencer;
 - agent's analysis port to the scoreboard, if this component is present in the testbench;
 - agent's request analysis port to the reference model, if this component is present in the testbench;

5.2.3 A2.3 Definition of the *SDT uVC*'s configuration object

The configuration objects for each provided *uVC*s can be found in
<ROOT>/marb/src/tb/uvc/<uvc>/src/cl_<uvc>.config.py .

Analyze the configuration object for the *SDT uVC*, inside
<ROOT>/marb/src/tb/uvc/sdt/src/cl_sdt.config.py .

The following parameters must be defined for the configuration object.

Parameter	Values (<i>default</i>)	Description
ADDR_WIDTH	<i>integer value (1)</i>	Set the 'addr' width for the interface
DATA_WIDTH	<i>integer value (1)</i>	Set the 'data' width for the interface
is_active	(UVM_ACTIVE) / UVM_PASSIVE	Set Agent type
vif	<i>object handler</i>	Handler for the virtual interface
driver	CONSUMER / PRODUCER	Set Driver type
num_consumer_seq	<i>integer value (None)</i>	Set the number of sequences that the consumer expects to receive from the producers. Default is 'None', which means will reply to all received
enable_transaction_coverage	(True) / False	Controls if transaction coverage is sampled or not
enable_delay_coverage	(True) / False	Control the coverage for the transaction delay, if exists any
seq_item_override	<i>SequenceItemOverride value (Default)</i>	Control knob for monitor sequence item overriding If default

HINT: The method `sdt_change_width`, provided in <ROOT>/marb/src/tb/uvc/sdt/src/sdt.common.py, can be used to define the correct width parameters in the *SDT uVC*.

5.2.4 A2.3 *Optional exercise*

As an extra step the `clock` and `reset` *uVC*s, can also be integrated. This step can be done after the mandatory steps to complete the assignment have been completed.

5.3 A3. Implementation of the testbench configuration object

Implement the configuration object for the testbench and integrate it in the base test. This configuration object must contain the handlers for the configuration objects of the available *uVC*s.

5.4 A4. Implementation of test cases, including sequences and virtual sequences

Implement two test cases:

- Random traffic test case with static priority;
- Random traffic test case with dynamic priority;

5.4.1 A4.1 Random traffic test case with static priority

Implement a random traffic test case, inside `<ROOT>/marb/src/tb/tests` , to send a random number of requests to the *Memory Arbiter* from all the different client interfaces. The clients must have a static prioritization, the default one. The behavior of the DUT shall be checked by inspecting the waveforms and ensuring all signals behave like expected.

5.4.2 A4.2 Random traffic test case with dynamic priority

Implement a random traffic test case, inside `<ROOT>/marb/src/tb/tests` , to send a random number of requests to the *Memory Arbiter* from all the different client interfaces. The clients must have a dynamic prioritization, defined randomly. Remember to adhere to the design requirements to change the dynamic priority of the *Memory Arbiter*. The behavior of the DUT shall be checked by inspecting the waveforms and ensuring all signals behave like expected.

5.5 A5 (*Optional*) Implementation and integration of the Reference model

Implement the Reference model to generate golden samples to be compared with the *Memory Arbiter* DUT output. The code can be written using C or Python language. Both the stimuli input and the registers' configuration information should be provided to the model to be used with both static and dynamic priority configurations. The model output shall be connected to the Scoreboard component to be compared with the DUT output.

Note. To verify the arbitration process, additional APs in the *SDT uVC* (`request_ap`) were defined and used. If analysis ports called `ap` are used, then the request from a client should be waited for to be served and then forwarded through the port. Consequently, by using these analysis ports, the sequence items will always be observed after the arbitration process (depending on the time of the request, but also the priority of the client). To overcome this limitation, `request_ap` were defined. The major benefit of these additional ports is that the request of the transaction is broadcast immediatly. As a result, concurrent requests on CIF0..2 can be observed by the reference model. The connection between the reference model and the *uVC* of the *SDT* producer is made through these `request_ap` ports.

5.6 A6 (*Optional*) Implementation and integration of the Scoreboard component

Implement the Scoreboard component to compare the DUT results against the Reference Model. The Scoreboard must generate an error for every mismatching results.

5.7 A7 (*Optional*) Implementation of checkers for the *SDT* protocol

Implementation of protocol checkers to ensure that the *SDT uVC* is behaving according to the protocol specifications.

The checkers should be implemented as Python coroutines, which are running in parallel with the other processes of the simulation. Follow the implementation steps as follows:

1. Create a file named `sdt_if_assertions`, inside the folder `<ROOT>/marb/src/tb/uvc/sdt/src`.
2. Add the protocol checkers inside this file.
3. Import the file in the `__init__.py` from the `<ROOT>/marb/src/tb/uvc/sdt/src`.
4. Connect the checker interface to the DUT signals and start the checking coroutine in the `base_test`, located in `<ROOT>/marb/src/tb/uvc/sdt/tb/cl_sdt_b2b_base_test.py`.
5. Connect the checker interface to the DUT signals and start the checking coroutine in the `base_test`, located in `<ROOT>/marb/src/tb/cl_marb_tb_base_test.py`.

5.8 A8 (Optional) Implementation of a coverage collector for the *Memory Arbiter* and generation of coverage reports

Implement a coverage collector for the *Memory Arbiter*, populate it with `covegroups` and generate a report with the coverage results.

The coverage collector class must respect the following specifications:

1. Must extend from the `uvm_subscriber` and be connected to the `monitor` of the MIF;
2. Contains a `coverpoint` which detects a `write` followed by a `read` to the same `address`. To achieve this, the coverage logic maintains state information that stores the type of the previous operation (`read` / `write`) and its `address`. When a new transaction occurs, it compares the current operation and address with the stored values to determine whether the condition has been met.

There are two possible ways to define this coverage behavior:

- (a) **Back-to-Back Accesses.** In this case, the coverpoint targets scenarios where the read immediately follows the write to the same address, with no intermediate accesses to any other address.

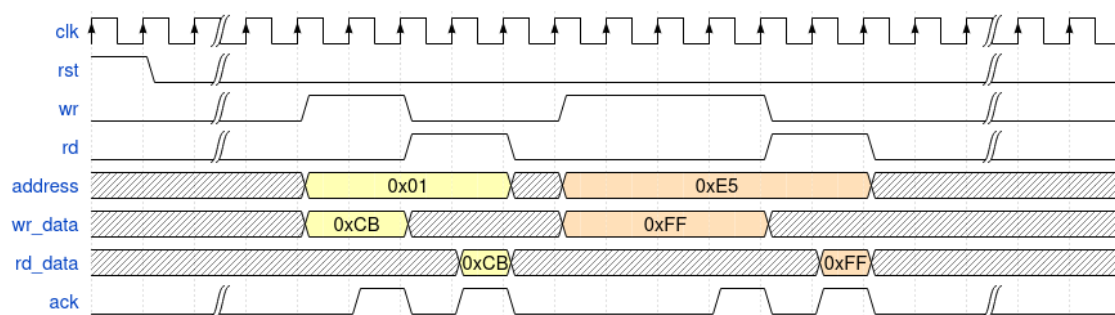


Fig. 5.1: Back-to-back accesses example.

- (b) **Non-consecutive accesses.** In this case, the coverpoint targets scenarios where the read does not immediately follow the write to the same address, allowing for intermediate accesses to any other address.

For implementation, the `coverpoint` must have a bit, as a `bin`, that for each `address` marks it as observed or not.

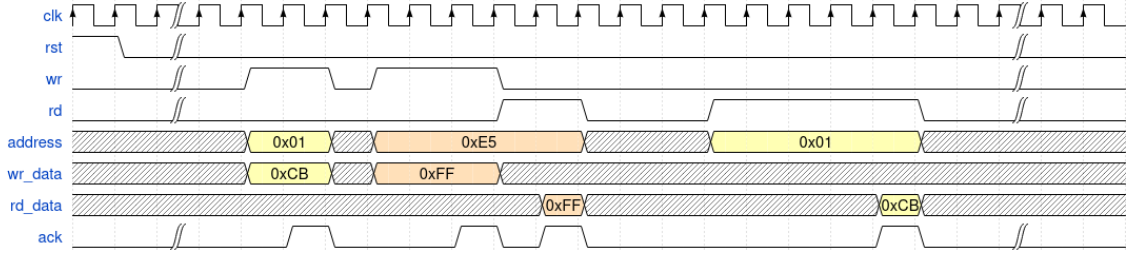


Fig. 5.2: Non-consecutive accesses.

3. Add a burst detection coverage. This coverage will evaluate the maximum length of the sequential addresses reached.

3.1 Add coverage which samples the start **address** and the number of sequential addresses. Thus, is needed to keep the **address** bus width low, e.g. 8 bits. Then the longest burst is 256 transactions. For each write invocation, must be validated that the **address** value is the following value of the previous one. If so, then the counter of the **address** must be incremented, if not then must be reset.

3.2 For each transaction the start **address** and the length must be sampled. The cross coverage should be added to get the full **address** space coverage. Note that as the **address** grows the lengths decrease, so the "ignore bins" option has to come into play.

4. Then, instantiate the coverage class inside the coverage collector and sample the coverage using the **write** method.

The PyUCIS-viewer tool can be used to visualize the coverage report by running e.g.:

```
(.venv) [username@servername tb]$ pyucis-viewer sim_build/<test-name>_cov.xml
```

5.8.1 A8.1 (Optional) Coverage scenario: A request of the three CIFs is made in parallel

The component must monitor the request signals and does coverage for the case when a request of all three CIFs is made in parallel.

5.9 A9 (Optional) Implementation of checker for the *Memory Arbitrator*

Implement a checker component that will monitor the **ack** signals of each CIF in order to check that only a single **ACK** is detected during an operation. Two (or more) CIFs cannot be **ACK**'ed in the same clock cycle.