Assignment

June 2024

Contents

1 Introduction **3**

2 Design of the Memory Arbiter **4**

3 SyoSil Data Transfer Protocol **6**

4 Memory Arbiter Testbench **7**

5 Assignment **10**
5.1 A1. Development of a Verification Plan 10
5.2 A2. Integration of *uVC*s in the testbench 10
5.2.1 A2.1 Implementation of the *SDT uVC*'s driver 11
5.2.2 A2.2 Integration *uVC*s in the testbench 11
5.2.3 A2.3 Definition of the *SDT uVC*'s configuration object 11
5.2.4 A2.3 *Optional* exercise 12
5.3 A3. Implementation of the testbench configuration object 12
5.4 A4. Implementation of test cases, including sequences and virtual sequences . . 12
5.4.1 A4.1 Random traffic test case with static priority 12
5.4.2 A4.2 Random traffic test case with dynamic priority 12
5.5 A5 *(Optional)* Implementation and integration of the Reference model 13
5.6 A6 *(Optional)* Implementation and integration of the Scoreboard component . . 13
5.7 A7 *(Optional)* Implementation of checkers for the *SDT* protocol 13
5.8 A8 *(Optional)* Implementation of a coverage collector for the *Memory Arbiter* and generation of coverage reports 13
5.9 A9 *(Optional)* Implementation of checker for the *Memory Arbiter* 14

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.

The provided *Memory Arbiter* IP arbitrates between 3 memory clients, the access to the memory for read and write operations. The *Memory Arbiter* has a default static priority for the clients connected to the arbiter, but the priority can be dynamically changed by writing into the two registers handling the priority, the "Control" and the "Dynamic Priority" registers.

The default static prioritization among the CIFs is as follows:

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

The IP has the following external interfaces/signals:

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

The Figure 2.1, shows a high level diagram of the *Memory Arbiter* IP. As can be seen in the Figure 2.1, the device uses two data transfer protocols: the SyoSil Data Transfer Protocol (*SDT*), that is used by the external clients to make requests to access the memory, and the Advanced Peripheral Bus (APB) for allowing direct operations into the device's registers, in order to configure it.

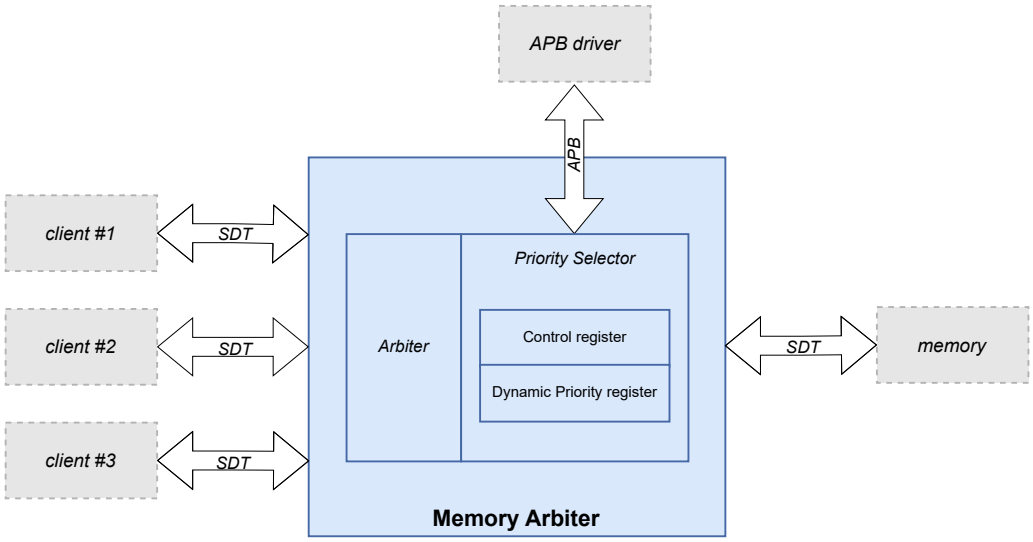

Fig. 2.1: High level diagram of the Memory Arbiter design.

The core logic of the *Memory Arbiter* analyses the three client interfaces (CIFs) and always selects the one with the highest priority to be served. The design always checks for the latest priority set. When a request has been selected the *Memory Arbiter* drives the memory interface with the request.

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 MIF should communicate with a single CIF at a time, two (or more) CIFs cannot be ACK'ed in the same clock cycle;
- **DR07:** 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;
- **DR08:** The MIF should close handshake/communication with CIF by sending an acknowledge signal;
- **DR09:** 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 3.1 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 3.1: SDT protocol

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

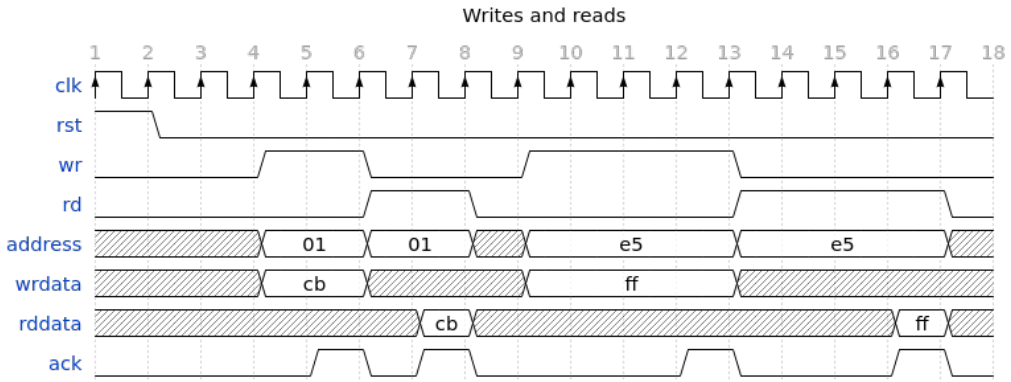

Fig. 3.1: Write and read operations using the *SDT* protocol.

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 type is illegal
- When `rd` or `wr` is asserted, then adder must not be `X`
- When `wr` is asserted, then wr_data must not be `X`

4 Memory Arbiter Testbench

The testbench is created by using the UVM base classes from the *PyUVM* library. An overview of the testbench setup can be seen in Figure 4.1. 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. These interfaces are Python objects implemented to facilitate the connection between the memory arbiter and the testbench.

The environment consists of three producer *uVC*s and a single consumer *uVC*, responsible for handling data between the testbench and the DUT through the interfaces. The three producer *uVC*s acts as clients, while the consumer *uVC* acts as a memory and handles the transactions.

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. The 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 environment also contains the APB-*uVC* responsible for writing to the registers in the DUT. To further handle writing to registers, the environment has a Register Model and an adapter for keeping track of the registers within the DUT. Additionally, the environment contains a Virtual Sequencer, a Scoreboard, and a Reference Model handler.

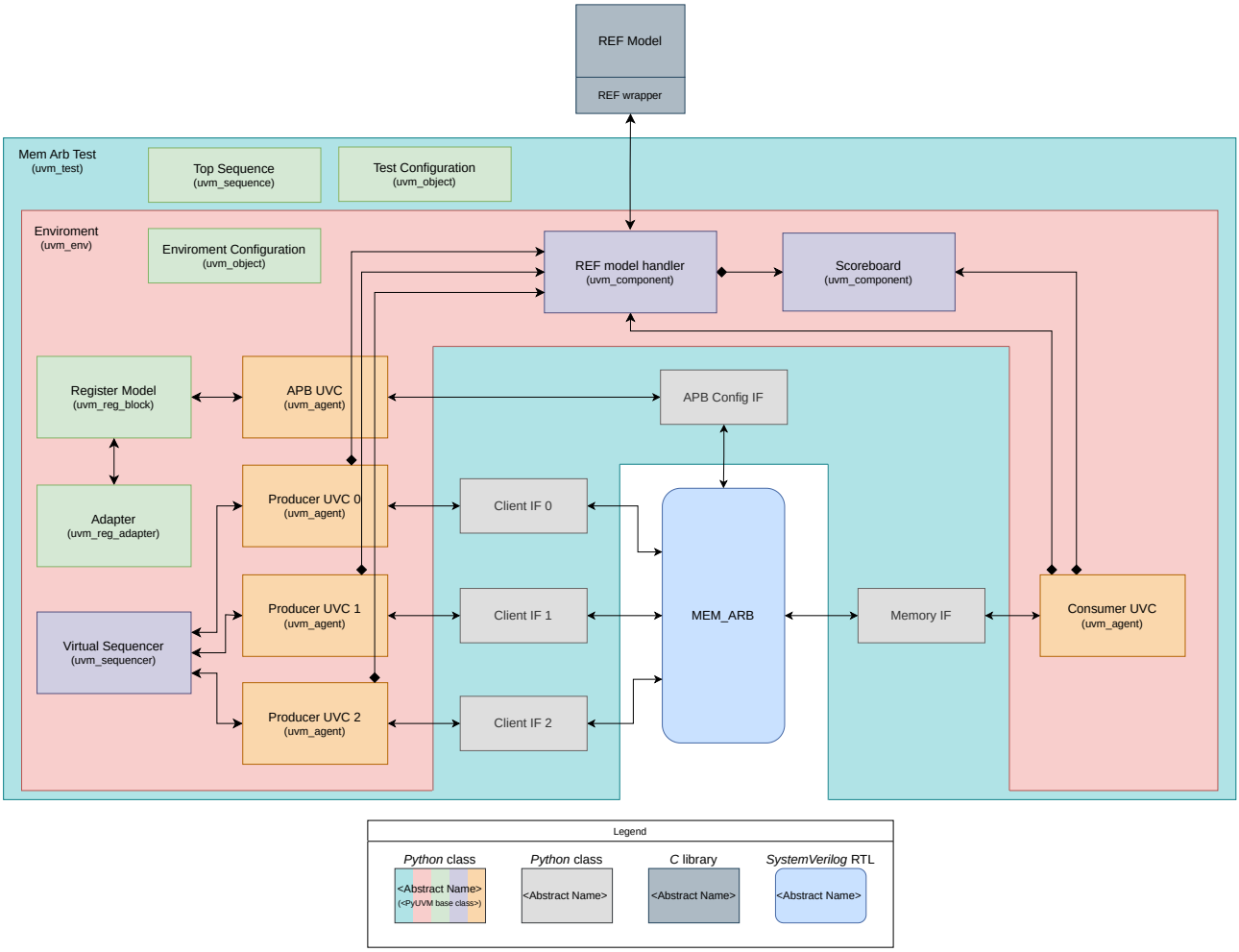

Fig. 4.1: Memory Arbiter testbench

The priority management of the clients connected to the *Memory Arbiter* is dependent on writing to registers, which requires a Register Model. The provided Register Model, Figure 4.2, contains the two registers of the *Memory Arbiter*, the "Control" and the "Dynamic Priority" (DPrio) registers. To use the Register Model in the testbench an adapter was needed for converting between register items and bus items. As the register uses the APB protocol, the adapter was responsible for converting APB-items to and from register items. 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.

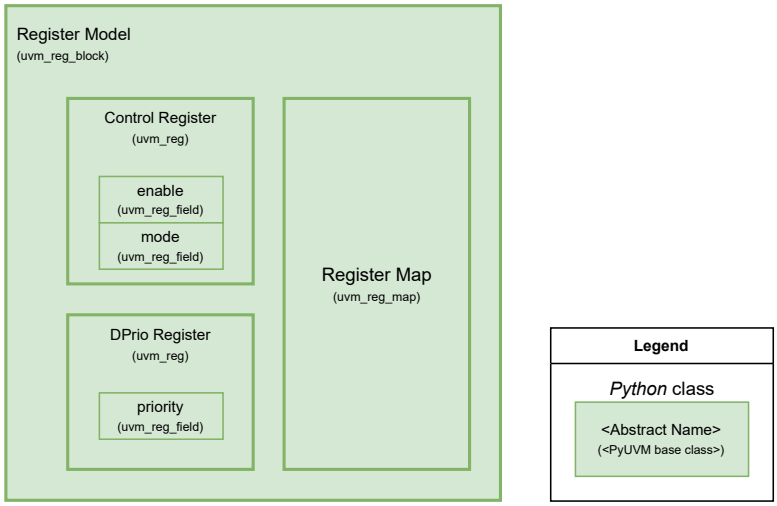

Fig. 4.2: Block diagram of the register model

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 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 single field, named `priority`, where is defined the order of the CIFs by id.

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 *uVC*s 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 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 an 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 *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 a *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;
   - agent's request analysis port to the reference model;

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.

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*.

Parameter	Values *(default)*	Description
`ADDR_WIDTH`	*integer value* **(1)**	Set the 'addr' width for the interface
`DATA_WIDTH`	*integer value* **(1)**	Set the 'data' width for the interface
`is_active`	`(UVM_ACTIVE)` / `UVM_PASSIVE`	Set Agent type
`vif`	*object handler*	Handler for the virtual interface
`driver`	`CONSUMER` / `PRODUCER`	Set Driver type
`num_consumer_seq`	*integer value* **(None)**	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`	*SequenceItemOverride value* **(Default)**	Control knob for monitor sequence item overriding If default

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. 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.

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. Locate the file `sdt_if_assertions`, inside the folder `<ROOT>/marb/src/tb/uvc/sdt` .
2. Add the protocol checkers inside this file as by following the same steps as in the exercises.
3. Connect the checker interface to the DUT signals and start the checking coroutine in `<ROOT>/marb/src/tb/uvc/sdt/tb/cl_sdt_b2b_base_test.py` .

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

Implement the 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`;
2. Must be connected to the monitor of the MIF;
3. Contains a `coverpoint` which detects a `read` followed by a `write` to the same `address`. This can be done by having some state which stores the previous command and address and then the `coverpoint` must have a bit, as a bin, that either it is observed or not.

4. Add a burst detection coverage. Add coverage which samples a start address and burst length. 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 is the following value of the previous one. If so then the counter must be incremented, if not then must be reset.

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

6. Then instantiate the coverage class inside the subscriber and sample the coverage in the write function.

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

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.