TLM2.0 compliant AXI Transactor Specification

Eyck Jentzsch

Contents

1 Preface **2**

1.1 About this Specification 2

1.2 References 2

2 Introduction **2**

3 Channel Signal Mapping **3**

3.1 Write channels 3

3.1.1 Write Address (AW) 3

3.1.2 Write Data (W) 3

3.1.3 Write Response (B) 4

3.2 Read channels 4

3.2.1 Read address (AR) 4

3.2.2 Read data (R) 4

3.3 Snoop channels 5

3.3.1 Snoop address (AC) 5

3.3.2 Snoop response (CR) 5

3.3.3 Snoop data (CD) 5

4 DMI and Debug Transport Communication **5**

5 Blocking Communication **5**

5.1 b_transport and b_snoop Call Sequence 6

6 Non-blocking Communication **6**

6.1 Extended Phases 6

6.2 Permitted Phase Transitions 6

6.2.1 Write Access 6

6.2.2 Read Access 7

6.2.3 Snoop Access 7

6.3 nb_transport Call Sequences for basic AXI/ACE Protocol Transactions 8

6.4 nb_transport Call Sequence for ACE Transaction Groups 8

6.5 nb_transport Transaction and Clock Boundaries 10

6.6 Non-ignorable Phases and Protocol Traits 10

7 Implementation Guideline **10**

7.1 Payload Extension 10

7.2 Phases Declarations and Protocol traits 15

7.3 Socket Interfaces and Sockets 16

8 Recommended Additions **18**
8.1 Convenience Layer 18
8.2 Transaction Tracing 18
8.3 Protocol Checker 19

1 Preface

1.1 About this Specification

This specification details the representation of the AXI and ACE protocol in an TLM2.0 compliant implementation. The definition of the protocol adheres to 'AMBA® AXI™ and ACE™ Protocol Specification' (see section 1.2). The specification focuses on AXI4 and ACE as long as nothing else is explicitly noted.

It is assumed that the reader is familiar with the TLM-2.0 language reference manual (see section 1.2) version TLM 2.0.1 and has some basic experience with TLM modeling. Basic understanding of the AXI and ACE protocol is beneficial.

1.2 References

This manual focuses on the extension of TLM2.0 to model the AMBA AXI and ACE protocol at loosly and approximately timed accuracy. For more details on the protocol and semantics, see the following manuals and specifications:

- AMBA® AXI™ and ACE™ Protocol Specification version 2, 22 February 2013
- IEEE Std. 1666 TLM-2.0 Language Reference Manual

2 Introduction

This document specifies the way sockets communicate to each other while modeling properties of the AXI protocol. As such all channels of an AXI interface are represented by a single TLM socket. These channels are:

- Write address (AW)
- Write data (W)
- Write response (B)
- Read address (AR)
- Read data (R)
- Snoop address (AC)
- Snoop response (CR)
- Snoop data (CD)

The specification describes the way sockets exchange information not how it is to be implemented.

3 Channel Signal Mapping

The following table lists the mapping of AXI/ACE signals to either the payload, its extensions or phases.

> Phases are defined for the non-blocking protocol only. Wherever possible the AXI protocol is mapped to the generic protocol phases to ease interoperability with the TLM2.0 standard. For more information about phases see section 6.6.

3.1 Write channels

3.1.1 Write Address (AW)

AXI/ACE signal	where	data structure	name	type
AWID	extension	axi::common	ID	int
AWADDR	payload	tlm::generic_payload	address	uint64_t
AWLEN	extension	axi::request	length	uint8_t
AWSIZE	extension	axi::request	size	uint3_t
AWBURST	extension	axi::request	burst	burst_e
AWLOCK	extension	axi::request	lock	uin4
AWCACHE	extension	axi::request	cache	cache_e
AWPROT	extension	axi::request	prot	uint3_t
AWQOS	extension	axi::request	qos	uint4_t
AWREGION	extension	axi::request	region	uint4_t
AWUSER	extension	axi::common	user	uint64_t
AWVALID	phase	BEGIN_REQ		
AWREADY	phase	END_REQ		
AWDOMAIN	extension	axi::request	domain	domain_e
AWSNOOP	extension	axi::request	snoop	snoop_e
AWBAR	extension	axi::request	barrier	barrier_e
AWUNIQUE	extension	axi::request	unique	bool

3.1.2 Write Data (W)

AXI/ACE signal	where	data structure	name	type
WID	extension	axi::common	ID	int
WDATA	payload	tlm::generic_payload	data/len	
WSTRB	payload	tlm::generic_payload	byte_enable	
WLAST	phase	BEGIN_REQ		
WUSER	extension	axi::common	user	uint64_t
WVALID	phase	BEGIN_PARTIAL_REQ		
		BEGIN_REQ		
WREADY	phase	END_PARTIAL_REQ		
		END_REQ		

3.1.3 Write Response (B)

AXI/ACE signal	where	data structure	name	type
BID	extension	axi::common	ID	int
BRESP	extension	axi:response	resp	resp_e
BUSER	extension	axi::common	user	uint64_t
BVALID	phase	BEGIN_RESP		
BREADY	phase	END_RESP		
WACK	phase/extension	ACK/axi::response	ack	bool

3.2 Read channels

3.2.1 Read address (AR)

AXI/ACE signal	where	data structure	name	type
ARID	extension	axi::common	ID	int
ARADDR	payload	tlm::generic_payload	address	uint64_t
ARLEN	extension	axi::request	len	uint8_t
ARSIZE	extension	axi::request	size	uint3_t
ARBURST	extension	axi::request	burst	burst_e
ARLOCK	extension	axi::request	lock	bool
ARCACHE	extension	axi::request	cache	cache_e
ARPROT	extension	axi::request	prot	uint3_t
ARQOS	extension	axi::request	qos	uint4_t
ARREGION	extension	axi::request	region	uint4_t
ARUSER	extension	axi::common	user	uint64_t
ARVALID	phase	BEGIN_REQ		
ARREADY	phase	END_REQ		
ARDOMAIN	extension	axi::request	domain	domain_e
ARSNOOP	extension	axi::request	snoop	snoop_e
ARBAR	extension	axi::request	barrier	barrier_e

3.2.2 Read data (R)

AXI/ACE signal	where	data structure	name	type
RID	extension	axi::common	ID	int
RDATA	payload	tlm::generic_payload	data/len	
RRESP	extension	axi:response	resp	resp_e
RLAST	phase	BEGIN_RESP		
RUSER	extension	axi::common	user	uint64_t
RVALID	phase	BEGIN_PARTIAL_RESP BEGIN_RESP		

RREADY	phase	END_PARTIAL_RESP
		END_RESP		

RACK	phase/extension	ACK/axi::response	ack	bool

3.3 Snoop channels

3.3.1 Snoop address (AC)

AXI/ACE signal	where	data structure	name	type
ACVALID	phase	BEGIN_REQ		
ACREADY	phase	END_REQ		
ACADDR	payload	tlm::generic_payload	address	uint64_t
ACSNOOP	extension	axi::request	snoop	snoop_e
ACPROT	extension	axi::request	prot	prot_e

3.3.2 Snoop response (CR)

AXI/ACE signal	where	data structure	name	type
CRVALID	phase	BEGIN_RESP		
CRREADY	phase	END_RESP		
CRRESP	extension	axi:response	resp	resp_e

3.3.3 Snoop data (CD)

AXI/ACE signal	where	data structure	name	type
CDVALID	phase	BEGIN_PARTIAL_RESP		
		BEGIN_RESP		
CDREADY	phase	END_PARTIAL_RESP		
		END_RESP		
CDDATA	payload	tlm::generic_payload	data/len	
CDLAST	phase	BEGIN_RESP		

4 DMI and Debug Transport Communication

The direct memory interface (DMI) and debug transport interface are specialized interfaces distinct from the transport interface, providing direct access and debug access to a resources owned by a target. DMI is intended to accelerate regular memory transactions in a loosely-timed simulation, whereas the debug transport interface is for debug access free of the delays or side-effects associated with regular transactions. For more details on debug transport and DMI please refer to the 'OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL'.

5 Blocking Communication

Blocking communication is mostly used in loosely-timed (LT) models or programmer view use cases. Here the communication is abstracted and described by 2 timing points: the start and the end of the transaction. AXI/ACE sockets use the b_transport_fw interface as described in the TLM-2.0 LRM. Additionally they are required to implement the b_transport_bw interface which allows to model snoop access at this high abstraction. For more details on LT modeling please refer to the 'OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL'.

5.1 b_transport and b_snoop Call Sequence

The call sequences for blocking transactions are the same than for the generic protocol one. The backward socket interface is been extended to allow for blocking snoop accesses. The semantics of the b_snoop access are the same than the b_transport call but in backward direction.

6 Non-blocking Communication

In the non-blocking communication protocol each transaction has multiple timing points. This way the timely description is of higher accuracy and suitable e.g. for architectural exploration.

Each socket interaction is characterized by the generic payload, the phase time points and the direction of communication (forward or backward interface). Therefore the AXI channels can be identified and it is possible to route them thru the same socket.

6.1 Extended Phases

The non-blocking transactions of the AXI TLM2.0 implementation use up to 4 additional phases:

- BEGIN_PARTIAL_REQ

  denoting the start of a burst write data transfer beat
- END_PARTIAL_REQ

  denoting the end of a burst write data transfer beat
- BEGIN_PARTIAL_RESP

  denoting the start of a burst read data transfer beat
- END_PARTIAL_RESP

  denoting the end of a burst read data transfer beat

6.2 Permitted Phase Transitions

Despite the TLM2.0 LRM's base protocol a path from an initiator to a target may convey up to 3 different phases concurrently for read, write, and snoop accesses (since snoop access is modeled as a read access from target to initiator there is a differentiation).

6.2.1 Write Access

Previous state	Calling path	Phase argument on call	Phase argument on return	Status on return	Next state
idle	fw	BEGIN_PARTIAL_REQ	-	Accepted	wdata

Previous state	Calling path	Phase argument on call	Phase argument on return	Status on return	Next state
idle	fw	BEGIN_PARTIAL_REQ	END_PARTIAL_REQ	Updated	~wdata
wdata	bw	END_PARTIAL_REQ	-	Accepted	~wdata
~wdata	fw	BEGIN_PARTIAL_REQ	-	Accepted	wdata
~wdata	fw	BEGIN_PARTIAL_REQ	END_PARTIAL_REQ	Updated	~wdata
wdata	bw	END_PARTIAL_REQ	-	Accepted	~wdata
~wdata	fw	BEGIN_REQ	-	Accepted	wdatal
~wdata	fw	BEGIN_REQ	END_REQ	Updated	~wdatal
wdatal	bw	END_REQ	-	Accepted	~wdatal
~wdatal	bw	BEGIN_RESP	-	Accepted	wresp
~wdatal	bw	BEGIN_RESP	END_RESP	Updated	idle
wresp	fw	END_RESP	-	Accepted	idle
~wdatal	bw	BEGIN_RESP	-	Accepted	wresp
~wdatal	bw	BEGIN_RESP	END_RESP	Updated	~wresp
wresp	fw	END_RESP	-	Accepted	~wresp
~wresp	fw	ACK	-	Accepted	idle

6.2.2 Read Access

Previous state	Calling path	Phase argument on call	Phase argument on return	Status on return	Next state
idle	fw	BEGIN_REQ	-	Accepted	raddr
idle	fw	BEGIN_REQ	END_REQ	Updated	~raddr
raddr	bw	END_REQ	-	Accepted	~raddr
~raddr	bw	BEGIN_PARTIAL_RESP	-	Accepted	rresp
~raddr	bw	BEGIN_PARTIAL_RESP	END_PARTIAL_RESP	Updated	~rresp
rresp	fw	END_PARTIAL_RESP	-	Accepted	~rresp
~rresp	bw	BEGIN_PARTIAL_RESP	-	Accepted	rresp
~rresp	bw	BEGIN_PARTIAL_RESP	END_PARTIAL_RESP	Updated	~rresp
rresp	fw	END_PARTIAL_RESP	-	Accepted	~rresp
~rresp	bw	BEGIN_RESP	-	Accepted	rrespl
~rresp	bw	BEGIN_RESP	END_RESP	Updated	idle
rrespl	fw	END_RESP	-	Accepted	idle
~rresp	bw	BEGIN_RESP	-	Accepted	rrespl
~rresp	bw	BEGIN_RESP	END_RESP	Updated	~rrespl
rrespl	fw	END_RESP	-	Accepted	~rrespl
~rrespl	fw	ACK	-	Accepted	idle

6.2.3 Snoop Access

Previous state	Calling path	Phase argument on call	Phase argument on return	Status on return	Next state
idle	bw	BEGIN_REQ	-	Accepted	caddr
idle	bw	BEGIN_REQ	END_REQ	Updated	~caddr
caddr	fw	END_REQ	-	Accepted	~caddr

Previous state	Calling path	Phase argument on call	Phase argument on return	Status on return	Next state
~caddr	fw	BEGIN_PARTIAL_RESP	-	Accepted	cresp
~caddr	fw	BEGIN_PARTIAL_RESP	END_PARTIAL_RESP	Updated	~cresp
cresp	bw	END_PARTIAL_RESP	-	Accepted	~cresp
~cresp	fw	BEGIN_PARTIAL_RESP	-	Accepted	cresp
~cresp	fw	BEGIN_PARTIAL_RESP	END_PARTIAL_RESP	Updated	~cresp
cresp	bw	END_PARTIAL_RESP	-	Accepted	~cresp
~cresp	fw	BEGIN_RESP	-	Accepted	crespl
~cresp	fw	BEGIN_RESP	END_RESP	Updated	idle
crespl	bw	END_RESP	-	Accepted	idle

6.3 nb_transport Call Sequences for basic AXI/ACE Protocol Transactions

The figure Figure 1 below shows the protocol sequence for an AXI read and write access as well as an ACE snoop access. The snoop sequence is the same than the AXI read but with flipped forward/backward roles.

The transactions on the address and response channels are mapped to `BEGIN_REQ/END_REQ` and `BEGIN_RESP/END_RESP` of the base protocol. The AXI TLM2 protocol rules are an amendment to those described in the TLM2.0 LRM. The most important changes are:

- Premature finish of accesses (1-phase access) is not allowed. A return status of `TLM_COMPLETED` shall never occur along with the `END_RESP` phase.
- Partial phases (partial request denoted by `BEGIN_PARTIAL_REQ/END_PARTIAL_REQ` and partial response denoted by `BEGIN_PARTIAL_RESP/END_PARTIAL_RESP`) are allowed to occur 0 to many times. In any case the request and response phase needs to be finished with a non-partial reqest/reponse signaling.
- Partial request and response phases of different accesses may be interleaved.
- For the AXI4 protocol interleaved partial request phases are not allowed.

6.4 nb_transport Call Sequence for ACE Transaction Groups

The AXI™ and ACE™ Protocol Specification defines transaction groups:

- Read transactions
- Clean transactions
- Make transactions
- Write transactions
- Evict transactions

Read and Write transactions behave as in section 6.3 described. Clean, Make, and Read barrier transactions are variants of the read transaction where the returned data is ignored. Evict and Write barrier transactions are modeled as write transactions without a data transfer. Hence they will never have a partial request phase (denoted by `BEGIN_PARTIAL_REQ/END_PARTIAL_REQ` signaling).

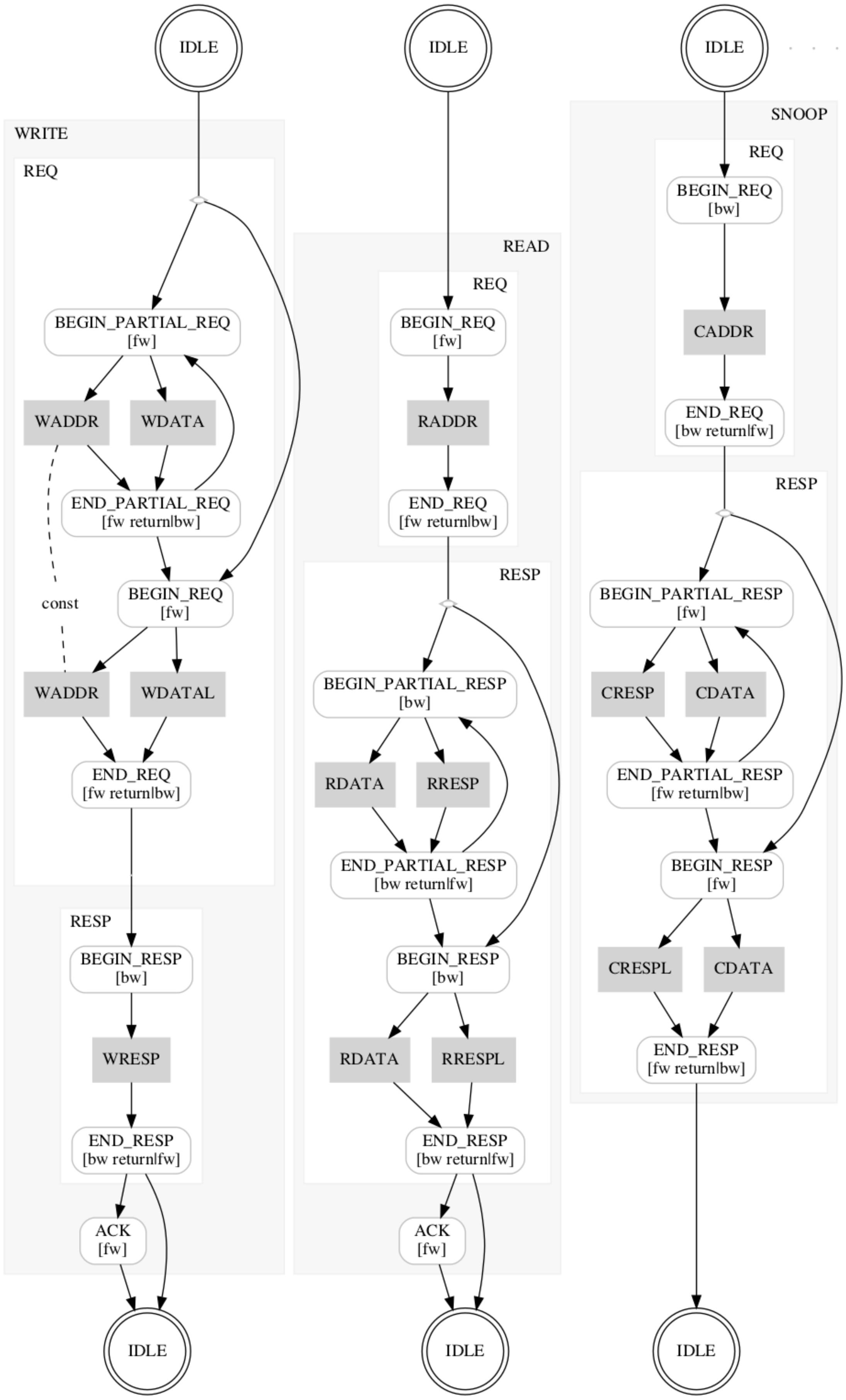

Figure 1: AXI protocol sequences

6.5 nb_transport Transaction and Clock Boundaries

Figure 2 shows by example the AXI single read and write accesses and how they map to clock boundaries. Burst reads and writes comprise additional clock cycles for the additional read or write data phases denoted by the `BEGIN_PARTIAL_*/END_PARTIAL_*` timing points.

This scheme allows to represent the AXI timing quite closely and it might be as fast as one transfer per clock.

6.6 Non-ignorable Phases and Protocol Traits

The specified AXI protocol defines two additional non-ignorable phase denoted by: `BEGIN_PARTIAL_REQ/END_PARTIAL_REQ` for the partial request phase and `BEGIN_PARTIAL_RESP/END_PARTIAL_RESP` for the partial response phase. These phases designate data transfers on the AXI channels.

As the AXI protocol defines separate channels for read and write the protocol allows up to 2 concurrent request phases at the same time on the forward interface: one for a read and one for the read channel. The distinction is made based on the command of the generic payload. The same applies to the response phase.

In case of using the ACE protocol there is an additional request phase at the backward interface and a response phase on the forward interface allowed representing the snoop accesses.

Since the aforementioned phases are non-ignorable phases a new protocol trait shall be defined.

7 Implementation Guideline

The following sections describe an implementation of the specification. As such it is not part of the specification and may be subject to change in the course of implementation.

7.1 Payload Extension

This section is going to describe the extensions provided by the AXI/ACE TLM2.0 transactor package. As outlined in section 3 there are three data structures representing the attributes of an AXI/ACE transaction. Along with the data members already described they provide some utility functions to partially decode the signals and their meaning. These are:

```
struct common {
    common() = default;
    void reset();

    enum class id_type {
        CTRL, DATA, RESP
    };

  void set_id(id_type chnl, unsigned int);
    unsigned int get_id(id_type chnl) const;
```

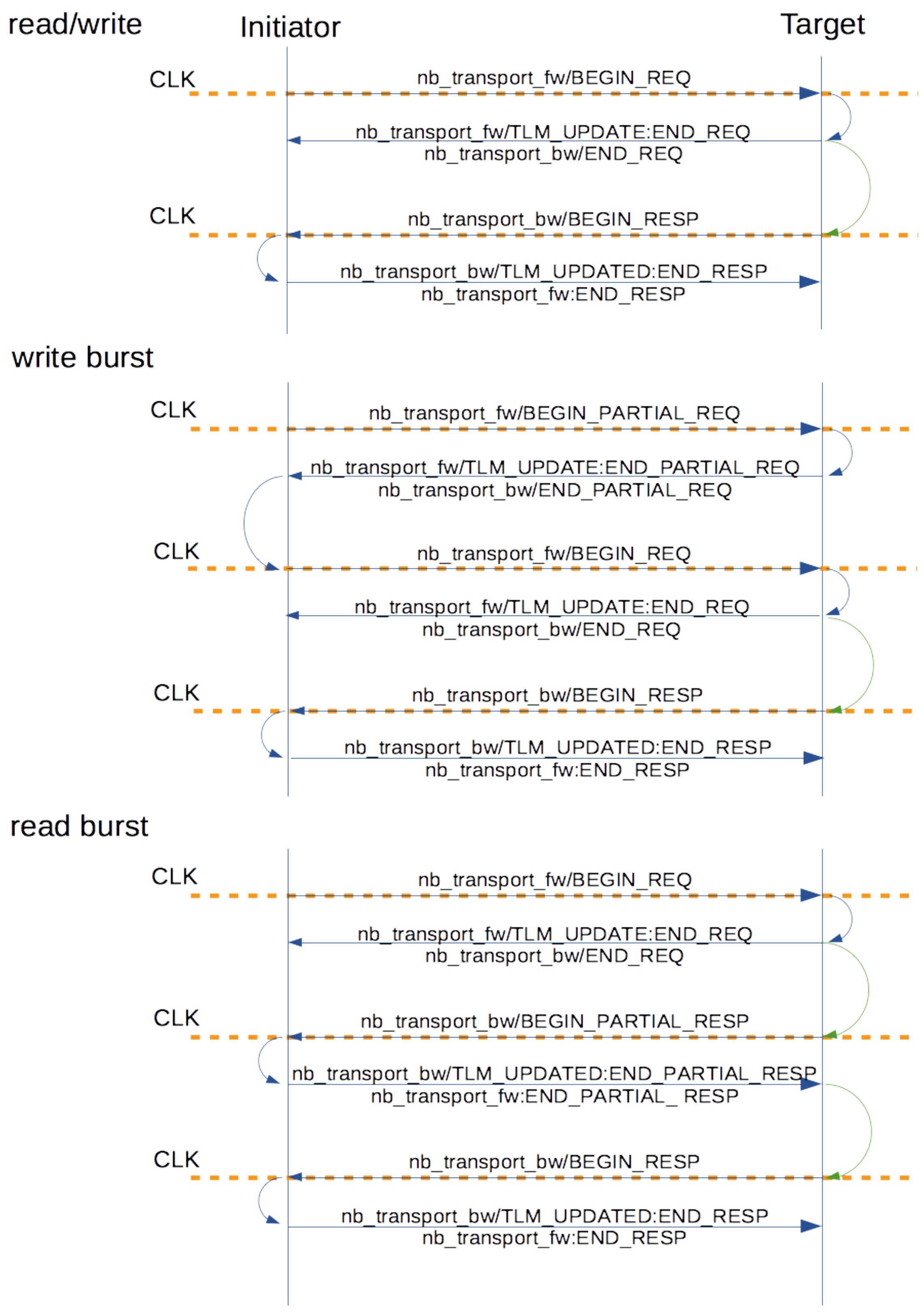
read/write
Initiator
Target
CLK
nb_transport_fw/BEGIN_REQ
nb_transport_fw/TLM_UPDATE:END_REQ
nb_transport_bw/END_REQ
CLK
nb_transport_bw/BEGIN_RESP
nb_transport_bw/TLM_UPDATED:END_RESP
nb_transport_fw:END_RESP
write burst
CLK
nb_transport_fw/BEGIN_PARTIAL_REQ
nb_transport_fw/TLM_UPDATE:END_PARTIAL_REQ
nb_transport_bw/END_PARTIAL_REQ
CLK
nb_transport_fw/BEGIN_REQ
nb_transport_fw/TLM_UPDATE:END_REQ
nb_transport_bw/END_REQ
CLK
nb_transport_bw/BEGIN_RESP
nb_transport_bw/TLM_UPDATED:END_RESP
nb_transport_fw:END_RESP
read burst
CLK
nb_transport_fw/BEGIN_REQ
nb_transport_fw/TLM_UPDATE:END_REQ
nb_transport_bw/END_REQ
CLK
nb_transport_bw/BEGIN_PARTIAL_RESP
nb_transport_bw/TLM_UPDATED:END_PARTIAL_RESP
nb_transport_fw:END_PARTIAL_ RESP
CLK
nb_transport_bw/BEGIN_RESP
nb_transport_bw/TLM_UPDATED:END_RESP
nb_transport_fw:END_RESP

Figure 2: AXI accesses w. clock boundaries

```
    void set_user(id_type chnl, unsigned int);
    unsigned int get_user(id_type chnl) const;
};
```

The request is defined as:

```
struct request {

    void reset();

    void set_length(uint8_t);
    uint8_t get_length() const;

    void set_size(uint8_t);
    uint8_t get_size() const;

    void set_burst(burst_e);
    burst_e get_burst() const;

    void set_prot(uint8_t);
    uint8_t get_prot() const;

  void set_privileged(bool = true);
    bool is_privileged() const;

    void set_non_secure(bool = true);
    bool is_non_secure() const;

    void set_instruction(bool = true);
    bool is_instruction() const;

    void set_cache(uint8_t);
    uint8_t get_cache() const;

    void set_qos(uint8_t);
    uint8_t get_qos() const;

    void set_region(uint8_t);
    uint8_t get_region() const;
};
```

Since the `AxCACHE` and `AxLOCK` signals have different interpretation the utility functions are moved into derived classes:

```
struct axi3: public request {

    axi3& operator=(const axi3&);

    void set_exclusive(bool = true);
    bool is_exclusive() const;

    void set_locked(bool = true);
    bool is_locked() const;

    void set_bufferable(bool = true);
    bool is_bufferable() const;

    void set_cacheable(bool = true);
    bool is_cacheable() const;

    void set_write_allocate(bool = true);
    bool is_write_allocate() const;

    void set_read_allocate(bool = true);
    bool is_read_allocate() const;
};

struct axi4: public request{

    axi4& operator=(const axi4&);

    void set_exclusive(bool = true);
    bool is_exclusive() const;

    void set_bufferable(bool = true);
    bool is_bufferable() const;

    void set_modifiable(bool = true);
    bool is_modifiable() const;

    void set_read_other_allocate(bool = true);
    bool is_read_other_allocate() const;

    void set_write_other_allocate(bool = true);
    bool is_write_other_allocate() const;
};

struct ace: public axi4 {

    ace& operator=(const ace& o);

    void set_domain(domain_e);
    domain_e get_domain() const;

    void set_snoop(snoop_e);
    snoop_e get_snoop() const;

    void set_barrier(bar_e);
    bar_e get_barrier() const;
    void set_unique(bool);
    bool get_unique() const;
};
```

The responses are represented as:

```
struct response {

    void reset();

    response& operator=(const response& o);

    static resp_e from_tlm_response_status(tlm::tlm_response_status);
    static tlm::tlm_response_status to_tlm_response_status(resp_e);

    void set_resp(resp_e);
    resp_e get_resp() const;

    bool is_okay() const;
    void set_okay();

    bool is_exokay() const;
    void set_exokay();

    bool is_slverr() const;
    void set_slverr();

    bool is_decerr() const;
    void set_decerr();
};

struct ace_response: public response {

    void set_cresp(uint8_t);
    uint8_t get_cresp() const;

  bool is_pass_dirty() const;
  void set_pass_dirty(bool = true);

  bool is_shared() const;
  void set_shared(bool = true);

  bool is_snoop_data_transfer() const;
  void set_snoop_data_transfer(bool = true);

  bool is_snoop_error() const;
  void set_snoop_error(bool = true);

  bool is_snoop_was_unique() const;
  void set_snoop_was_unique(bool = true);
```

```
    bool is_ack() const;
    void set_ack(bool = true);
};
```

Those different structures are combined in an generic payload extension. The use of just one extension eases the implementation of initiators and targets, careful data layout ensures that the memory footprint is as little as possible:

```
template<typename REQ, typename RESP=response>
struct axi_extension :
        public common,
        public REQ,
        public RESP
{

    void reset();
    void reset(const REQ *);

    void add_to_response_array(response&);
    const std::vector<response> &get_response_array() const;
  std::vector<response> &get_response_array();

    void set_response_array_complete(bool = true);
    bool is_response_array_complete();
};
```

The definition as template allows to easily create an AXI3 as well as an AXI4 and an ACE extensions without duplicated declarations. This allows even to easily add an AXI4-Lite extension. Purpose of the response array is to collect the responses of all read data beats for further use e.g. in protocol state machine implementations.

7.2 Phases Declarations and Protocol traits

According to the specified protocol modeling 4 additional non-ignorable phases need to be defined:

```
// additional AXI/ACE phases
DECLARE_EXTENDED_PHASE(BEGIN_PARTIAL_REQ);
DECLARE_EXTENDED_PHASE(END_PARTIAL_REQ);
DECLARE_EXTENDED_PHASE(BEGIN_PARTIAL_RESP);
DECLARE_EXTENDED_PHASE(END_PARTIAL_RESP);
DECLARE_EXTENDED_PHASE(ACK);
```

Since these are non-ignorable a specific protocol traits needs to be defined to comply with the TLM2.0 LRM:

```
using axi_payload = tlm::tlm_generic_payload;
using axi_phase = tlm::tlm_phase;

// axi protocol traits class
struct axi_protocol_types {
    typedef axi_payload tlm_payload_type;
    typedef axi_phase tlm_phase_type;
};
```

7.3 Socket Interfaces and Sockets

The socket interface is for the backward path extended to allow snooping in a blocking manner. The other interfaces are re-used from the tlm base protocol specification:

```
// AXI socket interfaces
template <typename TRANS = tlm::tlm_generic_payload>
class ace_bw_blocking_transport_if : public virtual sc_core::sc_interface {
public:
  virtual void b_snoop(TRANS& trans, sc_core::sc_time& t) = 0;
};

// alias declaration for the forward interface
template <typename TYPES = tlm::tlm_base_protocol_types>
using axi_fw_transport_if = tlm::tlm_fw_transport_if<TYPES>;
// alias declaration for the backward interface:
template <typename TYPES = tlm::tlm_base_protocol_types>
using axi_bw_transport_if = tlm::tlm_bw_transport_if<TYPES>;
// alias declaration for the ACE forward interface
template <typename TYPES = tlm::tlm_base_protocol_types>
using ace_fw_transport_if = tlm::tlm_fw_transport_if<TYPES>;
// the ACE backward interface:
template <typename TYPES = tlm::tlm_base_protocol_types>
class ace_bw_transport_if
  : public tlm::tlm_bw_transport_if<TYPES>
  , public virtual ace_bw_blocking_transport_if<
          typename TYPES::tlm_payload_type>
{};
```

Based on the definitions so far the initiator and target sockets are declared as follows:

```
template <unsigned int BUSWIDTH = 32
        , typename TYPES = axi_protocol_types
        , int N = 1
        , sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND>
struct axi_initiator_socket :
        public tlm::tlm_base_initiator_socket <BUSWIDTH,
                axi_fw_transport_if<TYPES>,
                axi_bw_transport_if<TYPES>, N, POL>
{
    using base_type =
            tlm::tlm_base_initiator_socket <BUSWIDTH,
                    axi_fw_transport_if<TYPES>,
                    axi_bw_transport_if<TYPES>, N, POL>;

    axi_initiator_socket() : base_type()
    { }

    explicit axi_initiator_socket(const char* name) :base_type(name)
    { }

    const char* kind() const override {
        return "axi_initiator_socket";
    }
};

template <unsigned int BUSWIDTH = 32
        , typename TYPES = axi_protocol_types
        , int N = 1
        , sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND>
struct ace_initiator_socket :
        public tlm::tlm_base_initiator_socket <BUSWIDTH,
                        ace_fw_transport_if<TYPES>,
                        ace_bw_transport_if<TYPES>, N, POL>
{
    using base_type =
                tlm::tlm_base_initiator_socket <BUSWIDTH,
                        ace_fw_transport_if<TYPES>,
                        ace_bw_transport_if<TYPES>, N, POL>;

    ace_initiator_socket() : base_type()
    { }

    explicit ace_initiator_socket(const char* name) :base_type(name)
    { }

    const char* kind() const override {
        return "ace_initiator_socket";
    }
};
```

```
template <unsigned int BUSWIDTH = 32
        , typename TYPES = axi_protocol_types
        , int N = 1
        , sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND>
struct axi_target_socket :
        public tlm::tlm_base_initiator_socket <BUSWIDTH,
                axi_fw_transport_if<TYPES>,
                axi_bw_transport_if<TYPES>, N ,POL>
{
    using base_type =
            tlm::tlm_base_initiator_socket <BUSWIDTH,
                    axi_fw_transport_if<TYPES>,
                    axi_bw_transport_if<TYPES>, N, POL>;

    axi_target_socket() : base_type()
    { }

    explicit axi_target_socket(const char* name) : base_type(name)
    { }

    const char* kind() const override {
        return "axi_target_socket";
    }
};

template <unsigned int BUSWIDTH = 32
        , typename TYPES = axi_protocol_types
        , int N = 1
        , sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND>
struct ace_target_socket :
        public tlm::tlm_base_target_socket <BUSWIDTH,
                        ace_fw_transport_if<TYPES>,
                        ace_bw_transport_if<TYPES>, N ,POL>
{
    using base_type =
                tlm::tlm_base_target_socket <BUSWIDTH,
                        ace_fw_transport_if<TYPES>,
                        ace_bw_transport_if<TYPES>, N, POL>;

    ace_target_socket() : base_type()
    { }

    explicit ace_target_socket(const char* name) : base_type(name)
    { }

    const char* kind() const override {
        return "ace_target_socket";
    }
};
```

8 Recommended Additions

8.1 Convenience Layer

The package provides a convenience layer based on the described data structures. part of the convenience layer are initiator and target socket handling the aforementioned protocol.

8.2 Transaction Tracing

Debugging complex communication schemes and structures is difficult and needs appropriate support for vizualization.

Therefor a tracing solution based of the SystemC Verification Library (SCV) should be implemented. This way the tracing is compatible with major EDA solutions as they build on top of the same interface.

8.3 Protocol Checker

To make sure the rules of this spec, the TLM2.0 LRM and the 'AMBA AXI and ACE Protocol Specification' are obeyed a protocol checker shoudl be implemented flagging erronous states and sequences. Used in conjunction with the transaction tracing it can refer to and highlight the offending transactions.