Introduction to Accellera TLM 2.0

Aravinda Thimmapuram

10th Sept 2015
Agenda

- Introduction to TLM and TLM 2.0 standard
- Case study of modelling a bus protocol using TLM 2.0
Scope and Pre-requisites

- Introductory tutorial on Accellera TLM 2.0
- Assumes audience are familiar with SystemC concepts
  - Modules
  - Ports and exports
  - Process
  - Channel
  - Interface
  - event
Introduction to Transaction Level Modeling

Pin accurate, cycle accurate

Simulate every event

RTL

Functional Model

Transaction level - function call

write(address, data)

100-10,000 X faster simulation
Reasons for Using TLM

- **Software development**
  - Accelerates product release schedule
- **Architectural modeling**
  - Fast enough
  - Ready before RTL
- **Hardware verification**
  - TLM = golden model

Diagram:

- Test bench
- Firmware / software
- RTL
- TLM

Red arrows indicate flow between stages.
Typical Use Cases for TLM

- Represents key architectural components of hardware platform
- Architectural exploration, performance modeling
- Software execution on virtual model of hardware platform
- Golden model for hardware functional verification
- Available before RTL
- Simulates much faster than RTL

Early!
Fast!
Why Do We Need TLM Standard?

Mismatch
a. Port types
b. API
c. Function params
d. Protocols
Interopability

Disadvantages
a. Develop time
b. Maintenance
c. Simulation speed
OSCI TLM Development

- **Apr 2005**
  - TLM -1.0
  - put, get and transport request-response interfaces

- **Dec 2006**
  - TLM-2.0-draft -1
  - Generic payload

- **Nov 2007**
  - TLM 2.0-draft-2
  - nb_transport
  - New payload & extensions

- **Jun 2008**
  - TLM-2.0
  - Unified interfaces and sockets
TLM-2 Requirements

- Transaction-level memory-mapped bus modeling
- Register accurate, functionally complete
- Fast enough to boot software O/S in seconds
- Loosely-timed and approximately-timed modeling
- Interoperable API for memory-mapped bus modeling
- Generic payload and extension mechanism
- Avoid adapters where possible

See TLM_2_0_requirements.pdf
Acronyms

- Initiator – module that initiates new transactions
- Target – module that responds to transactions initiated by initiator
- Transaction – data structure (C++ object) passed between initiators and targets using function calls
Use Cases, Coding Styles and Mechanisms

**Use cases**

- Software development
- Software performance
- Architectural analysis
- Hardware verification

**TLM-2 Coding styles**

- Loosely-timed
- Approximately-timed

**Mechanisms**

- Blocking interface
- DMI
- Quantum
- Sockets
- Generic payload
- Phases
- Non-blocking interface
Coding Styles

- **Loosely-timed** = as fast as possible
  - Only sufficient timing detail to boot O/S and run multi-core systems
  - Processes can run ahead of simulation time (temporal decoupling)
  - Each transaction has 2 timing points: begin and end
  - Uses direct memory interface (DMI)

- **Approximately-timed** = just accurate enough for performance modeling
  - *aka* cycle-approximate or cycle-count-accurate
  - Sufficient for architectural exploration
  - Processes run in lock-step with simulation time
  - Each transaction has 4 timing points (extensible)
The TLM 2.0 Classes

Interoperability layer for bus modeling

- Generic payload
- Phases
- Initiator and target sockets

TLM-2 core interfaces:
- Blocking transport interface
- Non-blocking transport interface
- Direct memory interface
- Debug transport interface

Utilities:
- Convenience sockets
- Payload event queues
- Quantum keeper
- Instance-specific extn

Analysis ports

Analysis interface

IEEE 166™ SystemC

TLM-1 standard
Interoperability Layer

1. Core interfaces and sockets

2. Generic payload
   - Command
   - Address
   - Data
   - Byte enables
   - Response status
   - Extensions

3. Base protocol
   - BEGIN_REQ
   - END_REQ
   - BEGIN_RESP
   - END_RESP

Maximal interoperability for memory-mapped bus models
Loosely Timed

Use cases

- Software development
- Software performance
- Architectural analysis
- Hardware verification

TLM-2 Coding styles

- Loosely-timed
- Approximately-timed

Mechanisms

- Blocking interface
- DMI
- Quantum
- Sockets
- Generic payload
- Phases
- Non-blocking interface
Initiator and Target Sockets

Initiator socket

- b_transport()
- nb_transport_fw()
- get_direct_mem_ptr()
- transport_dbg()

Target socket

- nb_transport_bw()
- invalidate_direct_mem_ptr()

Sockets provide fw and bw paths and group interfaces
Sockets

- Transaction involve communication in both direction

- Initiator and Target Sockets makes it convenient
Sockets

- Implemented as SystemC channels

```cpp
struct Initiator : sc_module, tlm::tlm_bw_transport_if<> {}
struct Target : sc_module, tlm::tlm_fw_transport_if<> {}
```
Sockets

• Example Initiator module

```cpp
struct initiator : sc_module, tlm::tlm_bw_transport_if<> {
  tlm::tlm_initiator_socket<> init_socket;

  ...

  // Must implement all backward calls
  tlm::tlm_sync_enum nb_transport_bw(...) {}
  void invalidate_direct_mem_ptr(...) {}
};
```
Sockets

- Example Target module

```cpp
struct target : sc_module, tlm::tlm_fw_transport_if<> {
    tlm::tlm_target_socket<> target_socket;

    ... // Must implement all forward calls
    void b_transport(...) {} 
    tlm::tlm_sync_enum nb_transport_fw(...) { } 
    bool get_direct_mem_ptr(...) {} 
    unsigned int transport_dbg(...) { } 
};
```
Benefit of Sockets

- Group the transport, DMI and debug transport interfaces
- Bind forward and backward paths with a single call
- Strong connection checking
- Have a bus width parameter

- Using core interfaces without sockets is not recommended
TLM-2 Core Interfaces - Transport

tlm_blocking_transport_if

```c
void b_transport( TRANS& , sc_time& ) ;
```

tlm_fw_nonblocking_transport_if

```c
tlm_sync_enum nb_transport_fw( TRANS& , PHASE& , sc_time& ) ;
```

tlm_bw_nonblocking_transport_if

```c
tlm_sync_enum nb_transport_bw( TRANS& , PHASE& , sc_time& ) ;
```
void invalidate_direct_mem_ptr(sc_dt::uint64 start_range, sc_dt::uint64 end_range);

bool get_direct_mem_ptr(TRANS& trans, tlm_dmi& dmi_data);

unsigned int transport_dbg(TRANS& trans);

May all use the generic payload transaction type
Blocking Transport

```
template <typename TRANS = tlm_generic_payload>
class tlm_blocking_transport_if : public virtual sc_core::sc_interface {
public:
    virtual void b_transport( TRANS& trans, sc_core::sc_time& t ) = 0;
};
```

Transaction type

- Transaction object
- Timing annotation
LT Mechanisms for Increasing Simulation Speed

- Impediments to simulation speed
  - Context switches
    - Reduce context switches with temporal decoupling
  - Function call hierarchy
    - By-pass bus with direct access to memory (DMI)
Blocking Transport – With Wait

Initiator

Simulation time = 100ns

Call

b_transport(t, 0ns)

Target

wait(40ns)

Simulation time = 140ns

Call

b_transport(t, 0ns)

Return

Simulation time = 180ns

Return

Initiator is blocked until return from b_transport
Blocking Transport – With Timing Annotation

Simulation time = 100ns

Local time = 100 + 40ns

Call b_transport(t, 0ns)

Local time = 100 + 0ns

Return

Local time = 100 + 40ns

Call b_transport(t, 40ns)

Return

Local time = 100 + 80ns

wait(80ns)

Simulation time = 180ns

Fewer context switches means higher simulation speed
DMI and Debug Transport

- **Direct Memory Interface**
  - Gives an initiator a direct pointer to memory in a target, e.g. an ISS
  - By-passes the sockets and transport calls
  - Read or write access by default
  - Extensions may permit other kinds of access, e.g. security mode
  - Target responsible for invalidating pointer

- **Debug Transport Interface**
  - Gives an initiator debug access to memory in a target
  - Delay-free
  - Side-effect-free

- May share transactions with transport interface
Non-blocking Transport

```cpp
enum tlm_sync_enum { TLM_ACCEPTED, TLM_UPDATED, TLM_COMPLETED };

template <
type name TRANS = tlm_generic_payload,
type name PHASE = tlm_phase>
class tlm_fw_nonblocking_transport_if : public virtual sc_core::sc_interface {
public:
  virtual tlm_sync_enum nb_transport( TRANS& trans, 
                                       PHASE& phase, 
                                       sc_core::sc_time& t ) = 0;
};
```

Trans, phase and time arguments set by caller and modified by callee
tlm_sync_enum

- **TLM_ACCEPTED**
  - Transaction, phase and timing arguments unmodified (ignored) on return
  - Target may respond later (depending on protocol)

- **TLM_UPDATED**
  - Transaction, phase and timing arguments updated (used) on return
  - Target has advanced the protocol state machine to the next state

- **TLM_COMPLETED**
  - Transaction, phase and timing arguments updated (used) on return
  - Target has advanced the protocol state machine straight to the final phase
Phases

• BEGIN_REQ
  • Initiator acquires the bus
  • Connection becomes busy and clocks further requests

• END_REQ
  • Target ‘accepts’ the request
  • Bus free to start additional requests

• BEGIN_RESP
  • Target acquires the bus to provide response
  • Bus becomes ‘busy’

• END_RESP
  • Initiator acknowledges response
  • Bus is freed
  • Payload reference freed-up
Using the Backward Path

<table>
<thead>
<tr>
<th>Phase</th>
<th>Initiator</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN_REQ</td>
<td>Call</td>
<td>TLM_ACCEPTED, -, -</td>
</tr>
<tr>
<td></td>
<td>Simulation time = 100ns</td>
<td></td>
</tr>
<tr>
<td>END_REQ</td>
<td>-, END_REQ, 0ns</td>
<td>Call</td>
</tr>
<tr>
<td></td>
<td>Simulation time = 110ns</td>
<td>Return</td>
</tr>
<tr>
<td>BEGIN_RESP</td>
<td>-, BEGIN_RESP, 0ns</td>
<td>Call</td>
</tr>
<tr>
<td></td>
<td>Simulation time = 120ns</td>
<td>Return</td>
</tr>
<tr>
<td>END_RESP</td>
<td>-, END_RESP, 0ns</td>
<td>Call</td>
</tr>
<tr>
<td></td>
<td>Simulation time = 130ns</td>
<td>Return</td>
</tr>
</tbody>
</table>

Transaction accepted now, caller asked to wait
Using the Return Path

Phase | Initiator | Target
--- | --- | ---
BEGIN_REQ | Simulation time = 100ns | -, BEGIN_REQ, 0ns
END_REQ | Simulation time = 110ns | TLM_UPDATED, END_REQ, 10ns
BEGIN_RESP | Simulation time = 150ns | -, BEGIN_RESP, 0ns
END_RESP | Simulation time = 155ns | Call, Return

Callee annotates delay to next transition, caller waits
Early Completion

Phase | Initiator | Target

BEGIN_REQ | Call | -, BEGIN_REQ, 0ns

END_RESP | Simulation time = 110ns | TLM_COMPLETED, -, 10ns

Simulation time = 100ns

Callee annotates delay to next transition, caller waits
Timing Annotation

Phase: BEGIN_REQ

Initiator:

Simulation time = 100ns

Call

Simulation time = 110ns

TLM_ACCEPTED, -, -

Return

Simulation time = 125ns

Payload

Event

Queue

Target:

Payload

Event

Queue

Simulation time = 135ns
The Generic Payload

- Typical attributes of memory-mapped busses
  - command, address, data, byte enables, single word transfers, burst transfers, streaming, response status

- Off-the-shelf general purpose payload
  - for abstract bus modeling
  - *ignorable* extensions allow full interoperability

- Used to model specific bus protocols
  - mandatory static extensions
  - compile-time type checking to avoid incompatibility
  - low implementation cost when bridging protocols

Specific protocols can use the same generic payload machinery
# Generic Payload Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Modifiable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command</td>
<td>tlm_command</td>
<td>No</td>
</tr>
<tr>
<td>Address</td>
<td>uint64</td>
<td>Interconnect only</td>
</tr>
<tr>
<td>Data pointer</td>
<td>unsigned char*</td>
<td>No (array – yes)</td>
</tr>
<tr>
<td>Data length</td>
<td>unsigned int</td>
<td>No</td>
</tr>
<tr>
<td>Byte enable pointer</td>
<td>unsigned char*</td>
<td>No (array – yes)</td>
</tr>
<tr>
<td>Byte enable length</td>
<td>unsigned int</td>
<td>No</td>
</tr>
<tr>
<td>Streaming width</td>
<td>unsigned int</td>
<td>No</td>
</tr>
<tr>
<td>DMI hint</td>
<td>bool</td>
<td>Yes</td>
</tr>
<tr>
<td>Response status</td>
<td>tlm_response_status</td>
<td>Target only</td>
</tr>
<tr>
<td>Extensions</td>
<td>(tlm_extension_base*)[]</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Try DMI!

Consider memory management.
enum tlm_command {
  TLM_READ_COMMAND,
  TLM_WRITE_COMMAND,
  TLM_IGNORE_COMMAND
};

const tlm_command get_command() ;  
void set_command( const tlm_command command ) ;

const sc_dt::uint64 get_address() ;
void set_address( const sc_dt::uint64 address ) ;

const unsigned char* get_data_ptr() ;
void set_data_ptr( unsigned char* data ) ;

const unsigned int get_data_length() ;
void set_data_length( const unsigned int length ) ;

Copy from target to data array
Copy from data array to target
Neither, but may use extensions

Data array owned by initiator
Number of bytes in data array
## Response Status

<table>
<thead>
<tr>
<th>enum tlm_response_status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLM_OK_RESPONSE</td>
<td>Successful</td>
</tr>
<tr>
<td>TLM_INCOMPLETE_RESPONSE</td>
<td>Transaction not delivered to target. (Default)</td>
</tr>
<tr>
<td>TLM_ADDRESS_ERROR_RESPONSE</td>
<td>Unable to act on address</td>
</tr>
<tr>
<td>TLM_COMMAND_ERROR_RESPONSE</td>
<td>Unable to execute command</td>
</tr>
<tr>
<td>TLM_BURST_ERROR_RESPONSE</td>
<td>Unable to act on data length or streaming width</td>
</tr>
<tr>
<td>TLM_BYTE_ENABLE_ERROR_RESPONSE</td>
<td>Unable to act on byte enable</td>
</tr>
<tr>
<td>TLM_GENERIC_ERROR_RESPONSE</td>
<td>Any other error</td>
</tr>
</tbody>
</table>
Generics Payload Example 1

```c
void thread_process() {  // The initiator
    tlm::tlm_generic_payload trans;
    sc_time delay = SC_ZERO_TIME;

    trans.set_command( tlm::TLM_WRITE_COMMAND );
    trans.set_data_length( 4 );
    trans.set_byte_enable_ptr( 0 );
    trans.set_streaming_width( 4 );

    for ( int i = 0; i < RUN_LENGTH; i += 4 ) {
        int word = i;
        trans.set_address( i );
        trans.set_data_ptr((unsigned char*)( &word ));
        trans.set_response_status( tlm::TLM_INCOMPLETE_RESPONSE );

        init_socket->b_transport( trans, delay );

        if ( trans.get_response_status() <= 0 )
            SC_REPORT_ERROR("TLM2", trans.get_response_string().c_str());
    }
```

Would usually pool transactions
Generic Payload Extension Methods

- Generic payload has an array-of-pointers to extensions
- One pointer per extension type
- Every transaction can potentially carry every extension type
- Flexible mechanism
Extension Example

struct my_extension : tlm_extension<my_extension>
{
    my_extension() : id(0) {}
    tlm_extension_base* clone() const { ... }
    virtual void copy_from(tlm_extension_base const &ext) { ... }
    int id;
};
...

tlm_generic_payload* trans = new tlm_generic_payload( mem_mgr );
trans->acquire();

my_extension* ext = new my_extension;
ext->id = 1;
trans.set_extension( ext );

socket->nb_transport_fw( *trans, phase, delay );
trans.release_extension<my_extension>();

trans->release();
Base Protocol - Coding Styles

- Loosely-timed is typically
  - Blocking transport interface, forward and return path
  - 2 timing points
  - Temporal decoupling and the quantum keeper
  - Direct memory interface

- Approximately-timed is typically
  - Non-blocking transport interface, forward and backward paths
  - 4 phases
  - Payload event queues

- Loosely-timed and approximately-timed are only coding styles
- The base protocol defines rules for phases and call order
Base Protocol Rules

- Base protocol phases
  - \texttt{BEGIN\_REQ} $\rightarrow$ \texttt{END\_REQ} $\rightarrow$ \texttt{BEGIN\_RESP} $\rightarrow$ \texttt{END\_RESP}
  - Must occur in non-decreasing simulation time order
  - Only permitted one outstanding request or response per socket
  - Phase must change with each call (other than ignorable phases)
  - May complete early

- Generic payload memory management rules
- Extensions must be ignorable
- Target is obliged to handle mixed \texttt{b\_transport} / \texttt{nb\_transport}
- Write response must come from target
#include "tlm.h" → TLM 2.0 interoperability standard
#include "tlm_utils/simple_initiator_socket.h" TLM Utility
#include "tlm_utils/simple_target_socket.h"

struct simple_initiator : sc_module {
    tlm_utils::simple_initiator_socket<simple_initiator> init_socket;
    SC_CTOR(simple_initiator) : init_socket("init_socket") { ...
};

struct memory : public sc_module {
    tlm_utils::simple_target_socket<memory> mem_socket;
    SC_CTOR(memory):mem_socket("mem_socket") { ...
};
```c
struct Top : sc_module {
    simple_initiator *initiator;
    memory *mem;

    SC_CTOR(Top) {
        initiator = new simple_initiator("simple_initiator");
        mem = new memory("memory");
        initiator->init_socket.bind(mem->mem_socket);
    }
};

int sc_main() {
    Top top("top");
    sc_start();
    return 0;
}
```
struct simple_initiator : sc_module {
    tlm_utils::simple_initiator_socket<simple_initiator> init_socket;
    SC_CTOR(simple_initiator) : init_socket("init_socket") {
        SC_THREAD(test_process);
    }
    void test_process() {
        while(1) {
            tlm::tlm_generic_payload *gp = new tlm::tlm_generic_payload;
            sc_time delay;
            // Set different fields of GP
            init_socket->b_transport(*gp, delay);
            ....
        }
    }
}
struct memory : sc_module {
    tlm_utils::simple_target_socket<memory> mem_socket;
    SC_CTOR(memory):mem_socket("mem_scoket") {
        ...
        mem_socket.register_b_transport(this, &memory::b_transport);
    }
    void b_transport(tlm::tlm_generic_payload &gp, sc_time &delay) { ...}
};
Summary

• TLM 2.0 standard addresses
  – Interoperability
  – Simulation speed
  – Bus based protocols

• TLM 2.0 Mechanisms summary
  – Sockets
  – Core interfaces
    • Blocking, non-blocking, DMI, and debug interfaces
  – Generic payload
  – Base protocol
    • LT for simulation speed
    • AT for accuracy and analysis
Usage of TLM 2.0 for bus protocol modelling

Aravinda Thimmapuram

26th Sep 2014
Introduction

• TLM 2.0 provides necessary constructs for the modelling of memory-mapped bus models
  – core interfaces and sockets
  – generic payload
  – base protocol
• Presents a case study of modelling an in-house bus protocol using TLM 2.0, illustrating
  – usage of the TLM 2.0 mechanisms
  – design choices available
  – trade-offs in terms of interoperability
Protocol Description

Initiator

Target

- cmd_valid
- cmd
- pop_cmd
- data_valid
- data
- pop_data

- cmd_valid
- cmd
- pop_cmd
- data_valid
- data
- pop_data
Protocol Description

• Protocol consists of two major parts
  – Request
  – Response

• Both Request and Response contain command (Cmd) and Data

• *Cmd* is a multi-bit packet with various control information (like address, Id ...)
  – can accommodate both request and response

• *Data* is also multi-bit packet which contains the raw data and data valid bits
  – Same structure can accommodate both write data and read response
Read Request/Response Illustration

Master

cmd_valid (Read request)

cmd (Read address, ID, etc)

pop_cmd (read req accepted)

Slave

cmd_valid (Ready with read response)

cmd (Read address, src/dst id etc)

pop_cmd (Ready to accept Read data)

data_valid (Ready with read data)

data (read data)

pop_data (Read data accepted)
Mapping to TLM 2.0 – Coding Style

Approximately Timed (AT) coding style is chosen as:

a. Use case of performance modelling requires higher accuracy and the need for more than two phases
Sockets

a. We use the standard socket classes - *tlm_initiator_socket* and *tlm_target_socket* as:
   i. They are part of interoperability layer

b. We are not using the convenient sockets as:
   i. We need support for hierarchical binding
   ii. We don’t want the automatic conversion from *b_transport* and *nb_transport*
Core interfaces

a. Since we are following AT coding style, we will be using the non-blocking transport interface
   a. `tlm_sync_enum nb_transport_fw( TRANS& , PHASE& , sc_time& );`
   b. `tlm_sync_enum nb_transport_bw( TRANS& , PHASE& , sc_time& );`

b. We provide *error response* for the following
   a. Blocking transport
   b. Debug calls
   c. DMI support
Generic Payload – Option 1

a. To model the data part of the protocol - we use the \textit{data pointer} and \textit{data length}

b. To model the \textit{cmd} part of the protocol – we make use of GP ignorable extensions
   a. \textit{tlm\_command\_type} does not suffice to handle all fields of protocol field \textit{cmd}

c. To model the protocol response we make use of \textit{Response status (tlm\_response\_status} type) fields of the GP

d. To make the extensions ignorable, set the GP \textit{Command} and \textit{address} fields with appropriate values from the protocol \textit{Cmd} field

Excellent interoperability
Generic Payload – Option 2

a. To model the data part of the protocol - we use the *data pointer, data_length* fields of the GP

b. To model the *cmd* part of the protocol – we make use of GP *mandatory* extensions

c. Define a new protocol traits class

```c
struct my_protocol_types {
    typedef tlm_generic_payload tlm_payload_type;
    typedef tlm_phase tlm_phase_type;
};

tlm_initiator_socket<32, my_protocol_types> my_socket;
```

Can connect to models with custom payload type

Only
Generic Payload – Option 3

a. Define a new custom payload type *un-related* to generic payload

```c
struct my_payload {
    my_payload();
    ...
};
```

a. Define a new protocol traits class

```c
struct my_protocol_types {
    typedef my_payload    tlm_payload_type;
    typedef tlm_phase     tlm_phase_type;
};
```

```c
tlm_initiator_socket<32, my_protocol_types > my_socket;
```

Sacrifice interoperability; you are on your own.
Base Protocol – Option1

a. Since we need to send *Data* and *Cmd* separately (for write), 4 phases of AT does not suffice

b. We define two new *ignorable* phase extensions – DATA_VALID and POP_DATA

```
DECLARE_EXTENDED_PHASE(DATA_VALID);
DECLARE_EXTENDED_PHASE(POP_DATA);
```

c. With this option we have to ensure that the Data is present with the BEGIN_REQ phase itself, so that the target that ignores this extension can still work

Excellent interoperability
Base Protocol – Option2

a. Since we need to send *Data* and *Cmd* separately, 4 timing points of AT does not suffice

b. We define a new protocol traits class with new phase class:

```cpp
struct my_protocol_types {
    typedef tlm_generic_payload tlm_payload_type;
    typedef my_tlm_phase tlm_phase_type;
};

tlm_initiator_socket<32, my_protocol_types > my_socket;
```

Sacrifice interoperability; you are on your own
Memory management

a. TLM 2.0 specifies memory manager interface with hooks to implement custom memory manager

b. We use own memory manager by overloading `free` and providing `allocate` function to allocate from the `free list`

i. Optimizes the heap memory usage
Putting it all together

• using AT coding style
  – using non blocking transport
• Using standard socket classes
• Using GP with ignorable extensions
• Using base protocol with ignorable phases
• Using own memory manager conforming to memory manager interface

protocol mapping with good interoperability
Putting it all together - READ 2/3
Putting it all together - WRITE

```
Master

nb_transport_fw

| tlm_phase = BEGIN_REQ (= cmd_Valid) |
| Payload extensions = CMD |
| rtn = TLM_ACCEPTED |

nb_transport_bw

| tlm_phase = END_REQ (= pop_cmd) |
| rtn = TLM_ACCEPTED |

Slave

nb_transport_fw

| tlm_phase = DATA_VALID(= data_valid) |
| rtn = TLM_ACCEPTED |

nb_transport_bw

| tlm_phase = POP_DATA(= pop_data) |
| rtn = TLM_ACCEPTED |

nb_transport_bw

| tlm_phase = BEGIN_RESP (= cmd_valid) |
| rtn = TLM_ACCEPTED |

nb_transport_fw

| tlm_phase = END_RESP (= pop_cmd) |
| rtn = TLM_ACCEPTED |
```
Summary

• We have presented use case of modeling a protocol using TLM 2.0 AT
• Paper presents different design choices available and trade-off in terms of interoperability
Thanks!
Blocking versus Non-blocking Transport

- **Blocking transport interface**
  - Includes timing annotation
  - Typically used with loosely-timed coding style
  - Forward path only

- **Non-blocking transport interface**
  - Includes timing annotation and transaction phases
  - Typically used with approximately-timed coding style
  - Called on forward and backward paths

- Share the same transaction type for interoperability
**The Time Quantum**

**Initiator**

- **Simulation time = 1us**
  - **Local time offset**
    - +950ns: \( \text{b}_\text{transport}(t, 950\text{ns}) \)
    - +970ns: \( \text{b}_\text{transport}(t, 970\text{ns}) \)  \( \text{Return} \)
    - +990ns: \( \text{b}_\text{transport}(t, 990\text{ns}) \)  \( \text{Return} \)
    - +1010ns: \( \text{b}_\text{transport}(t, 1010\text{ns}) \)  \( \text{Return} \)

- **wait(1us)**
  - **Simulation time = 2us**
    - +0ns: \( \text{b}_\text{transport}(t, 0\text{ns}) \)

**Target**

- **Quantum = 1us**
The Quantum Keeper (tlm_quantumkeeper)

- Quantum is user-configurable

- Processes can check local time against quantum
Direct Memory Interface

status = get_direct_mem_ptr( transaction, dmi_data );

invalidate_direct_mem_ptr( start_range, end_range );

Transport, DMI and debug may all use the generic payload

Interconnect may modify address and invalidated range
DMI Transaction and DMI Data

DMI Transaction

Requests read or write access
For a given address
Permits extensions

class tlm_dmi

unsigned char* dmi_ptr
uint64 dmi_start_address
uint64 dmi_end_address
dmi_type_e dmi_type;
sc_time read_latency
sc_time write_latency

Direct memory pointer
Region granted for given access type
Read, write or read/write
Latencies to be observed by initiator
num_bytes = transport_dbg( transaction );

tlm_transport_dbg_if

Uses forward path only

Interconnect may modify address, target reads or writes data
class tlm_generic_payload

class tlm_generic_payload {
public:

    // Constructors, memory management
    tlm_generic_payload () ;
    tlm_generic_payload(tlm_mm_interface& mm) ;
    virtual ~tlm_generic_payload () ;
    void reset();

    void set_mm(tlm_mm_interface* mm);
    bool has_mm();
    void acquire();
    void release();
    int get_ref_count();

    void deep_copy_into(tlm_generic_payload& other) const;

...}

Not a template

Construct & set mm
Frees all extensions
Frees mm’d extensions

mm is optional

Incr reference count
Decr reference count, 0 => free trans
Memory Management Rules

- b_transport – memory managed by initiator, or reference counting (set_mm)
- nb_transport – reference counting only
  - Reference counting requires heap allocation
  - Transaction automatically freed when reference count == 0
  - free() can be overridden in memory manager for transactions
  - free() can be overridden for extensions

- When b_transport calls nb_transport, must add reference counting
  - Can only return when reference count == 0
- b_transport can check for reference counting, or assume it could be present
The Standard Error Response

- A target shall either
  - Execute the command and set TLM_OK_RESPONSE
  - Set the response status attribute to an error response
  - Call the SystemC report handler and set TLM_OK_RESPONSE
Base Protocol and tlm_phase

- The base protocol = tlm_generic_payload + tlm_phase
- tlm_phase has 4 phases, but can be extended to add new phases

```cpp
enum tlm_phase_enum { UNINITIALIZED_PHASE = 0,
                      BEGIN_REQ=1, END_REQ, BEGIN_RESP, END_RESP };

class tlm_phase {
public:
    tlm_phase();
    tlm_phase( unsigned int id );
    tlm_phase( const tlm_phase_enum& standard );
    tlm_phase& operator= ( const tlm_phase_enum& standard );
    operator unsigned int() const;
};

#define DECLARE_EXTENDED_PHASE(name_arg) \
    class tlm_phase_##name_arg : public tlm::tlm_phase { \
        ...
```