SystemC Transaction Level Modeling Standards and Methodology Guidelines

The implementation of TLM 2.0

by

John Aynsley
CTO
Doulos
The Implementation of TLM 2.0

- Introduction, Definitions and Principles
- Analysis Interface
- Timing Annotation
- Generic Payloads
- Pass-by-Pointer
Layered Standards

- Application
- SoC IP
- TLM 2.x
  - OSCI Draft, Dec 06
  - OSCI Std, summer 05
  - IEEE Std 1666
  - ANSI Std
  - SystemC
  - C++
  - Interoperability Layer
  - Transport Layer
TLM 1.0 = Message Passing
Improving on TLM 1.0

- Agreeing on the message format (tlm_bus)
- Timing annotation
- Efficiency for passing long messages (pass-by-pointer)
OSCI TLM 2.x Agenda

- Timing annotation on core interfaces
- Generic payloads for PV and PVT bus models (tlm_bus)
- Analysis interface

- Cycle accurate modeling
- Debug interface
- Configuration interface
- Interrupt modeling
- Memory map and register modeling
TLM Roadmap

2.0 standard draft for Public review
- Timed TLM core i/f
- Analysis interface
- PV / PVT payloads
- Examples

2.0 draft upgrade
- Including results from initial WG analysis of users comments

2.0 standard
Official release
- Including results from final WG analysis of users feedback
- Begin LRM

IP/SoC’06
DATE’07
DAC’07

Dec.4 ‘06    Feb.9 ‘07
Users feedback
Definition of Terms 1

TLM Initiator Port

TLM Target Export

System Transaction

TLM Initiator Port

TLM Target Export

System Initiator

TLM Initiator

TLM Target

TLM Transactions

TLM Initiator Port

TLM Target Export

TLM Initiator

TLM Target

TLM Target Export

TLM Initiator Port

TLM Target

TLM Target
Definition of Terms 2

- A system component may be initiator and target

- Initiator = Master
- Target = Slave

- TLM Core Interface = unidirectional (put/get) and bidirectional (transport) interfaces in the TLM kit

- TLM Bus = generic request and response payloads
TLM 2.0 Draft 1 Kit

TLM2.0 examples
Includes Powerpoint

History
Empty
Bidirectional Transport Interface

```cpp
template < typename REQ , typename RSP >
class tlm_transport_if : public virtual sc_interface {
public:
    virtual RSP transport( const REQ & ) = 0;

    virtual void transport( const REQ &req , RSP &rsp ) {
        rsp = transport( req );
    }
};
```

- Note obligation to use effective pass-by-value
Effective Pass-by-Value

Pass by value

```cpp
T get();
```

- Obligation not to modify actual argument during call
- Obligation not to use reference after the call (must copy value)

Pass by const reference

```cpp
void put( const T& );
```

Pass by reference

```cpp
void get( T& );
```

- Obligation not to read or modify actual argument during call
- Obligation not to read reference during the call
TLM transaction only valid for duration of function call

(system transaction sliced-and-diced into independent chunks) ...

... except for pass-by-pointer mode
The Implementation of TLM 2.0

- Introduction, Definitions and Principles
- Analysis Interface
- Timing Annotation
- Generic Payloads
- Pass-by-Pointer
Analysis Ports

- Non-intrusive monitoring of transactions going through TLM ports
  - A SystemC implementation of the observer pattern
- Can connect zero, one or many observers to a single analysis port
- Non-blocking, non-negotiated interface

Slide derived from OSCI presentation  TLM_2.0_Overview.pdf
**analysis_if and analysis_port**

```cpp
template < typename T >
class analysis_if : public virtual sc_interface {
public:
  virtual void write( const T &t ) = 0;
};

template < typename T>
class analysis_port :
  public sc_object, public virtual analysis_if< T > {
  ...
  void operator() ( analysis_if<T> &_if );
  void bind ( analysis_if<T> &_if );
  bool unbind ( analysis_if<T> &_if );

  void write( const T &t );
};
```
The Implementation of TLM 2.0

- Introduction, Definitions and Principles
- Analysis Interface
- Timing Annotation
- Generic Payloads
- Pass-by-Pointer
# TLM1 Core Interfaces

<table>
<thead>
<tr>
<th></th>
<th>Untimed</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bidirectional</strong></td>
<td></td>
</tr>
<tr>
<td>Blocking</td>
<td>void transport(const REQ&amp;, RSP&amp;);</td>
</tr>
<tr>
<td><strong>Unidirectional</strong></td>
<td></td>
</tr>
<tr>
<td>Blocking</td>
<td>void put(const T&amp;); void get(T&amp;); void peek(T&amp;);</td>
</tr>
<tr>
<td><strong>Unidirectional</strong></td>
<td>bool nb_put(const T&amp;); bool nb_can_put(); sc_event &amp;ok_to_put();</td>
</tr>
<tr>
<td>Non-Blocking</td>
<td>bool nb_get(T&amp;); bool nb_can_get(); sc_event &amp;ok_to_get();</td>
</tr>
<tr>
<td></td>
<td>bool nb.peek(T&amp;); ...</td>
</tr>
</tbody>
</table>

Slides derived from TLM2.0 kit
Bidirectional Interface with Explicit Timing

PV Initiator

transport()

PV Target

transport()

PV target model

```cpp
void transport(const REQ& rq, RSP& rp)
{
    // do processing
    // ...

    unsigned int latency = ...;
    wait(latency * clk_period);

    rp.get_status().set_ok();
}
```

- Initiation interval ≥ latency
Unidirectional Interfaces with Explicit Timing

- methodCall()
- eventNotification

- put(REQ)
- ok_to_get
- peek(REQ)
- get(REQ)
- ok_to_put
- put(RSP)
- peek(RSP)
- get(RSP)
- ok_to_put

Δ t_{m,\text{pop}}

Δ t_{s,\text{pop}}

Δ t_{\text{latency}}

Initiator — tlm_req_rsp_channel — Target
### TLM2.0 Core Interfaces

<table>
<thead>
<tr>
<th></th>
<th>Untimed</th>
<th>Timed</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bidirectional Blocking</strong></td>
<td>void transport(const REQ&amp;, RSP&amp;);</td>
<td>void transport(const REQ&amp;, RSP&amp;, sc_time&amp;);</td>
</tr>
<tr>
<td><strong>Unidirectional Blocking</strong></td>
<td>void put(const T&amp;); void get(T&amp;); void peek(T&amp;);</td>
<td></td>
</tr>
<tr>
<td><strong>Unidirectional Non-Blocking</strong></td>
<td>bool nb_put(const T&amp;); bool nb_can_put(); sc_event &amp;ok_to_put();</td>
<td>bool nb_put(const T&amp;); bool nb_can_put(const sc_time&amp;);</td>
</tr>
<tr>
<td></td>
<td>bool nb_get(T&amp;); bool nb_can_get(); sc_event &amp;ok_to_get();</td>
<td>bool nb_get(T&amp;, const sc_time&amp;);</td>
</tr>
<tr>
<td></td>
<td>bool nb.peek(T&amp;);</td>
<td>bool nb_can_get(const sc_time&amp;);</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Bidirectional Interface with Timing Annotation

PV Initiator  PV Target

transport()  transport()

$\Delta t_{\text{latency}}$

PV target model

```cpp
void transport(const REQ& rq, RSP& rp, sc_time& latency)
{
    // do processing ...
    double lat = ...;
    latency = lat * clk_period;
    rp.get_status().set_ok();
}
```

- Benefits:
  - Not calling wait => fast
  - Flexible - initiation interval < latency
  - Defers realization of timing

Guideline
Explicit versus Implicit Timing

Compare

Conceptually, a delay in the initiator?

No context switch!

Initiator

wait(del);
nb_put(t);

nb_put(t,del);

Initiator

wait(del)

nb_put(t)

nb_put(t,del)

del
Unidirectional Interfaces with Timing Annotation

Initiator

nb_put(REQ)

\(\Delta t_{s, \text{pop}}\)

ok_to_put

\(\Delta t_{\text{latency}}\)

Target

nb_put(RSP, \(\Delta t_{\text{latency}}\))

ok_to_get

\(\Delta t_{m, \text{pop}}\)

ok_to_put

peek(REQ)

nb_get(REQ, \(\Delta t_{s, \text{pop}}\))

peek(RSP)

nb_get(RSP, \(\Delta t_{m, \text{pop}}\))
Mixing Uni- and Bi-directional Timed Models

- Defer realization of timing, re-use PV peripheral
TLM Topology

PV:

System Initiator

transport(req, rsp)

System Target

PVT:

System Initiator

put(req)

put(rsp)

put(req)

System Target

See pvt_put_example

PVT:

System Initiator

put(req)

get(rsp)

get(req)

System Target

put(rsp)
Annotated Request-Response Channel

- (It ain’t simple!)
template < typename T >
class delayed_analysis_if : public virtual sc_interface {
public:
    virtual void write( const T& transaction,
                       const sc_time& time ) = 0;
};

template< typename T>
class tlm_peq : public sc_module,
               public virtual delayed_analysis_if<T>
{
    public:
        analysis_port<T> ap;
        ...
        int size() const;
        void write(const T& transaction, const sc_time& time);
};
Delayed Interface Rules

- `nb_put(t, del)` is immediate, but `t` only visible to `get()` after `del`
- `nb_get(t, del)` is immediate, but space only visible to `put()` after `del`

- Cannot predict overflow of a fixed-length fifo
  - Either accept a conservative approach (Draft 1)
    - `nb_can_put()` == false, even though fifo is not full
  - or accept that fifo might throw an exception
    - `write()` is a “non-negotiated” interface
More Rules

- \( nb\_get(t,\text{del}) \) is immediate ...
  ... so \( \text{ok\_to\_get}() \) is notified immediately on the next request

- System target must keep track of its own busy period
  ... and only notice \( \text{ok\_to\_get}() \) event when it is ready

See .\examples\pvt_annotated_examples\channels\tlm_pvt_annotated_fifo
Example - tlm_annotated_req_rsp_channel

SC_MODULE(Top) {
    typedef
tlm::tlm_annotated_req_rsp_channel<cx::REQ,cx::RSP> channel_type;
    channel_type *channel;
    cx_master    *master;
    cx_slave     *slave;

    SC_CTOR(Top): mem_size(200) {
        master = new cx_master("M1", mem_size, 0);
        slave  = new cx_slave("S1", mem_size, 0,
                      sc_time(10, SC_NS), // latency per word
                      sc_time(33, SC_NS)); // recovery delay

        channel = new channel_type<cx::REQ,cx::RSP>("C1", 4, 4);
        master->bus_port(channel->master_export);
        slave  ->bus_port(channel->slave_export);
    }
    const int mem_size;
};
Example - CX Slave

```cpp
void cx_slave::run() {
    sc_time delay;
    while (true) {
        cx::REQ req;
        if ( !bus_port->nb_peek(req) ) {
            wait( bus_port->ok_to_get() );
            bus_port->nb_peek(req);
        }
        bus_port->nb_get(req, recovery_delay);
        if (req.get_command() == tlm::WRITE) {
            const int size = req.get_block_size();
            const cx::ADDRESS_T address = req.get_address();
            const cx::DATA_T *data_ptr = req.get_data_ptr();
            for (int i = 0; i < size; i++)
                mem.at( address - slave_base + i ) = data_ptr[i];
            delay = latency_per_word * size;
        }
        cx::RSP rsp;
        if ( !bus_port->nb_put(rsp, delay) ) {
            wait( bus_port->ok_to_put() );
            bus_port->nb_put(rsp, delay);
        }
    }
    
    ...
The Implementation of TLM 2.0

- Introduction, Definitions and Principles
- Analysis Interface
- Timing Annotation
- **Generic Payloads**
- Pass-by-Pointer
Generic Payloads

- tlm_request and tlm_response transactions
- Use for generic PV and PVT modeling whenever possible
- Use standard interpretation for fields whenever possible
- Use custom extension mechanism only when necessary
- Use as starting point for specific protocol implementation
enum tlm_command {READ, WRITE};
enum tlm_mode {REGULAR, DEBUG, CONTROL};
enum tlm_block_mode {INCREMENT, STREAMING, WRAP};

template<typename ADDRESS, typename DATA, ...>
class tlm_request { ... 
    tlm_command m_command;
    tlm_mode m_mode;
    ADDRESS m_address;
    DATA *m_data; // deep copy // size of data array
    unsigned int m_block_size;
    const unsigned int *m_byte_enable; // may be null // size of byte en array
    unsigned int m_byte_enable_period;
    tlm_block_mode m_block_mode;
    unsigned int m_block_address_incr; // bytes-per-word
    unsigned int m_priority;
    unsigned int m_master_thread_id;
    unsigned int m_transaction_id;
    unsigned int m_tlm_export_id; // identifies target export
    std::vector<tlm_custom_base*> *m_custom_vector_ptr; }; // custom extensions
class tlm_status { ...
    unsigned int m_status;
    enum status_list {TLM_SUCCESS=0,TLM_ERROR,TLM_NO_RESPONSE};
};

template<typename DATA, ...>
class tlm_response { ...

protected:
    DATA *m_data;
    unsigned int m_block_size;
    tlm_status m_status;
    unsigned int m_priority;
    unsigned int m_master_thread_id;
    unsigned int m_transaction_id;
    unsigned int m_tlm_export_id;
    std::vector<tlm_custom_base *> *m_custom_vector_ptr;
};
Example - Trivial PV Master using tlm_req

typedef tlm::tlm_request<int, int> Request;

SC_MODULE(Master)
{
   sc_port<tlm_transport_if<Request, int> > port;

   SC_CTOR(Master) {
      SC_THREAD(T);
   }

   void T() {
      for (int i = 64; i < 72; i++) {
         Request req;
         req.set_command( static_cast<tlm::tlm_command>(rand() % 2) );
         req.set_address(i);
         req.set_data(i);
         int rsp ;
         port->transport(req, rsp);
         ...
      }
   }
}
Example - PV Slave and Top Level

```c
struct Slave: sc_channel, tlm_transport_if<Request, int>
{
    ...
    int transport( const Request& req )
    {
        wait(10, SC_NS);
        int old_data = mem[req.get_address()];
        if (req.get_command() == tlm::WRITE)
            mem[req.get_address()] = req.get_data();
        return old_data;
    }
    int mem[128];
};

SC_MODULE(Top)
{
    Master *master;
    Slave  *slave;
    SC_CTOR(Top) {
        master = new Master("master");
        slave  = new Slave("slave");
        master->port.bind(*slave);
    }
};
```
Example - PV Master Port doing a READ

```cpp
pv::DATA_T pv_mport::read(const pv::ADDRESS_T &addr) {
    request_type request;
    // Defaults: mode = REGULAR
    // block_size = 1
    // byte_enable = NULL
    request.set_command(tlm::READ);
    request.set_address(addr);
    request.set_data(0);
    request.set_master_thread_id(m_id);
    request.set_transaction_id(req_count++);

    response_type response = (*this)->transport(request);

    if (response.get_status().is_error() )
        SC_REPORT_ERROR("pv_master","Read error");
    return response.get_data();
}
```
The Implementation of TLM 2.0

- ... 

- Generic Payloads 
  - Block transfers and byte enables 
  - The three ‘id’s 
  - Custom extensions 

- Pass-by-Pointer
Block Transfers

<table>
<thead>
<tr>
<th>ADDRESS address</th>
<th>unsigned int index</th>
<th>DATA *data</th>
<th>unsigned int *byte_enable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xF10</td>
<td>0xF10</td>
<td>0xF10</td>
<td></td>
</tr>
<tr>
<td>0xF14</td>
<td>0xF14</td>
<td>0xF10</td>
<td></td>
</tr>
<tr>
<td>0xF18</td>
<td>0xF18</td>
<td>0xF10</td>
<td></td>
</tr>
<tr>
<td>0xF1C</td>
<td>0xF1C</td>
<td>0xF10</td>
<td></td>
</tr>
<tr>
<td>0xF20</td>
<td>0xF00</td>
<td>0xF10</td>
<td></td>
</tr>
<tr>
<td>0xF24</td>
<td>0xF04</td>
<td>0xF10</td>
<td></td>
</tr>
<tr>
<td>0xF28</td>
<td>0xF08</td>
<td>0xF10</td>
<td></td>
</tr>
<tr>
<td>0xF2C</td>
<td>0xF0C</td>
<td>0xF10</td>
<td></td>
</tr>
</tbody>
</table>

INCREMENT          WRAP       STREAMING

block_address_incr

byte_enable_period

block_size

index

0
1
2
3
4
5
6
7
byte_enable and Simulated Endianness

```c
enum tlm_endianness { TLM_LITTLE_ENDIAN, TLM_BIG_ENDIAN };
```

<table>
<thead>
<tr>
<th>Endianness</th>
<th>Byte Enable (MASK)</th>
<th>Address Offset</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Little</td>
<td>0x1</td>
<td>Add+0</td>
<td>LSB</td>
</tr>
<tr>
<td></td>
<td>0x2</td>
<td>Add+1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x4</td>
<td>Add+2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x8</td>
<td>Add+3</td>
<td>MSB</td>
</tr>
<tr>
<td>Big</td>
<td>0x1</td>
<td>Add+0</td>
<td>MSB</td>
</tr>
<tr>
<td></td>
<td>0x2</td>
<td>Add+1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x4</td>
<td>Add+2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x8</td>
<td>Add+3</td>
<td>LSB</td>
</tr>
</tbody>
</table>

See ./examples/byte_enable_examples
Block Transfers - Rules

- \( \text{block\_size} \) = \# elements in data array
- \( \text{block\_address\_incr} \) = \# bytes-per-word
- \# bytes = \( \text{block\_size} \times \text{block\_address\_incr} \)
- \( \text{byte\_enable\_period} \) = \# elements in byte enable array
- \( \text{data}[\text{index}] \) \( \sim \) \( \text{byte\_enable}[\text{index} \mod \text{byte\_enable\_period}] \)
- Hence data word limited to 32 bytes
- High-order bits of each byte_enable word often unused
- Given data[\text{index}] with INCREMENT mode,

  \[
  \text{address} = [\text{base address}] + \text{block\_address\_incr} \times \text{index}
  \]
The Implementation of TLM 2.0

- ... 
- **Generic Payloads**
  - Block transfers and byte enables
  - The three ‘id’s
  - Custom extensions
- **Pass-by-Pointer**
The Three ‘id’s

Initiator

- priority = 0
- master_thread_id = 0
- transaction_id = 0
- tlm_export_id = 0

Target

- priority = 0
- master_thread_id = 1
- transaction_id = 2
- tlm_export_id = 0

- priority = 0
- master_thread_id = 1
- transaction_id = 1
- tlm_export_id = 1
tlm_initiator_port, tlm_target_port

tlm_export_id must be set by the initiator!

```cpp
tlm_initiator_port<tlm_blocking_put_if<tlm_request<int,int> > > port;
tlm_request<int,int> request;
request.set_tlm_export_id(port.get_target_port_list()[0]->get_tlm_export_id());
port->put(request);

tlm_target_port<...> export0;
tlm_target_port<...> export1;
export0.set_tlm_export_id(0);
export1.set_tlm_export_id(1);
```

See ./examples/export_id_example
The Implementation of TLM 2.0

• ... 

• **Generic Payloads**
  – Block transfers and byte enables
  – The three ‘id’s
  – Custom extensions

• Pass-by-Pointer
TLM Interoperability

typedef uint64 ADDRESS;
typedef uint64 DATA;
typedef tlm_request<ADDRESS,DATA> req_type;
typedef tlm_blocking_put_if<req_type> put_if;

- tlm_request/response + int address/data => 'TLM interoperability'

Generic Payload
Core Interface

TLM Initiator

TLM Target

sc_port<put_if> port; sc_export<put_if> export;
Custom Extensions

```cpp
class tlm_custom_base {
    public:
        virtual ~tlm_custom_base() {}
        virtual tlm_custom_base *clone() = 0;
    }

std::vector<tlm_custom_base *> *m_custom_vector_ptr;
```

- Extensions compromise interoperability!
Interconnect Components and Extensions

```cpp
std::vector<tlm_custom_base *> *m_custom_vector_ptr;

void tlm_request<...>::set_custom_vector_ptr( { std::vector< tlm_custom_base* > *from ) {
... m_custom_vector_ptr = 
    new std::vector< tlm_custom_base* >(from->size());
tlm_custom_base *t_custom_ptr;
for( int i = 0; i < from->size(); i++ ) {  
    if( (t_custom_ptr = (*from)[i]) )  
        (*m_custom_vector_ptr)[i] = t_custom_ptr->clone();
} }
```
The Implementation of TLM 2.0

- Introduction, Definitions and Principles
- Analysis Interface
- Timing Annotation
- Generic Payloads
- Pass-by-Pointer
Fundamental Principle of TLM 2.0 draft 1

(Method of) Core TLM Interface       TLM Transaction

put( transaction );

- TLM transaction only valid for duration of function call
  (system transaction sliced-and-diced into independent chunks) ...

... except for pass-by-pointer mode

An optimization, for simulation efficiency
Request/Response Arrays

template<typename ADDRESS, typename DATA,
        tlm_data_mode DATA_MODE = TLM_PASS_BY_COPY>
class tlm_request {

    ...  DATA                          *m_data;
    const unsigned int            *m_byte_enable;
    std::vector<tlm_custom_base*> *m_custom_vector_ptr;

    ...

};

template<typename DATA,
        tlm_data_mode DATA_MODE = TLM_PASS_BY_COPY>
class tlm_response {

    ...  DATA                          *m_data;
    std::vector<tlm_custom_base*> *m_custom_vector_ptr;

    ...

};
tlm_data_mode

• TLM_PASS_BY_COPY (default)

• TLM_PASS_BY_POINTER
Pass-by-Copy Rules

- **Must** copy transaction+arrays if it lives beyond function call
- Not obliged to copy if transaction finished with before return
- Request and response arrays are independent
Pass-by-Pointer
Pass-by-pointer Rules

- **Pass-by-Pointer Initiator** means the original TLM Initiator of the request.
- P-by-P Initiator provides storage for arrays: static, stack (preferred), or heap.
- P-by-P Initiator initialises the arrays (except data array for a read).
- For a write, P-by-P Target must copy from the data array.
- For a read, P-by-P Target must copy to the data array (aka *copy-at-slave*).
- P-by-P Target is obliged to copy all common fields from request to response, including array pointers.
- Lifetime of arrays extends until after P-by-P Initiator receives response.
- Only the P-by-P Initiator may de-allocate the arrays.
- Each P-by-P transaction has exactly one P-by-P Initiator and Target.
Mixing Modes
Alternative Approach (e.g. GreenBus)

System Initiator → System Interconnect → System Target

smart_ptr

Single container for system transaction

Phase

Standard transaction attributes, becoming cumulatively valid (base class)

Custom attributes (derived class)
## Generic Protocol Phases

<table>
<thead>
<tr>
<th>Phase</th>
<th>ReqValid</th>
<th>ReqAccepted</th>
<th>DataValid</th>
<th>DataAccepted</th>
<th>RespValid</th>
<th>RespAccepted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set by</td>
<td>Initiator</td>
<td>Target</td>
<td>Initiator</td>
<td>Target (write)</td>
<td>Target</td>
<td>Initiator</td>
</tr>
</tbody>
</table>

- **Command, Address, Burst length, Thread id**

- **Data**

- **Attribute Validity**

- **Status**
The Implementation of TLM 2.0

- Introduction, Definitions and Principles
- Analysis Interface
- Timing Annotation
- Generic Payloads
- Pass-by-Pointer

Download from www.systemc.org
Summary of Guidelines

Use message passing for TL-communication

Use the effective pass-by-value rules

Use analysis ports for non-intrusive monitoring

Use timing annotation to defer timing realization

Use 1 thread per master for PV, per slave too for PVT

Use generic payloads tlm_request and tlm_response