ECE382M.20: System-on-Chip (SoC) Design

Lecture 4 – SystemC Transaction-Level Modeling

Sources:
M. Radetzki, Univ. of Stuttgart
OSCI TLM 2.0 Documentation

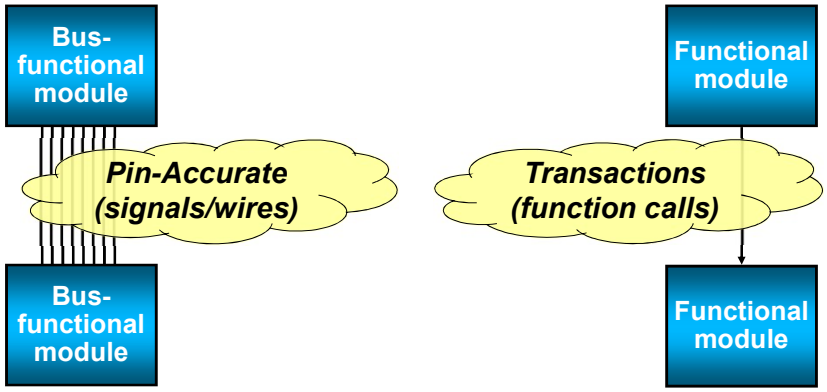
Andreas Gerstlauer
Electrical and Computer Engineering
The University of Texas at Austin
gerstl@ece.utexas.edu

The University of Texas at Austin
Chandra Department of Electrical and Computer Engineering
Cockrell School of Engineering

Lecture 4: Outline

- **SystemC Transaction-Level Modeling (TLM)**
  - SystemC TLM 2.0 core library
  - Initiators and targets, payloads, sockets
  - Coding styles & abstraction levels
- **Advanced modeling approaches**
  - Temporal decoupling
  - Time quanta & quantum keeper
  - Direct memory interfaces (DMI)

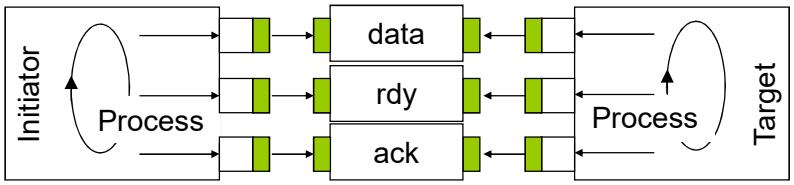
Transaction-Level Modeling (TLM)



- **Pin-accurate model (PAM)**
  - Simulate every event (protocols)
- **Transaction-level model (TLM)**
  - Communications by transactions (abstract channels)
  - Granularity of transactions? Dynamic effects?

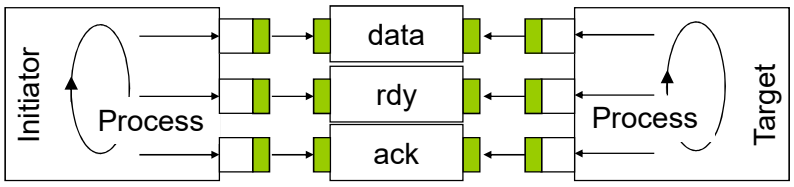
Source: OSCI TLM-2.0

Inlined Pin-Accurate Model (PAM)




```
class top : public sc_module {
  public:
    sc_signal<char> data;
    sc_signal<bool> rdy;
    sc_signal<bool> ack;

    initiator *initiator_inst;
    target *target_inst;

    top(sc_module_name name) : sc_module(name) {
      data = -1; rdy = 0; ack = 0;

      initiator_inst = new initiator("Initiator1");
      initiator_inst->data(data);
      initiator_inst->rdy(rdy);
      initiator_inst->ack(ack);

      target_inst = new target("Target1");
      target_inst->data(data);
      …
  }
};
```

Inlined Pin-Accurate Model (PAM)

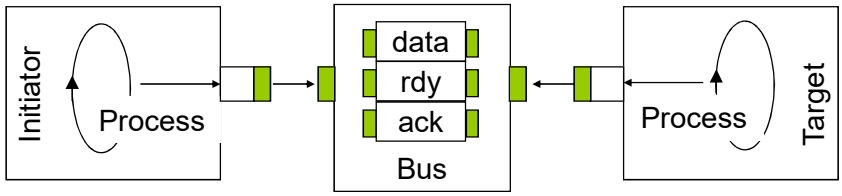



```
class initiator : public sc_module {
  public:
    sc_out<char> data;
    sc_out<bool> rdy;
    sc_in<bool> ack;

    SC_HAS_PROCESS(initiator);
    initiator(sc_module_name name):
                    sc_module(name) {
      SC_THREAD(main);
    }

    void main() {
      …                    void write(char)
      data = c;
      wait(…, SC_NS);
      rdy = 1;                    write(c);
      while(!ack)
        wait(ack->default_event());
      …
    }
};
```

Communication protocol

```
class target : public sc_module {
  public:
    sc_in<char> data;
    sc_in<bool> rdy;
    sc_out<bool> ack;

    SC_HAS_PROCESS(target);
    target(sc_module_name name):
                    sc_module(name) {
      SC_THREAD(main);
    }

    void main() {
      …                    void read(&char)
      while(!rdy)
        wait(rdy->default_event());
      c = data;                   read(c);
      wait(…, SC_NS);
      ack = 1;
      …
    }
};
```

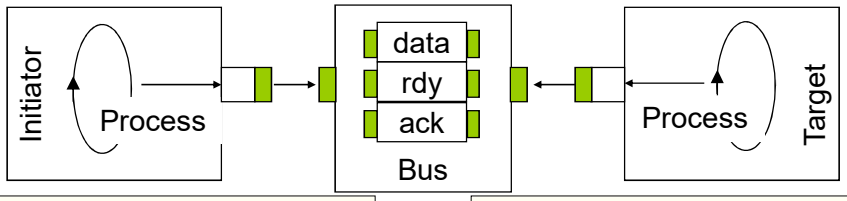

Channel-Encapsulated PAM




```
class initiator_if :
   virtual public sc_interface
{
   public:
     virtual void write(char) = 0;
};

class target_if :
   virtual public sc_interface
{
   public:
     virtual void read(char &) = 0;
};
```

```
class bus : public sc_channel {
   public:
     bus(sc_module_name name): sc_channel(name) {
       data = -1; rdy = 0; ack = 0;
     }

     void write(char c) {
       data = c;
       wait(…, SC_NS);
       rdy = 1;
       while(!ack) wait(ack->default_event());
     }

     void read(char &c) {
       while(!rdy) wait(rdy->default_event());
       …
     }

   private:
     sc_signal<char> data;
     sc_signal<bool> rdy;
     sc_signal<bool> ack;
  }
};
```

Channel-Encapsulated PAM

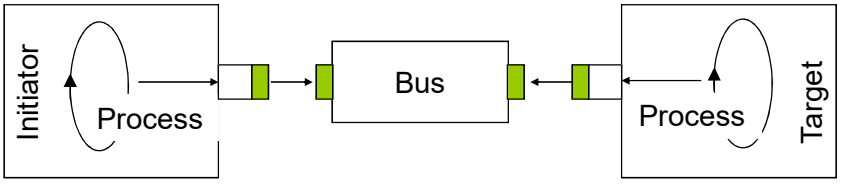



```
class initiator : public sc_module {
  public:
    sc_port<initiator_if> bus;

    …

    void main() {
      …
      bus->write(c);
      …
    }
};
```

```
class target : public sc_module {
  public:
    sc_port<target_if> bus;

    …

    void main() {
      …
      bus->read(c);
      …
    }
};
```

```
class top : public sc_module {
   public:
     bus *bus_inst;
     initiator *initiator_inst;
     target *target_inst;

     top(sc_module_name name) : sc_module(name) {
       bus_inst = new bus("Bus1");

       initiator_inst = new initiator("Initiator1");
       initiator_inst->bus(*bus_inst);
       …
```

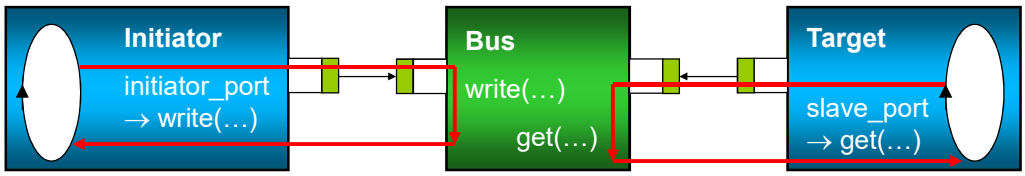

Transaction-Level Model (TLM)



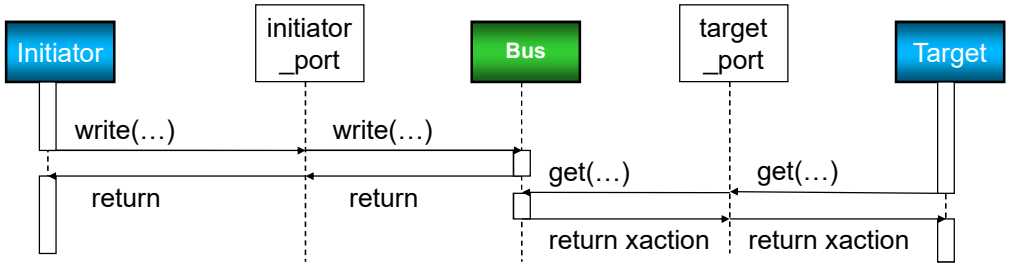

```
class bus : public sc_channel {
   public:
     bus(sc_module_name name): sc_channel(name), valid(false) {}

     void write(char c) {
       data = c;
       wait(…, SC_NS);  // model transaction timing
       valid = true;
       rdy.notify();
       wait(ack);
     }

     void read(char &c) {
       if(!valid) wait(rdy);
       c = data;
       valid = false;
       ack.notify();
     }

   private:
     char data;
     bool valid;
     sc_event rdy, ack;
  }
};
```

Active Targets

- **Targets have own thread, actively get bus transactions**

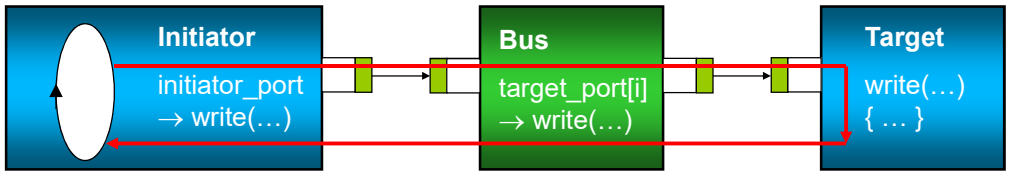


- May model an active, non-blocking bus with transaction delays

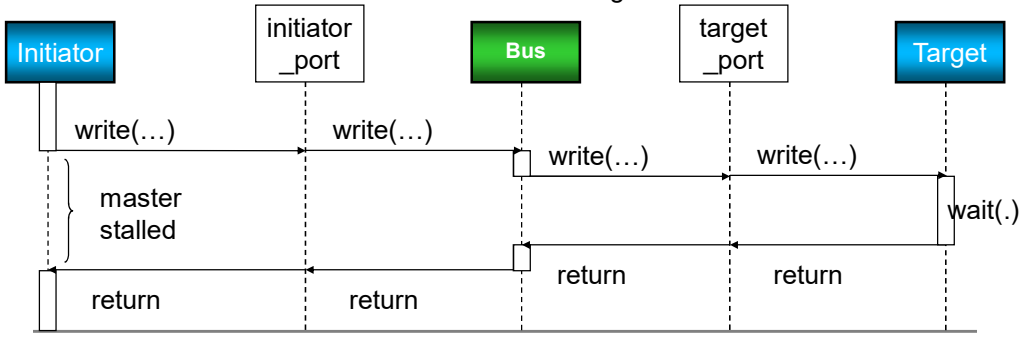


Passive Targets

- **Target methods executed by initiator thread (`sc_export`)**

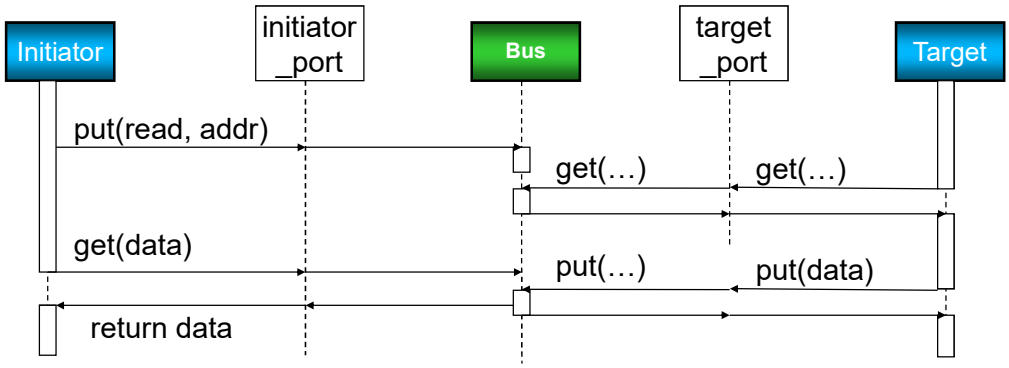


- No context switch, no event handshaking overhead
- Directly deposit into/extract from the target (memory/memory-mapped)
- But: the initiator is blocked while bus / target uses its thread



Bi-Directional Transactions

- **To avoid blocking initiator, split transaction into two parts**
  - Putting transaction request on the bus (address & type)
  - Getting the result later (data)
- **Modeled as two uni-directional transactions**
  - Incurs simulation (function call) overhead



SystemC/TLM 2.0

Use cases

Software development | Software performance | Architectural analysis | Hardware verification

TLM-2 Coding styles

Loosely-timed

Approximately-timed

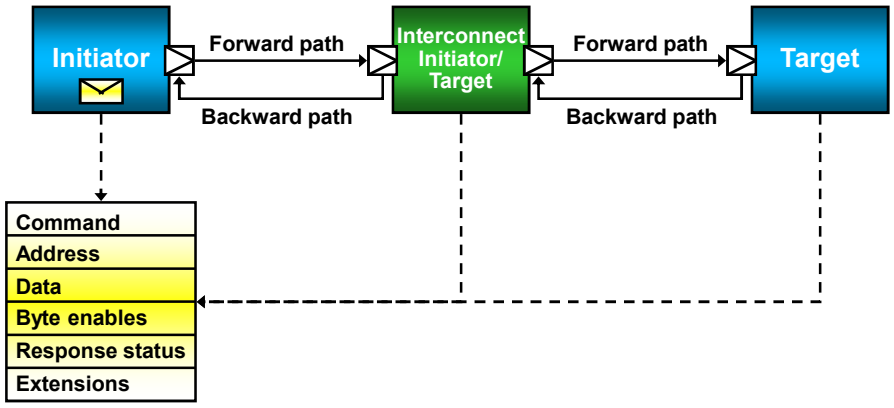
Mechanisms

Blocking interface | DMI | Quantum | Sockets | Generic payload | Phases | Non-blocking interface

Source: OSCI TLM-2.0

Initiators, Targets and Transactions

- Pointer to transaction object is passed from module to module using forward and backward call paths
- Transactions are of generic payload type



Source: OSCI TLM-2.0

Generic Payload

```
class  tlm_generic_payload {                                  Not a template
public:

  // Constructors, memory management
  tlm_generic_payload () ;
  tlm_generic_payload(tlm_mm_interface& mm);                  Construct & set mm
  virtual  ~tlm_generic_payload ();                           Frees all extensions
  void reset();                                               Frees mm'd extensions

  void set_mm(tlm_mm_interface* mm);                          mm is optional
  bool has_mm();
  void acquire();                                             Incr reference count
  void release();                                             Decr reference count,
  int get_ref_count();                                        0 => free trans

  void deep_copy_from(const  tlm_generic_payload&  other);

  ...
};
```

Source: OSCI TLM-2.0

Generic Payload Attributes

Attribute	Type	Modifiable?	
Command	tlm_command	No	
Address	uint64	Intercnct. only	
Data pointer	unsigned char*	No (array – yes)	*Array owned by initiator*
Data length	unsigned int	No	
Byte enable pointer	unsigned char*	No (array – yes)	*Array owned by initiator*
Byte enable length	unsigned int	No	
Streaming width	unsigned int	No	
DMI hint	bool	Yes	*Try DMI !*
Response status	tlm_response_status	Target only	
Extensions	(tlm_extension_base*) [ ]	Yes	*Consider memory management*

Source: OSCI TLM-2.0

Command, Address and Data

```
enum tlm_command {
    TLM_READ_COMMAND,
    TLM_WRITE_COMMAND,
    TLM_IGNORE_COMMAND
};
```

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

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

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

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

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

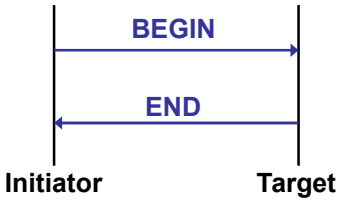
Data array owned by initiator

Number of bytes in data array

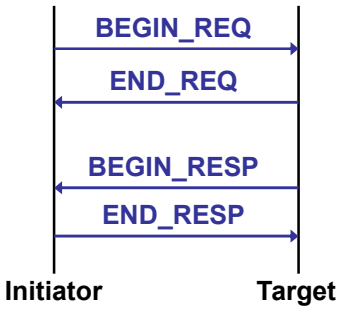
Source: OSCI TLM-2.0

SystemC/TLM 2.0 Coding Styles

- **Loosely-timed**
  - Sufficient timing detail to boot OS and simulate multi-core systems
  - Each transaction has 2 timing points: *begin* (call) and *end* (return)
  - One bi-directional call



- **Approximately-timed**
  - Cycle-approximate or cycle-count-accurate
  - Sufficient for architectural exploration
  - Each transaction has at least 4 timing points
    - 4 forward/backward uni-directional calls
    - Must immediately return (no `wait()`)



Source: OSCI TLM-2.0

Transport Interfaces

```
template < typename  TRANS  =  tlm_generic_payload >
```

- **Blocking transport interface**
  - Typically used with loosely-timed coding style
  - **`tlm_blocking_transport_if`**

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

- **Non-blocking transport interface**
  - Typically used with approximately-timed coding style
  - Includes transaction phases
  - **`tlm_fw_nonblocking_transport_if`**

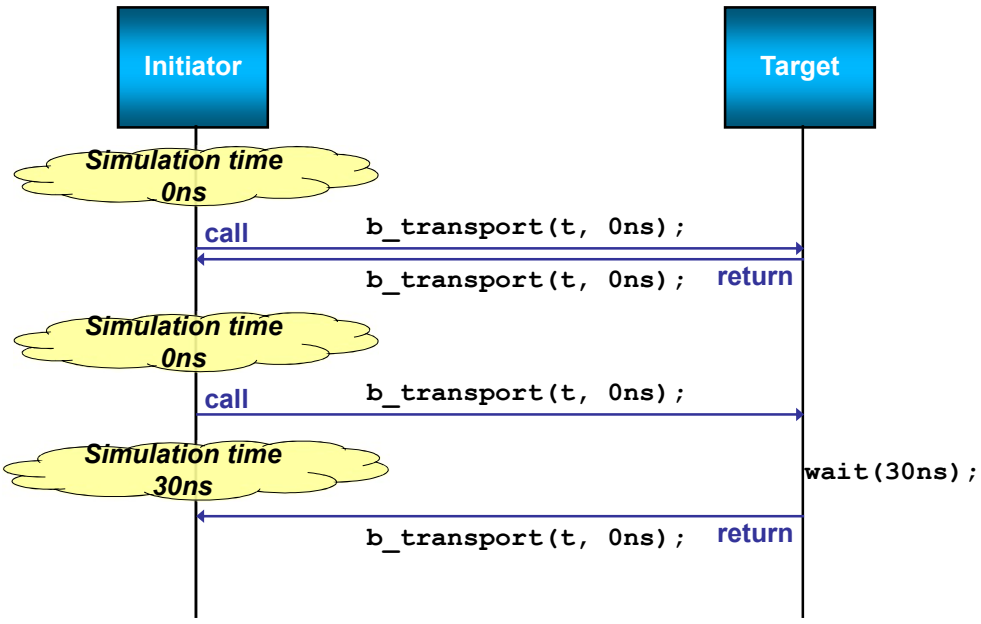
```
tlm_sync_enum nb_transport_fw(TRANS&, PHASE&, sc_time&);
```

  - **`tlm_bw_nonblocking_transport_if`**

```
tlm_sync_enum nb_transport_bw(TRANS&, PHASE&, sc_time&);
```

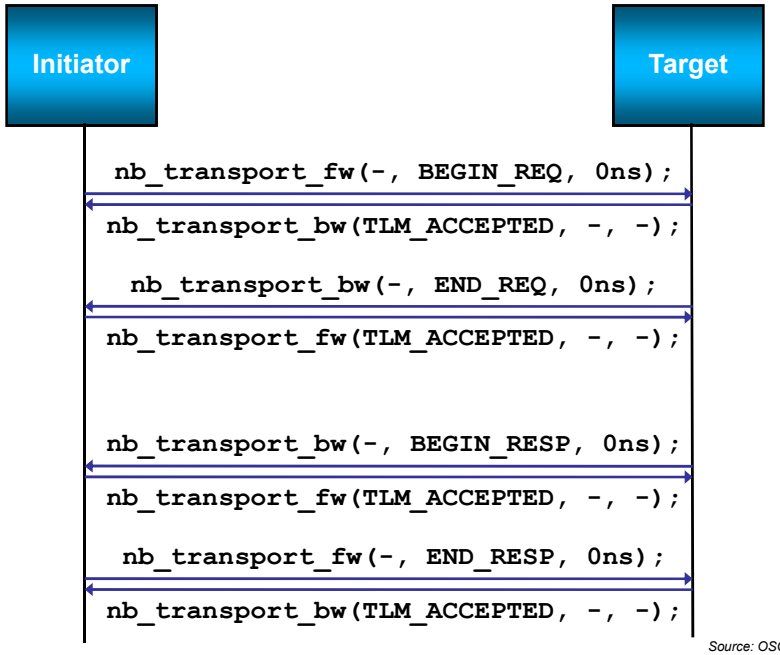
Source: OSCI TLM-2.0

Loosely Timed (LT) Model



Source: OSCI TLM-2.0

Approximately Timed (AT) Model



Source: OSCI TLM-2.0

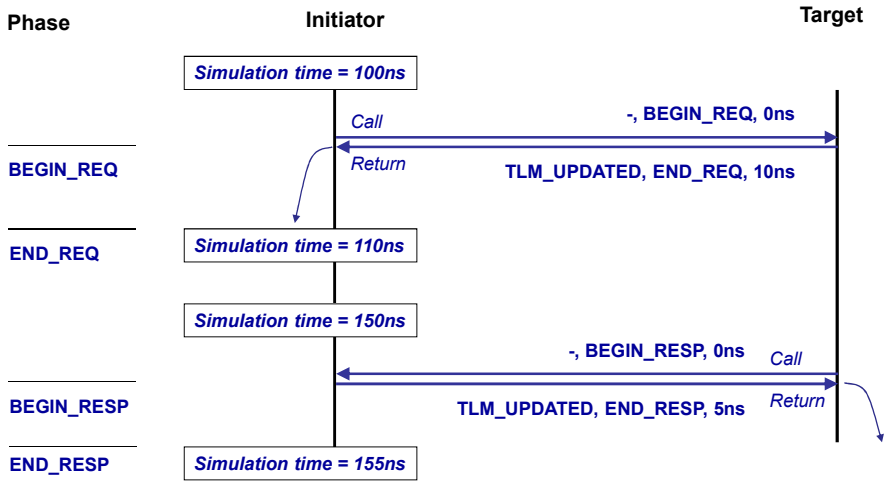
NB Return Values (tlm_sync_enum)

- **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

Source: OSCI TLM-2.0

Using the Return Path

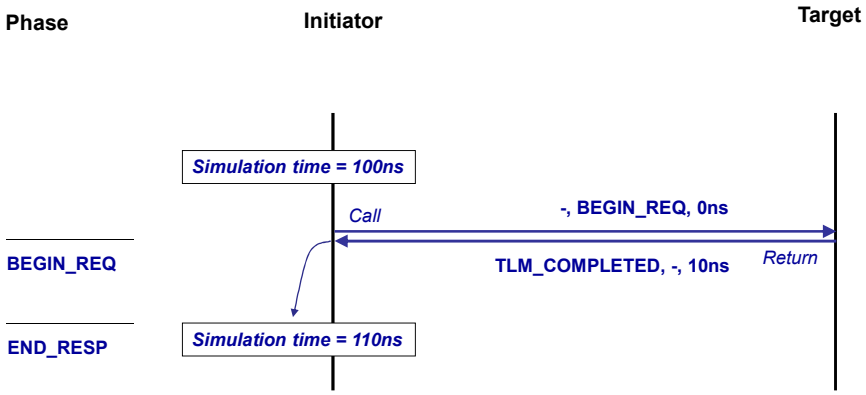
- **Callee annotates delay to next transaction**
  - Caller waits



Source: OSCI TLM-2.0

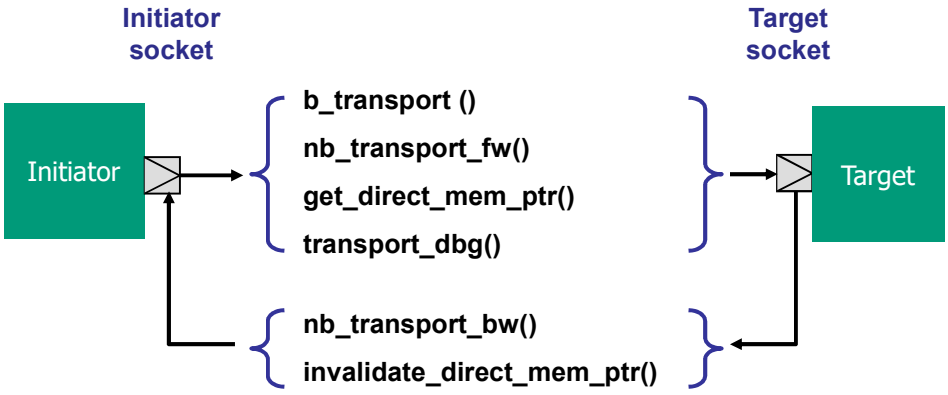
Early Completion

- Callee annotates delay to next transaction
  - Caller waits



Source: OSCI TLM-2.0

Sockets



- Sockets combine fw and bw paths and group interfaces

Source: OSCI TLM-2.0

Convenience Sockets

- **The "simple" sockets**
  - **`simple_initiator_socket`** and **`simple_target_socket`**
  - In namespace **`tlm_utils`**
  - Derived from base sockets **`tlm_initiator_socket`** and **`tlm_target_socket`**
- **"simple" because they are simple to use**
  - Do not bind sockets (ports) to objects (implementations)
  - Instead, register methods with each socket
  - Do not allow hierarchical binding
- **Not obliged to register both `b_transport` and `nb_transport`**
  - Automatic conversion (assumes base protocol)
  - Variant with no conversion – passthrough_target_socket

Source: OSCI TLM-2.0

Simple Socket Example (Initiator)

```
class Initiator: public sc_module
{
  tlm_utils::simple_initiator_socket<Initiator> init_socket;

  SC_CTOR(Initiator): init_socket("init_socket")
  {
    // Register methods for backward path
    init_socket.register_nb_transport_bw(this, &Initiator::nb_transport_bw);
    init_socket.register_invalidate_direct_mem_ptr(this,
                                    &Initiator::invalidate_direct_mem_ptr);
    SC_THREAD(thread);
  }

  void thread() { ...
      // Call methods on forward path
      init_socket->b_transport( trans, delay );
      init_socket->nb_transport_fw( trans, phase, delay );
      init_socket->get_direct_mem_ptr( trans, dmi_data );
      init_socket->transport_dbg( trans );
  }

  // Methods on backward path
  virtual tlm::tlm_sync_enum nb_transport_bw( ... );
  virtual void invalidate_direct_mem_ptr( ...);
};
```

Source: OSCI TLM-2.0

Simple Socket Example (Target)

```
class Target: public sc_module
{
  tlm_utils::simple_target_socket<Target> targ_socket;

  SC_CTOR(Target): targ_socket("targ_socket")
  {
    // Register methods for forward path
    targ_socket.register_nb_transport_fw(   this, &Target::nb_transport_fw);
    targ_socket.register_b_transport(       this, &Target::b_transport);
    targ_socket.register_transport_dbg(     this, &Target::transport_dbg);
    targ_socket.register_get_direct_mem_ptr(this,
                                            &Target::get_direct_mem_ptr);
    SC_THREAD(thread);
  }

  void thread() { ...
    // Call methods on backward path
    targ_socket->nb_transport_bw( ... );
    targ_socket->invalidate_direct_mem_ptr( ... );
  }

  // Methods on forward path
  virtual void b_transport( ... );
  virtual tlm::tlm_sync_enum  nb_transport_fw( ... );
  virtual bool get_direct_mem_ptr( ... );
  virtual unsigned int transport_dbg( ... );
};
```

Source: OSCI TLM-2.0

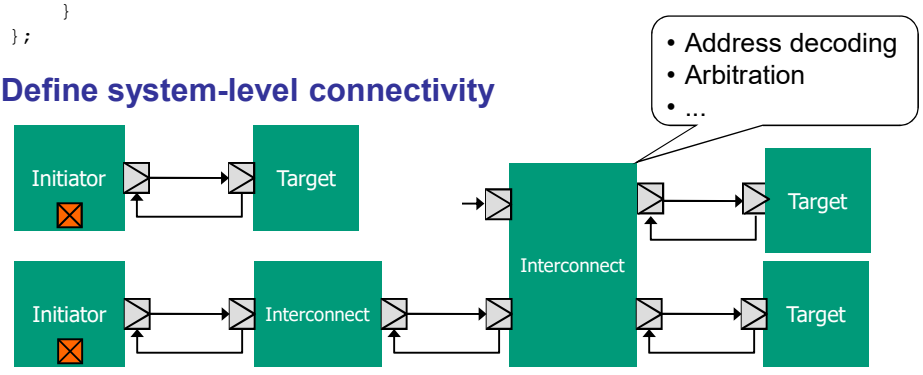
ECE382M.20: SoC Design, Lecture 4 © 2023 A. Gerstlauer 27

Simple Socket Example (Top)

- **Bind initiator to target sockets**

```
SC_MODULE(Top)  {
    Initiator  *init;
    Target     *targ;
    SC_CTOR(Top)  {
        init  = new Initiator("init");
        targ = new Target("targ");
        init->init_socket.bind( targ->targ_socket );
    }
};
```

- **Define system-level connectivity**



Source: OSCI TLM-2.0

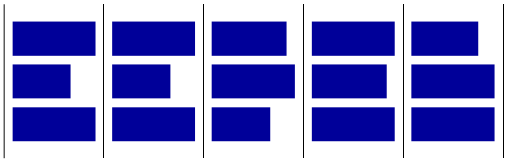
ECE382M.20: SoC Design, Lecture 4 © 2023 A. Gerstlauer 28

Lecture 4: Outline

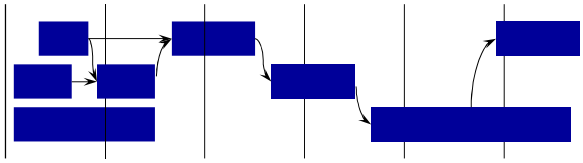
- ✓ SystemC Transaction-Level Modeling (TLM)
  - ✓ SystemC TLM 2.0 core library
  - ✓ Initiators and targets, payloads, sockets
  - ✓ Coding styles & abstraction levels
- **Advanced modeling approaches**
  - Temporal decoupling
  - Time quanta & quantum keeper
  - Direct memory interfaces (DMI)

Temporal Decoupling

- **Loose coupling**
  - OS and driver SW development
  - Local process time
  - Every process runs ahead until data is missing or a time quantum boundary was reached (local/global time synchronization)

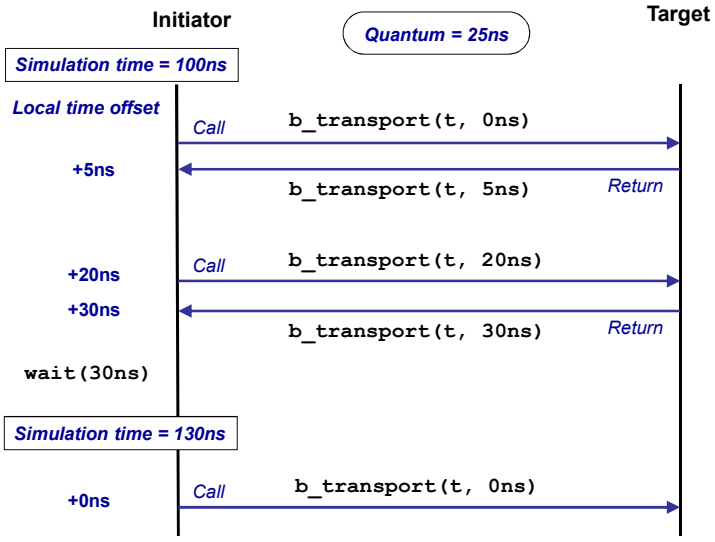


- **Approximate coupling**
  - Architecture trade-off
  - Each process has the global SystemC time, processes synchronize
  - Time may be accurate or estimated



Source: W. Ecker, Infineon

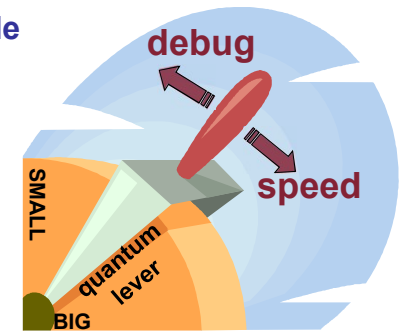
Time Quantum



Source: OSCI TLM-2.0

The Quantum Keeper (tlm_quantumkeeper)

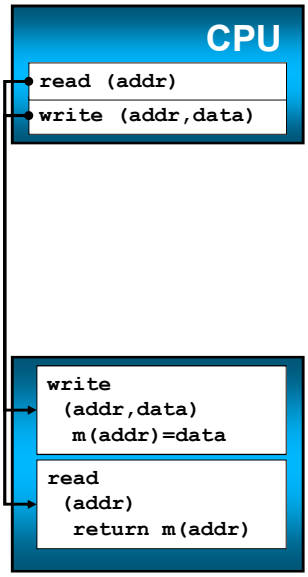
- Quantum is user-configurable



- Processes can check local time against quantum

Source: OSCI TLM-2.0

Direct Memory Interface (DMI)



- **Implementation approach**
  - Direct member function call of target object
    - In master context
  - Pure virtual interface classes defined interface
  - Function call encapsulates and hides all interconnect details
  - Port/export provides connection semantics (incl. restrictions)
- ➢ **Direct memory interface**
  - ➢ Faster (no context switch)
- ➢ **Debug interface**

Source: W. Ecker, Infineon

SystemC DMI

- **Direct Memory Interface (DMI)**
  - Gives an initiator a direct pointer to memory in a target
  - By-passes the sockets and transport calls
  - Read or write access by default
  - Extensions may permit other kinds of access
  - Target responsible for invalidating pointer
- **SystemC interfaces**
  - **`tlm_fw_direct_mem_if`**

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

  - **`tlm_bw_direct_mem_if`**

```
void invalidate_direct_mem_ptr(sc_dt::uint64 start_range,
                               sc_dt::uint64 end_range);
```

Source: OSCI TLM-2.0

Lecture 4: Summary

- **SystemC Transaction-Level Modeling (TLM)**
  - TLM 2.0 class library on top of base language
    - Evolution from TLM 1.0
  - Detailed documentation as part of TLM install
    - `/usr/local/packages/systemc-2.3.3/docs/tlm/release`
- **Speed vs. accuracy in communication modeling**
  - Various TLM modeling concepts & approaches
    - Modeling of arbitration, bus pipelining, split transactions, …
  - Virtual platforms at varying levels of abstraction
    - Loosely timed (LT) TLM – programmer's view
    - Approximately timed (AT) TLM – architect's view
    - Cycle-accurate – verification view
    - Pin-accurate – RTL hand-off