System Modelling
Transaction Level Modelling

Prof. Dr.- Ing. Sorin Alexander Huss

Integrierte Schaltungen und Systeme
Fachbereich Informatik
Technische Universität Darmstadt

SS 2007
Transaction-Level Models with SystemC

- TLM Methodology
- Introduction to SystemC
- Transaction Level Modelling in SystemC

TLM provides a methodology for

- Early software development
- Architecture analysis
- Functional verification
Novel design flow for embedded systems

1. Customer Specification
2. Paper Specification
3. HW/SW Partitioning
4. TLM
5. Hardware Development
6. Concurrent HW/SW Engineering Based on TLM
7. Software Development
8. Test Chip
9. System Integration & Validation
10. Chip Fabrication
Triple abstraction structure

1. Functional view
2. Architecture view
3. Micro-architecture view
1. Functional view

- abstracts the expected behavior of a given system
- executable specification of the system function composed of algorithmic SW
- no consideration of any implementation details - such as architecture or address mapping information - and of performance figures
2. Architecture view

- provides the platform for associated SW development
- provides the reference model for verification purposes, i.e., for the generation of functional verification tests of subsequent implementation models
3. Micro-architecture view

- captures all required information for timed and cycle-accurate simulation of RT level models
- validates low-level embedded SW in the real HW simulation environment
- validation of micro-architecture for real-time requirements
Principles of TLM

- Modelling of components as **modules**
- Communication structure by means of **channels**
- Modules and channels are bound to each other by communication **ports**
- A set of data is exchanged by a **transaction**
- **System synchronisation** is an explicit action between modules, e.g., interrupt by DMA to notify a transfer completion
Modelling approach

TLM modules must hold the following characteristics:

- Bit-true behavior of the component
- Register-accurate interface of the component
- System synchronisation managed by the component
TLM vs. RT level simulation

TLM Simulation

RTL Simulation

S1

S2

F

F

F_{TLM}

F_{RTL}
Modelling accuracy

- Granularity of communication data
  Levels: *application packet, bus packet, bus size*
  (Example video transfer: frame, line, pixel)

- Timing accuracy
  Levels: *untimed, approximately-timed, timed TLM*
Modelling accuracy (2)

Legend:
- RTL: register transfer level
- BCA: bus cycle accurate
- CA: cycle accurate
- PV: programmer’s view
- VT: programmer’s view plus timing

Data Granularity

Bus Size

Bus Packet

Application Packet

Algorithmic Model

TLM−PV

Timed Accuracy

Untimed
Approximately−timed
Cycle−accurate
Untimed TLM - Model of Computation

Characteristics

- Concurrent execution of independent processes
- Respect for causal process dependencies by using system synchronisation
- Bit-true behavior
- Bit-true communication
Execution order within P1, P2, P3: System synchronisation between concurrent processes:

1. P11 → P12 for P1
2. P21 → P22 for P2
3. P31 → P32 for P3
4. P11 → P22
5. P22 → P12
Untimed TLM - System Synchronisation (2)

Different overall execution order (process interleaves) examples:

1. P21 → P11 → P22 → P12 → P31 → P32
2. P31 → P32 → P21 → P11 → P22 → P12

Synchronisation kinds:

**Emit-synchronisation**  A process sends out a synchr. that may influence the behavior or state of other processes.

**Receive-synchronisation**  A process waits for an event from the system that may influence its behavior or state.
### Untimed TLM - Deterministic Behavior

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><em>Activate</em> or resume a process</td>
</tr>
<tr>
<td>2</td>
<td>Read input data for control flow and data processing</td>
</tr>
<tr>
<td>3</td>
<td>Computation</td>
</tr>
<tr>
<td>4</td>
<td>Write output data if there is any</td>
</tr>
<tr>
<td>5</td>
<td>Return to Step 2 if more computation is required</td>
</tr>
<tr>
<td>6</td>
<td>Synchronisation:</td>
</tr>
<tr>
<td></td>
<td>a) if it is 'emit-synchronisation', then return to Step 2</td>
</tr>
<tr>
<td></td>
<td>b) if it is 'receive-synchronisation', then the process will be <em>suspended</em>.</td>
</tr>
</tbody>
</table>
Note: Functional delays inserted at *architecture* level. Insertion of functional delays must not change causal dependencies in a given system!
Objectives:

- Benchmarking of the performance of a given *micro-architecture*
- Fine tuning of the micro-architecture
- Optimizing the SW for a given micro-architecture to meet real-time constraints

Modelling approaches:

- *Time annotated* untimed model
- Standalone *timed model*
Time annotated TLM

- Well-suited in case of a fairly good match of the untimed model structure to the micro-architectural model (re-use of untimed TLM): Simply **insert parametrized wait statements** related to the concerning computation times.

- In general: **Define delay of each possible activation-synchronisation** in a process based on the control flow of the component. This may result in rather complex graphs and in a large set of timing attributes.
Detached model incorporated with timing information: High-level analytical timing model *without* functional information. Suitable in case of large differences between the structures of the algorithm and of the related micro-architecture.

**Dependency of timing behaviour on functional component behaviour:** Control of the timed model by an untimed TLM by tracing and forwarding of all functional events produced by the untimed model to the standalone timed model.
Combined Untimed and Standalone Timed TLM

Model architecture

**Untimed TLM**
- Architectural States
- Data

**Standalone Timed Model**
- Micro-architectural States & Timing constraints
- Data Port
- Bus Ports

Timing Control Unit

Traces of Functional Events

Threads

Model execution

- Thread Activation
- Un-timed TLM
- Standalone Timed Model

Read Compute Write

Delay Write Read Read Delay

Synchronization

0 delay

0ns 0ns 10ns 10ns
0 delay
TLM as Unique Reference Model

- Verification Team
- Algorithm Team
- Software Team
- Hardware Team

TLM
TLM-oriented HW/SW Development

- Hardware Development
  - Untimed TLM
  - Timed TLM
  - RTL
  - Reciprocal Improvement

- Software Development
  - Functional SW
  - Time-level SW
  - Real-time SW
  - SW Validation

Reciprocal Improvement
Introduction to SystemC

SystemC

- Is a programming language implemented as a C++ class library such that C++ is a subset of SystemC. It is an IEEE Standard and it is freely downloadable from www.systemc.org.

- Enhances C++ such that it becomes more suitable for specific modelling purposes of embedded systems, i.e., tightly cooperating HW and SW functional modules.

- Supports concurrency, timed behaviour, HW specific data types, and structural descriptions of embedded systems models.

- Incorporates a strict separation of computation and communication tasks within a distributed information processing system.
**Interface** A set of methods which are usable for particular communication channels. It is described using an abstract base class in C++ containing pure virtual methods only.

**Module** An entity for structuring architectural descriptions. It may contain several processes, which execute concurrently.
Channel A communication path between processes. It may be as simple as a signal or rather complex featuring an own internal structure and processes (hierarchical channel). It may be refined in an object-oriented manner.

Port This is the communication access point of a module. A port is typed according to the type of the interface it is bound to.

Summary:
- **Channels** are accessed via interfaces.
- **Modules** access channels via ports.
- **Ports** are related to the type of interface they may connect to.
Concept: Modules are linked to channels via interfaces.

Signals are predefined channels
#include "systemc.h"

struct MAC: sc_module {
    sc_in<int> a;
    sc_in<int> b;
    sc_out<int> acc;
    sc_signal<int> prod;
    void action1() { prod = a * b; }
    void action2() { acc = acc + prod; }

    SCCTOR (MAC) {
        SC_METHOD(action1);
        sensitive << a << b;
        SC_METHOD(action2);
        sensitive << prod;
    }
};
**SystemC Processes**

**Types of processes:**
- SC_METHOD
- SC_THREAD
- SC_CTHREAD

**Declaration and activation:**

```cpp
SC_CTOR(Module) {
    SC_THREAD(thread);
    sensitive << a << b; // static sensitivity list
}

void thread() {
    for (; ;) {
        wait(10, SC_NS); // dynamic sensitivity – time value in nanoseconds
        wait(e);         // dynamic sensitivity – event
    }
}
```
### SystemC Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sc_int&lt;&gt;</code></td>
<td>64-bit signed integer</td>
</tr>
<tr>
<td><code>sc_uint&lt;&gt;</code></td>
<td>64-bit unsigned integer</td>
</tr>
<tr>
<td><code>sc_bigint&lt;&gt;</code></td>
<td>arbitrary precision signed integer</td>
</tr>
<tr>
<td><code>sc_biguint&lt;&gt;</code></td>
<td>arbitrary precision unsigned integer</td>
</tr>
<tr>
<td><code>sc_logic</code></td>
<td>4-valued single bit</td>
</tr>
<tr>
<td><code>sc_bv&lt;&gt;</code></td>
<td>vector of 2-valued single bits</td>
</tr>
<tr>
<td><code>sc_lv&lt;&gt;</code></td>
<td>vector of 4-valued single bits</td>
</tr>
<tr>
<td><code>sc_fixed&lt;&gt;</code></td>
<td>templated signed fixed point</td>
</tr>
<tr>
<td><code>sc_ufixed&lt;&gt;</code></td>
<td>templated unsigned fixed point</td>
</tr>
</tbody>
</table>

...
SystemC TLM

- What is TLM?
  - Communication uses function calls
    ```
    burst_read(char* buf, int addr, int len);
    ```
- Why is TLM interesting?
  - Fast and compact
  - Integrate HW and SW models
  - Early platform for SW development, easy to distribute
  - Early system exploration and verification
  - Verification reuse
How is TLM being adopted?
- Widely used for verification
- TLM for design is starting at major electronics companies

Is it really worth the effort?
- Yes, particularly for platform-based design and verification

What will help proliferate TLM?
- Standard TLM APIs and guidelines
- Availability of TLM platform IP
- Tool support

SystemC TLM Standard
- OSCI TLM 1.0 standard released June 2005
- OSCI TLM 2.0 draft released Nov 2006
TLM Abstraction Levels

- **Algorithmic Level (AL)**
  - Foundation: No Implementation Aspects
  - Functional

- **Programmer’s View (PV)**
  - Foundation: Memory Map
  - Bus generic
  - Masters/Slaves

- **Programmer’s View + Timing (PVT)**
  - Foundation: Timed Protocol
  - Bus architecture
  - Timing approx.

- **Cycle Accurate Level (CA)**
  - Foundation: Clock Edge
  - Word transfers
  - Cycle-accurate

- **RT Level (RT)**
  - Foundation: Registers, logic
  - Signal/Pin/Bit
  - Cycle-accurate

*Model at a few levels that target the “pain” and risk in your D&V flow*
TLM 1.0 API Goals

- **Support** design & verification IP reuse
- Provide common „foundation“ API for TLM
- Usability
- Safety
- Speed
- **Generality**
  - Abstraction Levels
  - HW / SW
  - Different communication architectures (bus, packet, NOC,...)
  - Different protocols
TLM 1.0 Key Concepts

- **Focus on SystemC interface classes**
  - Define small set of generic, reusable TLM interfaces
  - Different components implement same interfaces
  - Same interface can be implemented
    - directly within a C/C++ function
    - via communication with other modules/channels in system

- **Object passing semantics**
  - Uses „message passing“ (= shared nothing - see Wikipedia)
  - Very similar to scfifo
  - Avoids problems with raw C/C++ pointers
  - Avoids problems with concurrent access to shared variables
TLM 1.0 Key Terms

- **Nonblocking**: Means function implementations *can never* call `wait()`
- **Blocking**: Means function implementations *might* call `wait()`
- **Unidirectional**: Data transferred in *one* direction
- **Bidirectional**: Data transferred in *two* directions
- **Write/Peek**: Write potentially overwrites data and can never block. Peek reads most recent valid value. Write/Peek are similar to write/read to a variable or signal
- **Put/Get**: Put queues data. Get consumes data. Put/Get are similar to writing/reading from a FIFO
- **Pop**: A pop is equivalent to a get in which the data returned is simply ignored
- **Master/Slave**: Best to avoid these terms. Instead see definitions of Initiator/Target later
Unidirectional vs. Bidirectional

- **Unidirectional interfaces** send data in only a single direction
  - Flow of control is in either one or both directions
- **Bidirectional interfaces** send data in both directions
  - Flow of control is in either one or both directions

**Examples:**
- A complete read transaction across a bus is bidirectional
- „Place read address on bus“ is unidirectional
- Burst write with a completion status returned is bidirectional
- Send IP-Packet is unidirectional

**Any complex protocol** can be broken down into a set of **unidirectional** and **bidirectional TLM accesses**
Primary TLM 1.0 Interface

- **Primary Unidirectional Interfaces**
  - `tlm_peek_if<T>`
  - `tlm_put_if<T> / tlm_get_if<T>`

- **Primary Bidirectional Interfaces**
  - `tlm_master_if<REQ, RSP> //puts REQ, gets RSP`
  - `tlm_slave_if<REQ, RSP> //gets REQ, puts RSP`
  - `tlm_transport_if<REQ, RSP> //REQ & RSP coupled in 1 function`
The TLM interface can be easily mapped to HW. Understanding this mapping helps you to understand how to use the TLM interface. Note that the TLM interfaces are also useful in non-HW parts of your system (e.g. testbenches, SW modeling).

- **write/peek** have **overwrite semantics** similar to writing to a variable or signal.
- **put/get** have **queuing semantics** similar to writing to a FIFO.

When values propagate asynchronously, combinational logic is implemented.
When values are held across clock edges, hardware registers are implied.
Peek and Pop

- Typical use is decentralized address decoding
  - All slaves peek at transaction
    - Because we’re not consuming the transaction, all slaves will see the same transaction
  - Slave that successfully decodes transaction then pops it
    - See example_4_3 in OSCI TLM 1.0 Kit
- Peek and pop also used in timing annotated models
  - See examples later
Transaction Level Modeling with TLM 1.0 API

Router Example

- Master calls transport() in router
- Router calls transport() in slave through 1 of 2 ports
- Slave implementation of transport() does the work

Arbitration Example

- Symbols:
  - an sc_port
  - an sc_export
  - Port binds to channel
  - A thread

Masters, FIFOs, Arbiter, Slave
Uses the same components as on the previous slide connected in different ways.
Thinking of Interfaces as Contracts - Provides vs. Requires

- `sc_ports` require a given interface type
- `sc_exports` and SystemC channels provide a given interface type
- Interface classes represent interfacing obligations == contract

Requires subset of provided interface, still plug-compatible
1. Interface for the master side of a bus:

```cpp
#include "systemc.h"

class master_if : virtual public sc_interface {
    public:
        virtual void write(int priority, int address, int data) = 0;
        virtual void read(int priority, int address, int* data) = 0;
};
```

Note: The interface `master_if` is derived from the class `sc_interface`. It just creates two methods: `read`, `write`. The bus is then created as a module, derived from the interface class.

(Source: J. Aynsley, A. Fitch (Doulos Ltd.) - Euro DesignCon 2004 tutorial)
2. Bus model

class Bus : public sc_module, public master_if {
    public:
        sc_in<bool> clock;
        sc_port<slave_if, 2> slave_port;

    Bus() :
    {
        SC_THREAD(busarbiter);
        sensitive << clock.pos(); //static sensitivity
    }

    void write(int priority, int address, int data);
    void read (int priority, int address, int* data);
    void busarbiter (...);

    private:
        bool* request;
        sc_event* proceed;
};
3. Bus write function (part of)

```c
void Bus::write(int priority, int address, int data) {
    request[priority] = true;
    wait(proceed[priority]);
    ...
}
```

Note: Set flag when master attempts to write to the bus and wait until permission is given. The call to `wait` overrides the sensitivity list of the master - it is an Interface Method Call, IMC.
4. Use of the bus

- Port declaration in the master module: `sc_port<master_if> busport;`
- Create and bind instances:

```c++
Bus MyBus(”MyBus”);
Master MyMaster(”MyMaster”); // bind port to interface of the bus
MyMaster.busport(MyBus);
```

Note: Now the code in `MyMaster` may invoke the read and write methods implemented in the bus.
Improving on TLM 1.0

- Agreeing on the message format (tlm_bus)
- Timing annotation
- Efficiency for passing long messages (pass-by-pointer)
Definition of Terms

System
Initiator

System
Target

TLM Initiator Port

TLM Target Export

TLM Initiator
TLM Target

TLM Initiator
TLM Target

TLM Target
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 payload
Effective Pass-by-Value

T get();
Pass by value

void put( const T& );
Pass by const reference
Obligation not to modify actual argument during call
Obligation not to use reference after call (must copy value)

void get( T& );
Pass by reference
Obligation not to read or modify actual argument during call
Obligation not to read reference during call
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
## TLM 1 Core Interfaces

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

PV Initiator — 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 $\geq$ latency
## TLM 2.0 Core Interface

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

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
Explicit vs. Implicit Timing

nb_put(t);
wait(del);

nb_put(t, del)

wait(del);
nb_put(t); nb_put(t, del)

Initiator

Initiator

nb_put(t)

nb_put(t, del)

wait(del)

No context switch!

Huss (ISS)  System Modelling  SS 2007  65 / 68
TLM Summary

TLM methodology is:
- HDL independent.
- An IEEE Standard.
- A key stone in modern embedded systems design flows.
Questions ?