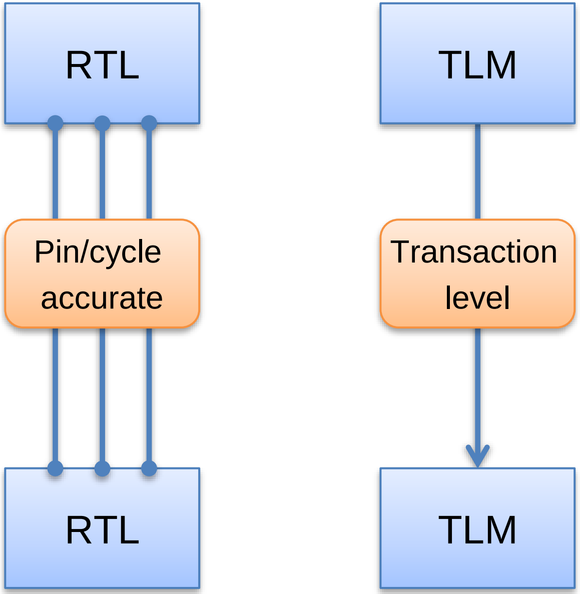
SystemC TLM

Stefano Spellini

Transaction Level Modeling (TLM)

```
process(clock)
IF (clock'event and clock = '1')
THEN
CASE fsm_state IS:
          WHEN s0 =>
           request_port <= '1';
           fsm_state := s1;
          WHEN s1 =>
           IF (grant_port = '1')
            THEN
            fsm_state := s2;
          WHEN s2 =>
           data_port <= data;
…          addr_port <= addr;
```



write(data,addr)
No clocks! No pins!

Simulate every event

100-10.000x faster simulation

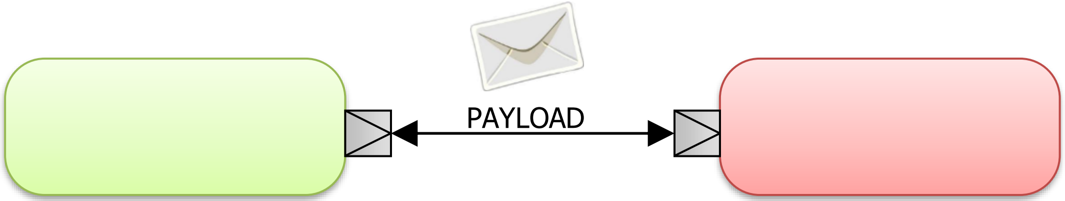
Motivations for TLM

- 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

Modeling at TLM

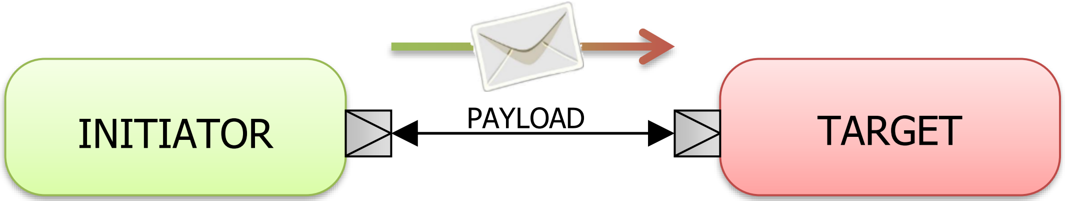
TLM transaction

- Relies on the notion of *transaction*
  - Data transfer from a design module to another
    - Read/Write operation
  - Represented by a payload object
    - Exchanged with primitive calls
    - Contains both data and control information



TLM actors: initiators and targets

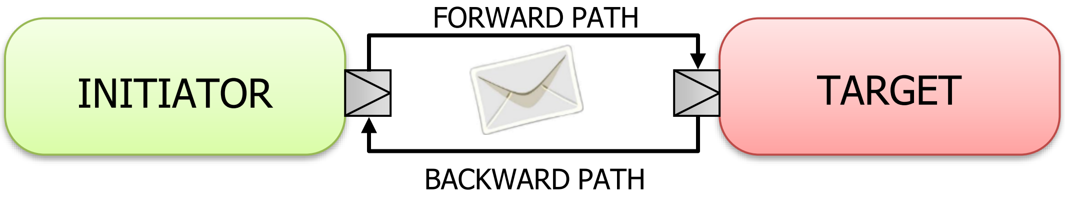
- Initiator
  - Initiates a transaction
    - Creates a transaction object
    - Calls target (or interconnect) methods to deliver it
- Target
  - Acts as final destination of a transaction
    - Elaborates the payload
- Master/Slave behavior typical of busses



TLM paths

- Forward path
  - Calling path from the initiator in the direction of a target
    - The Master calls the Slave

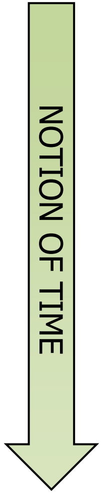
- Backward path
  - Calling path by which a target makes method call backs in the direction of the initiator
    - The Slave answers to the Master



TLM Coding Styles

- Not levels of abstraction!
  - Defines how time and data are related together, according to designers’ perception
  - More of a guideline, designers enjoy several degrees of freedom

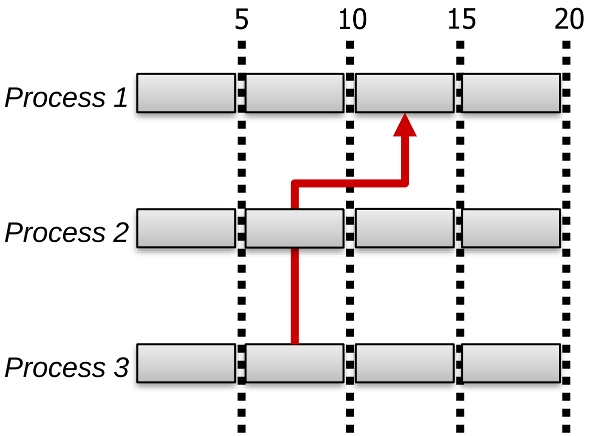
- Specify how time and data are related together
  - Untimed
    - Blocking interface
    - Predefined synchronization points
  - Loosely timed
    - Blocking interface
    - Two synchronization points (invocation and return)
    - Temporal decoupling
  - Approximately timed
    - Non blocking transport interface
    - Timing annotation and multiple phases during the lifetime of a transaction
    - Beginning/end of request
    - Beginning/end of response



NOTION OF TIME

Temporal Decoupling

- Loosened Synchronization
  - Individual processes can run ahead for a certain time slice without advancing general simulation time
    - Delayed Synchronization with the system
    - Explicit Synchronization point
    - Very fast simulation



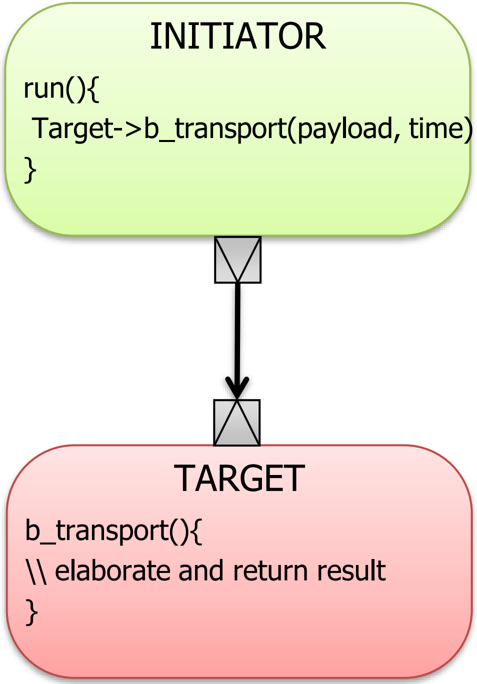
Generic Payload

- Standard transaction objects
  - Modeled on memory-mapped bus
    - Command,
    - address,
    - data,
    - byte enables, ..
  - Transfered with function calls

Attribute	Type
Command	Tlm_command
Address	Uint64
Data pointer	Unsigned char *
Data length	Unsigned int
Byte enable pointer	Unsigned char *
Byte enable length	Unsigned int
Streaming width	Unsigned int
DMI hint	Bool
Response status	Tlm_response_status
Extensions	Tlm_extension_base*

TLM blocking transport interface

- Support
  - Untimed
  - Loosely timed
- Initiator completes a transaction with the target in one function call
  - 2 synchronization points
    - Invocation
    - Return
- Uses only forward path



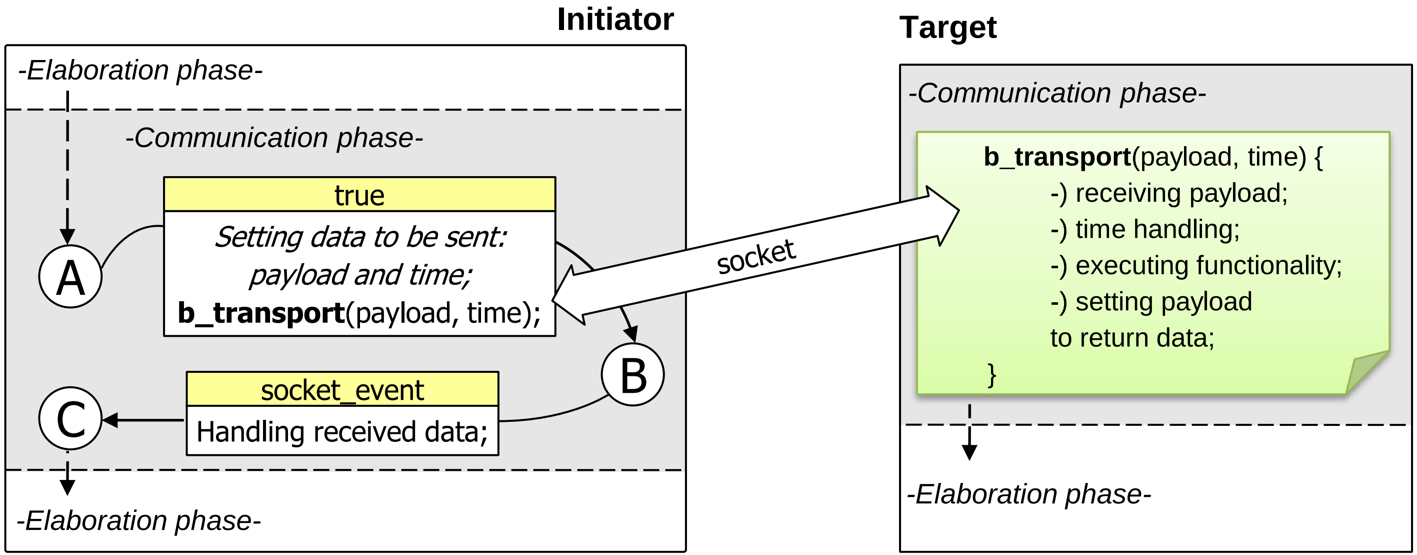
INITIATOR

```
run(){
 Target->b_transport(payload, time)
}
```

TARGET

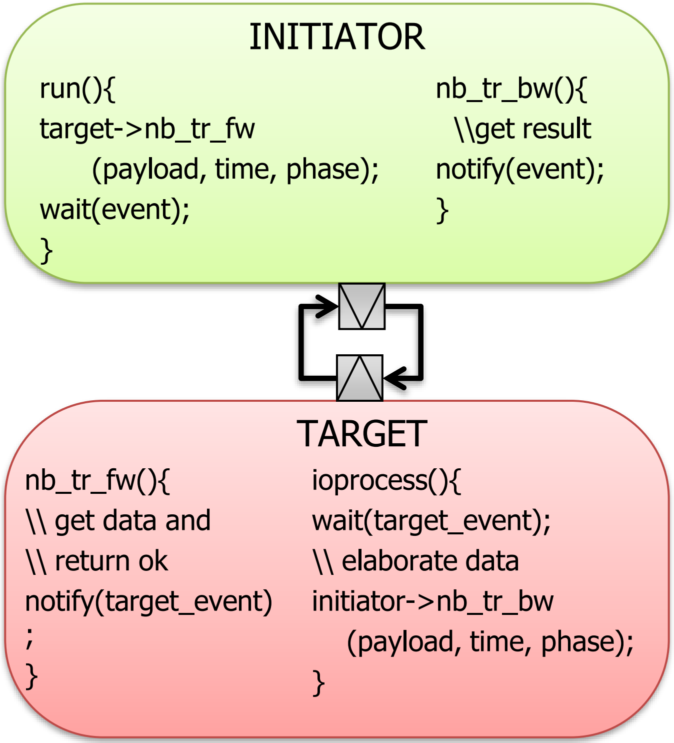
```
b_transport(){
\\ elaborate and return result
}
```

TLM blocking transport interface (Cont.d)



TLM non-blocking transport interface

- Support
  - Approximately timed
- Detail a sequence of interactions between initiator and target
  - Phases
    - Begin/end request
    - Begin/end response
- Uses both forward path and backward path
- Classic protocol:
  - 4-phase handshaking protocol
    - AT4



INITIATOR

```
run(){
target->nb_tr_fw
      (payload, time, phase);
wait(event);
}
```

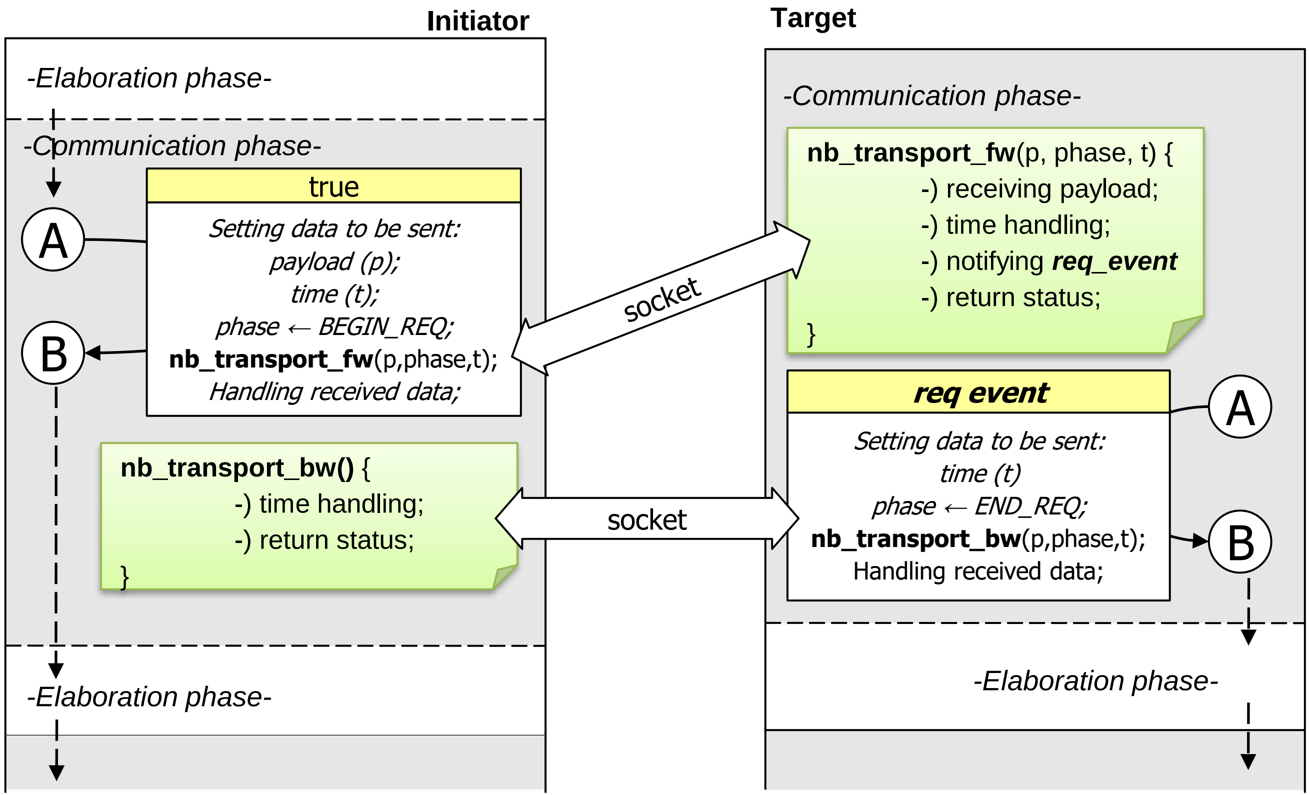
```
nb_tr_bw(){
  \\get result
notify(event);
}
```

TARGET

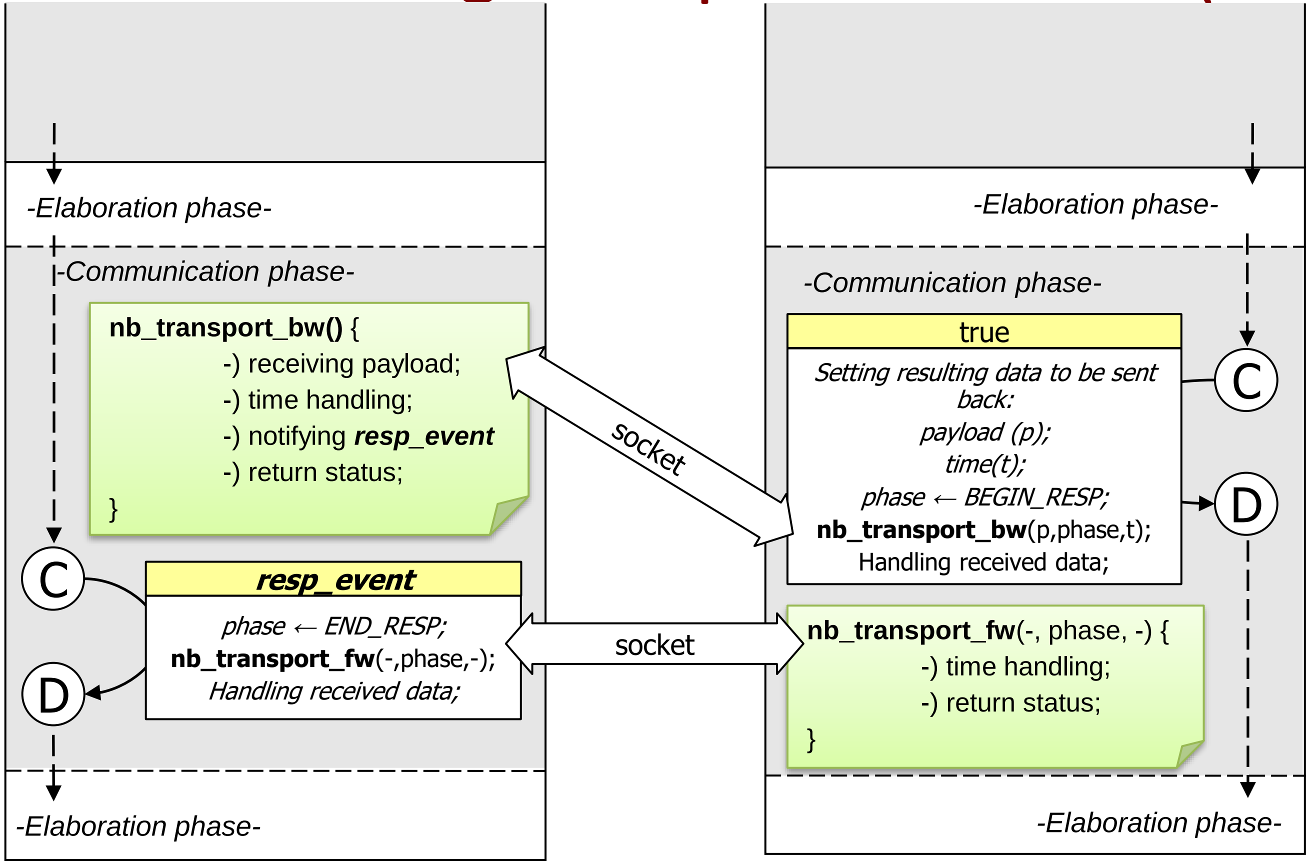
```
nb_tr_fw(){
\\ get data and
\\ return ok
notify(target_event)
;
}
```

```
ioprocess(){
wait(target_event);
\\ elaborate data
initiator->nb_tr_bw
     (payload, time, phase);
}
```

TLM non-blocking transport interface (Cont.d)



TLM non-blocking transport interface (Cont.d)



SystemC Installation (ERRATA CORRIGE)

- Download SystemC
  - http://www.accellera.org/downloads/standards/systemc
  - From the elearning portal (SystemC 2.3.3 Source Code)
- Unzip the archive

```
$> tar -xzvf systemc-2.3.3.tar.gz
```

- Configure the installation

```
$> cd systemc-2.3.3
$> mkdir build
$> mkdir -p ~/pse_libraries/systemc
$> cd build
$> cmake -DCMAKE_INSTALL_PREFIX=/home/<your_user>/pse_libraries/systemc
         -DBUILD_SHARED_LIBS=OFF ..
```

- Compile

```
$> make -j
$> make install
```

SystemC into action

- Uncompress the archive
  - $ tar xzf 02_sources.tar.gz
  - $ cd 02_sources/
  - $ ls
    - dist div root root_lesson root_comparison
  - $ cd root_lesson
  - $ ls
    - AT4 LT UT RTL

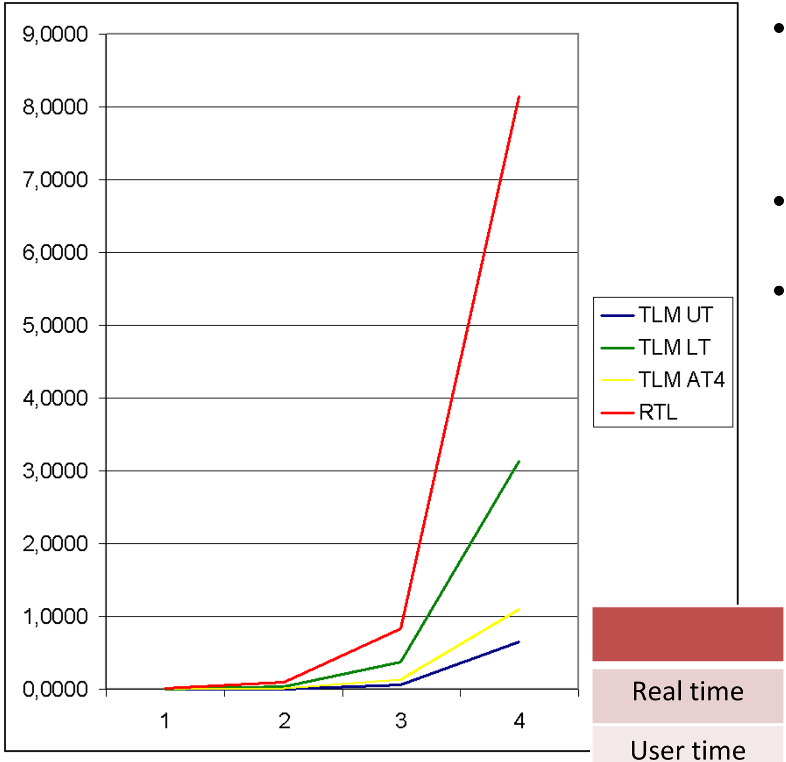
- Compile and run each TLM implementation, e.g.:
  - $ cd UT/
  - $ mkdir build && cd build
  - $ cmake –DCMAKE_PREFIX_PATH=/home/<your_username>/pse_libraries/systemc ..
  - $ make –j
  - $ ./root_UT

SystemC into action

- Go to the root_comparison folder
  - Root with 128000 interactions between target and initiator
- Compile all the TLM/RTL implementations, e.g.:
  - $ cd RTL/
  - $ mkdir build && cd build
  - $ cmake –DCMAKE_PREFIX_PATH=/home/<your_username>/pse_libraries/systemc ..
  - $ make –j
  - $ ./root_RTL

- Compare execution time with the ***time*** shell command
  - Real time
    - Elapsed time between invocation and termination
  - User time
    - Amount of time spent by the CPU performing actions for a program
  - System time
    - Amount of time spent by the CPU performing system calls on behalf of the program

SystemC into action



- TLM focuses on functionality
  - Less simulation events
  - Less time accurate
  - Faster
- RTL is more precise
  - Slower
- Use the command **`time`**
  - `$> time ./root*`
  - **Imprecise** but simple to use!
  - Always use **long simulations**!

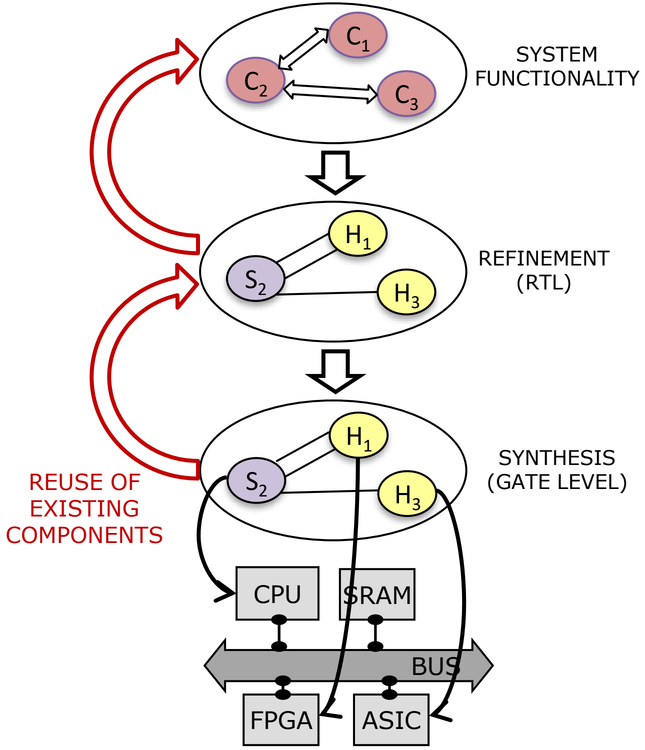
	UT	LT	AT4	RTL
Real time				
User time				
System time				

Connecting RTL and TLM

BASIC TRANSACTORS

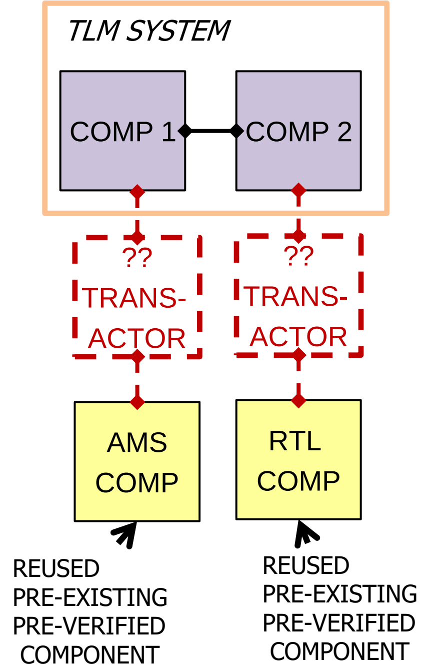
Design

- Design flow
  - In theory: top-down approach
    - Choose and describe the functionality
    - Implement it with abstract communication (TLM)
    - Implement proper communication protocol (RTL)
    - Gate-level synthesis
  - In practice: reuse
    - Fundamental to meet time to market constraints and to save money (design+verification)



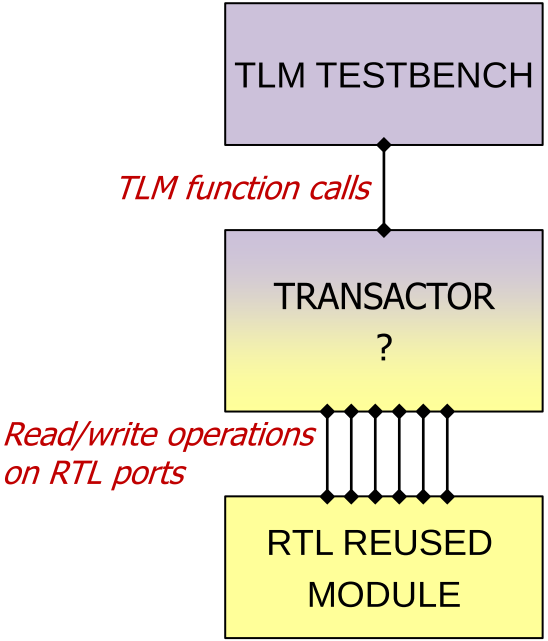
Mixed Modeling

- Mixed modeling
  - Implied by reuse
    - Integrate components developed at different levels of abstractions
  - Transactor
    - Translator in the communication between modules implemented at different levels of abstraction
    - Allows mixed modeling and testbench reuse


TLM SYSTEM
COMP 1
COMP 2
??
TRANS-ACTOR
??
TRANS-ACTOR
AMS COMP
RTL COMP
REUSED PRE-EXISTING PRE-VERIFIED COMPONENT
REUSED PRE-EXISTING PRE-VERIFIED COMPONENT

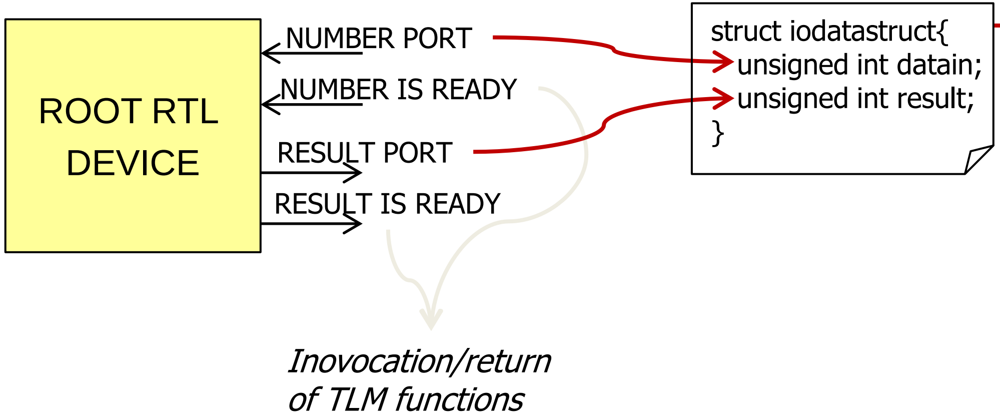
Transactor: main idea

- Two APIs
  - Higher level of abstraction (TLM)
    - TLM standard functions
  - Lower level of abstraction (RTL)
    - Sequence of read/write operations on RTL ports



Transactor: mapping

- Define a mapping between RTL ports and TLM payload fields

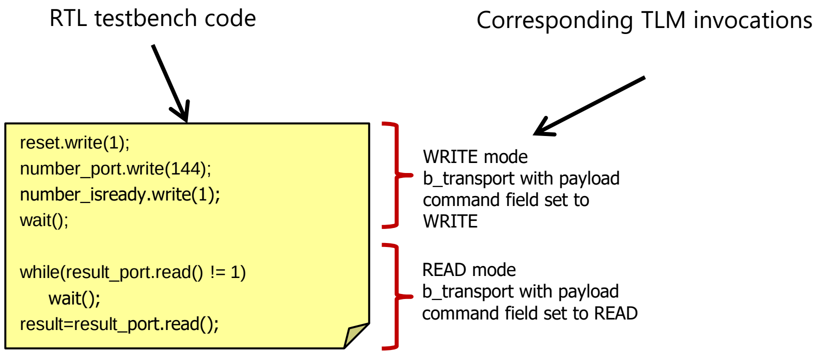


```
struct iodatastruct{
unsigned int datain;
unsigned int result;
}
```

Attribute
Command
Address
Data pointer
Data length
Byte enable pointer
Byte enable length
Streaming width
DMI hint
Response status
Extensions

Transactor: communication

- Translation of TLM function calls to sequences of RTL signal operations
  - From blocking/non blocking primitives to cycle accurate temporization and handshaking
  - Root:
    - Two modes, read and write
    - Correspond to b_transport invocations with different command parameters



RTL testbench code

Corresponding TLM invocations

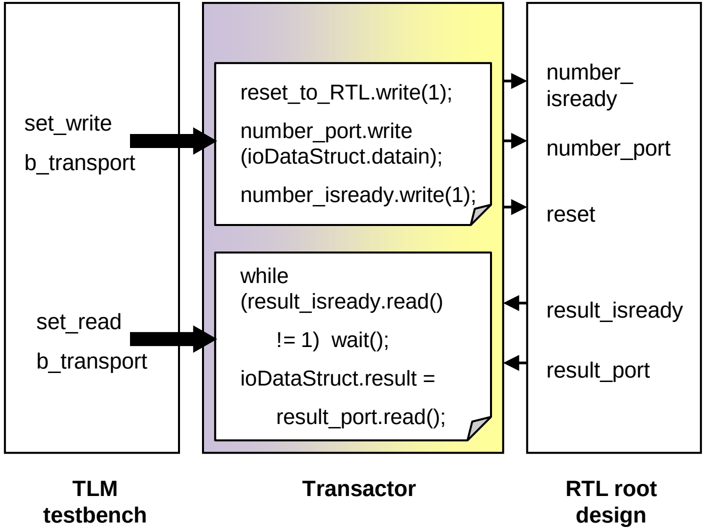
```
reset.write(1);
number_port.write(144);
number_isready.write(1);
wait();

while(result_port.read() != 1)
     wait();
result=result_port.read();
```

WRITE mode
b_transport with payload
command field set to
WRITE

READ mode
b_transport with payload
command field set to READ

Transactor: structure

- Transactor structure
  - B_transport interface towards TLM
  - Depending on the command
    - Activates the WRITE process to set number_port and number_isready port
    - Activates the READ process to get the value of the result_port and result_isready port



```
reset_to_RTL.write(1);
number_port.write
(ioDataStruct.datain);
number_isready.write(1);
```

```
while
(result_isready.read()
      != 1)  wait();
ioDataStruct.result =
      result_port.read();
```

Transactor: Example

```
void root_RTL_transactor::b_transport
        (tlm::tlm_generic_payload& trans, sc_time& t){
 tlm::tlm_command cmd = trans.get_command();
 switch (cmd) {
  case tlm::TLM_WRITE_COMMAND:
   ioDataStruct = *((iostruct*) trans.get_data_ptr());
   begin_write.notify();
   wait(end_write);
   break;
  case tlm::TLM_READ_COMMAND:
   ioDataStruct = *((iostruct*) trans.get_data_ptr());
   begin_read.notify();
   wait(end_read);
   break;
  default:
   break;
}
```

```
void root_RTL_transactor::WRITEPROCESS(){
 while (true) {
  wait(begin_write);
  reset_to_RTL.write(1);
  p_Out_data.write(ioDataStruct.datain);
  p_Out_enable.write(1);
  end_write.notify();
  wait();
}}
```

```
void root_RTL_transactor::READPROCESS(){
 while (true) {
  wait(begin_read);
  while(p_In_enable.read() != 1)
  wait();
  ioDataStruct.result=p_In_data.read();
  end_read.notify();
}}
```

ASSIGNMENT (PART 1)

Fixed-Point Rational Binary Multiplication

- Algorithm for Binary Multiplication
- Requirements:
  - Operands: 16 bit **Unsigned Fixed-Point rational** value
    - 8 bit integer
    - 8 bit fraction
  - **Forbidden** use of sc_fixed
  - Pay attention to the span of the **output**!
  - Max port width: **8 bit**
    - **No Limit on internal signals**
  - Handshake protocol

```
unsigned int product, temp;
unsigned int a, b, result;
temp = a;
product = 0;
for (0 to size of temp){
            if (i-th bit of b is 1)
            product += temp;
            temp shifted of one bit to the
left;
}
result = product;
```

- Well known algorithm, many examples on the web
  - Search for “Binary Multiplication” on Google!

Example with integers

6 * 9 = 54

Initialization:
a = 0110; // 6
b = 1001; // 9
temp = 00000110; // 6
product = 00000000;

Cicle 0:
b[0] is 1, thus:
product = 0 + 0110 = 00000110; // 6
temp = 00001100; // 12

Cicle 1:
b[1] is 0, thus:
temp = 00011000; // 24

Cicle 2:
b[2] = 0, thus:
temp = 00110000; // 48

Cicle 3:
b[3] = 1, thus:
product = 00000110 + 00110000 = 00110110; // 54
temp = 01100000; // 96
result = product; // 54

Fixed-point Matrix Multiplication

- Let **A** an **m*n** matrix and **B** an **n*p** matrix
- The *matrix product* **C** = **AB** is defined to be the **m*p** matrix such that
  - $c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$
- Implement a **Fixed-point Matrix Multiplication Module** in **Verilog @ RTL**
  - You can develop:
    - Two modules (one for the multiplication step and one for the matrix element-by-element addition)
    - A single module doing everything
- Also write a testbench for the design
- Requirements:
  - **Module must be synthesizable**
  - **DO NOT USE ANY IEEE LIBRARY FOR IMPLEMENTING FIXED-POINT MULTIPLICATION**
  - **Matrix dimensions must be non-fixed**
  - **You must check the compatibility of A and B for multiplication (cols(A) = rows(B))**

Fixed-point Matrix Multiplication

- Let **A** an **m*n** matrix and **B** an **n*p** matrix
- The *matrix product* **C** = **AB** is defined to be the **m*p** matrix such that
  - $c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$

- Implement **Fixed-point Matrix Multiplication Module** in **SystemC @ TLM**
  - **Choose a coding style**
  - **Remember: TLM is a higher abstraction level -> abstract the protocols/behaviors that can be abstracted**

Assignment Report

- Write a Report for the current assignment
- Contents of the RTL part:
  - **FSMD** and **EFSM** of the module(s)
  - **Motivate the implementation choices you made**
  - Show simulation traces
    - Detail the developed protocol and its timings
  - Demonstrate it is synthesizable on the FPGA of reference
    - Pynq Z1 by Xilinx

- Contents of the TLM part:
  - **Motivate the implementation choices you made**
  - Motivate the chosen coding style (LT, UT, AT4)
  - Demonstrate that TLM is faster than RTL
    - Compare simulation times

Assignments Folder Structure

- **ONE directory** compressed in **ONE archive (tar.gz)**
  - VRXXXXXX_Name_Surname
  - VRXXXXXX_Name_Surname**.tar.gz**
    - tar czf VRXXXXXX_Name_Surname.tar.gz VRXXXXXX_Name_Surname/
- **Within** the directory: **2 sub-directory**
  - **Report:**
    - **Multiple PDFs**: one PDF file per assignment part
      - VRXXXXXX_Name_Surname_YY.pdf
      - YY = 01, 02, 03
  - **Solution:**
    - **1 Directory per report!**
      - **01, 02, 03**

Example of Folders Tree

- VR123456_Stefano_Spellini/
  - Reports/
    - VR123456_Stefano_Spellini_01.pdf
    - VR123456_Stefano_Spellini_02.pdf
    - VR123456_Stefano_Spellini_03.pdf
    - ….
  - Solutions/
    - 01/
      - Something
    - 02/
      - Something
        - Something
          - include, CMakeLists.txt, src
    - 03/
      - ….

- **Everything within VR123456_Stefano_Spellini.tar.gz**