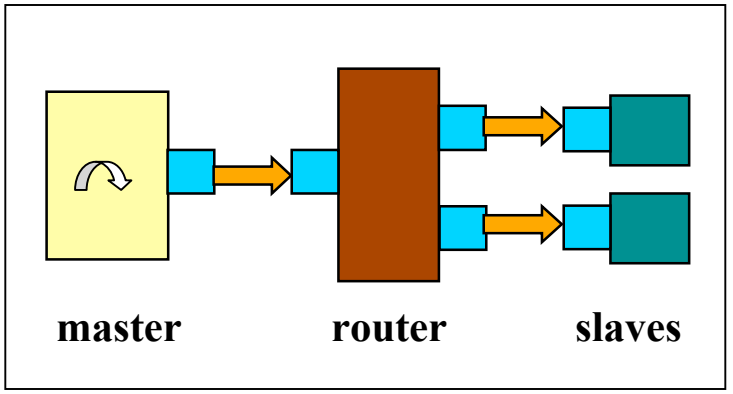
A Tutorial Introduction to the SystemC TLM Standard

Stuart Swan
Cadence Design Systems, Inc
February 2006

SystemC Transaction Level Modeling



- *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

SystemC Transaction Level Modeling

- *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

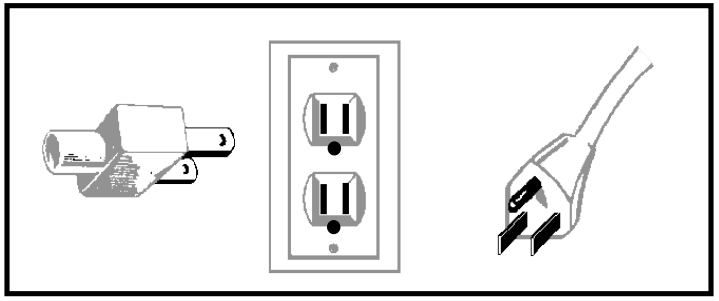
May 2005: OSCI Releases SystemC TLM Standard

- TLM Standard API provides the foundation layer to develop interoperable SystemC TLM IP
- Full press release available at www.systemc.org
- Companies endorsing TLM standard within press release:
  - Cadence, CoWare, Forte, Mentor, Philips, ST, Synopsys
  - Atrenta, Calypto, Celoxica, Chip Vision, ESLX, Summit, Synfora
  - OCP-IP
- TLM kit, whitepaper, and examples publicly available at www.systemc.org
- See also June 6 2005 online articles in EETimes and EDN
- TLM standard is already in use in industry
- IEEE standardization process to begin soon

TLM API Goals

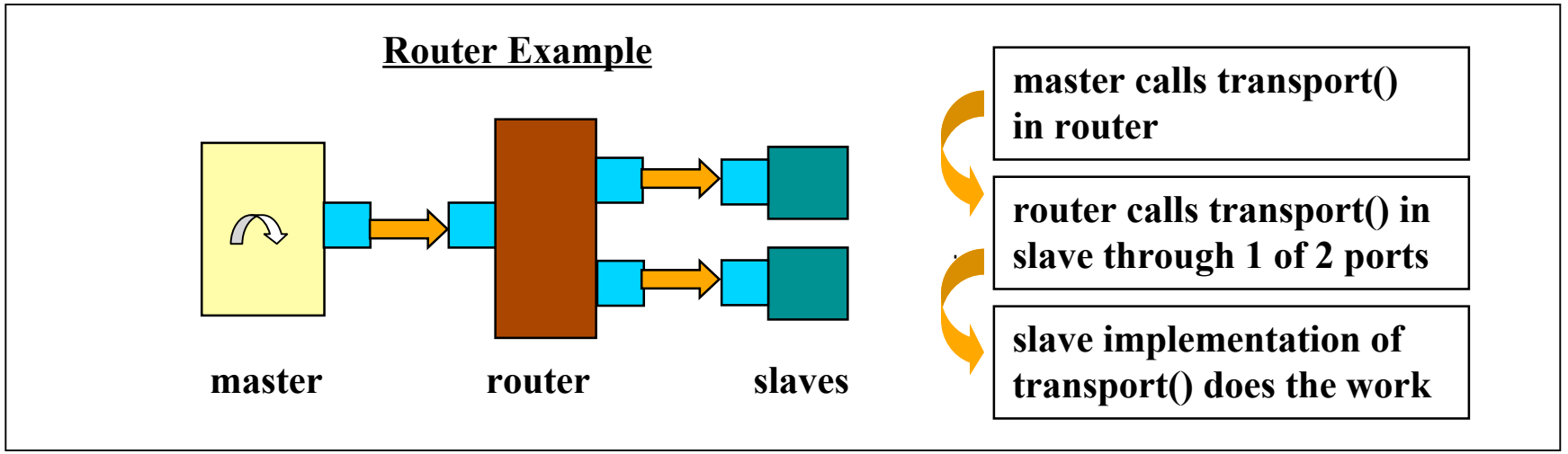
- Support design & verification IP reuse
- Provide common TLM recipe
- Usability
- Safety
- Speed
- Generality
  - Abstraction Levels
  - HW / SW
  - Different communication architectures (bus, packet, NOC, ...)
  - Different protocols

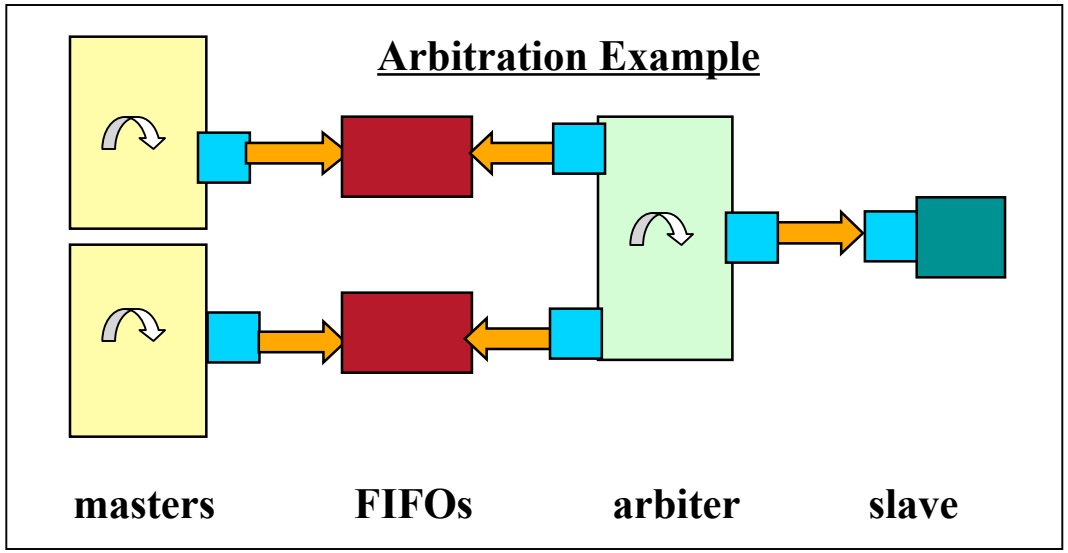
Key Concepts

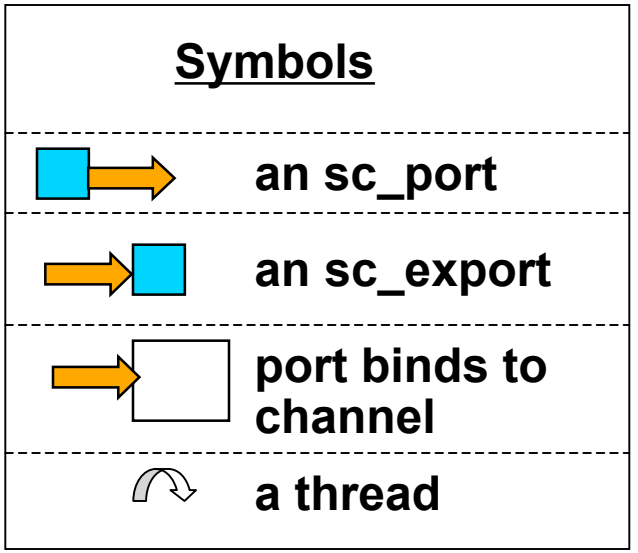


- Focus on SystemC interface classes
  - Define small set of generic, reusable TLM interfaces
  - Different components implement same interfaces
  - Same interface can be implemented
    - directly within a C/C++ function, or
    - via communication with other modules/channels in system
- Object passing semantics
  - Similar to sc_fifo, effectively pass-by-value
  - Avoids problems with raw C/C++ pointers
  - Leverage C++ smart pointers and containers where needed

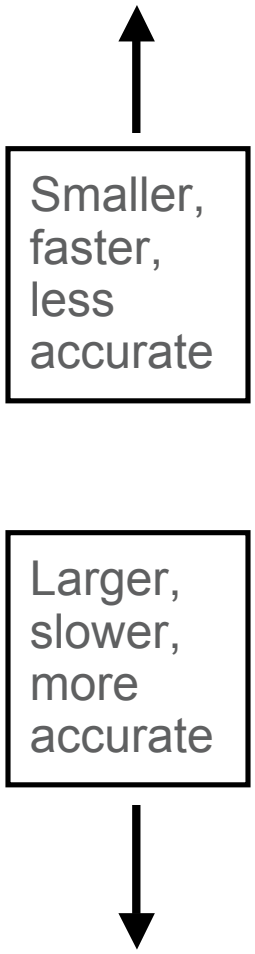
Transaction Level Modeling with the TLM API







TLM Abstraction Levels

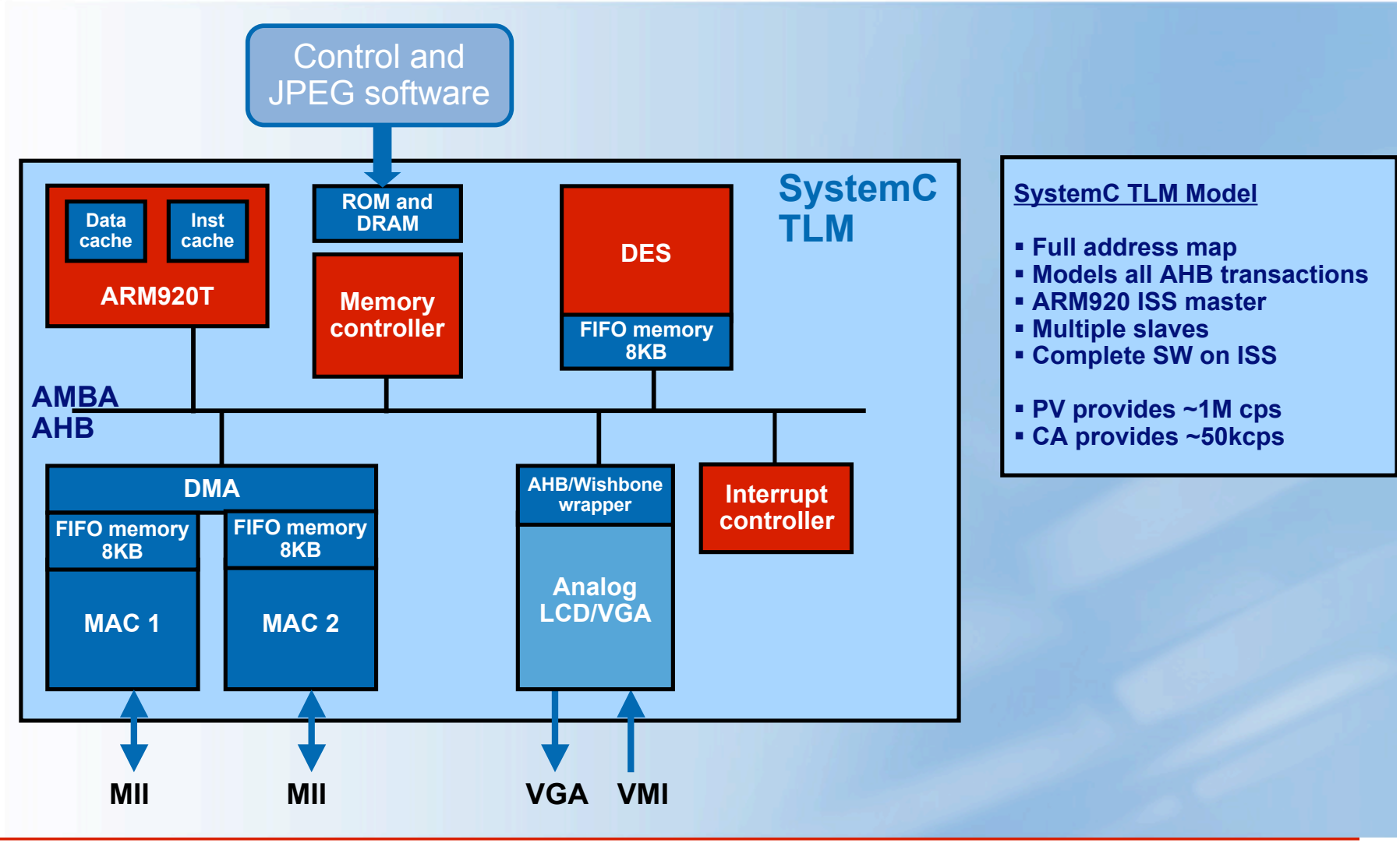


Algorithmic Level (AL)	Functional
Foundation: No Implementation Aspects	
Programmer's View (PV)	Bus generic
Foundation: Memory Map	Masters/Slaves
Programmer's View + Timing (PVT)	Bus architecture
Foundation: Timed Protocol	Timing approx.
Cycle Accurate Level (CA)	Word transfers
Foundation: Clock Edge	Cycle-accurate
RT Level (RT)	Signal/Pin/Bit
Foundation: Registers, logic	Cycle-accurate

Model at a few levels that target the "pain" and risk in your D&V flow

Example TLM Application #1

Wireless Picture Frame based on ARM920T

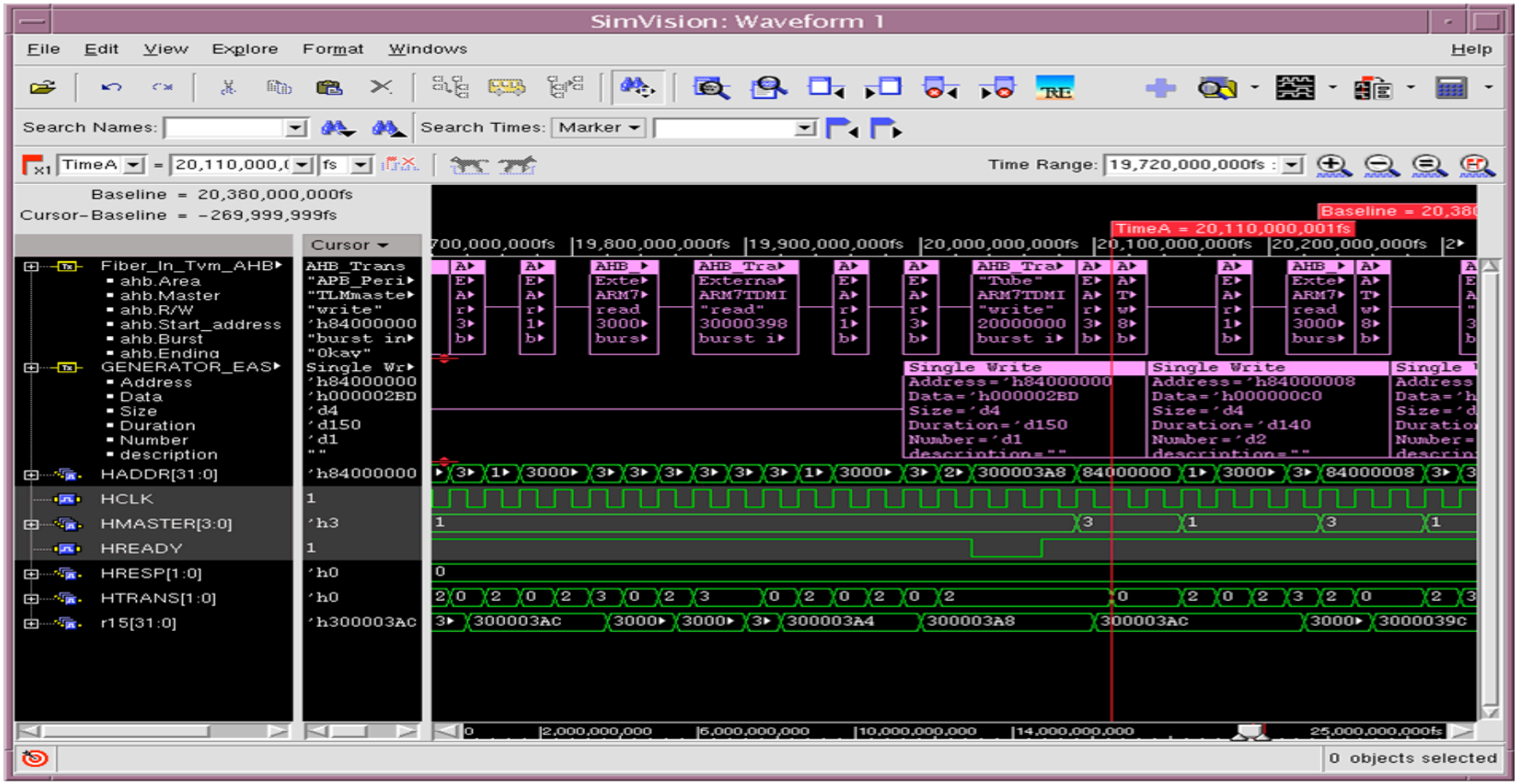


SystemC TLM Model

- Full address map
- Models all AHB transactions
- ARM920 ISS master
- Multiple slaves
- Complete SW on ISS

- PV provides ~1M cps
- CA provides ~50kcps

TLM Eases System Debug and Analysis



Review of Key TLM 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
- **Poke/Peek:** Poke overwrites data and can never block. Peek reads most recent valid value. Poke/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:** A master initiates activity by issuing a *request*. A slave passively waits for requests and returns a *response*.

Unidirectional versus Bidirectional

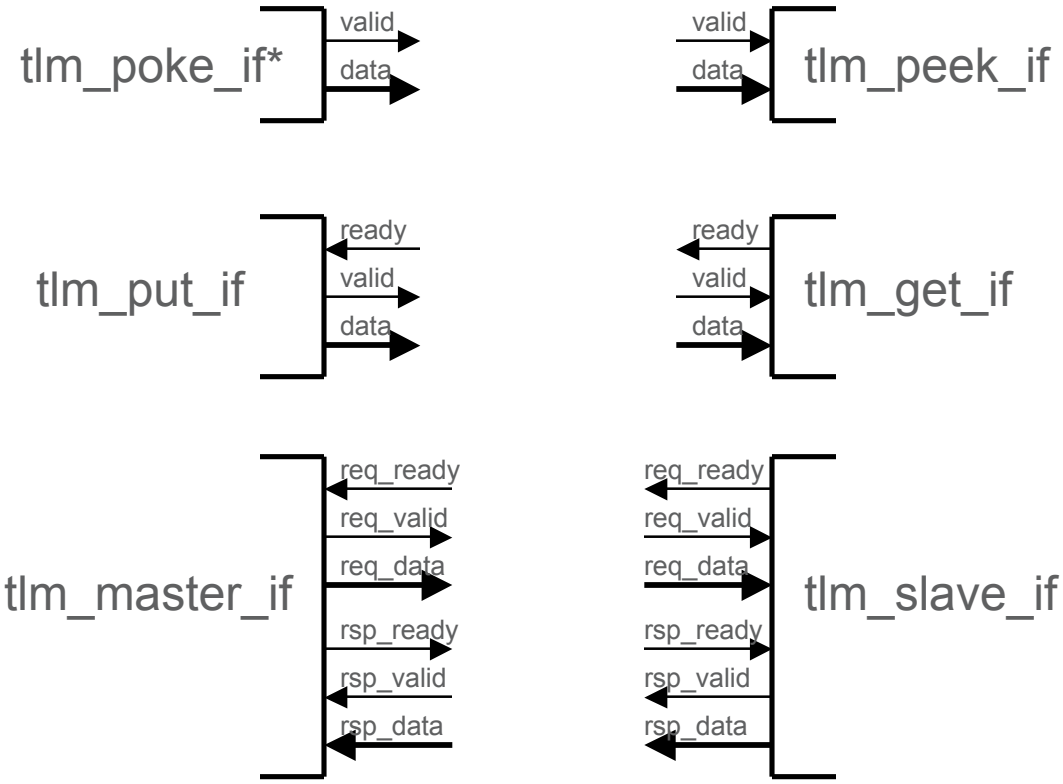
- Unidirectional interfaces send data in only a single direction, and flow of control is in either or both directions.
- Bidirectional interfaces send data in both directions, and flow of control is in either or both directions.
- Examples:
  - A complete read transaction across a bus is *bidirectional*
  - “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 accesses that use the TLM API

Primary TLM Interfaces

- Primary Unidirectional Interfaces
  - tlm_poke_if<T> / tlm_peek_if<T>
  - tlm_put_if<T> / tlm_get_if<T>
- Primary Bidirectional Interfaces
  - tlm_master_if<REQ, RSP>
  - tlm_slave_if<REQ, RSP>
  - tlm_transport_if<REQ, RSP>

- *Note: tlm_poke_if should be added to OSCI TLM kit soon*

Hardware Implied by TLM Interfaces



The TLM interfaces can be easily mapped to HW. Understanding this mapping helps you to understand how to use the TLM interfaces.

Note that the TLM interfaces are also useful in non-HW parts of your system (e.g. testbenches, SW modeling).

Poke/peek have overwrite semantics similar to writing to a variable or signal

Put/get have queuing semantics similar to writing to a FIFO

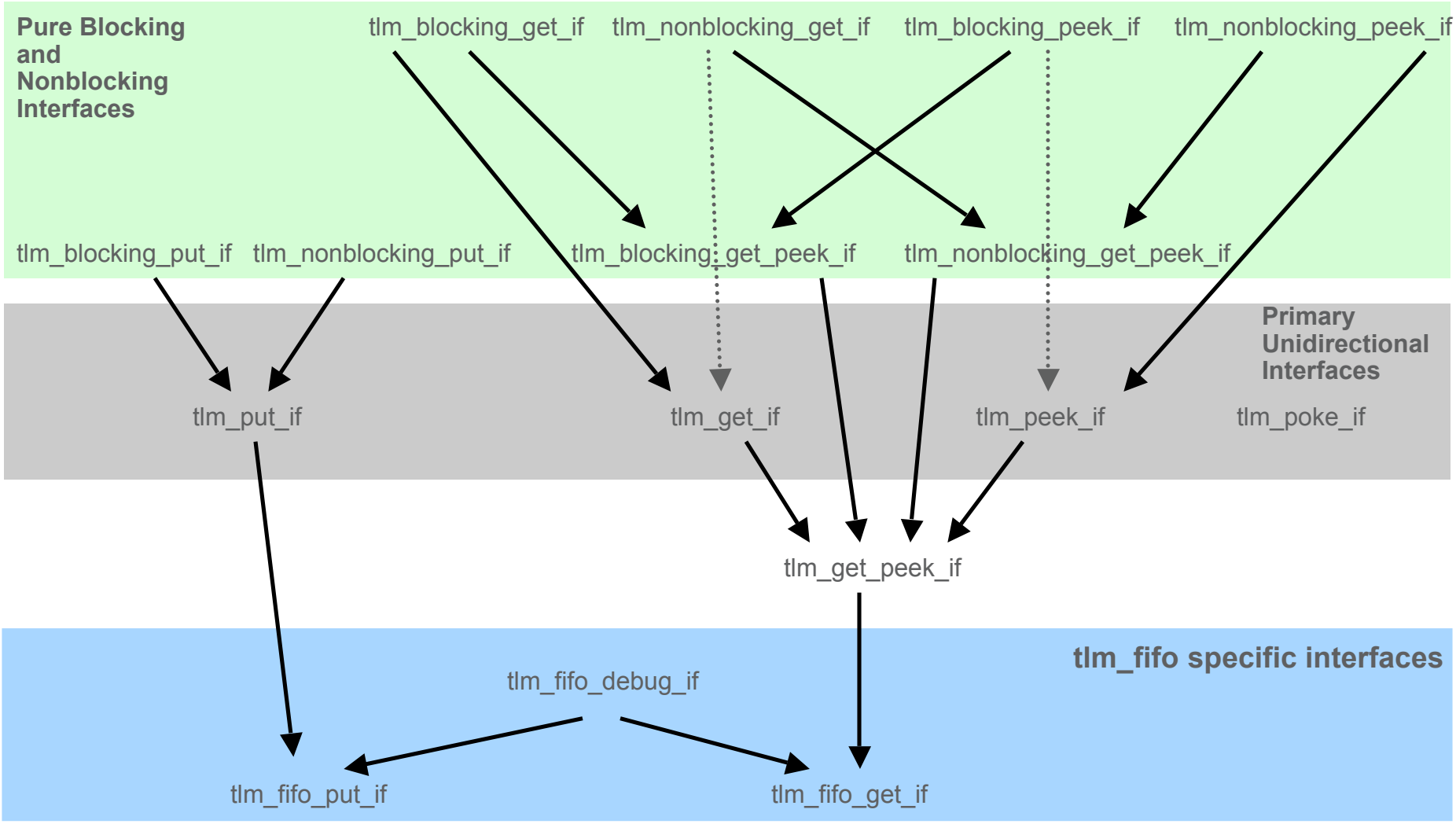
When values propagate asynchronously, combinational logic is implied.

When values are held across clock edges, hardware registers are implied.

tlm_transport_if implies same HW as tlm_master_if, but also requests and responses are tightly coupled.

tlm_poke_if is not yet in OSCI TLM standard, should be added soon.

TLM Unidirectional Interfaces Inheritance Diagram



Peek and Pop

- Typical use for peek and pop is decentralized address decoding by multiple slaves
  - All slaves peek at transaction
    - Because we're not consuming the transaction, all slaves will see the same transaction
  - The slave that successfully decodes the transaction issues a pop by calling get and ignoring the returned data
    - See example_4_3 in OSCI TLM kit

tlm_transport_if vs. tlm_master_if

- The tlm_transport_if interface is used to model *tightly coupled* request and response pairs
- This interface can be implemented by a single function call, OR by two fifos in each direction.
- If you need to model *loosely coupled* requests and responses, do not use tlm_transport_if. Instead interface directly to the request and response fifos using the tlm_master_if.

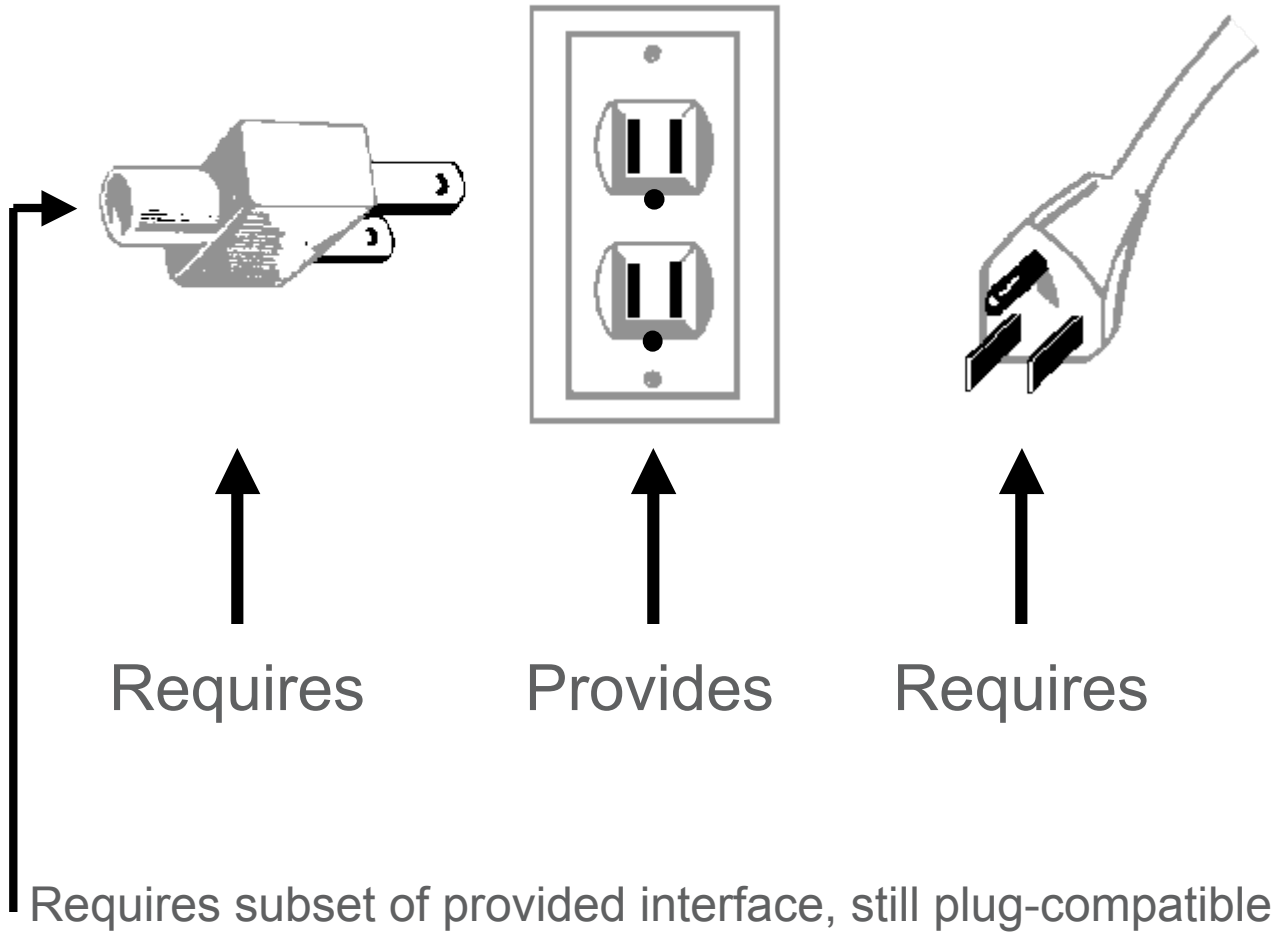
```
// bidirectional blocking transport interface

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

```
// bidirectional master interface

template < typename REQ , typename RSP >
class tlm_master_if :
  public virtual tlm_put_if< REQ > ,
  public virtual tlm_get_peek_if< RSP > {};
```


Thinking of Interfaces as Contracts – Provides versus Requires

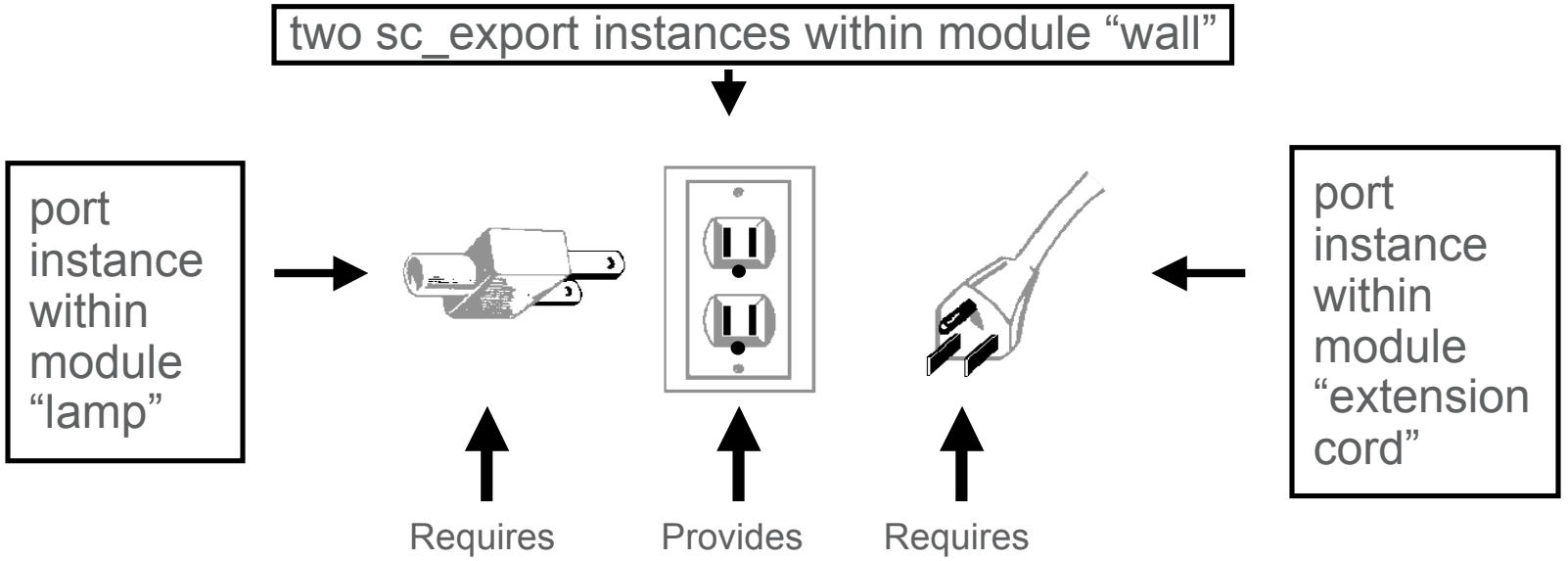


Importance of sc_export in SystemC 2.1

- sc_ports facilitate modular design by precisely declaring interfaces ***required*** at a module boundary
- sc_exports facilitate modular design by precisely declaring interfaces ***provided*** at a module boundary
- sc_ports and sc_exports allow interfaces to be passed through each level of the hierarchy
- Use of sc_port and sc_export improves modularity by avoiding reliance on explicit multilevel paths
- sc_export permits direct function call interfaces for TLM without introduction of extra process switches

Summarizing Provides / Requires

- A channel that implements an interface class by inheriting from that class ***provides*** that interface to the outside world
- An sc_export<IF> member within a module or channel ***provides*** that interface to the outside world
- An sc_port<IF> member within a module or channel ***requires*** that interface from the outside world



Design your components to maximize opportunities for reuse

- In SystemC, channels and sc_exports that ***provide*** more than ports actually ***require*** are still “plug compatible”.
  - Mechanism that achieves this is C++ implicit conversion to base classes
  - Having a hierarchy of interface classes is thus the key enabler for this feature
- ***Require*** the most minimal interfaces that are possible in a given situation
  - e.g. sc_port<tlm_nonblocking_get_if<T> > rather than sc_port<tlm_get_if<T> >
- ***Provide*** the maximal interfaces that makes sense in a given situation
  - e.g. sc_export<tlm_put_if<T> > rather than sc_export<tlm_blocking_put_if<T> >

Leverage tlm_fifo to connect incompatible interfaces

- tlm_fifo ***provides*** all the put, get and peek interfaces in blocking and nonblocking forms, so tlm_fifo can be used to connect any two unidirectional tlm interfaces except tlm_poke_if.
- For example, can do blocking puts into tlm_fifo, and nonblocking gets out of it.

Develop and use a library of generic TLM components

- The OSCI TLM already contains several good examples of generic TLM components:
  - tlm_fifo<T>
  - tlm_req_rsp_channel<REQ, RSP>
  - tlm_transport_channel<REQ, RSP>
  - router<ADDRESS, REQ, RSP>
  - simple_arb<REQ, RSP> // not a fully generic arbiter, example only
- Leverage existing TLM generic components as much as possible.
- Create your own generic TLM modules, channels, adapters, transactors, etc., when needed.
  - e.g. generic crossbar, pipeline, parallel to serial adaptor, cache, etc.

Deterministic Modeling

- Use a good strategy for deterministic modeling (a.k.a. “avoid races”)
  - Enables reproducibility of simulation results across simulators
  - Aids refinement: a properly refined design will give same result
- Strategies: (See “System Design With SystemC” page 120)
  - Use two-phase primitive channels such as `tlm_fifo`, `tlm_poke_channel`, `tlm_req_rsp_channel`, `sc_fifo`, `sc_signal` for all communication between SystemC processes
  - TLM models and systems that model arbitration commonly use an explicit two-phase synchronization scheme (SDWS ch. 8)
  - Use a deterministic model of computation such as KPN or SDF (SDWS ch. 5)
  - Combine above approaches for systems with mixed abstraction levels

TLM Interface Style

- The TLM interface style is the same as sc_fifo, SystemC as a whole, and other C++ libraries, e.g. stl
  - Inbound data is always passed by const &
    - eg bool nb_put(const &)
  - Outbound data returned by value if we can guarantee that there will be data to return
    - eg T get(tlm_tag<T> *)
  - If we cannot guarantee that data will come back, we return the status and pass in a non const & :
    - eg bool nb_get(T &)
  - We never use pointers
  - We never use non const & for inbound data
- This is not pure pass-by-value, but it shares the need to provide copy constructors and destructors with pass-by-value
  - Think of it as being *effectively* pass-by-value

“Effective” Pass-by-Value

- Benefits
  - Eliminates problems/bugs associated with pointers and explicit dynamic memory allocation and deallocation
  - Helps eliminate problems/bugs in which multiple SystemC processes write to the same shared variable (i.e. “races”)
  - Lifetime and ownership of objects is very simple and clear
  - Helps you reason about concurrent systems
  - Enables use of C++ smart containers and handles
- Potential Drawbacks
  - Naive passing of large objects may lead to performance problems
    - Most common problem is passing large vectors or arrays of data by value

The TLM Copy-on-Write Vector

- To pass large vectors or arrays very efficiently, use the TLM copy-on-write vector – tlm_cow_vec<T>
- tlm_cow_vec<T> is nearly identical to std::vector<T>
  - The two can be easily swapped for each other.
  - Both use strict “by-value” semantics for assignment, construction, reading/writing elements, copying slices, etc.
  - Unlike std::vector<T>, tlm_cow_vec<T> does *not* support any operations that involve resizing the vector.
  - Unlike std::vector<T>, tlm_cow_vec<T> is very “smart” and does not actually copy any of the underlying data or allocate new data unless required. But this is all hidden to the user.
  - When tlm_cow_vec<T> is passed by value, copied, assigned, sliced, etc., only a very small amount of data in a handle is actually exchanged.
- tlm_cow_vec<T> should be added soon to OSCI TLM kit
- Contact stuart@cadence.com for more information about tlm_cow_vec<T>.

Conclusions

- Benefits of TLM:
  - Fast and compact
  - Integrate HW and SW models
  - Early platform for SW development, easy to distribute
  - Early system exploration and verification
  - Verification reuse
- TLM is the next level of design and verification abstraction in EDA, and the shift is now starting.
- The OSCI TLM standard is available now and is already in use, and should foster the development of a TLM IP ecosystem.

