February 28, 2011

TLM methodology to enable Architecture Exploration via co-simulation of SystemC models with legacy C/C++ models

Knute Lingaard
Lead Modeling Architect

Navaneet Kumar
Modeling Architect

freescale™ semiconductor

Outline

- Architecture Exploration & Virtual Platform
- Problem with Virtual Platform Integration
- Is TLM 2.0 an answer?
- TLM Methodology
- TLM Protocol Definition
- TLM Adapter & Wrapper
- TLM – Challenges and Learning's
- Simulation Control Challenges
- Conclusion
- Future work
- References
- QA

freescale™ semiconductor

Architecture Exploration & Virtual Platform

- Virtual Platforms are increasingly becoming important for Architecture Exploration
  - Created by integrating functional and cycle-accurate models of C/C++/SystemC
- SystemC is recognized as a standard to create cycle-accurate models at multiple levels of abstraction
- Seasoned organizations contain large collection of legacy high-fidelity models not built using SystemC framework
  - Porting these models to SystemC is not an option nor it is necessary
- While new models can be developed using SystemC, reusing legacy models and frameworks allow quicker platform integration and shortens the overall exploration cycle
  - This only works if the models are developed within the same organization

freescale semiconductor ™

Problem with Platform Integration

- Platform Integration can be difficult between models built using different frameworks
- Mismatch in Interface/ Modeling framework
  - Legacy C++ models have their own C++ class library extensions different from SystemC
  - These models cannot directly connect nor interoperate with SystemC models
- Duality of Simulation Control
  - Legacy C++ models have their own simulation control semantics
  - SystemC scheduler is dominant; forcing the simulation control mechanism to be either shared or completely controlled by SystemC
- Goal is to arrive at a common methodology and infrastructure that allows easy and quick integration/ co-simulation of legacy C++ models with SystemC models

freescale semiconductor ™

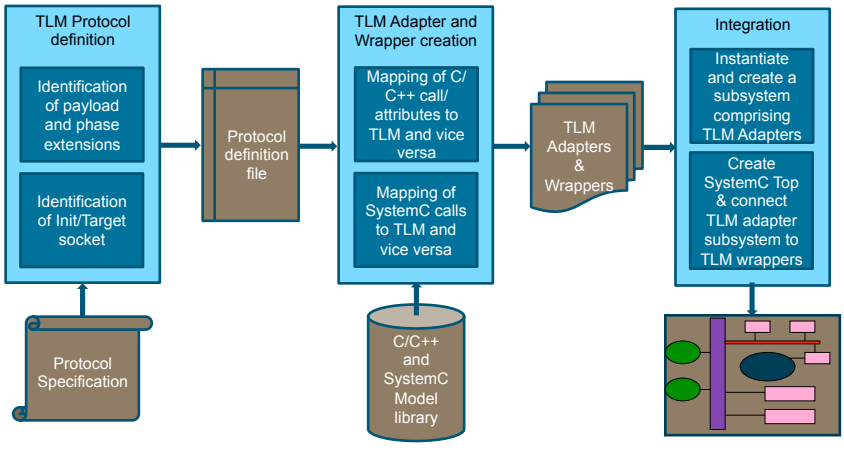
Is TLM 2.0 an answer?

- TLM 2.0 is an interoperability standard
  - Defined set of core interfaces - Non-Blocking/ Blocking/ Debug/ and DMI
  - Extension mechanism works seamlessly with generic payload and phases
- Built on top of SystemC
  - Easy to wrap SystemC models into TLM
- TLM enables connections of C++ legacy models to SystemC models
  - Generic payload extensions that can carry any number of extra payload information
  - TLM phase mechanism allow for bus protocols definitions and state machine
  - TLM is a single API that can be shared between models (as compared to multiple separate function calls each with a different payload signature)
- Internally, Freescale has working proof of this connectivity!

5

freescale semiconductor

TLM Methodology



6

freescale semiconductor

TLM Protocol Definition

- Identification of phase extensions
  - What new phases are required to model specific bus protocol?
  - Guideline for direction (forward path or backward path) and timing point.
  - Extended using *DECLARE_EXTENDED_PHASE* macro.
- Identification of payload extension
  - Which fields of generic payload are applicable as it is?
  - What new payload extensions are required to carry extra payload?
  - Extended by inheriting from *tlm::tlm_extension<>* class.
- Identification of Initiator/ Target socket
  - Do we need to connect multiple targets or just one target?
  - Do we need to connect multiple initiator or just one initiator?
  - Can choose from *tlm::multi_passthrough_xyz_socket* or *tlm::tlm_xyz_socket* – where xyz is initiator or target as applicable.

freescale semiconductor

TLM Protocol Definition – contd.

```
/**
 * Phase: RDR
 *
 * Direction: backward path - Inbound/Outbound Slave
 *            to Inbound/Outbound Master
 *
 * Description: Sent by Slave to communicate that
 *              read data is ready for delivery.
 *
 * Timing: May be issued no sooner than the clock
 *         following the receipt of that
 *         transaction's REQ event.
 *
 * Relevant payload:
 *    Generic payload:
 *       m_length, m_data
 *    Extensions:
 *       TagExtension
 *
 * Return value: TLM_ACCEPTED
 */
DECLARE_EXTENDED_PHASE(RDR);
```

```
/**
 * \brief Payload extension for transferring tag
 */
class TagExtension:
   public tlm::tlm_extension<TagExtension>
{
public:
   /** Constructor
    *  \param tag Tag of the transaction
    */
   TagExtension(const Tag & tag)
      : tag_(tag)
   { }

   /** Clone the extension
    *  \return A copy of this instance
    */
   virtual tlm::tlm_extension_base* clone() const
   {
      TagExtension * t = new TagExtension(*this);
      return t;
   }

   /** Copy from the given extension
    *  \param ext Reference to another tlm extension
    */
   virtual void copy_from(tlm::tlm_extension_base const &ext)
   {
      *this = static_cast<TagExtension const &>(ext);
   }

   /** Obtain tag
    *  \return Tag value
    */
   const Tag & getTag() const
   {
      return tag_;
   }

private:
   Tag tag_; //< Tag value

}; // End of class TagExtension
```


freescale semiconductor

TLM Adapter

- Key Ideas
  - Non-Blocking transport interface provides true cycle -approximate model communication
    - Possibility to specify more phases in the transaction lifetime
    - Choice of forward path and backward path
    - Return type *tlm_sync_enum* adds more meaning to payload and phases on return path
  - No use of SC_METHOD or SC_THREAD
  - Zero delay model
    - Delay set to SC_ZERO_TIME and typically ignored
  - Doesn't buffer any transaction; just focused on immediate protocol conversion from C/C++ to TLM and vice versa!
- Created TLM adapters for Freescale Internal Interconnects

9

freescale semiconductor ™

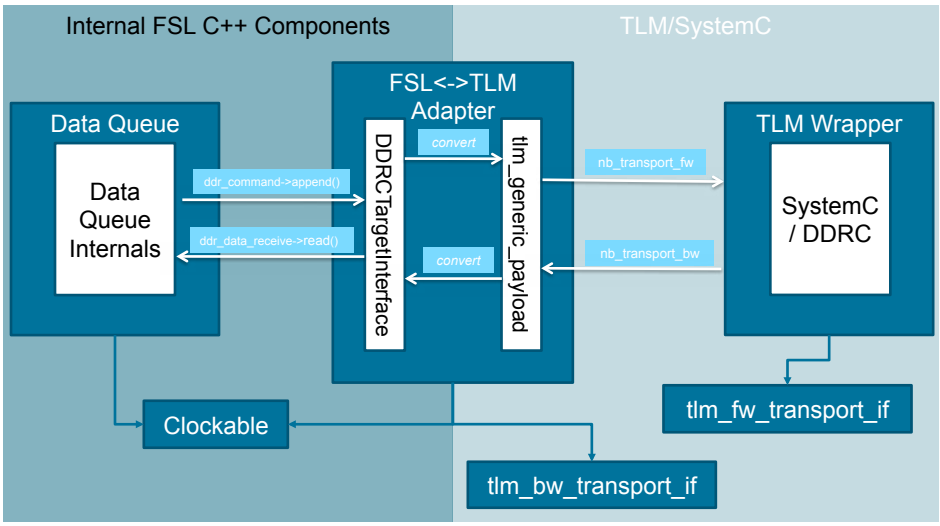
TLM Wrapper

- Key Ideas
  - Uses Non-Blocking transport interface for communication (like TLM Adapter)
  - Zero delay model
    - Reads SystemC signals as soon as they become valid and calls TLM API (for outgoing transaction)
    - Gets a TLM calls and converts into SystemC signal write (for incoming transaction)
  - Doesn't buffer any transaction; just focused on immediate protocol conversion from SystemC to TLM and vice versa!
- Created TLM wrappers for Core, IO device and DDRC

10

freescale semiconductor ™

TLM Adapter – A DDRC Example



11

freescale semiconductor

TLM – Challenges and Learning's

- Payload management (aka Memory management)
  - What is the lifetime of the payload? Should it be shared across multiple phases?
  - TLM 2.0 forces user to track memory management of payloads and extensions with the use of acquire and release
  - Payload reuse is good, but dangerous
    - It requires work and trust on the developer to ensure proper memory management and often leads to memory tracking issues for complex protocols
    - Often time, it is not possible to share a payload with two or more entities at same time (e.g. the extension update made by one might override the update made by second entity – as only one extension per type is kept in payload object)
  - Not taking advantage of reuse is inefficient
    - Each phase can be considered a separate transaction; initiator of the transaction allocates a new payload and then releases it right after the transport call is returned
      - Leads to frequent allocations and de-allocations
    - Memory management complexities are simplified; less work on user's part
  - FSL models follow reuse mechanisms wherever feasible
    - Between projects, the teams agreed on the reuse policies of TLM adapters/wrapper's payloads

12

freescale semiconductor

TLM – Challenges and Learning's – contd.

- Handling of the BUSWIDTH template parameter
  - Greatest design hardship experienced with TLM that, as a design choice, provides little to no gain
  - Forces developers to expose more code internals than desired
    - Current C++ standard requires template code to be defined in header files
    - To prevent exposure of our internal IP, TLM Adapter was made independent of templated socket
    - The TLM adapter just knows forward and backward non-blocking interfaces
    - TLM sockets implementation are moved internally, providing specializations that *hardcoded* the BUSWIDTH parameter to the proper sizes
    - Internally to the model, we ignore the BUSWIDTH parameter, which defeats the original intent
- Out of order transaction tracking
  - A complex protocol required us to track out-of-order transaction phases (between forward and backward path) and correlate response payloads with that of request payload
  - TLM generic payload extension mechanism provided the solution
    - Created a specific *UIDExtension* (a payload extension that carries 32bit unique id)
    - Target, being unaware of the *UIDExtension*, simply returns it with the original payload on bw path
    - Initiator can check the response payload and match with corresponding request payload

13

freescale semiconductor

Simulation Control Challenges

- Clock Control and Determinism
  - Internal C++ modeling framework has its own clocking mechanism
    - Clock has two phases – Tick and Update (something like Evaluate and Update in SystemC)
  - Between disparate clocks, determinism is ensured via a clocking manager
    - All clocks in the system register to the clock manager
    - The clock manager handles the appropriate ticking and updating order between clocks
  - When integrated with SystemC scheduler, the clocking manager couldn't be used
    - C++ clocks working under SystemC clocks became out of sync
    - The clock thread ordering became important , without it, there's indeterminism!
  - To counteract, we introduced a clocking proxy
    - Each proxy encapsulates one FSL clock and is controlled via one SystemC clock thread
  - SystemC clock thread separates C++ Clock's Tick and Update by a delta cycle
    - Irrespective of ordering between two such clock threads, update made by one clock object is only visible after (at least) a delta cycle to another clock object – thus ensuring determinism.

```
SC_THREAD(coreClockTick_);
sensitive << core_clock_port_.pos();
dont_initialize();
```

```
void SystemCSimConfigurator::coreClockTick_()
{
   while (true)
   {
      // Run update phase and then tick phase
      core_clock_proxy->RunUpdatePhase();
      wait(SC_ZERO_TIME);
      core_clock_proxy->RunTickPhase();

      wait(); // wait for next posedge here
   }
}
```


14

freescale semiconductor

Simulation Control Challenges – contd.

- End of Simulation Control
  - With multiple initiators, each running its own binary, it becomes difficult to decide when to stop simulation
  - sc_start(<some-time>) is not a good choice – you don't know how much time budget is sufficient
  - We created a custom handler called IntegrationEnvironmentHandler
  - Each initiator has access to the handler and calls simulationCompleted() when done.

```
// This class is a handler for sub-models integration
class IntegrationEnvironmentHandler
{
public:
   IntegrationEnvironmentHandler(unsigned int numOfSubModels)
      : NUM_OF_SUB_MODELS(numOfSubModels),
        numStopped_(0)
   { }

   //! Called by sub models to stop simulation
   void simulationCompleted()
   {
      ++numStopped_;

      // Call sc_stop when sufficient number of modules are stopped
      if (numStopped_ == NUM_OF_SUB_MODELS)
      {
         sc_stop();
      }
   }

private:
   const unsigned int NUM_OF_SUB_MODELS;

   unsigned int numStopped_; // num of sub-models that called to stop
};
```


freescale™ semiconductor

Conclusion

- TLM 2.0 based methodology is now a proven technique to integrate and co-simulate legacy C/C++ models with SystemC/ TLM models
  - TLM Adapters and Wrappers can be designed in a smart way that doesn't degrade cyclic performance.
  - Possibility to add payload-extensions and phase-extensions to suit specific protocol as desired.
  - Payload Management is crucial; Reuse is better.
- Non-Blocking TLM API is the perfect choice for modeling cycle-approximate communication
  - Choice of forward path and backward path
  - Meaningful return path by use of *tlm_sync_enum* return type
- TLM built on top of SystemC provides enough hooks to ensure clock control and determinism.

freescale™ semiconductor

Future work

- We're exploring TLM 2.0 as a standard to integrate our legacy C++ models with any 3rd party model (SystemC or non-SystemC)
  - Enables our modeling technology to be usable in tools that understands TLM frameworks
- With non-SystemC 3rd party model, we can possibly avoid SystemC scheduler invocation at run time. The simulation ownership can remain with the internal modeling framework; whereas TLM provides us the external communication platform.
  - We did a small pilot to co-simulate our C++ model directly with another C++/TLM model via TLM mechanism. Results have been encouraging..! We could use SystemC just for linking without invoking the scheduler at run time.

freescale semiconductor ™

References

- OSCI TLM 2.0 Language Reference Manual, July 2009. www.systemc.org
- Van Moll, H.W.M.; Corporaal, H.; Reyes, V.; Boonen, M. "*Fast and Accurate Protocol Specific Bus Modeling using TLM 2.0*". DATE '09.

freescale semiconductor ™

Q & A

and thank you…

freescale™ semiconductor

freescale™ semiconductor