Early Hardware/Software Integration Using SystemC 2.0

Jon Connell, ARM.
Bruce Johnson, Synopsys, Inc.

Class 552, ESC San Francisco 2002

Abstract

Capabilities added to SystemC 2.0 provide the needed expressiveness and abstraction to model processor-based systems. By representing the system at a transaction-based level, the hardware and software teams share a common abstraction and verification environment. Models of microprocessors are a natural addition to a system modeled at the transaction level. Hardware-software interactions are first defined as processor independent transactions. This view is then refined by adding a processor model which enables the verification of the hardware-software interactions from a very abstract, purely transactional basis, all the way down to the level of verifying the interaction of the software, processor, RTOS, and hardware subsystems.

Background

SystemC is the standard design and verification language that spans from concept to implementation in hardware and software. The Open SystemC Initiative (OSCI) is a consortium of major EDA and IP companies that contributes to and governs SystemC development and distribution. SystemC users may develop models using SystemC along with standard ANSI C++ compilers. SystemC was first introduced in September 1999. For more information about SystemC, refer to the SystemC web site: www.systemc.org.

Prior to the introduction of SystemC in 1999 there were many proprietary C or C++ based environments. Since these environments are not based on an open standard, their usefulness is limited since model availability from IP vendors is non-existent. SystemC has become the de facto standard for system level design. As a result, IP vendors are starting to provide SystemC compatible models of their IP. As model availability increases, so too will adoption increase. After reviewing the material presented in this course it will be apparent that adopting SystemC and system level design needs to be on your organization's technology roadmap.

The Need for System-Level Modeling

IP companies have heralded a new age in platform-based design for a number of years ever since semiconductor integration capacity reached the point where entire systems could theoretically be integrated into a single die. So why haven't we seen a huge explosion in platform-based design? The key is the scope of the platform: only now are platforms being defined which include a wide assortment of elements from System-Level Design (SLD): the RTL hardware definition, bus architecture, power management strategy, device drivers, OS ports, and application software. To be successful, however, a platform will need more than this; an essential element for enabling differentiation will prove to be an advanced systems modeling and verification environment. Developers require a variety of views of the entire platform from RTL, system models, software development models, and real hardware development boards.

Each view of the platform reflects the same system architecture, and designers can use test software in any of the higher-level views, providing a high degree of confidence in the design prior to tape out. This provides a valuable environment in which to investigate system bandwidth and performance requirements. System views must be extendible, allowing designers to exploit the advantages of a well-supported, pre-verified base platform of hardware and software IP, whilst differentiating their own application with their own IP.

	Platform Methodology	Extension possibilities
Unit Testing	Platform constructed from well-tested Star IP	Standard interfaces such as AMBA and uITRON ensure extensions work with platform
Integration Testing	Bus Functional Models are used to demonstrate that the platform RTL is wired up correctly	Additional IP can be added to the platform RTL

System Validation	Coverification with processor models and RTL runs real software for firmware validation	Additional hardware IP can be added to the RTL and new drivers validated
System Modeling & Software Development	SystemC models of the platform execute at high-speed with transaction accuracy to post and test application software	Additional models can be added and ported to the OS for OS porting and software development

At the system-level, availability of software becomes critical and it is no longer reasonable for the software team to wait for a prototype system. Coverification can move the integration schedule forward to the point where RTL is available, but this still delays the software integration to a point where much of the hardware design is complete. System and software designers would still be lacking a common environment. Consider instead a design flow where the system is first specified using SystemC, then partitioned into hardware and software blocks and handed to the respective teams. This executable specification is a key enabler for both teams. The software engineer not only has a platform for the development of his software, he has a C++ based simulation environment he can easily utilize.

One of the key techniques used in this design flow is the modeling of the system at the transaction level. Transaction level modeling (TLM) is simply a higher abstraction level for modeling. Systems modeled in RTL are concerned about the hardware details such as pin-level behavior of their system. With TLM it's possible to accurately model many aspects of a system at a higher (e.g. Read and Write) level. During this class we will introduce a TLM of a simple bus. By using TLM we simplifying the modeling effort and we also gain simulation speed.

Following the brief introduction to transaction level modeling we will provide an example system constructed using transaction level models of a bus, peripheral devices, and software. We will then show how this platform model may be utilized as a software development platform so that the the original processor independent software may be refined, ultimately to that which is targeted for a specific processor. We will then demonstrate how a cycle accurate ARM processor model may be added to the system.

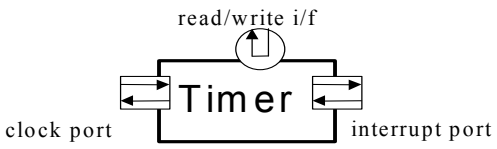
Introduction to SystemC Modeling: The Simple Bus Model

Synopsys has created an example transaction level bus model and contributed this to OSCI; it is open source. We are using this model to demonstrate the operation of transaction level bus models.

The model is referred to as simple because it is meant to be instructive on the use of transaction level modeling rather than on the development of models for sophisticated bus architectures.

To begin, let us introduce some basic SystemC concepts and nomenclature. A system may be modeled as a collection of **modules** that contain **processes, ports, channels,** and even other modules. Processes define the behavior of a particular module and provide a method for expressing concurrency. A **channel** implements one or more **interfaces**, where an interface is simply a collection of method (a.k.a. function) definitions. A process accesses a channel's interface via a port on the module.

This scheme is most easily understood via the following example showing how a countdown timer might be implemented:



The timer has a read/write interface that allows the configuration register(s) to be read or written. When the counter expires, an interrupt is asserted on the interrupt port. And, of course, the timer is connected to a clock (another port).

The process for the timer would be sensitized to a clock edge. When the process is executed it would decrement the counter. If the counter were zero it would assert an interrupt. The timer process might look like:

```
void timer::tick()
{
   ...
   if (--this->count == 0)
      this->interrupt_port = 1;
   ...
}
```

The implementation of the read interfaces might look like:

```
void timer::read(int register, unsigned char *value)
{
   *value = this->regs[register];
}
```

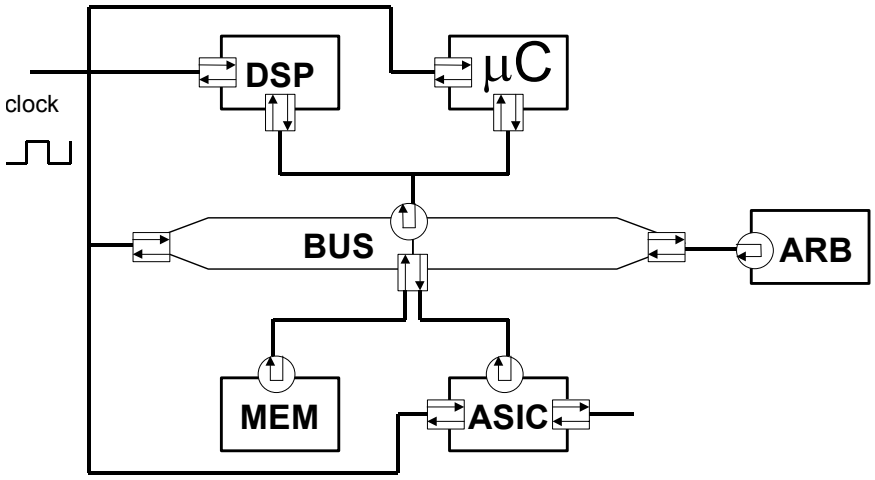
Note that the timer itself implements the read/write interface. Modules which implement a channel's interface are known as hierarchical channels.

The connection of another module to the timer's read/write interface is accomplished by defining a port and connecting the other module's port to the timer's port. Keeping things generic, we'll call this other module top. The following fragment shows how top can instantiate, connect, and interface to the timer:

```
void top::top()
{
  ...
  timer_p = new timer(...);
  this->timer_conn(timer_p);  // bind the port to the channel
  ...
}
void top::reset_timer(...)
{
   this->timer_con->write(CONTROL_REG, RESET_VALUE);
}

void top::tick()
{
  if (time == 0)
    reset_timer(...)
  ...
}
```

Now let's have a look at the simple bus model which is shown in the following diagram:



This system includes two bus masters: **DSP** and **uC**, a bus arbiter, a memory, an ASIC, and a clock. The masters, bus, and ASIC are all sensitized to the clock. The masters each have a port connecting them to the bus, which is a hierarchical channel. The bus has a port which connects it to slave devices (MEM and ASIC in the diagram). The slave devices are also modeled as hierarchical channels. Notice that ports may be connected to multiple channels, as is the case between the bus, memory, and the ASIC. Now we'll step through the operation of this system.

During system initialization each of the entities are instantiated, and the masters and slaves are registered with the bus. For example, the slave addresses are registered with the bus so that the bus may perform the address decoding for memory accesses.

During the rising clock edge any masters needing the bus, request it. The bus then has a process that is sensitive to the falling edge of the clock. If there were multiple requests, the arbiter is called. Once master-ship is determined, the busses memory map is consulted and slave accesses are made.

The following sections will provide more details of the interfaces just described.

Master Interfaces

Masters have three interfaces: blocking, non-blocking, and direct.

The blocking interface includes reads, writes, and burst methods. Each of these methods does not return until the request has completed. This flavor of interface is useful for abstract software behavior. The following methods are included in the blocking interface:

```
virtual simple_bus_status burst_read(unsigned int unique_priority,
                                     int *data,
                                     unsigned int start_address,
                                     unsigned int length = 1,
                                     bool lock = false);

virtual simple_bus_status burst_write(
                                    unsigned int unique_priority,
                                    int *data,
                                    unsigned int start_address,
                                    unsigned int length = 1,
```

```
                                        bool lock = false);
```

The non-blocking interface also includes read, write, and burst methods, but this interface is a cycle-callable interface. Masters using this interface interact with the bus model on a cycle-by-cycle basis; that's why we refer to it as cycle-callable. This interface is useful for the integration of cycle-accurate processor models.

```
virtual void read(unsigned int unique_priority,
                  int *data,
                  unsigned int address,
                  bool lock = false);
virtual void write(unsigned int unique_priority,
                   int *data,
                   unsigned int address,
                   bool lock = false);

virtual simple_bus_status get_status(unsigned int
unique_priority);
```

The direct interface is provided to allow "back door" memory accesses to slave devices without requiring simulation to progress. This interface is particularly useful for implementing a debugger interface for a processor model. For example, when the user requests to see the value of a variable, the debugger turns that into the appropriate memory read requests which could then be satisfied by these functions.

```
virtual bool direct_read(int *data, unsigned int address);
virtual bool direct_write(int *data, unsigned int address);
```

Slave Interfaces

The slave interface includes functions called by the bus model to identify the address range occupied by the slave, to perform the read and write operations, and to satisfy direct memory access requests.

```
unsigned start_address();
unsigned end_address();

status read(data*, address);
status write(data*, address);

status direct_read(data*, address)
status direct_write(data*, address);
```

As you can see using the simple bus infrastructure it would be very easy to, for example, write a cycle accurate model of a slave device, connect it to the bus and verify the implementation. To verify a new slave a master device acting as a stimulus generator would be added to the system.

While we've by no means given a thorough description of SystemC or transaction level modeling it should be evident that SystemC does provide an incredibly useful infrastructure for the modeling and verification of systems at higher level of abstraction than has been customarily used in designs today. For additional information please refer to the SystemC web site: www.systemc.org.

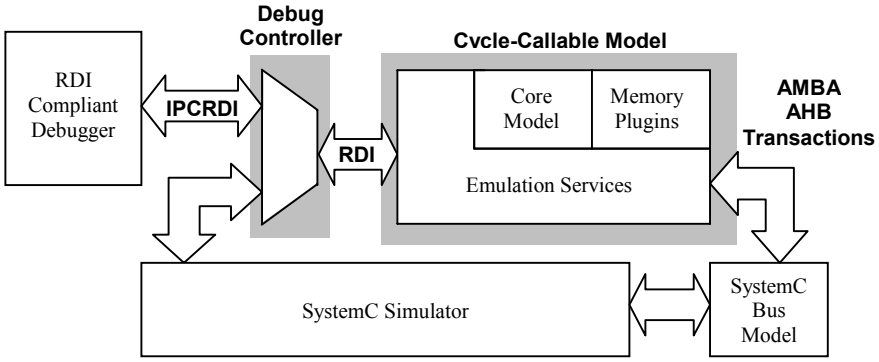
Cycle-Accurate Processor Models

We briefly mentioned that a processor model could be added to a transaction level model of a system. What's a processor model? A processor model is a program that simulates the behavior of the target processor within the context of the overall SystemC simulation. In a typical embedded developer's toolkit,

an Integrated Debug Environment (IDE) contains a connection to a single ISS representing a single CPU with a simple memory system. Execution within the ISS occurs at instruction boundaries and interaction with the memory system is via an address map only.

Complex systems can easily have multiple processing units, made up of CPU, DSP and application-specific cores. For each of these to interact, they must present a SystemC module that has a clock port, a connection to an appropriate bus, and probably some asynchronous input ports like reset and interrupts. Within the Simple Bus Model framework we've presented, you can see that a processor model would be modeled as a bus master module. The processor model would be cycle-synchronous with the rest of the SystemC simulation: for each cycle of the SystemC simulation there would be a corresponding cycle within the processor model. This integration scheme provides the greatest accuracy and, since the system is modeled at the transaction level, the overall simulation speed is extremely fast.

This level of processor model integration provides a tight integration between the model and the rest of the system:



The CCM Interfaces

The Cycle-Based Model (CCM) exposes two important interfaces to the designer which are used to interface the CCM to a cycle-based simulation environment: a bus-transaction interface, and a Remote Debug Interface (RDI).

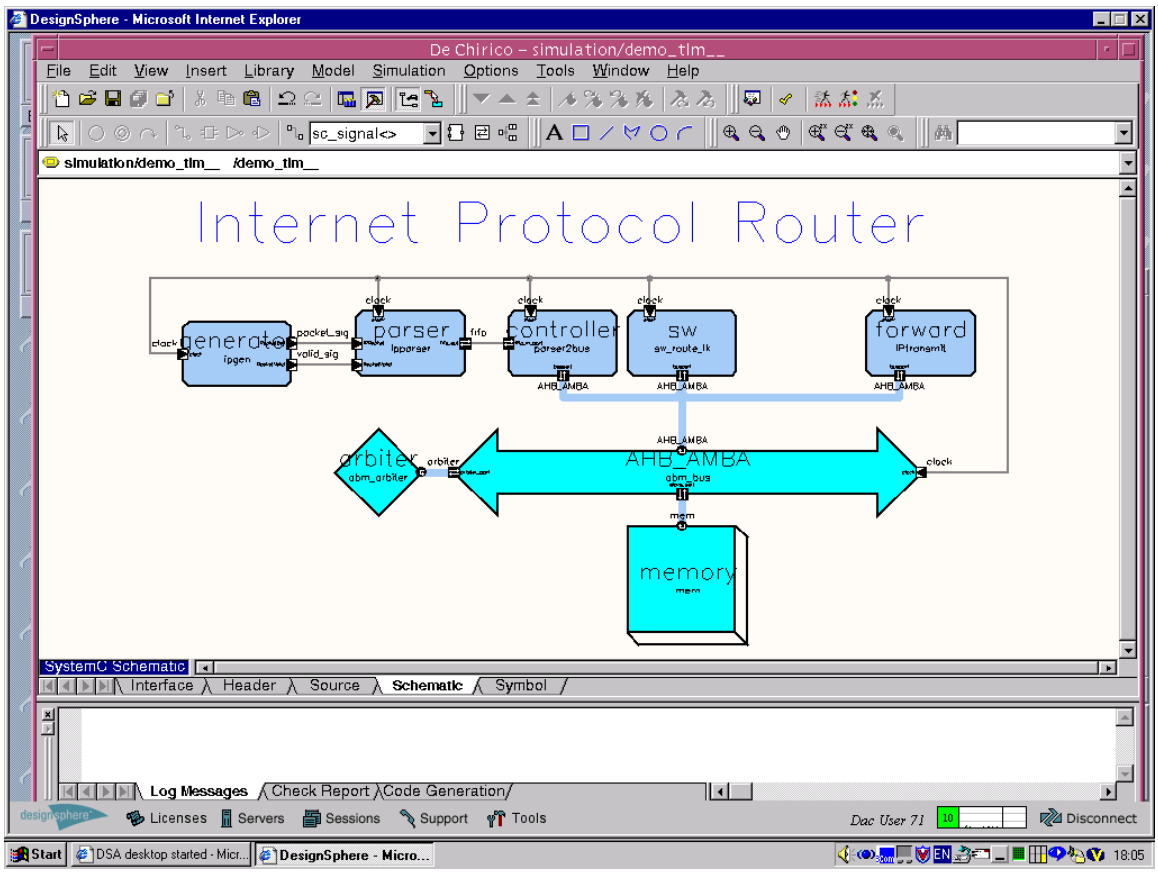
A simple callable interface provides pipeline-accurate AHB transactions which can easily be translated into bus transactions at the cycle-level. The advantage of using such a generic interface is that the same processor model can be used in a variety of abstractions. The speed of the interface is directly proportional to the level of detail extracted from the transactions provided by the processor model. Should the designer wish to re-use the model in a signal-level interface, this is still possible using the same model.

The RDI is a standard debug interface for ARM cores which is extended for the CCM to include an interprocess layer to allow the debugger to reside in a separate process. In a typical IDE, this is unnecessary since the IDE controls the ISS execution and its behavior is predictable. In a complete system, the processor is but one part and execution is controlled by the SystemC simulator as shown above. Now a debug controller must be inserted between the debugger and the CCM to ensure that requests to clock the model are correctly multiplexed with requests from the debugger to continue program execution. Also, the user will expect to be able to interact with the software debugger even when the system is stopped such as when a hardware breakpoint inside another model is reached.

Example Design Flow: The IP Router design

Now that we've shown the modeling building blocks available, we're going to demonstrate the application of TLM and processor models in an example design. The design we've chosen is an Internet Protocol router. This design is constructing using a cycle-accurate bus model for the AMBA AHB bus. We will focus on the software block of the design, reviewing the entire design is well beyond the scope of the material we could cover within this class.

The design is depicted in the following diagram:



Operational Overview

An IP Packet traffic generator, **generator** in the diagram, provides the stimulus. This generates IP packets at the byte level along with a control signal.

The **parser** parses each packet header, extracting information like IP version and destination address, then sends the packet and descriptor to the DMA controller **(controller)**, which takes care of storing the packet in memory.

The **sw** block is responsible for calculating the destination address of the packet and updating the in-memory packet information.

The remaining block is the IP transmitter, **forward**, in the diagram. Depending on the destination address, it will forward the packet.

The DMA controller, IP router, and IP transmitter blocks are connected to the AHB AMBA BUS that is defined by ARM. They are all masters and they all access the memory that is modeled as a slave.

Software Specification

As you would expect, the initial software representation of the system was modeled purely in SystemC, without taking into consideration any processor or runtime environment. This version of the code would be used to verify the hardware/software partitioning and overall system operation. This code could then be handed to the software team along with the rest of the executable system (aka platform) for actual processor specific implementation and verification.

The software algorithm is very simple: loop continuously checking for new IP packets, each IP packet is then examined and its destination determined using a simple table lookup. Recall that the IP packet is placed in memory by the DMA controller. At this stage of refinement, the software is modeled as a single process within a SystemC module.

The software module (sw in the diagram) is derived from the standard SystemC module class, sc_module:

```
class sw_route_lk
: public sc_module

public:
   // master port interface to the bus.
   abm_master_port busport;

  // process
  void sw_code();
  ...
```

Among other things the module contains a port, **abm_master_port**, which provides the connection between the software and the bus, and a function , **sw_code**, which represents the process for the module

The following code fragment is one representation of a simple memory search for packets needing to have their destination address calculated:

```
// look for a bank with data
do
            {
                ...
                address = get_destination_address(…);
                busport.burst_read(&local_flags,
                        address,
                        1,
                        ABM_SINGLE,
                        4,
                        false );
   } while ((local_flags == BANK_EMPTY)
                            ||(local_flags == BANK_LOOKUP_DONE));
```

You'll notice that the single 32 bit read is accomplished via the burst_read method call on the bus master port, which in turn requests the bus on behalf of this master and sees to it that the memory read transaction is performed. When the transaction is complete the memory contents will be contained in the local_flags' variable.

The software developer would refine this code and eventually retarget it for execution on the target processor. Prior to adding a processor model, the refinement process would execute the code natively. When appropriate a model of the target processor could be substituted for the **sw** module. The code contained in the **sw_code** method would then be cross compiled, linked, and executed using the target processor. In our example, the obvious modification to the sw_code module would be to change the memory access so that the code would appear as follows:

```
// look for a bank with data
do
        {
            ...
            address = get_destination_address(…);
            local_flags = *address;
  } while ((local_flags == BANK_EMPTY)
            ||(local_flags == BANK_LOOKUP_DONE));
```

While the translation is straightforward, it also wouldn't be hard to implement a more structured #ifdef scheme where abstract memory accesses were handled via a function call and target compiled code is handled via a pointer de-reference.

Another important observation is that while the above software may be functionally correct, it fails to take into consideration a few things:

- availability of processor to perform the operation (e.g. processor not busy doing something else)
- the amount of time it takes to perform the operation (e.g. instruction fetches and cycles to perform operations)
- bus availability
- RTOS scheduling

It would be possible to at least partially address these considerations by refining the sw_code method to include delay models to account for the performance and availability of the processor. The addition of modeling infrastructure to address the software flow is planned for SystemC 3.0.

Early System Integration

Recall from our introduction, that for an SLD process to be successful, it must provide system designers with a continuum from specification through to implementation. Having determined that their design is functionally correct and that their hardware/software partitioning meets their design requirements, the designer can use the approach we described earlier of replacing the software component with a processor model.The model of the system then becomes a development platform for the system software. This would happen long before any physical prototype or RTL simulation environments were to become available!

For the IP Router design we replaced the **sw** module with a cycle-accurate processor model for the ARM920. The operation of the processor model and SystemC was very intuitive as well as simulating fast. Debugging, from the software perspective, was accomplished using the ARM ADU debugger, while debugging from the SystemC perspective was accomplished using DDD. It would be possible for an engineer who was unfamiliar with the hardware implementation to view the system from the perspective of the code running on the target processor, using ADU in our case. Similarly, and engineer unfamiliar with the code running on the target processor could debug the system using DDD. Use of one or the other, or both, debuggers is optional.

Moving on from here, it would be possible to employ techniques like coverification to demonstrate that the software also performs as expected on the RTL implementation.

Conclusion

We've demonstrated a development flow which enables software integration earlier in the design process than is possible using more customary design flows. This flow is enabled by using the open source SystemC infrastructure. The flow is very natural and does not require the software team to learn new tools; from their perspective their platform is written in C++ , may be run on the workstation, and can be debugged using the workstation-native debug tools. Additionally, the software team can look forward to refining their code using their target processor, its development environment, and SystemC, long before they could expect to see other development platforms (RTL or HW-based).