Transaction-level modeling of bus-based systems with SystemC 2.0

Ric Hilderink, Thorsten Grötker

Synopsys, Inc.
Efficient platform modeling

- Get to executable platform model ASAP
- Simulation speed >> 100k cycles/sec

Moving from pin-level to transaction-level models (TLM) is mandatory!
Outline

Idea:

- Based on an example show how SystemC 2.0 enables efficient platform modeling.
- Introduce some key language elements in the process.
Example: Simple bus model

- Cycle-accurate transaction-level model.
- “Simple” =
  - No pipelining
  - No split transactions
  - No master wait states
  - No request/acknowledge scheme
  - ...

**NB:** Of course, these features can be modeled at the transaction level
Interface Method Calls (IMC)

- Modules communicate via channels.
- Channels implement interfaces.
- An interface is a set of methods (functions).
- Ports
  - Modules have ports.
  - Ports are connected to channels.
  - Modules access channels through ports.

... some_port->some_method(some_data);
...

...
module::process() {
    ...
    port->some_method(42);
    ...
}
Hierarchical channels

- Channels can be hierarchical, i.e. they can contain modules, processes, and channels.
- A module that implements an interface is a hierarchical channel.
Example system (napkin view)
SystemC 2.0 transaction-level model
SystemC 2.0 transaction-level model
SystemC 2.0 transaction-level model

The bus is implemented as a hierarchical channel!
Arbiter and slaves are implemented as channels too!
SystemC 2.0 transaction-level model

Arbiter has been made a separate module to allow for customization!
SystemC 2.0 transaction-level model

Optionally, ports can be connected to multiple channels!
Optionally, ports can be connected to multiple channels!

```cpp
sc_port<class IF, unsigned n_channels = 1>
```
Rising clock edge

Masters request bus access.
The bus has a process that is sensitive to the falling edge.
Falling clock edge

The arbiter is called. It will grant a single master access to the bus.
Falling clock edge

Then, a slave is accessed after consulting the memory map.
Master interfaces of the bus

- **Blocking:**
  - Complete bursts
  - Used by high-level models

- **Non-blocking:**
  - Cycle-based
  - Used by processor models

- **Direct:**
  - Immediate slave access
  - Put SW debugger to work
Blocking master interface

- status burst_read(unique_priority, data*, start_address, length=1, lock=false);
- status burst_write(unique_priority, data*, start_address, length=1, lock=false);

“Blocking” because call returns only after complete transmission is finished.

Master is identified by its unique priority.
Dynamic Sensitivity

- **SystemC 1.0**
  - Static sensitivity
    - Processes are made sensitive to a fixed set of signals during elaboration

- **SystemC 2.0**
  - Static sensitivity
  - Dynamic sensitivity
    - The sensitivity (activation condition) of a process can be altered during simulation (after elaboration)
    - Main features: events and extended wait() method
Waiting

wait(); // as in SystemC 1.0
wait(event); // wait for event
wait(e1 | e2 | e3); // wait for first event
wait(e1 & e2 & e3); // wait for all events
wait(200, SC_NS); // wait for 200ns

// wait with timeout
wait(200, SC_NS, e1 | e2);
wait(200, SC_NS, e1 & e2);
Dynamic sensitivity

status bus::burst_write(...) {
  ...
  wait(transmission_done);
  ...
}

Master won’t be activated until transmission is completed!

Statically sensitive to clock
⇒ activated every cycle

Master won’t be activated until transmission is completed!
Dynamic sensitivity

Statically sensitive to clock
⇒ activated every cycle

Advantages:
- Easy-to-use interface (blocking interface)
- Simulation speed
Non-blocking master interface

- \texttt{status\ get\_status(unique\_priority);}  
- \texttt{status\ read(unique\_priority, data*, address, lock=false);}  
- \texttt{status\ write(unique\_priority, data*, address, lock=false);}  

“Non-blocking” because calls return immediately.  
Less convenient than blocking API but caller remains in control (needed e.g. for most processor models).
Direct master interface

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

Provides direct access to slaves (using the bus’ address map).
- Immediate access $\Rightarrow$ simulated time does not advance
- No arbitration

Use for SW debuggers or decoupling of HW and SW.
Use with care!
Slave interface
Slave interfaces

- unsigned start_address();
- unsigned end_address();
- status read(data*, address);
- status write(data*, address);
- status direct_read(data*, address);
- status direct_write(data*, address);

address
mapping

regular
I/O

debug
interface
What’s so cool about transaction-level bus models?

They are ...

- relatively easy to develop and extend
- easy to use
- fast
  - use of IMC $\Rightarrow$ function calls instead of HW signals and control FSMs
  - use of dynamic sensitivity $\Rightarrow$ reduce unnecessary process activations
Key language elements used in the example

- Interface method calls (IMC)
- Hierarchical channels
- Connecting ports to multiple channels
- Dynamic sensitivity / waiting
Conclusions

SystemC 2.0 enables efficient platform modeling.

- Ease of modeling
  ⇒ get to executable platform model ASAP
- Simulation speed

Still not convinced?

Try it out!  (see following slides)
How to install

> cd <systemc_installation_directory>/examples/systemc
> gtar zxvf simple_bus_v2.tgz

This will create a directory 'simple_bus'. Go to this directory and build the executable, e.g.

For gcc-2.95.2 on Solaris:
> gmake -f Makefile.gcc
...

Now you can run the executable, e.g.
> simple_bus.x

See README.txt for detailed information!
The testbench

see simple_bus_test.h
The testbench

Blocking master: uses blocking bus interface to read and write data

Arbiter: Priority-based arbitration, supports bus locking
The testbench

Non-blocking master: Uses non-blocking bus interface for data I/O

Direct master: uses direct interface of bus to print debug information
The testbench

Fast memory: zero wait states

Slow memory: configurable number of wait states
The testbench (cont’d)

- Most modules are configurable
  - Masters
    - Priority (not direct master)
    - Delay / timeout
    - Bus locking on/off (not direct master)
  - Slaves
    - Address ranges
    - Number of wait-states (only slow memory)
  - Bus, arbiter, direct master
    - Verbosity
- Change parameter settings in simple_bus_test.h
- See README.txt for details
That’s it!

Thank you and have fun trying it out!