

SystemC Cosimulation and Emulation of Multiprocessor SoC designs

By,

Luca Benini, Davide Bertozzi, Davide Bruni, Nicola Drago, Franco Fummi, Massimo Poncino

Presented by
Poovaiyah m a

Native ISS-SystemC Integration for the Co-Simulation of Multi-Processor SoC

F. Fummi
M. Poncino

S. Martini
G. Perbellini

Dipartimento di Informatica
Università di Verona

Centro di Competenza per la
Progettazione di Sistemi Dedicati

What is the topic all about?

- SystemC – open source c/c++ simulation environment.
- Co simulation – event driven h/w +ISS simulation
- Emulation – emulates the functions of the original h/w
- Multiprocessor – parallel processing of n processor cores.
- SoC – system on chip.

System on chip

- system on chip designs with the core interacting with the rest of the system.
- Development toolkit:
 - cross compiler
 - ISS (instruction set simulator)
- HDL for hardware.
- C/C++ for ISS.

System design environment

- SystemC is an open source C/C++ simulation environment.
- Modules specifying hardware blocks and communication channels.
- Advantage:
 - same software language to describe both hardware as well as software.
 - no need for cross compilation. simulate the entire system within a single engine.

Requirement to be met?

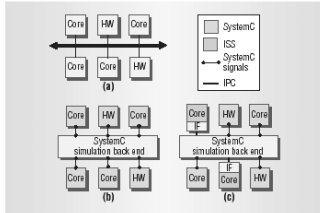
- Synchronization!!!!!!!!!!!!
- Modeling software execution consistently with the rest of the system is micro architecture level simulation
 - MAL approach: faster than RTL.
- Embed the ISS within the co simulation environment to simulate the core at a higher abstraction level.

To achieve this they use IPC (inter process communication)

Ways and its limitations

MAL- micro-architecture level simulation (faster than RTL)

Embed the ISS within the Cosimulation environment



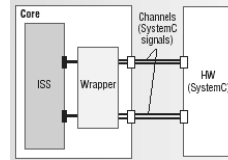
Bus wrapper/ISS interface is proprietary
 IPC paradigm is effective only when communication is infrequent.
 This is the "IPC PERFORMANCE BOTTLENECK"

Addressing this problem

ISS is embedded in the core

Implement all blocks as systemC modules.

Wrapper is within the core.



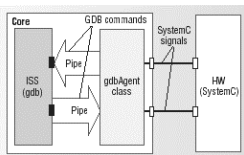
Cosimulation interfaces

● Remote ISS cosimulation

● Linked ISS cosimulation

Remote ISS :

This strategy implements the wrapper as an adhoc class gdbagent, whose main function is to execute the GDB and control its execution.



The GDB agent class constructor loads and executes the GDB, and creates two unix pipes to establish a bidirectional communication channel for exchanging GDB commands

GDB Agent

● The constructor of GDB Agent loads and executes the GDB

- Create 2 pipes for bidirectional communication.
- The execution is controlled by various methods:
 - Quit, Run, Next.
 - SetFile, SetBreakpoint.
 - ContBreak.
 - SendCommand
 - getVariable, SetVariable.

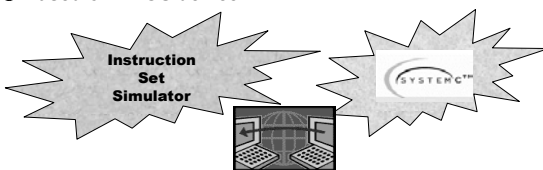
Simulators Interaction

● SystemC Kernel controlled, by using ad-hoc SystemC ports

● Based on GDB commands

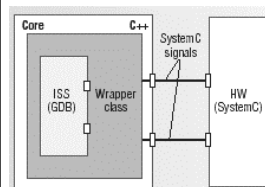
- SystemC-ISS Interface to data exchange

● Based on RTOS device

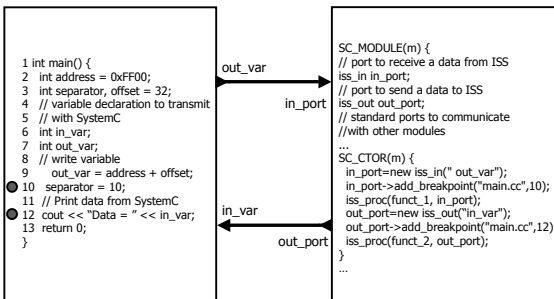


Linked ISS Cosimulation

When interaction between a processor instance and the rest of the system is tight, IPC becomes burdensome. Thus the remote ISS approach is to completely embed the ISS within the systemC simulator as a C++ class.



Example



ISS

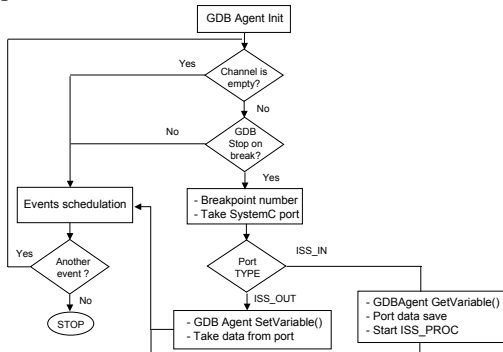


GDB-Kernel Programming Model

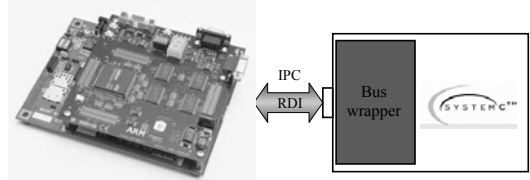
New objects do not change the behaviour of existing objects

- ports to read data sent by ISS, they are derived by template class *sc_in*
 - o *iss_in*
- ports to write data to ISS (derived by *sc_out*)
 - o *iss_out*
- methods that execute if new data are sent by ISS to SystemC ports.
 - o *iss_proc*
- ISS program sends data from SystemC by GDB Agent
 - o breakpoint
- ISS program receives data from SystemC by GDB Agent
 - o breakpoint

SystemC Kernel modifications



Emulation interface



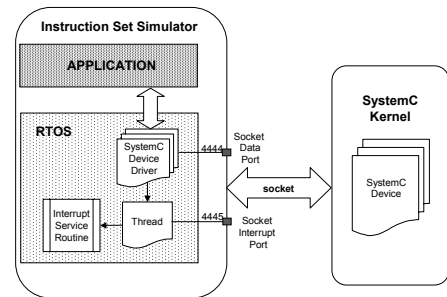
- IPC Interface between *bus wrapper* and ISS
 - o Remote Debugging Interface (RDI) of the GNU GDB
- The GDB use allows to apply co-simulation scheme when ISS is replaced by the real processor
- Virtual in circuit emulation → conventional in circuit → h/w replaces core

Driver-Kernel Programming Model

A device driver is designed as follows:

- The driver API have to communicate with the SystemC kernel via an ad-hoc Socket port (called Socket Data Port)
- The driver has to create a Thread to listen the interrupts reported from the SystemC device
 - The interrupts has generated by SystemC device via an ad-hoc Socket port (called Socket Interrupt Port)
 - When the Interrupt occurs, the Interrupt Service Routine has to be started

Driver-Kernel Programming Model



Example ISS

```

int main() {
    // Define the Interrupt Service Routine
    DRIVER.attachInterruptServiceRoutine(ISR);
    return 0;
}
// Interrupt Service Routine
ISR() {
    int address = 0x4F00;
    int separator, offset = 32;
    // variable declaration to transmit
    // with SystemC
    int in_var;
    int out_var;
    // write variable via DRIVER
    DRIVER.SET_ADDR(address + offset);
    // Print data from SystemC
    cout << "Data = " << DRIVER.GET_DATA();
}

```

```

SC_MODULE(m) {
    // port to receive a data from DRIVER
    driver_in in_port;
    // port to send a data to DRIVER
    driver_out out_port;
    // port to send an interrupt
    driver_int interrupt;
    ...
    SC_CTOR(m) {
        in_port=new driver_in(100);
        driver_proc(funcnt_1, in_port);
        out_port=new driver_out(200);
        iss_proc(funcnt_2, out_port);
        interrupt=new driver_int();
    }
    ...
    interrupt->set();
}

```

```

SET_ADDR(int addr) {
    // Send data to SystemC PORT 100,
    // via Socket
    socketSendData(message, 100);
}
data GET_DATA() {
    // Send data to SystemC PORT 200,
    // via Socket
    socketSendData(message, 200);
    return data;
}
attachInterruptServiceRoutine(sr()) {
    // Set ISR called when an Interrupt
    // occurs.
}

```

DRIVER

Socket Interrupt Port Socket Data Port

Speedup

- Linked ISS is an order of magnitude faster than MAL.
- Remote ISS is faster than MAL by a factor of about two.

Thank you

GNU was launched in 1984 to develop a complete Unix-like operating system which is free software: the GNU system. (GNU is a recursive acronym for "GNU's Not Unix"; it is pronounced "guh-noo".)

GNU's Not Unix!

What is GDB?

GDB, the GNU Project debugger, allows you to see what is going on 'inside' another program while it executes -- or what another program was doing at the moment it crashed.

Free as in Freedom

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

The program being debugged can be written in C, C++, Pascal, Objective-C (and many other languages). These programs might be executing on the same machine as GDB (native) or on another machine (remote). GDB can run on most popular UNIX and Microsoft Windows variants.

Multi-language Cosimulation

- Using several simulators concurrently
 - "Backplane" = network & software for inter-simulator communication & synchronization
 - Analogy: PCBs plugged into physical backplane

Operation of Emulator