Virtual Development Environment Based on SystemC for Embedded Systems*

Sang-Young Cho, Yoojin Chung, and Jung-Bae Lee

[1] Computer Science & Information Communications Engineering Division
Hankuk University of Foreign Studies, Yongin, Kyeonggi, Korea
{sycho,chungyj}@hufs.ac.kr
[2] Computer Information Department
Sunmoon University, Asan, Chungnam, Korea
jblee@sunmoon.ac.kr

Abstract. Virtual development environment increases efficiency of embedded system development because it enables developers to develop, execute, and verify an embedded system without real hardware. We implemented a virtual development environment that is based on SystemC, a system modeling language. This environment was implemented by linking the AxD debugger with a SystemC-based hardware simulation environment through the RDI interface. We minimized modification of SystemC simulation engine so that the environment can be easily changed or extended with various SystemC models. Also, by using RDI, any debugging controller that support RDI can be used to develop an embedded software on the simulation environment. We executed example applications on the developed environment to verify operations of our implemented models and debugging functions. Our environment targets in ARM cores that are widely used in commercial business.

Keywords: Virtual development environment, SystemC, Embedded system development, Remote debug interface, Hardware simulation.

1 Introduction

The biggest challenge of bringing an embedded system solution to market is delivering the solution on time and with complete functionality required because market is highly competitive and demands of consumers rapidly change. Most embedded systems are made up of a variety of Intellectual Property (IP) including hardware IP's (processors and peripherals) as well as software IP's (operating systems, device drivers and middlewares, like a protocol stack).

Virtual Development Environment(VDE) is an embedded system development environment that can verify a hardware prototype, develop a software without a real hardware, or be used for co-design of hardware and software.

* This research was supported by the MIC Korea under the ITRC support program supervised by the IITA.

Y. Shi et al. (Eds.): ICCS 2007, Part IV, LNCS 4490, pp. 949–956, 2007.

This environment usually provides a hardware simulation model, simulation engine, and other tools that are useful in software development, and thus increases efficiency of embedded system development [1,2,3].

Virtual Platform[1] is Virtio's commercial virtual development environment. It supports many different types of processors such as ARM, X-Scale, and MIPS. Virtual Platform supports to develop software for a target system. MaxSim[2] is comprehensive SoC development environment provided by ARM. It consists of fast and easy modeling and simulation, and tools that provide debugging. Additionally, it enables system and hardware developers to compose the most suitable architecture quickly and accurately, and software developers to develop software before actual hardware comes out by providing VDE. Virtual ESC[3], made by Summit, is a set of 'Fast ISS' models and 'platform-based packages', which are composed of TLM-based busses, memory, and peripheral device models. According to its setting, it can comprise and run many different abstract-level systems, and debug related software and firmware. Also, it can create virtual prototypes for fast software development. The commercial VDE's described above integrate hardware simulation tools and software development tools tightly. Therefore it it limited that an embedded system developer uses various software tools and hardware simulation tools from different companies or organizations flexibly.

In this paper, we describe our design and implementation of a SystemC-based virtual development environment for developing embedded systems. We built virtual target hardware environment that is composed of various SystemC hardware models, and software development environment. We implemented the virtual hardware environment by implementing ARM processor core, its memory model, and other hardware IPs using SystemC. Also, we linked the hardware simulation environment with AxD(ARM eXtended Debugger), a debug controller provided by ARM's software development environment, through RDI(Remote Debug Interface). This system can be run with any debugger that implements RDI, and applied to any SystemC-modeled virtual hardware system.

The rest of our paper is organized as follows. In section 2, we describe related studies that are necessary for developing our environment. In section 3, we explain design and implementation of our virtual development environment for embedded system development. Finally, section 4 ends our paper with conclusion.

2 Related Studies

2.1 SystemC

SystemC is a system modeling language that can model and operate hardware at system-level. SystemC can easily express a complex SoC core at a high level while having all the merits of hardware description languages. It was developed using C++ classes. Hence, SystemC can be effectively used for simulation environment in checking not only hardware operation but also software one. Also, it supports TLM(Transaction-Level Modeling)[4,5].

We implemented the virtual development environment using SystemC version 2.0.1. Version 2.0.1 provides not only register-level transmission modeling, but also algorithm-and-function-level modeling. SystemC class libraries provide essential classes for modeling system structure. They supports hardware timing, concurrency, and reaction, which are not included in standard C++. SystemC allows developers to describe hardware and software, and their interface under C++ environment. Main parts of SystemC are as follows.

- Module: A container class that can include other modules or processes.
- Process: Used for modeling functionality and defined within a module.
- Ports and Signals: Signals connect modules through ports. (Modules have ports, through which they are connected to other modules.)
- Clock: SystemC's special signal (act as a system clock during simulation.)
- Cycle-based simulation: Supports an untimed model and includes high-level function model to RTL-level, which has clock cycle accuracy.

Figure 1 shows an example of system modeling in SystemC. A module can include processes or other modules due to its hierarchical structure. Processes run concurrently and do function modeling. Thus, they cannot have a hierarchical structure. Processes can communicate with each other through channels. Signals are modeled in the simplest form of a channel.

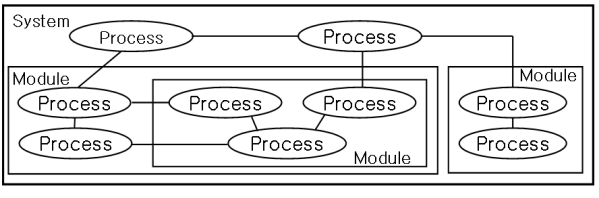

Fig. 1. A system modeling of SystemC

2.2 RDI (Remote Debug Interface)

RDI[6] is a C-based procedural interface that interfaces an ARM certified debug controller with a debug target. An RDI debug controller is a program that generates specific requests to a debug target through RDI. An RDI debug target then controls the received requests coming through RDI, so that the target hardware or software can perform debugging procedures. RDI consists of interface related to the controlling of the running of a debug target, and read/write operations of the states of the debug target. We used the RDI 1.5.1 that is an in-process interface and RID requests are handled through an RDI procedure vector in the form of windows DLL called WinRDI.

The primary purposes for WinRDI are as follows.

- To offer general mechanism to gain the RDI interface.

- To offer processes needed to adapt the versions of the debug controller and the debug target.
- To allow the setting of the debug target if necessary.

3 Design and Implementation

To build the development environment, we used SystemC to make the ARM7 core and memory. We also designed and implemented an interface using WinRDI so that the built core and memory can be connected with ARM's debug controller. Figure 2 shows the overall operation of the developed environment.

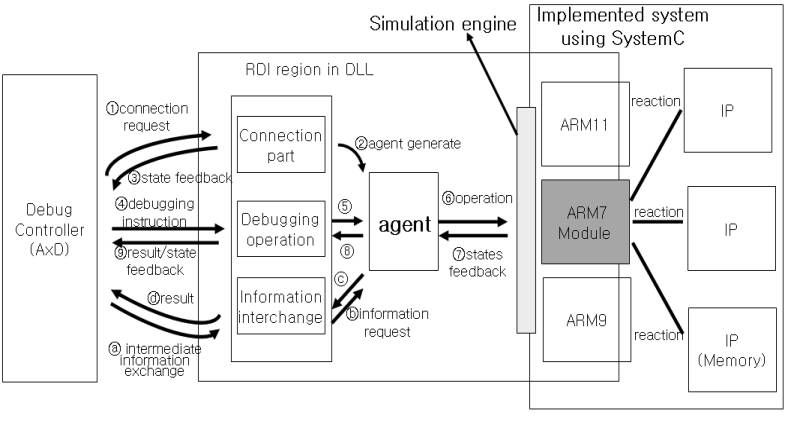

Fig. 2. The overall operation of the implemented VDE

If we set AxD (Debug controller)[7] to connect the implemented DLL, functions related to the connection are called and initializes the implemented interface. Next, AxD calls Info function on occasion and receives relevant information related to the modules and other peripherals that were requested at specific times. After the connection is set up, an agent is created and AxD recognizes SystemC-based modules and IPs according to its setting.

After this, if we run an application, the debugging becomes possible. When the debugging functions of AxD are called according to the debugging-related buttons that were pressed, the relevant functions of the agent are called, and the agent reads/sends the information that AxD want; AxD can set the values of registers and memory of the debug target.

RDI can be divided into three parts: the connection part, the debugging part, and the information exchange part. In the connection part, AxD sets the target according to RDI using the DLL. The connection part also enables AxD to connect to the target by processing the information of the target. At this point, we can create an agent and handle many targets. In our work, we connected only one target. In the debugging part, debugging functions are referenced

in Traget_ProVec of the agent so that appropriate debugging functions can be called. Finally, the information exchange part is called occasionally when AxD needs the state of the target.

3.1 Implementation of the Interface for the Connection to AxD

In this subsection, we describe interface functions that are necessary to connect debug controllers with connection procedures. The followings are the implemented entry points necessary for the connection with AxD.

- WinRDI_Valid_RDI_DLL: This function can be called at any time and checks the exact name of the implemented DLL.
- WinRDI_GetVersion: This function can be called at any point after the WinRDI_SetVersion is called. It is defined in winrdi.h and returns what type of RDI or WinRDI is supported in the DLL using a macro.
- WinRDI_Get_DLL_Description: This function shows the user the name of the DLL in null-terminated ASCII values.
- WinRDI_GetRDIProcVec: This can be called by the debug controller at any time. It creates the RDI entry point including struct RDI_ProVect for the debug target and returns the pointer.
- WinRDI_Register_Yield_Callback: The debug controller cannot enter into RDI_StepProc and RDI_ExecuteProc while the target is running. This function is used to solve that problem.
- WinRDI_Info: This function returns the information that the debug controller requests.
- WinRDI_SetVersion: This function returns the version that the debug controller demands from the target.
- WinRDI_Initialise: The DLL does the initialization required by this function and prepares RDI_OpenAgentProc.

To allow AxD to initialize the debug target, we made AxD to operate according to the procedure of RDI calls. (The termination procedure takes symmetrically to the initialization process.) At first, AxD initialize the agent for debug target and verify its handle. Then, AxD counts the number of modules (processors) of the debug target. Finally, for all the modules of the debug connection, AxD opens them and initializes each modules.

3.2 Modified Parts of SystemC.lib

SystemC generally uses SystemC.lib for modeling and simulation. Therefore, to connect with a debug controller, we analyzed the internal procedure of SystemC.lib and modified it according to our needs. To do this, we removed main() of SystemC.lib, which starts the simulation, and re-built SystemC.lib. Then we connected the starting function, sc_main(), of the SystemC.lib with the OpenAgentProc() of the implemented rdi.dll. Thus, the simulation can be controlled by the OpenAgentProc(). The class Csimul is implemented for the connection of RDI and SystemC simulation modules. Figure 3 shows the CSimul's behavior.

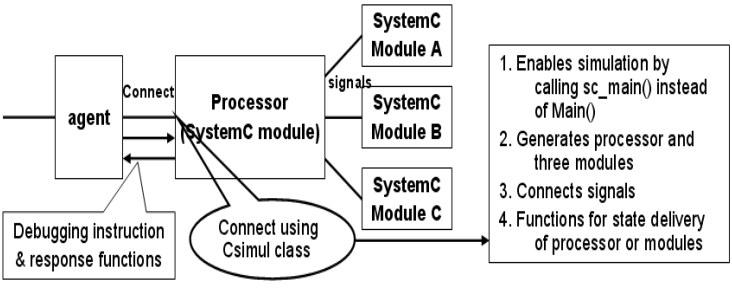

Fig. 3. CSimul class for the connection of RDI and SystemC simulation

The behavior of class CSimul is as follows.

1. Make sc_signal to control input/output wires of modules.
2. Connect signals after a core in SystemC is made.
3. Connect signals after memory in SystemC is made.
4. Create functions necessary for reading and writing of the state of the core.
5. Allows the result of simulation to be saved in the data waveform *vcd*.

To implement an ARM processor SystemC model, we used the ARM7 core C-model in GNU's GDB. The core was coded in C and provides simulation environment that, linked with GDB, runs assembly instructions. To change the C-model core into a SystemC model, we encapsulated the interface into SystemC while maintaining the internal part to run in C. Also, to make the control of debugging possible, we read internal information from pipelining functions, and the simulation runs in steps after saving the information. We also designed simple synchronized SystemC memory to run with SystemC core model.

3.3 SystemC Module Controlling Method for Debugging

In this subsection, we explain a method for the debug controller to debug a target while controlling the SystemC modules at the same time.

The simulation can start when all the necessary modules are created and the connection between signals and modules are validated. RDI calls CSimul's method, CSimul.init(), to control the starting of the simulation. This initialization process means the starting of the most top module of the simulation, and the actual simulation starts when sc_start() is called from the most top phase. The function sc_first() can have a double-type parameter and many time units as its value. If we want to run the simulation without a time limit, we can put a negative number for the parameter. All these functions run while the clock signal ticks for an assigned amount of units. When the time unit is all spent, the SystemC scheduler is called.

The simulation can be terminated at any time by calling sc_stop() with no parameters. But it is difficult to understand all the details above and to implement

the exact operations for each debugging step that the debug controller requires. To solve this problem, we used sc_initialize() function that starts the clock and controls the simulation, rather than sc_start(), which SystemC provides.

We initialize the SystemC scheduler using sc_initialize() function, not sc_start(). Then, we will be able to write values on the signals and simulate the result of the set values. This function takes a double-type parameter and can call the SystemC scheduler.

To implement one-step operation, we used sc_cycle() function and some internal variables such as F_pc, E_pc, and A_pc. The followings are implemented RDI debugging functions: RDI_ReadProc, RDI_WriteProc, RDI_CPUReadProc, RDU_CPUWriteProc, RDI_SetBreakProc, RDI_SetpProc, RDI_ExecuteProc, and etc.

3.4 Verification of the Developed Environment

The virtual development environment consists of the debug controller AxD, implemented interface, and SystemC modules. To verify the developed environment, we ran the developed environment and used vcd-file creation function of SystemC simulation environment to output the operation states and check its waves. We first checked the simulation of hardware models, the core and memory, through the vcd file. Next, using CodeWarrior of ARM Developer Suit v1.2, we had made many applications in assembly and C languages to verify wether the core module and the memory module correctly worked according to the debugging instructions (setbreak, getbreak, go, run, memory read/write, register, and read/write, etc.) generated from the debug controller through RDI. And thus, we checked whether the values of the core and memory changed and whether we could read/write the states. Figure 4 is the picture captured for checking the debug operation of an assembly and C programs.

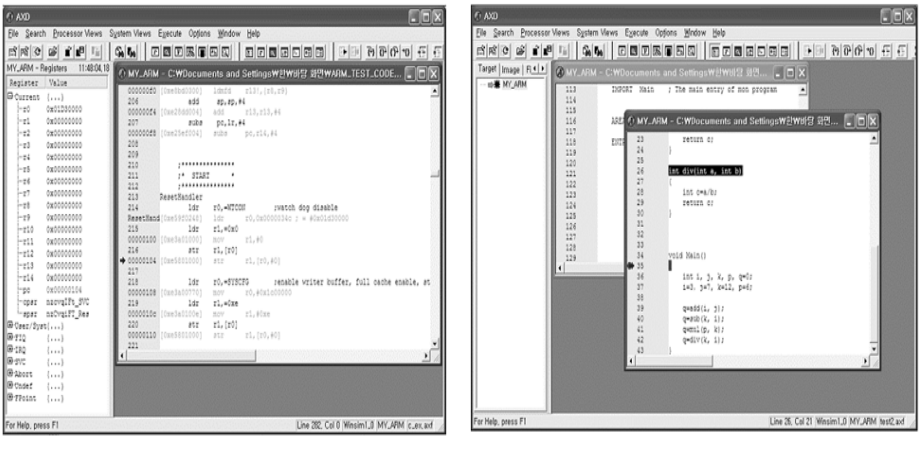

Fig. 4. An assembly program and a C program debugging test

4 Conclusion

In this paper, we described the design and implementation of a SystemC-based virtual development environment for developing embedded systems. The virtual development environment reduces the cost of the development of the embedded system by enabling engineers to write embedded software without real hardware. We implemented a virtual development environment that is based on SystemC. This environment was implemented by linking the AxD debugger with a SystemC-based hardware simulation environment through the RDI 1.5.1 interface. We minimized the modification of SystemC simulation engine so that the environment can be easily changed or extended with various SystemC models. The hardware simulation environment employed an ARM core that is the most commonly used one in commercial business. We implemented several SystemC-based hardware IPs for the virtual hardware environment.

For the verification of the developed environment, the core and memory were linked with the debug controller AxD. We ran example applications and verified the accuracy of our environment by checking the debug operations. The developed virtual development environment can be used in many phases of embedded software development such as developing a device driver, porting an OS, and developing an application.

The developed environment used the ARM processor core that is very popular in commercial business, so the environment is very useful in various application areas. Our environment uses the SystemC-based hardware simulation environment for the system-level design. So, the processor model, memory model, and IP model can be extended easily, and this environment can be run with many open SystemC models.

References

1. Virtio Corp. *VPDA295 Virtual Platform*, http://www.virtop.com/products/page/0,2573,33,00.html
2. ARM Corp. *Virtual Prototyping Solution*, http://www.arm.com/products/DevTools/MaxCore.html
3. Summit Design Corp. *Platform based Solutions*, http://www.summit-design.com/content.asp
4. Grotker, T. and Grotker, T., *System Design With SystemC*, Kluwer Academic Pub., (2002)
5. Ghenassia, F., *Transaction Level Modeling With SystemC*, Springer Verlag, (2006)
6. ARM Corp. *ARM RDI 1.5.1 RDI-0057-CUST-ESPC-B document*, (2003) p.206
7. ARM Corp. *ARM Develop Suite version 1.2 Debug Target Guide*, (2002)