1

SYSTEMC-BASED ELECTRONIC SYSTEM LEVEL DESIGN METHODOLOGY FOR SoC DESIGN-SPACE EXPLORATION

Yuan Wen, HAU
Mohamed Khalil-Hani

5.1 INTRODUCTION

The trend today is to apply embedded systems based on System-on-Chip (SoC) in the design of electronic systems. Such SoC solutions typically consist of embedded processor(s), embedded memories, hardware accelerators (or IP cores), high-speed communication interfaces and reconfigurable logic. Consequently, the development of these electronic systems has become increasingly complex, as they impose more severe demands (such as lower cost, higher performance, product quality, security, and time-to-market). In addition, as Moore's Law drives further the capabilities of digital hardware, there is a demand for greater number of functionalities to be conceived in a more constrained design space (Camposano, 1997).

One of the key challenges of the SoC design is the partitioning of system functionality across the HW/SW dichotomy. A functionality once relegated to software is now, for enhanced performance, implemented in hardware, while hardware components must integrate with higher-level software APIs. In current CAD methodology, *a priori* definition of the partition is

made, thus creating separate hardware and software specifications. Changes to the HW/SW partitioning necessitate extensive redesign which usually ends up with sub-optimal designs. Furthermore, with the introduction of embedded processors in FPGAs, digital designers are exposed to a new field of CAD, which involves the concurrent development of both hardware and software (program executed on the embedded processor).

Another critical drawback of the current CAD methodology is that it is RTL-centric which, due the increase in circuit complexity, suffers from long simulation time, which is gradually becoming unacceptable. It is clear that speed of simulating a complete system is a critical factor when designing a complex digital system such as an SoC. This verification issue is further exacerbated when the amount of test vectors needed for verification rises by a factor of 100 every six years, which is 10 times the increase of the number of gates on a chip as stated by Moore's law (Camposano, 1997). In addition, the complete verification and validation of the system functionality is often not possible until a fully working prototype has been built. This is especially true for a design in a highly distributed and heterogeneous environment, such as a networked embedded system (Klingauf and Gunzel, 2005).

Clearly, in order to reduce development time and cost, computer-aided design (CAD) tools must now include features that facilitate design-space and architecture exploration, and promote design at a higher level of abstraction. Among the solutions to the above-mentioned design issues, being actively pursued today is to capture the design at the Electronic System Level (ESL) of abstraction, and applying the standard design language of SystemC (Bocchio et al., 2005; SystemC Homepage). This approach will require a hardware-software (HW/SW) co-design and co-simulation framework that facilitates design-space exploration and provide

high simulation speed. References (Fummi et al., 2007; Benign et al., 2003; Yuyama et al., 2004) have proposed co-simulation methodologies based on SystemC with an Instruction Set Simulator (ISS) as a model of the processor in a common source file. In (Sayinta et al., 2003), a SystemC abstract model of the system was used as a golden reference model for enabling a vertical reuse of testbenches during the whole design process. Work on SystemC-VHDL co-simulation has been reported in (Bombana and Bruschi, 2003; Maciel et al., 2007), in which discussions were provided on the advantages of the technique to validate new models and reuse previous designs. References (Hodjat et al., 2005; Sakiyama et al., 2006; Gezel2 Homepage; Schaumont and Verbauwhede, 2006) proposed GEZEL co-design platform, and illustrated its application in the design of an elliptic curve cryptographic coprocessor. Much research is ongoing on this subject of ESL, but however, most current SoC design platform still suffers from limited architecture exploration, lacks distributed, real-time support and standard design language and IP reuse facilities.

In this chapter, we present a SystemC-based ESL HW/SW co-design methodology used in the design of SoC-based embedded systems. This methodology aims to provide the ability to quickly develop and evaluate complex SoC and embedded system designs. The associated co-simulation platform is implemented entirely in SystemC language, except for the Instruction Set Simulator (ISS), which is wrapped under SystemC. This methodology and the design platform enable early system functionality verification, as well as new algorithm exploration before the final implementation prototype is available. It can be used to validate the behaviour for both the hardware and the software modules of the embedded SoC, as well as the interaction between them with timed/cycle-accuracy. By having an early simulation model, it permits the evaluation of the complete system at an early stage of the design flow. This can

avoid extensive redesigning, which can contribute to significantly long design time and incur high design cost, and furthermore, the result usually ends up with sub-optimal designs. Besides, the framework also aims to facilitate architecture exploration that assists the system designer in finding the best HW/SW dichotomy. Another key advantage is the speed of simulation at the system level is significantly faster than RTL simulation of the whole system. It is also independent of any vendor specific tools.

Section 1.2 of this chapter describes the proposed SystemC-based ESL co-design methodology. Section 1.3 briefly describes the proposed co-design/co-simulation platform. Section 1.4 presents the design of an Elliptic Curve Crypto (ECC) SoC as a case study to illustrate the design flow, and the design refinement steps applied are described in Section 1.5. Results and conclusions are provided in Sections 1.6 and 1.7 respectively.

5.2 PROPOSED SYSTEMC-BASED ESL HW/SW CO-DESIGN METHODOLOGY

Figure 1.1 shows the architecture of the proposed SystemC-based ESL HW/SW co-design methodology, which consists of four design abstraction levels, namely: (a) specifications level - UML modelling, (b) system level (functional) - SystemC modelling, (c) system level (architectural) - modelling in SystemC and C/C++, and (d) RTL - modelling in HDL and C/C++. In the proposed methodology, SystemC is mainly used as system modelling language and simulation kernel at the system levels of abstraction. It is an extension of C++ class library, which provides both modelling and simulation kernels based on discrete event structures. It can be used to effectively create cycle-accurate models of software algorithms, hardware architectures, interfaces, and system-level designs (SystemC Homepage). One of the key strengths of SystemC is that it allows modelling at different levels of abstraction, supports the refinement of high level models down

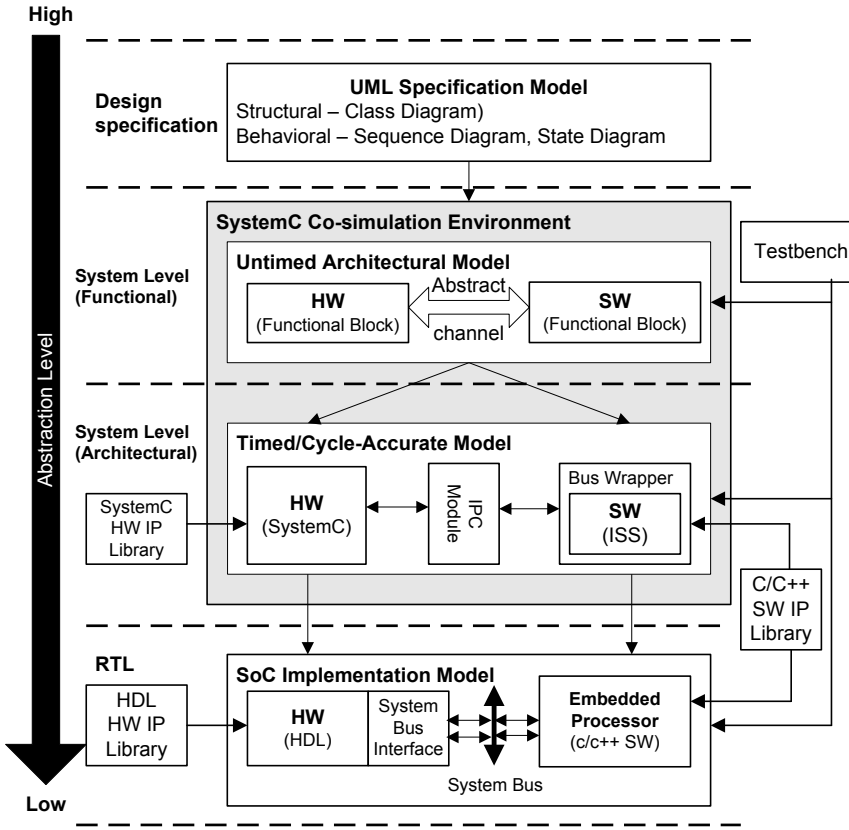

Figure 1.1 HW/SW co-design environment.

to low levels abstraction and even combining them into a single model (Grotker et al., 2002; Gerlach and Rosenstiel, 2000). SystemC also can model both hardware and software modules in the same source file. A SystemC model is an *executable specification* of a system, which means a C++ program that exhibits the same behaviour as the system when executed.

With reference to the diagram of the proposed HW/SW co-design methodology depicted in Figure 1.1, we now describe the design flow from specification model to the final implementation model

targeted for prototyping in an FPGA development board. The design flow begins with the modelling of the specifications using UML. The light-weight UML 2.0 can be used to model almost any type of application, running on any combination of hardware, operating systems, programming languages, and networks (UML Homepage). In this work, the UML class diagram is used to describe the static architecture, while the UML sequence and state diagrams are used to describe the dynamic behaviour of the system to be designed. Essentially, the UML specification model serves as a "schematic" design entry or *paper specification model* for the SystemC *executable model*.

From the UML specification model, we obtain a system level functional model. This is a SystemC untimed functional model (UTF) of the system architecture, and it is the golden reference model to verify the system functionality and algorithm in the following design abstraction level. It bridges the algorithmic world (possibly validated in Matlab) to the lower levels in the design hierarchy.

At the next design level, the system level architectural model is obtained, where partitioning is made for hardware (HW) and software (SW). Timed functional models (TF) of the HW and SW are designed in parallel, with the HW functional models being described behaviourally in SystemC, while the SW and hardware device driver firmware (FW) blocks are written in C and executed in an Instruction Set Simulator (ISS), which is wrapped under a SystemC bus wrapper. Depending on the modelling accuracy required, when needed, cycle-accurate (CA) of TF can be created for performance-critical modules. Co-simulation of the SystemC HW and SystemC-ISS SW can now be performed to verify the functionality of the whole system.

This co-simulation can be used to check the interoperability of a

single designed module (HW or SW) with the rest of the system without having the whole system being first implemented. Also, the co-simulation can provide hardware profiling which assists the system designer to find the best HW/SW partitioning such that a balance between area constraints and targeted system performance (area-speed design tradeoffs) can be achieved.

Once the system architecture and HW/SW partitioning has been chosen, the system-level design above is refined further to RTL model for implementation. At this implementation level, all models of the computation and communication detail are taken into consideration. The main advantage of this methodology is that, by having an early simulation model of the complete system, system verification can be made well before the final implementation of the design is available; unlike in the traditional RTL methodology, system validation can only be performed when a fully functional prototype has been built. HW/SW integration can be explored at high level of design abstraction, and design errors can be discovered in early design cycle to shorten the product time-to-market.

5.3 SYSTEMC-BASED HW/SW CO-SIMULATION PLATFORM

Co-simulation design environments, currently under research, can be categorized as: (1) *homogeneous*, (2) *semi-homogeneous*, and (3) *heterogeneous* types (Fummi et al., 2007). The work in (Ptolemy Homepage; Berkeley, 1999; Slomka et al. 2000) pioneered the *homogeneous* co-simulation environment, in which a single engine is used for the simulation of both HW and SW components. Good simulation performance can be obtained with this method. However, the technique is just adequate in the very first stage of the development, before the HW/SW partitioning. It is because the HW and SW design flow need different techniques and tools when the abstraction level decreases toward a real

implementation.

Heterogeneous co-simulation environments use distinct simulation kernel and different language descriptions for HW and SW side. HW is normally described using HDL, while SW is modelled using high level language, such as C/C++/Java. It allows the mix-abstraction level co-simulation to produce a higher timing accuracy model, but it is suffered with the low simulation speed compared to the homogeneous environment. (Bombana and Bruschi, 2003; Maciel et al., 2007) are two of the examples that propose heterogeneous co-simulation platform.

By using SystemC as a hardware description language and the simulation backbone, a *semi-homogeneous* co-simulation environment can be created. The software parts are executed by a C/C++ Instruction Set Simulator (ISS), which simulate the behaviour of a general purpose processor. The hardware parts are described using SystemC. Co-simulation is performed in a SystemC environment, with the SystemC kernel as the master simulator. Other similar work using SystemC is reported in (Fummi et al., 2007; Yuyama, 2004; Sayinta et al., 2003). There is also other semi-homogeneous co-simulation environment available, such as GEZEL (Gezel2 Homepage), using the similar approach.

The HW/SW co-simulation environment proposed in this work is categorized into the *semi-homogeneous* co-simulation type. Due to lack of space in this chapter, we will only provide a description of the design of our co-simulation platform at the System Level (architectural) design abstraction level (refer to Figure 1.1). Figure 1.2 shows the design of the proposed co-simulation environment at this system abstraction level, which is built on the SystemC kernel as the master simulator. The resulting co-design framework facilitates architecture exploration that assists the system designer in finding the best HW/SW partitioning. In addition, the proposed

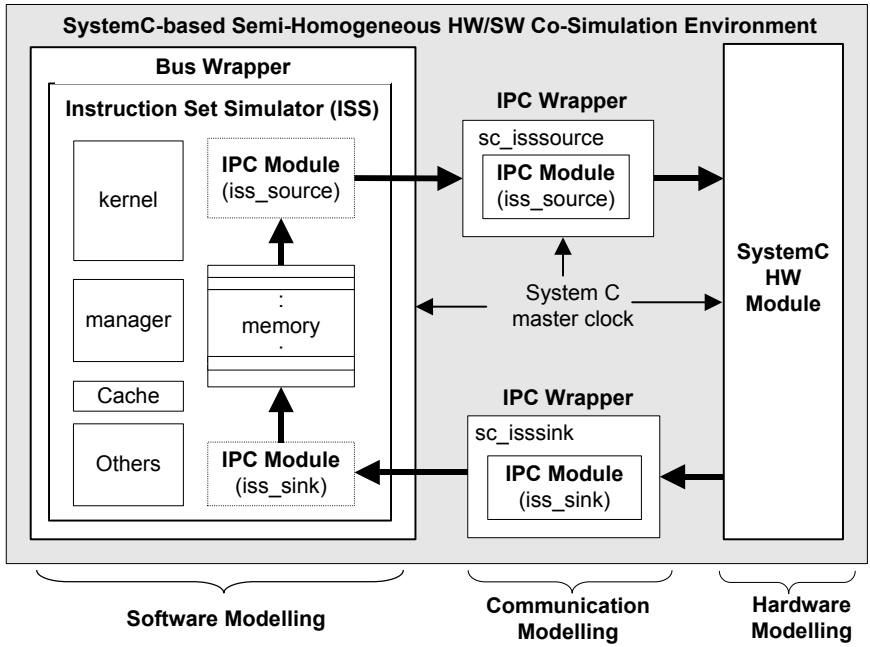

Figure 1.2 Proposed SystemC-based co-simulation platform.

HW/SW co-simulation platform can be used to validate the behaviour for both the hardware and the software modules of the embedded SoC, as well as the interaction between them, thus permitting the evaluation of the complete system at an early stage of the design flow, before any implementation prototype is built.

4.3.1 Software, Hardware, and Communication Modeling

In the proposed co-simulation platform, the software model-of-computation (MoC) is scripted using C/C++ high level programming language and simulated by an instruction set simulator (ISS). An ISS is a simulation model running in a general PC, to mimics the behaviour of a dedicated mainframe or microprocessor. It is built based on the Instruction Set Architecture (ISA) of a dedicated processor. Although SystemC can be used to create the ISS, its simulation speed is generally slower compared to

the ISS developed by the native C++ language. Besides, there is many C/C++ ISS of different processors already available compared to the SystemC ISS. In the proposed design platform, we choose the SimIt-ARM 2.1 (SimIt-ARM Homepage) as the ISS to simulate the software model. It simulates the StrongARM architecture. The motivation of choosing the SimIt-ARM is because it contains an instruction- and a cycle-accurate simulator, which meets our requirement to model a system in time- or cycle-accuracy. Besides, it is open source and has a very high simulation speed, as well as high accuracy. The ARM cross compiler toolkit is also available and well-established. In addition, the ISS supports memory address mapping to access the coprocessor.

Hardware MoC is modelled in SystemC to model, abstracted either behaviourally or RTL. Timed- or cycle accurate model is generated. The hardware MoC is also pin-accurate, and this is done by having special I/O pins to control the data flow with the ISS via IPC module and bus wrapper. To operate the hardware MoC, a device driver firmware (FW) need to be scripted using C/C++ programming language and simulated by the ISS.

In the communication modelling between the HW and SW partition, there are two main components involved, namely: (a) Interprocess communication (IPC) module, and (b) the bus wrapper. The IPC handles the communication between the ISS and the SystemC simulator. The bus wrapper is to ensure the synchronization between the system simulation and the ISS. It translates the information coming from the ISS into cycle-accurate bus transactions. These communication models require the ISS source code modification.

4.3.2 Interprocess Communication (IPC)

Interprocess communication protocol is a technique which describes various ways to exchange data between threads in one or

more processes. Its objective is to realise the efficient and fast data transfer and sharing. There are single-host IPC and IPC across a network. Single-host IPCs are such as shared memory and message passing using pipe, FIFO, and message queue. The IPC across a network are such as socket. In practice, single-host IPC is often much faster and sometimes simpler than IPC across a network.

Since SimIt-ARM ISS supports memory address mapping in accessing the coprocessor, we choose the shared memory of single-host IPC method to allow the data communication between the HW/SW partitions. It is the fastest form of IPC and no kernel involvement occurs in exchanging information between the processes. The memory within the SimIt-ARM ISS is used as the shared memory to communicate with the other hardware coprocessor. The shared memory declares a given section of memory to be used by several processors in parallel. The software simulated in the ISS can communicate with the coprocessors by accessing user-defined memory locations through memory address mapping. The ISS accesses these coprocessors using the same instructions that it uses to access memory. The concept is same as the memory mapped I/O. In general SoC design, the embedded processor also exchanges the data with the other I/O, coprocessor or HW accelerators using memory address mapping.

To enable the co-simulation, the macro (*COSIM_STUB*) of the SimIt-ARM needs to be defined. Besides, we need to add two IPC modules into ISS, so that it can send/receive data to/from hardware MoC through shared memory. In this context, we named the IPC modules as *iss_source* and *iss_sink*. The *iss_source* is to send the data from ISS to hardware MoC, while the *iss_sink* is vice versa. Referring to Figure 1.2, since the IPC protocol is based on shared memory, these two IPC modules are integrated with the ISS memory block. Figure 1.3 shows the pseudo code illustrating the behaviour of *iss_source* when the ISS is writing a data to the shared memory (send data to hardware MoC). The same case is

```
1. If write memory,
      1.1  Check the memory address registration.
      1.2  If address is registered for hw/sw co-simulation,
         1.2.1 Retrieve the IPC module dedicated to the
               memory address.
         1.2.2 Check the condition flag of the IPC module
               a. If FALSE, write the data to the data register
               b. If TRUE, back to Step 1.2.2.
      1.3  If not registered, do normal memory writing
      1.4  Back to Step 1.
```

Figure 1.3 Pseudo code of *iss_source* IPC module behaviour to write memory (send data to hardware).

applied to *iss_sink* with slightly different behaviour.

4.3.3 Bus Wrapper

In the proposed ESL platform, we need two wrappers: (1) ISS bus wrapper, and (2) IPC wrappers. It is to synchronize the data communication between the SystemC hardware MoC with the C++ ISS under control of the SystemC master simulation kernel.

Figure 1.4 shows the behavioural flowchart of the ISS bus wrapper. We use SystemC clock signal (*sc_clock*) as the system master clock to synchronize the ISS embedded in the bus wrapper, named *sc_iss*. When the SystemC clock triggered, the ISS is also triggered for one clock cycle to update the states and internal values. For the case of SimIt-ARM ISS, the clock cycle update operation is *clock_tick()* function.

The same method goes to the IPC wrapper design. The IPC wrapper is to wrap the C++ IPC module within ISS in the SystemC simulation kernel for data communication between HW/SW partitions with proper synchronization. It also translates the data type from ISS C++ variable to a specific signal that can communicate with pin-accurate SystemC hardware model. There are two wrappers need to be modelled: (a) *sc_isssource*, and (b)

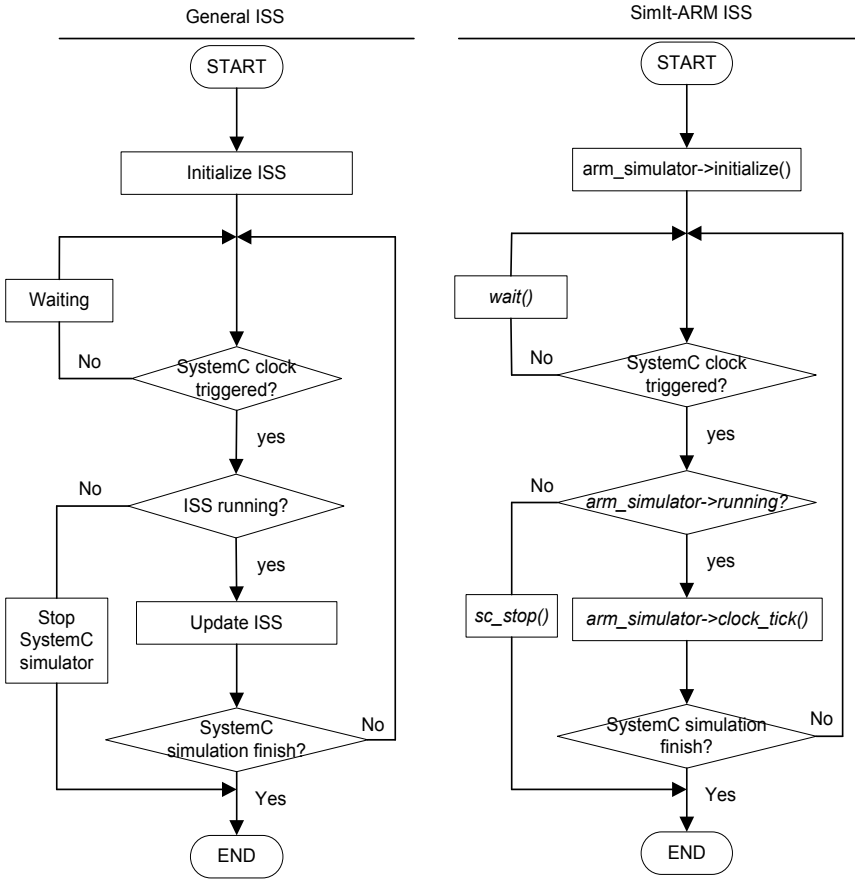

Figure 1.4 Behavioural flow chart of *sc_iss* bus wrapper.

sc_isssink. The *sc_isssource* is to wrap the *iss_source* IPC module, to send the data from ISS to SystemC hardware MoC. The *sc_isssink* is to wrap the *iss_sink* IPC module, to receive the data from SystemC hardware MoC and read by the ISS.

The only difference between these two IPC wrappers is the *sc_isssource* is synchronized by the SystemC master clock. Besides, the data communication between the ISS and the SystemC hardware is controlled using a simple handshake protocol, which acts as a mutex control. The mutex control (a condition flag)

controls the data availability of the data register in the *iss_source* IPC module within the ISS. For *sc_isssink*, the wrapper is without the SystemC clock and the handshake protocol is not required. To achieve the proper data synchronization between hardware and software partition, the system designer can implements the high-level two-way handshake protocol in device driver or API.

5.4 CASE STUDY: ELLIPTIC CURVE CRYPTO SoC

Typically, the EC-based crypto SoC is targeted to perform several EC-based cryptographic schemes, such as Elliptic Curve Digital Signature Algorithm (ECDSA), Elliptic Curve Diffie-Hellman (ECDH) key exchange protocol, etc. Figure 1.5 shows the arithmetic hierarchy of EC-based computations on prime finite field, *GF(p)*. The main computational operator is the point (or scalar) multiplication, which applies EC divisor operations of point addition and point doubling in *GF(p)*. These divisor operations require several *GF(p)* field arithmetic operations, mainly modular division, modular multiplication, modular addition and modular subtraction.

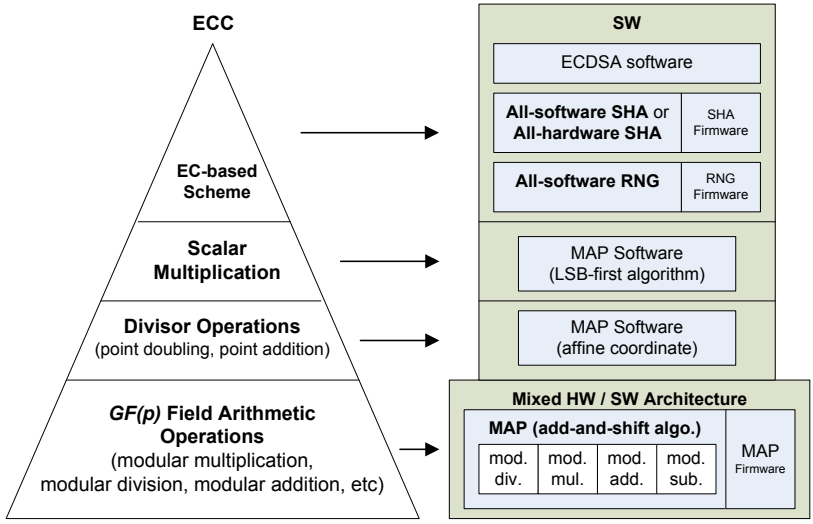

Figure 1.5 Arithmetic Hierarchy of EC-based system.

Referring to Figure 1.5, the field arithmetic operations (at the lowest level in the hierarchy) are realized in a Modular Arithmetic Processor (MAP), which can be partitioned either into HW and SW. HW is realized as behavioural model with timed-accurate based on add-and-shift algorithms as presented in (Hlavac, 2003; Shantz, 2001). The HW is paired with firmware (FW) as the device driver. SW implements the multi-precision prime field arithmetic algorithm with the C source code taken from (Rosing, 1999). The operations in the ECC arithmetic hierarchy above the field arithmetic operators are all realized in SW. The system designer can decide the final HW/SW architecture based on the profiling metrics output provided by the HW/SW co-simulation platform to fit their design constraint and targeted system performance.

5.5 REFINING THE DESIGN: FROM SPECIFICATIONS TO IMPLEMENTATION

In this section, we scope our discussion to the design and model refinement of ECDSA digital signature crypto subsystem. The ECDSA subsystem involves the modelling of several crypto operations, which include key deployment, message signing and signature verification. Due to lack of space, only simple exemplary models that are part of the message signing module in ECDSA are presented in this section. The corresponding algorithm to be mapped to the SoC is given in Figure 1.6. In the figure, the Modular Arithmetic Processor (MAP) is main module that implements the field arithmetic operations of ECC.

1. Generate a random number, k
2. Compute $R = (R_x, R_y) = k*G$
3. $r = R_x \pmod{n}$
4. $e = \text{SHA-1}(M)$
5. $s = k^{-1} (e + du \,.\, r) \pmod{n}$
6. Return signature = (r, s)

Figure 1.6 ECDSA message signing algorithm.

1.5.1 UML Specification Modeling

As mentioned above, MAP is partitioned into HW and SW, and the UML class diagram, shown in Figure 1.7, models the structure of MAP. Each of the class models in the class diagram performs a specific function, either carried out by hardware or software partition. Figure 1.8 shows the UML sequence diagram corresponding to the ECDSA signing algorithm given in Figure 1.6. It depicts the dynamic behaviour of system. Communication between each class model is through message passing. The functional behaviour within each class model can be described using UML state diagram.

1.5.2 SystemC Functional Modeling

The system-level model (functional) is now derived from the UML specifications model described in previous subsection. From the UML class diagram (Figure 1.7) each class is modelled to a

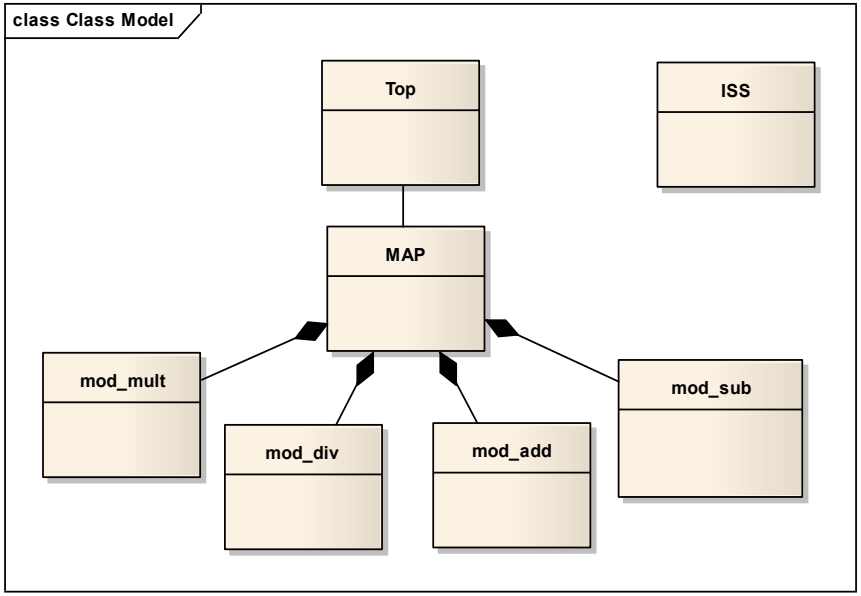

Figure 1.7 UML class diagram of modular arithmetic processor.

SystemC module (*SC_MODULE*). The behaviour of each module is derived from the corresponding state diagram (not shown here, for lack of space), and is modelled in a SystemC process (*SC_THREAD* or *SC_METHOD*). From the UML sequence diagram (Figure 1.8), an abstract FIFO channel in SystemC is derived to model the communication between the SystemC modules. (The FIFO is one of the simplest channels to control the data flow.) Figure 1.9 shows the resulting block diagram of SystemC untimed functional model. SystemC simulation of this model verifies the system architecture and functionality.

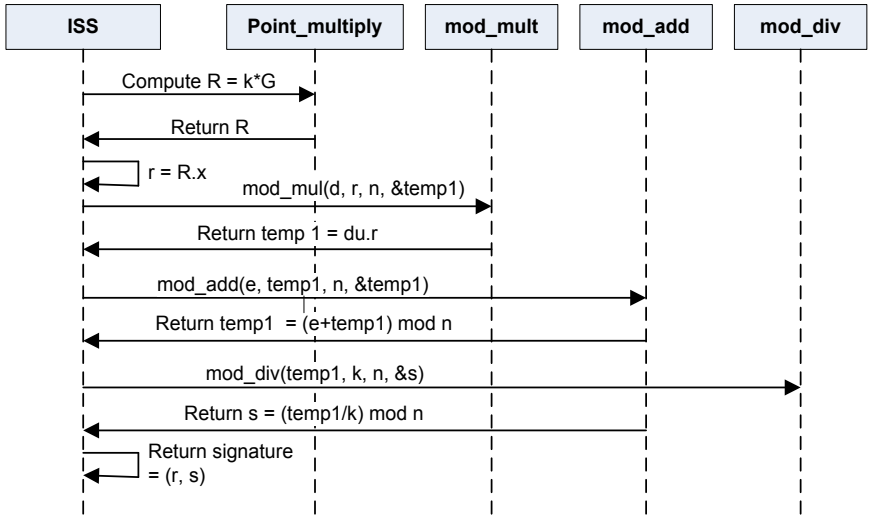

Figure 1.8 Sequence diagram of ECDSA signing operation.

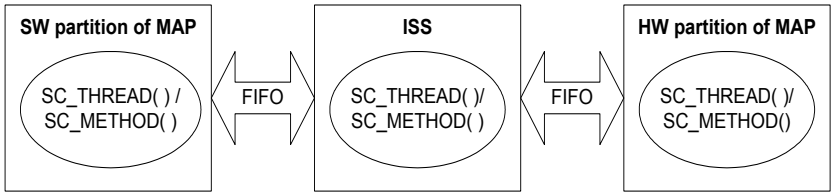

Figure 1.9 SystemC architectural model.

1.5.3 SystemC Timed Architectural Modeling

At this abstraction level, the designer performs the HW/SW partitioning of the system. Figure 1.10 shows the block diagram of SystemC timed functional model (TF). The HW blocks of the MAP is still in SystemC, while the SW module is now refined into C code, that is, as SW MoC for execution in the ISS.

For SystemC co-simulation to be performed, an inter-process communication (IPC) module with wrapper is created, and the ISS is wrapped with a SystemC bus wrapper. The IPC modules facilitate data communication between HW and SW. The bus wrapper and IPC wrapper ensure proper synchronization and handle the datatype conversion between the partitions.

In this case study, the ISS is SimIt-ARM version 2.2 (SimIT-ARM) to provide cycle-accurate simulations. Shared memory is used for inter-process communication. Communication between components (SW-SW, SW-HW) can be done using this shared memory, through SystemC signals. SW simulated in the ISS communicates with HW by accessing user-defined memory locations sitting in the ISS. Figure 1.11 shows an example of SystemC source code of ISS source module in IPC.

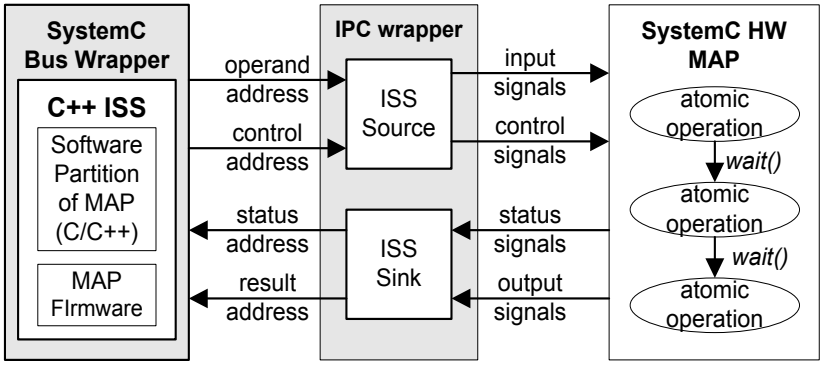

Figure 1.10 SystemC Timed Functional (TF) model.

```
#include "sc_armsource.h"
#include "armsim.hpp"
#include <iostream>
using namespace std;
//--------------------------------
//     sc_armsource
//--------------------------------
// Constructor
sc_armsource::sc_armsource(sc_module_name nm)
: sc_module(nm) , address(0) , arm_sim(0) , interface_id(0)
{    src = new arm_source;
     SC_THREAD(device_read);
          sensitive << clk.pos();  }
//Destructor
sc_armsource::~sc_armsource()
{    delete src;  }
// Set the parameter of the arm_source
void sc_armsource::setparam(sc_arm* _arm_sim, unsigned int _address)
{    arm_sim = _arm_sim;
     address = _address;
     interface_id = arm_sim->sim->mem->register_addr(address);
     register_armsource(interface_id, this->src);   }
void sc_armsource::device_read() {
     while(1)     {
          while (!(src->interface_written))    {
               //cout << "Waiting ARM processor writing data..." << endl;
               wait();        }
          src->reset_flag( );
    wait();    } }
```

Figure 1.11 SystemC code of IPC module (ISS source).

The SystemC hardware MoCs are pin-accurate, and this is done by having special I/O pins to control the data flow (unlike in the case of the abstract FIFO channel in architectural model). The communication delay between each component is estimated through access count on every IPC module. The hardware MoC can be made timed- or cycle-accurate. In each hardware MoC, the designer needs to determine every atomic operation that can be executed in parallel of every process of HW blocks. These atomic

operations, which can be executed in parallel, are separated from each other to different states by adding *wait()* statements with specific delay times. In this way, the SystemC model is able to provide cycle-accurate timing estimates, in terms of cycle count.

1.5.4 SystemC Implementation Modeling

At this abstraction level, HW MoC is refined to an RTL model (FSM-datapath). The resulting SystemC implementation model in RTL is shown in Figure 1.12, and it can provide a more accurate estimation of timing performance of the system. This SystemC RTL model is made to be synthesizable. SW and communication models are remaining unchanged.

The SystemC implementation model is then refined further to produce a RTL model for prototyping into an Altera FPGA hardware development board, either through auto translator tool or manual translation. The result is the system design shown in Figure 1.13, where SW on ISS is now translated to C code in Altera Nios II processor, and HW is RTL design in VHDL which is synthesize to the MAP co-processor IP core. The communication models are refined into a system bus interface module based on Altera Avalon Memory-Mapped (Avalon-MM) system bus specification.

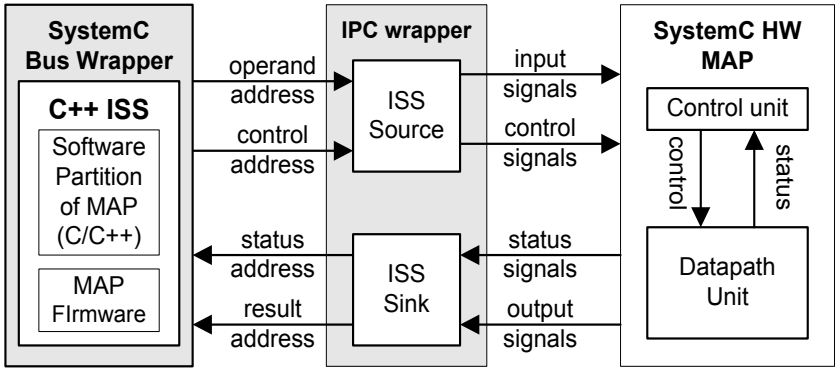

Figure 1.12 SystemC RTL model.

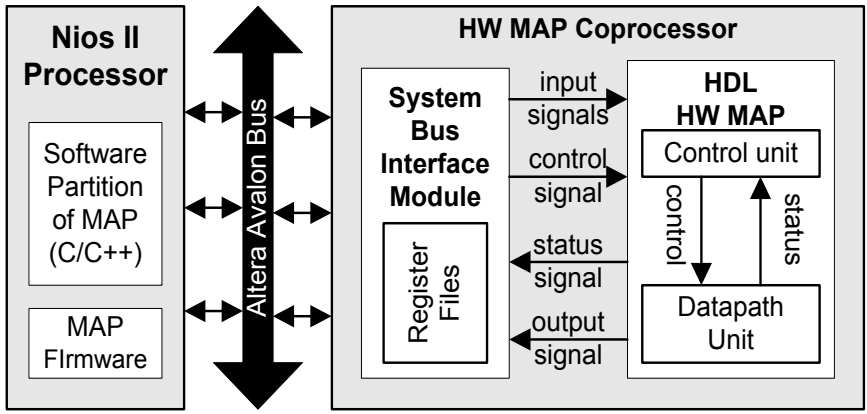

Figure 1.13 RTL Implementation model.

5.6 EXPERIMENTAL RESULTS

Performance test was conducted on the system to measure the execution times of the MAP processor designed in different HW/SW partitioning. The test vectors are taken from (Certicom, 1999). Table 1.1 shows the execution speed performances of the Modular Arithmetic Processor (MAP) in ECC in different configurations of the HW/SW partitioning. The result is dedicated to digital signature signing, but the performance metric is similar to other ECDSA operations, which is key deployment and digital signature verification operations. As mentioned earlier, the field arithmetic functions of modular addition, subtraction, multiplication and division are being partitioned either in HW or SW. In the table, columns marked with an 'HW' indicate that the field operation is performed by HW partition, while columns marked with an 'SW' indicate that the field operation is performed in SW partition. The execution speed-up is computed as follows:

$$\text{Speed-up} = \frac{\text{Execution time of system implemented completely in SW}}{\text{Execution time of system partitioned in SW/HW}}$$

The test results in Table 1.1 suggest that, for this case study, the

Table 1.1 Execution speed performance of ECDSA signature signing in different HW/SW partitioning

Mod Add	Mod Sub	Mod Mult	Mod Div	Computation cycle count (%)	Communication cycle count (%)	Total cycle count	Speed-up
SW	SW	SW	SW	100.00	0.00	1,521,056,507	1
SW	SW	SW	**HW**	99.87	0.13	344,953,590	≈ 4
SW	SW	**HW**	**HW**	87.45	12.55	20,464,814	≈ 74
HW	**HW**	**HW**	**HW**	46.07	53.93	8,715,993	≈ 174

best HW/SW partitioning is such that, the field operations of modular division and multiplication be computed in HW, while the modular addition and subtraction operations be performed in SW, taking into account the speed-area tradeoff. Intuitively, this is as expected, as modular multiplication and division are highly compute-intensive, hence should be offloaded to a HW accelerator for enhanced system speed. With this kind of performance profiling, the designer can explore the design-space and evaluate complex SoC solutions quickly and efficiently. For example, if the designer desires to design a server with an extremely high performance elliptic curve processing power, then results from Table 1.1 indicates that all the field arithmetic computations should be hardware accelerated. Such an implementation will have its system performance to be about 174 times speed gain over a design that is completely implemented in SW.

Performance tests were also conducted to compare the simulation speeds between SystemC timed functional (system level) model with the corresponding VHDL RTL model. SystemC simulation is performed using a terminal in Ubuntu Linux open source environment. The RTL simulation is performed on the VHDL design synthesized for implementation in an Altera Stratix Nios II-based FPGA development board using ModelSim 6.0. Both SystemC model and the VHDL model apply the same test vectors. Table 1.2 shows the simulation speed of prime field arithmetic computation, elliptic curve arithmetic computation, and ECDSA operations. For this simulation speed comparison, the simulation is

Table 1.2 Simulation Speed: System-level Model in SystemC against RTL Model in VHDL

Operations	Simulation speed (seconds)		Simulation speed gain
	RTL simulation	SystemC simulation	
Prime Field Arithmetic Computation, *GF(P)*			
Mod Division	20.28	0.072	281
Mod Multiply	20.66	0.048	430
Mod Add	20.34	0.061	333
Mod Subtract	20.20	0.038	531
Elliptic Curve Arithmetic Computation			
Point Add	24.98	0.092	271
Point Doubling	26.88	0.104	258
Point Multiply	1406.39	6.644	211
Elliptic Curve Digital Signature Algorithm (ECDSA)			
Key Deployment	1401.24	6.558	213
Signature Signing	1550.53	6.691	231
Signature Verification	3007.04	13.074	230

run on Intel Core2 CPU T5500 running at 1.66 GHz with 1GB RAM. The simulation speed gain is computed as follows:

$$\text{Simulation speed gain} = \frac{\text{RTL simulation}}{\text{SystemC simulation}}$$

It is observed that the simulation speed of the SystemC timed functional model is far more efficient than the VHDL RTL model, that is, at least 200 times faster.

5.7 CONCLUSION

This chapter has presented a SystemC-based HW/SW co-design methodology and co-simulation environment for design of embedded SoC. Details of design refinements illustrating the design flow, as well as the hardware modelling, software modelling and communication modelling are provided using a case study on the design of an elliptic curve crypto SoC. This methodology and the co-simulation platform is aimed to enable

early SoC design space exploration and system verification, fast simulation speed, and testbench reuse. The advantages of the proposed co-design framework achieved include: (a) a unified HW/SW representation, overcoming the difficulties in verifying the complete system and overcoming the incompatilibities across the HW/SW boundary, (b) facilitate design space explorations which allow different configurations of HW/SW partitioning to be evaluated early in the design, leading to more optimal designs faster, and (c) provide a well-defined modelling and co-design flow, which simplifies specification revision, redesign, leading to much improved design time-to-market.

Current implementation of the co-design platform has some limitations. Among these drawbacks include: for performance analysis, only execution speed is available, area profiling is not yet available; heterogeneous co-simulation between SystemC and HDL models is not yet available; RTOS modelling is not yet supported; manual code translation from one abstraction level to another one is prone to error and is time consuming – need automatic translators. These outstanding issues are the subject for further work in this research.

REFERENCES

Benign, L. et al. (2003). SystemC co-simulation and emulation of multi-processor Systems-on-Chip. *IEEE Computer*. 36(4):53-59.

Bocchio, S., E. Riccobene, A. Rosti and P. Scandurra. (2005). A HW/SW co-design environment based on UML and SystemC. *Forum on Specification & Design Languages*.

Bombana, M. and F. Bruschi. (2003). SystemC-VHDL co-simulation and synthesis in the HW domain. *Proc. Design, Automation and Test in Europe Conference and Exhibition*.

Camposano, R. (1997). Automating system implementation from system specification. *Talk at Synopsys University Day*, Aachen.

Certicom. (1999). GEC2: test vectors for SEC1. Working Draft Version 0.3.

Fummi, F., M. Loghi, G. Perbellini and M. Poncino. (2007) SystemC co-simulation for core-based embedded system. *Springer Journal, Design Automation Embedded System.* 11:141-166.

Gerlach, J. and W. Rosenstiel. (2000). System level design using the SystemC modeling platform. *Workshop on System Design Automation*. :185-189.

Grotker, T., S. Liao, G. Martin and S. Swan. (2002). *System Design with SystemC*, Kluwer.

Hlavac, J. (2003). *ALU for computing SLE's in modular arithmetic*. Diploma Thesis. Czech Technical University in Prague.

Hodjat, A., L. Batina, D. Hwang and I. Verbauwhede. (2005). A hyperelliptic curve cryto co-processor for an 8051 microcontroller. *IEEE Workshop on Signal Processing Systems*.

Home – Open SystemC Initiative (OSCI). www.systemc.org.

Klingauf, W. and R. Gunzel. (2005). From TLM to FPGA: rapid prototyping with SystemC and Transaction Level Modelling. *Proc. of Int. Conf. of Field Programmable Technology* :285-286.

Maciel, R. et al. (2007). A methodology and toolset to enable SystemC and VHDL co-simulation. *IEEE Computer Society Annual Symp. on VLSI.*

Main Page – Gezel2. http://rijndael.ece.vt.edu/gezel2

Object Management Group – UML. www.uml.org.

Ptolemy Project Home Page. http://ptolemy.eecs.berkeley.edu/index.htm.

Rosing, M. (1999). *Implementing elliptic curve cryptography*, Manning.

Sakiyama, K., L. Batina, B. Preneel and I. Verbauwhede. (2006). Superscalar co-processor for high-speed elliptic curve-based cryptography. In Cryptographic Hardware and Embedded Systems – CHES 2006, *Lecture Notes in Computer Science*, Springer-Verlag. 4249/2006:415-429.

Sayinta, A. et al. (2003). A mixed abstraction level co-simulation case study using SystemC for System on Chip verification. *Proc. Design, Automation and Test in Europe Conference and Exhibition*.

Schaumont, P. and I. Verbauwhede. (2006). A component-based design environment for ESL design. *IEEE Design & Test of Computers.* :338-347.

Shantz, S. C. (2001). From Euclid's GCD to Montgomery multiplication to the great divide. *Technical Report TR-2001-95*, Sun Microsystems Laboratories.

SimIt-ARM. http://simit-arm.sourceforge.net/.

Slomka, F., M. Dorfel, R. Munzenberger and R. Hofmann. (2000). Hardware/software co-design and rapid prototyping of embedded systems. *IEEE Design & Test of Computers*. 17(2):28-38.

University of California, Berkeley. (1999). A framework for hardware-software co-design of embedded systems. POLIS Release 0.4, December 1999.

Yuyama, Y., M. Aramoto, K. Kobayashi and H. Onodera. (2004). RTL/ISS co-modeling methodology for embedded processor using SystemC. *Proc. Int. Symp. on Circuits and Systems.* 5:V-305–V-308.