Verification of Embedded Real-Time Systems Using Hardware/Software Co-simulation

Mohammed El Shobaki
Mälardalen University, IDt/CUS
P.O Box 833, S-721 23 Västerås, Sweden
E-mail: mohammed.el.shobaki@mdh.se

Abstract

This paper presents hardware/software co-simulation and discusses its suitability in the development of real-time systems. Due to the facts that time-to-market challenge has increased the need for shortening the development process, new techniques and methodologies are introduced. A fashionable technique is to use co-simulation environments in the verification phase of a design process. As this technique is being adopted, and the fact that companies from a wide industry area has become using it, real-time issues are becoming relevant. This paper will discuss the use of a co-simulation tool for verification of embedded real-time systems.

1. Introduction

The article describe the use and experience of co-simulation as a verification tool in the real-time system design process (special embedded systems) and how today's verification methods can be complement to today's verification process.

The verification process occupies an increasing part of the total development time for real-time systems and today it is often the bottleneck in the development process. The increase in time for the validation stage is mainly dependent on four parameters:

1. Increased software complexity
2. Increased hardware complexity
3. Complex and high frequency of interaction between hardware and software,
4. The requirements of ”right first time”.

To shorten the development process it is a key demand to decrease the verification time.

The new tools for developing application-specific circuits (ASIC) have drastically reduced the design time and today the verification time is the bottle neck in the development process for ASICs [1]. Both in the software and hardware design processes the verification time is over 60-80%.

Real-time systems [2] are known as computer systems employed in environments where software execution has to meet timing constraints. Such systems are often realised in the fashion of an embedded computer system. Since the embedded system is a computer system it requires both hardware and software. Typically, the hardware consists of a CPU, memory, I/O components, and perhaps an ASIC together with some glue logic, all of which communicate over a common bus. The software part is divided into tasks of which then are scheduled, in the presence of a real-time operating system [2] (RTOS), to execute on the hardware.

In section 2, today's verification methods are briefly described as well as the new approach with Co-verification as a complement. Co-simulation techniques and methods used in two of the leading commercial tools are presented in section 3. Finally, real-time aspects on using co-simulation for embedded real-time system verification, are discussed in section 4.

2. Verification methods

In today's design processes the verification phase has become the major part, not only because it is time consuming but also due to the increasing complexity in both hardware and software. The partitioning of functionality in both hardware and software also increases the frequency of interaction between them.

These parameters among others make co-verification important. Today there are different methods for Co-verification, but the goal, which is to verify software and hardware execution respectively, remains the same.

2. 1 Software verification

In the absence of target hardware, verification of software code is today mostly managed using cross-development tools (compilers, debuggers, simulators, etc.). Whenever a peripheral hardware component is to be accessed, a piece of code (so-called a ”stub”) simulating the component is executed instead. Thus, at least verification of the functional behavior of the software can be achieved. Another method for software verification is to use the more realistic approach which uses existing hardware, typically implemented as a prototype, see below. This method also enables a more timing accurate verification. The major drawback is that software is verified late in the design process.

2. 2 Hardware verification

On the hardware side typically one is interested in verification of the interaction (accesses, handshaking, interrupts, signals, etc.) between software and ASICs and other system components. ASICs are typically designed at register transfer level (RT-level, see [3]). This level represents a complete functional model of the ASIC. The model must be verified in detail to demonstrate correct functioning together with the surrounding components and the software. One approach to achieve this is to use testbenches. In a testbench model the ASIC to be verified is instanced as a component, refer to figure 1. By using models of the surrounding components (e.g. CPUs, RAM, I/O, etc.) [3] stimulation input can be generated, thus enabling verification of the responses according to specification. Typically this is simulated on a workstation, often at a slow simulation speed if the designs are large. Consequently, simulation of software execution is a slow process which makes it diificult to simulate a complete program.

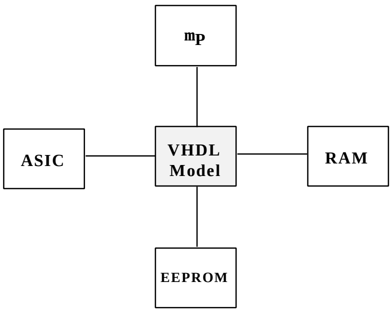

Figure 1: Testbench for a system simulation

After verification at the RT-level has been performed, the verification process typically continues on a fast prototype. Today a fast prototype is implemented in either a FPGA (Field Programmable Gate Array [3, 9]) or in a hardware emulator (typically uses FPGAs). The latter, is used to enhance speed of the hardware simulation tools on a workstation, thus enabling faster execution of software and complete programs can be verified. The major advantage when using FPGAs is the ability to make changes to a design very quickly compared to the traditional ASIC fabrication. While the emulator preserves observability into a design, the use of FPGAs only has limited observability. To view internal signal states in a FPGA one has to route the signals out to external pins. One disadvantage in the FPGA/emulator technique is that the timing is much slower in comparison with that in the ASIC. Also, both emulation and FPGA are relatively expensive to use.

2. 3 Co-simulation

Co-simulation for verification has recently been introduced as an alternative to testbenches and in some cases to fast prototyping. In fact, the idea of co-simulation was derived from using testbenches with processor models. The idea of the new method is to have real software execution as the event driver in a testbench and also to reduce the impact of software simulation time in the traditional testbench. An engine models the CPU which is instanced by a testbench (typically using VHDL or Verilog). There are different methods used to run the engine, but the overall technique in common is to conceal it from the processor’s interfacing to the hardware. Figure 2 illustrates a schematic overview of the connection of the hardware with the software through a controlling unit, a so-called co-simulation kernel.

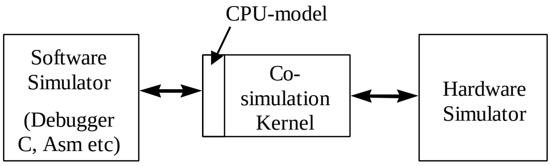

Figure 2: A co-simulation environment

3. Co-simulation tools

Today there are two noted commercial tools available, Eaglei [4] from ViewLogic, and Seamless [5] from Mentor Graphics. An evaluation on both tools is presented in [6]. They are very much similar but they use different techniques. An overview of the techniques used in these tools is presented below.

3.1 Seamless' Co-Verification Environment

In Seamless, the processor's functionality is separated from its interface. A Bus Interface Model (BIM) simulates the input/output pin behaviour for the hardware portion of the simulation. The software portion executes as a separate process, allowing much faster execution, either on an Instruction Set Simulator (ISS) or as Native Compiled Software (NCS). The ISS executes machine code produced by cross-compilers for specific processors. NCS is software compiled for execution on the host-machine. Communication between SW and HW is controlled by the co-simulation Kernel (CSK). Figure 3 shows the architectural structure in which Seamless operates.

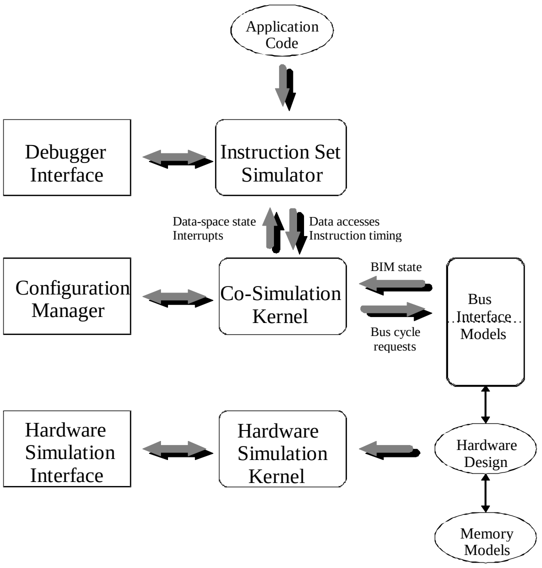

Figure 3: Seamless' architecture (ref. [6])

Supported ISSs and BIMs, respectively, are developed for the most popular processors on the market. Examples on processors include the x86 family, 68k, and the PowerPCs. These processors are not always fully modelled and there are some limitations. Some of these limitations are the lack of (or reduced functionality in) caches and memory management units (MMUs).

Apart from supported processor models, there are also different types of memory models available. These memory models have a particular connection to Seamless which enables optimisation of bus-cycles (generated by the BIM) for instruction fetches and data access. Supported types include SRAMs, DRAMs, FIFOs and register elements.

3.2 EagleI

Similar to the CSK in Seamless, Eaglei uses the VSP (Virtual Software Processor) to control the co-simulation. Eaglei supports three different models, VSP/Link, VSP/Sim and VSP/Tap, each suitable at different stages in a design process. The VSP/Link model uses a technique similar to the NCS execution (see below) which also is the fastest model. Also here, the VSP/Sim model is like an ISS with a true cycle behaviour. Not represented in Seamless' environment, is the VSP/Tap model which is a VSP that includes hardware in the form of an In-Circuit-Emulator (ICE). This technique is similar to the ISS but with the extension of a hardware accelerator. By using an ICE the observability needed is kept, thus it's possible to investigate internal registers and memory.

3.3 Native Compiled Software

Simulation using NCS is the fastest method when compared with the ISS approach because software is run directly on the workstation. NCS is easily produced by compiling software coded in any high-level language. Thus debugging of NCS can be done using a standard workstation debugger (e.g. dbx). The connection to the hardware process is done by placing calls to the VSP/BIM through an Application Program Interface (API). This interaction only drives the VSP/BIM pins to their defined values and cycles. The modelled processor's internal registers and cache memory is not available in this approach.

3.4 Instruction Set Simulator

An ISS is a software application that models the functional behaviour of a processor's instruction set. It runs much faster than a hardware simulation because it need not to cope with a processor's internal signal transitions. Since it is machine code for the target processor that is executed, you are free to use any language supported by the cross-development tools. The ISS reports the number of clock cycles required for a given instruction to the VSP/CSK. Notification of external events (e.g. interrupts, resets) from the VSP/BIM are reported to the ISS by way of the VSP/CSK.

4. Real-time Co-simulation

To allow true timing examination a model of a real-time system has to be as equivalent as the real hardware upon which the software will execute. This leaves out co-simulation using execution of NCS (refer to section 3.3) because it does not use the same instruction set, and consequently the correct clock-cycles needed for each instruction, as for the target processor. This means that the ISS (refer to section 3.4) will be used for our purposes.

One of the major strengths when co-simulating is that timing information can easily be retrieved from the hardware simulator. As software execution proceeds, the system clock (i.e. which drives the processor) propagates in time with the exact amounts needed for instruction execution, accessing memory and peripherals, and others. This could be utilised for a number of real-time applications.

4.1 Software execution timing

As a contrary method to static computation of software execution time (SET), timing information is determined dynamically on the fly. Static computation of SET is done by counting the number of clock cycles needed for a piece of code without executing it. This requires knowledge of cycle duration for each assembly instruction. For a piece of code in a high-level language this also needs compilation to assembly instructions for the appropriate processor. The dynamic approach is to measure the time elapsed (or count the clock cycles) for the examined piece of code, simply by reading the time/clock in the hardware simulation window. This measurement also includes duration of instruction and data fetches.

4.2 Timing access of memory and peripherals

Timing duration for accesses to memory is more of an automatic matter. If the information is to be used for computing SET it is already included in the dynamic measurement (previous section). Duration of accesses to memory are measured using the hardware simulator’s facilities for reading time or counting clock cycles. The same goes with timing of accesses to ASICs and other peripherals. This feature can be convenient for determining duration of a service call to an ASIC. Often in a service-call, there are complex interactions between the software code and the hardware involved. Verification of these interaction can be a complex matter, and often there is need for advanced and expensive instrumentation. Former methods uses monitors (debuggers) for examination of a processor’s internal registers, and logic analysers and/or logic disassemblers (with high sampling frequency capabilities and enough memory to hold a complete service-call) to examine bus-communication with hardware.

4.3 Interrupt and context-switching latency

Latency between the event of an asserted interrupt line and execution of the corresponding interrupt service routine (ISR) has traditionally been nontrivial to measure without probing the software code with additional supporting instructions. As discussed previously, it is the propagation of the system clock that drives the software simulator. This means that software execution can be controlled by simulation of the hardware portion in a co-simulation, which in turn means that the hardware simulation is only depending on parameters such as simulation time, events, and signal triggers. To verify timing of interrupt response, the hardware simulator’s facilities can be used to stop the simulation by triggering on the signal in request and from there continue simulation until the first instruction from the ISR is fetched. By triggering on instruction fetches in the hardware simulator, this technique can be used for various applications. Verification of context-switch timing is one mode of application where this can be very useful.

4.4 Verification of multiprocessing

The ability to connect more the one ISS to one single hardware platform is being supported in the leading co-simulation tools. As a result of this, complex hardware architectures incorporating several processors can be modelled for co-simulation. This has a great deal when processor communication (e.g. task synchronisation) is to be verified. Figure 5 illustrates a model of an architecture which can be Co-simulated in practise. Software developed to run for each of the processors in the system is managed by a dedicated ISS, thus allowing simultaneous verification. In practise this could be seen as multiple simulator/debugger windows each respectively dedicated for one processor. In such complex systems using several processors

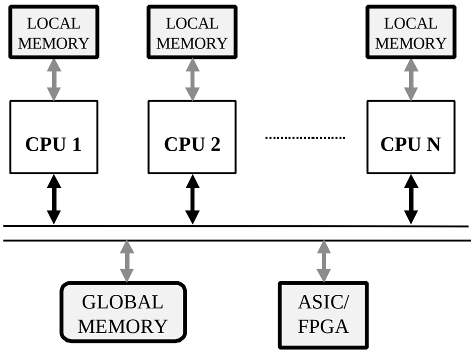

Figure 4. Multiprocessor architecture

4.5 Limitations

For almost all processor models available partial or complete internal functionality is left out for some reasons. Memory management units (MMUs) and internal cache memory are examples. Leaving out such functions could and/or should decrease resemblance with the real-time behaviour of the processor. From a real-time aspect this has a great deal, because if execution time is based on a processor whose for instance the cache memory has been left out, the cost of a cache miss could result in timing violations and missed deadlines. Other weaknesses in using a model for a real-time system, arise if there are incorrectness in the processor model, ASICs, memory models, interface logic, etc., which can not be revealed until implemented in hardware. If software is suited to work with the incorrect model there is no way to tell the real-time behaviour on the implemented hardware.

5. Conclusions

Co-simulation has increased system observability. By simulating software execution using an ISS on a cycle accurate model, timing information can easily be retrieved and be used as feedback when determining task execution flow. Co-simulation has shown to be suitable for verification of task switching, IRQ responsetime, and software access of hardware components.

Due to the slow speed of hardware simulation it is difficult to verify large applications and complete systems. Thus, co-simulation will yet not verify all possible interference's between hardware and software. In many cases it surely will reduce design time, but as a verification tool for ASIC design in large systems it will still not exclude the need of FPGAs (Field Programmable Gate Array) for fast prototyping (see [9]).

Lack of functionality like caches and MMUs in processor models (Seamless') are still a problem. A model has to be as equivalent to the real hardware as possible, especially when it comes in use in embedded real-time systems.

6. References

[1] Stefan Sjöholm and Lennart Lindh, ”The need for Co-simulation in ASIC-verification”. Euromicro Conference 1997, September 1-4, Budapest, Hungary.

[2] J. A. Stankovic and K. Ramamritham, Tutorial ”Hard Real-Time Systems”, IEEE Computer Society Press, ISBN 0-8186-0819-6

[3] Stefan Sjöholm and Lennart Lindh, ”VHDL for Designers”, ISBN 0-13-473414-9

[4] Electronic Engineer, Electronic Design Automation, September 1995

[5] Mentor Graphics, ”Seamless Co-Verification Environment, User's Reference Manual”, 1996

[6] Andreas Löfgren, Torbjörn Olsson, Stefan Sjöholm, Lennart Lindh, Internal report: ”Evaluation report on HW/SW Co-simulation using Eaglei and Seamless”. Mälardalen University, December 1996, Västerås, Sweden.

[7] Motorola, ”PowerQUICC, MPC860 User's Manual”, 1996

[8] Joakim Adomat, Johan Furunäs, Lennart Lindh, and Johan Stärner, ”Real-Time Kernel in hardware RTU: A step towards deterministic and high performance real-time systems”, The 8th IEEE Real-Time Workshop, June, 1996, L´Aquila, Italy.

[9] Lennart Lindh, Johan Stärner, Joakim Adomat, ”Experiences with VHDL and FPGAs”, VHDL-Forum, April 24-27, 1995, Nantes, France.

[10] Mentor Graphics (Microtec), ”Introduction to XRAY Pro”, 1996