Design-For-Debug in Hardware/Software Co-Design

H.P.E. Vranken[1], M.P.J. Stevens[1]
[1]Eindhoven University of Technology
Dep. of Electrical Engineering
{vranken,stevens}@eb.ele.tue.nl

M.T.M. Segers[1,2]
[2]Philips Semiconductors
Eindhoven, Netherlands
segers@sce.philips.nl

Abstract

The increasing complexity of hardware/software systems is handled effectively by hardware/software codesign methods. However, the debugging of hardware/software systems is still a very troublesome process. This is mainly due to the limited accessibility to the internals of embedded hardware/software systems. Debugging is also hindered by the nature of the design errors encountered during hardware/software debugging.

We present a structured design-for-debug strategy to address the problems of hardware/software debugging. Our design-for-debug strategy is an integral part of hardware/software codesign. Furthermore, we re-use the hardware design-for-test facilities to reduce the overhead costs of design-for-debug. Two examples are provided to illustrate our design-for-debug strategy.

1. Introduction

Hardware and software together constitute the heart of digital systems like consumer electronics, portable and personal communication systems, telecommunication systems and embedded control systems. Along with the ongoing miniaturization, more and more functions are integrated into these systems. The driving force behind this remarkable evolution is above all the progress in VLSI technology. More and more functions can be integrated into a single IC: yesterday's systems are today's chips. Furthermore, the amount of software in digital systems is increasing significantly. For instance, a high-end television set contained less than 64 Kbytes in the late 1980s, while today's models hold more than 500 Kbytes [11]. Software provides programmable and flexible systems, and exchanging hardware for software can be cost-effective.

The increasing complexity of hardware/software systems has raised the need for new hardware/software codesign methods. Many promising research initiatives on hardware/software codesign have been initiated, like [5], [8], [6], [12], and [1]. Codesign focuses on the areas of system specification, architectural design, hardware-software partitioning, and interfacing during hardware synthesis and software synthesis. Coverification techniques like formal verification and (co-)simulation are used to verify the correctness of the system design.

Formal verification and (co)-simulation are very powerful techniques. However, exhaustive simulation and complete formal verification is unfeasible for complex systems. Furthermore, simulation and formal verification are applied on abstract models of a system and the system environment. Although these conceptual models are very effective for handling complexity, they obviously cannot incorporate all real-world details. In general, we have to make a trade-off between abstraction level, simulation/verification speed, and time/data resolution. At high levels of abstraction, simulation and verification can be very fast, but time and data resolution is rather low. At lower levels of abstraction, higher data and time resolutions can be obtained, but simulation and verification is much slower.

The bottom line is that simulation and verification are not sufficient to guarantee the absence of design errors. Hence, it is required to detect the remaining design errors during testing of the prototype implementation of the system. However, the ongoing miniaturization and increasing complexity of systems cause that testing and debugging have become bottlenecks in the design process. Despite the improvements brought by codesign methods, the integration, testing and debugging of hardware/software systems is still a time-consuming, costly, and troublesome process.

The major problems encountered during testing and debugging of hardware/software systems are the nature of the errors and the limited accessibility to the system's internals. In order to improve this situation, design-for-debug should be integrated into hardware/software codesign methods. Unfortunately, structured methods for design-for-debug in hardware/software systems are still lacking. Therefore, our research aims at developing a structured approach for design-for-debug that can be integrated into hardware/software codesign.

0-8186-7895-X/97 $10.00

In this paper, first our notion of hardware/software codesign is presented in section 2. In section 3, we elaborate on the accessibility problem in hardware/software systems. In section 4 we discuss the nature of errors typically encountered during testing and debugging of hardware/software systems. Our solution to these problems is a structured design-for-debug approach, that is presented in section 5. Two small examples are given in section 6, and finally section 7 lists our conclusions.

2. Hardware/software codesign

Figure 1 shows our view on hardware/software codesign. During system specification, the functional behavior of the system is captured in a hierarchical model of communicating processes. The objective of architecture selection is to find an appropriate system architecture in which the communicating processes and communication channels are mapped onto hardware and software modules. In general, a number of alternative architectures is explored, evaluating requirements on performance, costs, flexibility, etc. The most appropriate architecture is selected. An architecture typically consists of software components stored in memories and executed on processors (microprocessors, DSPs, ASIPs), and hardware components (e.g. ASICs, FPGAs).

The next step is to refine the selected architecture, which implies adding more details to the descriptions of the hardware and software components and their interfaces. After architecture refinement, hardware synthesis and software synthesis can proceed in parallel. In hardware synthesis of custom hardware components, we proceed from behavioral-level VHDL descriptions to RTL descriptions and finally to gate-level descriptions. Subsequently, the physical hardware design is done (routing, placement, etc.) and prototypes of the hardware devices are manufactured. In software synthesis, we proceed from algorithms in C code to assembly language and finally to object code. Process scheduling and including real-time kernels are also part of software synthesis. Co-simulation of software components and hardware components is performed at various levels of abstraction during the synthesis process.

Finally, during hardware/software integration, all the hardware and software components are integrated and the final system implementation is realized. An incremental approach to system integration, testing and debugging is required. First, the individual hardware and software components are tested and debugged in isolation. Hardware testing implies both testing for manufacturing defects and testing for design errors. Software testing implies testing for design errors ('bugs'). Once a design error in hardware or software is found, the exact cause of the error must be examined so the error can be corrected. In the remainder of this paper, we will use the term 'debugging' to indicate the process of testing and isolating design errors in hardware and/or software. Note that in this definition debugging does not include testing for hardware manufacturing defects.

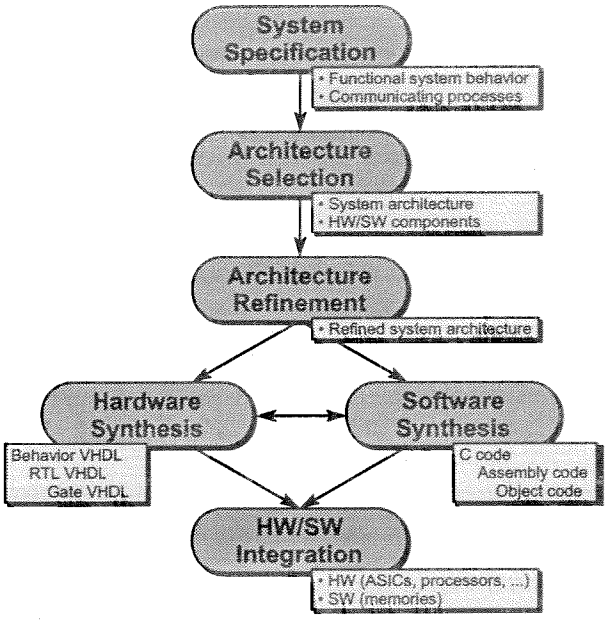

Figure 1. Hardware/software codesign

3. The accessibility problem

One of the major problems during hardware/software debugging is the accessibility problem: it is extremely difficult or even impossible to access the internals of hardware and software components once they are embedded in the system.

In the past decades, several tools and techniques have been developed to support hardware/software debugging, like in-circuit emulators, software monitors, and logic analyzers. However, the ongoing miniaturization and the integration of numerous functions on a single chip impede the use of these traditional tools. For instance, in-circuit emulators and logic analyzers cannot be used for processor cores that are embedded within ASICs or multi-chip-modules. Consequently, the time and costs spent on integration and debugging of hardware/software systems are still very large.

Therefore, a design-for-debug strategy is required to deal with the accessibility problem. Debug capabilities should be built into the hardware/software systems during codesign to improve controllability and observability.

4. Characterizing design errors

Besides the accessibility problem, debugging of hardware/software systems is also very difficult due to the nature of the design errors that are encountered during hard-

ware/software debugging. A first indication that these are complicated errors is the simple fact that these design errors were not detected in all the simulation and verification efforts during codesign.

The design errors that are often subject of hardware/-software debugging can be characterized as follows: these design errors are often related to dynamic system behavior, that is how the system evolves in real time; these errors are often transient or intermittent, which means that they may appear and disappear spontaneously; these errors often occur under exceptional conditions; and finally, these errors are often related to hardware/software interaction. Although these design errors usually form only a small fraction of all errors, they take the larger portion of the total debugging effort.

Concrete examples of the design errors that we will encounter during hardware/software debugging are faulty communication and synchronization protocols, buffer overflow, incorrect access to shared data, deadlock, starvation, incorrect process scheduling, violation of performance constraints, and unanticipated interactions.

It can be concluded that these design errors are often related to interactions between concurrent processes: they often reside in communication and synchronization protocols and in access to shared resources. During debugging, information on process interactions can be obtained by accessing the communication interfaces (hardware-hardware interfaces, hardware-software interfaces, and software-software interfaces) and the state information of processes. Hence, access to communication interfaces and process states should be considered in a design-for-debug strategy.

5. Design-for-debug

It has been shown that limited accessibility and the characteristics of design errors cause the major problems in hardware/software debugging. It is felt that a design-for-debug approach is required to improve this situation.

Design-for-debug is not an entirely new approach. In hardware, design-for-test (DFT) is well developed. Currently, DFT techniques are available and widely applied in ICs (scan paths, BIST [4]), in printed circuit boards (boundary scan architecture [2]), and in hardware systems ([3]). Although hardware DFT primarily aims at testing for manufacturing defects, scan paths and the boundary scan architecture have been used successfully for debugging purposes ([9], [7]). Modern processor chips include design-for-debug features like hardware breakpoints, software breakpoints and various debug modes (e.g. [10]). In software, design-for-debug is performed by means of software instrumentation and including software monitors.

The current design-for-debug approaches are rather ad hoc. We argued that design errors in hardware/software systems often reside in communication interfaces. When applying the current design-for-debug approaches, it may still be very difficult or impossible to access some communication interfaces. Furthermore, the current design-for-debug approaches focus on hardware or software. We feel however that design-for-debug should already be addressed in the early stages of hardware/software codesign, even before hardware/software partitioning. This approach guarantees that we can access the required communication interfaces during hardware/software debugging.

Our design-for-debug strategy consists of three stages:

First, we perform an accessibility analysis on the system specification. This analysis measures the effort for reaching communication interfaces and state information of processes from the outside of the system. In general, this analysis indicates that potential accessibility problems reside in accessing the communication interfaces and the state information of processes that are deeply embedded in the system.

To remove these potential accessibility problems, we add Points of Control and Observation (PCOs [13], [14]) in the system specification. PCOs provide direct access (observation and/or control) to embedded system resources like communication interfaces and process state information.

Second, during architecture selection and architecture refinement, we decide on how to realize the PCOs. In some cases, test equipment can be used to access communication interfaces or process states. For instance, a communication interface that is implemented as a hardware bus can be accessed using a logic analyzer. The PCO is now realized with a logic analyzer.

In other cases however, test equipment cannot be used. For instance, an internal hardware bus on a chip or communication between two software processes using a shared variable in a processor register. In these situations, we decide to realize the PCOs in hardware and/or software.

Third, during hardware synthesis and software synthesis, we decide how to implement the PCOs in hardware and/or software. In some cases, it may be possible to use existing hardware DFT features like scan paths and the boundary scan architecture. This is very attractive, because scan paths and boundary scan provide excellent observability and controllability for debugging purposes at no extra costs. However, scan paths can only be accessed by serially shifting data in and out. Furthermore, shift operations on internal scan chains are usually intrusive: the state of scan registers is affected during shifting. Hence, implementing PCOs using scan architectures is less suitable for debugging real-time applications. If hardware DFT facilities are not applicable for implementing PCOs, we may have to add dedicated hardware and/or software. The application domain puts constraints on the extra costs allowed for dedicated hardware/software PCOs.

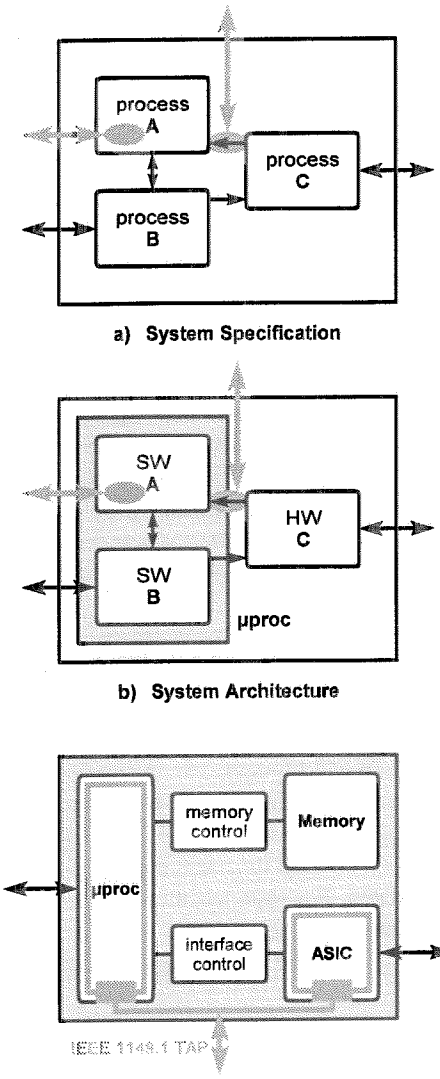

Figure 2. System-on-chip

6. Examples

In this section, two examples are provided to illustrate how our design-for-debug strategy may work in practice.

Figure 2 shows a simple system specification consisting of three communicating processes. Analysis indicates that potential accessibility problems reside in accessing the state information of process A and in accessing the communication interface between process A and process C. We insert PCOs to improve accessibility, as shown in figure 2a. In the system architecture, process A and process B are implemented as software processes running on a microprocessor, while process C is implemented in hardware, as shown in figure 2b. Because the entire system will be implemented on a single chip, we cannot use test equipment to realize the PCOs. Therefore, we decide to implement the PCOs in hardware and/or software. The final system implementation consists of a single chip containing a processor core, an ASIC core, memory control logic and interface control logic (figure 2c).

If this IC is used in consumer electronics products, then overhead costs for dedicated design-for-debug features are intolerable. Fortunately, the IC incorporates the boundary scan architecture that can be re-used to implement the PCOs. Re-using the hardware DFT facilities for debugging is very attractive. DFT features provide access for debugging purposes at no extra costs. However, as indicated before, the debugging capabilities may be limited when using hardware DFT facilities, particularly when debugging real-time applications.

Figure 3 shows design-for-debug in the design of a telecommunication system. The system specification consists of a three-layered protocol stack. For conformance testing and system debugging, it is required that the boundaries between the protocol layers can be accessed. Therefore, we insert PCOs at the protocol boundaries, as shown in figure 3a. In the system architecture, the layered structure is lost, because hardware and software components implement parts of multiple layers. We decide to add extra hardware and software to implement the PCOs at the protocol boundaries, as shown in figure 3b. The final system implementation consists of an ASIC, and software running on a micro-controller and a DSP chip. We added dedicated hardware in the ASIC and software monitors to implement the PCOs, as shown in figure 3c. The PCOs provide access to the virtual protocol boundaries in the hard ware/software implementation of the system. The design-for-debug facilities introduced overhead costs. However, these overhead costs can be justified because conformance testing and system debugging is considerably improved. In [14] and [15] an example is provided of implementing PCOs in a broadband ISDN system.

It can be concluded that the overhead costs of design-for-debug features are primarily constrained by the application domain. We listed two extremes: in the first example we used hardware DFT features only to implement PCOs cost-effectively in a consumer electronics product; in the second example, we added dedicated hardware and software to implement PCOs in a telecommunication system. A mixed approach is also possible, in which both hardware DFT facilities and dedicated hardware/software are used to implement PCOs. An interesting option is to add design-for-debug features only in prototype implementations. After system debugging is completed, the debug features are removed and not included in the shipped products. This approach can reduce the effort spent on hardware/software debugging and shorten time-to-market, without overhead costs in the final system.

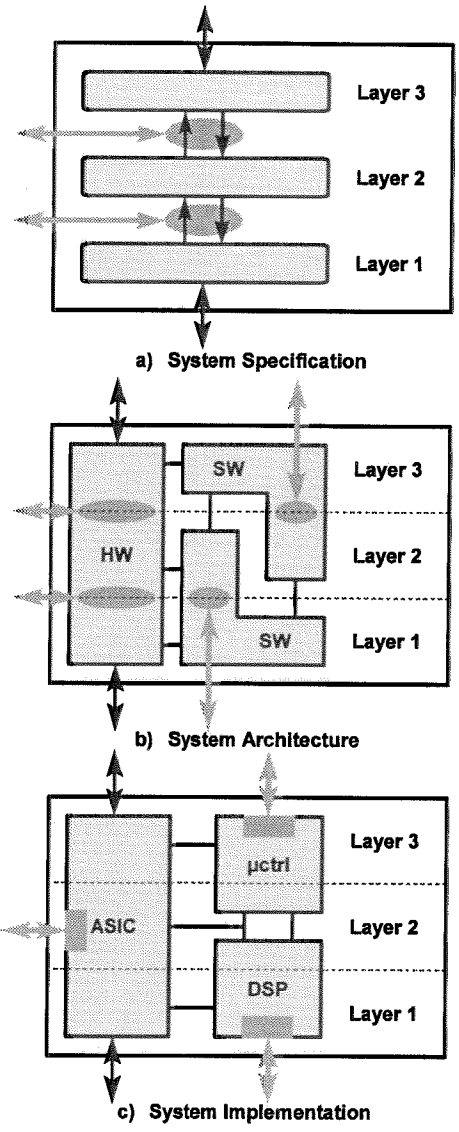

Figure 3. Telecommunication system

7. Conclusions

Hardware/software codesign methods considerably improve the design of complex hardware/software systems. Unfortunately, debugging hardware/software systems is still a very troublesome process due to limited accessibility to the internals of embedded hardware and software components. Also the nature of the design errors encountered impedes hardware/software debugging. Design-for-debug is required to deal with this.

We present a design-for-debug strategy that is an integral part of hardware/software codesign. The system specification is analyzed to detect potential accessibility problems. PCOs are inserted to improve accessibility. During architecture selection and refinement, decisions are made on how to realize the PCOs: using test equipment, or implementing PCOs in hardware/software. In the latter case, hardware DFT facilities like scan paths and boundary scan can be re-used to realize the PCOs, or dedicated hardware and/or software is required to implement the PCOs. The application domain constraints the overhead costs required to implement PCOs. We illustrated our design-for-debug strategy in two application domains: consumer electronics and telecommunication systems.

References

[1] Rapid Prototyping of Application-Specific Signal Processors (RASSP). *http://rassp.scra.org.*

[2] *IEEE 1149.1 Standard Test Access Port and Boundary Scan Architecture*. IEEE, 1990.

[3] *IEEE 1149.5 Standard Module Test and Maintenance Bus*. IEEE, 1995.

[4] M. Abramovici, M. Breuer, and A. Friedman. *Digital Systems Testing and Testable Design - Revised Printing*. IEEE Press, 1995.

[5] T. Ben Ismail and A. Jerraya. Synthesis steps and design models for codesign. *IEEE Computer*, 28(2):44–52, February 1995.

[6] R. Camposano and J. Wilberg. Embedded system design. *Design Automation for Embedded Systems*, 1(1-2):5–50, January 1996.

[7] P. Fleming and D. McClean. Scan-based design verification - an alternative approach. *ATE and Instrumentation Conference West*, 1990.

[8] D. Gajski and F. Vahid. Specification and design of embedded hardware-software systems. *IEEE Design & Test of Computers*, 12(1):53–67, Spring 1995.

[9] A. Halliday, G. Young, and A. Crouch. Prototype testing simplified by scannable buffers and latches. *IEEE Proceedings International Test Conference*, pages 174–181, 1989.

[10] H. Hao and R. Avra. Structured design-for-debug - the superSPARC II methodology and implementation. *IEEE Proceedings International Test Conference*, pages 175–183, 1995.

[11] J. Rooijmans, H. Aerts, and M. van Genuchten. Software quality in consumer electronics products. *IEEE Software*, 13(1):55–64, January 1996.

[12] D. Verkest, K. van Rompaey, I. Bolsens, and H. de Man. Coware - a design environment for heterogeneous hardware/software systems. *Design Automation for Embedded Systems*, 1(4):357–386, October 1996.

[13] H. Vranken, M. Stevens, M. Segers, and J. van Rhee. System-level testability of hardware/software systems. *IEEE Proceedings International Test Conference*, pages 134–142, 1994.

[14] H. Vranken, M. Witteman, and R. van Wuijtswinkel. Design for testability in hardware-software systems. *IEEE Design & Test of Computers*, 13(3):79–87, Fall 1996.

[15] M. Witteman and R. van Wuijtswinkel. ATM broadband testing using the ferry principle. *Protocol Test Systems VI*, pages 125–138, 1994.