Specification and Design of Embedded Systems

R. Dömer, D. Gajski, J. Zhu

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA

Abstract

With the rising complexity of digital designs and the deep sub-micron era right ahead, the specification and the design of embedded systems has to move to higher levels of abstraction. Co-design, the design of systems involving both hardware and software parts, consists of a set of refinement tasks that map an abstract specification of the design onto the intended system architecture. This article describes a generic co-design methodology including specification of the design at a high level of abstraction and step-wise refinement.

1 Introduction

The continuing decrease in geometry size and increase in chip density raises the complexity of digital systems tremendously. Deep sub-micron design, dealing with process technologies of 0.25μm and below, leads to millions of gates on a chip. This makes it possible to integrate a complete complex system on a single chip. System-on-a-chip is desirable especially for multi-media applications and portable devices where embedded systems save space, power and costs.

However, the problem is that such large complexities are beyond the size that established electronic design automation (EDA) tools (e.g. high-level synthesis) can handle. Even more, time-to-market requirements are still shrinking, so an automated design flow with efficient tools is necessary.

One solution to handle such complexities is to move to higher levels of abstraction. This essentially means reducing the number of objects that have to be considered in a particular design task. Obviously, this can be done by introducing a new level of hierarchy, which is grouping of objects. This is a well-known technique in computer aided design (CAD). For example at the logic level transistors are grouped to logic gates, at the register transfer level (RTL) gates are grouped to build registers, ALUs and other RTL components.

At the system level the components used are off-the-shelf or application specific processors, memories, busses, and application specific integrated circuits (ASICs). Furthermore, the integration of intellectual property (IP) and reuse of formerly designed circuits as core cells becomes an important issue.

System design simultaneously involves hardware design as well as software design and therefore is called co-design. Co-design usually starts from a formal, abstract specification of the intended design and performs a sequence of refinement tasks which eventually map the initial specification onto a target architecture.

The following section presents a generic co-design methodology including design modeling, step-wise refinement, simulation and verification. The essential co-design tasks, namely allocation, partitioning, scheduling, and communication synthesis, are discussed in detail.

2 A Generic Co-Design Methodology

This section presents a methodology that converts a design specification into an architecture leading to manufacturing by use of standard methods and CAD tools.

As shown in Fig. 1, co-design starts from a high-level specification which describes the functionality as well as the performance, power, cost, and other constraints of the intended design. During the co-design process, the designer will go through a series of well-defined design steps that include allocation, partitioning, scheduling and communication synthesis, which form the synthesis flow of the methodology.

The result of the synthesis flow will then be fed into the backend tools, shown in the lower part of Fig. 1. Here, a compiler is used to implement the functionality mapped to processors (software synthesis), a high-level synthesizer is used to implement the functionality mapped to ASICs (hardware synthesis), and an interface synthesizer is used to implement the functionality of required interfaces.

During each design step, the design model will be statically analyzed in order to estimate certain quality

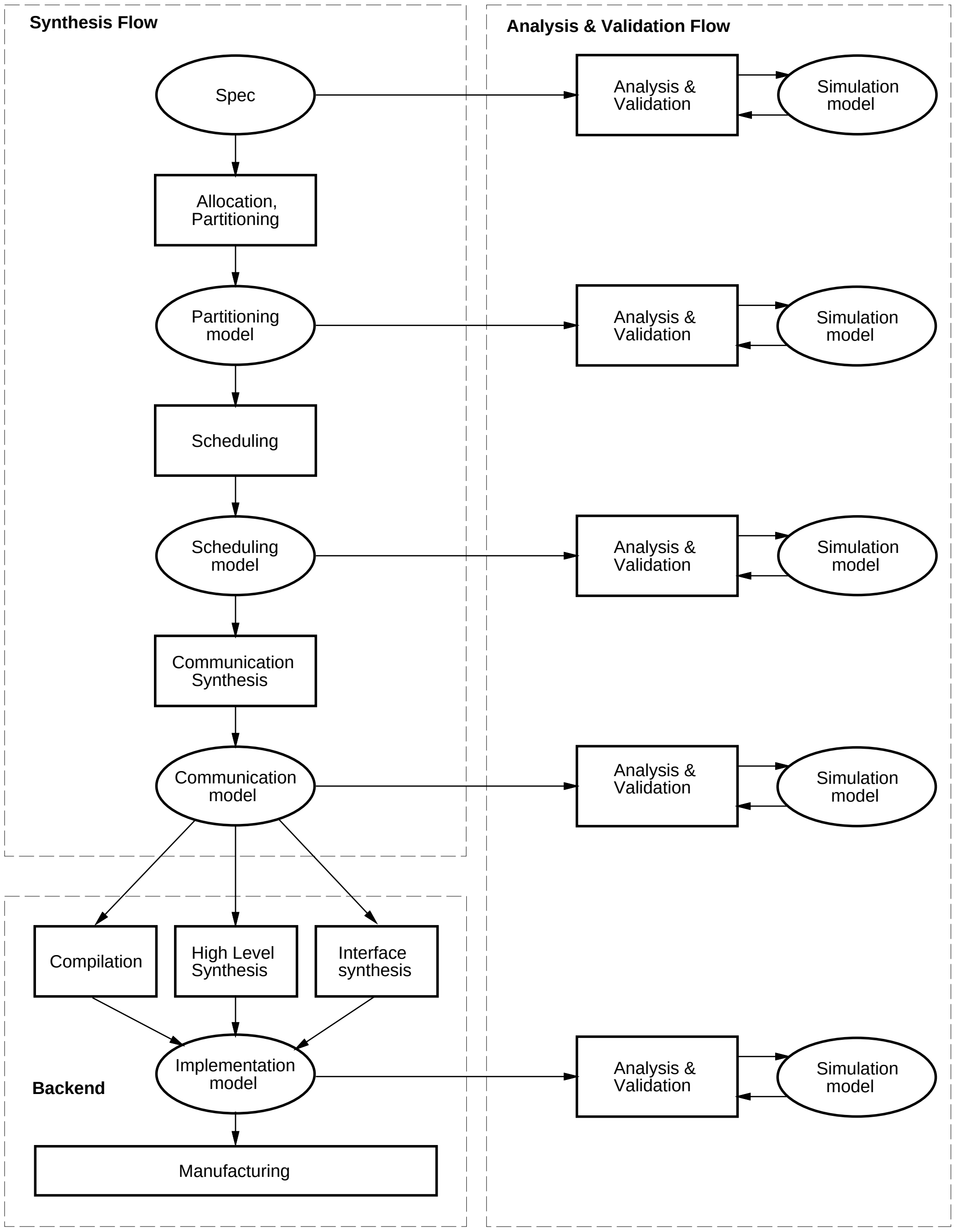

Figure 1: A generic co-design methodology.

metrics and to check whether these satisfy the constraints. This design model will also be used to generate a simulation model, which is used to dynamically validate the functional correctness of the design. In case the validation fails, a debugger can be used to locate and fix the errors. Simulation is also used to collect profiling information which in turn will improve the accuracy of the quality metrics estimation. This set of tasks forms the analysis and validation flow of the methodology as shown on the right part of Fig. 1.

It should be noted that for this methodology it is desirable that all the design models are captured in the same language [7]. Only in this case the analysis and validation flow can use the same tools at each stage. Also, all the tools can share the same design representation, possibly even the same data structures. Furthermore, the designer can compare the models at different design stages easily.

2.1 System specification

The system specification should describe the functionality of the system without specifying the implementation. It should also be executable to make it dynamically verifiable.

Although the language for specifying and modeling a system is an important issue, we use in this article only a graphical representation that focusses on the hierarchy, concurrency and transitions between behaviors of the design. For more information about languages supporting these and other concepts please refer to [2, 5, 7].

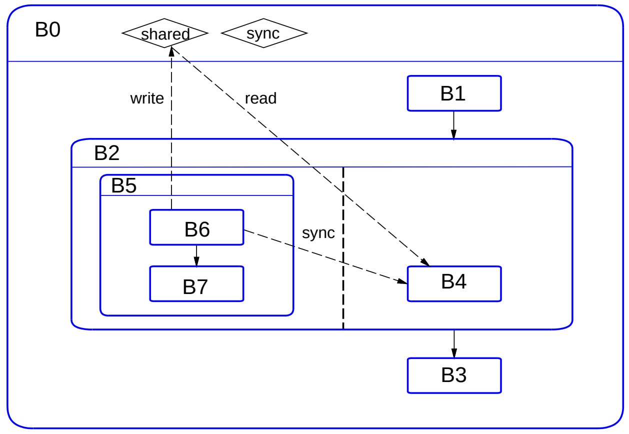

Figure 2: System specification

Fig. 2 shows an example where the system itself is specified as the top behavior B0, which contains an integer variable *shared* and a boolean variable *sync*. Behavior B0 contains three child behaviors, B1, B2, B3, with sequential ordering. While B1 and B3 are atomic behaviors specified by a sequence of imperative statements, B2 is a composite behavior consisting of two concurrent behaviors B4 and B5. B5 in turn consists of B6 and B7 in sequential order. While most of the actual behavior of an atomic behavior is omitted in the figure for space reasons, we do show a producer-consumer example relevant for later discussion: B6 computes a value for variable *shared*, and B4 consumes this value by reading *shared*. Since B6 and B4 are executed in parallel, they synchronize with the variable *sync* using signal/wait primitives to make sure that B4 accesses *shared* only after B6 has produced the value for it.

2.2 Allocation and Partitioning

Given a library of system components such as processors, memories and custom IP modules, the task of allocation is defined as the selection of the type and number of these components, as well as the determination of their interconnection, in such a way that the functionality of the system can be implemented, the constraints are satisfied, and the objective cost function is minimized. Allocation is usually done manually by designers and is the starting point of design exploration.

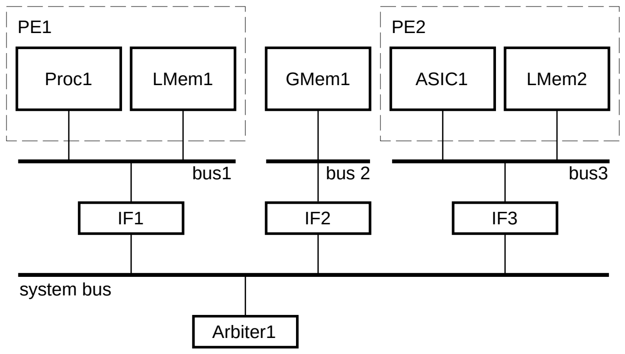

Figure 3: Target architecture model

The result of the allocation task can be a customization of a generic target architecture. Fig. 3 shows an example of an architecture that consists of two processing elements and one global memory. Both processing elements have a local memory connected to their local busses and can access the global memory via interfaces and system busses.

Partitioning defines the mapping between the set of behaviors in the specification and the set of allocated components in the selected architecture. The quality of such a mapping is determined by how well the result can meet the design constraints and minimize the design costs.

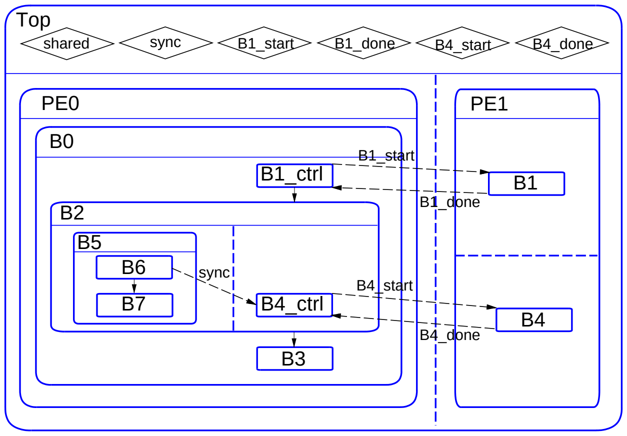

Figure 4: System model after partitioning

The system model after allocation and partitioning must reflect the partitioning decision and must be complete in order to allow validation. As shown in Fig. 4 an additional level of hierarchy is inserted in the design model which describes the selected partitioning into two processing elements (PE), PE0 and PE1. Additional controlling behaviors are also inserted whenever child behaviors are assigned to different PEs than their parents. For example, in Fig. 4, behavior B1_ctrl and B4_ctrl are inserted in order to control the execution of B1 and B4, respectively. Furthermore, in order to maintain the functional equivalence between the partitioned model and the original specification, synchronization operations between PEs must be inserted. In Fig. 4 synchronization variables , *B1_start, B1_done, B4_start, B4_done* are added so that the execution of B1 and B4, which are assigned to PE1, can be controlled by their controlling behaviors B1_ctrl and B4_ctrl through inter-PE synchronization.

However, the model after partitioning is still far from implementation for two reasons: there are concurrent behaviors in each PE that have to be serialized; and different PEs communicate through global variables which have to be localized.

2.3 Scheduling

Given a set of behaviors and optionally a set of performance constraints, the scheduling task determines a total order in invocation time of the behaviors running on the same PE, while respecting the partial order imposed by dependencies in the functionality as well as minimizing the synchronization overhead between the PEs and context switching overhead within the PEs.

Depending upon how much information on the partial order of the behaviors is available at compile time, several different scheduling strategies can be used. At one extreme, where ordering information is unknown until runtime, the system implementation often relies on the dynamic scheduler of an underlying runtime system. In this case, the model after scheduling is not much different from the model after partitioning, except that a runtime system is added to perform the scheduling. In general this strategy suffers from context switching overhead when a running task is blocked and a new task is scheduled.

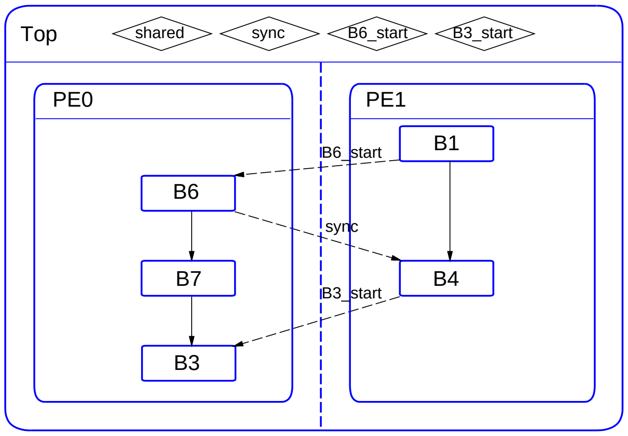

Figure 5: System model after scheduling

On the other extreme, if the partial order is completely known at compile time, a static scheduling strategy can be taken, provided a good estimation on the execution time of each behavior can be obtained. This strategy eliminates the context switching overhead completely, but may suffer from inter-PE synchronization especially in the case of inaccurate performance estimation. On the other hand, the strategy based on dynamic scheduling does not have this problem because whenever a behavior is blocked for inter-PE synchronization, the scheduler will select another one to execute. Therefore the selection of the scheduling strategy should be based on the trade-off between context switching overhead and CPU utilization.

The model generated after static scheduling will remove the concurrency among behaviors inside the same PE. As shown in Fig. 5, all child behaviors in PE0 are now sequentially ordered. In order to maintain the partial order across the PEs, synchronization between them must be inserted. For example, B6 is synchronized by *B6_start*, which will be asserted by B1 when it finishes. Note that B1_ctrl and B4_ctrl in Fig. 4 now are eliminated by the optimization carried out by static scheduling.

It should be mentioned that here we define the tasks, rather than the algorithms of co-design. Good algorithms are free to combine several tasks together. For example, an algorithm can perform the partitioning and static scheduling at the same time, in which case intermediate results, such as B1_ctrl and B4_ctrl, are not generated at all.

2.4 Communication Synthesis

Up to this stage, the communication and synchronization between concurrent behaviors are accomplished through shared variable accesses. The task of communication synthesis is to resolve the shared variable accesses into an appropriate inter-PE communication scheme at implementation level. If the shared variable is a shared memory the communication synthesizer will determine the location of such variables and change all accesses to the shared variables in the model into statements that read or write to the corresponding addresses. If the variable is in the local memory of one particular PE all accesses to this shared variable in the models of other PEs have to be changed into function calls to message passing primitives such as send and receive. In both cases the synthesizer also has to insert interfaces for the PEs and shared memories to adapt to different protocols on the buses.

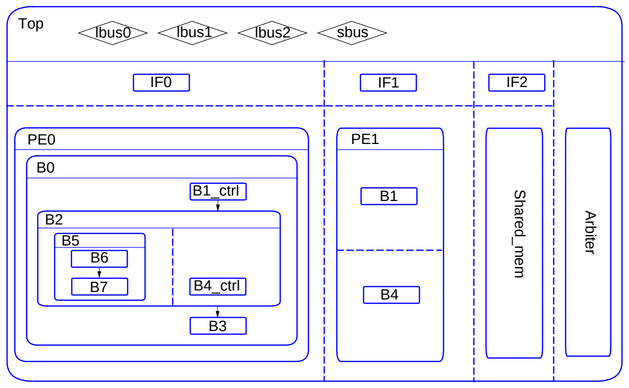

Figure 6: System model after communication synthesis

The generated model after communication synthesis, as shown in Fig. 6, differs from previous models in several aspects. New behaviors for interfaces, shared memories and arbiters are inserted at the highest level of the hierarchy. In Fig. 6 the added behaviors are *IF0, IF1, IF2, Shared_mem*, and *Arbiter*.

The shared variables from the previous model are all resolved. They either exist in shared memory or in local memory of one or more PEs. The communication channels of different PEs now become local or system buses. In Fig. 6, we have chosen to put all the global variables in *Shared_mem*, and hence all the global declarations in the top behavior are moved to the behavior *Shared_mem*. New global variables in the top behavior are the buses *lbus0, lbus1, lbus2, sbus*.

If necessary, a communication layer is inserted into the runtime system of each PE. The communication layer is composed of a set of inter-PE communication primitives in the form of driver routines or interrupt service routines, each of which contain a stream of I/O instructions, which in turn talk to the corresponding interfaces. The accesses to the shared variables in the previous model are transformed into function calls to these communication primitives. For the simple case of Fig. 6 the communication synthesizer will determine the addresses for all variables in *Shared_mem* and transforms all accesses to these variables appropriately. Direct accesses to the variables are exchanged with reading and writing to the corresponding addresses.

2.5 Analysis and validation flow

Before each design refinement, the input design model must be functionally validated through simulation or formal verification. It also needs to be analyzed, either statically, or dynamically with the help of the simulator or estimator, in order to obtain an estimation of the quality metrics, which will be evaluated by the synthesizer to make good design decisions. This motivates the set of tools to be used in the analysis and validation flow of the methodology in Fig. 1. Such a tool set typically includes a static analyzer, a simulator, a debugger, a profiler and a visualizer.

The static analyzer associates each behavior with quality metrics such as program size and program performance in case it is to be implemented as software, or metrics of hardware area and hardware performance if it is to be implemented as an ASIC. To achieve a fast estimation with satisfactory accuracy, the analyzer relies on probabilistic techniques and the knowledge of backend tools such as a compiler and high-level synthesizer.

The simulator serves the dual purpose of functional validation and dynamic analysis. The simulation model runs on a simulation engine, which in the form of a runtime library provides an implementation for the simulation tasks such as simulation time advance and synchronization among concurrent behaviors.

Simulation can be performed at different levels of accuracy, such as functional, cycle-based, and discrete-event simulation. A functionally accurate simulation compiles and executes the design model directly on a host machine without paying special attention to simulation time. A clock-cycle-accurate simulation executes the design model in a clock-by-clock fashion. A discrete-event simulation incorporates a even more sophisticated timing model of the components, such as gate delay. Obviously there is a trade-off between simulation accuracy and simulator execution time.

It should be noted that, while most design methodologies adopt a fixed accuracy simulation at each design stage, applying a mixed accuracy model is also possible. For example, consider a behavior representing a piece of software that performs some computation and then sends the result to an ASIC. While the part of the software which communicates with the ASIC needs to be simulated at cycle level so that tricky timing problems become visible, it is not necessary to simulate the computation part with the same accuracy.

A debugger renders the simulation with break point and single step ability. This makes it possible to examine the state of a behavior dynamically. A visualizer can graphically display the hierarchy tree of the design model as well as make dynamic data visible in different views and keep them synchronized at all times. All these efforts are invaluable in quickly locating and fixing design errors.

A profiler is a good complement of a static analyzer for obtaining dynamic information such as branching probability. Traditionally, this is achieved by instrumenting the design description, for example, by inserting a counter at every conditional branch to keep track of the number of branch executions.

2.6 Backend

At the stage of the backend, as shown in the lower part of Fig. 1, the leaf behaviors of the design model will be fed into different tools in order to obtain their implementations. If the behavior is assigned to a standard processor, it will be fed into a compiler for this processor which translates the design description into machine code for the target processor. If the behavior is to be mapped on an ASIC, it will be synthesized by a high-level synthesis tool which translates the behavioral design model into a netlist of RTL components. If the behavior is an interface, it will be fed into an interface synthesis tool.

As shown in Fig. 3, an interface is defined as a special type of ASIC which links its associated PE (via the local bus) with other components of the system (via the system bus). Such an interface implements the behavior of a communication channel. An example of such an interface translates a read cycle on a processor bus into a read cycle on the system bus. The communication tasks between different PEs are implemented jointly by the driver routines and interrupt service routines implemented in software and the interface circuitry implemented in hardware. While partitioning the communication task into software and hardware, and model generation for the two parts is the job of communication synthesis, the task of generating an RTL design from the interface model is the job of interface synthesis. The synthesized interface must harmonize the hardware protocols of the communicating components.

3 Conclusion

Codesign represents the methodology for specification and design of systems that include hardware and software components. A codesign methodology consists of design tasks for refining the design and the models representing the results of these refinements. In this article a co-design methodology was presented starting from a design specification at a high level of abstraction which is then stepwise refined using allocation, partitioning, scheduling, and communication synthesis.

4 References

[1] F. Balarin, M. Chiodo, A. Jurecska, H. Hsieh, A. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, B. Tabbara *Hardware-Software Co-Design of Embedded Systems: A Polis Approach.* Kluwer Academic Publishers, 1997.

[2] D. Gajski, F. Vahid, S. Narayan, J. Gong, *Specification and Design of Embedded Systems*, New Jersey, Prentice Hall, 1994.

[3] C. Liem. *Retargetable Compilers for Embedded Core Processors: Methods and Experiences in Industrial Applications.* Kluwer Academic Publishers, 1997.

[4] P. Marwedel, G. Goosens. *Code Generation for Embedded Processors.* Kluwer Academic Publishers, 1995.

[5] J. Staunstrup, W. Wolf, et al. *Hardware/Software Co-Design: Principles and Practice.* Kluwer Academic Publishers, 1997.

[6] T. Y. Yen, W. Wolf. *Hardware-software Co-synthesis of Distributed Embedded Systems.* Kluwer Academic Publishers, 1997.

[7] J. Zhu, R. Dömer, D. Gajski. *Syntax and Semantics of the SpecC Language.* Proceedings of the Synthesis and System Integration of Mixed Technologies 1997, Osaka, Japan, December 1997.