Specification and Design of Embedded Hardware-Software Systems

DANIEL D. GAJSKI
University of California, Irvine
FRANK VAHID
University of California, Riverside

Embedded-system specification and design consists of describing a system's desired functionality and mapping that functionality for implementation by a set of system components such as processors, ASICs, memories, and buses. This tutorial discusses the key problems of system specification and design, including specification capture, design exploration, hierarchical modeling, software and hardware synthesis, and cosimulation. The authors highlight existing tools and methods for solving those problems and describe a "specify-explore-refine" methodology for meeting today's embedded-system product development requirements.

EMBEDDED SYSTEMS HAVE become commonplace in recent years. Examples include automobile cruise control, fuel injection systems, aircraft autopilots, telecommunication products, interactive television processors, network switches, video focusing units, robot controllers, and numerous medical devices. An embedded system's functionality is usually fixed and is primarily determined by the system's interactions with its environment. Embedded systems also usually have numerous modes of operation, must respond rapidly to exceptions, and possess a great deal of concurrency. Thus, designing a complex embedded system poses a difficult problem for designers.

As an illustration of the embedded-system design task, consider the design of an interactive TV processor (ITVP) system for support of interactive multimedia. The system stores video frames and displays them as still pictures with accompanying text and audio. The system resides in a set-top box similar to a cable-TV box, and a user interacts by selecting menu items with a keypad. Figure 1 (next page) shows a diagram of the overall system.

Designing the digital subsystem involves creating a specification of the subsystem's functionality, called a functional specification, and mapping it to a system-level architecture, as shown in Figure 2. The subsystem consists of six components: three memories, two ASICs (application-specific ICs), and a processor. Memory1 stores two arrays that hold audio bytes; Memory2 stores a video array. Memory3 stores a font array and an array of characters to be displayed on the screen. ASIC1 implements functions that store incoming audio and generate audio on demand. ASIC2 implements functions that store and generate video frames and store special command bytes encoded in the audio-video (AV) input. Finally, the processor component implements functions that process special AV commands, main computer commands, and user commands, and that overlay characters on the screen.

0740-7475/95/$04.00 © 1995 IEEE

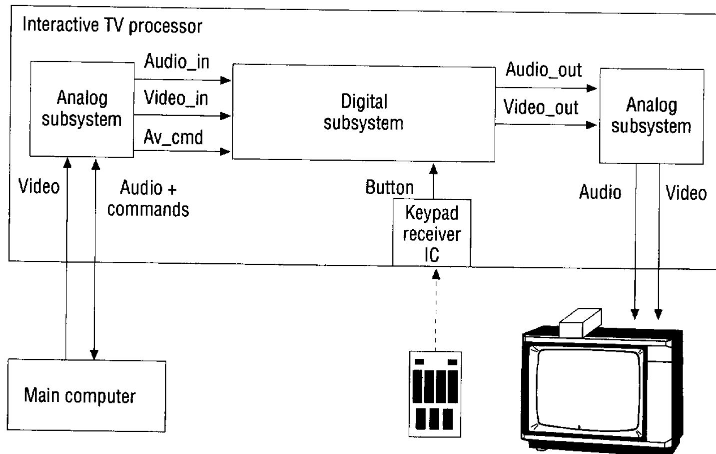


Figure 1. ITVP environment.

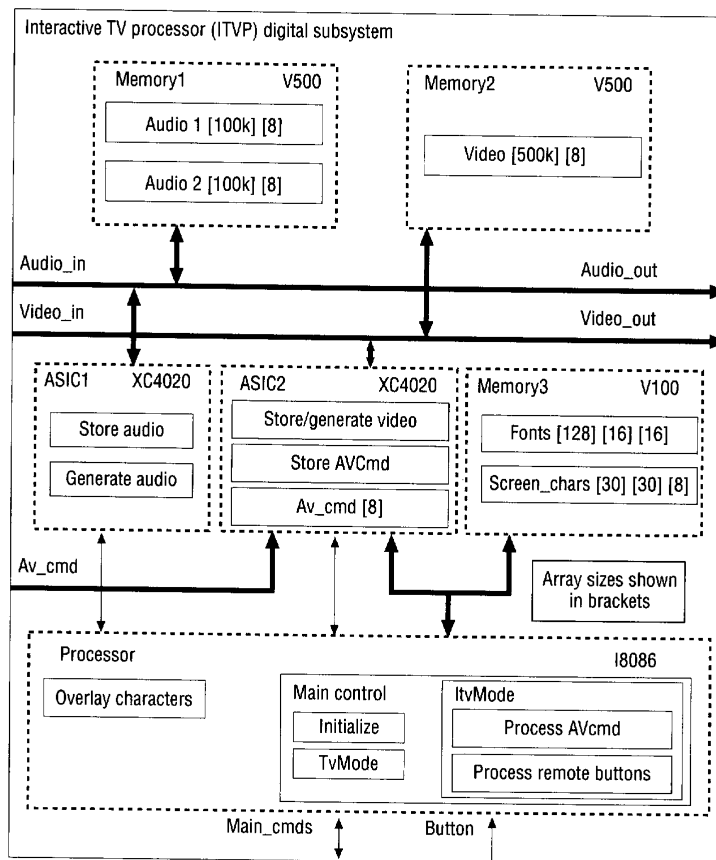


Figure 2. ITVP system-level design option.

Today's embedded-system designer has little assistance in performing system design tasks. No widely accepted methodology or tool is available to help the designer create a functional specification and map it to a system-level architecture. Most system designers work in an ad-hoc manner, relying heavily on informal and manual techniques and exploring only a handful of possibilities. A hierarchical modeling methodology can improve the situation. In such a methodology, we first precisely specify the system's functionality, explore numerous system-level implementations with the aid of tools, and automatically generate a refined description, which represents any implementation decisions.

More specifically, the following tasks, illustrated in Figure 3, are necessary to create a system-level design:

1. *Specification capture:* To specify the desired system functionality, we decompose the functionality into pieces by creating a conceptual model of the system. We generate a description of this model in a language. We validate this description by simulation or verification techniques. The result of specification capture is a functional specification, which lacks any implementation detail.
2. *Exploration:* We explore numerous design alternatives to find one that best satisfies our constraints. To do this, we transform the initial description into one more suitable for implementation. We allocate a set of system components and specify their physical and performance constraints, as in the example in Figure 2, where we allocated three memories, two ASICs, one processor, and several buses. We partition the functional specification among allocated components. For guidance in these exploration subproblems, we estimate each alternative design's quality.

3. *Specification refinement:* We refine the initial specification into a new description that reflects the decisions we have made during exploration. To do this, we move each variable into a memory, insert interface protocols between components, and add arbiters to linearize concurrent accesses to a single resource. Then we generate a system description detailing the system's processors, memories, and buses. We use cosimulation to verify that this refined description is equivalent to the initial specification. The result of specification refinement is a system-level description that possesses some implementation details of the system-level architecture we have developed, but otherwise is still largely functional.
4. *Software and hardware design:* We create an implementation for each component, using software and hardware design techniques. A standard processor component requires software synthesis, which determines a software execution order satisfying resource and performance constraints. We can obtain an ASIC's design through high-level (behavioral) synthesis,[1,2] which converts the behavioral description into a structure of components from a register-transfer (RT) library containing microarchitectural components such as ALUs, registers, counters, register files, and memories. The control logic and some RT components are synthesized with finite-state-machine and logic synthesis techniques.[3,4] The result of software and hardware design is an RT-level description, which contains optimized C code for software and RT-level netlists for custom components.
5. *Physical design:* This step generates manufacturing data for each component. For software, this is as sim-

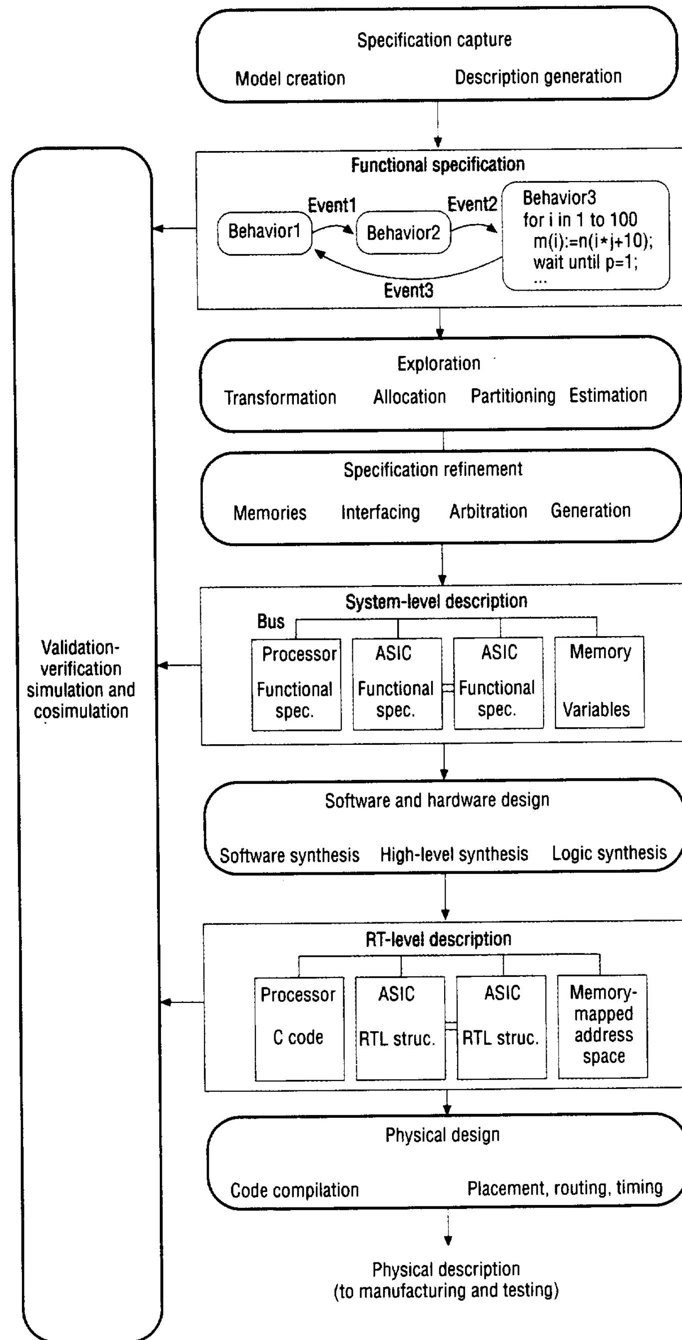


Figure 3. *System-level design process using hierarchical modeling.*

ple as compiling code into an instruction set sequence. For hardware, we convert an RT-level netlist into layout data for gate arrays, field-programmable gate arrays (FPGAs), or custom ASICs, using physical design tools for placement, routing, and timing.

These five tasks roughly define embedded-system design methodology from product conceptualization to manufacturing. After each task, we generate a more refined system description, reflecting the decisions made in that task. This hierarchical modeling methodology enables high productivity by preserving consistency through all levels and thus avoiding unnecessary iteration. Each model verifies different system properties. The functional specification verifies the completeness and correctness of system functions. The system-level description verifies system performance and communication protocols. The RT-level description verifies the developed software code and the custom design's operation during each clock cycle. The physical description verifies the system's detailed timing and electrical characteristics. Hierarchical modeling distinguishes modern, system-level methodologies from past methodologies, which captured only the physical model, late in the design cycle, making specification or architecture changes nearly impossible. (Further discussions can be found elsewhere.[5,6])

Specification capture

Specification capture unambiguously defines desired system functionality. In other words, a specification tells us what the system's response would be to any sequence of input values. Specification capture is a difficult problem because today's systems are complex, because the designer may not know a system's functionality precisely at the outset, and because specification techniques are often imprecise. To make the problem worse, the specification capture stage does not receive nearly as much attention as subsequent implementation stages, and thus many functional errors are not detected until a low-level implementation is available. Unfortunately, functional errors are far more difficult to correct at late stages of product development than during the specification stage.[7]

To remedy this situation, most researchers propose the use of a formal specification language, which allows creation of a precise specification that can be simulated, thus helping detect and correct functional errors at an early stage and reducing overall design time. Capturing a precise specification of a complex system in a formal language is a complex task. It is not a simple process of "writing down" a well-understood functionality; rather, it is the process of learning, understanding, organizing, and defining a functionality. Specification capture consists of three subtasks: model creation, description generation, and simulation. We usually must iterate these tasks several times before we obtain a complete and correct functional specification.

Model creation. To specify a system's functionality, we must first decompose that functionality and describe the relationships between the pieces. For example, we decomposed the ITVP's functionality into functions such as video storing, audio storing, video generation, and audio generation. We would express the relationships between those functions in terms of their execution order and the data passing between them. In general, a model is a formalization of allowable pieces and their relationships.

There are many models for describing a system's functionality. One is the dataflow graph,[1,8] which decomposes functionality into activities that transform data (such as a piece of a program) and the dataflow between those activities. Another model is the finite-state machine (FSM), which represents the system as a set of states and a set of arcs that indicate transition of the system from one state to another when certain events occur. Extensions of this model include hierarchy and concurrency.[9] A third model, communicating sequential processes (CSP),[10] decomposes the system into a set of concurrently executing processes, each of which executes a sequence of program instructions including variable assignments, loops, branches, and procedure calls. A fourth model, the program-state machine (PSM),[5] combines the previous two models by permitting each state of a hierarchical/concurrent FSM to contain actions described by means of program instructions. Other models include Petri nets, flowcharts, entity-relationship diagrams, Jackson diagrams, control-dataflow graphs, object-oriented models, and queuing models.

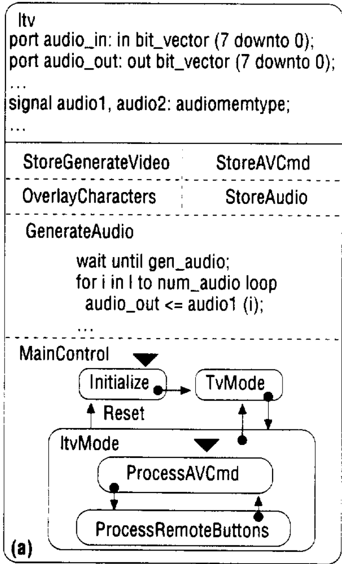
No model is ideal for all classes of systems. For example, the dataflow model may be most natural for a system that repeats the same data transformations over time on streams of data, such as a digital-signal-processing system. The FSM model may be most appropriate for a system that does not perform complex computations but must respond to complex sequences of external events, such as a control-dominated system. The CSP model is most appropriate for systems that perform complex data transformations, possibly in parallel, such as many software applications. The PSM model in many ways subsumes the FSM and CSP models, so it is appropriate not only for control-dominated systems but also for data-dominated systems such as software applications.

However, the best model is the one that most closely matches the characteristics of the system it models. For this reason, we must define the characteristics of embedded hardware-software systems. They include hierarchy, concurrency, state transitions, exceptions,

and program instructions. Figure 4a illustrates a partial PSM model for the ITVP. We decompose the ITVP into concurrent processes (six are shown). We describe the GenerateAudio process as program instructions. We describe the MainControl process as a state machine, with transitions based on certain exceptions. We further describe each state as concurrent processes, a state machine, or program instructions. Because this system description uses a combination of hierarchy, concurrency, state transitions, exceptions, and sequential instructions, the PSM model captures it most easily.

Description generation. The choice of a model is the most important influence on our ability to understand and define system functionality during specification. Once we've chosen the appropriate model, we must capture system functionality in a functional specification, using one of many different languages. A functional specification is easy to generate if there is a one-to-one correspondence between model characteristics and language constructs. If a language construct does not exist for a particular characteristic, we must try to find a set of constructs that describes that characteristic. This leads to a less readable description, possibly with more functional errors.

There are several languages that designers commonly use to specify functionality. VHDL and Verilog are popular standards that support easy description of a CSP model through their process and sequential-statement constructs. They are also commonly used to describe FSMs, although neither language possesses explicit constructs directly supporting state transitions. Esterel[11] is similar to those languages, adding constructs to support exceptions. Statecharts[9] supports description of hierarchical and concurrent FSMs, including exceptions. SpecCharts[5] supports capture of the CSP model, hierarchical/concurrent FSMs,




```
entity ItvE is port (
   audio_in : in bit_vector (7 downto 0);
   audio_out : out bit_vector ( 7 downto 0);
   ...);
end ItvE;

architecture ItvA of ItvE is
begin
behavior Itv type concurrent subbehaviors is
   type audiomemtype is array ...
   signal audio 1, audio2 : audiomemtype;
   ...
 begin
   behavior StoreGenerateVideo ...
   ...
   behavior StoreAVCmd
     ...
   behavior OverlayCharacters
     ...
   behavior StoreAudio
     ...
   behavior GenerateAudio type code is
   begin
     wait until gen_audio;
     for i in 1 to num_audio loop
        audio_out < = audio1 (i);
     ...
   behavior MainControl type
              sequential subbehaviors is
   begin
     Initialize: (TOC, true, TvMode);
     TvMode: (TOC, true, ItvMode);
     ItvMode: (TOC, true, TvMode),
              (TI, reset, Initialize);
   ...
 end ItvA;
```
(b)

```
entity ItvE is port (
   audio_in : in bit_vector (7 downto 0);
   audio_out : out bit_vector (7 downto 0);
   ...);
end ItvE;

architecture ItvA of ItvE is
begin
behavior Itv type concurrent subbehaviors is
   signal abus : audiobustype;
   signal abusreq, abusack: bit;
   signal vbus : ...
   ...
begin
   ...
   behavior Memory1 type code is
      signal audio1, audio2 : audiomemtype;
   begin
      -- code for accessing audio arrays over
         audiobus
   ...
   behavior Memory2
   ...
   behavior ASIC1 type concurrent subbehaviors is
   begin
      behavior StoreAudio ...
      behavior GenerateAudio type code is
      begin
         wait until gen_audio;
         for i in 1 to num_audio loop
            audio_out < =
              ReadMemory 1 (abus, i);
   ...
   behavior ASIC2
   ...
   behavior Memory3
   ...
   behavior Processor type concurrent
    subbehaviors is
   begin
      behaviorOverlayCharacters type code is
      ...
      behavior MainControl type
               sequential subbehaviors is
      begin
         Initialize: (TOC, true, TvMode);
         TvMode: (TOC, true, ItvMode);
         ItvMode: (TOC, true, TvMode),
                  (TI, reset, Initialize);
...
end ItvA;
```
(c)

Figure 4. *ITVP specification: PSM model* ***(a)****; model described in a language* ***(b)****; and refined ITVP specification* ***(c)****.*

Table 1. Language support of embedded-system model characteristics: Features fully supported (F), partially supported (P), not supported (N), and not applicable (N/A).

	Embedded-system features					
Language	State transitions	Behavioral hierarchy	Concurrency	Program constructs	Exceptions	Behavioral completion
VHDL	N	P	S	S	N	S
Verilog	N	S	S	S	S	S
HardwareC	N	P	S	S	N	S
CSP	N	S	S	S	N	S
Statecharts	S	S	S	N	S	N
SDL	S	P	S	N	N	S
Silage	N/A	N/A	S	N/A	N/A	N/A
Esterel	N	S	S	S	S	S
SpecCharts	S	S	S	S	S	S

and the PSM model. SDL,[12] a CCITT (International Consultative Committee for Telegraph and Telephone) standard, supports description of hierarchical dataflow diagrams with an FSM at the leaf level. Finally, Silage[13] supports easy description of dataflow models through its data stream and recurrence constructs. Table 1 summarizes several languages' abilities to capture characteristics commonly found in models of embedded systems.

Figure 4b shows the PSM model of the ITVP, captured with the SpecCharts language. Since SpecCharts was designed to capture PSM models, there is a nearly one-to-one correspondence between the model characteristics and the language constructs. A language that does not support all PSM characteristics, such as VHDL, requires greater effort and more lines of code. (Hence, the issue of graphical versus textual languages is not nearly as important as that of good model support by a language).

Exploration

Given a functional specification of a system, the designer must create a system-level design of interconnected components, each component implementing a portion of that specification. A design's acceptability depends on how well it satisfies constraints on design metrics such as performance, size, power, and cost. Evaluating a design takes substantial time and effort, so designers usually examine only a few potential designs, often those that they can evaluate quickly because of previous experience.

By using a formal specification, we can automatically explore large numbers of potential designs rapidly. Exploration involves four interdependent subproblems: allocation, partitioning, transformation, and estimation. We need not solve these problems in the given order, and we will usually need to iterate many times before we are satisfied with our system-level design.

Allocation. Allocation is the problem of finding a set of system components to implement the system's functions. Figure 2 showed an example allocation for the ITVP. The allocation provides two V500 memories and one V100, two Xilinx XC4020 ASICs, one Intel 8086 processor, and three buses.

The designer usually has hundreds of components to choose from. At one extreme are very fast but expensive custom-hardware components, such as ASICs. At the other extreme are cheaper but slower general-purpose programmable microprocessors. Between the two extremes lie innumerable components that vary in cost, performance, modifiability, power, size, reliability, and design effort. They include a variety of microprocessors, microcontrollers, FPGAs, parallel processors, and the newly evolving application-specific instruction-set processors (ASIPs).[14] In addition, hundreds of predesigned components that implement a particular function are available, such as memories, arbiters, DMA controllers, floating-point multipliers, and fast Fourier transforms. Adding to the number of choices are cores or megacells, in which processors or other predesigned components can be embedded within ASIC components.

New components surface every year. We can characterize components by instruction sets, parameterized descriptions, or number of hardware objects. General-purpose processors are characterized by instruction sets. Any part of a specification implemented with a processor must be converted to a sequence of instructions. Many special-purpose components, such as floating-point multipliers, Fourier transforms, and DMA controllers, are characterized by a parameterized function. These components execute the same program with slight variations defined by the parameters. Any part of a specification executed on

these components must be transformed to match the parameterized descriptions exactly. Finally, ASICs, FPGAs, and gate arrays are characterized by the numbers of hardware objects, such as transistors, combinational-logic blocks, or gates, that they can contain. Any part of a specification implemented on an ASIC must be converted to an interconnection of RT- and logic-level components.

The designer's job, therefore, is to choose the proper mix of components from an enormous number of possibilities. Newly developed system design tools and techniques assist in making this choice. One new approach automatically allocates processors to implement a given set of functions in a manner satisfying performance and cost constraints.[15] A recently described design environment provides rapid feedback of performance, size, and cost metrics for a given distribution of functions on any allocation of processors, ASICs, memories, and buses.[5] Researchers have developed tools that assist in mapping a specification onto a fixed allocation of one processor, one ASIC, one memory, and one bus.[16,17] Others have developed tools that assist in mapping onto a single processor with multiple ASICs.[18,19]

Partitioning. Given a functional specification and an allocation of system components, we need to partition the specification and assign each part to one of the allocated components. We can distinguish three types of specification objects, which must be partitioned separately. One type is a variable, which stores data values. Variables in the specification must be assigned to memory components. The second object type is a behavior, which transforms data values. A behavior, which may consist of programming statements such as assignment, if, and loop statements, generates a new set of values for a subset of variables. Behaviors must be assigned to custom or standard processors. The third object type is a channel, which transfers data from one behavior to another. Channels must be assigned to buses.

Specification partitioning must satisfy constraints. Constraints may exist on the number of program bytes for a processor or microcontroller, the number of gates or pins on an ASIC, the number of words in a memory, the execution time of a function, or the bit rate of an I/O port.

There are two very different approaches to system partitioning. In structural partitioning, we implement the system with fine-grained structural objects, such as gates; those objects are then partitioned among several custom components. Although easy to automate, this approach does not consider software implementations. It also does not consider intercomponent delay during implementation because the design is complete before partitioning. The other approach, functional partitioning, divides the various system functions into groups and assigns each group to a system component. Each group is then implemented as software (for a processor) or hardware (for an ASIC).

In developing a functional-partitioning technique, we must consider several issues. First, we must define object granularity, which establishes the smallest indivisible functional objects used in partitioning, such as jobs, processes, subroutines, loops, blocks of statements, statements, arithmetic-level operations, or Boolean expressions. Higher granularity means fewer objects, enabling easier interaction, faster runtime for partitioning algorithms, and faster estimations—but fewer possible partitions.

Second, we select the design metrics we will use to define a good partition. Common metrics include monetary cost, performance, communication rates, power consumption, silicon area, package size, testability, reliability, program size, data size, and memory size.

Third, we select a model by which to estimate metric values. Estimation is necessary because we can't spare the hours or days necessary to build a design for each possible partition, especially if we wish to examine hundreds or thousands of possibilities.

Fourth, we must combine multiple estimated metric values into a single cost value that defines a partition's quality by using an objective function. Because those metric values often compete with one another (that is, when one value increases, another decreases), we usually need to weigh each value in the objective function by its relative importance to the overall design. An objective function thus gives us a way to compare two partitions and to select one that satisfies constraints.

Fifth, we need partitioning algorithms to efficiently explore a subset of the huge number of possible partitions. Commonly used classes of algorithms include clustering, iterative-improvement, genetic, and custom algorithms. Some algorithms, such as clustering, are fast; others, such as genetic algorithms, are slower but often find better solutions.

A variety of techniques have evolved to assist the designer perform functional partitioning. The three main categories are hardware partitioning, hardware-software partitioning, and interactive partitioning environments. Hardware techniques aim to partition functionality among hardware modules, such as ASICs or blocks on an ASIC. Most hardware techniques partition at the granularity of arithmetic operations, differing in the partitioning algorithms they employ. For example, researchers have developed hardware techniques using clustering algorithms,[20] integer-linear programming,[21,22] manual partitioning, and iterative-improvement algorithms.[23] Other hardware-partitioning techniques operate at a higher level of granularity; one technique,[24] for example, partitions processes and subroutines among ASICs, using clustering and iterative-improvement algorithms.

Hardware-software partitioning techniques focus on partitioning function-

ality among hardware and software components. (Previous *IEEE D&T* articles provide overviews of hardware-software partitioning, including discussion of granularity and estimation.[19,25]) One technique[26] partitions at the statement level of granularity, using clustering algorithms. Other approaches partition at the statement,[16] statement sequence,[17] and subroutine levels,[27,28] using iterative-improvement algorithms.

In the third category are general environments that support interactive or automated partitioning of all three types of specification objects (variables, behaviors, and channels) among a variety of system components (such as processors, ASICs, memories, and buses). Such an environment can be used for hardware partitioning as well as for hardware-software partitioning.[5]

Figure 2 shows a partition of the main functional objects from Figure 4b among the allocated system components. Because audio must perform at very high speeds, we implement the audio storage and generation functions with an ASIC. Audio and video can be generated simultaneously, so we store audio and video data in separate memories, preventing memory access contention. The remaining functions don't require the speed of an ASIC, and thus we can implement them more cheaply on the processor.

Transformation. So far, we have assumed that the specification consists of functions that can be implemented one-to-one on system components. However, we derived those functions from a specification intended for readability, so implementing them directly may not lead to the best design. For example, we may have introduced a procedure in the specification for readability, but implementing a distinct hardware module to implement that procedure may produce a performance bottleneck. Instead, we may prefer to "in-line" the procedure into each calling process and implement it as a part of that process, resulting in better performance. In another example, we create a specification consisting of two concurrent processes, but implementing a separate controller for each process is too costly. Instead, we can merge the two processes so that they will execute serially in a single controller.

In-lining procedures and merging processes are common examples of specification transformations. A transformation reorganizes the specification, thus changing the organization of any subsequent implementation. Other transformations include flattening a hierarchy, splitting processes, grouping statements into procedures (procedure "ex-lining"), and merging variables into arrays.

A number of techniques to automate transformations have evolved. Several transformations, including procedure in-lining and process splitting, allow a designer to trade off area and performance.[29,30] A process-merging technique provides fine-grained scheduling of operations from the two processes. The technique achieves good performance and reduces hardware from two controllers to one.[31] Many design tools apply optimizing transformations with origins in software compilation to the internal representation of behavior.

Estimation. We would like to evaluate metric values, such as performance and area, for a large number of system-level designs, to find a design that best satisfies constraints. We could derive those values from the system's implementation, but that requires far too much time if we wish to examine more than a few designs. Instead, we can estimate metric values by creating a rough (thus quick) hardware and software implementation for each system component.

Accuracy and speed are competing factors in the development of an estimator. Accuracy results from a more complete implementation, while speed results from a less detailed one. For example, we could rapidly estimate hardware size by allocating and counting functional units, and then making quick statistical estimates of the number of registers, multiplexers, and controller gates. Clearly, the time saved over generating a complete implementation comes at the expense of less accuracy. In general, only rough estimates are necessary during system design. For example, to learn whether a set of functions can be implemented by a gate array with 100,000 gates, we only need an idea of whether the functions require much more or much less than 100,000 gates.

Techniques for estimating the common metrics of hardware size, software size, and performance are not completely accurate because the mapping of a behavioral description into hardware or software is not straightforward (one-to-one). The complexity is introduced by optimization at different abstraction levels. Since the estimator does not know the algorithms used in optimizing compilers, predicting the code reduction or performance enhancement resulting from optimization is difficult. Predicting the effects on performance of architectural features such as caching, pipelining, and multiple instruction issue is also difficult, since estimators do not compute code and data dynamics. Similarly, control logic optimization, library mapping in data paths, and state minimization make predicting hardware performance and size difficult.

We can estimate the hardware size of a set of functions by roughly synthesizing a controller and data path to implement the functions; applying algorithms to schedule operations into control steps; allocating functional, storage, and interconnection units; and binding data values and operations to units. We must determine the number and type of RT objects required, including registers, register files, functional units, multiplexers, buses, wires, state registers, and control logic. Unfortunately, the algorithms for doing that are computationally expen-

sive, so estimators usually generate only a subset of objects.

Once we have determined the required objects, we can estimate size for a variety of technologies. For an FPGA implementation, we would estimate the total number of combinational-logic blocks by summing the CLBs used for each object. For a gate array, we would sum the equivalent gates needed for each object. For a custom implementation, we would sum the transistors for each object, or we would compute the bounding box area after performing object placement and routing.

We can estimate the software size of a given set of functions by compiling the functions into a given processor's instruction set. Alternatively, if the appropriate compiler is unavailable, we can compile into a generic instruction set.[32,33] After tabulating the number of the given processor's instructions needed to implement each generic instruction, we can estimate software size by summing tabulated numbers for all generic instructions in the compiled generic code.

We generally are interested in estimating two types of performance metrics: function execution time and bus communication bit rates. For each type, we may be interested in minimum, maximum, or average values. These metrics can be estimated at various levels of accuracy. For coarse but quick estimates, we can use queuing models. With this approach, we (manually) associate execution time and communication frequency statistics of each function on a given system component type. Then we use queuing models to determine statistical execution times and communication rates for the overall system.

Program-level models will give somewhat more accurate performance estimates. With this approach, determining minimum and maximum performance requires analysis of the possible paths through each function in the specification—difficult for all but very simple functions. Determining average performance requires dynamic profiling, in which we simulate the specification with typical input stimuli and determine the branch probabilities. Once we have determined the possible paths or the branch probabilities, we must determine the performance for the given set of system components. For functions assigned to hardware components, we must map the functions to RT-level units and determine the minimum/maximum or average execution frequency of each control step from the paths or branch probabilities, respectively. Then, multiplying the expected number of control steps by the clock cycle produces the execution time, and the frequency of each control step gives us the communication rates.

On the other hand, to estimate performance of functions assigned to software components, we compile the functions into the instruction set of the given processor and determine each instruction's minimum/maximum or average execution frequency. Then, multiplying each instruction's expected number of executions by its execution time produces the total execution time, and the frequency of communication instructions gives us the communication rates. As in software size estimation, we can again use generic instructions and tabulation to estimate software performance when a compiler for a given processor is unavailable.

Software performance estimation for some processors requires even more effort to account for pipelining, caching, and interrupts. For a pipeline, the execution rate depends heavily on the way that instructions are paired. To obtain more accurate estimates, therefore, we might seek additional information on each instruction's execution time based on what statement follows or precedes it. For caching, each memory access may take a different amount of time, depending on whether or not the data being accessed is in the cache. We can use statistical hit/miss ratios to determine average access time, we can assume for worst-case estimates that the data is never in the cache, or we can analyze the data replacement policy in use to determine if the data will be in the cache. For interrupts, accuracy might improve if we somehow determine the frequency of interrupts and the time to service each.

Finally, software performance estimation may include the case in which multiple concurrent tasks are assigned to a single processor. In this case, we must take into account the fact that each task will be able to execute on the processor only for particular intervals of time.

Researchers have suggested a variety of estimation tools and techniques. For hardware estimation, several techniques estimate the size and performance of a group of arithmetic operations. For example, one technique obtains estimates by summing previously assigned weights associated with each operation.[23] Another technique roughly synthesizes hardware to implement the operations.[20] Another approach is to consider multiple groups of operations. A set of possible rough implementations is determined for each group, and then a global analysis picks one implementation for each group, satisfying global constraints on size and performance for all the groups.[34] Other hardware estimation techniques estimate for a group of coarse-grained functions, rather than arithmetic operations. One technique roughly synthesizes hardware for each group of functions, and uses a special data structure that permits rapid, incremental hardware modification as functions are moved between groups.[5]

Software performance estimation techniques include dynamic profiling to estimate execution time during hardware-software partitioning.[32,35] Another technique is to perform path analysis to determine minimum or maximum execution times, the latter with the help of user annotations.[36,37] Wolf provides a summary of software performance esti-

Table 2. ITVP estimates example.

Metric	Estimate	Constraint
Size (ASIC1)	8,000 gates	<10,000 gates
Size (Processor)	5,500 bytes	<4,000 bytes
Size (Memory1)	100 Kbytes	<100 Kbytes
Size (Memory2)	500 Kbytes	<500 Kbytes
Bit rate (audio_out)	10 Mbytes/s	>8 Mbytes/s

mation techniques.[38]

Table 2 lists estimated values for several design metrics for the system-level ITVP design of Figure 2. The designer (or automated algorithms) can use this information to decide how to improve the design. For example, noting that the design violates the program-memory size constraint for the processor and that 2,000 gates are available on the ASIC, the designer may try moving a function from the processor to the ASIC.

Specification refinement

After creating a specification of system functionality and exploring alternative system-level designs, we must refine the initial functional specification by incorporating the implementation style and details we have selected. We call this refined specification a system-level description because it is a mixture of structural and functional parts. Such a description consists of interconnected system components, with each component functionally specified.

Refinement is an important concept. In past approaches, designers generated only one description, close to the point at which the design was ready for manufacturing. To preserve consistent design flow, today's designers are replacing this single-description approach with hierarchical modeling, in which they derive successively more detailed descriptions from more abstract descriptions throughout the design process.

The refinement process consists of adding details about memories, interfacing, and arbitration to the system's functionality and then generating a system-level description.

Memories. During exploration, we may have grouped variables for storage in a particular memory. These variables are no longer directly accessible by each process. Now we must create a memory description, move the variable declarations to that memory description, and insert the memory access protocol into every part of the system description that accesses a variable in the memory. We may also add other details, such as specific memory addresses for each variable, to the newly created memory description.

Interfacing. Partitioning functions among system components usually introduces the need to communicate data between components. For example, a specification may include a function that reads a variable. If the function and the variable are assigned to different components, then the variable's value must be transferred over a bus. The addition of specification details that describe communication between components is called interfacing. Interfacing involves several problems: bus size generation, protocol generation, and protocol matching.

Bus size generation determines the width of the bus that will implement a group of communication channels, given a set of bit rate and bus width constraints. Although we assigned a width to each bus during allocation, now we can optimize the bus width to use as few wires as possible while still satisfying performance constraints. Two publications describe approaches to bus size generation.[5,39]

Protocol generation determines the exact mechanism for transferring data over a fixed-width bus. We must determine the type of control to be used, such as a full handshake, a half handshake, or a fixed-time access. We must also determine how to distinguish data destined for different locations, perhaps by sending an address over the data lines first or by adding address wires. Finally, we must determine how to decompose the data for serial transmission, in case the bus width is narrower than the number of data bits we wish to transfer.

Protocol matching enables communication between components when one component uses a fixed protocol. Such a case arises when we implement certain functions in software running on an off-the-shelf processor. If the other component is an ASIC, that ASIC must implement a protocol that complements the fixed protocol. If the other component uses a fixed but different protocol, we must insert between the two components hardware (a transducer) that can receive and send data with each protocol.

Several techniques address the problems of interfacing. Researchers have developed protocol-specifying techniques that extend traditional timing diagrams.[40,41] In one approach, the detailed I/O structure and protocols of library modules are hidden from the designer, who can simply interconnect those modules using high-level primitives. Interface controllers are then synthesized automatically to permit communication between modules.[42]

Arbitration. When concurrently executing processes access the same resource, such as a bus or a memory, we need to ensure that only one process accesses that resource at a given time. Arbitration resolves simultaneous access

requests by granting permission to only one process at a time. During refinement, we must insert new arbiter processes into the specification where needed.

There are two types of schemes for determining priority during arbitration. A fixed-priority scheme assigns a priority to each process statically; the priority never changes. A dynamic-priority scheme determines the priority of a process at runtime, based on the access pattern of the processes. A round-robin dynamic-priority scheme assigns lowest priority to the process most recently granted access. A first-come-first-served dynamic-priority scheme grants access to processes in the order that they requested access.

Fixed-priority schemes have simple implementations but may leave a low-priority process waiting for very long periods, even forever if higher-priority processes continuously request access. Dynamic-priority schemes have more complex implementations but ensure fair access for all processes.

Generation. After introducing the refinement details just described, we must generate a system-level description from the functional specification. In doing so, we must ensure that the new description is readable, modifiable, and modular, and that different designers can implement different parts. We must also ensure that the description is suitable for further processing by synthesis or compilation tools. Finally, we must ensure that the description is simulatable, so that we can continue to verify system functionality. Several algorithms exist for generating a system-level description after partitioning.[5]

Figure 4c shows a SpecCharts system-level description of the ITVP, reflecting the allocation and partition of Figure 2. The description now includes the memory, ASIC, and processor components, as well as declarations of buses and control signals among those components. It also describes the functionality of each component. For example, ASIC1 is to implement the StoreAudio and GenerateAudio functions. We have modified the GenerateAudio function (in Figure 4) by replacing the read of audio1 with a procedure call that executes a protocol (ReadMemory1, which reads audio1 from Memory1 over a bus).

Software synthesis

A system-level description usually possesses complex features not found in traditional programming languages such as C. A typical compiler usually cannot compile these features. Software synthesis is the task of converting a complex description into a traditional software program compilable by traditional compilers.

One complex feature of system-level descriptions is the definition of concurrent tasks. Two concurrent tasks mapped to a single processor must be scheduled to execute sequentially.[43] Such scheduling must ensure that every task has a chance to execute—in other words, that no task is "starved." Another problem is minimizing the amount of "busy-waiting," the time the processor spends waiting for some external event. A third problem is satisfying timing constraints for each task. For example, a given task may have to capture and process data arriving at a specific rate. For another example, a task may have to output data at a certain rate to ensure satisfactory system performance. Scheduling must guarantee such tasks a minimal execution rate.

Several techniques exist for performing such scheduling.[16,38,43,44] One uses a global task scheduler, which activates each task (or portion thereof) by calling each as a subroutine. This technique may require overhead to maintain the state of each task as it switches from one to the other. Another technique reduces this overhead by maintaining data locally within each task and modifying each task to relinquish control of the processor whenever it must wait for an event or an interrupt occurs. Choosing a technique usually involves a trade-off between performance and program size

Hardware synthesis

After generating a refined system-level description, we must create, or synthesize, hardware for the description parts to be implemented on custom components, such as ASICs or FPGAs. Hardware synthesis combines high-level synthesis, sequential synthesis, logic synthesis, and technology mapping.

High-level synthesis transforms a system component's functional description into a structure of RT components such as registers, multiplexers, and ALUs. This structure usually consists of two parts: a controller implementing a finite-state machine and a data path executing arithmetic operations. We refer to such a structure as a finite-state machine with data path, or FSMD.[1] The controller controls register transfers in the data path and generates signals for communication with the external world.

Several interdependent tasks make up high-level synthesis. First, we compile the input executable specification into an internal representation. The internal representation exposes control and data dependencies between arithmetic operations, such as additions and comparisons. Next, allocation selects, from an RT component database, the storage, function, and bus units to be used in the design. Third, scheduling maps operations to control steps, each usually representing one clock period or clock phase. Scheduling is necessary because all operations usually cannot execute at once due to data dependencies or due to an insufficient number of units capable of executing particular operations. Finally, binding maps scalar and array variables to registers and memories, operations to function units, and transfers to buses. A variety of algorithms, tools, and environments for high-level synthesis are de-

```
entity ItvTestE is
end ItvTestE;

architecture ItvTestA of ItvTestE is
begin
  component ItvE port (
    audio_in : in bit_vector...
    audio_out : out bit_vector...
    ... );
  end component;
  --port maps
  ...
process
begin
  -- input audio
  for i in 1 to num_audio loop
    audio_in < = audio_data (i);
    wait for atime;
  ...
  -- send audio output cmd
  gen_audio < = '1';
  ...
  -- check audio output
  for i in 1 to num_audio loop
    assert (audio_out = audio_data (i))
        report "audio sample mismatch";
    wait for atime;
  ...
...end ItvTestA;
```

Figure 5. *ITVP simulation test bench.*

scribed in the literature.[1,2]

Sequential and logic synthesis transform an FSM to a hardware structure consisting of a state register and a combinational circuit that generates the next state as well as the controller's outputs. The tasks involved in creating this structure include state minimization, state encoding, logic minimization, and technology mapping. State minimization reduces the number of states in an FSM by replacing equivalent states with a single state. Two states are equivalent if the output sequence for any input sequence does not depend on which of the two states we start in. State minimization is important because the number of states determines the size of the state register and control logic. State encoding assigns binary codes to symbolic states. The goal is to obtain a binary code that minimizes the controller's combinational logic. After encoding, logic minimization reduces the size or delay of the combinational logic. Technology mapping transforms a technology-independent logic network produced by the logic minimizer into a netlist of standard gates from a particular gate library. A variety of sequential and logic synthesis techniques are available.[3,4]

Simulation and cosimulation

Somehow we must validate that our initial specification is complete and correct. A specification is complete if it includes all possible input sequences that the environment might provide to our system. A specification is correct if it generates expected output for every such input sequence. To validate the correctness and completeness of our specification, we can apply formal verification techniques or simulation techniques.

Formal verification techniques usually involve making assertions about the specification and then proving that those assertions hold. For example, we may assert that all states of an FSM are reachable and then use a theorem prover to prove that assertion. Simulation involves executing the specification and then comparing the generated output sequence with the sequence of expected values. Today, simulation is the most common verification technique. Presently, neither formal verification nor simulation entirely validates a specification's completeness because far too many possible input sequences exist for even moderate-size systems.

As an example of simulation, Figure 5 shows a simple test case for the ITVP, written in VHDL. After instantiating an ITVP component, we input a sequence of audio data, give the ITVP a command to output that data, and then check that the output data matches the input data. If not, we generate an error message.

Simulation is useful not only to verify the initial functional specification, but also to verify the more detailed design descriptions generated throughout the design process. In particular, we must ensure that a design's functionality is consistent with the initial specification, detect possible performance bottlenecks arising from the mapping of the abstract specification to real components with limited resources, and ensure that the design satisfies detailed timing constraints for communication and synchronization.

Simulation of the more detailed descriptions may take place at various levels of abstraction. The design process in Figure 3 includes four different models of the system: The functional specification, describing only functionality without implementation, is useful in product definition, customer contracting, and marketing. The system-level description, the system bus model, is useful in performance estimation and bottleneck detection. The RT-level description gives the hardware design on the clock-cycle level and the processor model on the instruction-set level. It lets us check application software as well as the correctness of the ASIC architecture. The fourth model is the physical description, which allows checking of detailed electrical and timing properties of ASICs and standard chips.

To investigate issues during the design process, we model different parts of the system at different abstraction levels. This hierarchical modeling typically requires different simulators. Integrating the simulations of a variety of models is called cosimulation. A common example of cosimulation is the simulation of interconnected RT- or logic-level components (hardware) along with instructions running on a processor (software): hardware-software cosimulation.

Hardware-software cosimulation has two competing goals: speed and correctness. Speed is the rate at which simulated time proceeds. Because simulations are usually orders of magnitude slower than a real implementation, speed is crucial if we are to simulate a reasonable number of input sequences. Correctness refers to

generation of proper output values by the simulation. Incorrect values may arise when we simulate different parts of the system separately at different speeds and fail to ensure that the various parts access shared data in the proper order (for example, ensure that one part does not read a memory location before another part is supposed to have updated that location).

A third goal, which usually competes with speed, is interactive debugging—the ability to step through system execution, examine intermediate values, and backtrack to debug the system.

For both hardware and software, speed varies depending on the chosen verification technique. For software, the slowest but most "debug-amenable" approaches use model simulation. In one such approach, we execute the software on a model of the target processor with a hardware simulator. We can write this model on one of several abstraction levels, including instruction-set, RT, and gate levels. Higher levels provide shorter simulation time at the expense of less detailed timing accuracy. A faster simulation approach would be to execute the software on a custom-built simulator for the target processor.

Instead of simulating, we can use faster, execution approaches. With one execution approach, we execute the software on our development processor (for example, our workstation), assuming that the description is written in a high-level language such as C, which can be compiled to the processor's instruction set. Alternatively, we can simply execute the software on the target processor. A common hybrid approach is called in-circuit emulation. An emulator consists of a package with the same I/O configuration as the target processor, along with a tool on which we can run the software and interactively debug it from our workstation.

For hardware, the most common simulation technique uses an event-driven hardware simulator. An alternative is to use a hardware emulator. In cases where speed is extremely important, we can create an FPGA implementation.

We maintain correctness only if we ensure that the software and hardware simulations access shared data in the proper order. The most straightforward method is to create a hardware model of the target processor and then simulate the hardware and software in synchrony, using the same hardware simulator. The hardware simulator thus serves as a supervisor, ensuring that data is accessed in correct order.

However, to speed up the simulation, we would like to use one of the software execution options mentioned earlier. In such cases, one way to ensure correctness is to explicitly synchronize all data transfers between hardware and software, so that no supervisor is necessary. A common method for describing such transfers is message passing. In message passing, all data transfers occur when a process explicitly sends data to another process that explicitly receives it. Alternatively, if it turns out that the software is the only data transfer initiator, the executing software serves as supervisor. Conversely, when the hardware is the only data transfer initiator, the hardware simulator serves as supervisor.

Several cosimulation techniques have appeared in the literature. In one approach, software executes on the development processor and communicates with a hardware simulator through Unix interprocess communication mechanisms, using message passing.[19,45] Another approach executes the software on a processor simulator connected to a hardware simulator. Facilities exist between the simulators to support message passing between the hardware and software.[16]

Two recent articles describe three cosimulation techniques.[17,35] One, primarily for timing analysis, uses estimators to predict the performance of parts of the specification (a cosimulation estimator is also described elsewhere[5]). The second technique simulates a cycle-accurate processor model in conjunction with the hardware, using a hardware simulator. The third technique uses a prototyping board that contains a RISC (reduced instruction set computing) processor and several FPGAs.

Another article describes a multiparadigm simulation environment (Ptolemy), which supports cosimulation of different domains, such as synchronous dataflow and digital hardware, and defines mechanisms for transferring data and synchronizing timing between domains.[25] The environment thus can support a variety of hardware-software cosimulation techniques. For example, one method discussed uses a cycle-accurate functional processor model, a digital-hardware domain representation of the RT-level hardware components, and the synchronous dataflow domain for functional abstraction of some analog hardware components. Thus, the mixed hardware-software system can be simulated within an integrated environment. Another article describes a new Ptolemy domain for simulating processes that communicate via message passing.[46] This domain simulates hardware and software; the simulation can also integrate physical implementations of some processes.

AN INFORMAL DESIGN methodology, with design capture and simulation late in the design cycle, was tolerable in the past. Then design complexity was low to medium, and new generations of products were introduced only every two to three years. With increased complexity and shorter time to market, however, the old methodology is no longer acceptable.

For embedded software and/or hardware systems, a new methodology, based on a hierarchy of models at different levels of abstraction, is necessary. Using this methodology, we start with a formal functional specification and derive the next-lower level model by ex-

ploring implementation issues and refining the higher level model with implementation selections made during exploration.

This specify-explore-refine methodology can help managers and designers cope with today's product development requirements. It can lead to substantial productivity gains through the early detection of functional errors and through faster exploration of design alternatives. The proposed methodology leads to excellent documentation of the system's desired functionality as well as all design decisions, making redesign much easier. It encourages concurrent engineering because the various system components possess precise functional descriptions derived from an overall system specification, which simplifies integration as well as design changes during implementation. It enables marketing departments to rapidly predict a system's size, performance, and design time, helping determine a product's feasibility. Such rapid prediction also helps engineering managers allocate human resources to a design.

When we tested the specify-explore-refine methodology on a medium-complexity (50,000 gates) fuzzy-logic controller, we reduced the design cycle from conceptualization to manufacturing to approximately 100 hours.[47] With the standard methodology, we estimate that this design would take about six months. As tools and techniques for the new methodology improve, we believe the design cycle of high-complexity systems can drop from the present 12 to 18 months to several hundred hours. If the goal is precisely defined and the design process well understood, a design should be straightforward and easily manageable.

Acknowledgments

We thank Sanjiv Narayan of Viewlogic and Jie Gong of UC Irvine for their substantial contributions to the ideas and techniques described in this article. We also thank Jörg Henkel of the Technical University of Braunschweig, Asawaree Kalavade of UC Berkeley, and Rajesh Gupta of the University of Illinois for their helpful discussions and comments.

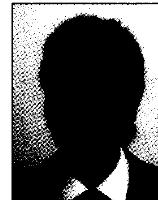
References

1. D. Gajski, N. Dutt, C. Wu, and Y. Lin, *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, Boston, 1991.
2. J. Vanhoof, K. Van Rompaey, I. Bolsens, and H. DeMan, "High-Level Synthesis for Real-Time Signal Processing," Kluwer, 1994.
3. G. DeMicheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, New York, 1994.
4. S. Devadas, G. Gosh, and K. Keutzer, *Logic Synthesis*, McGraw-Hill, New York, 1994.
5. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*, Prentice-Hall, Englewood Cliffs, N.J., 1994.
6. K. Buchenrieder, *Hardware/Software Codesign—An Annotated Bibliography*, IT Press, Hartenstein, Chicago, 1995.
7. D. Gabel, "Software Engineering," *IEEE Spectrum*, Vol. 31, No. 1, Jan. 1994, pp. 38-41.
8. W.S. Davis, *Tools and Techniques for Structured Systems Analysis and Design*, Addison-Wesley, Reading, Mass., 1983.
9. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring, "Statemate: A Working Environment for the Development of Complex Reactive Systems," *Proc. Int'l Conf. Software Engineering*, IEEE Computer Society Press, Los Alamitos, Calif., 1988, pp. 396-406.
10. C. Hoare, "Communicating Sequential Processes," *Comm. ACM*, Vol. 21, No. 8, 1978, pp. 666-677.
11. N. Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer, Boston, 1993.
12. F. Belina, D. Hogrefe, and A. Sarma, *SDL with Applications from Protocol Specifications*, Prentice-Hall, Englewood Cliffs, N.J., 1991.
13. P. Hilfinger and J. Rabaey, *Anatomy of a Silicon Compiler*, Kluwer, Boston, 1992.
14. J. Praet, G. Goossens, D. Lanneer, and H. DeMan, "Instruction Set Definition and Instruction Selection for ASIPs," *Proc. Int'l Workshop High-Level Synthesis*, 1993, Assn. of Computing Machinery, New York, pp. 11-16.
15. S. Prakash and A. Parker, "Synthesis of Application-Specific Multiprocessor Architectures," *Proc. 28th Design Automation Conf.*, IEEE CS Press, 1991, pp. 8-13.
16. R. Gupta and G. DeMicheli, "Hardware-Software Cosynthesis for Digital Systems," *IEEE Design & Test of Computers*, Vol. 10, No. 3, Oct. 1993, pp. 29-41.
17. R. Ernst, J. Henkel, and T. Benner, "Hardware-Software Cosynthesis for Microcontrollers," *IEEE Design & Test of Computers*, Vol. 10, No. 4, Dec. 1993, pp. 64-75.
18. M. Srivastava and R. Brodersen, "Rapid-Prototyping of Hardware and Software in a Unified Framework," *Proc. Int'l Conf. Computer-Aided Design*, IEEE CS Press, 1992, pp. 152-155.
19. D. Thomas, J. Adams, and H. Schmit, "A Model and Methodology for Hardware/Software Codesign," *IEEE Design & Test of Computers*, Vol. 10, No. 3, Sept. 1993, pp. 6-15.
20. M. McFarland and T. Kowalski, "Incorporating Bottom-up Design into Hardware Synthesis," *IEEE Trans. Computer-Aided Design*, Sept. 1990, pp. 938-950.
21. C. Gebotys, "An Optimization Approach to the Synthesis of Multichip Architectures," *IEEE Trans. Very Large Scale Integration Systems*, Vol. 2, No. 1, 1994, pp. 11-20.
22. Y. Chen, Y. Hsu, and C. King, "MULTIPAR: Behavioral Partition for Synthesizing Multiprocessor Architectures," *IEEE Trans. Very Large Scale Integration Systems*, Vol. 2, No. 1, Mar. 1994, pp. 21-32.
23. R. Gupta and G. DeMicheli, "Partitioning of Functional Models of Synchronous Digital Systems," *Proc. Int'l Conf. Computer-Aided Design*, IEEE CS Press, 1990, pp. 216-219.
24. F. Vahid and D. Gajski, "Specification Partitioning for System Design," *Proc. Design Automation Conf.*, ACM, 1992, pp. 219-224.
25. A. Kalavade and E. Lee, "A Hardware-Software Codesign Methodology for DSP Applications," *IEEE Design & Test of Computers*, Vol. 10, No. 3, Sept. 1993, pp. 16-28.
26. X. Xiong, E. Barros, and W. Rosentiel, "A Method for Partitioning UNITY Language in

Hardware and Software," *Proc. European Design Automation Conf. (EuroDAC)*, IEEE CS Press, 1994.

27. F. Vahid, J. Gong, and D. Gajski, "A Binary-Constraint Search Algorithm for Minimizing Hardware During Hardware-Software Partioning," *Proc. European Design Automation Conf.*, IEEE CS Press, 1994.
28. P. Eles, Z. Peng, and A. Doboli, "VHDL System-Level Specification and Partitioning in a Hardware/Software Cosynthesis Environment, *Proc. Int'l Workshop Harware/Software Codesign*, IEEE CS Press,1994, pp. 49-55.
29. R. Walker and D. Thomas, "Behavioral Transformation for Algorithmic Level IC Design," *IEEE Trans. Computer-Aided Design*, Oct. 1989.
30. T. Ismail, K. O'Brien, and A.A. Jerraya, "Interactive System-Level Paritioning with Partif," *Proc. European Design Automation Conf.*, IEEE CS Press, 1994.
31. J. Hagerman, "Synthesis of Multiple Process Digital Systems," PhD thesis, Carnegie Mellon Univ., Pittsburgh, 1994.
32. J. Gong, D. Gajski, and S. Narayan, "Software Estimation from Executable Specifications," *Proc. European Design Automation Conf. (EuroDAC)*, IEEE CS Press, 1995.
33. S. Antoniazzi, A. Balboni, W. Fornaciari, D. Sciuto, "A Methodology for Control Dominated System Design" *Proc. Int'l Workshop Hardware/Software Codesign*, 1994, pp. 2-9.
34. K. Kucukcakar and A. Parker, "CHOP: A Constraint-Driven System-Level Partitioner," *Proc. Design Automation Conf.*, ACM, 1991, pp. 514-519.
35. W. Ye, R. Ernst, T. Benner, and J. Henkel, "Fast Timing Analysis for Hardware-Software Co-Synthesis," *Proc. Int'l Conf. Computer Design*, IEEE CS Press, 1993, pp. 452-457.
36. C. Park and A. Shaw, "Experiments with a Program Timing Tool Based on Source-Level Timing Scheme," *Computer*, Vol. 24, No. 5, May 1991, pp. 48-57.
37. P. Puschner and C. Koza, "Calculating the Maximum Execution Times of Real-Time Programs," *J. Real-Time Systems*, Vol. 1, 1989, pp. 159-176.
38. W. Wolf, "Hardware-Software Co-Design of Embedded Systems," *Proc. IEEE*, Vol. 82, No. 7, 1994, pp. 967-989.
39. D. Filo, D. Ku, C. Coelho, and G. DeMicheli, "Interface Optimization for Concurrent Systems Under Timing Constraints," *IEEE Trans. Very Large Scale Integration Systems*, Vol. 1, Sept. 1993, pp. 268-281.
40. G. Borriello, "Specification and Synthesis of Interface Logic," in *High-Level VLSI Synthesis*, R. Camposano and W. Wolf, eds., Kluwer, Boston, 1991.
41. P. Moeschler, H. Amann, and F. Pellandini, "High-Level Modeling Using Extended Timing Diagrams," *Proc. European Design Automation Conf. (EuroDAC)*, IEEE CS Press, 1993.
42. J. Sun and R. Brodersen, "Design of System Interface Modules," *Proc. Int'l Conf. Computer-Aided Design*, IEEE CS Press, 1992, pp. 478-481.
43. S. Levi and A. Agrawala, *Real-Time System Design*, McGraw-Hill, New York, 1990.
44. G. Andrews and F. Schneider, "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys*, Vol. 15, Mar. 1983, pp. 3-44.
45. D. Becker, R. Singh, and S. Tell, "An Engineering Environment for Hardware/Software Cosimulation," *Proc. Design Automation Conf.*, ACM, 1992, pp. 129-134.
46. S. Lee and J. Rabaey, "A Hardware-Software Cosimulation Environment," *Proc. Int'l Workshop Hardware-Software Co-Design*, 1993.
47. L. Ramachandran, D. Gajski, S. Narayan, F. Vahid, and P. Fung, "Towards Achieving a 100-hour Design Cycle: A Test Case," *Proc. European Design Automation Conf. (EuroDAC)*, ACM, 1994.

Daniel D. Gajski is a professor in the Department of Computer Science at the University of California, Irvine. Previously, he was a professor in the Department of Computer Science at the University of Illinois in Urbana-Champaign. Still earlier, he worked in industry. His current interests are CAD environments, ASIC and system design methodology, high-level synthesis, and hardware-software codesign. Gajski served as technical chair of the High-Level Synthesis Workshop in 1992 and general chair of the High-Level Synthesis Symposium in 1994. He received the Dipl. Ing. and MS in electrical engineering from the University of Zagreb and the PhD in computer and information sciences from the University of Pennsylvania. He is a senior member of the Computer Society and a fellow of the IEEE.



Frank Vahid is an assistant professor in the Computer Science Department at the University of California, Riverside. His research interests include hardware-software codesign, system specification and design, behavioral partitioning, and behavioral synthesis. He holds a BS in computer engineering from the University of Illinois. He holds an MS and PhD in computer science from the University of California, Irvine, where he was an SRC fellow and a chancellor's fellow.

Send correspondence about this article to Daniel D. Gajski, UC Irvine, Dept. of Computer Science, Irvine, CA 92717-3425; or gajski@ics.uci.edu.