System Design Methodologies: Aiming at the 100 h Design Cycle

Daniel D. Gajski, *Fellow, IEEE,* Sanjiv Narayan, *Member, IEEE,* Loganath Ramachandran, *Member, IEEE,* Frank Vahid, *Member, IEEE,* and Peter Fung, *Member, IEEE*

Abstract— **As methodologies and tools for chip-level design mature, design effort becomes focused on increasingly higher levels of abstraction. We present a tutorial on a design methodology for chip and system design and present a test case that justifies the future goal of a 100 h design cycle.**

I. Introduction

With the increasing acceptance of automation of the lower-level design tasks, designers are increasingly focusing their efforts at the more abstract stages of the system-design process. The system functionality can best be understood during the earlier design steps, before design effort has been expended on implementation details. By focusing at these higher levels of abstraction, the designer can make a more significant impact on the quality of the end-product than by focusing at lower levels. This is the primary reason why the early phases of the design process have become so crucial in system design.

As a design progresses from concept to manufacturing, it goes through several levels of abstractions. At each abstraction level, different design views are created in different design representations. In this section, we will present common design representations and examine how they vary from one abstraction level to another.

A. Design Representations

The different representations of a design differ in the type of information they convey. The three most frequently used representations are those that emphasize the behavioral, structural and physical aspects of the product.

A **behavioral representation** treats the design simply as a black box, while specifying its behavior as a function of its input values and expired time. In other words, a behavioral representation describes the system's functionality, but tells us nothing about its implementation; it defines how the black box would respond to any combination of input values, but omits any indications about how we would design that box.

Levels	Behavioral forms	Structural components	Physical objects
Transistor	Differential eq., current–voltage diagrams	Transistors, resistors, capacitors	Analog and digital cells
Gate	Boolean equations, finite–state machines	Gates, flip–flops	Modules, units
Register	Algorithms, flowcharts, instruction sets, generalized FSM	Adders, comparators, registers, counters, register files, queues	Microchips, ASICs
Processor	Executable spec., programs	Processors, controllers, memories, ASICs	PCBs, MCMs

Fig. 1. Design representation and abstraction levels.

A **structural representation**, by contrast, begins to answer some of these questions, as it serves to define the black box in terms of a set of components and their connections. In other words, this representation focuses on specifying the product's implementation, and even though the functionality of the black box can be derived from its interconnected components, the structural representation does not describe the functionality explicitly.

A **physical representation** carries the implementation of the design one step further by specifying the physical characteristics of the components described in the structural representation. For instance, a physical representation would provide the dimensions and location of each component, as well as the physical characteristics of the connections between them. Thus, while the structural representation provides the design's connectivity, the physical representation describes the spatial relationships among these interconnected components, describing the weight, size, heat dissipation, power consumption and position of each input or output pin in the manufactured design.

In general, the process of designing a system proceeds from a behavioral representation to a structural representation and finally to a physical representation, gaining technology-specific implementation details along the way. While we need these details of the implementation in order to manufacture the product, they tend to obscure the system's functionality, thereby impeding the designer in his or her attempt to ensure that the system functions correctly. Consider, for example, a simple system which can add or multiply two 32 b numbers. The behavioral representation would simply consist of two equations: $o := a + b$ and $o := a \times b$. The structural representation, however, consists of several interconnected

Manuscript received August 16, 1994; revised December 21, 1994 and March 16, 1995. The design of the fuzzy logic controller was supported by a grant from the Matsushita Electric Works Research and Development Laboratory, San Jose, CA. The Semiconductor Research Corporation supported the Specsyn and VSS projects at the University of California, Irvine, under Grant 92-DJ-146.

D. D. Gajski is with the Department of Information and Computer Science, University of California, Irvine, CA 92717 USA.

S. Narayan is with Viewlogic Systems Inc., Marlboro, MA 01752 USA.

L. Ramachandran is with LSI Logic Corporation, Milpitas, CA 95035 USA.

F. Vahid is with the Department of Computer Science, University of California, Riverside, CA 92521 USA.

P. Fung is with the Matsushita Electric Works R&D Laboratory, Inc. San Jose, CA 95134 USA.

Publisher Item Identifier S 1063-8210(96)01868-9.

registers, arithmetic units, and multiplexers, which could make the system's functionality difficult to discern, especially if the number of components is very large or if a particular component's functionality is only partially used. Thus, if we are to focus on the increasingly crucial problem of functional correctness, then we must recognize that designers will be more successful and create better products when working with behavioral representations as opposed to working with structural or physical representations.

B. Levels of Abstraction

Each of the three types of design representations discussed in the previous section lends itself to several different levels of abstraction, or granularity. The different levels can be distinguished from each other on the basis of the types of objects they use, which fall into four categories: transistors, gates, registers and processor components. These different levels of abstraction are summarized in relation to each type of representation in Fig. 1.

At the **transistor level**, the main components are transistors, resistors and capacitors. These objects can be combined to form analog and digital circuits that satisfy a given functionality. On this level, functionality is usually described by a set of differential equations or by some form of current-voltage relationships. Finally, a physical representation of such a circuit, called a **cell**, would consist of transistor-level components and the wires connecting them. Such cells often are defined in terms of their component layouts.

On the **gate level**, the main components are combinational logic gates and flip-flops that perform Boolean operations and act as basic memory elements. These gates and flip-flops can be grouped and placed on a silicon wafer in order to form arithmetic and storage modules. These modules can be described behaviorally by logic equations and finite-state machine diagrams.

The main components on the **register level** (RT) are arithmetic and storage units (such as adders, comparators, multipliers, counters, registers, register-files, data buffers, and queues). Each of these RT components is a physical object, having fixed dimensions, a fixed propagation time, and fixed positions for its primary input and output pins on the boundary of the module. Register-level components are used in the design of microchips, which can be described by flowcharts, instruction sets, generalized finite-state machines or state tables.

Finally, the highest level of abstraction is called the **processor level** where the basic components are processors, memories, controllers, interfaces and custom microchips called application-specific integrated circuits (ASIC's). One or more of these components can be placed and soldered on a printed-circuit board (PCB), and interconnected by wires that are printed on the board surface. In order to reduce the dimensions of the board, a silicon substrate may be used to connect the microchips, instead of using the PCB's, in which case the package would be called a multichip module (MCM). The systems composed of processor-level components can be described behaviorally in several different ways, as a natural language description, as an executable specification in a hardware description language, or finally, as algorithms or programs in a programming language.

It is important to keep in mind the fact that designers can only focus their efforts on the level at which the system is comprehensible to them. This level is generally the one where the number of objects is relatively small. For example, a designer might comprehend a system consisting of 10 Boolean equations, but certainly not one of 10 000 equations. In the latter case, he would have to keep moving to higher levels of abstraction until he reaches the point where the system can be represented by a manageable number of objects, such as by a few algorithms. At the lower levels of abstraction, the system can be managed only when we divide it into small pieces and distribute it among a number of designers, or when we use automated tools that are able to manage the complexity. Fortunately, as new design tools emerge at the lower levels, designers are free to focus on the higher levels, where decisions tend to impact quality much more heavily than at lower levels.

II. Design Methodologies

In the previous section we presented the various levels of abstraction that every electronic system will go through as the design process evolves from conceptualization to manufacturing. The set of specific tasks in this design process, the particular order in which they are to be executed and a set of CAD tools to be used during the execution of each task forms a **design methodology**. The design methodologies that will be discussed in this paper are shown graphically in Fig. 2.

We will now briefly discuss two design methodologies that are predominant in industrial environments today.

A. Capture and Simulate Methodology

For the last 25 years, the majority of ASIC and system houses used a design process that was based on a **capture-and-simulate** design methodology. In this methodology, one starts with a specific set of requirements for the product, usually supplied by the marketing departments. Since these requirements do not contain any information about the implementation of the product, a small team of chief architects would then produce a rough block diagram of the chip architecture, that would serve as a preliminary, albeit incomplete, specification. In some cases, this initial block diagram would be refined further before being given to a team of logic and layout designers whose task is to convert each functional block into a logic or circuit schematic that will finally be captured by schematic capture tools, and simulated to verify its functionality, timing and fault coverage. This captured schematic can also be used to drive the physical design tools for the placement and routing of gates in gate-array technologies, or it can be used in custom technologies to map gates into standard or custom cells before placement and routing.

B. Describe and Synthesize Methodology

The acceptance of logic synthesis in the last few years as an integral part of the design process has led to an evolutionary

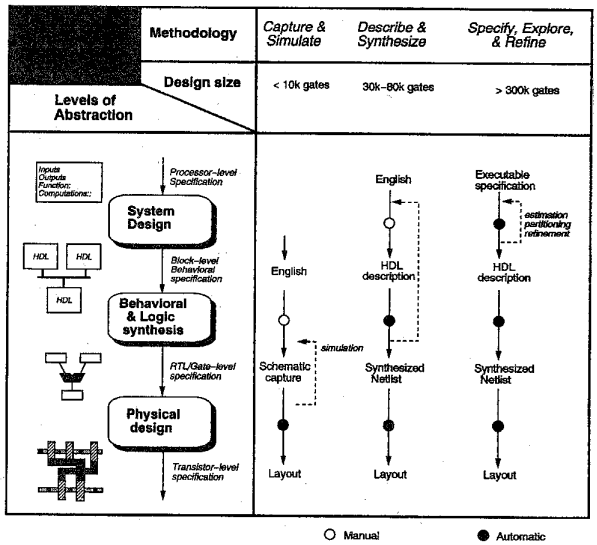


Fig. 2. Design methodologies: the dotted lines show the iteration loop.

change in design methodology, where the capture-and-simulate approach is steadily giving way to a **describe-and-synthesize** methodology. The advantage of this new methodology is that it allows us to describe a design in a purely behavioral form, void of any technology-specific implementation details; specifically, we can describe the design using Boolean equations and finite-state machine diagrams. In this methodology, the design structure is generated by automatic synthesis using CAD tools, instead of by manual design, since manual design is very tedious for all but trivial circuits.

The describe-and-synthesize methodology can be applied on several levels of abstraction. On the gate level, functional and control units could be synthesized using **logic synthesis**. For example, functional units such as ALU's, comparators, and multipliers, can be described by Boolean equations, and are traditionally synthesized in two phases. In the first phase, called logic minimization, the number of "and" and "or" operators (or, equivalently, the number of literals) in the Boolean equations are minimized while simultaneously satisfying cost and time constraints. In the second phase, called technology mapping, these minimized Boolean equations are then implemented using the logic gates from the given gate library in a selected technology. A comprehensive survey of logic synthesis techniques may be found in [1].

The control units, on the other hand, would be defined by finite-state machine diagrams, and then also synthesized in two phases. In the first phase, called state minimization, the number of states is minimized and a binary encoding assigned to each state so that the cost of implementing the next-state and output function will be reduced. In the second phase, the next-state and output functions defined by the Boolean equations are optimized through logic minimization and technology mapping, as described above.

On the register level, the microchips, which represent processors, memories and ASIC's, can be synthesized using **behavioral** (or **high-level**) **synthesis** techniques. The structure of these microchips consists of the functional, storage and control units that have been predesigned and stored in a register-level library. The behavior of these microchips can be described by means of programs, algorithms, flowcharts, dataflow graphs, instruction sets or by generalized finite-state machines, in which each state can perform arbitrarily complex computations.

We transform such a behavioral description into a structural one by applying three major synthesis tasks: allocation, scheduling and binding. The purpose of **allocation** is to determine the number of register-level components or resources that are required for the implementation. In other words, it determines the number of functional units, the operations executed by each unit, the number of pipeline stages, and the delay for each operation, as well as the cost and size of each unit. The allocation task must also determine the number of storage units, such as registers, register files, queues, and memories that are needed. In addition to the size and cost of each storage unit, the allocation task determines the number of ports and the access time for each storage unit. Finally, allocation has to also determine the number, size, protocol, and delay of each bus in the system, and the various options for connecting the functional and storage units to these buses.

Once resources have been allocated, the task of **scheduling** is intended to partition the behavioral description into time intervals, called control steps. During each control step, which is usually one clock-cycle long, data will be transferred from one register to another, and if necessary transformed by a functional unit during the transfer. All the register transfers in each control step are executed concurrently. The performance of the design (measured in total number of control steps) is roughly proportional to the number of available resources in each control step.

It is important to note that the scheduling task determines the operations that are executed in each control step, but it does not assign them to particular register-level components. This job is performed by the **binding** task, which assigns variables to storage units, operations to functional units, and ensures that a communication path or bus is assigned for each transfer of data between the storage units and the functional units. An introduction to high-level synthesis concepts can be found in [2].

By using logic and behavioral synthesis, the describe-and-synthesize methodology allows designers to describe a microchip's functionality with a behavioral description (that is void of any structure, engineering or technological information), and then synthesize a structural description consisting of register-transfer level (RTL) components. These RTL components can later be synthesized into gate-level components.

After the successful introduction of logic and behavioral synthesis tools, system designers and the CAD community alike are questioning whether the describe-and-synthesize methodology can be expanded to apply to complete systems, including software and hardware design. In other words, there is a possibility that productivity gains might be even higher if we continue this trend beyond chip-level design, and focus on even higher levels of abstraction. In the next section, we will describe the requirements and the essential issues for such a system-level methodology.

III. The Specify, Explore and Refine Methodology

System design is the task of mapping the system's functionality to a set of system components such that design constraints on parameters such as monetary cost, performance, and power are satisfied. Example **system components** include standard processors and microcontrollers, memories, buses, and custom ASIC's.

In the software domain, there exist several well-established design methodologies [3] for designing large software systems. **Structured Analysis** [4] relies on data flow diagrams to represent the organization of the software. Each node in the diagram represents a subtask performed by the software, and the arrows between the nodes represent the flow of data between the various subtasks. **Structured Design** [5] attempts to reduce complexity by dividing a large program into a set of independent modules and provides measures for minimizing the data references across modules. A "structure chart" is used to represent the modules in the system and the calls between them. Techniques such as **Flowcharts, State Transition Diagrams**, and **Jackson Diagrams** provide graphical notations for representing iteration, branching and sequencing in software specifications. For applications such as database design where the organization of the data outweighs all other aspects of the design, **Entity relationship diagrams** [6] have been employed to represent entities (a unique type of data that possesses one or more specific attributes) and relationships (a "fact" relevant to its entities) between them. **Object Oriented Programming** reduces complexity by allowing "objects" (an object is an encapsulation of data and functions that operate on the data) to be grouped together into a hierarchy of classes, allowing an object to inherit common characteristics from the class, as opposed to specifying them for each object individually.

For hardware design, design methodologies are not well established at higher levels of abstractions that deal with the design of entire systems. Every design organization has its own methodology for the design of large systems. These methodologies typically involve informal techniques that may vary from one group to another, even within the same organization. Traditionally, system design starts off with the chief architects creating informal block diagrams (representing system components), from which a specification is written after some preliminary design has been performed. The design decisions that lead to this block diagram, however, are usually based on a given designer's personal experience, rather than on a thorough exploration of all the possible algorithmic, architectural, and technological alternatives. Furthermore, these block diagrams are usually created without a full understanding of the system's functionality. This kind of delay in defining the system's functionality often results in a design cycle that is longer than necessary, since inconsistencies that are discovered late in the process require time-consuming design iterations.

In order to address the problems outlined above, we propose a **Specify-Explore-Refine** (SER) methodology for system design. In the **specify** phase of the SER methodology, the systems's functionality is captured using an executable specification language. The **explore** phase consists of an evaluation of different mappings of the system functionality to different system components, with a view to satisfy design constraints. Finally, in the **refine** phase, the specification is updated to reflect the decisions made during the exploration phase.

The result of the SER methodology is a set of interconnected system components, each with its own functional specification. Each component can then be concurrently implemented. A standard component (such as a microprocessor) requires software compilation of the functional specification into machine code, whereas custom components require synthesis of the specification into register-transfer structure. The first task is accomplished with standard compilers while the second one uses behavioral and logic synthesis.

We now describe each of the three stages of the SER methodology individually.

A. System Specification

As mentioned above, the traditional "back-of-the-envelope" design approach deals with system functionality in an ad-hoc manner in that the specification of each system component is written after system design has already been performed. It would be preferable to devote more effort to specifying the system's functionality in the earliest stage of the process, before any design decisions have been made, since such early effort could lead to large overall savings. Specifically, there are great advantages to be derived by working with a specification, particularly an executable specification, since this would not only capture the product's functionality, but can also be used by marketing departments to study the competitiveness of the product in the marketplace. In addition, an executable specification can serve as documentation during all steps of the design process, especially insofar as it fosters concurrent engineering, by clearly defining the functionality and the interface for the various subsystems assigned to various members of the design team. Furthermore, any change in any of these subsystems would be easy to incorporate, and its impact on other parts of the system could be rapidly evaluated. An executable specification is also amenable to the automatic verification of different design properties, as well as the functionality of the system. In addition, an executable specification has the advantage of enabling automation during design exploration and design synthesis. Finally, an executable specification can also continue to serve as a starting point for all the product upgrades that occur during the life-time of the product, as well as supporting product maintenance.

The selection of a language for writing these specifications emerges as one of the main issues in a system methodology. Such a language must be easy to capture, to understand, and to use for interfacing with CAD tools. It must also be able to capture all the system's characteristics and allow the easy synthesis of their implementations. Finally, such a language should be able to model these systems and their implementations in a manner that is readable and complete, without being overbearing to the designer. Many languages have been used for executable specification, including VHDL [7], Verilog [26], HardwareC [8], [9], CSP (Communicating Sequential Processes) [10], Statecharts [11], SDL (Specification and Description Language) [12], Silage [13], and

Language	Embedded System Features					
	State Transitions	Behavioral Hierarchy	Concurrency	Program Constructs	Exceptions	Behavioral Completion
VHDL	○	◉	●	●	○	●
Verilog	○	●	●	●	●	●
HardwareC	○	◉	●	●	○	●
CSP	○	●	●	●	○	●
Statecharts	●	●	●	○	●	○
SDL	●	◉	●	○	○	●
Silage	-	-	●	-	-	-
Esterel	○	●	●	●	●	●
SpecCharts	●	●	●	●	●	●

● Feature fully supported ◉ Feature partially supported ○ Feature not supported - Not applicable

Fig. 3. Language support for conceptual model characteristics of embedded systems.

Esterel [14]. Such languages are used to capture common conceptual models such as finite-state machines (FSM), FSM's with complex expressions, hierarchical and concurrent FSM's, dataflow diagrams, Petri-nets, and communicating sequential processes. Unfortunately, these conceptual models are not adequate for concisely describing all the characteristics of a class of systems referred to as **embedded systems**.

An embedded system is typically designed to perform a set of functions as part of a larger system and constantly responds to external events and interrupts in real time. We modeled several embedded systems including a bus controller, a robot arm-tracking system, an MPEG decoder, an image processing system, fuzzy-logic controllers, a microwave-transmitter controller, a medical volume-measuring system, a telephone answering machine, an interactive TV processor, an aircraft collision avoidance system, and an ethernet co-processor. The following five characteristics were common to the models of all the above systems: state-transitions, hierarchical sequential and concurrent behavior decomposition, exceptions, behavioral completion and algorithmic computations specified with programming constructs. Fig. 3 lists the embedded system characteristics that are supported by each of the above languages. It is clear from the figure that none of these languages and their underlying model are by themselves sufficient to model embedded systems concisely and precisely. To overcome this limitation, a model called program-state machines can be used.

1) Program-State Machines: Program-state machines (PSM) [15] are essentially a combination of the hierarchical finite-state machine and programming language paradigms. A system is specified as a hierarchy of **program-states**, where each program-state represents a mode of computation and may include standard programming declarations such as variables, types, and subroutines. At any given time only a subset of program-states are active, i.e., are actively carrying out their computations. A single root program-state represents the entire system and is always active.

A program-state may either be a **composite** program-state or a **leaf** program-state. A composite program-state may be hierarchically decomposed either into a set of **concurrent** program-substates (all program-substates are active when the program-state is active), or into a set of **sequential** program-substates (only one of the program-substates is active at a time when the program-state is active). Fig. 4 shows the program state machine for a simple computer system. *CPU* and *CLK_GEN* are concurrent program-substates of *SYSTEM*, while *RESET* and *ACTIVE* are sequential program-substates of *CPU*. A sequentially decomposed program-state contains a set of transition arcs to represent the sequencing between the program-substates. There are two types of transition arcs. A **transition-on-completion arc (TOC)** is traversed when the source program-substate has completed its computation and the associated arc condition evaluates to true (e.g., the transition from *RESET* to *ACTIVE* originating in a bold dot). A **transition-immediately arc (TI)** is traversed immediately when the arc condition becomes true, regardless of whether or not the source program-substate has completed its computation (e.g., the transition labeled *rising(RESET_IN)* from *ACTIVE* to *RESET*, originating at the boundary of *ACTIVE*). A leaf program-state (e.g., *FETCH*) is at the bottom of the behavioral hierarchy and has its computation described using programming language statements.

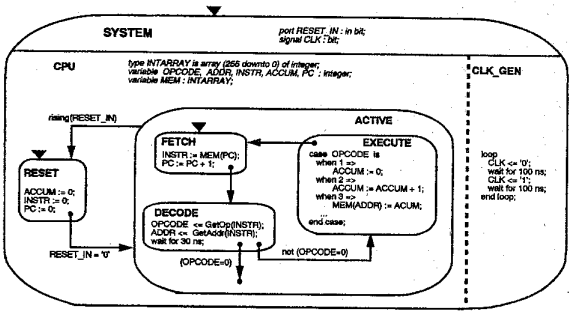


Fig. 4. Program-state machine model of a simple computer system.

The PSM model supports all embedded-system characteristics in an elegant manner. In addition, as the system is refined, the programming constructs can be used to describe portions destined for software implementation while the state-transitions describe portions destined for hardware implementation, all with a single uniform representation that eliminates the need for multiple languages. Since no language currently exists that supports all the PSM characteristics, we developed a VHDL front-end language called SpecCharts [16].

Our subsequent system-design methodology is not strictly dependent on use of SpecCharts. Other languages may be used to capture the PSM model, with some extra effort. However, SpecCharts is in closest accord with SER system-design methodology, and yields the most concise and readable specifications.

B. Design Exploration

Our approach to system design consists of three well-defined tasks on three classes of functional objects. These tasks are summarized in Fig. 5.

The three classes of functional objects that comprise any executable specification are **variables**, **behaviors**, and **channels**.

Functional objects	**Specification** *(e.g. VHDL)*	**Exploration**		**Refinement**
		Allocation	**Partitioning**	
Variables	signals, variables	Memories	Variables to memories	Address assignment
Behaviors	processes, procedures	Processors	Behaviors to processors	Interfacing
Channels	global signals, ports, port maps	Buses	Channels to buses	Arbitration/protocols

Fig. 5. System design: tasks.

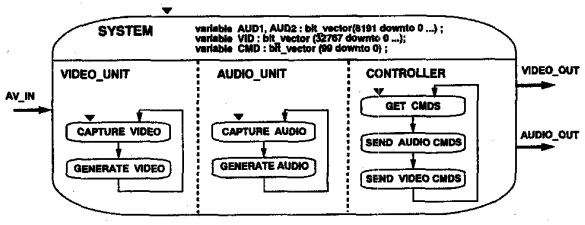


Fig. 6. Partial SpecChart of an A/V controller.

Variables store data, behaviors transform data, and channels transfer data between behaviors. Equivalent VHDL constructs that illustrate these three classes of functional objects are listed in the specification column in the figure. In our terminology, a behavior is a nontrivial algorithmic-level computation that together with other behaviors describe all system actions (identical to the "task" concept described in [17]). It corresponds to a block of statements in the specification such as a loop body, procedure, or process.

As an illustration of the three classes of functional objects in a specification consider Fig. 6 which shows a partial SpecChart description of an audio/video (A/V) controller. The AV controller receives a stream of audio and video streams over the *AV_IN* channel, stores them temporarily in memories and when instructed by a controller, performs some computations on the samples and outputs the audio and video sample on the two distinct channels *AUDIO_OUT* and *VIDEO_OUT*. The AV controller consists of a hierarchy of behaviors of which only the top three levels are shown in Fig. 6. At the top most level, we have the behavior *SYSTEM*. This behavior has four array variables: *AUD1* and *AUD2* to store audio samples, *VID* to store video samples and *CMD* which stores the set of stream manipulation commands. The *AUDIO_UNIT* and *VIDEO_UNIT* detect, capture, store and generate audio and video samples respectively. The *CONTROLLER* fetches instructions from the *CMD* memory and issues appropriate instructions to the *AUDIO_UNIT* and *VIDEO_UNIT*.

For each of these objects, there are three tasks to be performed in the SER methodology: allocation, partitioning, and refinement. **Allocation** adds system components to the design. There are three classes of system components that can be allocated. One class of system components consists of memories, such as RAM's, ROM's, register-files, and registers. Memories are used for storing scalar and array variables. Another class of components consists of standard processors and microcontrollers as well as custom ASIC "processors". These standard/custom processors are used to implement behaviors. A third class of "component" consists

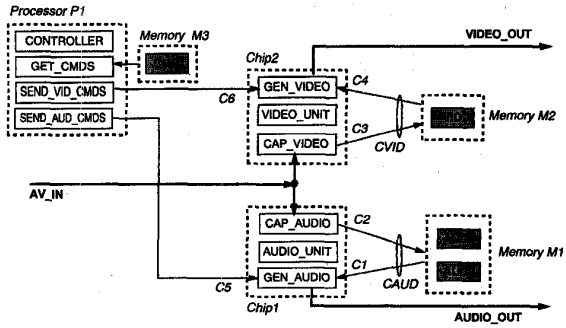


Fig. 7. Allocation and partitioning for the A/V controller.

of physical buses. Buses are used to implement groups of communication channels.

The designer may specify appropriate constraints or parameters that are required to define each allocated component. For example, processors are defined by their instruction sets and the execution speed for each instruction; memories are defined by their sizes, read/write protocols and access times; ASIC's are defined by their sizes in terms of gates (gate arrays) or number of transistors (custom design), propagation delays for gates or transistors, package size, and allowed power consumption, among other things. Buses are defined by the number of data/control lines, the protocol used and its bandwidth.

For the A/V controller, the designer has allocated six system components shown in Fig. 7—a processor *P1*, two chips *Chip1, Chip2*, and three memories *M1, M2, M3*. For each of the allocated components, the designer will have specified the appropriate characteristics.

Once the components have been allocated and constrained, the functional objects in the specification are then **partitioned** into the allocated software and hardware components. Variables are mapped to memories, behaviors are mapped to standard/custom processors, and channels are mapped to buses. Each mapping is many to one. Standard partitioning algorithms, such as clustering or simulated annealing, can be applied. Various closeness criteria [15] can be used to determine which objects should be clustered together. For behaviors, common criteria include interconnection, communication, sequentiality, and hardware sharability. For variables and channels, common criteria include sequential access, common accessors, and width similarity.

Depending on the constraints specified for the system, different portions of the design may be partitioned to be eventually implemented as software or hardware. The software part of the system can be further subdivided into two or more parts, each running on a separate processor. For example, a host processor could perform the slower system functions while one or more coprocessors could be executing the faster data transformations. Similarly, the hardware part of the system may be partitioned into one or more ASIC's.

One mapping of the functional objects in the A/V controller specification to the allocated system components is shown in Fig. 7. For example, the *CONTROLLER* behavior was not time critical and was thus mapped to a software implementation

on processor *P1*. The *AUDIO_UNIT* and *VIDEO_UNIT* were mapped to chips *Chip1* and *Chip2* respectively. Since the two arrays (i.e. *AUD1* and *AUD2*) used for storing audio samples are accessed sequentially, they can be mapped to the same memory (*M1*) without any performance degradation of the the behavior *AUDIO_UNIT*.

Since each different allocation of system components and each different partition will produce one candidate system implementation, evaluation of these various options will require designers to estimate quality metrics such as performance, cost, power consumption, testing and packaging costs for each implementation. Each set of estimated quality metrics is compared to the given requirements and the candidate implementation that satisfies the requirements optimally is selected. Thus, design exploration from the specification in the SER methodology will allow designers to find the most cost-effective solutions.

C. Specification Refinement

System partitioning in the SER methodology causes a regrouping of the objects in the specification. Once the best solution has been found, the specification will need to be refined to reflect the allocation and partitioning decisions, such that the different objects in the specification are moved to the appropriate components, and communication is maintained between the separated pieces. For example, many variables in the description may be moved into one partition and grouped onto a single memory, or a single behavioral description may be split across multiple chips. It is necessary to add behaviors to the individual chips to maintain correct functionality for the given allocation and partitioning. In our first example, since many variables are stored in the same memory, address translation must be introduced for each memory access. In the second example, where behaviors are split among chips, additional interface descriptions must be introduced into both the chips to maintain correct communication. These sets of tasks are called **refinement**.

Refinement migrates the design from a pure functional specification toward a structural implementation. Successive refinement steps generate a sequence of models, each with more structural detail than the previous one. For example, at the system level, a structural description might define the chips and buses of a system. Each chip may be refined functionally to consist of a set of behaviors or processes. Behavioral synthesis is a refinement step that will produce a RT-level structural netlist to implement each process. Each RT component (e.g., adders, etc.) may have a behavioral description of its functionality associated with it. Thus, each refinement step results in a progression of detail which is added to the specification. Refinement does not change the functionality, it simply adds details of the implementation to the system model.

In the SER methodology, refinement adds new behaviors to maintain the correct functionality for a given allocation and partitioning. Variables partitioned among memories require memory address translation. Behaviors separated among components must be modified to maintain correct communication. Channels mapped to buses require interface synthesis to determine communication protocols, and arbiter synthesis to resolve any simultaneous bus requests. A refined specification is then generated consisting of a set of interconnected system-components, the functionality of each of which is completely specified.

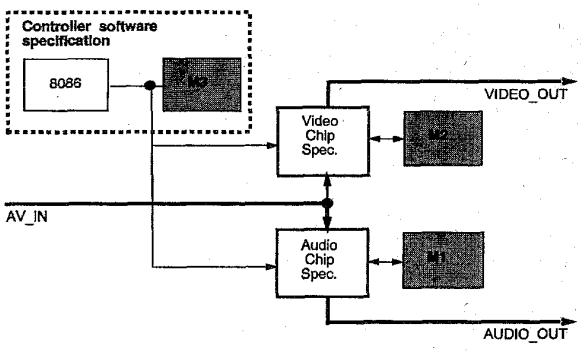


Fig. 8. System specification after refinement.

After such refinement, the specification will reflect the product architecture (as did the block diagram created by the chief architect in the traditional approach), with all the system components and the communications among them well defined. Each component has its own specification, which can then be compiled by a standard compiler in the case of computations assigned to the processors, or synthesized with behavioral and logic synthesis tools in the case of computations assigned to custom ASIC's. There are, of course, differences between the refined specification and the architect's block diagram. First, in that the refined specification has been obtained only after a thorough and organized exploration of the solution space; and second, in that the refined specification, being derived formally from the original specifications, is far more likely to be consistent, eliminating the need for expensive, time-consuming design iterations.

Fig. 8 shows the refined specification of the A/V controller. There are three memory components, which implement the variables in the specification. Hardware for the *AUDIO_UNIT* and *VIDEO_UNIT* is designed manually or by applying hardware synthesis tools on the audio and video chip specifications. The *CONTROLLER* behavior is now represented by software specification. This is compiled by a software compiler to execute on processor *P1*. Finally, the three memories, the two synthesize hardware chips, and the processor are all connected together and assembled on a board. Thus, the entire A/V controller design is structured to exploit concurrent engineering and consequently, reduce the time to market.

IV. Fuzzy Logic Controller

In order to demonstrate the advantages of using the higher abstraction levels, we selected an industrial design used in consumer and industrial electronic applications. The particular application that we used was a fuzzy logic based temperature controller. Our goal was to specify the design at the system level and synthesize the entire design into a set of FPGA's. Another important goal was to gauge the approximate time re-

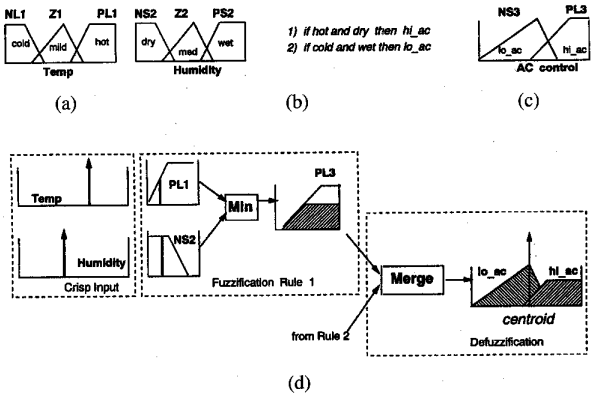


Fig. 9. Fuzzy Controller: (a) front-end fuzzy sets, (b) rules, (c) back-end fuzzy sets, and (d) operation of fuzzy controller.

quired for the complete design process from conceptualization to implementation.

In this section we provide the details of the fuzzy logic controller. We then discuss the fuzzy logic design experiment and the tradeoffs that were achieved in each stage of the SER design process.

A. Fuzzy Logic Controller: Basic Principles

The basic idea in fuzzy logic [18] is that everyday crisp values such as "temperature is 25 C" can be expressed as having a membership value in fuzzy sets such as "temperature is 0.9 hot" and "temperature is 0.01 cold," where hot and cold are fuzzy sets and the values 0.9 and 0.01 represent the degree of membership in the respective sets.

A fuzzy logic controller incorporates three databases. **Front-end fuzzy sets** or **membership functions** determine to what degree an input fact belongs to them. Each input variable has its own set of membership functions. **A rule base** governs the behavior of the fuzzy controller. Typically, these rules are in the form of *if..then* statements such as "if NL1 and NL2 then PS3", where NL1 and NL2 represent degree of membership in the membership functions, and PS3 represents the strength of the output control. **Back-end fuzzy sets** determine the value of the outputs of the fuzzy controller based on the degree to which the various rules were satisfied. For an air-conditioning system control application, Fig. 9(a), (b), and (c) show the front-end fuzzy sets for two variables *temp* and *humidity*, two rules, and the back-end fuzzy sets for an air-conditioning fuzzy logic controller respectively.

The operations of the fuzzy controller are summarized in Fig. 9(d). Briefly, the fuzzy controller [19] performs the following functions.

1) **Fuzzification** converts the crisp input values for the variables into a fuzzy membership value. The operations involved in this step include many table lookups and comparisons.
2) **Rule Application** applies all the rules and produces a fuzzy output membership value. The operations in this step include truncation and convolution of the membership functions.
3) **Defuzzification** converts the fuzzy output membership values produced by the rules. The operations here include computing the centroid of the back end membership functions.

B. Design Specification

As explained earlier, for the first step of the SER methodology we needed a VHDL based specification mechanism that supported behavioral hierarchy and concurrency. We used the SpecCharts language [16] for design specification since it satisfied all our requirements. The level at which the design is captured in SpecCharts is very close to the designers conceptualization of the design. Besides, the language is a front-end for VHDL-based specifications and thus, provides a path to subsequent simulation and synthesis tools. In addition it directly supports behavioral hierarchy and state-machine based control flow.

The design specification consisted of five behaviors expressed using VHDL process statements. Four of these processes modeled the rules (R1$\cdots$R4) in the system. The fifth behavior modeled the defuzzification function (i.e., centroid computation).

During specification, it was not clear whether to evaluate the four rules sequentially or concurrently. We decided to explore both these options and estimate the size and performance for each of these options during system design. Hence we wrote two different specifications of the design. In the first specification, we created a hierarchical state, in which the four rules were executed sequentially. In the second specification, we created a similar hierarchical state, but the four rules were executed concurrently. With SpecCharts it took less than 10 min to specify these two hierarchical descriptions from the leaf behaviors.

Since SpecCharts can be easily translated to VHDL, we were able to simulate both the sequential and the parallel specifications using a commercial VHDL simulator. It took us about three days to write both these specifications. The total lines of SpecChart code for the specification was about 300 lines.

C. System Design

Specsyn [20], a system design and exploration tool based on the SER methodology, was used to explore various architectures by partitioning the design into several chips. Since our goal was a final mapping onto FPGA's, the system components that were allocated for partitioning purposes were only FPGA's.

Many FPGA allocations were tried during system design. For each allocation provided to it, Specsyn partitioned the entire behavior into the allocated units such that the performance was optimal. In fact, Specsyn evaluates hundreds of different partitions in a few minutes, before deriving the most efficient partition. Specsyn contains built-in performance and cost estimators, that provide quick and accurate on-the-fly estimates of these quality metrics. These estimates drive the optimization process during partitioning [21].

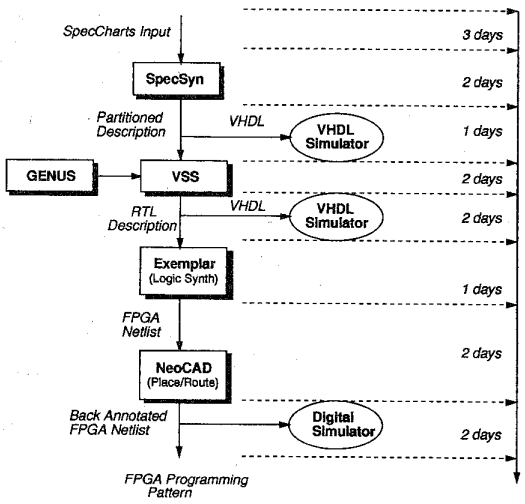


Fig. 10. Fuzzy controller—design steps.

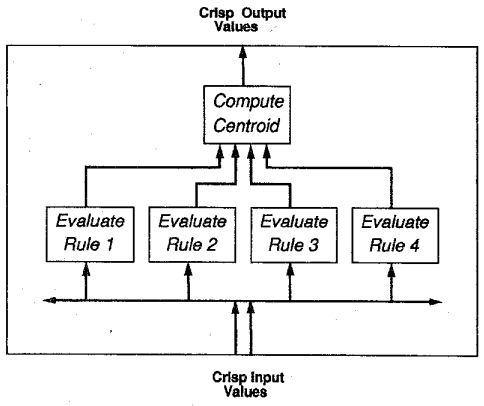


Fig. 11. Partitioning results.

For example, Specsyn's performance estimation feature was used to compare both the sequential and the concurrent rule evaluation styles. Partitioning and performance estimation not only revealed that the parallel algorithm is faster and requires more area than the sequential one, but also how this tradeoff impacts the overall system performance of the fuzzy controller (the true design criteria). The performance and cost estimators allowed a rapid comparison between the two alternatives. Due to constraints on the overall performance, a concurrent implementation was selected for the four rules in the fuzzy logic controller. The automated estimators allow such decisions to be made in a matter of minutes instead of a few days or even weeks required by a manual investigation.

With a couple of days experimentation, we derived the best partition for the design. The refined design at the system-level (shown in Fig. 11) consists of five major blocks, with appropriate interface protocols introduced between them for communication.

Thus, at the end of the system-level exploration phase of the SER methodology, the objects in the original specification were partitioned into five chips. A VHDL behavioral description was generated for each of the five chips and simulated to verify the partitioning and the interfaces synthesized between the chips.

```
entity ch0e is
port ( f0: in integer;   f1: in integer;   gclk: in bit;   exec: in bit;
       done: out bit;   initr0m: in bit;   addrr0m: in integer;
       datar0m: in bv7;  initrRu0: in bit;  datatrRu0: in bv7;
       addrtrRu0: in integer;   testdoutr0m: out bv7;
       testr0m: in bit;   testaddrr0m: in integer;   testtrRu0: in bit;
       testaddrtrRu0: in integer;   testdouttrRu0: out bv7);
end ch0e;

architecture ch0a of ch0e is

       type MembRuleArr is array (integer range <>) of bv7;
       type MembFunctArr is array (integer range <>) of bv7;
       signal r0m: MembRuleArr(383 downto 0);
       signal trRu0: MembFunctArr(127 downto 0);

begin
P0: process
       variable if1,itr, ir0mtr: integer;
       variable r0mf0, r0mf1, r0mtr, truncVal, temptr: bv7;
begin
   if exec = '1' then
       done <= '0';  if1 := 128+f1;  r0mf0 := r0m(f0); r0mf1 := r0m(f1);
       if r0mf0 < r0mf1 then truncVal := r0mf0; else truncVal := r0mf1; end if;
       itr := 0;
       while itr < 128 loop
          ir0mtr := itr+256;  r0mtr := r0m(ir0mtr);
          if truncVal < r0mtr then  temptr := truncVal;  else temptr := r0mtr;  end if;
          trRu0(itr); itr := itr+1;
       end loop;
       done <= '1';
   elsif initr0m = '1'  then  r0m(addrr0m) <= datar0m
   elsif initrRu0 = '1'  then  trRu0(addrtrRu0) <= datatrRu0;
   elsif testr0m = '1'  then  testdoutr0m <= r0m(testaddrr0m);
   elsif testtrRu0 = '1' then  testdouttrRu0 <= trRu0(testaddrtrRu0;
   end if;
end process P0;
end ch0a;
```

Fig. 12. Rule description in VHDL.

D. Architectural Design

Behavioral descriptions for each of the five chips were obtained after system design. In Fig. 12 we show fragments of the generated VHDL behavior for a single rule evaluation (EVAL_R1). We used VSS (VDHL Synthesis System) [22] for further exploration and refinement of the design at the microarchitecture level and synthesis of the RTL structure. VSS performs scheduling, allocation, binding and array variable mapping and produces the RTL design. Components required during synthesis are obtained from the Genus library [23], which is a library of parametrized RTL components.

By varying the amount of allocated resources, we were able to explore a wide range of area—delay tradeoffs at the architectural level. We list some of the exploration scenarios that were attempted with the VSS system.

a) We tried different allocations of functional units. The performance of the design did not improve with more functional units, because this description (shown in Fig. 12) does not contain many parallelizable operations.
b) We changed the type of functional unit resources. Instead of providing a separate adder and a separate comparator we allocated a single ALU which can perform both these operations. This changed the design characteristic. Although the number of states required increased because of fewer resources, the utilization of the allocated components increased substantially.
c) Since the fuzzy logic control is a memory intensive application, changes in the allocation of memory resources drastically affected the final design. By increasing the number of ports in the memory we were able to significantly improve the performance. We also

allocated slower and faster memories to consider the impact of memory access time on the overall design. We finally fixed the number of ports and the access time to correspond to the values in the Xilinx FPGA's.

d) We selected a 100 ns clock for the design, since most of the functional unit delays in the FPGA library was in the order of 30-40 ns, and data accesses were in the order of 60 ns.

Since VSS uses fast algorithms for scheduling, variable merging and binding, it was able to produce the architectural design for a given allocation within a minute. Thus we were able to synthesize and verify almost about 10 different allocations in less than a day.

The output produced by VSS is fully simulatable. However, the amount of VHDL code is quite large since the details are at the RT level. The VHDL code for each of the FPGA's was about 2500 lines, and the total lines of VHDL for all the FPGA's was over 10 000 lines. However, this code was directly simulated on a commercial simulator to verify correctness of the synthesized design.

E. Technology Adaptation

The RTL VHDL output from BdA was the input for commercially available logic synthesis tools which produce gate-level schematics from the RT level. We used the logic synthesis tool from Exemplar for synthesizing the gate level design. The target synthesis architecture were the Xilinx FPGA's. A few minor modifications in the VHDL description of some of the GENUS components was necessary, in order to enable the logic synthesis tool to incorporate technology specific macros or semi-custom cells for complex components such as fast adders and memories into logic synthesis. These macros or cells greatly enhanced the performance of the system when compared to synthesizing the complex functions directly.

The technology-dependent schematic was the input for place and route tools, the final piece of the design puzzle. A commercial tool from NeoCAD was used for placement and routing of the FPGA's. The role of these tools are well known. After place and route a backannotated circuit schematic was generated for final digital simulation. The digital simulation incorporated timing delays due to logic components and routing delays. This provides final verification before proceeding to actual silicon. The total time spent in the final phase of the design process was about five days.

However, during the final digital simulation of the back annotated design it was found that the wire delays were very large (almost 75 ns), making it impossible to satisfy the clock constraint of 100 ns. Thus a second iteration through the design process was necessary.

F. Second Design Iteration

The design was resynthesized with a clock cycle of 200 ns. In Fig. 13, we show some of the synthesis characteristics for one of the FPGA's (EVAL_R1). The final design occupied 360 CLB's in the FPGA. The clock constraint of 200 ns was achieved after placement and routing of the FPGA. The total time required to evaluate each rule was 180 μs. The design (shown in Fig. 14) produced during the second iteration was able to run through the simulation successfully.

DESIGN DESCR	Num Lines of SpecChart Code	50 lines
	Lines of VHDL after SpecChart	100 lines
	Lines of VHDL after VSS	2500 lines
COST	FPGA type	Xilinx 4000
	Number of CLBs	360
	Number of Gates	9K
CLOCK	Clock Cycle Constraint	200 ns
	Clock Cycle After VSS	70 ns
	Clock Cycle After NeoCAD	145 ns
PERF	Performance Constraint	200 us
	Perf. Estimated by SpecSyn	155 us
	Perf. Achieved by VSS	155 us
	Perf. after NeoCAD	180 us

Fig. 13. Results: EVAL_R1.

Since the synthesis process was rerun from the VSS level onwards, another nine days of work was required before the chip passed through the test vectors. In the future, the quality of the area and delay estimation tools have to be improved to avoid this second iteration of the design. The final implementation, consisting of five Xilinx 4010 FPGA's on an Aptix GP4 field programmable circuit board (FPCB), performed correctly under the performance constraint clock speed of 200 ns.

V. Conclusions

In this paper we have reviewed system and chip level methodologies. We have illustrated the power of these methodologies with an industrial design, that was a) specified at the highest levels of abstractions, b) synthesized at the system level, c) further synthesized at the RT level, and finally d) implemented with FPGA's using commercial tools.

The design quality resulting from our 100 h methodology is comparable to manual designs. In one experiment, we compared number of transistors of a manual design with that resulting from using the SER approach with SpecCharts as the input language. The example chosen was the telephone answering machine [15]. An English specification was given to two designers. One designer took a manual approach to design where the datapath was first created by hand and the controlling state machine described using Berkeley's KISS format, on which the Mustang [24] and MIS [25] synthesis tools was applied to generate the controller logic. The other designer specified the system with SpecCharts, flattened the hierarchy automatically and translated the flattened SpecCharts to VHDL automatically. From the VHDL the designer then created a datapath and controlling state machine manually (but automatable with existing commercial synthesis tools), and applied the Mustang and MIS synthesis tools to generate the controller logic. The results of creating the design using the two approaches is shown in Fig. 15. The important thing to note is that the number of transistors in the final design obtained by the SpecCharts designer is not greater than that

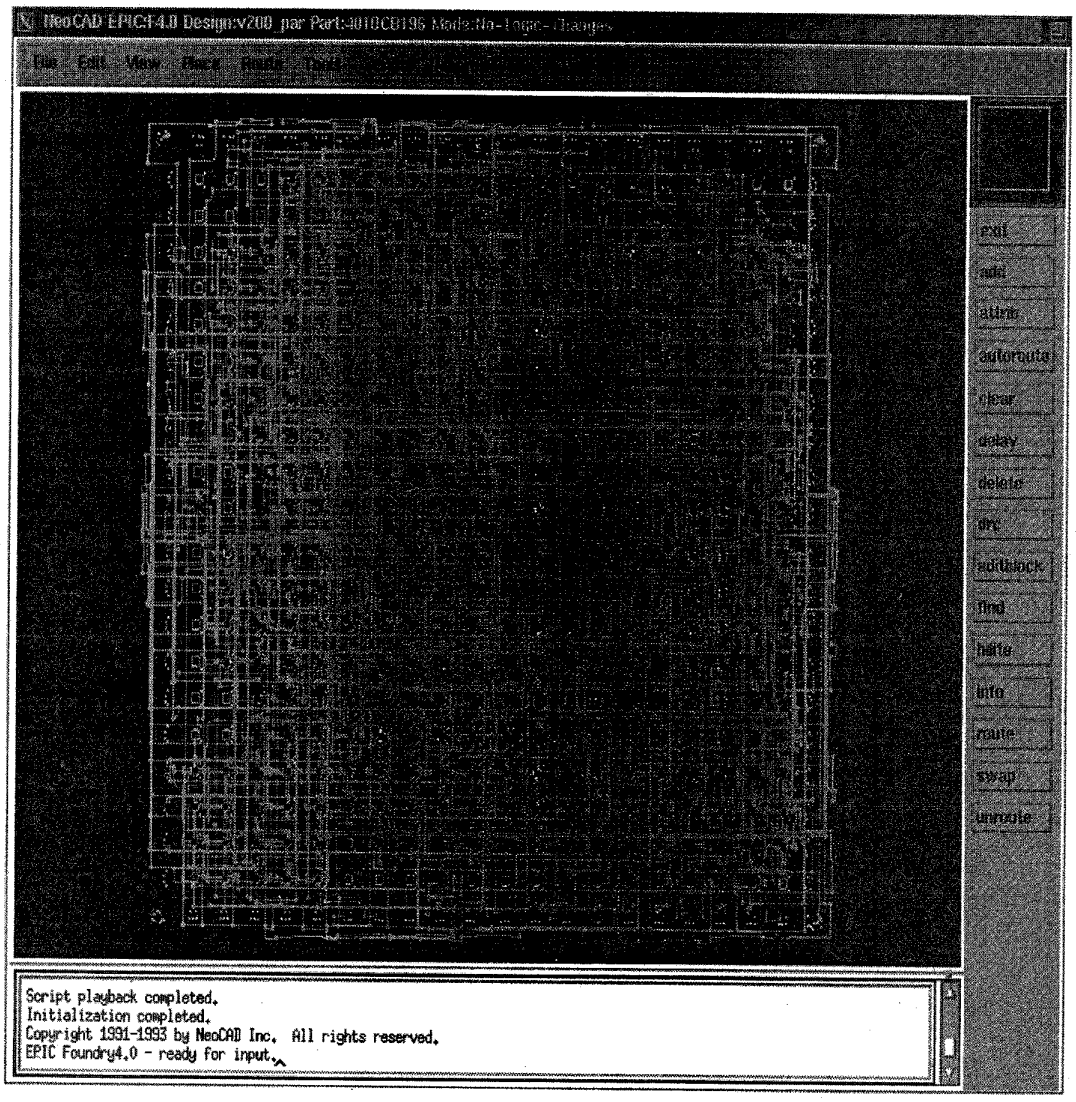


Fig. 14. Placement and routing results: EVAL_R1.

obtained by the manual designer. In this case, the number is actually fewer, resulting from fewer control states. The manual designer had to keep the state machine readable since it was the functional specification of the system and hence had to be modified as functional errors and omissions were detected. Keeping the state machine readable prevented him from sharing many states. Conversely, SpecCharts was the functional specification for the other designer, so readability of the state machine was not an issue. The designer used an algorithm to convert SpecCharts to a state machine that resulted in many shared states.

Since the designer deals with higher levels of abstraction, the SER approach allows a large design space to be explored by quickly changing some of the exploration parameters. For example, numerous partitionings at the system level and numerous microarchitectures were quickly designed (in a few minutes). Providing hooks to VHDL allowed verification of the designs at each of the abstraction levels using simulation.

Design attribute	Designed from English	Designed from SpecCharts
control transistors	3130	2630
datapath transistors	2277	2251
total transistors	5407	4881
total pins	38	38

Fig. 15. Design quality from SpecCharts versus English specifications.

The advantage of the 100 h methodology is that it promises significant improvements in productivity, since precise specification, automatic exploration and refinement would help us to avoid long design cycles with many iterations. Such a methodology would only require designers to select technology, to allocate components and to specify requirements, before exploring automatically hundreds of design alternatives in a single day and then refining the specification by adding more structural detail as architectural and technological decisions

are made. This process can be repeated until designers reach a completely structural description containing components at the proper level of abstraction, defined by the available component library.

The presented methodology achieves these productivity gains in three ways.

1) **Less Design Time:** The methodology requires the entire system to be captured using a specification language before system design is performed. Partitioning, estimation, allocation and refinement tools can be developed to operate on the specification and automate the system design process, significantly reducing the overall design time.
2) **Better Designs**: Built-in estimators provide the designer with rapid feedback after each system design task. The designer has the capability of interactively exploring a larger design space rapidly which will lead to faster, cheaper and smaller designs.
3) **Less Redesign Time**: Requiring the designer to capture the conceptual view using a specification language and then subsequently refining the specification by applying system design tools, creates very comprehensive documentation. The various design decisions made during system design can be easily comprehended in any subsequent redesign effort.

Our test case demonstrates clearly that it is possible to achieve a total design cycle time of 100 h, which makes this methodology extremely useful in cases where designs have to be delivered within a few weeks. To meet this demand, the design process must migrate to higher abstraction levels, where the amount of detail is at a minimum.

ACKNOWLEDGMENT

The authors would like to thank Y. Shimmei (Matsushita Electric Works Research and Development Laboratory) for his constant encouragement, suggestions and useful discussions during the course of this project. They would also like to acknowledge the reviewers for their suggestions and comments.

REFERENCES

[1] S. Devadas, A. Ghosh, and K. Kuetzer, *Logic Synthesis*. New York: McGraw-Hill, 1994.
[2] D. D. Gajski and L. Ramachandran, "Introduction to high-level synthesis," *IEEE Design Test Comput.*, Winter 1994.
[3] W. Stevens, *Software Design: Concepts and Methods*. Englewood Cliffs, NJ: Prentice Hall, 1991.
[4] T. DeMarco, *Structured Analysis and System Specification*. New York: Yourdon Press, 1978.
[5] E. Yourdon and L. Constantine, *Structured Design*. Prentice Hall, 1979.
[6] P. S. Chen, "The entity-relationship approach to logical data base design," Q.E.D. Inform. Sci., Wellesley, MA, 1977.
[7] *IEEE Standard VHDL Language Reference Manual*, 1988.
[8] D. Ku and G. DeMicheli, "Synthesis of ASIC's with Hercules and Hebe," in *High-Level VLSI Synthesis*, R. Camposano and W. Wolf, Eds. New York: Kluwer Academic, 1991.
[9] D. C. Ku and G. DeMicheli, "HardwareC—A language for hardware design," Stanford Univ., Tech. Rep. CSL-TR-90-419, 1988.
[10] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.
[11] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Programming 8*, 1987.
[12] F. Belina, D. Hogrefe, and A. Sarma, *SDL with Applications from Protocol Specifications*. Englewood Cliffs, NJ: Prentice Hall, 1991.
[13] P. Hilfinger and J. Rabey, *Anatomy of a Silicon Compiler*. New York: Kluwer-Academic, 1992.
[14] N. Halbwachs, *Synchronous Programming of Reactive Systems*. New York: Kluwer-Academic, 1993.
[15] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*. Englewood Cliffs, NJ: Prentice Hall, 1994.
[16] F. Vahid, S. Narayan, and D. D. Gajski, "SpecCharts: A VDHL front-end for embedded systems," in *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 694–706, 1995.
[17] D. E. Thomas, J. K. Adams, and H. Schmit, "A model and methodology for hardware/software codesign," *IEEE Design & Test of Comput.*, pp. 6–15, 1993.
[18] G. J. Klir and T. A. Folger, *Fuzzy Sets, Uncertainty and Information*. Englewood Cliffs, NJ, Prentice Hall, 1988.
[19] G. C.Lee, "Fuzzy logic in control systems: Fuzzy logic controller—Parts I and II," *IEEE Trans. Syst., Man, Cybern.*, pp. 404–435, Mar./Apr. 1990.
[20] D. D. Gajski, F. Vahid, and S. Narayan, "A system-design methodology: Executable-specification refinement," in *Proc. European Conf. Design Automat. (EDAC)*, 1994.
[21] F. Vahid and D. D. Gajski, "Specification partitioning for system design," in *Proc. Design Automat. Conf.*, 1992.
[22] L. Ramachandran, N. Holmes, and D. D. Gajski, "The design process for behavioral synthesis from VHDL," U.C. Irvine, Dep. ICS, Tech. Rep. 94-04, 1994.
[23] P. Jha, T. Hadley, and N. Dutt, "The GENUS user manual and C programming library," U.C. Irvine, Dep. ICS, Tech. Rep. 93-32, 1993.
[24] S. Devadas, H.K.T. Ma, A.R. Newton, and A. Sangiovanni-Vincentelli, "MUSTANG: State assignment of finite state machines targeting multi-level logic implementations," *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 1290–1299, Dec. 1988.
[25] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A multiple-level logic optimization system," *IEEE Trans. Computer-Aided Design*, vol. 6, pp. 1062–1080, Nov. 1987.
[26] D. Thomas and P. Moorby, *The Verilog Hardware Description Language*. Boston, MA: Kluwer-Academic, 1991.

Daniel D. Gajski (M'77–SM'83–F'94) for a photograph and biography, see this issue, p. 5.

Sanjiv Narayan (M'95) received the B.S. degree in computer science in 1988 from the Indian Institute of Technology, New Delhi, India, and the M.S. and Ph.D. degrees in information and computer science from the University of California—Irvine in 1994.

He is a member of the R&D group at Viewlogic Systems, Inc., Marlboro, MA. His research interests include system specification and design, hardware description languages, and behavioral synthesis.

Dr. Narayan is the recipient of the Director's Gold Medal from IIT, New Delhi. He was a Chancellor's Fellow at the University of California—Irvine.

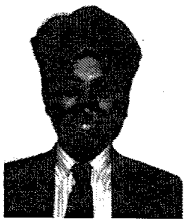
Loganath Ramachandran (M'95) received his B.Tech in Electrical Engineering from the Indian Institute of Technology, Madras in 1985, and his M.S and Ph.D in Computer Science from University of California, Irvine in 1991 and 1994, respectively.

He is a Senior Software Engineer in the Software R&D Department of LSI Logic Corporation, Milpitas, CA. Previously, he worked as a Software Engineer in the Design Automation Division of Texas Instruments, India. His current interests include system-level specification and synthesis, hardware-software codesign, high-level synthesis and multimedia architectures.

Dr. Ramachandran is a recipient of the Chancellor's fellowship from the University of California, Irvine.

Frank Vahid (S'89–M'95) received the B.S. degree in electrical engineering from the University of Illinios, Urbana–Champaign in 1988 and the M.S. and Ph.D. degrees in computer science from the University of California, Irvine in 1990 and 1994, respectively, where he was an SRC Fellow.

He is currently an Assistant Professor in the Department of Computer Science at the University of California, Riverside. His research interests include embedded-system specification and design, hardware/software codesign, and behavioral synthesis.



Peter Fung (S'88–M'94) received the B.S. degree in electrical engineering and computer science from the University of California at Berkeley in 1990 and the M.S. degree in electrical and computer engineering from Carnegie Mellon University in 1992.

He is currently a research engineer at Matsushita Electric Works Research & Development Laboratory, San Jose, CA. His research interest is in the area of electronic design automation with emphasis on embedded systems, behavioral synthesis, and rapid prototyping.