# The SpecSyn Design Process and Human Interface

Daniel Gajski
Jie Gong
Frank Vahid
Sanjiv Narayan

Technical Report #93-3

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-8059

Email : narayan@ics.uci.edu

## Abstract

*This report describes a presentation on the design methodology and the user's view of the SpecSyn system design framework. Given an abstract specification of a system, we present specification capture and the subsequent refinements that will result in synthesizable descriptions. The advantages of the underlying methodology compared to current approaches are highlighted.*

# Contents


**1 Introduction** **2**

1.1 Current Design Methodology . . . . . 4

1.2 Proposed Design Methodology . . . . . 6

**2 System Design Tasks** **8**

**3 System Design of an Audio-Video Processor** **10**

3.1 Capturing the Conceptual View . . . . . 10

3.2 Main Design Display . . . . . 12

3.3 Allocating Hardware / Software Modules . . . . . 14

3.4 Main Display after Allocating HW / SW Modules . . . . . 16

3.5 Partitioning Behaviors into Hardware & Software . . . . . 18

3.6 Main Display after Hardware/Software Partitioning . . . . . 20

3.7 Closeness based Partitioning . . . . . 22

3.7.1 Variable Partitioning . . . . . 22

3.7.2 Behavioral Partitioning . . . . . 22

3.7.3 Channel Partitioning . . . . . 22

3.8 Closeness-based Variable Partitioning . . . . . 24

3.9 Iterative-based Variable Partitioning . . . . . 26

3.10 System Modules after Partitioning . . . . . 28

3.11 Binding Generic Memories to Library Components . . . . . 30

3.12 Main Display after Allocation/Partitioning . . . . . 32

3.13 Refinement : Hardware/Software Interfacing . . . . . 34

3.14 Refinement : Arbiter Process Generation . . . . . 36

3.15 Refinement : Protocol Selection . . . . . 38

3.16 Refinement : Interface Process Generation . . . . . 40

3.17 System Design Output . . . . . 42

**4 Conclusions** **44**

**5 Acknowledgements** **46**

**6 References** **46**

# 1 Introduction

As design tools and methodologies become more advanced and reliable, they can be applied to tasks at progressively higher levels of abstraction in the design process, replacing ad hoc approaches. Previously, the highest-level tools addressed the task of transforming a behavioral description of a system hardware module into an equivalent register-transfer level structure, usually assumed to be implemented on a single chip, through a process called high-level synthesis. Prior to applying such tools, the system functionality must be divided among a set of system modules, where each module may represent a processor, a microcontroller, an ASIC, a memory, a predesigned chip or component, a block on a chip, and so on. These modules have a well-defined set of interconnections. We define *system design* to be the set of tasks that map system functionality to system modules [1]. High-level synthesis can be applied to obtain implementations of a subset of the modules which represent hardware, while other modules require different implementation techniques such as compilation.

Our motivation for developing a system design methodology and tool is to deal adequately with the rapidly changing implementation technologies and the growing use of different technologies in a single system. Such developments require an organized and well-documented approach that includes rapid automated estimations of the relative quality of alternative mappings of functionality to modules. Also necessary is the ability to design each module concurrently such that integration of these modules requires only minimal time.

## What is SYSTEM DESIGN?

*" Given a conceptual view of the system's functionality, SYSTEM DESIGN is the set of tasks which will produce a set of completely specified interconnected MODULES implementing that functionality. "*

Module – hardware, software, chip, block on a chip, memory

## Motivation

Systems are getting increasingly complex

- several chips,
- several technologies,
- HW/SW components, etc.

Copyright (c) 1993
UC Irvine CADLAB

## 1.1 Current Design Methodology

There are three main stages in the current design methodology.

In the first stage, a conceptual view of the system's desired functionality is developed. This view is sometimes captured as an English specification.

In the second stage, the system design stage, the functionality is mapped to a set of system modules. A system designer selects modules and maps functionality through an informal process relying on past experience and mental or "back-of-the-envelope" estimations of design characteristics such as cost, performance, size, and power. The resulting modules are usually represented as a block diagram and the functionality of each module is defined with English or with a behavioral specification language.

In the last stage, an implementation is realized for each of the system modules using various design tools. At the end of this stage each module is represented as a netlist, a layout, software, or as a library component.

The current approach has several disadvantages:

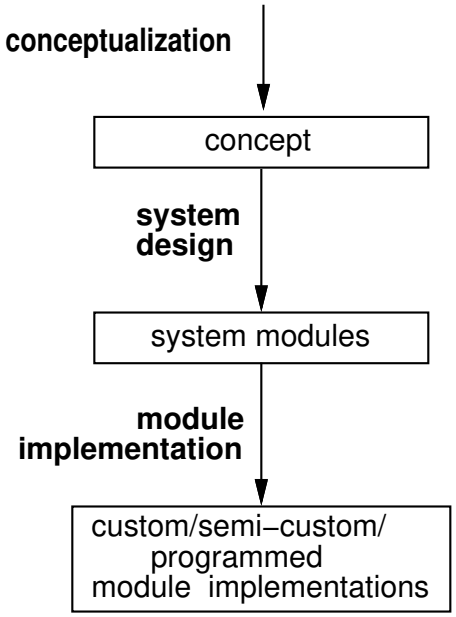
1. **High occurrence of late functionality modifications**: System functionality is tested, via simulation or prototyping, only after obtaining the various modules. At this stage it is often determined that the functionality is incorrect or incomplete, but it is more difficult to make major changes or additions to the system's functionality at this stage than it is at the conceptualization stage. Minor changes can be made, but sweeping changes require repeating the system design step.

2. **Delayed feedback on design decisions**: Most of the design decisions are based on previous design experience and "rules of thumb". It is possible to estimate the design parameters accurately only after performing system design. The lack of design metrics during system design introduces a significant delay before the effects of any design decision can be gauged.

3. **Incomplete search of design space**: Due to the long time required for any qualitative feedback on design decisions, system design is often performed without examining the many options available at each design step.

4. **Lack of system-design documentation**: The only documentation usually consists of the module descriptions, from which it can be difficult to understand the desired system functionality and the system design decisions. This lack of documentation makes redesign of the system for another application difficult, and personnel changes can have disastrous consequences.

# Current Methodology




| | *Design Representation* | *Tools* | *Design Evaluation* |
|---|---|---|---|
| conceptualization | English | | |
| system design | | Manual | Manual |
| module implementation | Specification Language or English + Block Diagrams | Manual, High–Level Synthesis, Logic Synthesis, Physical Design | Estimators, Simulators, Lib. components, Layouts |
| custom/semi–custom/ programmed module implementations | netlists, layouts, C code | | |

**Disadvantages :**

- Too many decisions without complete specification.
- Feedback on system design decisions takes too much time.
- Incomplete search of design space.
- Lack of documentation.

Copyright (c) 1993
UC Irvine CADLAB

## 1.2 Proposed Design Methodology

In order to address some of the shortcomings inherent in the current methodology, we propose an alternative approach to system design.

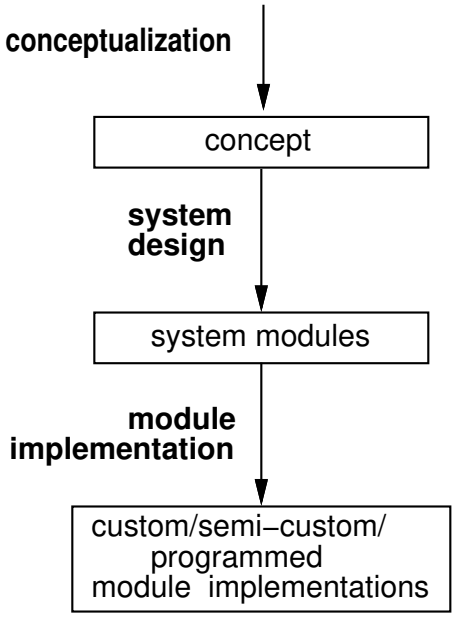
In the proposed methodology [2], the conceptual view of system functionality is first captured using an executable specification language, such as VHDL [3, 4] or SpecCharts [1, 5, 6, 7]. The specification language should have the capability of capturing the conceptual view with minimal designer effort. By using an executable language [8, 9], the completeness and correctness of the system's functionality can be ascertained before any design has been performed, so changes are made early and thus are not difficult.

A system-design tool is then used to present an abstract design view that allows the designer to allocate system modules and partition functionality among those modules. Each such mapping is evaluated by *automated estimators* [10, 11], providing rapid feedback on the relative quality of alternative solutions. Once a satisfactory design is found, the original system specification is refined into a set of module specifications. To further reduce the workload of the designer, the tasks of module allocation, partitioning, and refinement can be automated by various tools [12, 13, 14].

Once the system modules have been completely specified as a result of system design tasks, we can use software generators and hardware design compilers [15, 16] to obtain a software/hardware implementation of each module.

By capturing desired functionality with an executable specification language and using automated estimators, the disadvantages of the current methodology are overcome.

# Proposed Methodology




| | *Design Representation* | *Tools* | *Design Evaluation* |
|---|---|---|---|
| conceptualization | **Specification Language** | | |
| system design | | **Partitioning, Allocation, Refinement** | **Estimators, Simulation** |
| module implementation | Specification Language or English + Block Diagrams | Manual, High–Level Synthesis, Logic Synthesis, Physical Design | Estimators, Simulators, Lib. components, Layouts |
| | netlists, layouts, C code | | |

**Advantages :**

- Functionality can be verified in initial stages
- Rapid feedback on design decisions using estimators
- Good documentation. Less RE–design time.
- Possibility of automation.

UC Irvine

Copyright (c) 1993
UC Irvine CADLAB

## 2 System Design Tasks

We now examine several of the tasks which constitute system design. We start with a desired functionality described with a specification language. There are three classes of abstract *functional* objects:

1. **Variables** represent all objects in the specification which represent data. For example, in a VHDL specification, the variable class of objects includes declared signals and variables, including both composite (array or record) or scalar types.

2. **Behaviors** represent *chunks* of computations in the specification. VHDL processes and procedures are considered behaviors. A variable is also considered a behavior whose functionality is the maintenance of data values.

3. **Channels** represent communications between behaviors. For example, in VHDL, a process accessing a global array defines an abstract communication channel between the process and the array. Two behaviors communicating over ports or global signals also defines a communication channel.

The goal of system design is to map these *functional objects* to a set of system-level *structural objects* such as processors, chips, blocks on a chip, memories and buses. There are three tasks required to achieve this:

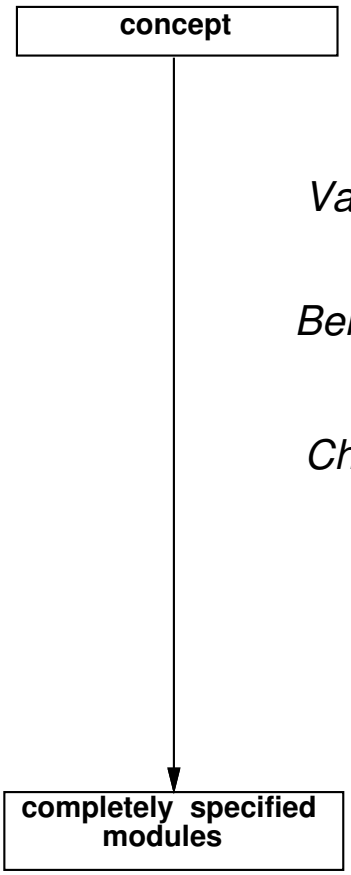
1. **Allocation**: This task instantiates a structural object such as a module or a bus. A module may represent (i.e. it is bound to) an actual library module, such as the Intel 8086 processor or a Xilinx FPGA with 10,000 gates and 40 pins. Or, it may represent a generic module, such as a processor with 3-address instructions and a 10MHz clock, or even more general modules such as hardware or software.

2. **Partitioning**: This task partitions, i.e. maps, the functional objects among the structural objects. This mapping is rarely one to one. Instead, multiple functional objects are mapped to a single structural object to improve some design metric such as area, interconnectivity or performance. For example, variables can be mapped to a single memory to minimize interconnections in the hardware implementation. Behaviors can be mapped to software or hardware. Behaviors mapped to hardware can be further partitioned to represent chips or a blocks on a chip. Software mapped behaviors can be partitioned among processors. Finally, channels can be mapped to a single bus.

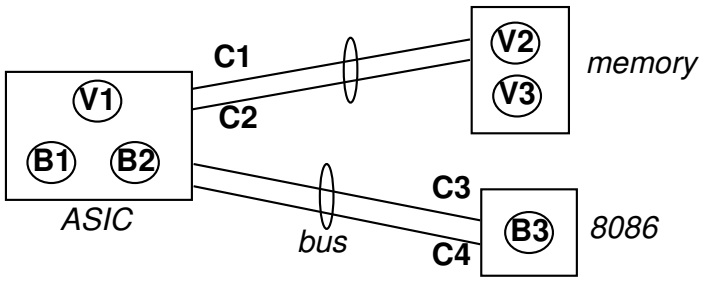
3. **Refinement**: After a satisfactory mapping of functional to structural objects has been achieved, a refined specification can be generated to serve as input to the next level of design or to perform simulations. Refinement tasks ensure that the specification is consistent after the application of the allocation and partitioning tasks. For example, mapping variables to a single memory requires memory address translation, where each variable is assigned a specific memory location and each reference to that variable is replaced by a memory access. Behavioral grouping requires that the communication be maintained by inserting appropriate communication channels between groups and selecting protocols to implement the data transfer. Channels which are grouped together to form a bus may require arbiters to resolve access conflicts.

In the figure, variable `V1` and behaviors `B1` and `B2` have been mapped to custom hardware. Variables `V2` and `V3` have been mapped to a memory, and behaviors `B2` and `B3` have been mapped for execution on an Intel 8086 processor. Channels `C3` and `C4` have been combined into a bus.

# System Design Tasks




| | *Allocation* | *Partitioning* | *Refinement* |
|---|---|---|---|
| *Variables* | Storage<br>Memories, Reg. Files | Variables to Storage | Memory addr. translation |
| *Behaviors* | Modules<br>HW–SW, Processors, Components, Chips, Blocks | Behaviors to Modules | Interfacing, Sequentialization |
| *Channels* | Buses | Channels to Buses | Arbiter Insertion<br>Protocol Merging |



Copyright (c) 1993
UC Irvine CADLAB

# 3 System Design of an Audio-Video Processor

## 3.1 Capturing the Conceptual View

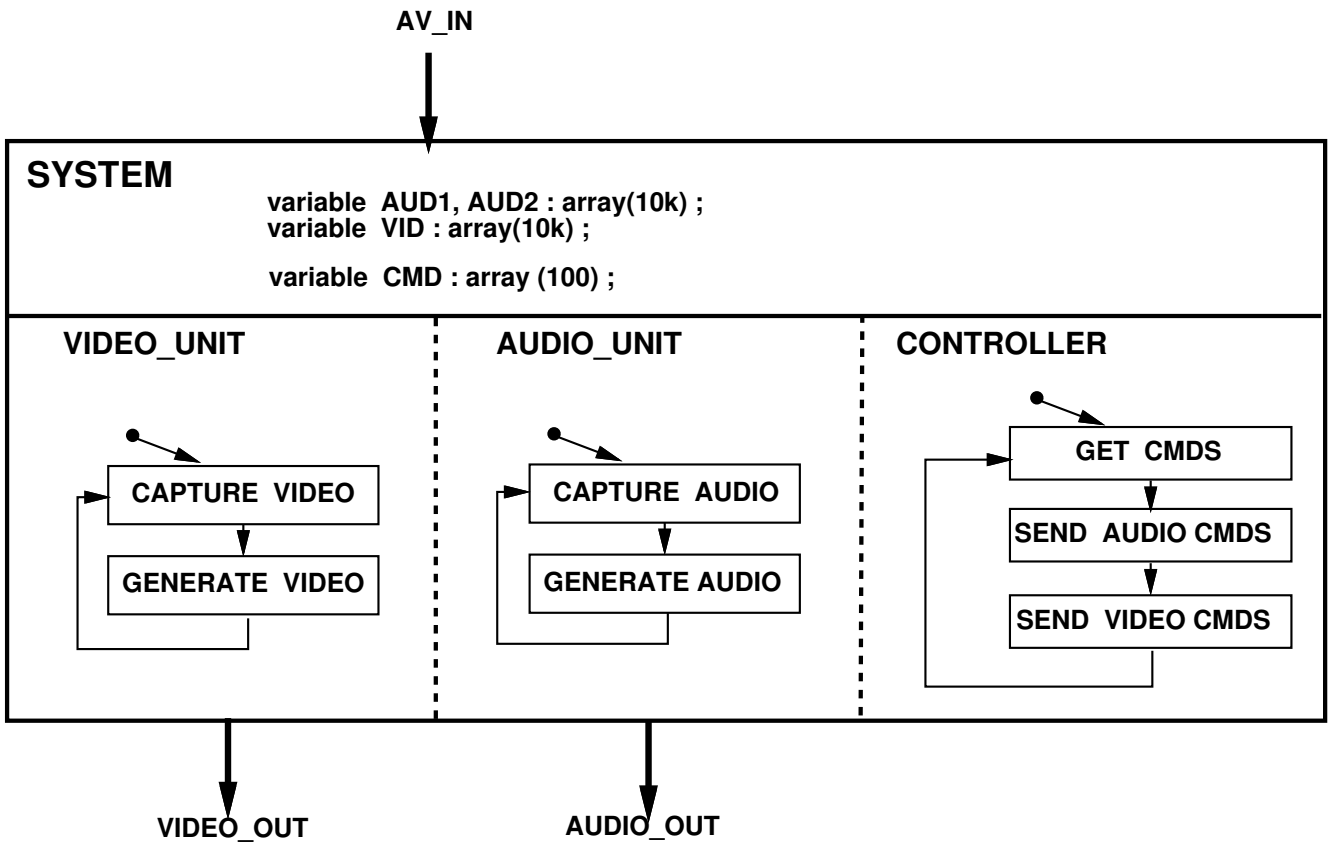
To see how the various system design tasks can enable us to transform a conceptual view into a set of well-defined modules, we will take a simple example of an audio-video (AV) processor.

The first step in system design is capturing the conceptual view of the system using a specification language. The figure shows the specification of the AV processor. The AV processor receives a stream of audio and video samples over the AV_IN bus, 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 buses AUDIO_OUT and VIDEO_OUT.

The AV processor consists of a hierarchy of behaviors of which only the top three levels are shown in the figure. 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. SYSTEM consists of three concurrent sub-behaviors: AUDIO_UNIT, VIDEO_UNIT and CONTROLLER. 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. The behavior AUDIO_UNIT further consists of two sequential sub-behaviors: CAPTURE_AUDIO and GENERATE_AUDIO. Details of the behaviors such as CAPTURE_AUDIO, CAPTURE_VIDEO, GET_COMMANDS etc. are also specified (not shown in the figure). A behavior at any level of the hierarchy of these behaviors can further consist of sequential/concurrent sub-behaviors or can be specified using VHDL sequential statements.

It must be emphasized here that the specification shown in the figure is one of many possible conceptual views of the AV processor. In this instance, there are three concurrent activities in the AV processor, and thus the corresponding specification has three concurrent behaviors: AUDIO_UNIT, VIDEO_UNIT and CONTROLLER. The AUDIO_UNIT consists of two sequential activities and these are represented as two sequential sub-behaviors: CAPTURE_AUDIO and GENERATE_AUDIO. Different system designers could visualize the same system in different ways.

# Specification of the AV Processor




UC Irvine
Copyright (c) 1993
UC Irvine CADLAB

## 3.2 Main Design Display

In this section, we present the main display of a tool which assists a designer perform system design tasks using our methodology. The main display consists of two windows: *Display Window* and *Command Window*.

**Display Window**

The Display Window displays the current state of the design. It lists all the objects in the design, i.e. variables, behaviors, channels derived from the specification, and their hierarchical groupings into modules and buses. The hierarchy is shown using indenting. In the figure, we see that initially the top object is the behavior `SYSTEM` comprised of the three sub-behaviors `AUDIO_UNIT, VIDEO_UNIT` and `CONTROLLER` and the four variables `AUD1, AUD2, VID` and `CMD`.

Associated with each object are estimations and constraints on object attributes such as size, cost, and performance. There is one column for each attribute, with an estimate/constraint entry for each object. Entrys are updated after each design task. In the example, the dollar cost for the entire `SYSTEM` is constrained to be less than $50 and the execution time of the behavior `AUDIO_UNIT` has a constraint of 40 time units. Since no modules have been allocated yet, no estimates exist.

Using the *Modify Display* button, the designer can select which portions of the object hierarchy and which attribute columns to display. The *Views* option enables the designer to display the objects sorted according to some combination of the attribute values. The *Cost Function* option allows the designer to select, view and edit cost functions which can be used during the various design tasks to compare alternative designs.

**Command Window**

The Command Window enables the designer to perform system design tasks. Having selected a class of objects, the designer can *allocate* modules and *partition* objects, each of which can be done in a *manual* or *automated* manner. To guide manual decisions, a variety of *hints* can be examined. A refined specification can be generated by applying the required *refinement* tasks. The various commands are examined in more detail later.

# Main Display

**Display Window** **Modify Display** **Views** **Cost FN** Overall Cost: ___

| | | Estimates / Constraints | | | | | |
|---|---|---|---|---|---|---|---|
| *Groups/Objects* | *Binding* | *$Cost* | *Pins* | *Size(hw)* | *Size(sw)* | *Perf(hw)* | *Perf(sw)* |
| **SYSTEM** | – | –/50 | 83 | – | – | – | – |
| **AUDIO_UNIT** | – | – | – | – | – | –/40 | –/40 |
| **CAP_AUDIO** | – | – | – | – | – | –/20 | –/20 |
| **GEN_AUDIO** | – | – | – | – | – | –/20 | –/20 |
| **VIDEO_UNIT** | – | – | – | – | – | –/20 | –/20 |
| **CAP_VIDEO** | – | – | – | – | – | –/10 | –/10 |
| **GEN_VIDEO** | – | – | – | – | – | –/10 | –/10 |
| **CONTROLLER** | – | – | – | – | – | –/20 | –/20 |
| **GET_CMDS** | – | – | – | – | – | –/10 | –/10 |
| **SEND _AUD_CMD** | – | – | – | – | – | –/5 | –/5 |
| **SEND_VID_CMD** | – | – | – | – | – | –/5 | –/5 |
| **AUD1** | – | – | – | – | – | – | – |
| **AUD2** | – | – | – | – | – | – | – |
| **VID** | – | – | – | – | – | – | – |
| **CMD** | – | – | – | – | – | – | – |

**Command Window**

| *Object class* | *Partitioning* | *Allocation* | *Refinement* |
|---|---|---|---|
| **Variables** | **Automated** | **Automated** | **Arbiter Gen** |
| **Behaviors** | **Manual** | **Manual** | **Protocols** |
| **Channels** | **Hints** | **Hints** | **Interface** |
| | **Backtrack** | **Backtrack** | **Backtrack** |

Copyright (c) 1993
UC Irvine CADLAB

### 3.3 Allocating Hardware / Software Modules

Suppose that the first system design tasks we wish to perform is the division of the functionality into software and hardware portions. A hardware implementation yields fast performance, but software has the advantages of low cost, shorter development time and ease of change late in the design cycle.

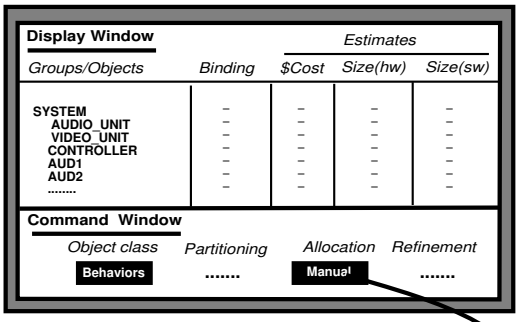
We first allocate two new modules named `HW` and `SW`. In order to obtain estimations for evaluating future partitions, the design tool must be informed as to what each module represents. In other words, each module possesses a set of attributes whose values are necessary for estimation purposes. An example of an attribute is the clock cycle time of a microprocessor. A module's attributes can be defined in one of two ways:

1. **Binding to a Library Component**: A module which is bound to a library component inherits all the attributes of that component. For example, the designer may bind the software module to the Intel 8086 microprocessor, which then associates the attributes of that microprocessor with the module, such as its clock cycle time.

2. **Defining a Generic Module**: Often, it may not be possible or desirable to specify exactly which library component is to be used to implement a module. For example, the designer may just want to bind the group of software behaviors to be implemented on a microprocessor with a 25 MHz clock and 16 bit address/data buses. In this case, a *generic module* is created and the designer assigns the desired values to the relevant module attributes. The estimators can now use this information to determine, for example, how many clock cycles a procedure will require to execute on that generic microprocessor component. Later on the designer may replace the generic module with a standard component or may actually design a new component possessing the specified attributes.

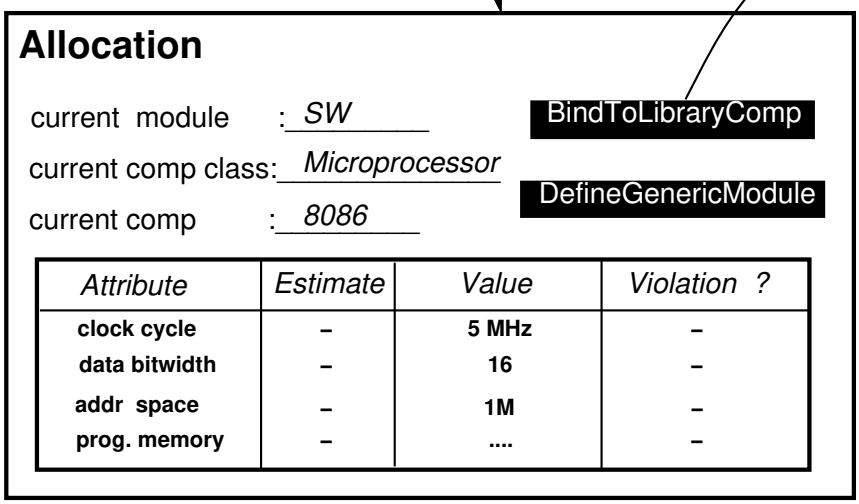
Let's assume that we wish to manually bind the software module to the 8086 microprocessor. From the *Command Window* the designer selects *Allocation => Manual*, which brings up the Allocation Display. The designer enters the name of the module to be allocated, in this case `SW`). The option *BindToLibraryComp* allows selection of the desired microprocessor from a list. The attributes for the currently selected component in the list are shown in the Allocation Display. For each attribute, the estimated values of that attribute is displayed in the *Estimate* column. The *Value* column lists the attribute values associated with the selected component. A *violation* (listed in the *Violation* column) is said to occur whenever the estimated required value (based on an existing mapping) of an attribute exceeds the corresponding value associated with the selected component. In this case since the module `SW` is empty, no estimates (and hence no violations) exist for the module. In case a generic module was being defined using *DefineGenericModule*, the designer would enter the desired values of the attributes associated with that generic module in the *Value* column. Similar to the software module, let's assume that the designer binds the `HW` module to a particular ASIC.

# Allocation: Hardware/Software Modules

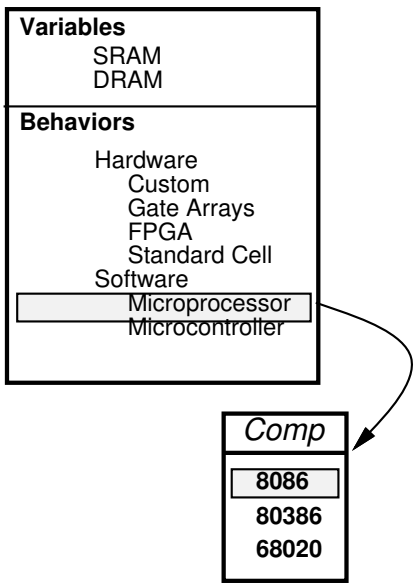



**Hardware** – Faster Performance

**Software**
- Cheaper to Implement
- Faster Development
- Late Specification Changes

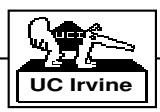



| Attribute | Estimate | Value | Violation ? |
|---|---|---|---|
| clock cycle | – | 5 MHz | – |
| data bitwidth | – | 16 | – |
| addr space | – | 1M | – |
| prog. memory | – | .... | – |










Copyright (c) 1993
UC Irvine CADLAB

### 3.4 Main Display after Allocating HW / SW Modules

We have now allocated two empty modules, `HW` and `SW`. These are reflected in the Main Display as two new objects in the list of objects. Estimates are displayed for the performance of each behavior in `HW` and in `SW`, and for the size in `HW` (area) and in `SW` (instructions and memory).

# Updated Main Display

( after allocating two modules, SW & HW )

**Display Window** **Modify Display** **Views** **Cost FN** Overall Cost: ___

| *Groups/Objects* | *Binding* | *Estimates / Constraints* $Cost | *Pins* | *Size(hw)* | *Size(sw)* | *Perf(hw)* | *Perf(sw)* |
|---|---|---|---|---|---|---|---|
| **SYSTEM** | – | –/50 | 83 | 12000 | 1260 | – | – |
| **AUDIO_UNIT** | – | – | – | 3000 | 380 | 33/40 | 50/40 |
| **CAP_AUDIO** | – | – | – | – | – | 18/20 | 28/20 |
| **GEN_AUDIO** | – | – | – | – | – | 15/20 | 22/20 |
| **VIDEO_UNIT** | – | – | – | 4000 | 200 | 15/20 | 45/20 |
| **CAP_VIDEO** | – | – | – | – | – | 10/10 | 20/10 |
| **GEN_VIDEO** | – | – | – | – | – | 5/10 | 25/10 |
| **CONTROLLER** | – | – | – | 1000 | 680 | 10/20 | 18/20 |
| **GET_CMDS** | – | – | – | – | – | 6/10 | 9/10 |
| **SEND _AUD_CMD** | – | – | – | – | – | 2/5 | 4/5 |
| **SEND_VID_CMD** | – | – | – | – | – | 2/5 | 5/5 |
| **AUD1** | – | – | – | 2000 | – | – | – |
| **AUD2** | – | – | – | 1000 | – | – | – |
| **VID** | – | – | – | 1000 | – | – | – |
| **CMD** | – | – | – | 200 | – | – | – |
| **SW** | 8086 | 10 | 40 | – | – | – | – |
| **HW** | Asic XX | 30 | 50 | – | – | – | – |

**Command Window** *Object class* ..... *Partitioning* ..... *Allocation* ..... *Refinement* .....

Copyright (c) 1993
UC Irvine CADLAB

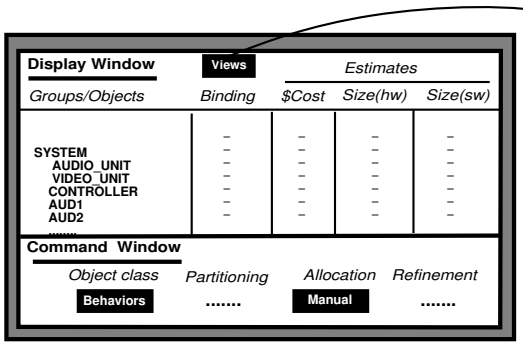
### 3.5 Partitioning Behaviors into Hardware & Software

We now consider one of three ways to partition functional objects into implementation modules.

Suppose we wish to implement all behaviors in software except those which require hardware to meet performance constraints. The simplest way to achieve this is to list the objects for which the `SW` performance estimate exceeds the constraint. From the *Display Window* the designer can select *Views* option to list all objects ordered by a weighted function of their attribute values. The performance constraint violation is the difference between the estimated software performance and the constraint specified for each of the behaviors. This is achieved by assigning a weight of 1 and -1 to the attributes *SW Performance* and *Performance Constraint* in the View Objects Display. The behaviors are listed in the View Objects Display in descending order of the values computed for the selected attributes.

We can see that the behaviors `VIDEO_UNIT, CAP_VIDEO, GEN_VIDEO, AUDIO_UNIT, CAP_AUDIO` & `GEN_AUDIO` will violate their performance constraints if they are implemented as software. These behaviors are thus implemented as hardware by mapping to the module `HW` created earlier. The remaining behaviors `CONTROLLER, SEND_AUD_CMDS, SEND_VID_CMDS` and `GET_CMDS` will meet the performance constraints if implemented as software and are thus mapped to the module `SW`. Mapping is done in a straightforward manner by using the *Partitioning => Manual* command.

# Partitioning : Behaviors into HW, SW




## Grouping Method 1

*Grouping based on attributes*

1. Select Attributes.
2. List objects based on attributes
3. Move objects to modules.

## View Objects

*Attribute Select*

| Attribute | Weight |
|---|---|
| Size | 0 |
| SW Perf | 1 |
| Perf Constraint | –1 |
| ..... | .. |

List Objects

*Objects List*

| Objects | Value |
|---|---|
| **VIDEO_UNIT** | **25** |
| **GEN_VIDEO** | **15** |
| **CAP_VIDEO** | **10** |
| **AUDIO_UNIT** | **10** |
| **CAP_AUDIO** | **8** |
| **GEN_AUDIO** | **2** |
| SEND_VID_CMDS | 0 |
| GET_CMDS | –1 |
| SEND_AUD_CMDS | –1 |
| CONTROLLER | –2 |

*hardware*

UC Irvine

Copyright (c) 1993
UC Irvine CADLAB

### 3.6 Main Display after Hardware/Software Partitioning

The updated Main Display is shown in the figure. AUDIO_UNIT and VIDEO_UNIT have been moved into HW. The behavior CONTROLLER and its sub-behaviors have been moved into SW. The corresponding estimates for the modules HW and SW are also updated. Thus, the area estimate of the module HW is 7000 units which is the sum of the areas of AUDIO_UNIT and VIDEO_UNIT assigned to the module.

# Updated Main Display

( after moving behaviors into Software and Hardware modules )

**Display Window** | **Modify Display** | **Views** | **Cost FN** | Overall Cost: ___

| *Groups/Objects* | *Binding* | *Estimates / Constraints* $Cost | Pins | Size(hw) | Size(sw) | Perf(hw) | Perf(sw) |
|---|---|---|---|---|---|---|---|
| **SYSTEM** | – | –/50 | 83 | 12000 | 1260 | – | – |
| **HW** | Asic XX | 30 | 42/50 | 7000 | – | – | – |
| **AUDIO_UNIT** | – | – | 32 | 3000 | 380 | 33/40 | 50/40 |
| **CAP_AUDIO** | – | – | – | – | – | 18/20 | 28/20 |
| **GEN_AUDIO** | – | – | – | – | – | 15/20 | 22/20 |
| **VIDEO_UNIT** | – | – | 16 | 4000 | 200 | 15/20 | 45/20 |
| **CAP_VIDEO** | – | – | – | – | – | 10/10 | 20/10 |
| **GEN_VIDEO** | – | – | – | – | – | 5/10 | 25/10 |
| **SW** | 8086 | 10 | 40 | 1000 | – | – | – |
| **CONTROLLER** | – | – | – | – | 680 | 10/20 | 18/20 |
| **GET_CMDS** | – | – | – | – | – | 6/10 | 9/10 |
| **SEND _AUD_CMD** | – | – | – | – | – | 2/5 | 4/5 |
| **SEND_VID_CMD** | – | – | – | – | – | 2/5 | 5/5 |
| **AUD1** | – | – | – | 2000 | – | – | – |
| **AUD2** | – | – | – | 1000 | – | – | – |
| **VID** | – | – | – | 1000 | – | – | – |
| **CMD** | – | – | – | 200 | – | – | – |

**Command Window** | *Object class* ..... | *Partitioning* ..... | *Allocation* ..... | *Refinement* .....

Copyright (c) 1993
UC Irvine CADLAB

## 3.7 Closeness based Partitioning

Another way to partition functional objects into implementation modules is through *closeness* computations. An object is said to be *close* to another object or group of objects if grouping them would result in a better design. Closeness measures usually rely on some indirect criteria that attempts to maximize the performance, satisfy capacity constraints or minimize interconnect. We discuss some of these criteria below.

### 3.7.1 Variable Partitioning

A variable is considered to be *close* to a group of variables, if it:

1. is *sequentially accessed* with respect to the variables already assigned to the group. This avoids performance degradation results from contention between concurrent accesses made to the variables in a memory.
2. *fits into the available space* in the module, thus satisfying any specified module capacity constraints.
3. is *accessed by the same behaviors* which access the other variables in the module. This allows sharing of interconnections between the behaviors and the variables they access.

### 3.7.2 Behavioral Partitioning

A behavior is considered to be *close* to a group of behaviors, if it:

1. *fits into the available capacity* in the module, thus satisfying any capacity constraints (such as chip area for hardware behaviors or program memory size for software behaviors).
2. *communicates frequently* with the behaviors assigned to the module. This leads to a better performance since access times for behaviors within the same module are faster than if an access is made to a behavior in another module.
3. is *accessed by the behaviors* already assigned to the module, thus minimizing interconnect between behavioral modules.

### 3.7.3 Channel Partitioning

A channel is considered to be *close* to a group of channels (a bus), if it:

1. is *sequentially accessed* over time with respect to the channels already assigned to the bus. This avoids performance degradation which would result if bus access conflicts were to occur.
2. *fits into the available channel bandwidth* in the bus, thus satisfying any constraints specified for the channel in terms of data transfer rates.
3. has *similar bitwidths* to the other channels assigned to the bus. This reduces the number of unused interconnection wires during communication.

# Partitioning : Variables, Behaviors, Channels

**Grouping Method 2**

*Closeness based Grouping*

**Closeness** : Measure of benefit of merging an object with another object(s)

| Objects | Why ? | Closeness Criteria |
|---|---|---|
| Variables | Minimize Interconnect | Fewer concurrent accesses<br>Fits into no. of words available<br>Accessed by same behaviors |
| Behaviors | Minimize Cost<br>Maximize Performance<br>Minimize Interconnect | Fits into available capacity<br>Higher communication frequency<br>Connected to same behaviors |
| Channels | Minimize Interconnect | Used exclusively<br>Fits into available channel bandwidth<br>Similar bitwidths |

### 3.8 Closeness-based Variable Partitioning

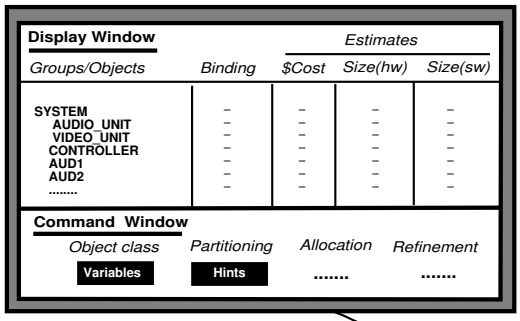
Let's assume we want to map variables into memories using the closeness-based partitioning technique. Similar to the software/hardware partitioning example, we allocate memory modules. We shall start by using generic memories, so we specify the attributes of each such as size and access times. Suppose we create three empty memories `M1, M2, M3`. We need to first seed the memories. We assume that the audio variable `AUD1` is assigned to `M1`. We wish to determine the variable that is the *closest* to the group `M1`. We select *Partitioning => Hints => Closeness* which brings up the Closeness Display.

The designer can select and weigh the various criteria to compute a closeness of each of the unassigned variables to the group `M1`. In the figure, the criteria selected are number of concurrent accesses and the number of common behaviors with respect to the variables in `M1`. The Closeness Display lists all the variables ordered by their closeness to `M1`. Based on the criteria selected, we see that `AUD2` is the closest to `M1`, followed by `CMD` and `VID`.

The designer can specify certain *match patterns* to restrict the objects that are listed. In the figure, the designer has specified that all variables that have a closeness measure less than 250 are to be listed.

Based on the closeness list, the designer can move variables to module `M1` using the manual partitioner. Let's say the designer selects `CMD` as the variable that will be assigned `M1`. Thus `M1` now has the variables `AUD1` and `CMD` in it. The process can be repeated to find which variables, if any, can be assigned to `M1`. The designer can also select variables to be moved to the other modules in a similar manner.

# Partitioning : Variables to Memories




## Grouping Method 2

*Closeness based Grouping*

1. Allocate memories
2. Seed a memory
3. Select Closeness Criteria
4. Move variables to the memory based on closeness

## Closeness

*Select Closeness Criteria*

| Criteria | Weight |
|---|---|
| Conc. Access | 3 |
| Size Fit | 0 |
| Common Behavior | 2 |

List

*Closeness to Group :* M1

| Object | Closeness |
|---|---|
| AUD2 | 10 |
| CMDS | 100 |
| VID | 200 |

Select

*Match Pattern*

| * | < 250 |
|---|---|

UC Irvine

Copyright (c) 1993
UC Irvine CADLAB

### 3.9 Iterative-based Variable Partitioning

A third way to partition functional objects into implementation modules is to iteratively improve an existing partitioning by moving objects between modules, guided by a global cost function.

Let's assume that as a result of variable partitioning based on closeness, the following groups were created:

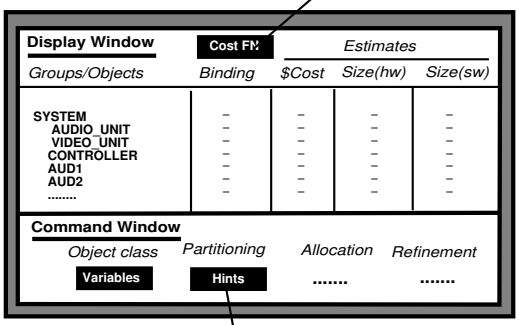
```
M1 : ( AUD1, CMD )
M2 : ( AUD2, VID )
M3 : (  )
```

The designer first selects the cost function which will be used to determine whether moving an object between groups results in any improvement. The improvement, or *gain*, is measured as a reduction in the cost function as a result of performing the set of moves. Selecting *CostFN* from the Main Display brings up the Cost Function Display. The designer can select the metrics that contribute to the overall design cost and can specify relative weights and constraints for them. For example, in the Cost Function Display in the figure, the designer has selected the size of variable group `M2`, specified a relative weight of 2 for the metric and constrained it to be 15K words. The current values of the metric are computed by the built-in estimators and displayed in the fourth column. The difference between the estimated value and constraints is listed in the fifth column as a bar graph, which gives the designer a good idea of which constraints are being violated and by how much. Thus in the figure, we can see that the constraints for *pin(HW)* and *size(M2)* are being violated.

Having selected the cost function, selecting *Partitioning => Hints => Iterative* brings up the Object Move Display. The designer can specify the number of objects he wishes to reassign among the groups. In the figure, the designer has selected the number of moves to be examined as 2. The tool will examine all sets of 2 moves between groups and list them in decreasing order of achievable gain. Thus, moving `AUD2` to `M1` and `CMD` to `M3` results in the maximum gain of 100 in the cost function. These two moves are made resulting in `AUD1` and `AUD2` being grouped together in `M1` and `VID` and `CMD` are assigned to the groups `M2` and `M3` respectively.

The designer can use match patterns to restrict the moves that are examined and displayed. For example, to find the best group to move `AUD2` into, the designer can set the match pattern for Object to `AUD2` and set the number of moves to 1. To find out which three objects can be moved into the group `M3`, the designer set the match pattern for Destination to `M3` and set the number of moves to 3.

# Partitioning : Variables into Memories




## Grouping Method 3

*Iterative Grouping*

1. Allocate memories, create initial partitioning
2. Select Cost Function
3. Move variables between memories based on cost improvement

## Object Move

*No. of Moves* 2 List

| Object | Destination | Gain |
|---|---|---|
| AUD2 CMD | M1 M3 | 100 |
| AUD2 VID | M1 M3 | 12 |
| *Match Pattern* * | * | > 10 |

## Cost Function

| *metric* | *weight* | *constr.* | *est.value* | *violation* |
|---|---|---|---|---|
| pins (HW) | 3 | 40 | 53 | |
| size (M2) | 2 | 15k | 20k | |
| size (M1) | 1 | 20k | 10.1k | |
| .... | .. | .. | .. | |
| | | | **Total: 310** | |

Copyright (c) 1993
UC Irvine CADLAB

### 3.10 System Modules after Partitioning

Allocation and partitioning can be performed for Behaviors and Channels as done for variables above. The final result is a set of modules and buses as shown in the figure.

- **Variables:** The final memory allocation and variable partitions were:

```
M1 : ( AUD1, AUD2 )
M2 : ( VID )
M3 : ( CMD )
```

- **Behaviors:** We have partitioned the behaviors into two modules `HW` and `SW`. To satisfy area constraints, the set of behaviors in `HW` are further grouped into two sub-modules, `CHIP1` and `CHIP2`. The final allocation and behavior partitioning achieved was:

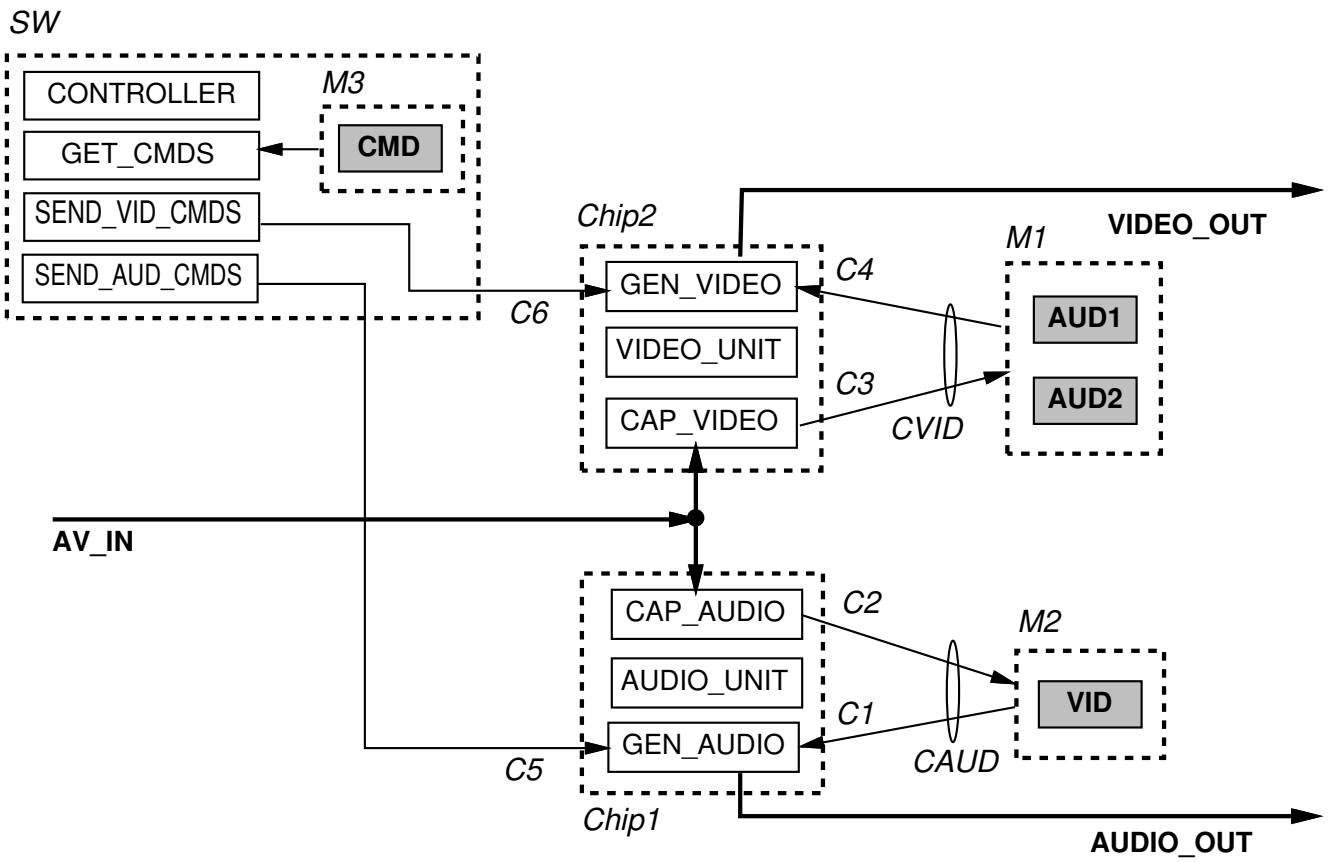
```
HW :
      CHIP1 : ( AUDIO_UNIT, CAP_AUDIO, GEN_AUDIO )
      CHIP2 : ( VIDEO_UNIT, CAP_VIDEO, GEN_VIDEO )

SW :  (CONTROLLER, GET_CMDS, SEND_VID_CMDS, SEND_AUD_CMDS)
```

- **Channel Groups** The channels `C1` and `C2` are mutually exclusive over time and have been merged to form the channel group `CAUD`. Similarly, channels `C3` and `C4` are merged to form the channel group `CVID`. The final channel grouping obtained was:

```
CAUD : ( C1, C2 )
CVID : ( C3, C4 )
C5
C6
```

# System Modules




Copyright (c) 1993
UC Irvine CADLAB

### 3.11 Binding Generic Memories to Library Components

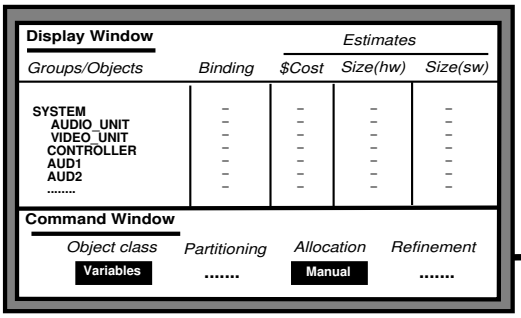
Recall that we initially allocated generic memories and partitioned variables among them. Now we wish to bind those memories to actual library memories. Clearly, binding to such an existing component eliminates the need to design the memory. After binding, estimates are updated and will reflect more closely the eventual final design.

From the Main display, we select *Allocation => Manual* to bring up the Allocation Display. The designer enters the memory to be bound, say `M1`. Estimates for `M1` are computed and appear in the Estimate column, indicating the requirements of the memory. For example, we see that a memory of at least 20k words is required, with 16-bit words and an address width of 15 bits.

By selecting *BindToLibraryComp* the designer obtains a list of library memory components. Selecting a component updates the Values column with the attributes of the selected component. If an estimated requirement of the memory such as size is not met, a value will appear in the Violation column. We can see that binding `M1` to the library component *M102* results in a data bitwidth violation: 16 bits are required, but the component only has 12.

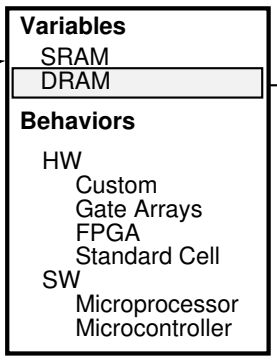
The designer can similarly bind modules containing behaviors to library components such as ASICs and microprocessors and bind buses to standard bus protocols.

# Component Binding




**Why ?**

- Reduce design time
- Provide better estimates




## Allocation

current object : *M1*

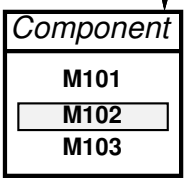
current comp class: *DRAM*

current comp : *M102*

BindToLibraryComp

DefineGenericModule

| *Characteristic* | *Estimate* | *Selected Comp.* | *Violation ?* |
|---|---|---|---|
| addr_width | 15 | 15 | – |
| data_width | 16 | 12 | 4 |
| num_words | 20K | 32K | – |
| word_width | – | – | – |



Copyright (c) 1993
UC Irvine CADLAB

### 3.12 Main Display after Allocation/Partitioning

After allocating system structural objects, some of which are bound to library components, and partitioning variables, behaviors and channels among those objects, the updated Main Display appears as shown in the figure.

The groupings were discussed earlier. It is of interest to note that binding the memory modules `M1` and `M2` to the library component *M10?* is reflected in the Binding column of the Display Window.

# Updated Main Display

( after partitioning variables, behaviors and channels among system structural objects )

**Display Window** | **Modify Display** | **Views** | **Cost FN** | Overall Cost: **1399**

| *Groups/Objects* | *Binding* | *Estimates / Constraints* $Cost | Pins | Size(hw) | Size(sw) | Perf(hw) | Perf(sw) |
|---|---|---|---|---|---|---|---|
| **SYSTEM** | – | 40/50 | 83 | 12000 | 1260 | – | – |
| **HW** | Asic XX | 30 | 42/50 | 7000 | – | – | – |
| **Chip1** | custom | – | 32 | 3000 | 380 | – | – |
| **AUDIO_UNIT** | – | – | – | – | – | 33/40 | 50/40 |
| **CAP_AUDIO** | – | – | – | – | – | 18/20 | 28/20 |
| **GEN_AUDIO** | – | | | | | 15/20 | 22/20 |
| **Chip2** | custom | – | 16 | 4000 | 200 | – | – |
| **VIDEO_UNIT** | – | – | – | – | – | 15/20 | 45/20 |
| **CAP_VIDEO** | – | – | – | – | – | 10/10 | 20/10 |
| **GEN_VIDEO** | – | – | – | – | – | 5/10 | 25/10 |
| **M1** | M102 | – | 32 | 2000 | – | – | – |
| **AUD1** | – | – | – | – | – | – | – |
| **AUD2** | – | – | – | – | – | – | – |
| **M2** | M102 | – | 32 | 2000 | – | – | – |
| **VID** | – | – | – | – | – | – | – |
| **CAUD** | – | – | 16 | – | – | – | – |
| **....** | – | – | – | – | – | – | – |
| **CVID** | – | – | 32 | – | – | – | – |
| **....** | ... | ... | ... | ... | ... | ... | ... |
| **SW** | 8086 | 10 | – | 1000 | 680 | – | – |
| **CONTROLLER** | – | – | – | – | 680 | 10/20 | 18/20 |
| **GET_CMDS** | – | – | – | – | – | 6/10 | 9/10 |
| **SEND _AUD_CMD** | – | – | – | – | – | 2/5 | 4/5 |
| **SEND_VID_CMD** | – | – | – | – | – | 2/5 | 5/5 |
| **M3** | – | – | – | 200 | – | – | – |
| **CMD** | – | – | – | – | – | – | – |

**Command Window** | *Object class* ..... | *Partitioning* ..... | *Allocation* ..... | *Refinement* .....

Copyright (c) 1993
UC Irvine CADLAB

### 3.13 Refinement : Hardware/Software Interfacing

After a satisfactory allocation and partitioning is found, the designer can begin creating a refined specification. We shall first consider adding details to represent the interface between the hardware and software modules.

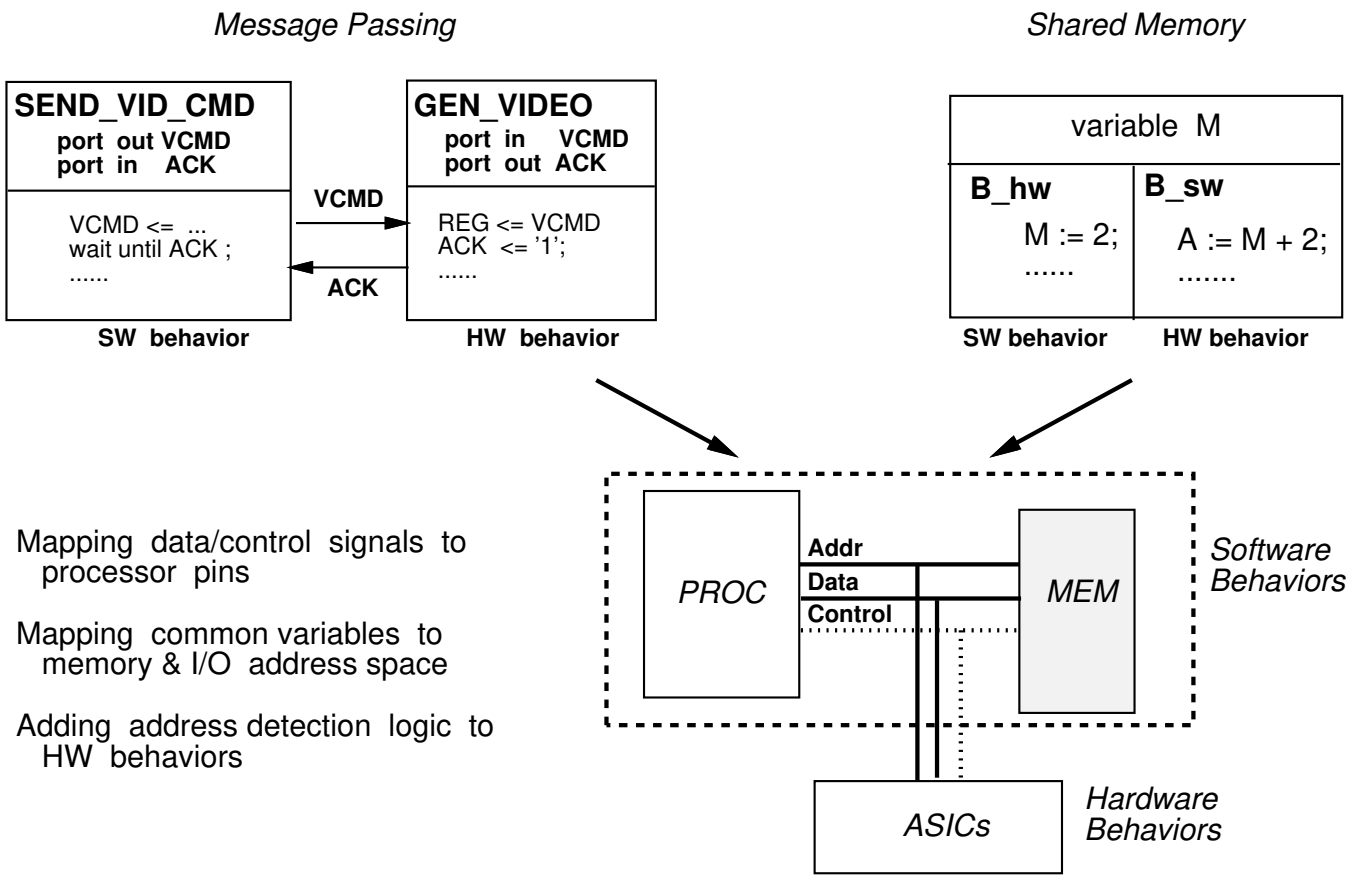
Communication between behaviors can be specified using the shared memory or message passing model. In the *shared memory* model, the behaviors communicate with each other by assigning and reading values to/from a global store such as a variable. The *message passing* model abstracts out the communication by viewing the communicating behaviors as being connected with *channels* over which the data or *message* is sent. An example of message passing is the behavior `SEND_VIDEO_CMDS` sending the video processing commands to the behavior `GEN_VIDEO` over channel `C6`.

Having partitioned the set of behaviors to be implemented as hardware or software, we need to implement the data transfers between them.

To implement the common variables shared between the behaviors, we have to determine whether the variable will be mapped to software (implemented as a location in the memory associated with the microprocessor on which the software behaviors will execute) or hardware. In either case, we have to determine the memory or I/O addresses for the common variables.

For message passing, the protocol associated with the communicating channel needs to be implemented. The data and control signals of the channel have to be mapped to the pins of the microprocessor or I/O addresses have to be allocated for them. In addition, we may need to add address detection capabilities to the hardware behaviors to enable them to detect microprocessor communication requests.

# Refinement : HW / SW Interfacing




- Mapping data/control signals to processor pins
- Mapping common variables to memory & I/O address space
- Adding address detection logic to HW behaviors

UC Irvine

Copyright (c) 1993
UC Irvine CADLAB

### 3.14 Refinement : Arbiter Process Generation

Mapping multiple channels to a single bus to reduce interconnect can result in bus access conflicts, i.e. two data transfers that take place over the bus at the same time. To resolve such conflicts, we must insert an arbiter process. Behaviors which wish to send data over the bus will assert a request signal, and the arbiter process will grant access rights (based on some priority scheme) using an acknowledge signal.

The channels in the group have to be assigned priorities which can based on several criteria:

1. *Channel Access Frequency*: Channels accessed very frequently should be assigned a higher priority.
2. *Communication Delay* : If the time required for each data transfer over a channel is very long, then that channel should be assigned a lower priority so as to minimize the average time that the channels have to wait to be granted access to the bus.
3. *Behaviors with performance constraint violations*: If a channel connects behaviors which have their performance constraints violated, the channel should be assigned a higher priority.

Let's assume that for some reason, the channels `CAUD` and `CVID` have been grouped together to form `AV_BUS`. Since the channels `CAUD` and `CVID` are used by the concurrent behaviors `AUDIO_UNIT` and `VIDEO_UNIT`, there may be access conflicts in sending data over `AV_BUS`.

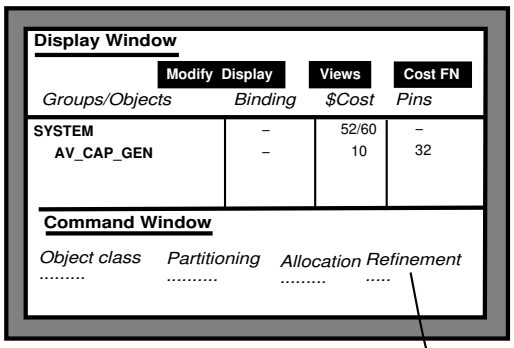
Selecting *Refinement => Arbiter Generation* from the Main Display will bring up the Arbiter Generation Display. The designer can list all the channel groups which have the possibility of access conflicts occurring. The Arbiter Display lists the channel group `AV_BUS` consisting of `CAUD` and `CVID` as having the potential for access conflicts. The designer selects the criteria and the associated weight that will be used to determine the priority of the channel. Based on these, the tool lists the constituent channels in decreasing order of the cost. Thus, the cost associated with the channel `CAUD` is 130.

Based on the cost, the designer can decide on the arbitration scheme that will be implemented. For example, if the costs of the channels in the group are approximately the same, then a *rotating priority* scheme may be selected. If however, the channels have different costs associated with them, then perhaps a *fixed priority* may be more suitable. In either case the designer orders the channels in the group by labeling them.

In the example, the designer has given the channel `CAUD` a higher priority ($Label = 1$) over the channel `CVID` ($Label = 2$). The designer may select the priority scheme. Let's say the designer selects the fixed priority scheme. The refinement tool will automatically do the following: (1) Updates the behaviors which send data over the channel by inserting request/acknowledge signals to communicate with the arbiter and (2) generate an arbiter process implementing the selected arbitration scheme which accepts requests and sends acknowledge signals to/from behaviors wishing to send data over the channels.

In case the designer wishes to implement his/her own arbitration scheme, the *Custom* option can be selected. In this case the appropriate request/acknowledge signals are generated, and an empty arbiter process is generated which will be edited by the designer to enter the desired arbitration scheme.

# Refinement : Arbiter Process Generation




Access frequency => higher priority

Communication delay => lower priority

Connects behaviors with perf. violation => higher priority

## Arbiter Generation

*Select Arbiter Cost*

| Criteria | Weight |
|---|---|
| **Access Freq** | **1** |
| **Comm. Delay** | **2** |
| **Behaviors with perf violations** | **0** |

*Display Cost*

| Group | SubGroups | Cost | Label |
|---|---|---|---|
| **AV_BUS** | **CAUD** **C1** **C2** | **130** | **1** |
| | **CVID** **C3** **C4** | **76** | **2** |
| * | | * | **> 10** |

*Match Pattern*

**List Access Conflicts** **Label SubGroups** **Apply**

*Arbitration Algorithm*

**Fixed Priority** **Rotating Priority** **Custom**

UC Irvine

Copyright (c) 1993
UC Irvine CADLAB

### 3.15 Refinement : Protocol Selection

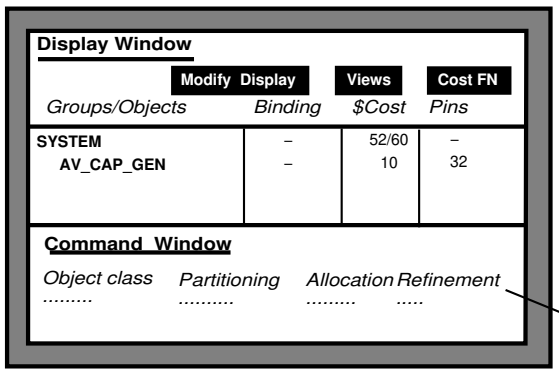
Another refinement task that can be performed is selecting a protocol to implement communication over a communication channel. During grouping and binding, default protocols are assigned to channels. The designer may wish to modify the protocols associated with the channels to improve costs.

Selecting *Refinement => Protocol* from the Main Display brings up the Protocol Selection Display. The designer can select various criteria based on which, the various protocols that can implement a channel are to be evaluated. Examples of such criteria are the bitwidth and the data-transfer rate associated with the protocol and performance constraint violations of the behaviors connected by the channel.

For each channel, the Protocol Selection Display lists the set of protocols which can be used to implement the data transfer based on the cost computed from the selected criteria and their associated weights.

For the channel `CVID`, four possible protocols are listed: `p_hsk32`(32 bit wide data with handshake), `p_hsk16`(16 bit wide data with handshake), `p_hsk8`(8 bit wide data with handshake) and `p_serial`(serial). The current protocol associated with the channel is highlighted. The designer can change the protocol by selecting any of the protocols listed for that channel.

# Refinement : Protocol Selection




- Select Protocol to improve cost.

## Protocol Selection

*Select Cost*

| Criteria | Weight |
|---|---|
| Channel Width | 1 |
| Channel Bit Rate | 0 |
| Perf. Constraint Violation | 2 |

List Protocols

Update Protocol

*Display Cost*

| Channel | Protocols | Cost |
|---|---|---|
| CAUD | p_halfhsk8 | 89 |
| | p_hsk8 | 120 |
| CVID | p_hsk32 | 40 |
| | p_hsk16 | 91 |
| | p_hsk8 | 130 |
| | p_serial | 210 |
| *Match Pattern* * | * | > 10 |

UC Irvine
Copyright (c) 1993
UC Irvine CADLAB

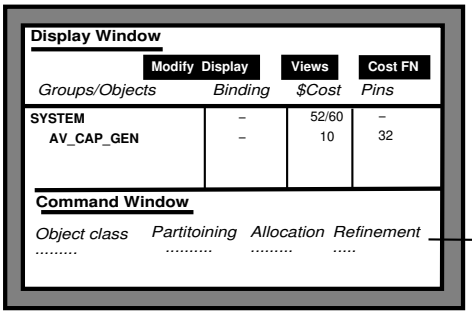
### 3.16 Refinement : Interface Process Generation

During Component Binding, the designer may decide to bind both the behaviors communicating over a channel to standard components. In this case, the corresponding protocols in both behaviors may be different and incompatible with each other. Thus, an *interface process* has to be generated and inserted between the two communicating behaviors to ensure compatibility. The two behaviors now communicate with the interface process which effectively masks the two incompatible protocols from each other.

Selecting *Refinement => Interface Process Generation* from the Main Display brings up the Interface Process Display. The designer can list the channels whose end-behaviors have incompatible protocols. In the figure, channel `C1` connects behaviors `B1` and `B4` which have the incompatible protocols `p_hsk16` and `p_hsk8` respectively associated with them.

The designer can choose to have the interface process generated automatically by the synthesis tools or manually specify a custom interface process to implement the transfer of data between the behaviors.

# Refinement : Interface Process Generation




- Behaviors may have differing and fixed protocols
- Protocols made compatible by inserting interface process

## Interface Process Generation

*Channels Display*

| Channels | Compatible ? | End Behaviors (Protocol) |
|---|---|---|
| C1 | N | B1 [ p_hsk16 ], B4 [ p_hsk8 ] |
| G1 | | |
| C2 | N | B1 [p_hsk16 ], B2 [p_half_hsk8 ] |
| C3 | N | V1 [p_hsk16 ], B2 [ p_half_hsk_8 ] |
| * | N | * |

*Match Pattern*

**List Channels**

**Generate Interface Process** **Custom Interface Process**

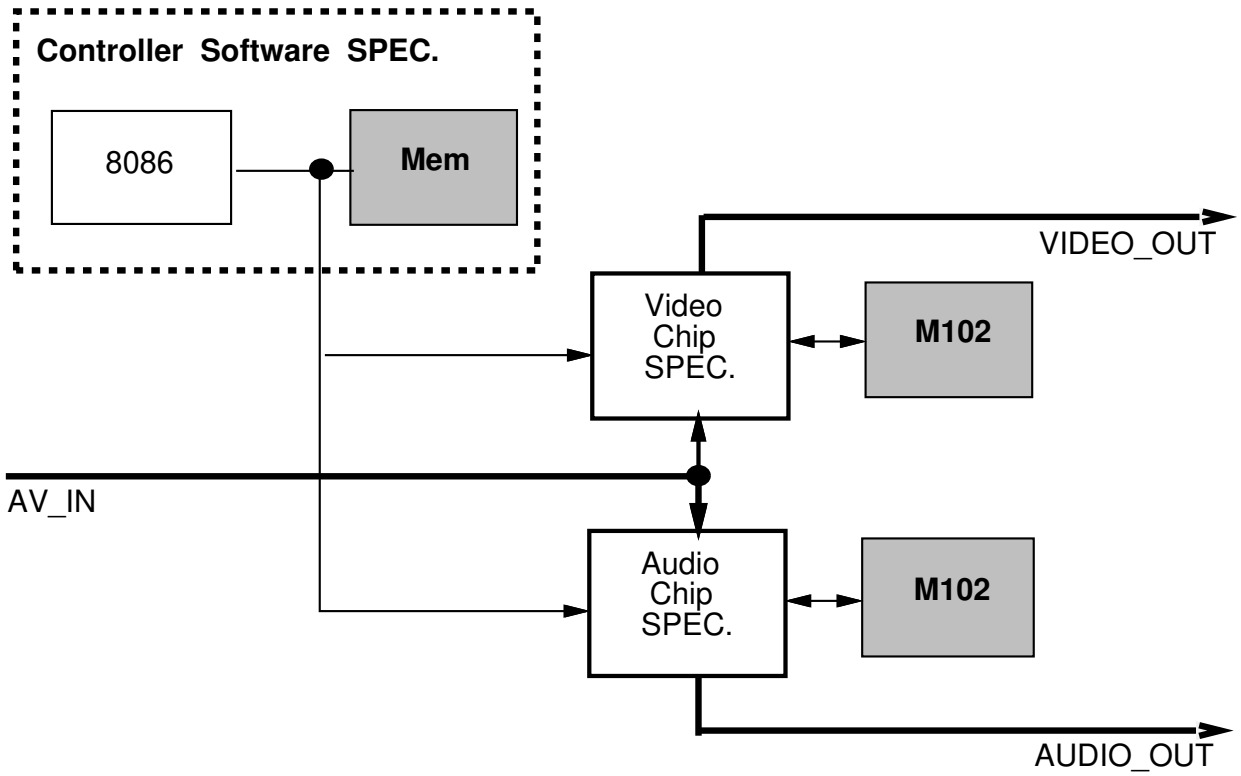
Copyright (c) 1993
UC Irvine CADLAB

### 3.17 System Design Output

As a result of performing the system design tasks on the AV Processor specification, we obtain a set of completely specified modules representing the design.

There are two identical library components, *M102*, which implement the variable groups `M1` and `M2`. 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 the software specification. This is compiled by a software compiler to execute on the desired microprocessor. Finally, the two *M102* memories, the two hardware chips, and the microprocessor and its associated memory are all connected together and assembled on a board.

# System Design Output




UC Irvine
Copyright (c) 1993
UC Irvine CADLAB

# 4 Conclusions

In this report we have presented a system design methodology which forms the basis of the system design framework, SpecSyn, being developed at UC Irvine. The framework offers the designer a set of system design tools to perform tasks such as allocation, binding, partitioning, and **specification refinement**. By applying these tools, the designer can transform his conceptual view specification to a set of completely specified modules which satisfy the system design constraints.

The proposed methodology has three main advantages over current approaches:

1. **Less Design Time:** The methodology requires the entire system to be captured using a specification language before system design is performed. Partitioning, estimation, binding and refinement tools can be developed which 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 may lead to faster, cheaper and smaller designs.

3. **Less RE-design 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. Personnel changes have a greatly reduced effect.

# Conclusion

**Tool to help users develop systems**

Rapid Estimates → *Explore larger design space*
*Faster/cheaper/smaller designs*

Automated Design Tasks → *Less design time*

Excellent Design Documentation → *Less RE–design time*

Copyright (c) 1993
UC Irvine CADLAB

## 5 Acknowledgements

This work was supported by the Semiconductor Research Corporation (grant #92-DJ-146). We are grateful for their support. We would also like to thank Bob Larsen of Rockwell International for his comments and suggestions.

## 6 References

[1] S. Narayan, F. Vahid, and D. Gajski, "System Specification and Synthesis with the SpecCharts Language," in *Proceedings of the International Conference on Computer-Aided Design*, 1991.

[2] D. Gajski, S. Narayan, and F. Vahid, "A System-Level Specification and Design Methodology." UC Irvine, Dept. of ICS, Technical Report 92-102,1992.

[3] *IEEE Standard VHDL Language Reference Manual*, 1988.

[4] R. Lipsett, C. Schaefer, and C. Ussery, *VHDL : Hardware Description and Design.* Kluwer Academic Publishers, 1989.

[5] F. Vahid, S. Narayan, and D. Gajski, "SpecCharts: A Language for System Level Synthesis," in *Proceedings of the International Symposium on Computer Hardware Description Languages and their Applications*, 1991.

[6] S. Narayan, "A Survey of System-Level Specification Languages." UC Irvine, Dept. of ICS, Technical Report 92-100,1992.

[7] S. Narayan, F. Vahid, and D. Gajski, "Modeling with SpecCharts." UC Irvine, Dept. of ICS, Technical Report 90-20,1990.

[8] S. Narayan, F. Vahid, and D. Gajski, "Translating System Specifications to VHDL," in *Proceedings of the European Conference on Design Automation*, 1991.

[9] F. Vahid and D. Gajski, "Obtaining Functionally Equivalent Simulations Using VHDL and a Time-shift Transformation," in *Proceedings of the International Conference on Computer-Aided Design*, 1991.

[10] S. Narayan and D. Gajski, "System Clock Estimation based on Clock Slack Minimization," in *Proceedings of the European Design Automation Conference*, 1992.

[11] S. Narayan and D. Gajski, "Area and Performance Estimation from System-Level Specifications." UC Irvine, Dept. of ICS, Technical Report 92-16,1992.

[12] F. Vahid and D. Gajski, "Specification Partitioning for System Design," in *Proceedings of the Design Automation Conference*, 1992.

[13] F. Vahid, S. Narayan, and D. Gajski, "Constant-Time Cost Evaluation for Behavioral Partitioning." UC Irvine, Dept. of ICS, Technical Report 92-29,1992.

[14] F. Vahid, "A Survey of Behavioral-Level Partitioning Systems." UC Irvine, Dept. of ICS, Technical Report 91-71,1991.

[15] J. Lis and D. Gajski, "Synthesis from VHDL," in *Proceedings of the International Conference on Computer Design*, 1988.

[16] J. Lis, *Behavioral Synthesis from VHDL Using Structured Modeling*. PhD thesis, University of California, Irvine, January 1992.