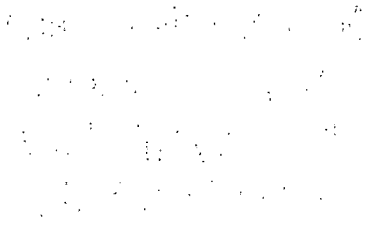
ESSENTIAL ISSUES AND POSSIBLE SOLUTIONS IN HIGH–LEVEL SYNTHESIS

Daniel D. Gajski
University of California
Irvine, California 92717

Technical Report No. 91–18



ESSENTIAL ISSUES AND POSSIBLE SOLUTIONS IN HIGH–LEVEL SYNTHESIS

Daniel D. Gajski
University of California
Irvine, California 92717

1 Wrong Trend vs. Wrong Focus

CAD technology has been very successful in the last ten years. CAD tools for layout and logic design have been exceptionally successful to the point that they dominate system and chip design methodologies throughout the industry in the U.S. and abroad. This widespread methodology consists of manually refining product specifications through system and chip architecture until the design is finally captured on the logic level and simulated. Standard–cell methodology and tools were developed for easy mapping of logic–level design into IC layout. Because of the huge investment in CAD tools, equipment and training, many people believe that this trend will continue by providing more sophisticated CAD tools for capture, simulation and synthesis of logic–level designs.

Logic level, however, is not a natural level for system designers. For example, when we want to indicate that 32–bit values of two variables, a and b, should be added and stored in the third variable, c, we simply write the expression $c = a + b$. We do not write 32 Boolean expressions with up to 64 variables each to indicate this simple operation. It is very difficult to imagine having complex multi–chip systems described in terms of 1 million or more Boolean equations.

If we equate layout–level of abstraction (transistors, wires and contacts) with machine–level programming then logic–level (gates, flip–flops and finite–state machines) can be equated with assembly–level programming. We know that complex software systems consisting of 1 million or more lines of code are not written in assembly language. Similarly, a complex hardware system of 1 million or more gates should not be captured, simulated or tested on the logic level of

abstraction. System designers think in terms of states and actions triggered by external or internal events, and in terms of computations and communications. Thus, we have to develop tools to capture, simulate and synthesize designs on higher abstraction levels close to the human level of reasoning in order to design large complex systems.

On the other hand, high–level synthesis research has been focused on scheduling, allocation and binding algorithms. In the first place, the design descriptions from industry and academia are simple. Since the most complex chips contain no more than one multiplier and one adder, trivial scheduling and allocation algorithms are adequate for synthesis. High–level synthesis, however, does not consist only of scheduling and allocation algorithms. It consists of converting system specification or description in terms of computations and communications into a set of available system components (DMAs, bus controllers, interface components, etc.) and synthesizing these components using custom, or semicustom technology.

In this paper we discuss relationships between languages, models and tools for synthesis–driven design–methodology. We will also discuss essential issues derived from those relationships and some possible solutions. We will also paint with a broad brush, an ideal system for high–level synthesis and propose solutions for some essential issues. Finally, we will discuss future research trends driven by this evolutionary extension of synthesis to higher abstraction levels.

2 Languages, Designs and Technologies

There is a strong correlation among description languages used to specify a design, the design itself and the technology used for implementation of that design (Fig.1).

Hardware description languages are used to describe the behavior of systems either on a chip level or board level. This behavioral description treats a design as a black box with well defined input and output ports, where outputs are defined as functions of inputs and time. On the other hand, a design can be represented structurally as a set of connected components from a given component library. Some components can be grouped together creating hierarchical descriptions which are much easier to understand. Technology introduces a set of constraints in design implementation. Those constraints may refer to a particular chip architecture such as RISC architecture, to a particular layout methodology such as standard cells, to a particular fabrication process such as CMOS or GaAS, or to a certain component library. The technology constraints also determine the quality of design and the time needed to finish a design. Also, the implementation technology determines the CAD tools needed for design. Similarly, each technology has preferred design styles whose characteristics (such as pipelining) should be abstracted into language constructs used to describe them.

These language constructs should be orthogonal, allowing unique, unam-

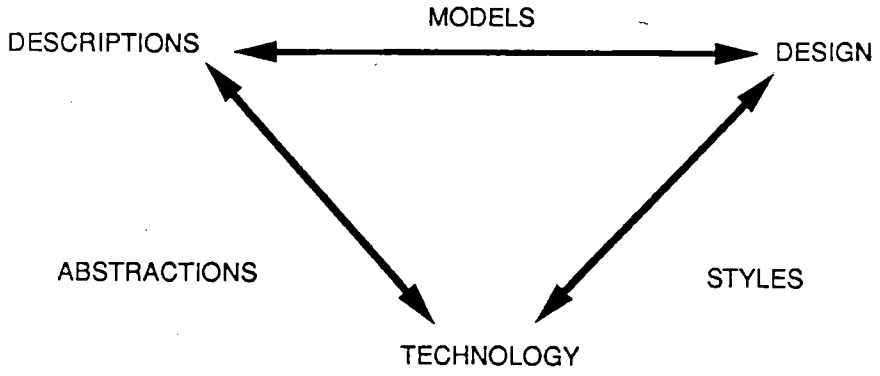


Figure 1: Description–Design–Technology Dependence

biguous descriptions, for each design. Each design, however, can be described or modeled in many languages in several different ways. Figure 2 shows two different descriptions of the same behavior and designs derived from each description. The signal ENIT when asserted starts the counter by setting $EN = 1$. When the counter reaches the limit, the comparator asserts its output and sets $EN = 0$, which stops the counter. The first model treats ENIT as a level signal that stays asserted while the counter is counting. The second description uses the positive edge of the ENIT signal to set $EN = 1$, and the output of the comparator to set $EN = 0$. As shown in Figure 2, this behavior will result in two different implementations. Since the modeler has chosen to use the positive edge of ENIT to indicate the moment when EN becomes equal to 1, the second implementation has an extra D flip–flop used to store the occurrence of the positive edge of the ENIT signal. The implementation shown in Figure 2b is correct but unnecessarily costly.

This simple example shows that different modeling practices result in different designs and that complex synthesis algorithms for disambiguation of the design descriptions will be required. The solution is to introduce *structured modeling* practices, similar to structured programming, which will limit the modeler to a unique description for each design [LiGa89], or to develop *orthogonal languages* whose syntax will disallow designers from writing different descriptions for the same design.

Similarly, a design implementation is not unique. For each function in the design there are several design styles each suitable for different design goals or constraints. For example, two different implementations of the EXOR function are shown in Figure 3a and 3b. 12–transistor design (shown in Figure 3a) is better suited for large loads since only 2 output transistors must be oversized. On the other hand, 10–transistor implementation (shown in Figure 3b) is better

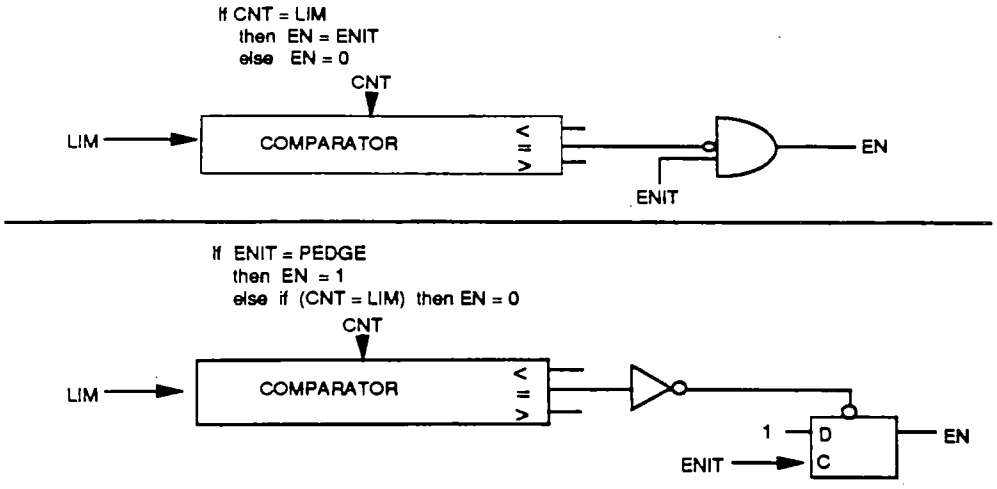


Figure 2: Two Different Descriptions of an Event (a) level sensitive, (b) edge sensitive

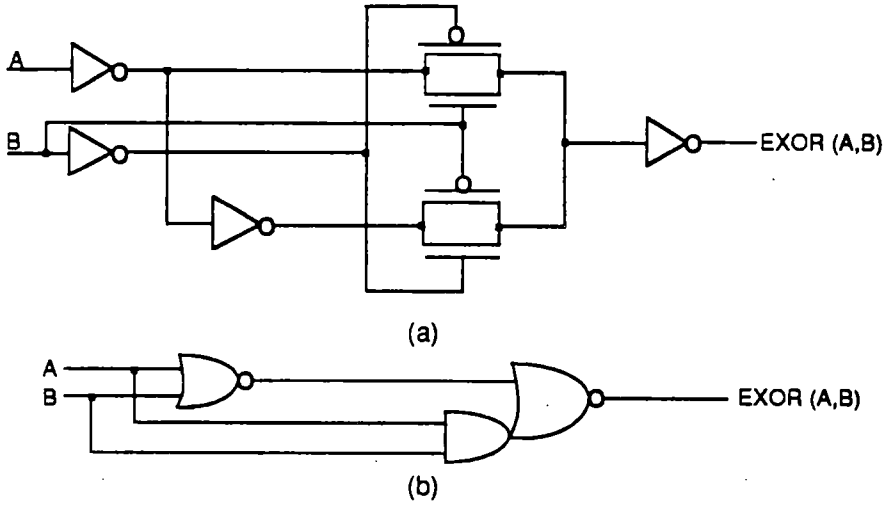


Figure 3: Two Styles of EXOR Implementation using (a) transmission gates, and (b) AND–OR–INVERT gate.

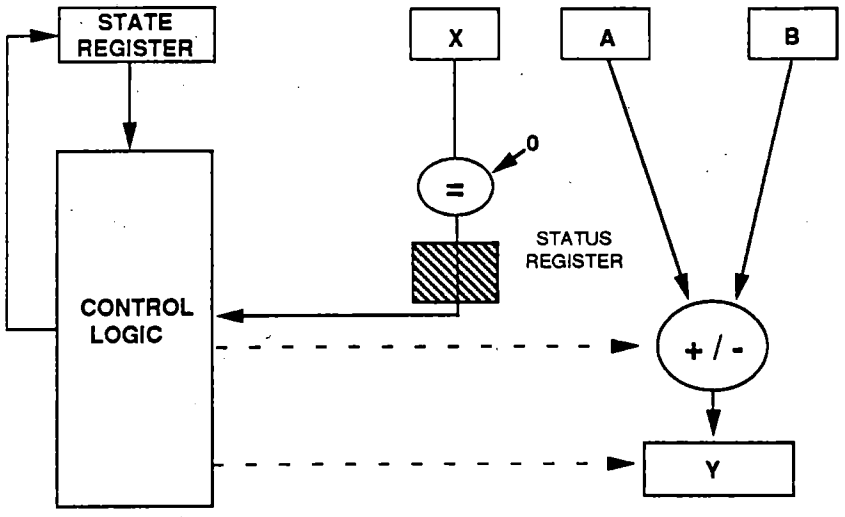


Figure 4: Two Different Design Styles: (a) without status register, and (b) with status register (shaded box)

suited for small loads since 10–transistor circuits need a smaller layout area than a 12–transistor circuit. In case of large loads, six large transistors at the output will take a much larger area than a 12–transistor circuit.

Thus, future synthesis systems must allow for making (automatically or manually) design tradeoffs using different design styles for different technologies and design goals. High–level synthesis systems will not be accepted by designers until this capability is available.

Design styles are also reflected in design descriptions. The more detailed the design model, the more it suggests a design style. Figure 4 suggests two different styles of data–path implementation of the statement: **if** $x = 0$ **then** $y = a + b$ **else** $y = a - b$. The condition $x = 0$ can be directly fed into the control logic or latched in the status register before tested in the control logic. In the former case, clock period is longer but the statement can be executed in one clock period while in the latter case the clock period is shorter, while the statement execution requires two clock periods. If the design description allows arbitrary expressions as conditions in *if statements*, the first design style (Fig.4a) is more suitable. If only single bits are allowed as conditions, the style with the status register (Fig. 4b) is better suited.

Using description models and style that do not match requires complicated model analysis and sophisticated style–transformation techniques. Thus, extension of synthesis to system levels requires a matching of models, abstractions, and design styles.

3 Essential Issues in Synthesis

Extending synthesis to higher levels also requires solving the increased complexities of the languages, designs, and technology on higher levels of abstractions.

The synthesis on the layout level deals with three objects: transistor, wire, and contact between two layers of material. Layout models have a good formal foundation in graph theory, while algorithms deal with ordering and connectivity of basically one object: a transistor.

Logic Synthesis is based on well known formalism of Boolean algebra and the number of objects, such as gates and flip–flops, is still a small number. Extending synthesis to higher levels of abstraction proves to be difficult because of the lack of theoretical formalism such as Boolean algebra or graph theory, the lack of unique unambiguous design descriptions, and because of an increased number of objects possibly including all real chips in the market!

3.1 Design Conceptualization

The main problem with a design description is its change with design over time. At the beginning the specification is vague with little or no implementation detail. More detail is added as the design evolves. Furthermore, different aspects or characteristics of the design are required by different members of the team. For this reason it is difficult to imagine the existence of one universal language that would serve every purpose. In addition, system description in such a language would be too cumbersome since very few designers are interested in all the aspects of a design.

Thus, we must develop the capability to generate different *design views* for optimization or verification of different aspects of a design. These views may be graphical or textual. For example, if a designer wants to equally utilize all resources, he or she may only be interested in state–by–state usage of available resources such as memories, registers, ALUs and buses. On the other hand, a person performing floorplanning would be interested in the shape and size of those resources. A graphical language for floorplanning that would allow viewing of the resource footprints and assigning their position on the chip would be beneficial in manually optimizing chip floorplans.

Thus, development of languages for specifying different aspects of design and intermediate forms to allow manual modification of synthesized design must be developed for design conceptualization. Such languages must be embedded in design environments that will also provide design quality metrics for exploration and design evaluation.

3.2 Database Issues

As mentioned above, a designer generally does not want to see all the aspects of the design at one time. A central database will store all aspects of the design and generate different design views on demand. The type and format of a view

should be specified through some *format schema*. The database that contains all aspects of the design will be initially built from the input description and then augmented as synthesis proceeds. The additional information could be provided by the synthesis tools or by a human designer through a graphical interface. The database should also be able to check consistency of upgrades and provide estimates in case of incomplete information.

Furthermore, a smart database of *generators* for complex components must be available. A user should be able to query the database for types of components, their functionality, area, delay, performance, layout height, width and shape. The database should be able to supply a component for any given functionality and constraint. For example, a user may require a 13–bit ALU with add, subtract, NAND and NOR functions or a 4K by 32–bit memory with 25 nanoseconds access time. The database does not store components with fixed parameters; instead it has the capability to generate them for a given set of parameters.

3.3 Technology Independence

Although technology independence is on everyone's wish list, very few CAD tools achieve technology independence in reality. On the layout level, technology independence can be achieved by laying dimensionless objects on a virtual grid, expanding the objects into their real size and compacting the complete layout. This approach requires only a change in the technology file containing spacing rules when going from one fabrication process to another. It requires, however, sophisticated compaction algorithms with local optimization in order to equal manual layouts.

Technology independence on the logic level is achieved by synthesizing design with generic gates and then mapping generic gates into library components by performing local transformations. This technique does not easily scale up to MSI components such as 4–bit ALUs, 4–bit counters or registers. Further scaling up the technology independence to microarchitectural components such as ALUs, multipliers, registers, memories, and buses requires component generators that are capable of generating generic components. Extending technology mapping to the system–level synthesis requires the capability of mapping sub–systems such as processors, DMAs, UARTs, and device controllers, among others. That requires (a) recognizing the functionality of the library components, and (b) techniques for synthesizing the missing functionality externally with glue logic components.

3.4 Design Learning

Learning has not been applied to CAD tools yet. There is a need for learning however. When a different ASIC library is used it is necessary to redesign higher level components to take advantage of the new library. For example, if the new library has a 4–bit adder, while the previous library had only a 2–bit adder, at

least the carry–look–ahead function must be redesigned. This redesign can be executed through learning *technology–adaptation* rules [KiGa90].

Secondly, learning could be used in optimization at layout, logic, and system–levels. Two main problems can be indicated: (a) learning optimization rules, and (b) learning control strategies. Control strategy determines the order in which optimization rules are applied. The new knowledge is obtained by monitoring improvements made by designers on the final synthesized design and generalizing those examples into rules, principles, and control strategies.

3.5 Design Synthesis Complexity

It may seem that extending synthesis to higher levels of abstraction will introduce more design levels and more tools and languages that designers must learn. It is possible to compress the entire synthesis into two levels, however.

The system design can be modeled with a set of processes communicating together through well defined protocols. One or more processes describe each system component such as a processor, DMA, bus arbiter or memory. Each process is implemented with a set of microarchitectural components and control logic, and can be succinctly described by a set of register–transfers. The basic components of system–level synthesis are register–transfer components such as adders, multipliers, registers, memories and buses. Thus, process synthesis and protocol synthesis represent system–level synthesis.

On the other hand, each of the register–transfer components can be synthesized with available logic and layout synthesis techniques. Logic synthesis is not only available for random or glue logic, but can be used for synthesis of many register–transfer components. Logic synthesis however, must be upgraded to include layout parameters such as transistor sizing, layout capacitances, wire resistivity and others. Also, new techniques for synthesis of interfaces, buses, memories and other regular structures must be developed.

Thus, two synthesis levels will eventually emerge: *system level synthesis* to translate a set of communicating processes into register–transfer components, and *component synthesis* to translate component descriptions into layouts (custom or semi–custom).

4 A Hypothetical System

The hypothetical synthesis system is shown in Figure 5. It uses the standard capture–and–simulate design methodology augmented by automatic synthesis. The designers may choose to manually refine the design or to use a synthesis tool.

4.1 Conceptualization Environment

A conceptualization environment is intended to capture, verify and estimate initial ideas. It supports designer's evaluations and tradeoff through every phase

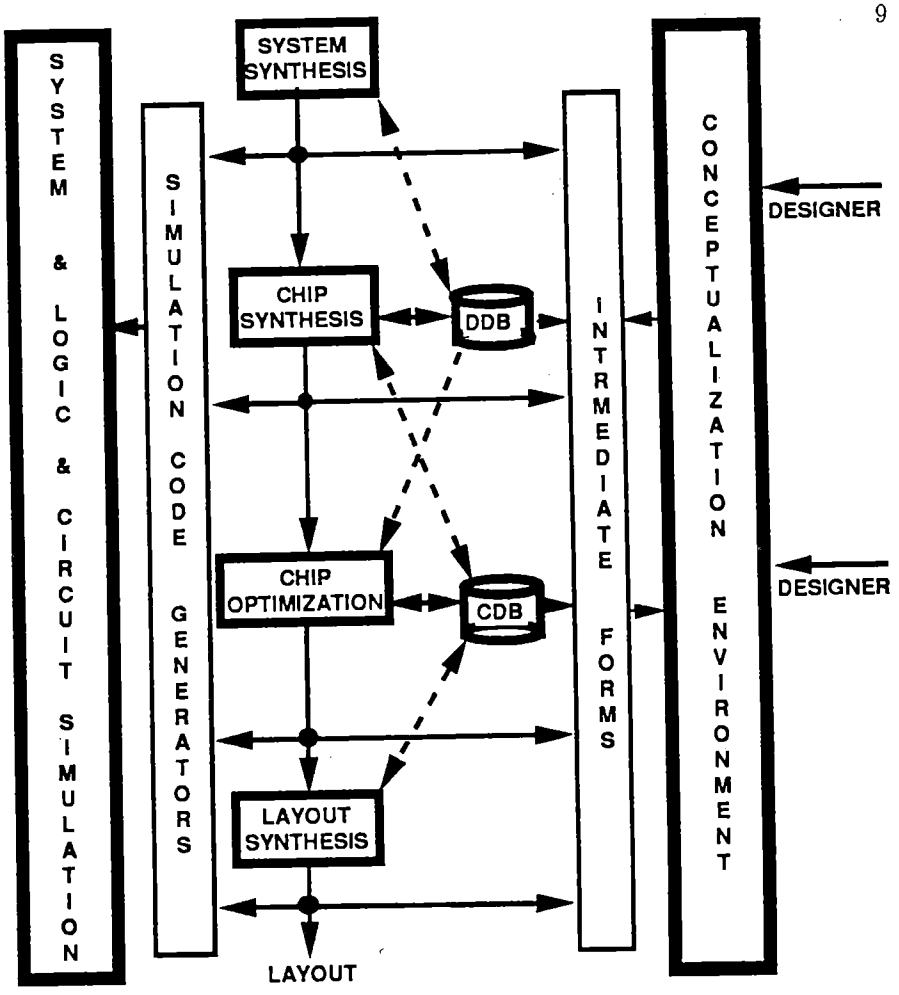


Figure 5: Hypothetical Synthesis System

of the design process by displaying *design quality metrics* at every step. Its secondary goal is to allow an easy integration of other tools for decision support [HaGa91].

A general environment provides design data on different levels of abstraction, keeps track of a current representation and maintains links between levels of abstraction. The user also views the data via displays and can modify it through a set of editors. In addition to manual modification, the data can be also modified automatically by design tools under supervision of the user. The quality analysis tools, on the other hand, must function without user control, and should rapidly calculate quality metrics for any complete or partially complete design. For different levels of completeness different quality analysis may be used.

Generally, there are three types of quality measures: *temporal, structural*

and *spatial* (corresponding roughly to behavioral, structural and physical design representations). A designer conceptualizes a system or design in terms of computational chunks stored and terminated by external or internal events. Each chunk is defined as a sequence of computational steps; that is, by a sequence of statements in some programming or description language. Since computation is mostly sequential the memory elements can be shared among different computational variables and all of the operations can be performed by a small number of functional units. Thus, temporal measures indicate possible sharing of resources over time. Such measures are variable lifetimes, or operator–usage frequencies. On this level of abstraction, a designer is basically interested in tradeoffs between serial and parallel computation, and finding a sufficient amount of parallelism to satisfy performance requirements or available resources.

On the structural level design is represented by a set of interconnected components. The quality measures are related to types of components used and their electrical and physical properties such as cost, delay and power dissipation.

The spatial quality measures are related to assignment of spatial attributes to structural descriptions. Here, the designer is interested in layout shape and position of pins on the layout boundary for the purpose of floorplanning an IC, or the size of the boards, racks and cabinets for mechanical design of the system.

4.2 Behavioral Intermediate Form

The traditional view of behavioral synthesis [McPC88] assumes that the synthesis automatically generates the structural design from a user specified abstract behavior.

Existing systems that perform behavioral synthesis do not permit the user to interact in the design synthesis and evaluation loop. If the synthesized design does not meet the constraints, the user is still forced to re–synthesize the design automatically from the abstract behavior by changing some high–level constraint [BrGa90]. The major drawback with this approach is that the user cannot impose structural constraints (in the form of an initial design structure), or provide design hints. In order for the user to guide the synthesis process, the following requirements must be satisfied:

1. *Partial design specification:* The user should be able to specify a partially designed structure as an initial constraint; the synthesis tools should then be able to complete the rest of the design.

2. *User–bindings:* The user should be able to selectively bind behavioral operators to particular states, behavioral operators to components, and behavioral variables to storage components or connections.

3. *Modification of compiled designs:* The user should be able to modify a

structural design between various synthesis tasks, or after the synthesis is completed.

4. *Coexistence of automatic and manual design philosophies:* The design may be refined by an expert designer, or by an automatic synthesis tool. Such a design paradigm requires consistency checks and a powerful simulation environment for verification of the behavior.

The need for such user–interaction is evidenced by work being performed both in the U.S. [ABWS89][WhON89] [DuGa89][ThBR87] and abroad [BrMS89] [YaIs89].

One such representation is Behavioral Intermediate Form (BIF) [DuHG90] that uses annotated textual state tables to support the above requirements. It facilitates easy translation to and from the data structures for synthesis algorithms, and thereby allows synthesis tools to be interchanged and upgraded. It serves as a useful linking mechanism between the behavior and the structure. Its model is general, since it can express hierarchy, concurrency, timing relationships and asynchronous behavior in a single, unifying intermediate form.

We use the environment shown in Figure 6 as a representative synthesis framework to show the utility of BIF. The figure is organized into three columns: the synthesis tasks on the left, the user interface on the right, and different design views of the intermediate representation in the middle: the state table, the unit list, the connectivity list, and the symbol list.

The user typically specifies the behavior of the design in a behavioral specification language such as VHDL. A language compiler parses the input into a data structure which is captured in the first level of the intermediate format, creating the symbol list and a hierarchical operations table. In addition, if the user specifies some structure along with the behavior, this structure is captured in the unit and connectivity lists. The abstract input is naturally described using sequences of groups of operations that form a multi–level hierarchy, where a group of operations at one level of hierarchy can be defined by a sequence of operation groups at a lower level. The lowest level of this hierarchy consists of operation sequences that are similar to basic blocks in a standard programming language. Each of these "basic blocks" is called a *super–state*, since it may span several machine states. BIF's hierarchical super–state table captures this behavior by describing the operations performed in each super–state, and the sequencing between super–states.

At the next level of abstraction, a state scheduler "slices" the super–states by assigning operation sequences to specific states of the synthesized design [PaGa86]. BIF's op–based state table is generated by this synthesis task. This table uses conditional triplets to capture the behavior of the design on a state–by–state basis. Each triplet describes the condition tested, the operations performed and the successor state. The successor state is entered only when its corresponding controlling event occurs.

Resource allocation determines the type and number of structural components needed to implement the structural design. BIF's unit–based state table

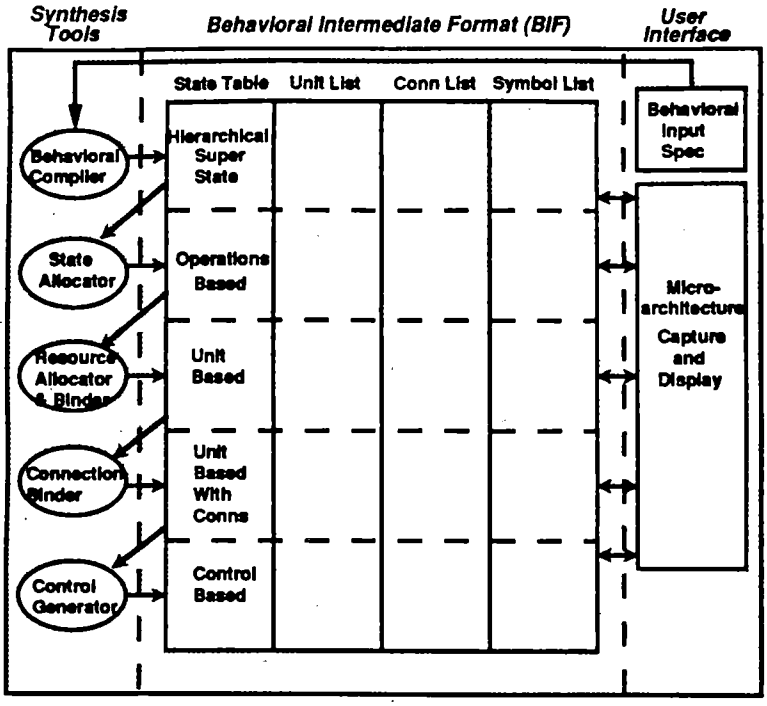


Figure 6: A Canonical Synthesis Environment

captures the structural operation of the design on a state–by–state basis after resource allocation and binding. Each table uses triplets to describe the unit generating the conditional, the units performing the conditional operations, and the event–controlled next state to be performed. The operations in the unit–based state table only specify which components are to be used as inputs for each operation; they do not specify the connection paths for these inputs.

The task of *connection binding* adds these connection paths to the unit–based state table to create a unit–based state table with connections. This table describes the complete structure of the synthesized data path, but lacks the control signals for the components.

Finally, the task of *control generation* creates control lines for every functional or storage unit that needs to be controlled. The control–based state table captures this functionality for each state by using triplets that describe the conditions, the control lines activated, and the subsequent event–controlled next state.

At each level of the synthesis process, the appropriate synthesis task can be performed automatically (by a set of algorithms and rules), or manually (by the user through the user interface). The user interface graphically displays the units, connections, and the state tables. This permits the user to comprehend the complete behavior and structure of the design at each level of abstraction.

Users can insert component definitions, component implementations, component generators or optimization tools to ICDB through the knowledge acquisition support mechanism.

4.3 Design Database

Complex systems are usually described by a set of processes communicating through global signals. The design synthesis (manual or automatic) proceeds sequentially converting one process at the time into a hardware design. Several different alternative implementations are usually generated for each process before the final design is completed. Thus, any design system must include a database for storing design data and all alternative implementations beyond one design session. Very little or no research has been performed in the area of databases for high–level synthesis.

Behavioral Design Database (BDDB) [RuGa91] is a design database that not only manages the *design data* produced and consumed by different behavioral synthesis tools, but it also maintains the *meta design information* relating these various chunks of design data according to semantic relationships, such as, hierarchy, version, and equivalence. BDDB thus forms the foundation for integrating different design tools into one cooperative CAD framework in which design tools communicate via common design data. Such a centralized data server also simplifies the tasks of consistency checking, design verification and controlled design exploration.

BDDB is based on a three–tiered design representation model for behavioral synthesis which is composed of the following graphs: the *conceptual*, the *behavioral* and the *structural graph model* [RuGa90]. The conceptual graph model captures the overall organization of the design data. It covers concepts, such as, the design entity hierarchy, the version derivation tree, and configurations. The behavioral graph model describes the design behavior. It corresponds to a hierarchical Control/Data Flow Graph representation that is augmented with timing constraints, events, state transition information, and structure bindings. The structural graph model captures the hierarchical graph structure of interconnected components augmented by timing constraints. It represents the design structure and its physical implementation.

BDDB solves the problem of point–to–point data translations between all pairs of design tools that exchange information by providing customized interfaces for these tools to the central design representation. These customized interfaces, also called *design views*, consist of (1) a subset of the information content of the global database, and (2) a reorganized type structure (view schema) in which the information is expressed. BDDB provides a *view description language* for the specification of these design views. A design tool uses this language to define its own local schema, through which it wishes to view the current design, as well as its own access operations on this schema. This organization has several advantages. First, it provides flexibility as new customized view types can be created on the fly using the view description language. Second, it guarantees extensibility as new information can be added to the global data schema without disturbing existing views.

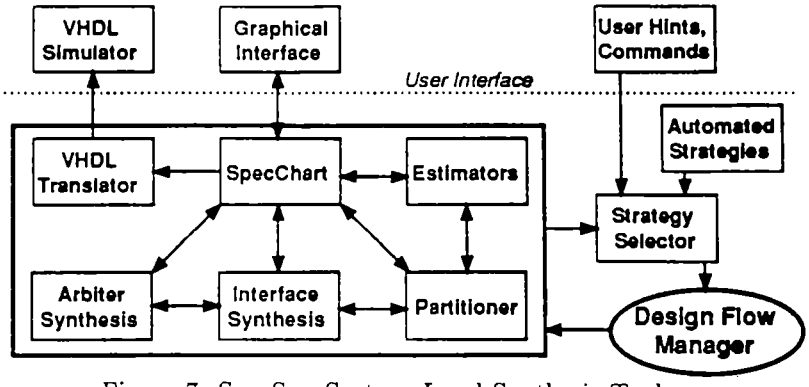


Figure 7: SpecSyn System–Level Synthesis Tool

4.4 System Synthesis

System–level synthesis refers to synthesis at the computer system level. The primary aim is to convert a system's specification into a set of one or more interconnected chips/modules. This involves determining the number of chips necessary to satisfy the given constraints *(estimation)*, distributing the specifications among the chips *(partitioning)*, finding a well–defined structure for each chip, and creating proper interchip/intermodule interfaces *(interface synthesis)*.

Such a CAD tool requires an executable specification language, with appropriate abstractions to concisely specify a system functionality and requirements. Designers do not usually think in terms of programming languages when designing systems; instead, they draw flow charts, boxes, arrows, etc. SpecCharts [VaNG90b] is an attempt to capture these conceptualizations. SpecCharts represent a multi–module system with a hierarchy of state diagrams, catering to the expression of concurrent behavior and constraints, and using VHDL sequential statements to describe leaf–state functionality. Due to this and other constructs which enable the omission of detail, such as protocol–based data transfer, the general behavior of a system can be easily discerned.

SpecSyn, shown in Figure 7 [VaNG90a], is a design tool to aid a designer perform system level specification and synthesis. Given a specification, which may include a set of constraints, the goal is to synthesize a set of interconnected chips or modules satisfying those constraints, some of which may be bound to prefabricated chips or modules.

Given the specification, *estimators* predict parameters such as area, exe-

cution time, pin count or power consumption for any given process. These estimations will be used by the SpecSyn partitioner. The SpecChart language allows for user estimations, which the estimation tool uses instead of computing its own, thus providing some user guidance to other tools.

Given a maximum size for a chip and a specified number of chips, a specification may need to be *partitioned* among chips. In other words, large data structures, or perhaps entire processes or only parts, may be moved to different chips, while maintaining the same basic functionality. This implies that the language used should have the ability to specify which portions of the specification lie on which chip, and should keep interchip communication simple.

During synthesis, it may be determined that access to a data structure must be limited (e.g., due to pin constraints). Concurrent processes which previously accessed the data structure freely must now be *arbitrated* between, and each access is now implemented via some type of protocol, such as a handshake. The language used should thus permit arbitration schemes to be defined, and should keep the protocol–based accesses simple.

Eventually, whatever method the specification language uses to simplify protocol–based data transfer, the abstraction will need to be replaced by low–level constructs such as ports, connections between them, and signal assignments and other code that implement the protocol. In addition, protocols may need to be matched, and optimizations of port (pin) usage may need to be done. These tasks are collectively referred to as *interface synthesis*. The specification language should be such that the synthesis of the low–level details of a protocol–based transfer from the high–level abstraction used is simple.

Each of the tasks discussed above must be performed in some particular order that leads to a design that satisfies the constraints. This could be done by automated strategies which contain predefined algorithms, or manually by permitting the designer to apply the tools directly, or by some combination thereof.

4.5 Chip Synthesis

After a system is partitioned into chips with a tool such as SpecSyn, for example, chip synthesis must generate a chip microarchitecture consisting of register transfer components such as ALUs, counters, register files, memories, buses, and I/0 drivers. The chip description may contain several different blocks or modules operating concurrently. Each block may be described using one of the four design models: *combinatorial, functional, register–transfer* and *temporal*. The combinatorial model is used to model purely combinatorial logic while the functional model is used for simple finite–state machines such as counters or control–dominated logic such as the 2910 controller. The register–transfer model is used to describe designs consisting of datapaths and control units. It describes the set of register assignments for every state of the control. The temporal model describes a computation without relation to any control state or any hardware implementation. It describes design as an ordered sequence of

assignments to variables.

Many languages such as VHDL allow the above design models to be described in several different ways. For example, all four models can be described in VHDL with sequential (process) type statements. Since each model requires different synthesis algorithms, each description must be annotated with the model type since language constructs are not sufficient to recognize the model type. Similarly, *structured modeling* (similar to structured programming) provides guidelines for writing synthesizable models by prescribing description templates for different design styles. Thus, *chip synthesis* consists of recognizing the design model in the description and reducing all different descriptions to a canonical form from which the final design will be synthesized and optimized for different goals.

A system based on these principles is the VHDL Synthesis System (VSS) [LiGa89] [LiGa91]. It consists of several modules. A Graph Compiler parses the VHDL input description into an internal representation called a *Control–Data Flow Graph* (CDFG). This internal representation is optimized toward a canonical format that can be realized efficiently using generic components from the GENUS Library [Dutt88]. The Design Compiler performs the mapping of CDFG into a structure of GENUS components. A Design Compiler uses different algorithms for different design models. For combinatorial and functional models, VSS will output the structural netlist in VHDL. For register–transfer and behavioral models, VSS will output, in addition to the VHDL netlist, a BIF state table which is derived from CDFG that has been annotated with state information and component and connectivity bindings.

Experiments with VSS show that human–quality designs can be achieved with the first three models, while design quality obtained from temporal models is very dependent on VHDL coding style.

4.6 Component Database

Behavioral–synthesis tools generate a microarchitecture design from behavioral or register–transfer descriptions. The generated microarchitecture consists of register–transfer components such as ALUs, multipliers, counters, decoders, register files and memories. These register–transfer components are usually composed of complex cells such as 4–bit ALUs, 4–bit counters, or 3–to–8 decoders or simple cells such as gates and flip–flops. Unlike basic logic components, register–transfer components have many options or parameters. One of the parameters is the component size in the number of bits. Other parameters are related to functionality, electrical and geometrical properties of the component. For example, counters may have increment and decrement options as well as load, set, and reset functions. Also, each component may have different delays and drive different loads on each of its output pins. Each component in addition may have several different options of aspect ratios for layout as well as position of I/O ports on the boundary of the module.

Thus, a component database for behavioral synthesis should generate com-

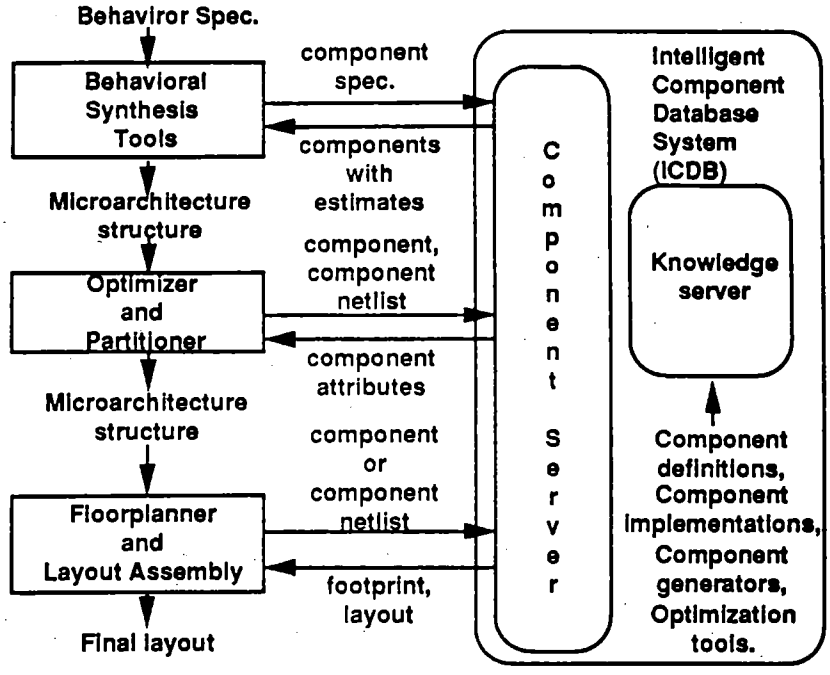


Figure 8: Component Database Role in High–Level Synthesis

ponents that fit specific design requirements and provide information about a component's electrical and layout characteristics for possible architectural tradeoffs. Since behavioral synthesis is in its infancy, very little attention has been given to component generation. Some preliminary work has been reported in [RDVG88], [JKMP89], [ThDW88] and [Wayn86].

Such a component server, called Intelligent Component Database (ICDB), and shown in Figure 8, has been developed at UCI [ChGa90]. ICDB can dynamically generate components for a given set of constraints and attributes.

During transformation of a behavioral description into a microarchitecture level structure, ICDB provides the delay, minimum clock width, area, and minimum setup and hold time for each component. For example, during operator scheduling, a synthesis tool can use the component delay time to determine the proper clock width. A behavioral synthesis tool can also use this information to decide whether to chain two operations together in a single clock, or whether to place an operation in multiple clock steps. When doing resource allocation, ICDB supplies to the synthesis tool a list of components that perform the requested function. This way the synthesis tool can select appropriate components according to the delay requirements. During design optimization,

tools can replace selected components with other components that better meet additional considerations, such as area shape for floorplanning or transistor sizing for different loading. In the microarchitecture optimization phase, ICDB is also queried to determine if components can be merged and whether merging can produce a better design. For example, a register and an incrementer can be merged into a counter. To achieve a good floorplan, the partitioner may try different ways of clustering components and retrieve their shape function from ICDB. It can also assign the pin positions of the combined object and ask ICDB to generate the layout according to this new plan.

ICDB is composed of two subsystems: (1) a knowledge acquisition support system and (2) a component server. The component server provides two types of facilities. It (1) generates components from a given component specification and (2) answers queries about generated components. A generated component is represented in a VHDL netlist for logic1–level structure or CIF for layout.

4.7 Chip Optimization

Chip optimization is performed on the register–transfer netlists containing components such as ALUs, counters and memories.

The first approach is to expand all components to gates and perform logic optimization. Logic optimization of large designs, however, may require large amounts of CPU time and memory. The same will be true for layout generation. Furthermore, some optimizations can be made at the microarchitecture level that cannot be made at the logic level.

The second approach involves only a partial expansion of the design. Various groups of the components can be combined into a single component and optimized. For example, random logic gates can be grouped together and passed to a logic optimization tool while more regularly structured components that will be laid out in a datapath (such as ALUs) are optimized separately and not combined with the surrounding logic.

Such a system for chip optimization that fills the gap between behavioral and logic synthesis tools is the MILO System (Fig.9) [VaGa88]. MILO uses a new methodology for microarchitecture–level optimization that greatly reduces the amount of technology–specific knowledge necessary to perform the optimizations. Microarchitecture components are generated by the ICDB for the given set of parameters. Thus, the microarchitecture optimizer does not need to deal with multiple logic optimization tools, layout module generators, transistor sizing tools, etc.

Often the chip architecture produced by behavioral synthesis tools such as VSS contain inefficiencies such as constants that can be propagated through a design, and common subexpressions that appear multiple times in the design, each time with replicated hardware. These can partly result from the fashion in which the user wrote the behavioral description. Also, optimization must modify the design in the direction of meeting time and area constraints. Tradeoffs must be made along different paths. On critical paths optimizations that reduce

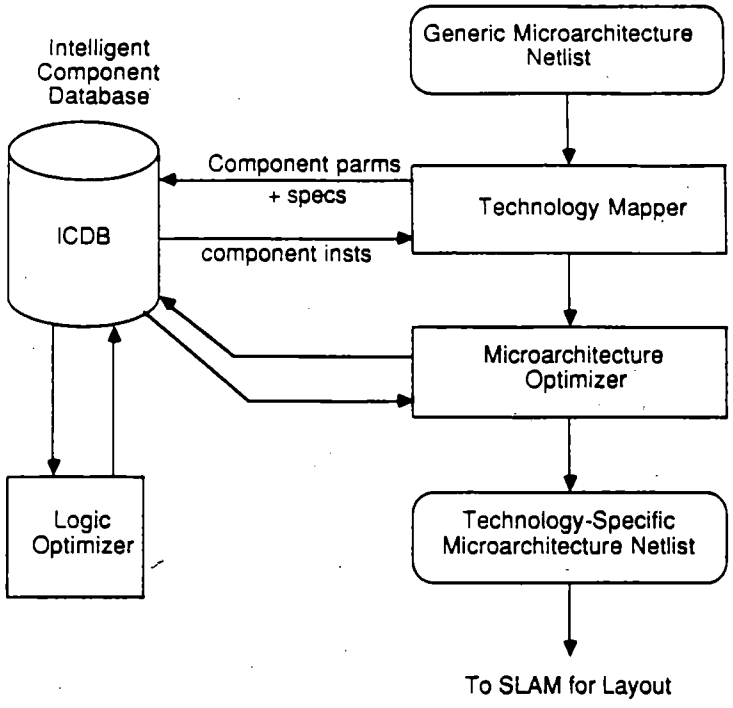


Figure 9: MILO System for Chip Optimization

time are required, possibly at the expense of increased area. Non–critical path optimizations attempt to reduce area as long as doing so does not create a new critical path. In performing these tradeoffs, the microarchitecture optimizer can select a different architectural style for the component, merge components and reoptimize their logic, insert buffers to improve drive capability, replace a set of components with a single component that performs the same function but more closely meets the constraints, restructure components to reduce delay (such as factoring multiplexors), duplicate logic to reduce delay, or change the layout style of the component. These type of improvements are nearly impossible to pursue once the design has been expanded into lower level logic.

Most of the time the microarchitecture is defined in terms of generic components.

The components in the generic netlist are converted to technology–specific components by a *technology mapper*. The technology mapper queries the database by providing the set of component parameters. The database returns one or more components that meet the specified parameters. From this set of components the technology mapper selects the component that contains the smallest set of functions required. For example, if a component with the ADD and SUBTRACT functions is requested, the database may return two components: an

ADD/SUBTRACT unit and an ALU. The technology mapper would select the ADD/SUBTRACT unit.

At this point the design consists of two levels. One is the microarchitecture netlist, the other is a technology–specific gate–level netlist for each microarchitecture component. The *microarchitecture optimizer* first employs rules that make transformations that should improve both time and area. For example, converting a register and incrementer into a counter. Next, the critical paths are identified and optimized. Once critical paths have been processed, the microarchitecture optimizer operates on non–critical components, making similar decisions as in the critical path improvement phase but this time with an eye toward area improvements. The microarchitecture optimizer then produces a VHDL netlist that is passed to the floorplanner/layout assembler for layout.

The microarchitecture optimizer uses a new methodology for selecting microarchitecture components to be used in the design. The microarchitecture optimizer does not perform component rearchitecting and does not have knowledge of tools for logic optimization, transistor sizing, and other component reoptimization techniques. Instead, these tasks are left to the component database. The component database contains a library of logic generators that produce a combinational and sequential representation that describes the low–level behavior of the component. One or more generators can be selected based on the parameters supplied by the synthesis tool. The component description is passed to a logic optimizer [VaGa88] with a set of time constraints. The logic optimizer produces a technology–specific design using components from a designated library or can generate complex gates and select transistor sizes for use in a custom layout [WuVG90]. The microarchitecture optimizer passes a set of time/area constraints to the database and the database examines possible ways to achieve the constraints. The database can choose from different architectural styles and can choose from multiple optimization tools to redesign the component. This frees the microarchitecture optimizer from dealing with technology concerns and managing component optimization tools. All of this is centralized in the database.

4.8 Layout Synthesis

Surveys of VLSI products reveal that most of the fabricated chips can be described by register–transfer schematics or netlists. The products in this category include DMA controllers, bus controllers, disc controllers, and programmable I/O interfaces; that is, basically all chips for computer design with the exception of CPUs and memories.

The preferred layout strategy for such designs is the use of standard cells. Standard cell methodology does not take into account the regular nature (bit–slice property) of register–transfer components which can be laid out using a bit–sliced layout style. Unfortunately, if the register–transfer components have different bit–widths, a large portion of layout area is wasted in the bit–sliced style.

Our *sliced layout architecture* [LaGW91] combines over–the–cell routing and datapath folding to achieve high layout densities. The sliced layout is a stack of register–transfer units. Each bit–slice has the same width, but unit heights vary with the unit functionality. The stack grows horizontally when the bit–width increases, and grows vertically when the number of units increases. The sliced stack uses an over–the–cell routing strategy with data signals running vertically in metal2 over the bit–slices. Power, ground, carry, and control lines are routed horizontally in the metal1 or polysilicon between the bit–slices.

Each bit–slice has a fixed horizontal pitch and a fixed number of metal2 routing tracks over the cell. Inputs and outputs of bit–slices can be connected to any of the tracks. Each unit, such as an ALU, multiplexer, register, adder, or shifter, is generated by a parameterizable generator.

Units often have varying bit–widths. These bit–width mismatches create empty space within the sliced stack's bounding box. A stack folding algorithm folds small units into this empty space. After forming the sliced stacks, the rest of the random logic is placed around the stacks under constraints such as input/output port positions and aspect ratio.

Our system for layout generation from register–transfer VHDL netlists, SLAM, is shown in Figure 10 [WuCG90], [WuGa90].

SLAM partitions register–transfer netlists into bit–sliced stacks and glue–logic modules. It also selects a floorplan style that optimizes area utilization. SLAM consists of four parts: (i) *Partitioner,* (ii) *Stack placer and router,* (iii) *Glue–logic binder,* and (iv) *Floorplanner.*

The SLAM compiler first constructs a connected graph from the netlist. The partitioner then separates component instances into sliceable or non–sliceable sets based on the connectivities of components and their functionalities. All of the necessary information for each component, such as type, area, and delay, is provided by ICDB. Furthermore, the partitioner folds small bit–sliced units, thereby filling the empty space. The stack–folding is a two–dimensional area filling process that considers both the bit–widths and the heights of the units. Thus, it can alleviate the height mismatching problem that results from abutting two different units horizontally. The bit–sliced stack will also be partitioned into multiple stacks if a better area utilization can be achieved.

The stack placer permutes the bit–sliced units to minimize the routing track density, and the stack router assigns the routing tracks between the connected ports. After the placement step, the stack router assigns the routing tracks to the connected units.

After forming the sliced stack, the glue–logic component binder first estimates the loads for each output pin in the glue–logic module. The loads are calculated by summing the input capacitances of driven bit–sliced units and the routing wire capacitances. In the binding step, the binder forwards the glue–logic netlist, output loads, and delay constraints to the database, and retrieves a netlist of gates with pin information from the database. This netlist also contains the transistor sizes for each component.

The floorplanner uses a constructive method to place the glue–logic around

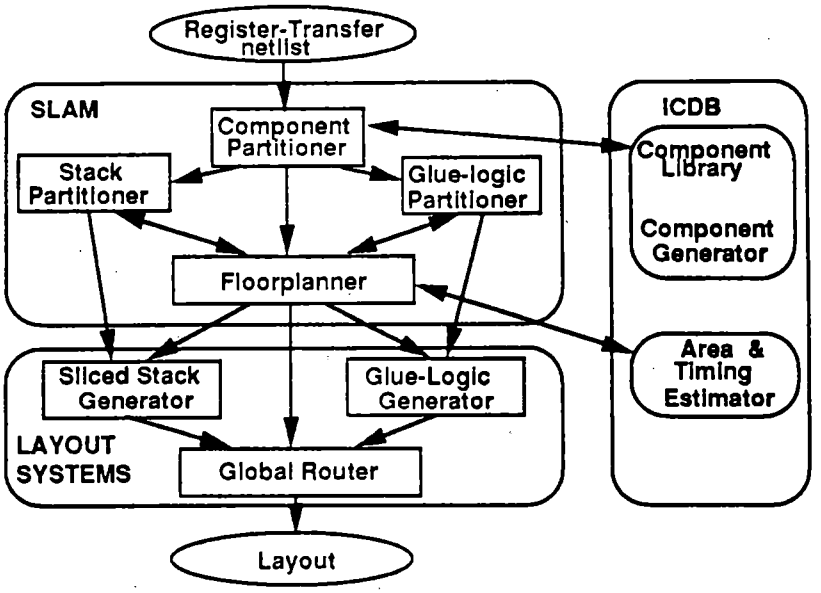


Figure 10: The SLAM Block Diagram

the stack module. The floorplanner determines the aspect ratio and location of the glue–logic module. To achieve minimal layout area, the floorplanner examines different floorplan styles and selects the one with the minimal area for the final floorplan. In addition, the floorplanner determines the ordering of input/output pins for the glue–logic that will minimize the wire crossing between stack and glue–logic modules.

In the final phase, the glue–logic module is generated by a striped layout generator [LiGa89], and the stack module is generated by generators using Mentor Graphics GDT tools. A global router then finishes the detailed routing between modules to generate the final layout.

4.9 VHDL Translators

The selection of a description language or an interface language is difficult in a CAD framework where many tools must communicate and where a user may select to manually synthesize the design.

One approach would be to define a universal language that would adequately describe all different levels and styles of a design.

Such a language should be able to model a design at least on transistor, logic, behavioral, and process levels, and describe adequately analog and digital, synchronous and asynchronous, pipelined and parallel, shared–memory and message–passing, hierarchical and concurrent design styles. Such a language is a

utopia. It would take a hundred years to reach agreement on a standard, and probably another fifty years to teach designers how to use it.

The second approach is to use a standard simulation language such as VHDL for synthesis. Such a language is, however, burdened by constructs necessary for simulation and not for synthesis. Furthermore, simulation models are not adequate for many design styles.

The third approach is to develop languages and intermediate forms for different design aspects and design styles. This way a language stays simple and descriptions closely mimic real designs. Furthermore, those languages must be simulatable or verifiable. That requires writing a new simulator or a translator to a simulation language, such as VHDL.

Note that a designer is not really interested in writing a design description in VHDL. He or she is only interested in finding the output values for a set of test vectors and to observe timing relationships among selected signals. Thus, an output report from any simulator will be satisfactory no matter how the input to the simulator was derived; i.e. from another language or any other form of captured design. A designer should not even know about the existence of any simulation language.

We have developed VHDL translators for BIF (described in Section 4.2) [DuCH91] and SpecCharts (described in section 4.3) [NaVG91]. We found that automatically–generated code is a bit larger in size and less elegant than manually written code but completely readable and quite satisfactory for simulation.

5 Future Research in Synthesis

Although logic and layout tools are available today, the extension of synthesis to higher levels requires solution of several fundamental problems:

1. Languages or intermediate forms for defining verifiable design views on higher levels of abstraction must be developed as well as formalisms to manipulate those views and transform them into design styles suitable for a given technology.

2. Design representations, databases, and tools for disambiguation of descriptions and consistency checking are needed to support high–level synthesis.

3. System–level notation, and algorithms for synthesis using components that include commercially available sub–systems, must follow language and database developments.

4. Technology for component specification and generation (component server) must be developed by combining sequential, logic, and layout synthesis.

6 Acknowledgments

I would like to thank all my present and past students without whom many of the ideas presented in this paper would not have been invented and verified.

I would also like to thank Pat Harris and Carmen Mendoza for their enthusiastic assistance in the production of this paper.

This work has been partially supported by NSF grant #MIP–8922851, SRC grant #90–DJ–146, State of California MICRO grants #90–046 and #90–047, and donations from Rockwell International, Texas Instruments, TRW, Western Digital, Silicon Systems, Inc., S–MOS, Inc., Vantage, and Mentor Graphics Silicon Sytems Division. The author is grateful for their support.

7 References

[ABWS89] T. Amon, G. Borriello, W. Winder and C. Sequin, "A Unified Behavioral/Structural Representation for Simulation and Synthesis," *High Level Synthesis Workshop,* 1989.

[BrGa90] F.Brewer and D. D.Gajski, "Chipee: A System for Constraint Driven Behavioral Synthesis," *IEEE Trans. CAD,* August, 1990.

[BrMS89] O. Bross, P. Marwedel and W. Schenk, "Incremental Synthesis and Support for Manual Binding in MIMOLA," *High Level Synthesis Workshop,* 1989.

[ChGa90] G.D.Chen, D. D. Gajski, "An Intelligent Component Database for Behavioral Synthesis", *Proc. DAC,* 1990.

[DuGa89] N.D. Dutt and D.D. Gajski, "Designer Controlled Behavioral Synthesis," *Proc. DAC,* 1989.

[DuHG90] N.D. Dutt, T. Hadley, D.D. Gajski, "An Intermediate Representation for Behavioral Synthesis," *Proc. DAC,* 1990.

[DuCH91] N.D. Dutt, J.Cho and T. Hadley, "A User Interface for VHDL Behavioral Modeling," *Symp. on Computer Hardware Description Languages,* Marseille, 1991.

[Dutt88] N.D. Dutt, "GENUS: A Generic Component Library for High Level Synthesis," *TR 88–22, U.C. Irvine,* 1988.

[HaGa91] T. Hadley, D.D. Gajski, "Decision Support Environment for Behavioral Synthesis," *TR 91–17,* ICS Dept., U.C. Irvine, 1991.

[JKMP89] R.Jain, K.Kuckukcakar, M.J.Milnar, and A.C.Parker, "Experience with the ADAM Synthesis System", *Proc. DAC,* 1989.

[KiGa90] J.R. Kipps, D.D. Gajski, "The Role of Learning in Logic Synthesis," *Journal of Pattern Recognition and Artificial Intelligence,* June, 1990.

[LaGW91] L.Larmore, D.D. Gajski, C-H.A.Wu, "Placement for Sliced Layout Architecture," *IEEE Trans. on ICCAD,* 1991 (to appear).

[LiGa89] J.Lis, D.D.Gajski, "VHDL Synthesis Using Structured Modeling," *Proc. DAC, 1989.*

[LiGa91] J.S. Lis, D.D. Gajski, "Behavioral Synthesis from VHDL using Structured Modeling," *TR 91-05,* ICS Dept., U.C. Irvine, 1991.

[McPC88] M.C. McFarland, A.C. Parker, R. Camposano, "Tutorial on High Level Synthesis," *Proc. DAC,* 1988.

[NaVa90] S. Narayan, F. Vahid, "Modeling with SpecCharts," TR 90-20, ICS Dept., U.C. Irvine, 1990.

[NaVG91] S. Narayan, F. Vahid, D. Gajski, "Translating System Specifications to VHDL;" *The European Conference on Design Automation,* Amsterdam, 1991.

[PLNG90] R. Potasman, J. Lis, A. Nicolau, D. Gajski, "Percolation Based Synthesis," *Proc. DAC,* 1990.

[PaGa86] B. Pangrle, D. Gajski, "Slicer: A State Synthesizer for Intelligent Silicon Compilation," *Proc. ICCAD,* 1986.

[RDVG88] J. Rabaey, H.DeMan, J. Vanboof, G. Gossens, R.H.J.M. Otten, "CATHEDRAL II: A Synthesis System for Multiprocessor DSP Systems" in *Silicon Compilation,* Daniel D. Gajski, ed., Addison-Wesley, 1988.

[RuGa90] E.A.Rundensteiner, D.D. Gajski, "A Design Representation for High-Level Synthesis," *TR 90-27,* ICS Dept., U.C. Irvine, 1990. 90-27, Sep. 1990.

[RuGB90] E.A. Rundensteiner, D.D. Gajski, L. Bic, "Component Synthesis Algorithm: Technology Mapping for Register Transfer Descriptions," *Proc. ICCAD, 1990.*

[RuGa91] E.A. Rundensteiner, D.D. Gajski, "A Design Data Base for Behavioral Synthesis," *TR 91-16,* ICS, U.C. Irvine, 1991.

[ThBR87] D.E. Thomas, R.L. Blackburn, J.V. Rajan, "Linking the Behavioral and Structural Domains of Representation for Digital System Design," *IEEE Trans.* CAD, January 1987.

[ThDW88] D.E. Thomas, E.M. Dirkes, R.A. Walker, J.V. Rajan, J.A. Nestor, R.L. Blackburn, "The System Architect's Workbench," *Proc. DAC, 1988.*

[VaGa88] N. Vander Zanden, D. Gajski, "MILO: A Microarchitecture and Logic Optimizer," *Proc. DAC, 1988.*

[**VaNG90a**] F. Vahid, S. Narayan, D. Gajski, "Synthesis from Specifications: Basic Concepts," *TECHCON '90*, San Jose CA, 1990.

[**VaNG90b**] F. Vahid, S. Narayan, D. Gajski, "SpecCharts: A Language for System Level Synthesis," *Symposium on Computer Hardware Description Languages,* Marseille, France, 1991.

[**Wayn86**] W. Wolf, "An Object Oriented Procedural Database for VLSI Chip Planning", *Proc. DAC,* 1986.

[**Wolf86**] W. Wolf, "An Object Oriented Procedural Database for VLSI Chip Planning," *Proc. DAC,* 1986.

[**WhON89**] G. Whitcomb, et.al, "The Hardware Data–Flow Representation and Synthesis Methodology," *High Level Synthesis Workshop,* 1989.

[**WuGa89**] C-H.A. Wu, D. Gajski, "SLAM: An Automated Structure to Layout Synthesis System," *TR 89–40*, ICS, UC Irvine, 1989.

[**WuCG90**] C-H.A. Wu, G. Chen, D. Gajski, "Silicon Compilation from Register-Transfer Schematics," *Proc. ISCAS,* 1990.

[**WuGa90**] C-H.A. Wu, D. Gajski, "Partioning Algorithms for Layout Synthesis from Register-Transfer Netlists," *Proc. ICCAD,* 1990.

[**WuVG90**] C-H.A. Wu, N. Vander Zanden, D. Gajski, "A New Algorithm for Transistor Sizing in Transistor Circuits," *Proc. EDAC,* Glasgow, Scotland, 1990.

[**WuGa91**] C-H.A. Wu, D. Gajski, "Glue–Logic Partitioning for Floorplans with a Rectilinear Datapath," *Proc. EDAC*, 1991.

[**YaIs89**] H. Yasuura, N. Ishiura, "Semantics of a Hardware Design Language for Japanese Standardization," *Proc. DAC,* 1989.