ized systems are IBM's HIS (Camposano, Bergamaschi, Haynes, Payer, Wu) and Princeton's PUBSS (Wolf, Takach, Lee) geared towards control intensive applications, in the first case processor-like designs, in the second case collections of small, communicating finite state machines. Birmingham, Gupta, and Siewiorek (U. Michigan and CMU) concentrate exclusively on processor boards to be build from a catalog of existing chip-level components. Digital signal processing applications are dealt with in articles by the IMEC group (Lanneer, Note, Depuydt, Pauwels, Catthoor, Goossens, De Man) and by Casavant, Hwang, and McNall. Last but not least, Gajski (UC Irvine) writes about essential issues in high-level synthesis and some solutions stressing practicality. Many of the chapters in this book describe the results of experiments with large, realistic examples, illustrating how much the field has matured in the past few years.

In each chapter, the reader will find the description of the work of a group that has contributed in some substantial way to the progress of high-level synthesis. Chapters (which are arranged in no special order) are broader in scope than a typical conference paper. They emphasize major concepts and strategies. On the other hand, they include reasonably detailed descriptions of important topics, giving the reader an appreciation of the depth of the work. The articles focus on the present status of the research, including a system description of the tools that have been developed along with the key technical contributions and results. A novice to the field thus will find a comprehensible introduction to the work being done. The active researcher may use the book as a reference for the state-of-the-art in high-level synthesis.

The book should be useful for two audiences: researchers who want to analyze, compare, and contrast the latest work in high-level synthesis; and potential users, both design tool developers and designers, who want to evaluate the possible usefulness of the research efforts described.

Acknowledgements

The idea of this book was born at the 27th Design Automation Conference, in June of 1990. Our major goal for this book was a timely presentation of the latest work in the field, which put unusual demands on both the contributors and publisher. We would like to express our sincere gratitude to the authors for their formidable effort of preparing the material in such a short time. We also thank Carl Harris for his constant support and the staff at Kluwer for their hard work.

Raul Camposano
Yorktown Heights, New York

Wayne Wolf
Princeton, New Jersey

1

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-

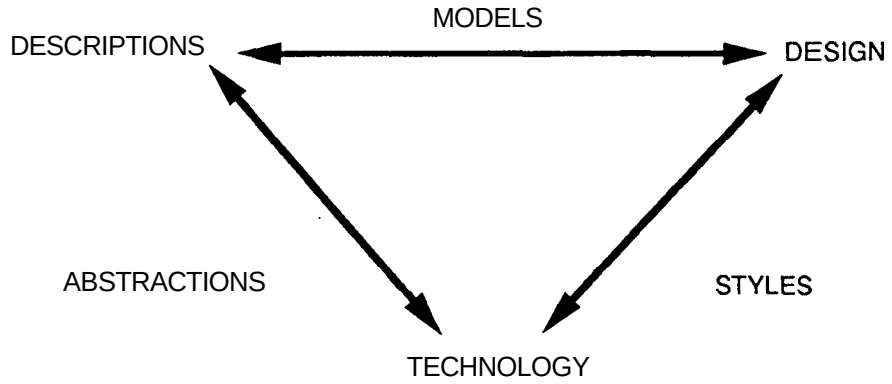

Figure 1: Description–Design–Technology Dependence

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

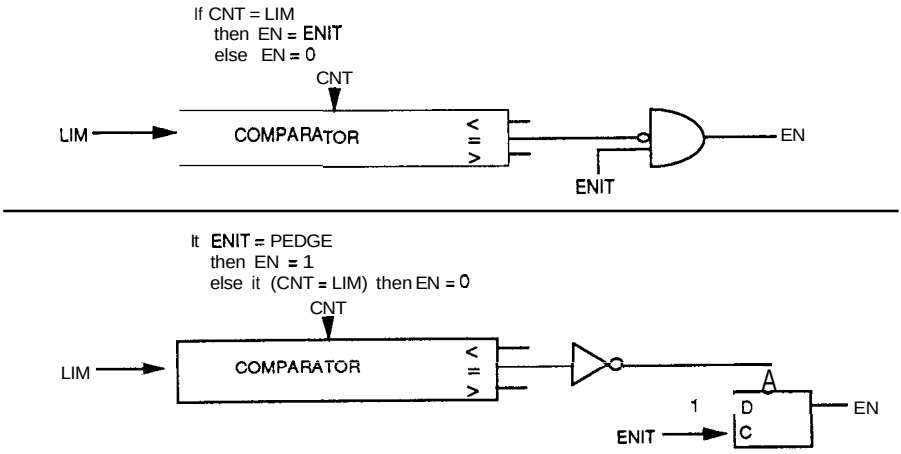

Figure **2:** Two Different Descriptions of an Event (a) level sensitive, (b) edge sensitive

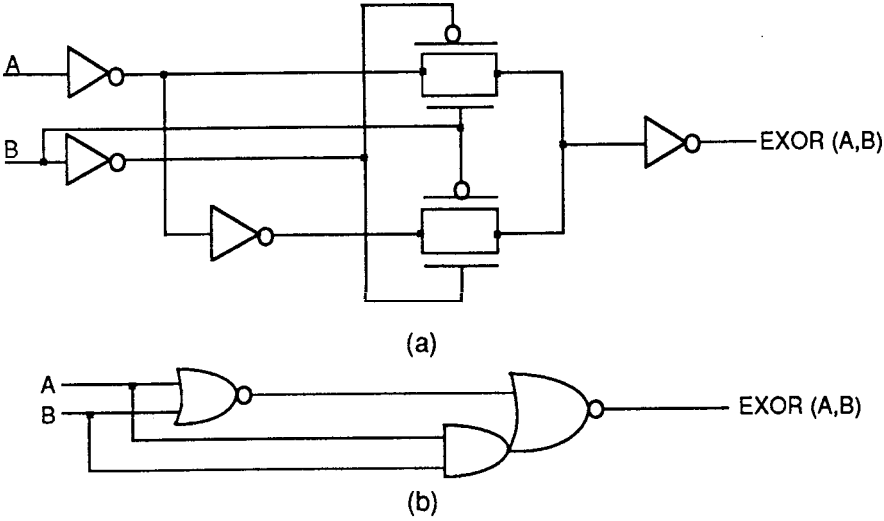

Figure **3:** Two Styles of EXOR Implementation using (a) transmission gates, and (b) AND–OR–INVERT gate.

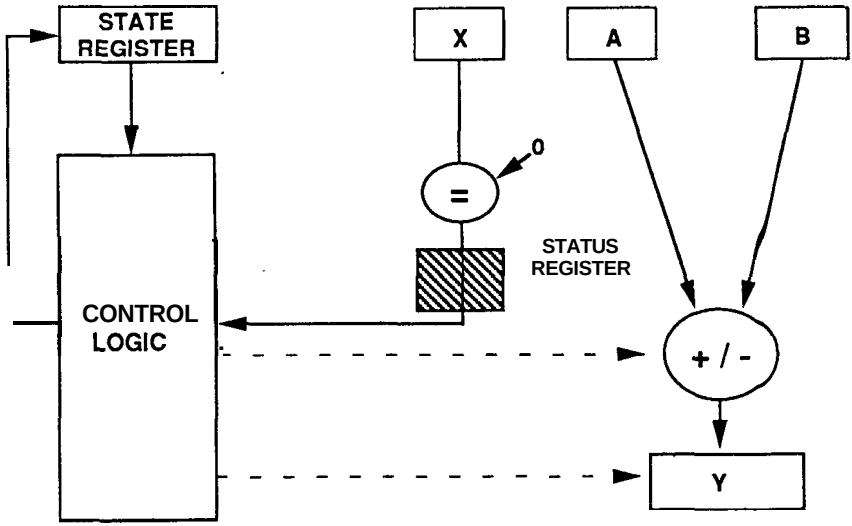

Figure 4: Two Different Design Styles: (a) without status register, and (b) with status register (shaded box)

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

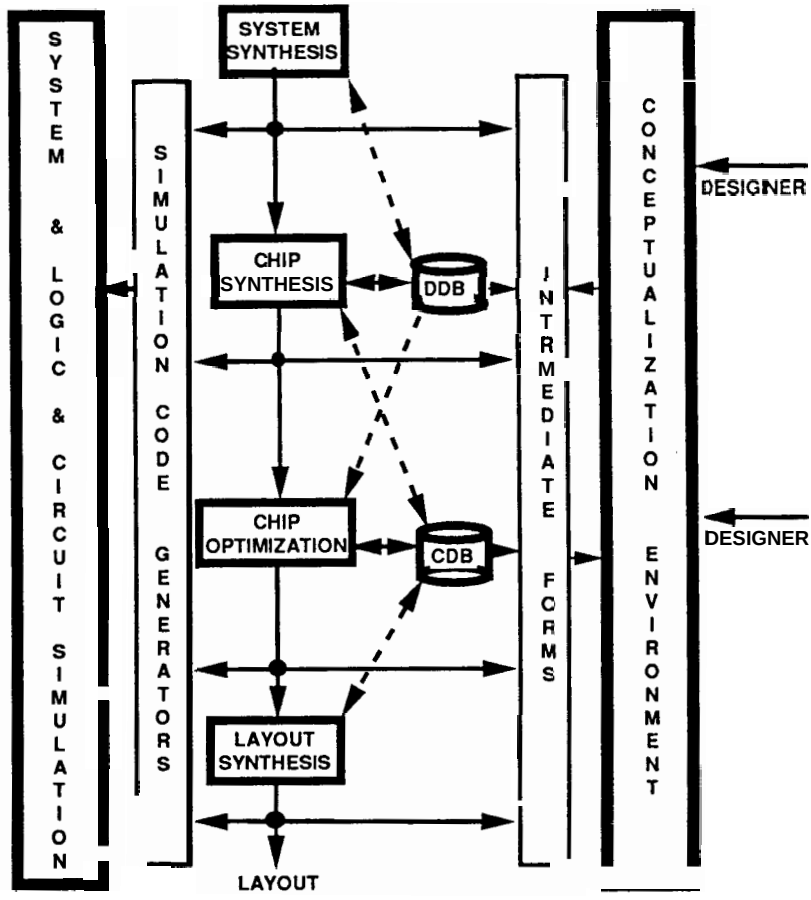

Figure 5: Hypothetical Synthesis System

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

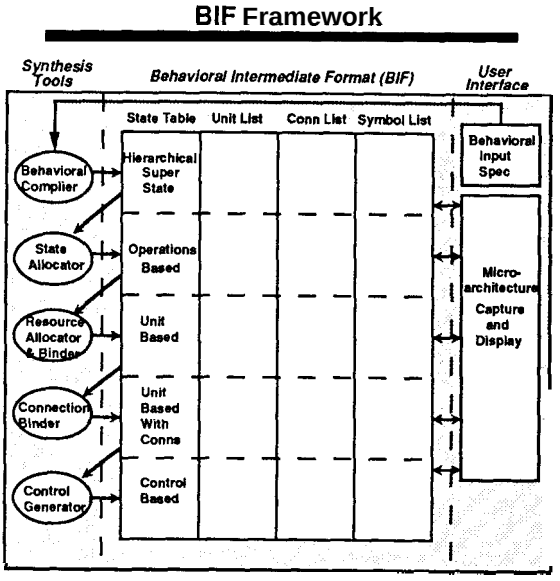

Figure 6: A Canonical Synthesis Environment

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.

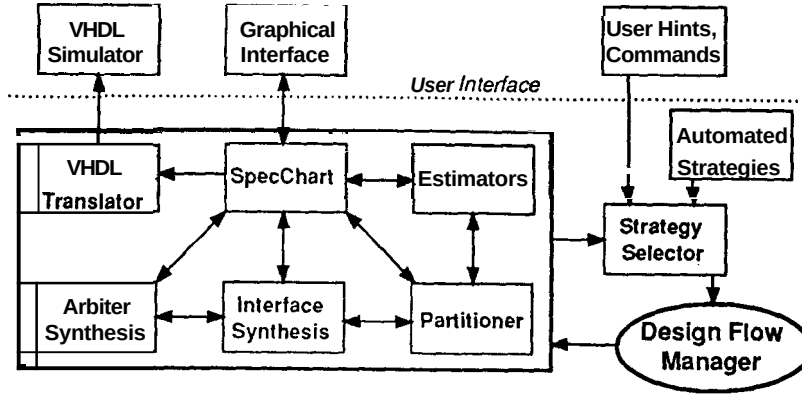

Figure 7: SpecSyn System–Level Synthesis Tool

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.

Give? 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/O 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

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 1/0 ports on the boundary of the module.

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

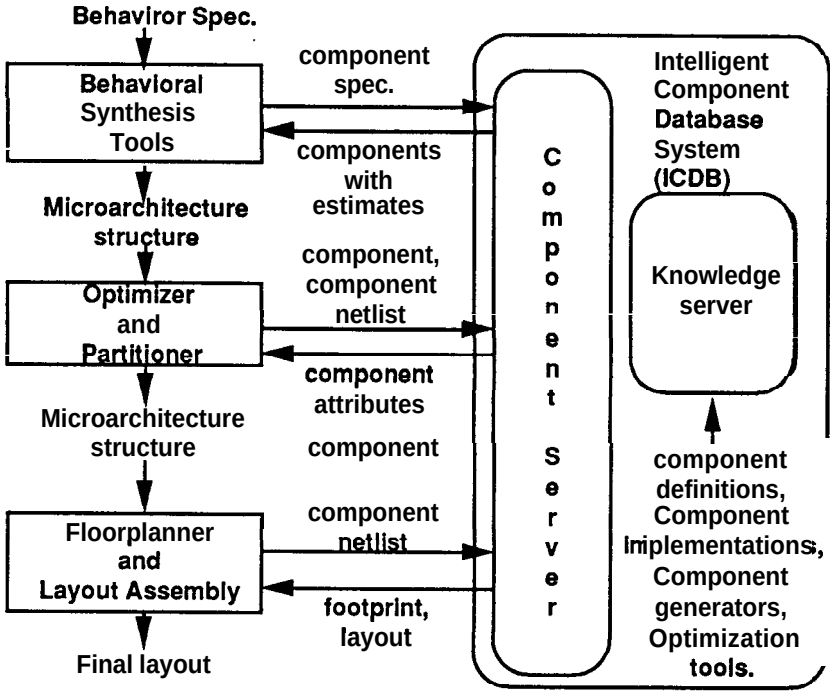

Figure 8: Component Database Role in High–Level Synthesis

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 logicl–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

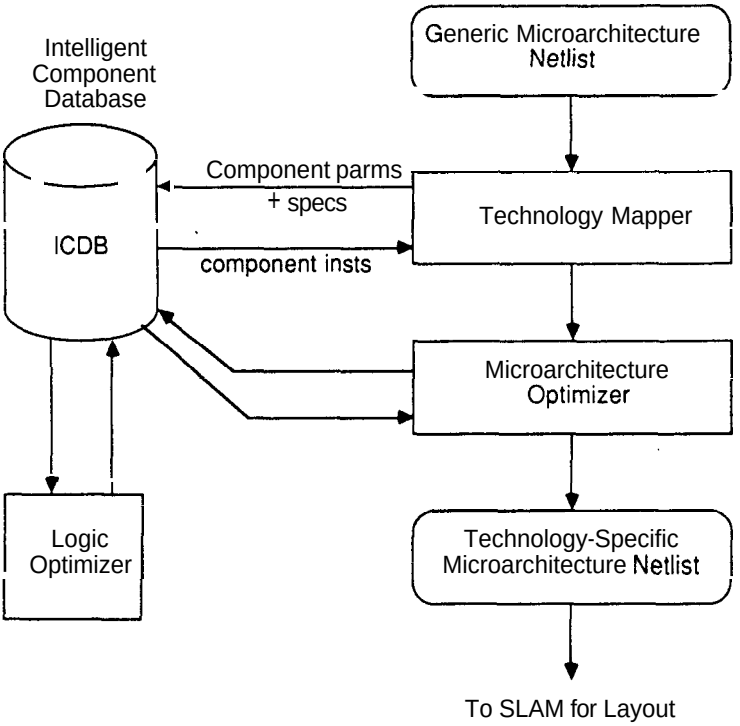

Figure 9: MILO System for Chip Optimization

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 **SUB-**TRACT 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 1/0 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

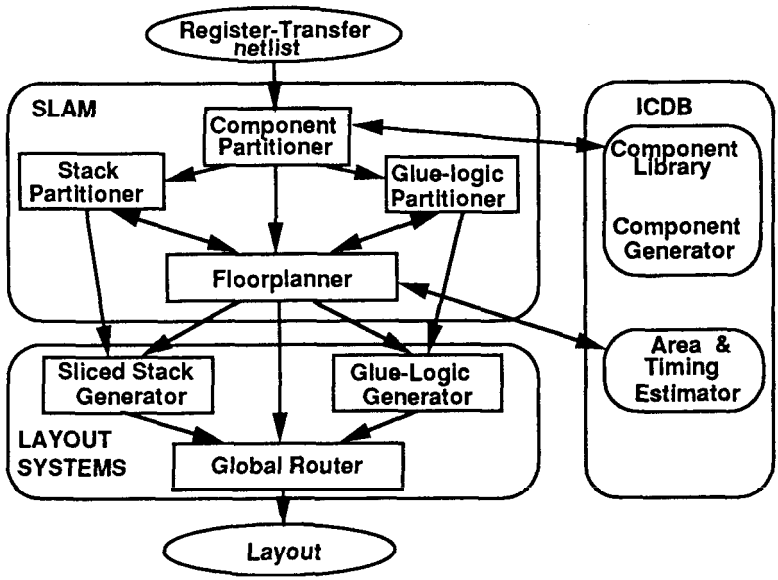

Figure 10: The SLAM Block Diagram

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. DA*C,* 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," T R 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.

2

Architectural synthesis for medium and high throughput signal processing with the new CATHEDRAL environment

Dirk Lanneer Stefaan Note Francis Depuydt
Marc Pauwels Francky Catthoor Gert Goossens
Hugo De Man[1]

IMEC, Kapeldreef 75, B-3001 Leuven, Belgium
Phone : +32/16/28.12.01
([1] Professor at Katholieke Universiteit Leuven)

1 Introduction

Integrating highly complex signal-processing systems in an application-specific way is becoming an economic necessity for system industry today. In recent years, research in high-level or architectural synthesis has therefore attempted to bridge the existing gap between systems and (custom) architectures. This effort has been complementary to the work on physical integration, which bridges the gap from detailed architecture to final implementation in, for instance, a custom ASIC, a sea-of-gates or a PLD realisation. The ability to handle real-life design problems is however still one of the most important challenges for researchers in these fields.

This paper describes the new **CATHEDRAL** environment for automated synthesis of IC architectures for real-time signal processing **(DSP),** in the medium to high throughput range (1 kHz to 20 MHz sample frequency). In particular we address applications with an irregular signal-flow in domains like audio, speech, telecommunication, image processing and video. The most important features of this environment are the use of a common synthesis framework and the introduction of global synthesis scripts which are tuned towards different architectural styles. The approach will be demonstrated with a practical audio application.

In order to be acceptable in a practical context, a compiler for architectural synthesis should satisfy the following requirements :

[4] F. Yassa, J. Jasica, R. Hartley, and S. Noujaim, "A silicon compiler for digital signal processing: Methodology, implementation, and applications," Proceedings of the IEEE, vol. 75, pp. 1272–1282, September 1987.

[5] R. Hartley and A. Casavant, "Tree-Height Minimization in Serial Architectures," in *ICCAD-89*, pp. 112–115, 1989.

[6] K. Hwang, A. Casavant, M. Dragomirecky, and M. d'Abreu, "Constrained Conditional Resource Sharing in Pipeline Synthesis," in *ICCAD-88*, pp. 52–55, 1988.

[7] N. Park, Synthesis of High-speed Digital Systems. PhD thesis, University of Southern California, October 1985.

[8] K. Hwang, A. Casavant, and M. d'Abreu, "Scheduling and Hardware Sharing in Pipelined Data Paths," in ICCAD-89, pp. 24–27, 1989.

[9] K. McNall and A. Casavant, "Automatic Operator Configuration in the Synthesis of Pipelined Architectures," in 27th Design Automation Conference, pp. 174–179, 1990.

[10] P. M. Kogge, The Architecture of Pipelined Computers. Hemisphere Publishing Corporation, 1981.

[11] P. Paulin and J. Knight, "Force-directed scheduling in automatic data path synthesis," in 24th Design Automation Conference, pp. 195–202, 1987.

[12] S. H. Unger, "Tree realizations of iterative circuits," IEEE Transactions on Computers, vol. C-26, pp. 365–383, April 1977.

[13] T. L. Morin and R. E. Marsten, "An algorithm for nonlinear knapsack problems," Management Science, vol. 22, pp. 1147–1158, June 1976.

[14] P. M. Melliar-Smith, "A Graphical Representation of Interval Logic," in CONCURRENCY 88, International Conference on Concurrency (F. H. Vogt, ed.), pp. 106–120, Springer-Verlag, 1987.

[15] T. Amon, G. Boriello, and W. Winder, "A Unified Behavioral/Structural Representation for Simulation and Synthesis," in Fourth International Workshop on High-Level Synthesis, IEEE, October 1989.

[16] D. Ku and G. D. Micheli, "Relative scheduling under timing constraints," in 27th Design Automation Conference, ACM/IEEE, June 1990.

[17] F. Kurdahi and A. Parker, "Wiring Space Estimation For Standard Cell Designs," Tech. Rep. DISC/84-5, University of Southern California, Digital Integrated Systems Center, November 1984.

4

The IBM High-Level Synthesis System

R. **Camposano**[1], R.A. **Bergamaschi**[1],
C.E. **Haynes**[2], **M. Payer**[1] and **S.M. Wu**[3]

[1] *IBM Thomas J. Watson Research Center*
Yorktown Heights, NY

[2] *IBM ABS*
Rochester, MN

[3] *IBM EDS DAZ*
Poughkeepsie, NY

1. INTRODUCTION

The High-level IBM Synthesis system **HIS** is the result of ongoing efforts at the T.J. Watson Research Center, the Advanced Business Systems Division and **IBM's** Electronic Design Systems. The main goal is to explore design automation for synchronous digital systems at levels above the logic level in a practical environment.

HIS has its roots in the Yorktown Silicon Compiler (YSC) [6, 7, 9, 10]. Although **YSC's** emphasis was on lower design levels than HIS, **i.e.** logic and physical design, it succeeded in synthesizing an 801 processing unit from a high-level, sequential description, thus demonstrating the feasibility of high-level synthesis for processor-like designs in our environment. The YSC research project was ended in 1987. By 1988 it was decided to build a high-level synthesis system using a VHDL subset as its input language [16]. Three sites within IBM were involved to emphasize practicality in a real design environment. Work since then has progressed to the point that the system called HIS is essentially completed.

HIS uses some ideas pioneered in the YSC, specially in the area of path-based scheduling [12]. Most algorithms however are new, improving the quality of the resulting designs. Moreover, HIS puts much more emphasis on practicality. VHDL [26], by now an established standard, was chosen as the input language

in the hope that describing real designs in VHDL would be part of IBM's design methodology in the near future. To integrate the system into the existing design environment, the output language BDL/CS [33] serves as the interface to the Logic Synthesis System LSS [20] and to various other tools such as simulation and Boolean verification. Figure 1 shows how HIS is integrated into the overall design environment.

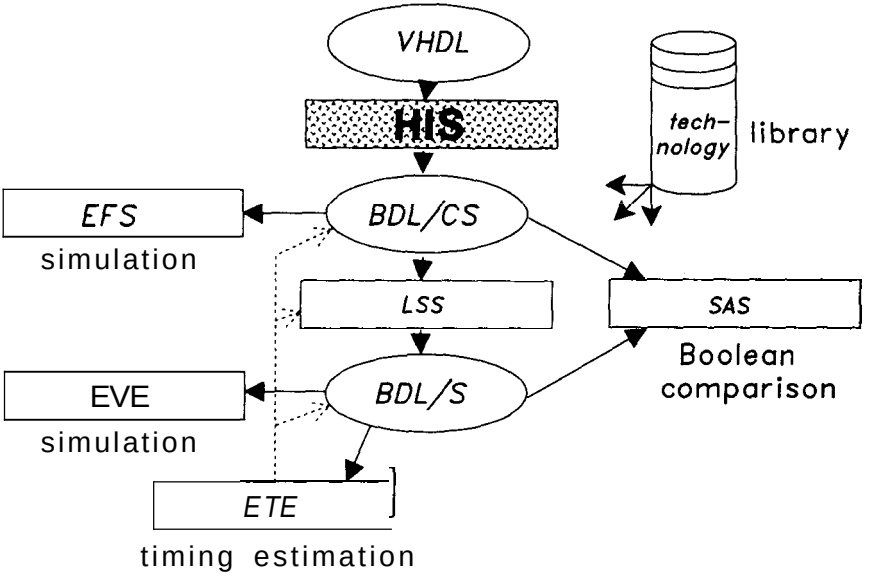

Figure 1. **The Design Environment**

Great care was also taken to implement a system that is fast enough to allow the synthesis of large designs in a reasonable time: HIS is two orders of magnitude faster than the YSC. A M6502 like microprocessor can be synthesized in a few minutes; smaller examples like the well-known 5th order elliptical filter [22] run in seconds.

The function that HIS covers is in essence high-level synthesis of synchronous digital systems, as defined in [31]. Its input is a sequential specification of the function of a synchronous digital system, and possibly constraints such as the number and/or the type of the hardware modules to use, timing constraints, etc. HIS assigns every operation in the specification to a control step (scheduling) and synthesizes the necessary hardware (allocation). The resulting design

consists of a finite state machine (FSM) that implements the control and a **netlist** specifying the data path.

This paper describes the algorithms used in HIS and gives results for a wide selection of designs done with HIS. The next section briefly addresses the **in-**core data base implemented for HIS and gives an overview of the system. Section 3 deals with data-flow analysis, necessary for the following steps. The scheduling algorithm **is** described in section 4. Module assignment, performed during scheduling, is explained in section 5. Section 6 shows the data-path allocation algorithms. Section 7 is dedicated to further tasks, essentially to the delay and size estimation, the logic minimization in HIS, the output language generation and the user interface. Results for various benchmark designs are given in section 8. The paper ends by drawing conclusions.

2. THE SYNTHESIS IN-CORE MODEL

In High-Level Synthesis, the design specification given in a high-level language is usually compiled into a graph representation consisting of a control and a data-flow graph. Examples of such representations are **CMU's** Value Trace **[30], USC's** Design Data Structure **[28],** the Yorktown Internal Format **[7], Irvine's** Behavioral Intermediate Format **[23],** etc. Unlike more general frameworks or databases, these formats are geared specifically towards the representation of behavior and high-level synthesis.

HIS uses a design representation called the SSIM (Sequential Synthesis In-core Model) **[17].** The SSIM represents control and data separately. Before high-level synthesis, the SSIM contains only the behavior of a design, **i.e.,** a control-flow graph (CFG) and a data-flow graph (DFG). An example is given in figure 2.

The CFG is a directed graph defined as CFG **=** (N, P), where the nodes *N* represent the operations such as assignment, addition, logical and, etc., and the edges P represent the precedence relation. The CFG represents the basic control constructs encountered in a sequential language (such as behavioral VHDL):

- Sequence. An edge $(n_1, n_2) \in$ P means that n_2 is executed after n_1. In figure 2, edge **(8,9)** is an example of required sequential behavior.
- Conditional execution. If an operation has more than one successor, exactly one of them is executed next. The selection of the successor in this case depends on a condition attached to the edge. A condition is a Boolean expression which evaluates to 1 if the next operation is to be executed and to **0** otherwise. In figure 2, operations 4 and 7 branch out to different successors conditionally. The conditions are indicated on the edges.

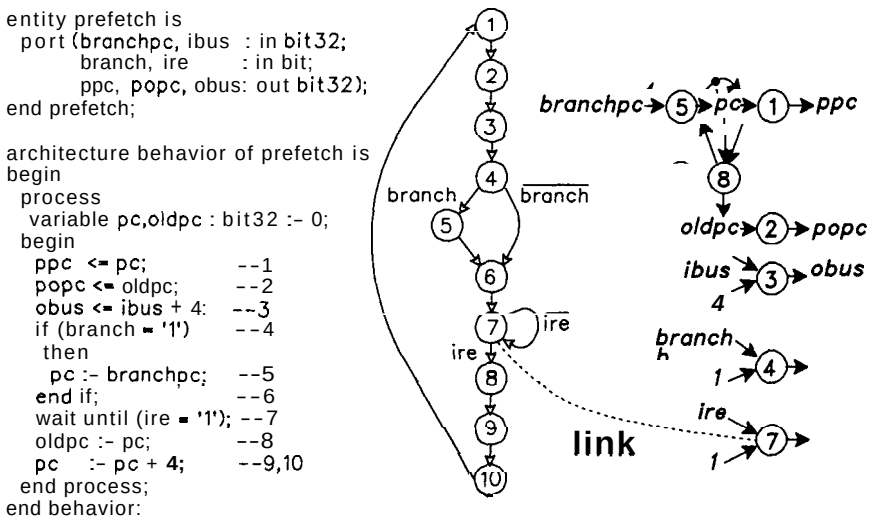

Figure 2. VHDL specification and the corresponding **CFG** and **DFG**

- Iteration. The CFG may contain cycles which indicate iterative behavior (loops). In figure 2, there are iterations at operations 7 and 10.

The DFG is a directed graph defined as DFG = (NU V, D). The nodes N are the operations, and V are the variables. Edges D indicate the data dependencies, **e.g.,** in figure 2 operation 1 usespc as an input andppc as an output. Notice that the DFG may be disconnected. Operations in the DFG and the **CFG** are the same and they are linked to indicate this.

High-level synthesis adds the design structure to the SSIM. Scheduling consists basically in control synthesis and adds a control *finite* state machine *(FSM)* to the SSIM. The synthesized FSM is represented by its state transition graph. Allocation consists basically in data-path synthesis and adds a data path to the SSIM. The synthesized data-path is also represented as a graph (netlist).

The SSIM is hierarchical, in the sense that several modules (processes, subprograms) can be maintained at the same time and synthesized separately. A module hierarchy graph represents the calling relations among the different modules. Figure 3 summarizes the SSIM design representation.

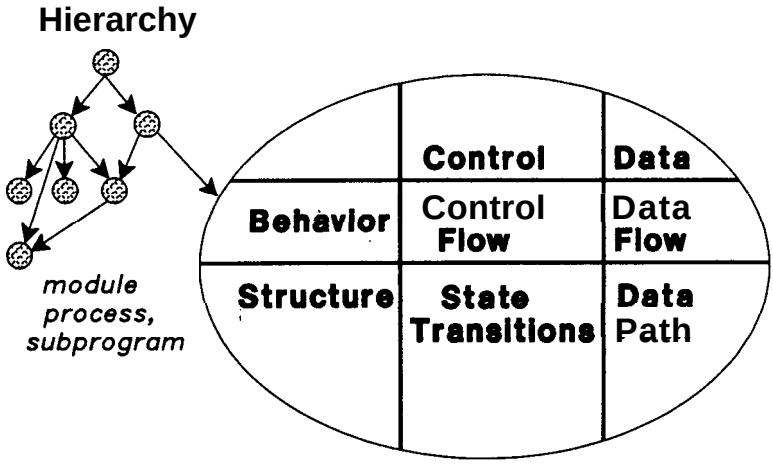

***Figure* 3. The Sequential Synthesis In-Core Model SSIM**

The complete SSIM resides in memory during high-level synthesis, thus achieving the necessary speeds for design space exploration and interactive design.

3. DATA-FLOW ANALYSIS

Data-flow analysis is a technique to determine the *lifetime* of values, which is used extensively in compiler construction [2]. High-level synthesis often uses data-flow analysis to determine the intended behavior, for example, unfolding variables [15, 38]. In HIS, lifetime information is used during allocation and scheduling.

HIS uses a modified global data-flow analysis based loosely on du-chaining (definition-use) [2]. Since most high-level synthesis literature does not address this step in detail and often restricts it to so called basic blocks (e.g., [38]), and since data-flow analysis can be computationally expensive, we include the exact algorithm used below. It first computes the reachability for each value assigned and then its lifetime. HIS performs global data-flow analysis not restricted to basic blocks. In this case the control flow (conditional execution and iteration) **must** be taken into account!

A value, denoted *val*, is defined as an assignment to a variable, $val = (op,v) \in D$, where $op \in N$, $v \in V$. A variable may be assigned multiple

values. The reachability, $reach_{val,q}$, of a value, val = (op, v), at an operation, q $\in$ N, is a Boolean value defined as 1, if, in the CFG, there is a path from op to q with no other assignment to the same variable. Otherwise, the reachability is 0. It is computed by the following algorithm:

1. Vval, q: $reach_{val,q} = 0$
2. Compute in depth-first order, until there is no change

$$reach_{val,q} = \bigvee_{r:(r,q)\in P} [(reach_{val,r} \vee (r = op)) \wedge (\overline{(r,v) \in D \wedge r \neq op})]$$

All reachabilities can be computed in parallel. The repetition of the second step is necessary due to iterations. Given n nodes in the CFG, the depth first search will be repeated at most n times. The depth first search itself is of complexity $O(n^2)$ and at each step at most n reachabilities (for each value, assuming each operation assigns at most one value) must be computed. The worst-case complexity thus is $O(n^4)$. But since the CFG has a small outdegree on average, and not many cycles, in practice the average complexity is only slightly worse than $O(n^2)$.

A value is alive at an operation q, if it reaches q and is used at q or later. The lifetime $life_{val,q}$ of value val at operation q is computed by the following algorithm:

1. Vval, q: $life_{val,q} = 0$
2. Compute in reverse depth-first order, until there is no change

$$life_{val,q} = reach_{val,q} \wedge ((v,q) \in D \vee \bigvee_{r:(q,r)\in P} [life_{val,r}])$$

The computational complexity of computing lifetimes is the same as that of computing reachability.

Only lifetimes are stored in the SSIM. They is attached as a bit vector to each control flow node, where each bit represents one value. A value being alive at an operation *op* indicates that it used as an input to *op* or to any succeeding operation. In figure 2, for example, the value (5,*pc*) is alive in operations 6, 7, 8, 9, while the value (10,*pc*) is alive in operations 1, 2, 3, 4, 6, 7, 8, 9.

4. AS-FAST-AS-POSSIBLE SCHEDULING

The scheduling approach in HIS differs from most scheduling techniques traditionally used in high-level synthesis (e.g., list scheduling [21], force-directed scheduling [35]). HIS uses As-Fast-As-Possible (AFAP) scheduling [11] for

each possible path in the CFG. A path in the CFG is defined by conditional branches; thus one path represents one possible execution determined by the conditions on the conditional branches. For example, in a microprocessor, a path represents a particular instruction with an addressing mode. Although the number of paths may grow exponentially with the number of nodes, the number of paths represents the number of different functions in the circuit, which is usually bounded. Optimizing the length of all paths (length in number of control steps), means minimizing the number of cycles in each instruction. AFAP scheduling emphasizes conditional branching rather than potential parallelism like in list or force-directed scheduling; we believe that in control dominated design and processor design this is the main problem.

Paths are computed by performing a depth-first traversal of the acyclic CFG. The CFG is made acyclic by removing the feedback edges in loops, e.g., in figure 2 edges (7,7) and (10,1) are removed. Paths are computed starting at a given first operation (operation 1 in the example). Since loops represent repetitions of sequences of operations, paths starting at the first operation in loops must also be considered. Figure 4a shows the three paths for the example of figure 2.

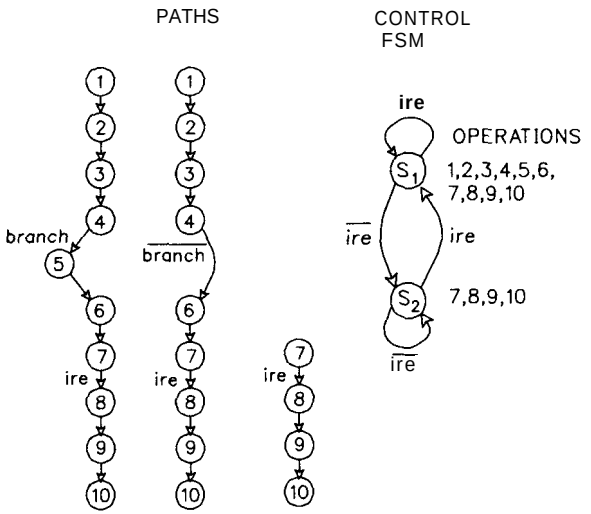

***Figure* 4. Paths (a) and Control FSM (b)**

4.1 Constraints

An implementation (and thus the schedule) is usually restricted by so called ***hardware constraints.*** These restrictions may be inherent to the implementation (in this case, synchronous digital systems) or may be explicitly specified. Examples of constraints inherent to the implementation are: 1/0 ports can either transmit or receive only one distinct value per control step, or registers can be written only once per control step. Constraints explicitly specified often indicate area and delay characteristics of the desired implementation, for example, the maximum clock cycle or the maximum number and type of the functional units to be used. Constraints are represented as ***path intervals*** in the ***CFG.*** A ***constraint interval*** represents a sequence of operations that have to be executed in two cycles, i.e., a new control step must start at some point within the interval. Figure 5 gives an example of constraints along a path. Constraint 3, for example, indicates that a new control step must begin at operation 5, 6, 7, or 8 because there is a conflict between operations 4 and 8 (e.g., both use a unique hardware resource such as an adder). Notice thnt in figure 4a there is no constraint even though ***pc*** is written in operations 5 and 9. No constraint is necessary for variable loading - instead the correct value is selected by a multiplexer as will be shown later.

If a maximum clock cycle is specified, then a constraint interval is derived by adding the delays of the data-dependent operations along a path, starting with the first operation and ending at the operation which exceeds the cycle limit. This process is repeated, starting with subsequent operations on the path, defining further constraint intervals. Since accurate cycle time predictions require exact timing models which in our experience, are impossible at this high-level, timing constraints should be used only as an admittedly imprecise guideline.

Constraints on the number of single-function functional units (e.g., the maximum number of subtracters) are derived in a way similar to timing constraints. Multi-function functional units, require a more elaborate constraint generation based on a previous module assignment step which will be explained in the next section.

Constraint intervals depend on the order of the operations in the control-flow graph. Since data-independent operations may be arbitrarily ordered, any valid ordering is possible. AFAP scheduling just uses the given order, which may be obtained, for example, by using a list scheduler.

4.2 Path Scheduling and Control Synthesis

Once the paths and the constraints are computed, scheduling is performed on each path independently. A path is scheduled by *cutting* it at different points; each cut represents the start of a new control step. Clearly, the minimum number of cuts will schedule the path in the minimum number of control steps (AFAP). To satisfy all constraints, each constraint must be covered by a cut. The problem of finding the minimum number of cuts that cover all constraints can be modelled as a clique covering problem, as indicated in figure 5. The constructed graph, where each constraint is represented by a node and overlapping constraints are joined by edges, is an interval graph. Hence, the problem of finding the minimum set of cuts which satisfies all constraints can be solved in linear time if the constraints are ordered. Note that at this point each cut may still comprise a range of nodes (e.g., cut 2 in figure 5).

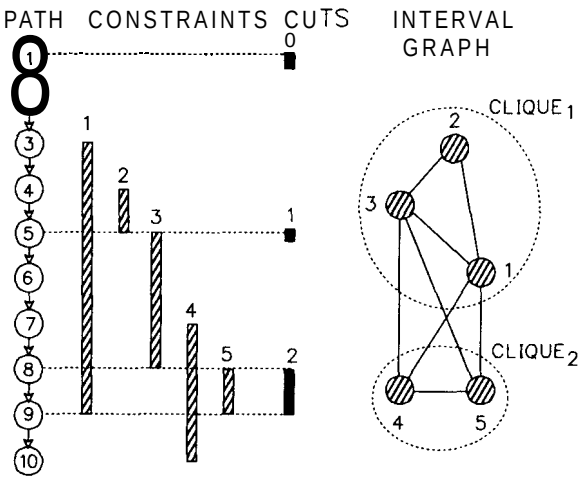

Figure 5. **Scheduling one Path**

The schedule for the complete design is obtained by combining the schedules for all paths. Cuts in different paths may overlap at common nodes. These cuts may be merged very much like constraints were merged above. Again the problem can be modelled as a clique covering. Each cut is represented by a node and overlapping cuts are joined by edges. The complexity of the computation of the minimum clique cover in this step is NP, since the resulting graph does not seem to have any special property. In practice, this graph is reasonably sparse so that exact algorithms can be used. In figure 4 only two final cuts re-

sult: cut_1 at operation 1 which starts both $path_1$ and $path_2$, and cut_2 at operation 7 which starts $path_3$.

After this step a cut may still contain more than one operation. In this case, any single operation within the cut can be selected as the operation to start a new control state. This selection may affect the final number of registers and the performance of iterations. The number of registers for a cut is given by the number of values that have to be stored. All values generated in one control step and used in a later control step must be stored. This number is given by the number of values alive at the fist operation in a control step which is determined by the position of the cut. Since data-flow analysis computed lifetimes for all values and all operations, it is enough to inspect each operation in the cut and pick the one with the least values alive to minimize the number of registers. Iterations are optimized by selecting a cut outside the loop body whenever possible (otherwise an extra control step is being introduced for each iteration).

The final control FSM is constructed as follows. Paths are divided by their cuts into path segments, which are placed in successive control steps. Path segments from different paths which share the same first operation are merged into the same state. The state transitions and their conditions are derived from the edges that join path segments and by logically ANDing the conditions on the control-flowedges leading from the first operation in a path segment to the first operation in the succeeding path segment. If multiple paths are merged into the same state, the corresponding conditions are logically ORed.

Figure 4 shows a finite-state machine example. The AFAP scheduling in this case requires some operations to be scheduled in more than one state, e.g., operations 7, 8, 9 and 10. More details on AFAP scheduling can be found in [11].

5. MODULE ASSIGNMENT

The module assignment problem arises when operations can be implemented by more than one type of functional unit (FU). Module assignment selects the FU to implement each operation. To illustrate this, consider the scheduling example in figure 6a. Given constraints specify that the final implementation should contain at most one FU of each type *FU1,* FU2 and FU3. Figure 6b shows all possible assignments of operations in the CFG to the available FUs. Three operations can be implemented by more than one FU. Figure 6c and 6d show two possible assignments along with the generated constraint intervals using only one of each FUs. Clearly, the assignment in figure 6d is better since

the two constraints overlap, generating only one cut. As a comparison, selecting the assignment of figure 6c requires two cuts to satisfy both constraints.

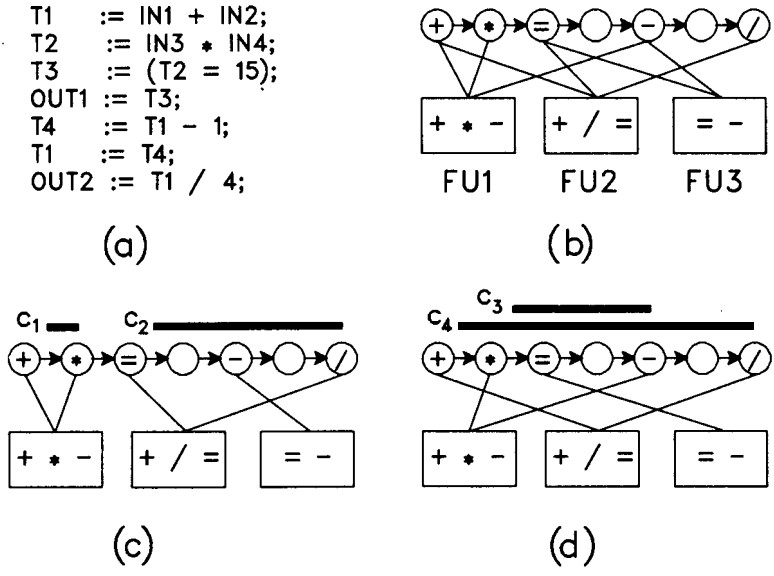

***Figure* 6. Example (a), all possible module assignments (b), non-optimal assignment (c), optimal assignment (d)**

The module assignment for a single path, such that the resulting schedule has the minimum number of states, can be solved optimally with a polynomial time algorithm (by applying repeatedly the *maximum* matching algorithm for bipartite graphs). In the presence of multiple paths however, the (optimal) assignment for an operation present on different paths may not be the same for all paths. We know of no efficient optimal way to solve the multiple path assignment problem, hence we developed the following heuristic. It avoids or delays the introduction of constraint intervals as much as possible. Fewer constraint intervals usually result in fewer states.

For a functional unit, FU, a path, p, and an operation, op, we define:

1. $N(FU)$ is the maximum number of instances of FU that can be used.
2. $F(FU)$ is the number of different functions that the FU can perform.
3. $U_p(FU)$ is the number of times FU has been used in path p

4. $\Delta_p(FU) = N(FU) - U_p(FU)$. If $\Delta_p(FU) > 0$, FU has not been fully used and it can be assigned to an operation without creating a constraint interval. In this case, the FU is considered ***available.***

5. $A(op)$ is the number of functional units which can implement the operation op and are available. $A(op)$ is called the ***availability*** of operation op.

The number of instances of FUs, and the functions they can perform, are constraints and are fixed for each functional unit. $U_p(FU)$, $\Delta_p(FU)$, and $A(op)$ may change after each assignment during the execution of the algorithm.

Given the set of all control-flow paths and the set of functional units allowed in the implementation, operations are assigned to functional units in sequential order, one path, p, at a time, as follows:

1. Sort the operations op in p in increasing order of $A(op)$. Operations with equal $A(op)$ keep their relative order in the CFG.
2. Assign operations to FUs in this order, one at a time, in the following way:
   a. Compute $\Delta_p(FU)$ for all FUs which can implement the operation.
   b. Select the FU with the largest $\Delta_p(FU) > 0$. If two or more functional units have the same $\Delta_p(FU) > 0$, select the one with the smallest $F(FU)$.
   c. If no FU with $\Delta_p(FU) > 0$ exists, then any assignment will cause the generation of constraint intervals. In this case, use the following three criteria in the given order to select an FU:
      1) the FU which generates the largest (less constraining) constraint interval,
      2) the FU with the smallest $|\Delta_p(FU)|$ (to balance FU use),
      3) the FU with the smallest $F(FU)$, i.e., the one with the smallest number of functions since it may be more difficult to use FUs with fewer functions in later assignments.
3. After all operations with the same $A(op)$ have been assigned, recompute $A(op)$ for the operations affected by these assignments and repeat steps 1 and 2, until all operations have been assigned.

The complexity of the module assignment algorithm for one path is derived as $O(n^2 \times f)$, where n is the number of nodes in the path and f is the number of functional units. Further details are given in [3].

In HIS, module assignment is performed during constraint generation, prior to scheduling (and allocation). In this way, all further synthesis steps can be performed with given modules, regardless of the library used.

6. PATH-BASED ALLOCATION

Three main allocation techniques used in high-level synthesis can be distinguished: heuristics (often combined with scheduling) such as force-directed scheduling and allocation [35] or greedy techniques [25], linear integer programming formulations [24, 29], and clique covering (or coloring) based allocation [39]. The allocation approach in HIS [4] falls into the third category and is performed after scheduling.

Allocation is performed in two separate steps. First, a ***complete, functional*** initial data path is generated. The initial data path is then optimized in a second step. Optimizations are based on the well known clique covering [39], and coloring [19] approaches.

6.1 Initial Data-Path Allocation

A data path is represented by boxes, nets and PLAs. Boxes define pieces of hardware such as adders, multiplexers, registers, etc. Nets interconnect boxes. PLAs are used to represent control information in a compact form. Inputs to a PLA are nets; the output of a PLA is used to generate control information. Figure 7 shows the initial data-path for the example in figure 2.

The initial data-path is created by:

1. Allocating functional units,
2. Allocating registers,
3. Defining the interconnections (multiplexers) between functional units and registers,
4. Deriving the control signals, i.e. load enable for registers, operation selection for functional units and selection for multiplexers.

Functional Units

Functional units are allocated initially by simply generating one single-function functional unit box for each operation in each path segment in the finite-state machine. An operation present in multiple path segments is mapped onto the same functional unit. A path segment was defined as the portion of a path that is scheduled in one control step (section 4.2). Path segments are simply called paths for the rest of this section.

Registers

Registers are allocated to store values which are generated in one control step and used in another control step. These values are determined using lifetime information generated during global data flow analysis. Each value ***alive*** at the

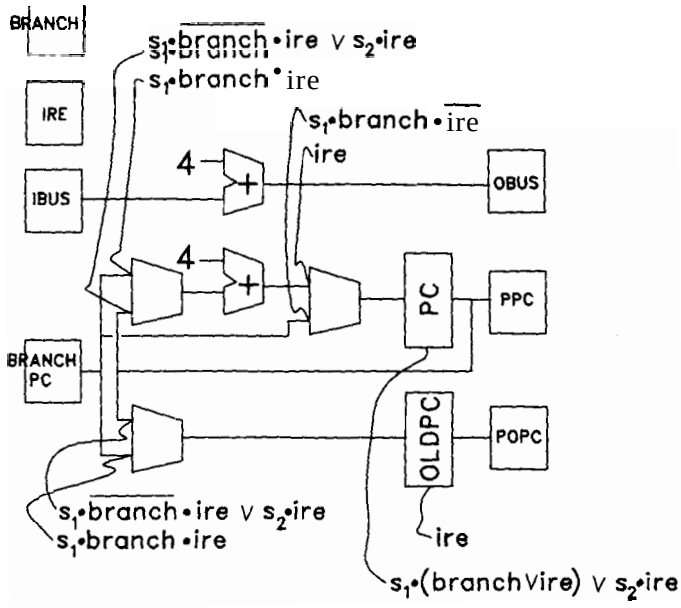

Figure 7. **Initial data-path**

first operation in a state was generated in some previous state, thus it must be stored. Assignments of values to the ***same variable*** share the same register rather than ***unfolding*** them into several registers. This is important for three reasons:

1. It ensures that stored variables in the original specification are kept as ***one*** register. This makes understanding of the synthesized design much easier.
2. The initial number of registers is much smaller than unfolding all variables.
3. If each assignment to a variable is mapped onto a separate register, it may be necessary for AFAP schedules to explicitly store control information, increasing the controller cost (see [4] for a more detailed explanation).

Storing all values assigned to a variable in the same register requires a careful analysis. Figure 7, for example, shows that ***pc*** can be loaded with three different values, i.e., ***pc*** + 4, ***branchpc*** and ***branchpc*** + 4. This is necessary to be able to potentially execute each path in just one cycle. Path analysis and lifetime information are used to decide what value to use if the variable is stored in a register.

Interconnections

Interconnections consist of multiplexers and nets. Initially, nets are used to interconnect functional units and registers as indicated in the original specification (the data-flow graph). Multiplexers are used for nets with more than one source. Path analysis and lifetime information are also used to generate the correct interconnection.

As an example, consider the assignment of pc (figure 2, figure 4 and figure 7). On the first path, pc must be loaded with branchpc (if ire is false) or with branchpc + 4 (if ire is true). In the second and third paths, pc is loaded with $pc + 4$ if ire is true. This information is given by the values that are alive on each path.

Control **Signals**

During allocation, the computed control information consits of load signals for registers, function selection signals for ALUs (multi-function FUs), and selection signals for multiplexers. Control signals are generated computing a Boolean condition for each operation, i.e., the operation executes if the condition is true. These conditions are computed for one path *ANDing* all conditions along the path. Operations along the path are numbered 1, 2, 3...Given c_{op_i} (the condition at operation op_i), $c_{(op_i, op_{i+1})}$ (the condition at the control-flow edge between op_i and op_{i+1}), and s (a Boolean variable that is true if the control FSM is in state s):

$$c_{op_{i+1}} = c_{op_i} \wedge c_{(op_i, op_{i+1})} \quad with \quad c_{op_1} = s$$

If an operation appears on more than one path, the conditions must be *ORed*. The condition $c_{op,s}$, under which a given operation op is executed in state s, is given by (a path is denotedp, $op \subset p$ indicates that the operation is on the path):

$$c_{op,s} = \bigvee_{p} c_{op_i} \quad [\forall subpaths\ p \mid p \in s \wedge op_i = op \wedge op \subset p]$$

An operation in multiple paths in a state may get its inputs from multiple sources. In this case a multiplexer is required to select the appropriate value. The condition under which a particular input v is selected by a multiplexer m feeding operation op, is given by:

$$sel_{v,m} = \bigvee_{s,p} c_{op_i}$$

V states s and paths p in $s \mid op_i =$ op A op$\subset p$ A v $\in$ $inputs(op,p,s)$

where $inputs(op,p,s)$ is the set of inputs of operation op if path p in state s is executed.

Registers are loaded at the end of the cycle. An assignment operation represents a register-load (denoted op → *r*) if the assigned value is alive at the first operation in the state executed after the last operation of pathp, i.e., the value is used in another state and must be stored. The condition for loading a value into a register will then be the condition defining the path containing the assignment operation. This path condition c_p is given by the AND of all conditions on the control-flow edges defining the path:

$$c_p = \bigwedge_{i=1}^{n-1} c_{(op_i, op_{i+1})}$$

where n is the number of operations in path p.

If a register *r* is loaded in multiple paths and states, its load signal will be given by the OR of the path conditions, for all paths containing assignments to the register:

$$load_r = \bigvee_{s,p} c_p \quad [\forall \textit{states } s \textit{ and paths } p \text{ in } s \mid \exists (op \rightarrow r) \wedge op \subset p\ ]$$

Control information is also generated for functional units, i.e., the condition controlling the execution of each operation c_{op_i}. This is necessary to be able to generate select operation signals for functional units when operations are later merged.

Notice that also purely combinational control within a state, for example, to enable operations not always executed (such as operations 9, 10 in figure 2), is supported by path analysis. This is particularly important in schedules with many conditional and/or chained operations. Figure 7 contains all control signals for multiplexer selection and register load in the initial data path of the example in figure 2.

6.2 Data-Path Optimizations

Optimizations in the data path comprise register, functional unit and multiplexer merging. First a global conflict graph is generated. Two hardware units are in conflict if they can not be merged. The conflict graph is then colored [19] to determine the smallest amount of hardware. We use a novel, extremely fast coloring algorithm [37].

For registers, conflicts are derived from the lifetimes. Lifetimes are computed on the control-flow graph and then projected to the states in the control FSM.

For functional units, conflicts depend on the mutual exclusiveness of the operations and on the potential area savings during merging. The conflict of operations can be computed in 'different ways:

1. ANDing their conditions. If the result is not false, the operations conflict (are not mutually exclusive).
2. Computing the transitive closure of the control flow graph in each state. Operation pairs in the transitive closure conflict.
3. Operations on the same path in a state conflict. Since all paths were already generated during scheduling, this is the method chosen. The conflicts between all operations in a path can be found in $O(n^2)$, where n is the number of operations in the path.

Additional conflict edges are added between operations whenever the area savings (resulting from the merging of the operations) are too small to justify merging. Functional units are merged by creating a new functional unit that executes all merged operations. The correct operation is selected by a control signal which is equal to the control signal of the original operation. Outputs are merged and inputs are multiplexed using the control signal of each operation to select the correct inputs.

Compatibility of multiplexers depends on the mutually exclusive selection of all their inputs (mutually exclusive data-transfers). Multiplexers are merged by connecting all inputs to the merged multiplexer and by merging the outputs. Repeated inputs can be collapsed, ORing the select signals. Mutiplexers can be further optimized by swapping commutative inputs to functional units, to increase input collapsing.

Figure 8 shows the optimized data path for the example. In this case, only two multiplexers could be merged.

7. FURTHER TASKS

For the sake of completeness, this section briefly discusses the remaining tasks in HIS.

Estimation

For both constraint generation and data path optimization, it is necessary to estimate sizes and delays. As mentioned previously, it is extremely difficult to estimate sizes and particularly delays accurately at a high level. For example, timing models often assume that chaining operations implies adding the cycle time. However, any carry-type implementation of an adder will allow chaining with very little time penalty (approximately an additional delay of one bit in the

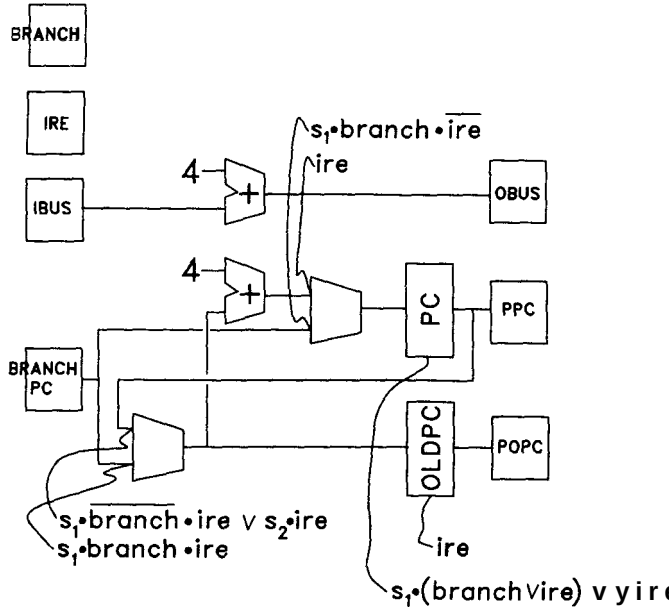

Figure 8. **Optimized data-path**

carry chain); time is by no means additive for chained operations. Even if gate level models are used, such as in logic synthesis, accurate timing prediction can not be achieved: layout and delays in the connection lines already play an important role in todays submicron technologies. We used a **CMOS** standard cell library and estimators which compute the area of functional units, multiplexers and registers by adding the area of the cells required to build them (i.e., the active area which is accurate for standard cells). For functional units implementing more than one function, the estimator computes the area taking into account the similarity between the functions (more details are given in [13]). The delay is computed on a cell by cell basis taking into account the cell loads and power levels. The models used are similar to typical logic synthesis delay models.

Logic Minimization

Logic minimization within HIS is performed to optimize control signals on the fly. Control signals are generated as two-level logic equations. The logic synthesis package Espresso [8] was integrated in HIS to perform logic minimiza-

tion. Maintaining the equations for control signals as PLAs proved to be simple.

Output generation

Output language generation is straightforward. Any language capable of representing netlists or RT-level hardware can be chosen. At present, HIS generates BDL/CS to be compatible with other IBM design tools. We are also considering generating data-flow (concurrent) VHDL.

User Interface

Finally, attention must be paid to the user interface. Unfortunately, more sophisticated graphic features usually hinder portability. So we chose to implement a text oriented, menu driven interface that allows automatic command sequence recording and use of command sequences ("scripts") to accommodate beginners and advanced users. Synthesis parameters (e.g., minimum savings for merging, number and type of functional units allowed, file names, etc.) are recorded as VHDL attributes and kept in separate files.

8. RESULTS

Most of the high-level synthesis workshop benchmarks [1] have been run through HIS. The benchmarks can be divided into four groups. The first one is the Intel 8251 UART chip [by J. Nestor, IIT] which was divided into 4 modules: Main (I8251), receiver part (RCV8251), transmitter part (TX8251), and hunt-mode receiving section (HUNT). The second group includes processors, namely the Motorola 6502 [by G.W. Leive, CMU] with its address calculation unit (ADDRC) and memory operation part (MEMOP) described separately, the FRISC (a small RISC computer as used in [5]), and a one instruction prefetch buffer (PREFETCH) [by R. Camposano, IBM]. The third group includes a Kalman Filter (KALMAN) [by D. Thomas, CMU] and a fifth order elliptical filter (ELLIPTF) [22]. Finally, there is a group of small examples taken from the literature: A differential equation solver (DIFFEQ) [36], an arithmetic circuit (MAHA) [34], a traffic light controller (MTLC, from [32], modified by the authors), the greatest common divider circuit (GCD, recursive version in [27], sequential version in [14]), and a 4 bit up and down loadable counter [18].

The examples were run on an IBM Risc System/6000 Model 520 workstation with 40Mb of memory. The results for scheduling are summarized in Table 1. No explicit constraints were given to obtain the shortest number of cycles on all paths, with the exception of ELLIPTF (FUs limited to 1 multiplier and 2 adders), MAHA (FUs limited to 1 adder and 1 subtracter) and DIFFEQ (FUs

limited to 1 adder, 1 subtracter and 2 multipliers). Results include the number of nodes in the control-flow graph, the number of paths, the number of states of the resulting schedule, the number of cycles of the longest/shortest/average path in the control FSM, and the scheduling time in CPU seconds. The number of nodes given, results from the particular way designs are represented in SSIM (this number may vary in other control/data-flow representations). The average number of cycles per path is one essential performance figure (the other being the cycle time). A more refined analysis would weight the paths according to their relative execution frequency rather than just averaging over all paths.

Design	CFnodes	Paths	States	Short/Long/Av.	CPU
I8251	100	144	15	2 / 5 / 3.3	2.0
RCV8251	84	184	23	3 / 13 / 4.4	2.0
TX8251	93	766	22	4 / 8 / 5.9	8.8
HUNT	33	14	12	4 / 7 / 5.5	0.1
M6502	398	414	106	7 / 57 / 23.2	12.5
ADDRC	59	125	14	5 / 16 / 9.1	0.9
MEMOP	20	4	6	1 / 3 / 2.3	0.1
FRISC	211	57	42	2 / 11 / 5.6	1.4
PREFETCH	18	5	4	1 / 2 / 1.5	0.1
KALMAN	101	25	24	19 / 20 / 19.5	0.3
ELLIPTF	46	1	13	13 / 13 / 13	0.1
DIFFEQ	21	2	4	1 / 4 / 2.5	0.1
MAHA	30	12	9	2 / 5 / 2.8	0.2
MTLC	14	7	8	6 / 6 / 6	0.1
GCD	18	5	2	1 / 1 / 1	0.1
COUNTER	12	3	1	1 / 1 / 1	0.1

Table 1. **AFAP Scheduling Results**

Allocation results are given in table 2. They include the data path width as given in the initial specification, the functional units used, the number of registers and number of register bits, the number and type of functional units, the number of multiplexers and multiplexer inputs, and the allocation CPU time. In the columns for registers, fu's and multiplexers, the numbers give the values after all optimizations, while the numbers in parenthesis indicate the values after initial allocation (before any optimization). The bias to enable FU merging was set, in most cases, to a savings of at least 10 cells of the CMOS standard cell library used.

Design	width	Reg./bits	FUs	Mux.(inputs)	CPU
18251	8	1/7 (1/7)	0 (0)	6/12 (6/12)	3.9
RCV8251	8	3/11 (5/19)	1- (2-)	9/22 (8/21)	5.9
TX8251	8	5/13 (5/13)	1- (1-)	12/31 (13/32)	6.2
HUNT	8	2/14 (4/18)	1- (1-)	4/11 (4/9)	0.6
M6502	8	27/122 (76/593)	6+,1-,1≥ (14+,6-,2≥)	61/305 (172/591)	195
ADDRC	8	4/40 (9/74)	3+ (9+)	12/35 (17/51)	1.2
MEMOP	8	1/8 (2/16)	1+,1- (1+,1-)	3/6 (4/8)	0.2
FRISC	16	8/96 (11/144)	2[+,-] (12+,12-)	24/76 (24/94)	6.1
PREFETCH	32	3/96 (3/96)	1+ (1+)	4/8 (4/8)	0.2
KALMAN	16	3/40 (6/64)	3+,3-,2x (5+,10-,5x)	20/58 (31/75)	2.2
ELLIPTF	16	10/160 (28/448)	2+,1x (26+,8x)	11/58 (0)	2.5
DIFFEQ	16	5/80 (10/160)	1+,1-,2x (2+,2-,6x)	10/23 (0)	0.7
MAHA	4	2/8 (8/32)	1+,1- (8+,8-)	7/23 (7/14)	0.7
MTLC	2	0 (0)	0 (0)	4/10 (4/10)	0.1
GCD	4	2/8 (2/8)	1- (2-)	9/25 (11/25)	0.3
COUNTER	4	1/4 (1/4)	1[+,-] (1+,1-)	1/2 (1/3)	0.1

FUs: + is adder, - is subtracter, x is multiplier, ≥ is greater-than-or-equal comparator, [+,-] is an ALU with addition and subtraction.

Table 2. **HIS Allocation Results**

Finally, table 3 gives the results after logic synthesis with LSS [20]. They are included to obtain more accurate figures on size and performance. The size is given in cells of a CMOS standard cell library. One cell corresponds to the area of one inverter. Registers are also included in the cell count; a one-bit register requires 10 cells. Since HIS is hierarchical, at this stage all used modules are expanded. The given sizes include all modules used (which are also listed). The number of logic levels also includes the expanded modules, except where they are followed by an asterisk.

Design	Size(cells)	Levels	Modules used
18251	458	9	
RCV8251	1301	9	PARTY, HUNT
TX8251	1076	19	**PARTY**
HUNT	476	8	
M6502	6455	24	**MEMOP**, ADDRC
ADDRC	1817	17	**MEMOP**
MEMOP	307	8	
FRISC	3485	19 *	**2xALU**16, **COM16**
PREFETCH	1771	5 *	ADD32
KALMAN	6446	51 *	**2xMUL16,** INPT, OUTP
ELLIPTF	5982	18 *	**MUL16**
DIFFEQ	6445	21 *	**2xMUL16, COM16**
MAHA	548	7	
MTLC	143	10	
GCD	385	22	
COUNTER	93	3 *	**ALU4**
PARTY	13		
INPT	35		
OUTP	35		
COM16	112		
ADD32	192		
ALU4	36		
ALU16	114		
MUL16	2148		

***Table* 3. Results after Logic Synthesis**

We do not include comparisons to other systems since it is quite difficult to ensure the same experimental procedure. Results of other systems on similar designs can be found in this volume.

9. CONCLUSIONS

This paper presented the IBM high-level synthesis system HIS. The applications envisioned for HIS are control-intensive, processor-like designs where loops and specially alternation (IF, CASE) play a major role.

Global data-flow analysis and path analysis are used for scheduling and allocation. AFAP (as-fast-as-possible) schedules are obtained by scheduling all

possible paths independently and possibly scheduling one operation in many different states depending on the path being executed. Module assignment determines the type of functional units used for each operation prior to scheduling, if constraints are specified. Allocation uses graph coloring to minimize the number of functional units, multiplexers and memory elements globally. Several steps in the synthesis method used, for example path analysis and graph coloring, are of non-polynomial time complexity in the worst case. In practice however, we have not encountered execution time problems during high-level synthesis. Logic synthesis remains the most time consuming task.

The complete system was built around an in-core data base which holds large designs in memory, resulting in a very high speed. This allows quick design-space exploration. In addition, care was taken to interface HIS to existing design tools to obtain a system that can be used in practice. The system has been used to synthesize a wide range of benchmark designs. Exploration of the use of high-level synthesis in day-to-day industrial practice is currently under way.

REFERENCES

[1] Benchmarks for the Fifth International Workshop on High-Level Synthesis, 1991. Available through electronic mail at HLSW@decwrl.dec.com

[2] A.V. Aho, R. Sethi, and J.D. Ullman, ***Compilers: Principles, Techniques and Took,*** Reading, ***MA:*** Addison-Wesley, 1986.

[3] R.A. Bergamaschi, R. Camposano, and M. Payer, "Area and Performance Optimizations in Path-Based Scheduling," ***EDAC'91***, Amsterdam, The Netherlands, February 1991.

[4] R.A. Bergamaschi, R. Camposano, and M. Payer, "Data path Synthesis Using Path Analysis," ***Proceedings 28th ACM/IEEE Design Automation Conference,*** San Francisco, CA, June 1991.

[5] V. Berstis, D. Brand, and R. Nair, "An Experiment in Silicon Compilation," ***1985 ISCAS Proceedings,*** pp. 655-658, Kyoto, June 1985.

[6] R.K. Brayton, N.L. Brenner, C.L. Chen, G. DeMicheli, C.T. McMullen, and R.H.J.M. Otten, "The YORKTOWN Silicon Compiler," ***1985 ISCAS Proceedings,*** pp. 391-394, Kyoto, June 1985.

[7] R.K. Brayton, R. Camposano, G. DeMicheli, R.H.J.M. Otten, and J.T.J. van Eijndhoven, "The Yorktown Silicon Compiler System," in D. Gajski, editor, ***Silicon Compilation,*** Addison-Wesley, 1988.

...Hachtel, C.T. McMullen, and A. Sangiovanni-Vincentelli, ***Logic Minimization Algorithms for VLSI Synthesis***, Kluwer Academic Publishers, **1984.**

[9] R. Camposano, "Structural Synthesis in the Yorktown Silicon Compiler," in C.H. Sequin, editor, ***VLSI'87, VLSI Design of Digital Systems***, pp. **61-72,** Vancouver: North-Holland, **1988.**

[10] R. Camposano, "Design Process Model in the Yorktown Silicon Compiler," ***Proc. 25th Design Automation Conference,*** Anaheim, California: ACM/IEEE, June **1988.**

[11] R. Camposano, "Path-Based Scheduling for Synthesis," ***IEEE Transactions on Computer-Aided Design,*** vol. **10,** no. **1,** pp. **85-93,** January **1991.**

[12] R. Camposano and R.A. Bergamaschi, "Synthesis using Path-Based Scheduling: Algorithms and Exercises,," ***Proceedings of*** the ***27th Design Automation Conference,*** pp. **450-455,** Orlando, Fa, June **1990.**

[13] R. Camposano and R.A. Bergamaschi, "Redesign Using State Splitting,," ***Proceedings of the First EDAC,*** pp. 157-161, Glasgow, UK, March **1990.**

[14] R. Camposano and W. Rosenstiel, "A Design Environment for the Synthesis of Integrated Circuits," ***11th Symposium on Microprocessing and Microprogramming EUROMICR0'85,*** Brussels, 1985.

[15] R. Camposano and W. Rosenstiel, "Synthesizing Circuits from Behavioral Descriptions," ***IEEE Transactions on Computer-Aided Design,*** vol. **8,** no. 2, pp. **171-180,** February **1989.**

[16] R. Camposano, L.F. Saunders, and R.M. Tabet, "High-Level Synthesis from VHDL," ***IEEE Design&Test of Computers,*** March, **1991.**

[17] R. Camposano and R.M. Tabet, "Design Representation for the Synthesis of Behavioral VHDL Models," in J.A. Darringer, F.J. Rammig, editor, ***Proceedings CHDL'89***, Elesevier Science Publishers, **1989.**

[18] R. Camposano and J.T.J van Eijndhoven, "Combined Synthesis of Control Logic and Data Path," ***Procedings of the ICCAD'87***, Santa Clara, Ca, November **1987.**

[19] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein, "Register Allocation via Coloring," ***Journal on Computer Languages,*** vol. **6,** pp. **47-57, 1981.**

[20] J.A. Darringer, D. Brand, J.V. Gerbi, W.H. Joyner Jr., and L. Trevillyan, "LSS: A System for Production Logic Synthesis," IBM *Journal of Research and Development*, vol. 28, no. 5, September 1984.

[21] S. Davidson, D. Landskov, B.D. Shriver, and P.W. Mallet, "Some Experiments in Local Microcode Compaction for Horizontal Machines," *IEEE Transactions on Computers*, vol. C-30, no. 7, pp. 460-477, July 1981.

[22] P. Dewilde, E. Deprettere, and R. Nouta, "Parallel and Pipelined VLSI Implementation of Signal Processing Algorithms," in S.Y. Kung, H.J. Whitehouse, T. Kailath, editor, *VLSI and Modem Signal Processing*, pp. 258-264, Prentice Hall, 1985.

[23] N.D. Dutt, T. Haddley, and D.D. Gajski, "An Intermediate Representation for Behavioral Synthesis," *Proceedings 27th ACM/IEEE DAC*, pp. 14-19, Orlando, FL, June 1990.

[24] L.J. Hafer and A.C. Parker, "A Formal Method for the Specification, Analysis and Design of Register-Transfer Level Digital Logic," *IEEE Transactions on CAD*, vol. 2, no. 1, pp. 4-18, January 1983.

[25] C.Y. Hitchcock III and D.E. Thomas, "A Method of Automated Data Path Synthesis," *Proceedings 20th Design Automation Conference*, pp. 484-489, June 1983.

[26] IEEE, Standard VHDL Language Reference Manual, New York: The Institute of Electrical and Electronics Engineers, Inc., March 1988.

[27] S.D. Johnson, *Synthesis of Digital Designs from Rcursion Equations*, Cambridge, Massachusetts: MIT Press, 1983.

[28] D.W. Knapp and A.C. Parker, "A Unified Representation for Design Information," *7th International Symposium on Computer Hardware Description Languages and their Applications*, pp. 337-353, Tokyo, August 1985.

[29] J. Lee, Y. Hsu, and Y. Lin, "A new Integer Linear Programming Formulation for the Scheduling Problem in Data-Path Synthesis," *ICCAD'89*, Santa Clara, Ca. November 1989.

[30] M.C. McFarland, The Value Trace: A Data Base for Automated Digital Design, Design Research Center, Carnegie-Mellon University, Report DRC-01-4-80. December 1978.

[31] M.C. McFarland, A.C. Parker, and R. Camposano, "The High-Level Synthesis of Digital Systems," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 301-318, February 1990.

[32] C. Mead and L. Conway, ***Introduction to VLSI Systems,*** Addison-Wesley, 1980.

[33] M. Monachino, "Design Verification System For Large-Scale LSI Designs," ***IBM J. Res. Develop.,*** vol. 26, no. 1, January 1982.

[34] A.C. Parker, J. Pizarro, and M. Mlinar, "MAHA: A Program for Datapath Synthesis," ***Proceedings of the 23rd Design Automation Conference,*** pp. 461-466, Las Vegas, June 1986.

[35] P.G. Paulin and J.P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's," ***IEEE Transactions on Computer-Aided Design,*** vol. 8, no. 6, pp. 661-679, June 1989.

[36] P.G. Paulin, J.P. Knight, and E.F. Girzyc, "HAL: A Multi-Paradigm Approach to Automatic Data-Path Synthesis," ***Proceedings 23rd Design Automation Conference,*** pp. 263-270, ACM/IEEE, June 1986.

[37] T.K. Philips, New Algorithms to Color Graphs and Find Maximum Cliques, Yorktown Heights: IBM Research Report, Computer Science, 1990.

[38] D. Thomas, E. Lagnese, R. Walker, J. Nestor, J. Rajan, and R. Blackburn, ***Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench,*** Boston: Kluwer Academic Publishers, 1990.

[39] C.J. Tseng and D.P. Siewiorek, "Automated Synthesis of Data Paths in Digital Systems," ***IEEE Transactions on Computer-Aided Design,*** vol. CAD-5, no. 3, pp. 379-395, July 1986.

5

MICON: Automated Design of Computer Systems

William P. Birmingham
EECS Dept
University of Michigan
Ann Arbor, Michigan 48109

Anurag P. Gupta
ECE Dept
Carnegie Mellon University
Pittsburgh, PA 15213

Daniel P. Siewiorek
ECE Dept
Carnegie Mellon University
Pittsburgh, PA 15213

1 Introduction

Semiconductor technology is providing tremendous opportunities for high processing power, reliable, and low-cost computers in the office or lab. Many of these computers, spanning the range from personal computers to super-mini's, are based on commercially available microprocessor family components. The high-end microprocessors available today commonly support virtual memory and main memory cache. Complementing these devices are a full array of high-performance dedicated processors (*e.g.* graphics controllers and numerical coprocessors) and communication components. A very sophisticated machine can be constructed almost entirely from off-the-shelf components. It appears that semiconductor designers have just begun to tap the well of possibilities, promising more performance and function in future chips.

However, there is a price for all this. Sophisticated components require sophisticated hardware designers; designers must be experts in high-speed logic design and computer architecture. The days of simple designs consisting of a processor and a serial input/output (SIO) interface are long gone. The smallest of today's computer systems contain at least disks, buses, and graphics terminal interfaces. Workstations must utilize a minimum of three levels of memory hierarchy to achieve maximal performance. In addition, such traditional extras as networking and graphics are standard equipment. All of this places a strain on hardware designers trying to keep abreast of a rapidly evolving technology.

In the marketplace where product lifecycles are measured in months, the competition is growing stronger. So, the hardware designer is being squeezed by three forces: a reduced design time, the ceaseless demand for increased