UC Irvine
ICS Technical Reports

Title
Automating technology adaptation in design synthesis

Permalink
https://escholarship.org/uc/item/1w81p3c8

Authors
Kipps, James R.
Gajski, Daniel D.

Publication Date
1989-12-05

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

ARCHIVES
Z
699
C3
no.89-43
c.2

AUTOMATING TECHNOLOGY ADAPTATION IN DESIGN SYNTHESIS

James R. Kipps **Daniel D. Gajski**

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Technical Report 89-43

5 December 1989

Copyright © 1989 University of California, Irvine

This paper was prepared as a proposal to the Knowledge Models and Cognitive Systems Program of the National Science Foundation's, Information, Robotics, and Intelligent Systems Division.

AUTOMATING TECHNOLOGY ADAPTATION IN DESIGN SYNTHESIS

James R. Kipps and Daniel D. Gajski

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

ABSTRACT

The goal of design synthesis is the generation of high-quality material designs from abstract specifications. Recent efforts in VLSI design synthesis (logic synthesis) have shown how heuristic techniques can generate human-quality hardware designs in a fraction of manual design time. Despite successes, the state of the art in logic synthesis is limited in terms of range and quality. Existing logic synthesis tools are largely restricted to designing combinational circuits described by Boolean equations. Such circuits compose only about 20 percent of a high-level design; the other 80 percent consists of complex components described in terms of their functionality, such as arithmetic and logic units, counters, and processors. Existing tools are also restricted to implementing designs with a small set of widely available physical cells, such as one- and two-level Boolean gates. While this restriction makes a synthesis tool independent of a particular component library, it also keeps the tool from using complex but nonstandard library components that might otherwise improve design quality.

To increase the capabilities and quality of synthesis tools, we propose to develop a new knowledge-intensive model of design synthesis that we call the **derivational-process model**. This model uses knowledge of **design styles** to decompose component specifications and generate designs that are appropriate to constraints. To keep this model robust to technology changes, we further propose to develop an adjunct learning component called the **model of technology adaptation**. This model uses knowledge of **fundamental principles of design** to acquire design rules that take advantage of complex library components or that reflect new design styles. The combination of these two models we refer to as **adaptive design synthesis**. This work contributes to recent work in Logic Synthesis, Knowledge-Based Design, and Machine Learning.

CONTENTS

1. Introduction 1

2. Logic Synthesis 2
   - 2.1 Methodology 2
   - 2.2 Approaches 3
   - 2.3 Limitations 4

3. Derivational-Process Model 6
   - 3.1 Designing by Function 6
   - 3.2 Synthesis as Search 8
   - 3.3 The Derivational-Process Model 9
   - 3.4 Derivation-Driven Search 11
   - 3.5 Synthesis Example 12
   - 3.6 Synthesis of Sequential Components 15
   - 3.7 The Robustness Problem 16

4. Technology Adaptation 17
   - 4.1 Knowledge Acquisition 17
   - 4.2 Fundamental Principles of Design 18
   - 4.3 Technology Fusing 19
   - 4.4 Edit Analysis 21
   - 4.5 Technology Adaptation as Learning 22

5. Conclusion 24

References 26

FIGURES

2.1 Arithmetic Component 5
3.1 Functional Design 7
3.2 Derivational-Process Model 10
3.3 ALU Decomposition Rule 13
3.4 Adder Decomposition Rule 14
3.5 Construction Rule for Adder w/Carry Enable 14
3.6 Construction Rules for Adder 15
3.7 Counter Decomposition Rule 16
3.8 Counter Construction Rule 16
4.1 Model of Technology Adaptation 18
4.2 Fundamental Principles of Design 20

1. INTRODUCTION

Synthesis is the combining of parts to form a whole. Synthesis in the VLSI domain is called logic synthesis. Parts are drawn from an ASIC (Application-Specific IC) vendor's library and combined to implement the design of high-level hardware components. The input to logic synthesis is a generic specification of the component to be designed, constraints on the design in terms of area, speed, and power, and a target library of physical components. The goal of logic synthesis is to output a high-quality physical design that implements the specified component, using parts from the vendor's library, and satisfies the design constraints. Recent efforts in logic synthesis have shown how heuristic techniques can generate human-quality hardware designs in a fraction of manual design time. Despite successes, current approaches to logic synthesis suffers from two major limitations: they are applicable to only a small portion of the components in a high-level design; and they are capable of considering only a subset of library components when implementing a design.

To improve the range and quality of logic synthesis tools beyond the state of the art, it is necessary to restructure the traditional model of logic synthesis. If logic synthesis tools are to cover all components in a high-level design, they should reflect design procedures used by human designers. Experienced design engineers use top-down refinement to **functionally decompose** regular-structured logic components into designs that can be implemented from a given component library. They use knowledge of **design styles** to select a decomposition that is appropriate to design constraints. Human designers are also robust to technology changes. They adapt readily when presented with new library components, using knowledge of **fundamental principles of design** to determine the best use of available components to implement a design. We propose to develop a knowledge-intensive model of design, called the **derivational-process model**, which accounts human proficiency and adaptability in the IC design process. The derivational-process model addresses the limitations of current approaches to logic synthesis through the use of functional decomposition, design styles, and fundamental principles of design. To demonstrate the validity of this model, we will develop a system, called **DTAS (Design and Technology Adaptation System)**, that can synthesize high-level combination and sequential logic components and still adjust to changes in the library of physical components.

This research impacts three areas of computer science: Logic Synthesis, Knowledge-Based Design, and Machine Learning. In regards to Logic Synthesis, we are taking an evolutionary "next step" from the existing state of the art. By focusing on functional decomposition, we show how to synthesize all components in a microarchitecture design, not just random logic and gate arrays; by focusing on design styles, we show how to improve design quality with global optimizations; and, by focusing on fundamental principles of design, we show how to take advantage of the target fabrication technology while retaining technology independence. In regards to Knowledge-Based Design, we are engineering a system that addresses domain-specific design issues, and, by doing so, we are providing a realistic example of the utility of knowledge-intensive systems that can be appreciated by IC designers. Design issues such as implementing designs from a parts library and maintaining robustness against library changes are of importance in domains other than VLSI, yet they have seldom

been examined in the Knowledge-Based Design literature. We are also formalizing a model of hierarchical design and optimization that can potentially be generalized to other design domains. In regards to Machine Learning, we are extrapolating analytic learning techniques to show how fundamental principles of (logic) design can be used to maintain and upgrade a knowledge base. These techniques rely on explanation-based learning (EBL) to extract new design knowledge from an analysis of ASIC libraries and design edits. Unlike other EBL research applied to VLSI, our domain theory is not limited to Boolean logic; rather, it is based on the fundamental principles of design surrounding the decomposition and implementation of combinational and sequential logic. Such a domain theory provides substantial knowledge about design semantics and allows us to address issues of granularity and generalization in a nonsyntactic manner.

In the remainder of this report, we describe our approach to adaptable design synthesis, including technical arguments substantiating our claims. Throughout we ground our discussion in terms of design synthesis in VLSI. In Section 2, we present background on logic synthesis, highlighting current approaches and limitations. In Section 3, we describe our derivational-process model of design synthesis for hardware components. In Section 4, we describe our model of technology adaptation. In Section 5, we overview our proposed research objectives. Finally, in Section 6, we outline our research schedule.

2. LOGIC SYNTHESIS

Design methods for integrated circuits have not kept pace with advances in IC fabrication technology. Manufacturers are capable of integrating highly complex circuits, printing 100,000 to one million transistors on a single chip. This level of integration has created a combinatorial explosion in the number of details required to realize low-volume, high-performance, special-purpose VLSI systems and has introduced a bottleneck in the IC product development cycle. Silicon compilation is an emerging technology intended to resolve this bottleneck. Advocates of silicon compilation take the view that expert design knowledge can be captured, proceduralized, and utilized to automate the generation of IC designs.

Logic synthesis refers to the portion of the silicon compilation process that focuses on the automatic translation of abstract hardware descriptions into physically-realizable logic designs. The goal of logic synthesis is to achieve a quality of design that is comparable to that which can be achieved by experienced design engineers. Recent efforts in logic synthesis have shown how rule-based and graph-matching techniques can be used to generate engineer-quality hardware designs at a fraction of the time required for designing manually.

2.1 Methodology

In a design methodology based on logic synthesis (de Gaus, 1989), the design engineer begins by describing the behavior of a system in some hardware description language, such as VHDL. This level of description indicates the intended functionality of the system rather than its implementation. Once the functionality has been verified, the designer reformulates the

design in terms of generic functional blocks at the microarchitecture level, such as registers, counters and control units, random combinational circuits described with Boolean equations, regular-structured arithmetic and logic units, etc. (Automated design from a hardware description language to the microarchitecture level is referred to as behavioral compilation or structural synthesis (Smith, 1988) and is being addressed by systems such as DAA (Kowalski, 1985) and VSS (Lis and Gajski, 1988).) The resulting microarchitecture design is typically represented as a register-transfer level (RTL) description that can be simulated and revised to arrive at an acceptable high-level design.

Ideally, a logic synthesis tool should start with this high-level system design and map it into an optimized configuration of physical components from an ASIC vendor's library. Existing synthesis tools are capable of synthesizing only about 20 percent of the high-level design, namely the combinational circuits described with Boolean equations. The other 80 precent of the components must be designed manually or with module generators.

The traditional model of logic synthesis consists of three phases: Boolean minimization, technology mapping, and physical optimization. In **Boolean minimization**, the equations describing the circuit are reduced to their smallest form, factoring common subterms. In **technology mapping**, the minimized Boolean equations are translated into a netlist of functionally equivalent Boolean gates selected from the vendor's library. In **physical optimization**, equivalence preserving transformations are applied to portions of the netlist until it conforms to design constraints.

2.2 Approaches

In the synthesis of combinational circuits described with Boolean equations, the process of physical optimization is crucial to achieving a high quality of design. There are two fundamental approaches to optimization, focusing on graph-matching and rule-based techniques. In the **graph-matching approach**, characterized by systems such as MIS (Brayton et al., 1986), DAGON (Keutzer, 1987), and SKOL (Bergamaschi, 1988), each logic equation is transformed into a canonical form, such as network of two-input NAND/INV (inverter) gates. The synthesis tool then examines every possible way to replace portions of this graph with one-and two-level Boolean gates from the ASIC library. The combination of replacements having the lowest gate count is selected as optimal.

In the **rule-based approach**, characterized by systems such as SOCRATES (de Gaus and Gregory, 1986), LSS (Joyner et al., 1986), and MILO (Vander Zanden and Gajski, 1988), each rule consists of a target configuration of physical gates and an alternative configuration that performs the same function more efficiently. Unlike the graph-matching approach, optimality is not judged by gate count but by how well the design meets constraints. This evaluation of optimality accounts for conflicts between area and time, e.g., the design with the smallest area is typically not the design with the shortest propagation delay. Unlike the graph-matching approach, rule-based synthesis tools do not exhaustively explore the space of physical design but use hueristic search techniques that restrict rule transformations to constraint-violating critical paths through a design.

Both graph-matching and rule-based approaches, as well as approaches that combine the two, work well for synthesizing random logic and gate arrays. They provide a high-level of design quality and ensures rapid adaptation to technology changes. The physical characteristics of Boolean gates, such as propagation delay, load factors, and cell area, can be stored in a "technology" table, which reduces technology mapping to a simple table look-up function. By only considering one- and two-level Boolean gates as well as other small components that are widely available in ASIC libraries, existing logic synthesis tools can be ported to new libraries with little effort. Further, Boolean gates that are unavailable in one library can be expressed in terms of supported gates, e.g., implementing an AND gate with a NAND and an INV. In terms of design quality, the literature reports performance measures equal to and sometimes better than those of human designers. Several logic synthesis tools have been so successful that they are now commercial products.

2.3 Limitations

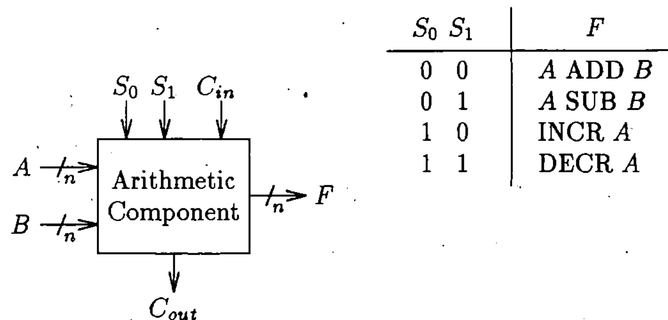
Current approaches to logic synthesis are limited in two ways: range and quality. In regards to range, they are only applicable to combinational circuits that can be described with Boolean equations, typically random logic and gate arrays, and do not scale up to complex regular-structured components described functionally. In regards to quality, they are restricted to implementing designs with only a subset of components in the ASIC library and cannot take advantage of components that might provide higher-quality designs.

Range Limitations

Although all logic components can be described with Boolean equations that relate inputs to outputs, it is not always desirable to do so and is seldom the best approach to logic design. As the number of inputs and outputs increase, Boolean descriptions become problematic to specify and manipulate. The upper bound on the number of minterms in a Boolean equation grows exponentially with each new input, until minimization of the equation becomes computationally intractable. While each output requires only one Boolean equation, the complexity of factoring a set of equations is also bounded by the number of minterms that must be considered.

To emphasize the problems this creates, consider synthesizing the arithmetic component shown in Figure 2.1. This component can perform four arithmetic operations on two n-bit inputs A and B: addition, subtraction, increment, and decrement. The n-bit output of the operations is generated at F. S is a 2-bit function select line; C_{in} is an input carry; and C_{out} is an output carry.

In the case of $n = 2$, there are 7 inputs and 3 outputs, requiring three Boolean equations with a possible 127 minterms each. When $n = 4$, there are 11 inputs and 5 outputs, requiring five equations with a possible 2048 minterms each. It is not too unrealistic to assume that existing logic synthesis tools can handle either of these cases. On the other hand, when $n = 16$, there are 35 inputs and 17 outputs, requiring 17 equations with a possible 2^{35}



S_0 S_1	F
0 0	A ADD B
0 1	A SUB B
1 0	INCR A
1 1	DECR A

Fig. 2.1 — Arithmetic Component

minterms each. While a Boolean description of this size can be computed, it cannot be minimized and factored by existing synthesis tools without the availability of considerable computing resources and without incurring considerable delay.

Many of the components in a high-level design are special-purpose combinational and sequential circuits such as the one shown above. In fact, 16 is a small value for n. Logic synthesis tools based on Boolean minimization and "peep hole" optimizations do not scale up to the complexity of these regular-structured components. Human designers can implement components much larger than this with little difficulty. While existing synthesis tools can be used for portions of the design, they require a wider range of design knowledge and capabilities to synthesize the design in its entirety.

Quality Limitations

The components in ASIC library provide optimized layouts for commonly occurring logic circuits. There are essentially two classes of library components, simple and complex. Simple components are at the SSI level; they include one-and two-level Boolean gates, one- and two-bit multiplexers and adders, etc. Complex components are at the MSI, LSI, and VLSI level; they include components as small as 4-bit adders to ALUs, controllers, and memories, upto entire processors. Typically, complex components provide better performance (e.g., smaller, faster, more powerful), than functionally equivalent configurations of simple cells. The use of complex library components during synthesis can ultimately improve design quality. Complex components are not used by existing logic synthesis tools because of their size and nonuniform library support.

Synthesis tools that take the graph-matching approach represent library components in a canonical Boolean form that is matched against a directed-acyclic graph representing the Boolean description of the component being synthesized. For simple library components, the complexity of graph-matching is relatively low, so the technique works well. As the number of inputs and outputs increases, the complexity grows by orders of magnitude, making the task computationally intractable.

Synthesis tools that take the rule-based approach can avoid the complexity of graph matching by specifying the component configuration being matched in terms of other complex component—as opposed to a configuration of Boolean gates. For example, there might be a rule for replacing a configuration of four 1-bit adders with a 4-bit adder from an ASIC

library. The problem with this approach is that complex components have more features than simple components. More features means wide variations in library support, which ultimately increases maintenance costs when the library a tool uses is upgraded or when the tool is moved to a different library. By restricting synthesis tools to simple components that are widely available in ASIC libraries, they become independent of any particular library.

As long as logic synthesis tools are applied to random combinational logic, the use of simple components does not compromise the quality of design. However, when synthesis tools are extended to regular-structured components, restricting designs to simple library components will ultimately trade off design quality for technology independence.

3. THE DERIVATIONAL-PROCESS MODEL

The state of the art in logic synthesis indicates a need to move beyond techniques that simply rely upon the manipulation of Boolean logic. In this section, we introduce the derivational-process model of design synthesis and demonstrate how its use of top-down design and knowledge of design styles allow it to synthesize a wider class of components and improve design quality.

3.1 Designing by Function

Human designers overcome the problems that limit current approaches to logic synthesis by viewing components as functions, rather than as sets of Boolean equations. By applying a top-down design strategy, they use fundamental principles of design to iteratively decompose high-level components into a hierarchy of increasingly smaller subcomponents. When there are alternative ways to decompose a component, knowledge of design styles is used to select the decomposition that best fits the design constraints. Decomposition stops when the design reaches a level of granularity that can be implemented with components from the ASIC library. When library components do not precisely fit, the same fundamental design principles used in decomposing the design can be applied from the bottom-up to augment or modify the library components until their physical characteristics match the requirements for implementation of the generic design.

For example, consider the arithmetic component discussed in Section 2. When $n = 16$, techniques based on Boolean minimization become overwhelmed by the size of the Boolean description. Figure 3.1 illustrates a top-down design method that decomposes the component based on its function. In Figure 3.1 (a), the arithmetic unit is decomposed into a 16-bit adder (FA) with an external combinational circuit (CC1) on attached to its inputs. In Figure 3.1 (b), the combination circuit is further decomposed into 16 identical circuits (CC2), one for each data input, and another combination circuit (CC3) controlling the carry input. CC2 and CC3 can be described by three simple Boolean equations and further synthesized from these. In Figure 3.1 (c), the 16-bit adder is decomposed into four 4-bit adders. In this case, a serial design style is selected to optimize for area, but, if time were the critical constraint,

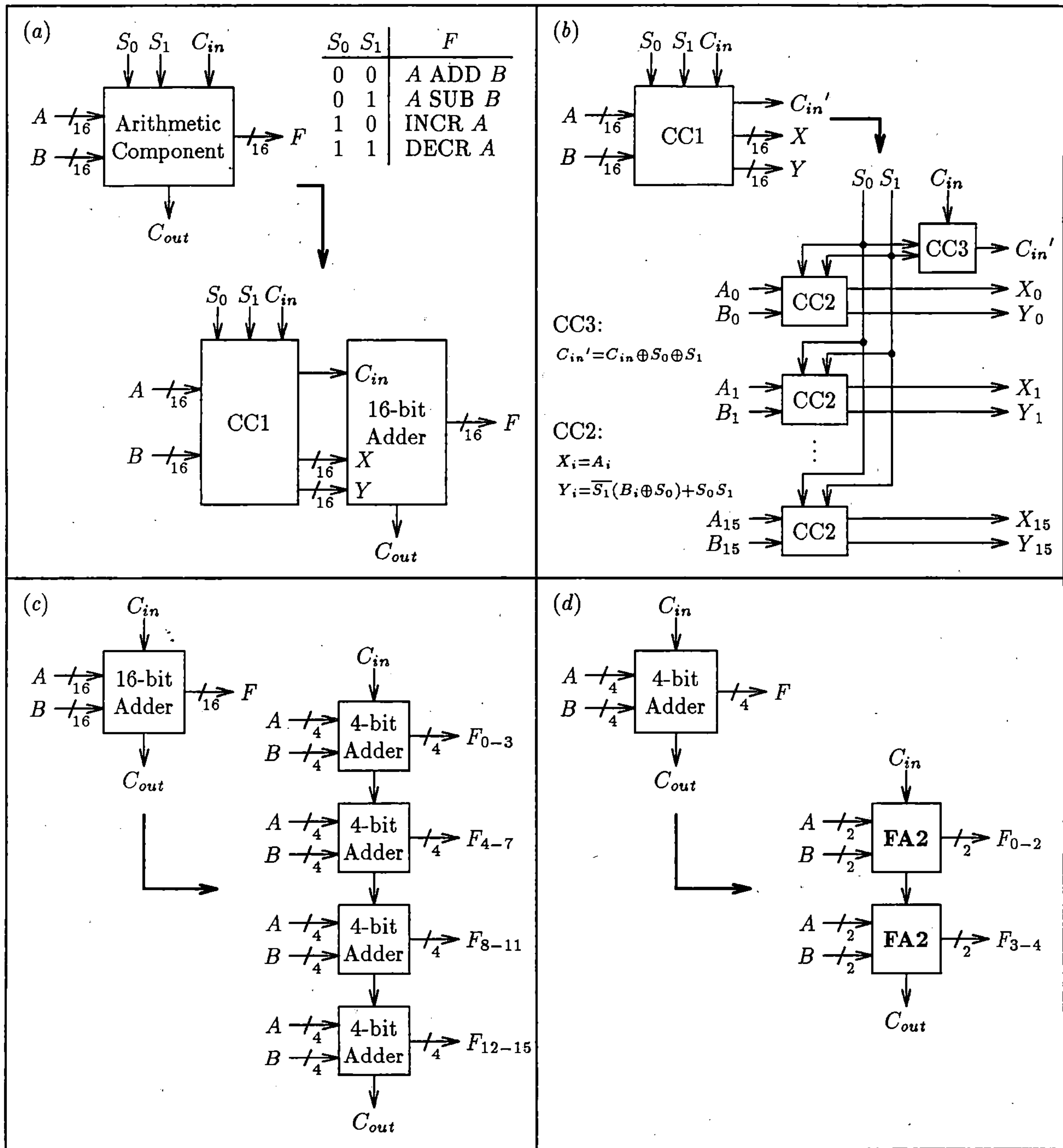


Fig. 3.1 — Functional Design

a parallel design style could be used. The 4-bit adders can be further decomposed into four 1-bit adders, each of which can be implemented with a simple configuration of Boolean gates.

If the target ASIC library contained adder components, then it is not necessary to completely decompose the generic adders to Boolean logic. For instance, if the library contained 4-bit adders, then decomposition could stop at this level, implementing the four 4-bit generic adders with four of the library adders. If, on the other hand, the library only

contains 2-bit adders (FA2), then the human designer can use fundamental principles of logic design to infer that the generic 4-bit adder can be implemented with two 2-bit library adders, as shown in Figure 3.1 (*d*).

We propose the derivational-process model of design as an approach to logic synthesis that reflects the design processes of human designers. This model extends the traditional view of logic synthesis in several unique ways, the four most significant of which include: the addition of a top-down design phase, **functional decomposition**; the use of **design styles** and **derivation-driven search** to generate a candidate set of designs that closely satisfy design constraints; a **model of technology adaptation** to ensure robustness to library changes; and the instantiation of **fundamental principles of design** to aid in acquiring both design and mapping knowledge.

3.2 Synthesis as Search

In the derivational-process model, logic synthesis is viewed as search through a two-dimensional space of designs (Kipps and Gajski, 1989). Along one dimension, designs vary by their degree of functional abstraction. Search moves from the most abstract design (e.g., the initial specification of the hardware component) to the most specific design (e.g., a physically realizable schematic). Along the other dimension, designs vary in their structural configuration. Search moves between designs at corresponding levels of abstraction (e.g., the level of physical components) looking for designs that conform to design constraints. In traversing this design space, Gajski and Brewer (1986) identify three issues that must be addressed by a synthesis tool: style selection, technology mapping, and optimization. We also add a fourth issue: minimization.

Style selection deals with the question of how to decompose generic components into functionally-equivalent configurations of generic subcomponents, i.e., search along the dimension of abstraction. There can be a variety of ways for decomposing a component, generally distinguished by design style. For instance, decomposing a 4-bit adder into four 1-bit adders using ripple carry denotes a serial design style, while decomposing into four 1-bit adders using a carry look-ahead generator denotes a parallel style. Different design styles can effect the eventual physical characteristics of a design in mutually exclusive ways. Serial styles often reduce design area but increase propagation delay, while parallel styles often reduce propagation delay but increase area.

Technology mapping deals with the related question of how to implement generic components with physical components from an ASIC library. This also constitutes search along the dimension of abstraction. Physical components can vary in their degree of fit to the generic components being implemented. Simple library components, such as Boolean gates, can have an almost one-to-one correspondence with simple generic components, possibly limited by fan-in and load factor restrictions. By comparison, complex library components, from adders to controllers to processors, can have wide degrees of variance in fit and require substantial augmentation. For example, if a library supports a 4-bit adder but the generic design requires a 5-bit adder, then the 4-bit physical adder will need to be augmented with

a 1-bit adder. A generic component can be implemented with the same library component in multiple ways, differing in choice of design style, or it can be implemented the same way with multiple library components, differing in their physical characteristics.

Optimization deals with the question of how to improve the physical characteristics of a design when it does not conform to design constraints. Unlike style selection and technology mapping, optimization performs search along the dimension of configuration. Search along this dimension can be restricted to critical paths through the design that violate constraints and by applicable optimization strategies that improve preformance. Common optimization strategies include the elimination of redundant gates, such as double inverters, the conversion of AND/OR implementations to NAND or NOR implementations, the replacement of one-level Boolean gate configurations with two-level Boolean gates, and other more complex strategies (Vander Zanden and Gajski, 1988). Strategies differ in regards to how they effect the physical characteristics of a design, some reducing area while increasing delay and vice versa. In addition, the effects of optimization strategies can be dependent on the order of their application.

Minimization deals with the question of how to reduce graphs defining abstract characteristics of component behavior. These include Boolean equations used to describe combinational circuits such as those handled by existing logic synthesis tools, state tables used to describe sequential circuits such as counters and multipliers, and protocol tables used to describe communication protocols such as those between processors and memories. Minimization is another example of search along the dimension of configuration. Although superficially similar to optimization, the states in the minimization search space are abstract descriptions of component behavior, not physical designs. Minimization provides a first step approximation to improved physical design characteristics. For instance, weak division (Brayton and McMullen, 1982) is factoring technique that can be used to minimize the number of terms in a Boolean equation (if area is a critical design constraint) or the number of levels (if delay is critical).

Existing logic synthesis tools typically focus on the issues of minimization and optimization, restricting search to the dimension of configuration. In the derivational-process model, we focus on all four issues. Search is conducted along the dimensions of both abstraction and configuration in order to support synthesis of beyond the level of simple combinational circuits.

3.3 The Derivational-Process Model

As depicted in Figure 3.2, synthesis in the derivational-process model is factored into a design phase and an optimization phase. The design phase is further factored into interacting processes of functional decomposition, design minimization, and technology mapping. Input to the model is a set of functional specifications for generic hardware components. Additional inputs include constraints on the physical characteristics of the design and an ASIC library of physical components.

Functional decomposition is a process of top-down hierarchical design. Selecting a design style appropriate to constraints, the functional decomposition process outputs a netlist of connected subcomponents. Although smaller, these subcomponents can likewise be specified at a high level and require additional decomposition. Subcomponents can also be combinational circuits with Boolean equations, in which case they are passed to the design minimization process. They can also be "closely" supported by complex components in the ASIC library, in which case they are treated as "leaf" nodes in the generic design and passed to the technology mapping process.

Technology mapping is a process that instantiates the leaf nodes of the generic design with configurations of physical components from the ASIC library. In simple cases, such as one- and two-level Boolean gates, the technology mapping process can draw components directly from the ASIC library. In cases where components are "close" in functionality to physical components in the library, the technology mapping process implements the leaf node with library components augmented by generic subcomponents. The generic subcomponents can be passed to the functional decomposition process for further refinement and design.

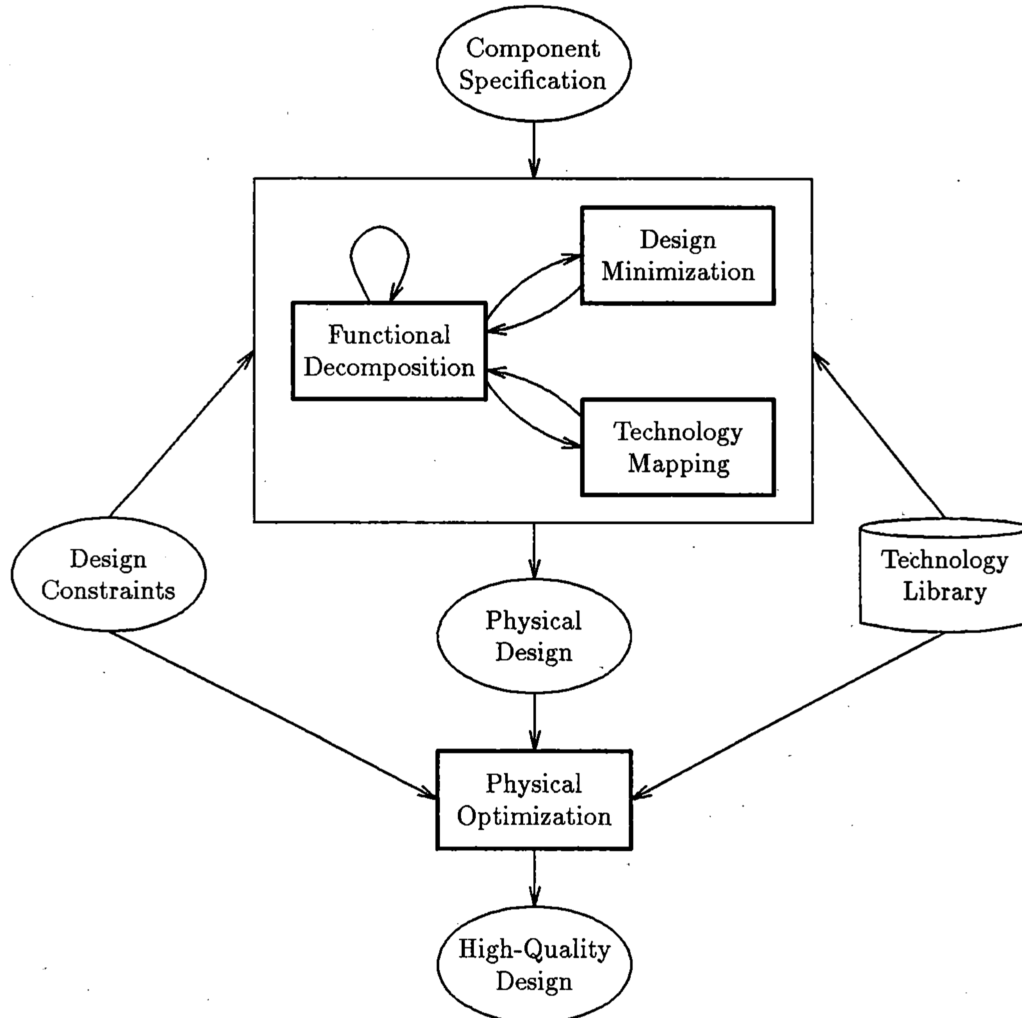


Fig. 3.2 — Derivational-Process Model of Synthesis

Design minimization is a process that subsumes a variety of graph reduction processes, such Boolean minimization, state reduction, and protocol analysis. As we are initially only synthesizing combinational microarchitecture components (components without feedback loops), design minimization can be equated to Boolean minimization, adequate techniques for which can be found in existing logic synthesis tools. When we begin synthesis of sequential microarchitecture components, we will incorporate state reduction techniques. As we scale up to processors and memories, we will also incorporate techniques for protocol minimization.

Eventually the design phase outputs a physical design, implemented with ASIC library components, which is then passed to the physical optimization phase for fine-tuning. **Physical optimization** is a process of refining the physical design along critical paths in order to meet design constraints. Initially we are not planning to expend effort in improving optimization techniques beyond the current state of the art.

Our primary focus in the development of the derivational-process model of design synthesis is on functional decomposition and technology mapping. The extended capabilities resulting from these processes is one of the aspects of this research that sets it apart from other efforts in Logic Synthesis. While we believe that design minimization and physical optimization are of equal importance to synthesis, these processes have been study extensively by the Logic Synthesis community. We recognize that scaling up to components at the microarchitecture level introduces new issues in techniques for minimization and optimization; these issues will be addressed at a later time.

3.4 Derivation-Driven Search

The processes of functional decomposition and technology mapping are controlled by **derivation-driven search**, the goal of which is to generate a set of candidate library-specific designs whose physical characteristics either meet or approximate the given design constraints. Derivation-driven search uses design rules to explore the space of generic and physical designs along the dimension of abstraction. Each design rule has a head that matches generic component specifications, a set of constraining conditions that must by satisfied by a specification for the rule to be applicable, and a body of actions that implements the specified component by generating a netlist of connected subcomponents.

There are two types of design rules, distinguished by their role in the design process, i.e., whether they are used during functional decomposition or technology mapping. Design rules for functional decomposition are called **decomposition rules**. Decomposition rules define how to achieve the functionality of a generic component in terms of connected subcomponents. Several decomposition rules that are applicable to the same generic component reflect alternative design styles. Design rules for technology mapping are called **construction rules**. Construction rules define how to implement a generic component by augmenting the functionality of a physical library component. Decomposition rules can be viewed as operating from the top-down, designing increasingly specific components, while construction rules can be viewed as operating from the bottom-up, designing increasingly abstract components.

Eventually, the most specific decomposition rules merge with the most abstract construction rules, denoting the transition from functional decomposition to technology mapping.

To illustrate the use of decomposition and construction rules in derivation-driven search, consider the example below in which we synthesize the design of an arithmetic logic unit (ALU). In this example, the library of components comes from LSI Logic, Inc. (LSI, 1987). To keep the explanation simple, the rules shown are generalized composites of the rules that would actually be necessary. Likewise, details of the cost function controlling search are ignored. The purpose of this example is merely to demonstrate how the use of derivation-directed search, functional decomposition, and technology mapping can extend synthesis to regular-structured components and improve design quality.

3.5 Synthesis Example

Assume our generic ALU can perform a set of basic arithmetic, comparison, and logic operations. A and B are n-bit data inputs, combined to generate an operation at output F. The function-select lines S distinguish the operation. Carry input C_{in} and carry output C_{out} are only useful during arithmetic operations. Output R carries the results of comparison operations.

The rules in Figure 3.3 represent two alternative design styles: a parallel design style is depicted in Figure 3.3 (a), and a serial style is depicted in Figure 3.3 (b). The central component in the parallel style is an adder (FA) with carry enable C_E. By controlling the inputs and outputs of the adder with various external combinational logic units (CC), it can be made to perform most arithmetic and comparison operations; by disabling the carry, it can be made to perform all sixteen logic operations on two variables. The serial style separates the arithmetic and comparison operations, which require an adder, from the logic operations, which only require a function generator (FG). The appropriate operations are selected by passing the outputs of these two components through a multiplexer (MUX).

There are also two styles for designing an adder, again along the lines of a parallel and serial. These are represented by the two decomposition rules shown in Figure 3.4. The first rule, Figure 3.4 (a), depicts the serial style (ripple carry) in which the carry output of each component adder is attached to the carry input of the next. The second rule, Figure 3.4 (b), depicts the parallel style (carry look-ahead) in which a carry look-ahead generator is used to compute the carry inputs to component adders. (The details of generating the combinational logic needed for the CC's as well as the logic used for the function generator FG, multiplexer MUX, and carry look-ahead generator CLA are not important to this example.)

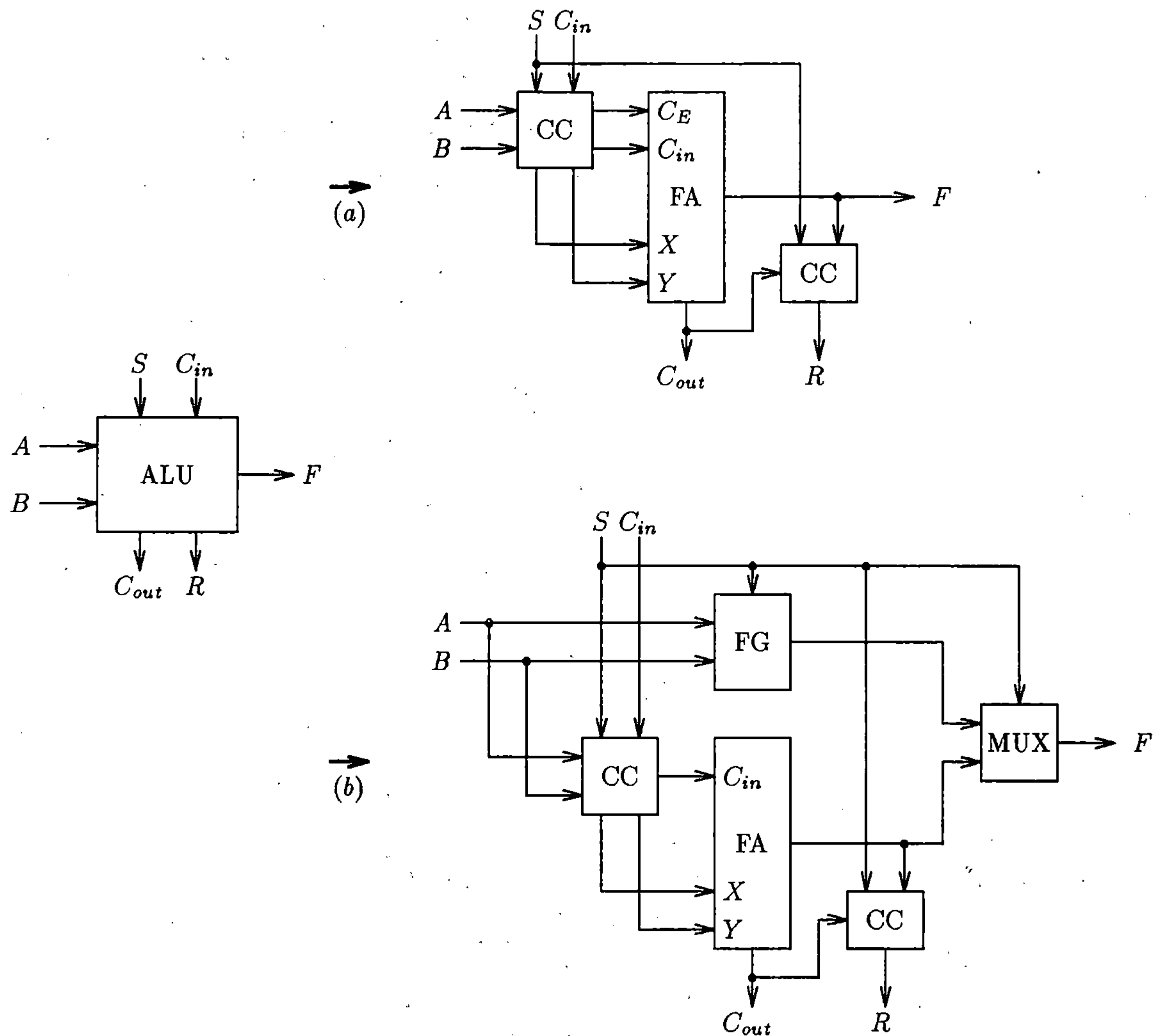


Fig. 3.3 — ALU Decomposition Rules

The advantage of using a parallel style in designing an ALU is that it integrates the combinational logic needed for the arithmetic and logical operations, eliminating redundant logic as well as a level of delay required for multiplexing. The advantage of a parallel style in designing an adder is that it reduces propagation delay, although it increases the size of the circuit. An adder can be decomposed into several levels of adders, so its design can actually use a combined parallel/serial style.

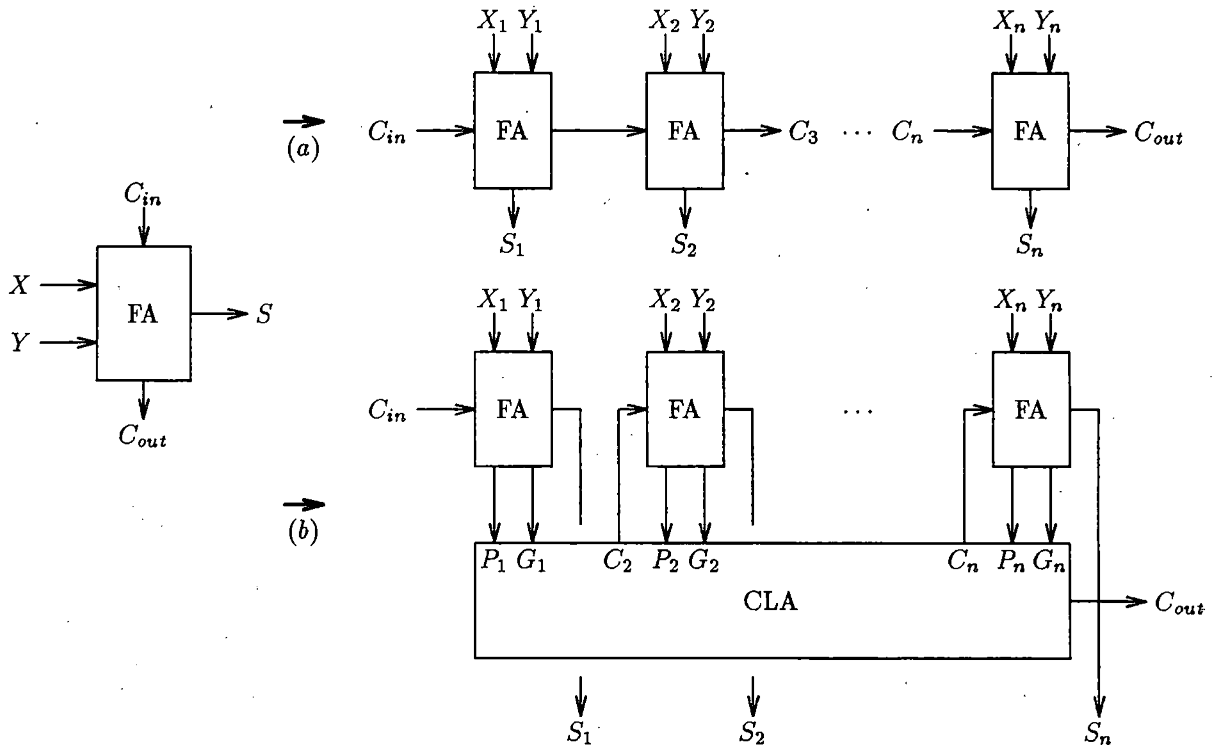


Fig. 3.4 — Adder Decomposition Rules

Construction rules come into play by providing the synthesis tool with knowledge of available library components and methods for their use. Among the components in the LSI Logic library are four different adders: FA16 (16-bit parallel adder), FA4 (4-bit adder), FA2 (2-bit adder), and FA1 (1-bit adder). Given the earlier decomposition rules, there are two types of generic n-bit adders needed in designing an ALU: one with a carry enable, and one without. The former, because none of the library adders come with a carry enable, can only be implemented from n FA1s augmented by an AND gate on its carry input, as shown in Figure 3.5. The latter can be constructed from any of the existing adders, depending on the size of n, as shown by the rules in Figure 3.6.

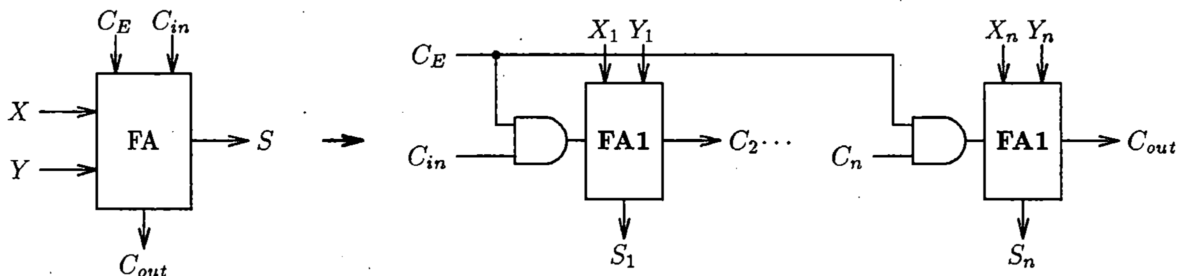


Fig. 3.5 — Construction Rule for Adder w/Carry Enable

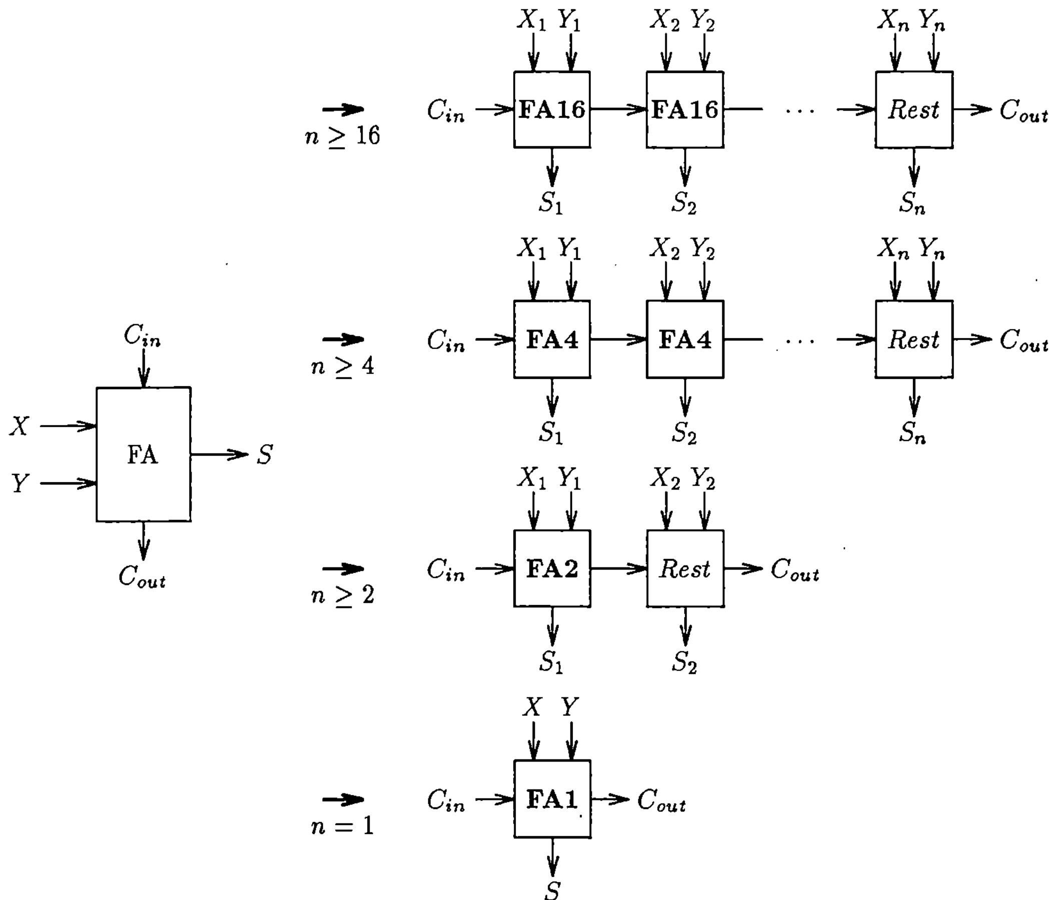


Fig. 3.6 — Construction Rules for Adders

Now assume that we wish to synthesize a 32-bit ALU. Having no knowledge of the complex library components, a design synthesis tool would probably use a parallel style to design the ALU, decomposing the design to the level of Boolean gates; unless time or area were extremely critical, a mixed parallel/serial style would be selected in designing the adder. With the construction rules, however, it is possible to accurately measure the trade off between the parallel and serial design styles in regards to the complex components available in the ASIC library. With the parallel style, the most complex library component that can be used is the FA1, while with the serial style it is possible to use any of the stock adders. Given that two FA16's sufficiently outperform 32 FA1's, then the design synthesis tool would prefer the serial design style for ALUs to the parallel and output a higher quality design than otherwise possible.

3.6 Synthesis of Sequential Components

Traditional approaches to logic synthesis are unable to synthesize sequential components, i.e., components that contain memory and a feedback loop, such as flip-flops and registers, counters, and accumulators. Although we have thus far stressed combinational logic in

our discussion and examples, the derivational-process model is also capable of synthesizing sequential logic.

Consider a rough example of how a counter can be synthesized. Figure 3.7 illustrates a design rule for an n-bit synchronous binary counter with three functions: load, count up, and count down. This rule decomposes the counter into n JK flip-flops with J and K inputs controlled by a combinational circuit (CC). The CC takes as inputs the current state of the flip-flops, the function-select line S, and an n-bit value to be loaded. The design of the CC can be generated from a set of state equations extracted from the state table for the counter and the excitation tables for the JK flip-flops.

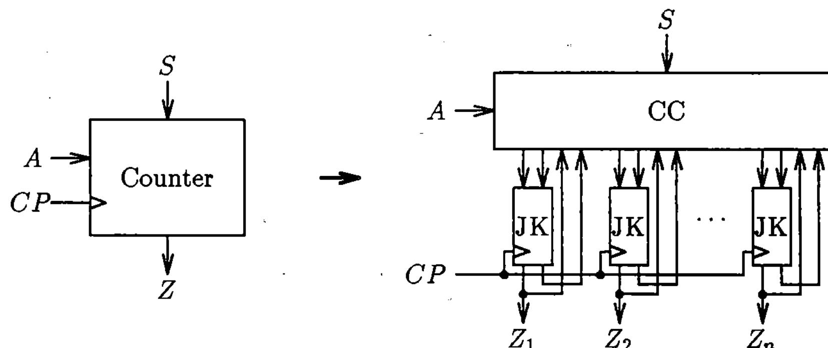


Fig. 3.7 — Counter Decomposition Rule

Figure 3.8 depicts a construction rule for implementing a decimal counter (BDC) from a stock 4-bit binary ripple counter, such as the CM16BR counter supported by LSI Logic, Inc. This construction rule places an external combinational circuit (CC) on the outputs of the library counter. The single output of CC is connected to the clear direct line CD of the counter and is enabled only when the output of the counter is binary ten, resetting the counter to zero.

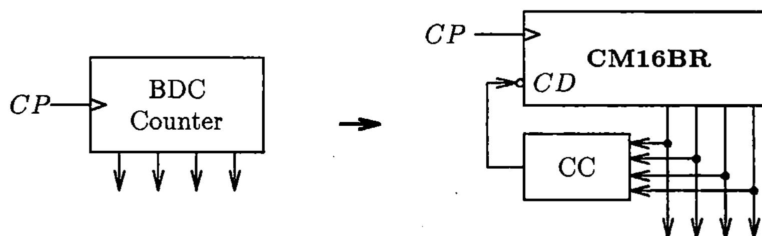


Fig. 3.8 — Counter Construction Rule

3.7 The Robustness Problem

The effectiveness of the derivational-process model of design synthesis can be measured against existing logic synthesis tools in terms of competence, quality, and robustness. Because our model scales up to regular-structured microarchitecture components, we will have improved competence. Because our model considers alternative design styles and complex library components, we will have improved design quality. However, because our model is

knowledge-intensive, utilizing library-specific components, we fall short in regards to robustness to library changes or to changes in the fabrication technology.

Construction rules that operate on complex library components (from adders and decoders up to entire processors) will have little or no transfer between fabrication technologies or even between libraries in the same technology. Decomposition rules, which work on generic components, are likely to have better transfer, but they may still embody design styles that are only appropriate for one technology or application area and ignore alternative styles that are appropriate for others. Generating new design rules, either when moving to a new technology or simply adapting a rule base to advances in the same technology, can be time consuming and error prone. It can also require a certain amount of expertise with the rule language and rule interactions. The level of effort required to maintain and upgrade a synthesis tool based on this technology-specific approach could be prohibitively expensive. We are attempting to overcoming this problem by automating technology adaptation.

4. TECHNOLOGY ADAPTATION

We propose to overcome the robustness problem with a model of technology adaptation. This model is intended to aid in the generation of design rules given a set of fundamental principles of design, knowledge of the ASIC vendor's library, and samples of user edits to synthesized designs.

4.1 Knowledge Acquisition

Addressing the robustness problem in design synthesis tools, i.e., the problem of encoding design rules to meet library and technology changes, is one of primary objectives of our proposed research. The robustness problem is essentially an instance of the knowledge acquisition bottleneck encountered in developing expert systems. The Machine Learning and Artificial Intelligence literature report many efforts to reduce cost and increase preformance of knowledge-based systems with semi-automated tools for aiding in the knowledge acquisition process. Examples include STRIPS (Fikes et al., 1972), which learned macro-operators for planning the blocks world domain, TEIRESIAS (Davis, 1982), which interactively repaired and extended the knowledge base of the MYCIN (Shortliffe, 1976) medical diagnostic system, LEX (Mitchell et al., 1983), which learned search control heuristics for problem solving in integral calculus, and LEAP (Mitchell et al., 1985), which learned new design rules for the VEXED (Mitchell et al., 1984) IC design system.

Researchers have addressed the knowledge acquisition bottleneck in other design systems with techniques for automating the acquisition of problem-solving knowledge. A similar approach can be used for achieving technology independence in design synthesis. Robustness can be restored by automating the process of technology adaptation with the application of techniques similar to those found in the Machine Learning literature. In this way, the role of learning in design synthesis can be viewed as one of knowledge maintenance (Kipps and Gajski, 1989).

Our model of technology adaptation and its relationship to design synthesis is illustrated in Figure 4.1. This model consists of two learning components: **technology fusing** and **edit analysis**.

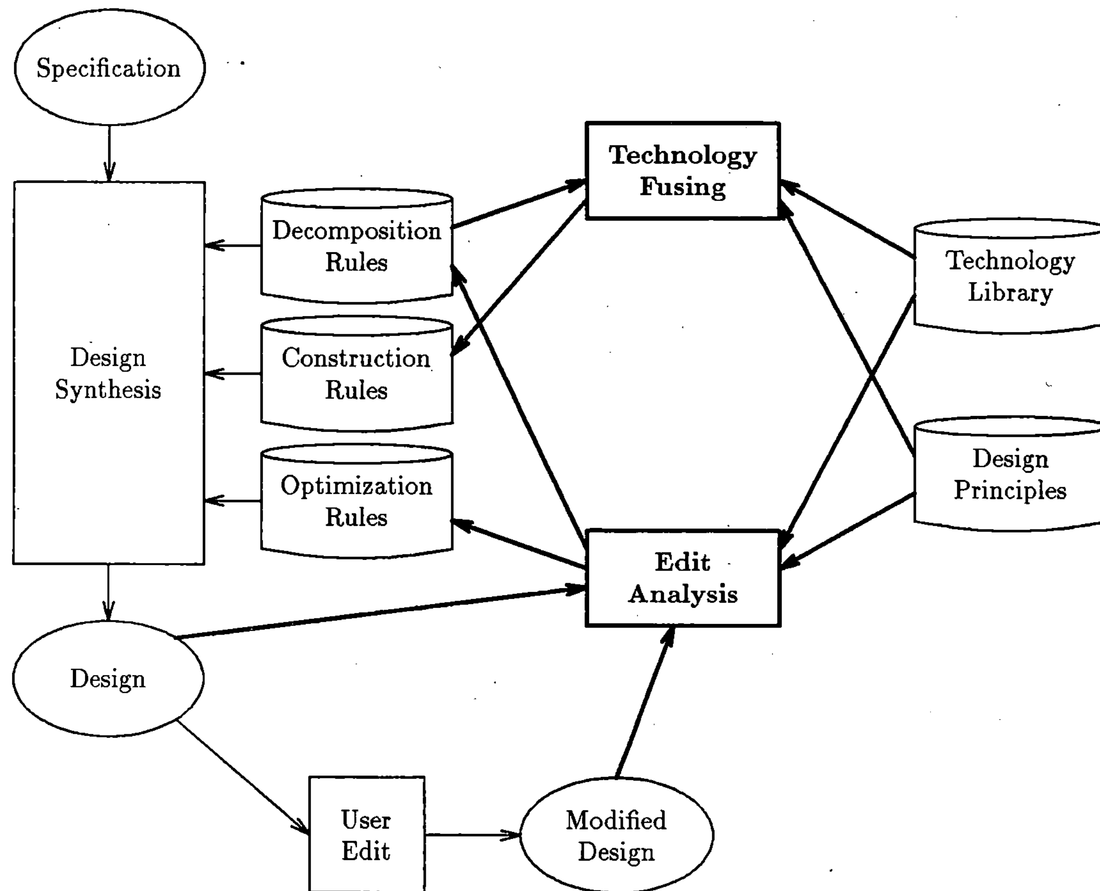


Fig. 4.1 — Model of Technology Adaptation

The purpose of technology fusing is to generate construction rules given knowledge of fundamental principles of design, a set of target generic components, and a library of physical components. This component operates as a preprocess to synthesis, generating design rules without having to run the synthesis tool. The purpose of edit analysis is to acquire new decomposition rules reflecting design styles appropriate to a library or application area. Edit analysis acquires new decomposition rules by examining changes made to synthesized designs by the user and explaining those changes in terms of fundamental principles of design.

4.2 Fundamental Principles of Design

As outlined above, both technology fusing and edit analysis depend upon knowledge of fundamental principles of design. When presented with new library components or a new application area, human designers adapt quickly. They do so by bringing to bear bits of knowledge and techniques for logic design that are essentially technology and application independent. This knowledge is what we refer to as the **fundamental principles of de-**

sign. The fundamental principles of design govern how generic designs can be organized, decomposed, implemented.

One aspect of our proposed research is to identify these fundamental principles of design and demonstrate their use. Those that we have identified so far include:

(1) **exclusion**, ignoring inputs and outputs;

(2) **externalization**, augmenting inputs and outputs with combinational circuits;

(3) **cascading**, combining (or splicing) components of the same type in sequence;

(4) **factoring**, combining like components in a tree or graph arrangement;

(5) **multiplexing**, combining components of different type through a multiplexer or bus.

Use of these principles is demonstrated by example in Figure 4.2. For this example, assume that the physical component is a simple 4-bit ALU (ALU4) with four operations: addition, subtraction, nand, and nor.

To generalize the data width of the ALU4 so as to implement a generic n-bit ALU with the same four operations, we can apply the first and third principles. When $n < 4$, as in Figure 4.2 (a), the ALU4 can be used to implement the n-bit ALU by setting the $4 - n$ least significant input pins to low and grounding the $4 - n$ least significant output pins. When $n > 4$, as in Figure 4.2 (b), the ALU4 can be used to implement the n-bit ALU by cascading $\lfloor \frac{n}{4} \rfloor$ ALU4's, rippling carries and attaching a generic $\mathrm{mod}(n, 4)$-bit ALU at the tail. To generalize the functionality of the ALU4, we can apply the second and fifth principles, augmenting I/O when the change in functionality is slight and multiplexing when a more complicated change is required. In Figure 4.2 (c), the ALU4 is generalized to implement an ALU with relational operators (R) by augmenting the select line, data output, and carry output with an external combinational circuit (CC). In Figure 4.2 (d), the ALU4 is generalized to implement an ALU with all 16 logic operations. Since this marks a substantial change from the functionality of the ALU4 it is not done by augmentation but by coupling the ALU4 to a function generator (FG) and passing the outputs through a multiplexer (MUX). Although not shown here, examples of the fourth principle, factoring, can be seen in variablizing the data width of components such as decoders and multipliers.

4.3 Technology Fusing

The objective of technology fusing is the generation of technology mapping knowledge, in the form of construction rules, subject to a set of decomposition rules and ASIC library components. Inputs to the technology fusing process include:

- a set of target generic components;
- a library of physical components;
- knowledge of fundamental principles of design.

Output are construction rules for implementing generic components with library components. This process is performed in advance of synthesis to allow immediate use the library.

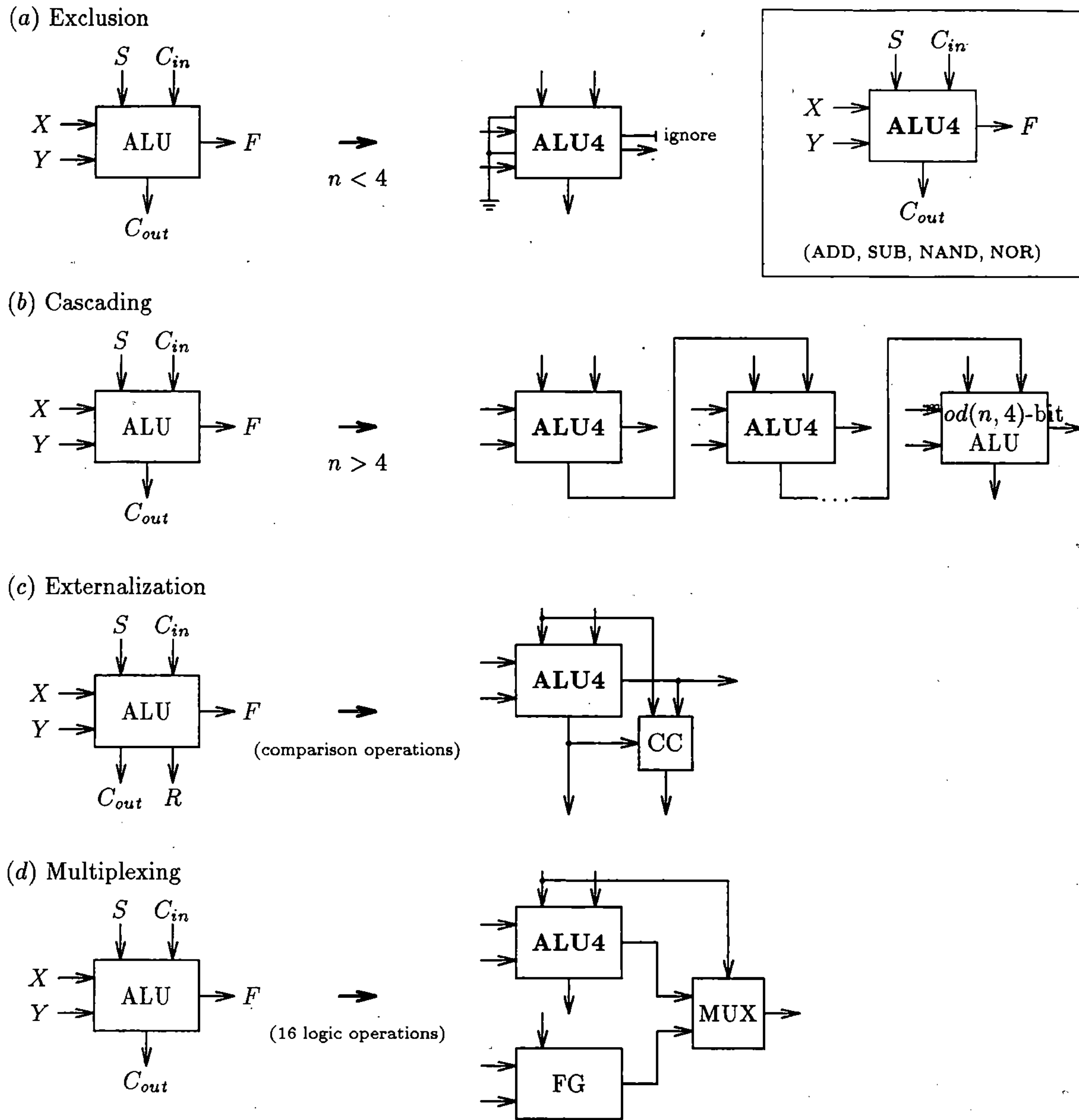


Fig. 4.2 — Fundamental Principles of Design

Technology fusing operates by generating rules that generalize the absolute characteristics of physical components. Characteristics of a component correspond to such things as the type and width of its inputs and outputs, and the functions the component performs. For physical components, characteristics are said to be absolute because their values are fixed and unchangeable. For generic components, characteristics can be absolute or variable. Variable characteristics are specified at design time. For instance, a decomposition rule may mention a generic n-bit adder, where n is set to 16 during functional decomposition. Physical components are drawn from a given ASIC library, while generic components are drawn from decomposition rules.

From derivations of the fundamental principles of design outlined above, technology fusing can generate construction rules using a form of means-ends analysis. As defined by Newell and Simon (1963), means-ends analysis is a problem-solving algorithm that finds solutions by reducing differences between an initial state and a goal state, using a set of operators for reducing individual differences.

For our purpose, the absolute characteristics of a physical component correspond to the initial state while the variable characteristics of a generic component correspond to the goal. Absolute characteristics that have not been generalized correspond to differences, and the fundamental principles of design correspond to operators for reducing those differences. For each physical component in the target technology library, technology fusing selects a generic component (extracted from the decomposition rules) that is "close" in functionality to the physical component. Technology fusing then iteratively generates a sequence of construction rules that implements the generic component using the physical component. On each iteration, technology fusing selects a difference between the two components and a design principle that could reduce that difference and generates a construction rule based on that principle.

Each generated construction rule implements a component that is a little closer in functionality to the generic component. Eventually, a construction rule is generated that exactly implements the generic component and iteration stops. The technology fusing process then continues to the next physical component.

4.4 Edit Analysis

The objective of edit analysis is the acquisition of design knowledge, in the form of decomposition rules, subject to user modifications to synthesized designs. Input to the edit analysis process includes:

- the top-level component specification;
- a list of changes made to the synthesized design;
- a library of physical components;
- knowledge of fundamental principles of design.

The output of the process is a set of decomposition rules that generalize an explanation of how the modified design implements the original component specification. Edit analysis is intended to be preformed as a postprocess to synthesis to allow users to fill in gaps in the knowledge base as they become apparent.

Edit analysis operates by capturing new design knowledge when deficiencies are discovered in the knowledge base of the synthesis tool. There are various reasons for these appearing in the knowledge base at all. The knowledge base of the synthesis tool may have been tailored for one application area and inappropriate for others. Another possibility is the need for new design styles after the encoding of the knowledge base. Likewise, the synthesis tool could be introduced to a ASIC library containing physical components for which

no generic component is a "close" fit, i.e., a situation requiring new decomposition rules for technology fusing to have been useful.

Deficiencies are said to be "detected" when the user finds it necessary to modify synthesized designs. Such modifications would trigger edit analysis in the hopes of filling in the missing design knowledge. Working interactively with the user, edit analysis would attempt to reconstruct the steps taken in formulating the modified design as instantiations of the fundamental principles of design described earlier. These steps would then be generalized and added to the design knowledge of the synthesis tool.

Working interactively with the user, edit analysis would reconstruct the steps in the decomposition of the design that explains how the user reached the new design in terms of the fundamental principles of design. The decomposition would form a proof that can be generalized to provide new decomposition rules. These new decomposition rules would be able to reproduce the steps taken by the user when designing similar components. As an example, consider the two decomposition rules for an ALU from Figure 3.2. Assume that our logic synthesis tool only has a design rule that uses a parallel style, as shown in Figure 3.2 (*a*). If a user modified the synthesized design of an ALU to reflect the serial design style, edit analysis could reconstruct the decomposition of the modified ALU design using principles such as cascading and multiplexing, and then generalize this decomposition to generate the rule in Figure 3.2 (*b*).

4.5 Technology Adaptation as Learning

Recent efforts in Knowledge-Based Design have been investigating the role learning plays in the human design process (Mostow, 1985). CGEN (Birmingham and Siewiorek, 1988) is the learning component of a knowledge-based design tool that queries the user for missing design steps. CGEN fills in the gaps in the knowledge base with rules that generalize the user's response. Another approach uses analytic learning techniques, such as explanation-based generalization (EBG) (Mitchell et al., 1986). In the LEAP system, Mitchel et al. (1985) proposed a "learning apprentice" system for acquiring design rules by observing the actions of an expert designer. LEAP records the actions of the designer, generates proofs (or explanations) that those actions satisfy the design constraints, and then generalizes the proofs to create the conditions for new design rules. This idea has been further extended to acquiring generalized design plans as macro-operators in the ARGO system (Huhns and Acosta, 1988).

When viewed from a Machine Learning perspective, the processes of technology fusing and edit analysis can be related to analytic learning techniques. The process of technology fusing is similar to that of learning macro-operators, while the process of edit analysis is similar to the learning apprentice approach taken in LEAP. Our work differs from other efforts in logic design in that we do not limit our "domain theory" to Boolean logic; rather, our domain theory incorporates the fundamental principles of (logic) design.

Technology Fusing as Learning Macro-Operators

A macro-operator is a sequence of primitive operators from a problem domain that solve a goal. Compiling primitive operators into macro-operators is a method for avoiding otherwise intractable search problems, such as nonserializable subgoals (Korf, 1985), or simply for reducing search. Macro-operators have been applied to domains such as robot planning (Fikes et al., 1972), puzzle solving with weak methods (Korf, 1985), and natural language text understanding (Mooney and DeJong, 1985).

In our models of synthesis, design rules can be viewed as macro-operators that combine a sequence of three design operations to synthesize one level of component decomposition. The first operation computes a distinct set of specifications for the subcomponents required by the synthesized design. The second operation generates (possibly replicating) component instances from the specifications. The third operation connects component instances into a netlist design. After a design rule is executed, the generated subcomponent specifications are added to a queue, and their designs are synthesized at a later iteration through the design process.

Given a generic and library component for which there is a "close" fit, the task of technology fusing is to generate a sequence of construction rules that implements the generic component with the library component. This process is similar to learning a sequence of macro-operators, where each construction rule is a macro-operator whose goal is to reduce one more difference between the generic component and the library component. Within this framework, the fundamental principles of design act as a theory of the domain that guides learning.

Edit Analysis as EBG

In recent years, explanation-based generalization (EBG) has been explored extensively as an alternative to data-intensive empirical learning methods (DeJong, 1983) (Kedar-Cabelli, 1985) (Mahadevan, 1985) (Mitchell et al., 1986) (DeJong and Mooney, 1986) (Pazzani, 1987) (Minton, 1988). For tasks in concept formation, EBG methods generalize concept descriptions from a single training instance by first constructing an explanation of how the training instance satisfies the definition of the concept under study. The features of the training instance required by the explanation are then used as the basis for formulating a general concept description. The justification for this description follows from the explanation constructed for the training instance. EBG is used in LEAP (Mitchell et al., 1985) as part of a learning apprentice system for acquiring design rules by observing the actions of an expert designer. LEAP records the actions of the designer, generates proofs that those actions satisfy the design constraints, and then generalizes the proofs to create the conditions for new design rules.

In our model of synthesis, the user is decoupled from the design process and, therefore, cannot lead the synthesis tool through the correct sequence of steps in decomposing the design. EBG techniques can still be applied during edit analysis to acquire new decomposition rules. By treating the fundamental principles of design as a theory of the domain, edit

analysis has a set of guide lines for reconstructing the steps taken by the user in developing the modified design. This reconstruction also acts as a proof of the new design's soundness. Each step in the reconstruction of the design can be related to a decomposition rule. If a rule for the step is not already present in the knowledge base, then the step can be generalized to give a new decomposition rule that can be used for similar situations occurring in the future.

A significant difference between our work and LEAP comes from the choice of domain theories. In LEAP, the domain theory for VLSI design consisted of postulates in Boolean logic, such as De Morgan's Law. Such a domain theory is adequate for explaining the structure of random logic. However, as with approaches to logic synthesis based on Boolean minimization, this domain theory does not scale up. Boolean logic does not explain the design of regular-structured logic components. On the other hand, our domain theory is based on fundamental principles of logic design, which do explain functional decomposition.

The fundamental principles of design also eliminate two problems sited by Mitchell: determining the grain size of the acquired rules, and selecting the portions of the proof to generalize. First, the proper grain size is exactly that portion of the proof explained by the design principle. This is simply an artifact of the design principles; they were develop to instruct designers on how to take the next step in the design process. Second, as we discussed with technology fusing, each design principles can be used to generalize some aspect of a physical component. This property can also be used to select the appropriate portions of the proof step to generalize.

5. CONCLUSION

The research proposed here improves the state of the art in design synthesis. It extends the focus of logic synthesis from combinational circuits described with Boolean equations to regular-structured components that are beyond the capabilities of existing logic synthesis tools. In particular, we propose the following efforts:

- **We propose to define techniques for synthesizing complex hardware components from their functional description using the derivational-process of design synthesis.**
- **We propose to define techniques for automating technology adaptation in design synthesis through the processes of technology fusing and edit analysis and the use of fundamental principles of design.**
- **We propose to exploit the processes of technology fusing and edit analysis for the acquisition of mapping and design knowledge.**
- **We propose to validate our model of adaptive design synthesis with a prototype tool for designing LSI-level components and measuring the level of effort required to maintain this tool given technology changes.**

While our current focus is on adaptive design synthesis in the VLSI domain, we hope to eventually generalize this model and apply it to other domains of design.

Our proposed research can be viewed as integrating current research in Artificial Intelligence and Computer-Aided Design. In AI, several research efforts are examining the role of learning in design. While examples are often drawn from VLSI, research objectives are on identifying general learning techniques, not solving problems in the design domain. Our research concentrates on solving specific problems in IC design, such as learning how to use a vendor's parts library. In CAD, researchers focus on automating aspects of the design process. While the CAD community recognizes the potential of techniques from AI, the transfer of technology has largely been from the area of Expert Systems and not Machine Learning. In our research, we are advancing a comprehensive model of synthesis that improves design quality and builds on techniques from Machine Learning to counter technology dependency.

By coupling our model of design synthesis with a technology adaptation component as outlined in this report, we will show how to produce a synthesis tool that can take advantage to technology-specific knowledge and still adapt readily to technology changes. The resulting tool will be capable of generating designs of comparable quality to those generated by human designers and designs of superior quality to those generated by existing synthesis tools. The technology adaptation component will automate the processes of tool maintenance and upgrade.

To validate our approach, we are developing an adaptable design synthesis tool. This tool is built around two separate inference engines. One is defined as a backward-chaining rule interpreter and is used for derivation-driven search during functional decomposition and technology mapping. The other is defined as a forward-chaining interpreter that performs state-driven search during optimization. The technology adaptation components center on the use of functional derivations of the fundamental principles of design discussed earlier. These derivations describe how design principles can be used for particular classes of components, such as decoders, encoders, shifters, counters, adders, etc. For instance, the cascade principle has a different derivation for logic units, which can be cascaded independently, than it has for arithmetic and comparison units, which must be connected by carries.

The research and development effort we propose can be broken down into the three phases listed below. In phase I, the synthesis tool will be oriented towards synthesizing combinational data path logic using MSI-level components from the LSI Logic Database and then transferring to LSI-level components from the Texas Instruments Database. For example, synthesizing a 32-bit ALU with FA16 fast adders from LSI Logic or with SN74181 4-bit ALUs from Texas Instruments. In phase II, we will introduce sequential-storage circuits, while in phase III, we will extend our work to the synthesis of microcomputers.

Validation will be done by comparison with other synthesis tools, as well as human designers. We will measure the effectiveness of our system in terms of design quality and robustness to technology changes: both changes to a known technology, and transfer to a new technology.

REFERENCES

Birmingham, W.P., D.P. Siewiorek, *Automated Knowledge Acquisition for a Computer Hardware Synthesis System*, EDRC 18-06-88, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, 1988.

Bergamaschi, R.A., "Automatic Synthesis and Technology Mapping of Combinational Logic," *Proceedings of the IEEE International Conference on Computer-Aided Design (ICCAD-88)*, pp. 466-469, Nov. 1988.

Brayton, R.K., C.T. McMullen, "The Decomposition and Factorization of Boolean Expressions," *Proceedings of the International Symposium on Circuits and Systems*, pp. 49-54, 1982.

Brayton, R.K., E. Detjens, S. Krishna, T. Ma, et al., "Multiple-Level Logic Optimization System," *Proceedings of the International Conference on Computer-Aided Design (ICCAD-86)*, pp. 356-359, Nov. 1986.

Darringer, J.A., W.H. Joyner, "A New Look at Logic Synthesis," *Proceedings of the 17th Design Automation Conference*, pp. 543-549, June 1980.

Davis, R., "Applications of Meta Level Knowledge to the Construction, Maintenance and Use of Large Knowledge Bases," *Knowledge-Based Systems in Artificial Intelligence*, R. Davis and D.B. Lenat (eds.), McGraw-Hill, New York, 1982.

de Geus, A.J., D.J. Gregory, "The Socrates Logic Synthesis and Optimization System," *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation*, G. De Micheli, A. Sangiovanni-Vincentelli, P. Antognetti (eds.), Martinus Nijhoff Publishers, Boston, MA, 1986.

de Geus, A.J., "Logic Synthesis Speeds ASIC Design," *IEEE Spectrum*. vol. 26, no. 8, pp. 27-31, Aug. 1989.

DeJong, G., "Acquiring Schemata through Understanding and Generalizing Plans," *Proceedings of the Eighth International Joint Conference on Artificial Intelligence (IJCAI-83)*, pp. 462-464, Aug. 1983.

DeJong, G., R. Mooney, "Explanation-Based Learning: An Alternative View," *Machine Learning*, vol. 1, no. 2, pp. 145-176, 1986.

Detjens, E., G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, A. Wang, "Technology Mapping in MIS," *Proceedings of the International Conference on Computer-Aided Design (ICCAD-87)*, pp. 116-119, Nov. 1987.

Fikes, R.E., P.E. Hart, N.J. Nilsson, "Learning and Executing Generalized Robot Plans," *Artificial Intelligence*, vol. 3, no. 4, pp. 251-288, 1972.

Gajski, D.D., F.D. Brewer, "Towards Intelligent Silicon Compilation," *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation*, G. De Micheli, A. Sangiovanni-Vincentelli, P. Antognetti (eds.), Martinus Nijhoff Publishers, Boston, MA, 1986.

Huhns, M.N., R.D. Acosta, "Argo: A System for Design by Analogy," *IEEE Expert*, pp. 53-68, Fall 1988.

Joyner, W.H., Jr., L.H. Trevillyan, D. Brand, T.A. Nix, S.C. Gundersen, "Technology Adaptation in Logic Synthesis," *Proceedings of the 23rd Design Automation Conference*, pp. 94-100, June 1986.

Kedar-Cabelli, S.T., "Purpose-Directed Analogy," *Proceedings of the Cognitive Science Society Conference*, 1985.

Keutzer, K., "DAGON: Technology Binding and Local Optimization by DAG Matching," *Proceedings of the 24th Design Automation Conference*, pp. 341-347, 1987.

Kipps, J.R., D.D. Gajski, "The Role of Learning in Logic Synthesis," *Proceedings of the IEEE Workshop on Tools for AI*, Oct. 1989.

Korf, R.E., *Learning to Solve Problems by Searching for Macro-Operators*, Pitman Advanced Publishing Program, Boston, MA, 1985.

Kowalski, T.J., *An Artificial Intelligence Approach to VLSI Design*, Kluwer Academic Publishers, Boston, MA, 1985.

Lis, J.S., D. Gajski, "Synthesis from VHDL," *Proceedings of the International Conference on Computer Design (ICCD-88)*, Nov. 1988.

LSI Logic, Inc., *CMOS Macrocell Manual*, Milipitas, CA, 1987.

Mahadevan, S., "Verification-Based Learning: A Generalization Strategy for Inferring Problem-Decomposition Methods," *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, Aug. 1985.

Mitchell, T.M., P.E. Utgoff, R.B. Banerji, "Learning by Experimentation: Acquiring and Refining Problem-Solving Hueristics," *Machine Learning*, R.S. Michalski, J.G. Carbonell, T.M. Mitchell (eds.), Tioga Publishing Company, Palo Alto, CA, 1983.

Mitchell, T.M., L.I. Steinberg, J.S. Shulman, "A Knowledge-Based Approach to Design," *Proceedings of the IEEE Workshop on Principles of Knowledge-Based Systems*, pp. 27-34, Dec. 1984.

Mitchell, T.M., S. Mahadevan, L.I. Steinberg, *LEAP: A Learning Apprentice for VLSI Design, Proceedings of the Ningh International Joint Conference on Artificial Intelligence (IJCAI-85)*, pp. 573-580, Aug. 1985.

Mitchell, T.M., R.M. Keller, S.T. Kedar-Cabelli, "Explanation-Based Generalization: A Unifying View," *Machine Learning*, vol. 1, no. 1, pp. 47-80, 1986.

Minton, S., *Learning Effective Search Control Knowledge: An Explanation-Based Approach*, Ph.D. Thesis, Carnegie Mellon University, Pittsburg, PA, 1988.

Mooney, R.J., G.F. DeJong, "Learning Schemata for Natural Language Processing," *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, pp. 573-580, Aug. 1985.

Mostow, J., "Toward Better Models of the Design Process," *The AI Magazine*, vol. 6, no. 1, pp. 44-57, Spring 1985.

Newell, A., H.A. Simon, "GPS: A Program that Simulates Human Thought," *Computers and Thought*, E.A. Feigenbaum and J. Feldman (eds.), McGraw-Hill, New York, NY, 1963.

Pazzani, M.J., "Failure-Driven Learning of Fault Diagnosis Heuristics," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 17, no. 3, June 1987.

Shortliffe, E.H., *Computer-Based Medical Consultations: MYCIN*, Elsevier, New York, 1976.

Smith, D., "What is Logic Synthesis?," *VLSI Design*, pp. 18-26, Oct. 1988.

Vander Zanden, N., D. Gajski, "MILO: Microarchitecture and Logic Optimizer," *Proceedings of the 25th Design Automation Conference*, pp. 403-408, 1988.