UNIVERSITY OF CALIFORNIA

Irvine

Library Insertion and Reuse of Datapath Components in High-Level Synthesis

A dissertation submitted in partial satisfaction of the

requirements for the degree Doctor of Philosophy

in Information and Computer Science

by

Roger Peñaranda Ang

Committee in charge:

Professor Nikil D. Dutt, Chair

Professor Daniel D. Gajski

Professor Fadi J. Kurdahi

1996

©1996

ROGER PEÑARANDA ANG

ALL RIGHTS RESERVED

The dissertation of Roger Peñaranda Ang is approved,

and is acceptable in quality and form for

publication on microfilm:

_____

_____

_____

Committee Chair

University of California, Irvine

1996

Dedication

To my wife, Natalie, and my son, Christian.

Contents

List of Figures vi

List of Tables viii

Acknowledgment ix

Curriculum Vitae x

Abstract xii

Chapter 1 Introduction **1**
1.1 Motivation 1
1.2 Current Problems with Component Reuse 3
1.3 Thesis Contributions and Overview 5

Chapter 2 Datapath Component Reuse in Behavioral Synthesis **7**

Chapter 3 Models and Representations **14**
3.1 High-Level Synthesis Model 14
3.2 Component Database Model 16
3.3 Representation of Behavior Bindings 20
3.4 Template Parse Tree 25
3.5 Summary 28

Chapter 4 Function Unit Allocation Algorithm **29**
4.1 Approach and Problem Definition 29
4.2 Algorithm Overview 32
4.3 Algorithm 34
4.4 Algorithm Complexity 37
4.5 Summary 39

Chapter 5 Operation Scheduling Algorithm **40**
5.1 Algorithm Overview and Examples 41
5.1.1 Example 1: Unconditional Assignments 43
5.1.2 Example 2: Conditional Assignments 47
5.1.3 Example 3: Conditional Loops 50

5.2 Basic Block Scheduling In Detail 52
5.2.1 Estimation of Critical Path Delay 53
5.2.2 Scheduling the Critical Path 54
5.2.3 Scheduling Remaining Operators 57
5.2.4 Search Reduction 60
5.3 Summary 62

Chapter 6 System Implementation and Experiments 63
6.1 Implementation & Experiment Design Flow 63
6.2 Experiments 68
6.3 Summary 92

Chapter 7 Conclusion 93
7.1 Summary of Dissertation 93
7.2 Summary of Contributions 94
7.3 Future Directions 96

Bibliography 99

List of Figures

1.1 Traditional high-level synthesis (HLS) 2

2.1 Adder: (a) RT component, (b) behavior in VHDL, (c) circuit synthesized from VHDL. 9
2.2 Synthesis RT models. 11

3.1 (a) Traditional HLS with generic components, (b) Synthesis for component reuse. 15
3.2 Interaction of HLS with models and representations. 16
3.3 Typical ALU. 17
3.4 RT-function sets describing simultaneous operations for ALU. . . . 18
3.5 Representation hierarchy in component database. 19
3.6 DFG templates for ALU RT functions. 20
3.7 Example VHDL and DFG. 21
3.8 Mapping DFG to RT functions. 21
3.9 RTDFG 23
3.10 Relationship between behavior and RTDFG data lifetimes. 23
3.11 Relationship between behavior, RTDFG transformation nodes, and links to RT structure. 24
3.12 Example behavioral templates and parse tree 26
3.13 Example parse tree match. 27

4.1 Example behavior to RT function mapping. 30
4.2 Allocation algorithm pseudo-code. 33
4.3 Example allocation binding: (a) initial graph. (b) first possible binding of node 1. (c) binding node 3 to Add_Subt1. (d) binding node 2 to Add_Subt1 to form solution. 35

5.1 Example of alternative ways to map operators to RT units that scheduler considers. 41
5.2 Example 1 - unconditional assignments: VHDL and data flow graph. 43
5.3 Operator mapping for calculating delay. 44
5.4 Scheduling options: (a) no operator chaining or multicycling. (b) combined operator chaining and multicycling. (c) exclusive operator chaining and multicycling. 45
5.5 Example 2 - conditional assignments: VHDL and equivalent ADD. 47

5.6 CDFG conditional branch translated to state transitions. 48
5.7 Scheduling conditions: (a) order operators are scheduled. (b) schedule for example with allocation of a multiplier and subtractor/comparator. 49
5.8 Example 3 - conditional loop: VHDL. 51
5.9 Conditional loop translated to state transitions. 51
5.10 Schedule for Example 3 with one Adder/Subtractor and one Subtractor/Comparator. 52
5.11 Scheduling of critical path related operators 56
5.12 Scheduling of remaining operators 59

6.1 ISE component capture interface. 64
6.2 VHDL description of experimental example. 70
6.3 Scheduling results for Integer Divider. 73
6.4 Scheduling results for Elliptic Filter, allocations 1 and 2. 74
6.5 Scheduling results for Elliptic Filter, allocations 3 and 4. 74
6.6 Scheduling results for Elliptic Filter, allocations 5 and 6. 75
6.7 Scheduling results for Experimental Example. 75
6.8 Scheduling results for Floating Point Adder/Subtractor. 76
6.9 Scheduling results for Greatest Common Divisor. 76
6.10 Area and performance of Integer Divider implementations. 78
6.11 Area and performance of Greatest Common Divisor algorithm implementations. 79
6.12 Area and performance of Experimental Example implementations. 80
6.13 Implementations of Elliptic Filter, allocation 1. 81
6.14 Implementations of Elliptic Filter, allocation 2. 82
6.15 Implementations of Elliptic Filter, allocation 3 and 4. 83
6.16 Implementations of Elliptic Filter, allocation 5. 84
6.17 Implementations of Elliptic Filter, allocation 6. 85
6.18 Implementations with partial use of the component allocation. . . 88
6.19 The effect of operation multicycling. 89
6.20 Design implementations for Floating Point Adder/Subtractor. . . . 90
6.21 Component allocation A for the floating point adder/subtractor. . 90
6.22 Comparison of Floating Point Adder/Subtractor implementations using allocation A. 91

List of Tables

5.1 Components used for scheduling examples. 43

6.1 Allocations of component types for experiments. 71
6.2 Allocations of component instances for experiments. 72

Acknowledgment

I would first like to thank the U. C. Irvine Chancellor's Office, the Graduate and Professional Opportunities Program (GPOP), the Semiconductor Research Corporation (SRC), and Dr. Nikil Dutt for their financial support. Of course, I would also like to thank the other members of my doctoral committee, Dr. Daniel Gajski and Dr. Fadi Kurdahi.

I would like to acknowledge and thank the many people who provided supporting work and input that helped make this dissertation possible. Thanks to Dr. Joe Lis, Dr. Elke Rundensteiner, and Dr. Loganath Ramachandran whose work helped me with my introduction to CAD for digital design. Special thanks to Dr. Viraphol Chaiyakul, Dr. Tedd Hadley, and Erica Juan whose work was used in conjunction with my research. Thanks to the other members and visitors of the CADLab, past and present: Smita Bakshi, Dr. Jie Gong, Tadatoshi Ishii, Dr. Sanjiv Narayan, Preeti Panda, Champaka Ramachandran, Alfred Thordarson, Dr. Frank Vahid, and Dr. Allen Wu. The things I learned from these people (there are wild ponies on Iceland and badly made sushi using pufferfish can actually kill you) have been some of the most memorable and enjoyable parts of my graduate education. But mostly, I would like to thank Nikil Dutt. All the things that I may have committed to text over these last several years do not begin to describe what I have learned from him.

Curriculum Vitae

Roger Peñaranda Ang

1989	B.S. in Information and Computer Science, University of California, Irvine, California. Graduated Cum Laude.
Various periods from 1989 to 1995	Teaching Assistant, Department of Information and Computer Science, University of California, Irvine, USA
1991-1995	Research Assistant, Department of Information and Computer Science, University of California, Irvine, USA
1991	M.S. in Information and Computer Science, University of California, Irvine, USA
Jan. - March 1996	Lecturer, Department of Information and Computer Science, University of California, Irvine, USA
1996	Ph.D. in Information and Computer Science, University of California, Irvine, USA Dissertation: *Library Insertion and Reuse of Datapath Components in High-Level Synthesis*

Publications

Selected conference papers

1. "An Algorithm for the Allocation of Functional Units from Realistic RT Component Libraries," Roger Ang and Nikil Dutt, *Proceedings of the 7th International Symposium on High-Level Synthesis,* May 1994.
2. "A Representation for the Binding of RT Component Functionality to HDL Behavior," Roger Ang and Nikil Dutt, *Proceedings of the Conference on Hardware Description Languages,* April 1993.

Selected technical reports

1. "Scheduling for Reuse of Datapath Components in the Interactive Synthesis Environment," Roger Ang and Nikil Dutt, *Technical Report 95-36, University of California at Irvine,* August 1995.
2. "Allocation of Functional Units from Realistic Component Libraries," Roger Ang and Nikil Dutt, *Technical Report 93-47, University of California at Irvine,* October 1993.
3. "On Linking RT Component Functionality to Abstract HDL Behavior," Roger Ang and Nikil Dutt, *Technical Report 92-97, University of California at Irvine,* revised July 1993.

Abstract of the Dissertation

Library Insertion and Reuse of Datapath Components in High-Level Synthesis

by

Roger Peñaranda Ang

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1996

Professor Nikil D. Dutt, Chair

This thesis presents a formulation for improved reuse of combinatorial datapath components in high-level synthesis. For combinatorial datapath components, I define new abstractions that encapsulate a broader range of components than previous models, and provide a tighter coupling of existing RT component libraries with high-level synthesis. I describe the role of datapath component reuse in high-level synthesis and discuss existing work related to this issue. I define the structures necessary for improved reuse of datapath components which include: a database that abstracts the functionality of combinatorial components, an intermediate representation to describe high-level behavior in terms of datapath component functionality, and a graph-matching mechanism for efficient mapping

of abstract operations to component functions. I illustrate the use of these models and representations in two algorithms: an algorithm to allocate function units to a behavioral description, and an algorithm to schedule operations of a behavioral description to a given set of components. I describe the experimental system implemented to demonstrate the models and algorithms, and the experiments conducted. The results of these experiments illustrate some of the advantages of this approach including development of design implementations with improved performance and reduced area.

Chapter 1

Introduction

1.1 Motivation

Within the electronics industry, the need to design and produce products faster has made automation of the design process a fact of life. Within the last several years, the rising complexity of the products being designed have made it clear that in order to effectively produce designs of increasing complexity, automation of the design process must move to higher level views of the product. This has been one of the driving forces behind the commercial use of design automation tools. Tools for automatic design generation or *synthesis* have widespread use at the transistor level and the logic gate level. The push to higher levels of abstraction has led to synthesis tools for register-transfer level design, and algorithmic/behavioral level design and beyond. However, despite some initial success with behavioral level design tools and widespread acceptance of synthesis tools for lower levels of design, widespread acceptance of behavioral level synthesis tools has been slow in coming. This is partly due to problems with effectively linking the results of synthesis at the higher levels with the tools used for the lower levels of

design. To understand this problem, a close look at the typical high-level synthesis (HLS) methodology is necessary.

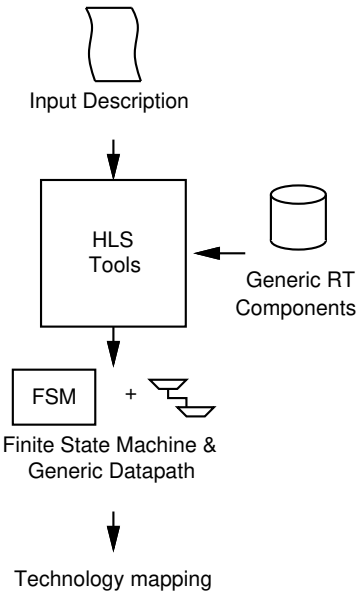


Figure 1.1: Traditional high-level synthesis (HLS)

Figure 1.1 illustrates the typical high-level synthesis methodology. In a typical HLS system, a high-level description of the design is input into the system of tools. The tools generate an implementation of the description using generic, technology-independent models of register-transfer (RT) components. The resulting implementation is a datapath consisting of interconnected instances of generic RT components and a finite state machine describing the controller. Register-transfer level synthesis will then refine this implementation. The generic RT components are mapped to RT components from an existing library, or logic synthesis is used to create a logic implementation of each generic component instance. This formulation of HLS is capable of producing some useful results, but for practical use, this method has severe limitations. When targeting a logic implementation,

HLS can produce efficient designs but can be time consuming because each implementation of an RT component is synthesized as if it never existed before. For example, if a design implementation has 2 16-bit adders, synthesis to logic will attempt to generate and optimize original implementations for each adder. Intuitively, this is redundant work. It would be more efficient to generate and optimize an implementation of a 16-bit adder and simply copy or *reuse* that implementation whenever a 16-bit adder is needed. In the long term, complete reliance on logic synthesis is not economical because this process does not allow the designer to reuse previously existing designs. Consequently, a long standing practice in the electronics industry is to develop and repeatedly use a library of RT components. Such reuse of an existing library of components in behavioral synthesis has had some success with digital signal processing applications. However, for other applications, behavioral synthesis is much less efficient in the way it reuses a library of components. My investigation into this problem revealed that the fundamental assumptions in the synthesis methodology are responsible for preventing efficient use of many RT component types.

1.2 Current Problems with Component Reuse

The traditional HLS process is constructed to be a technology-independent process. It is used to produce a result that can be targeted to a variety of implementation technologies (e.g. FPGA, standard cells). This means the synthesis process is constructed with the assumption that specific information about the implementation technology is unavailable. However, this is contrary to the basic idea

of reusing an existing RT library which has specific information about each component within it. Consequently, several aspects about the traditional formulation of behavioral synthesis prevent effectively reusing existing RT libraries:

- The generic RT component library used by a behavioral synthesis system acts as a canonical set of components. However, each designer may have a different or custom set of components he wishes to use. If the generic RT component library does not have a mechanism to be extended or customized, the designer may be forced to abandon the parts of his library that are not represented in the generic RT component library.
- Because of their technology-independent nature, the generic components may not well represent actual RT components. Often, the generic components represent simplified models of the behavior and timing of the actual RT components that would be used for the final implementation. As a result, the synthesis tools using the generic components can generate inefficient designs when trying to reuse real RT components.
- HLS tools are built with fixed assumptions about how behavior is mapped to components. The underlying algorithms of synthesis tools are constructed with assumptions about which generic components can perform certain operations. Unfortunately, when components exhibit functionality that does not conform to these assumptions, that functionality is ignored.
- HLS tools cannot exchange the component library they use. Because the assumptions about the components have been embedded within the algorithms, changes to the library may require developing different algorithms and new tools.

It is these aspects of the traditional formulation of HLS that I address in this thesis.

1.3 Thesis Contributions and Overview

My work as described in this thesis makes the following novel contributions:

1. I define a database representation to abstract RT component functionality and organize the component characteristics, such as area and delays, for use with behavioral synthesis.
2. I define a structure for synthesis, the Register-Transfer Data Flow Graph (RTDFG), that represents behavior in terms of RT component functionality and unambiguously links the input behavior with the RT structure being developed. This structure bridges the conceptual differences in the way input descriptions and RT structures operate.
3. I describe a new formulation of behavioral synthesis for reuse of combinatorial datapath components. In this formulation, all information about how components can be used for synthesis is encapsulated separately from the algorithms. Thus, algorithms constructed for this formulation can adapted to interchanged RT libraries for synthesis.
4. I describe a mechanism, the *template parse tree*, to efficiently match behavior to RT component functionality. This structure aids in the task of determining which RT components can perform particular operations.

The remainder of this thesis presents my work towards improved component library reuse in behavioral synthesis.

Chapter 2: I describe the role of reuse in behavioral synthesis and discuss works related to this issue and how they stand relative to my work.

Chapter 3: I give an overview of my formulation for synthesis, and describe the models and representations used in the formulation.

Chapters 4 and 5: I describe two algorithms: an algorithm to allocate function units to a behavioral description, and an algorithm to schedule operations of a behavioral description to a given set of components. The purpose of these algorithms is to illustrate the use of my models and representations and demonstrate some of the algorithmic complexities involved in synthesis to reuse RT components.

Chapter 6: I describe the system used for my experiments, and present my experimental examples and results. The algorithms in Chapters 4 and 5 are combined with other work to illustrate the impact component reuse has across the synthesis trajectory from input description to datapath RT structure.

Chapter 7: Summarizes my results and contributions, and discusses possible future work.

Chapter 2

Datapath Component Reuse in Behavioral Synthesis

In this chapter, I discuss the role of RT component reuse in behavioral synthesis, citing other works as examples of certain concepts. I also give an overview of my formulation for behavioral synthesis, and differentiate my work from previous research. As described in Chapter 1, traditional HLS is formulated as a technology-independent approach. The result of HLS is passed to lower synthesis tools that perform *technology mapping* to generate different implementations of the circuit design. For example, the design to be implemented can use logic gates from a gate array library, or components from an existing RT component library. To generate a logic implementation of the design, a logic gate implementation of each component of the datapath needs to be created using techniques such as those described in [CaTr89, ChGa90, Kipp91]. Coupled with logic optimization tools, efficient implementations can be created. But this process is time consuming so efficient implementations using components from an existing RT component library is key to wide acceptance of behavioral synthesis. The numerous advantages of design reuse in electronic design automation are discussed in detail in

[GiCa93]. These advantages include increased productivity by relieving designers of reinventing portions of a design, increased performance by enabling reuse of highly optimized blocks, and increased design quality through components with proven reliability. But to generate a design using components from an existing RT component library, post-HLS technology mapping must match the generic components in the datapath to specific components in an existing RT library, e.g., [Jha95]. However, inefficiencies in the way datapath components are used in synthesis cannot be dealt with in the post-HLS technology mapping phase because the inefficiencies are introduced during the HLS process itself.

The problems with RT component reuse in HLS are due to the fundamental assumptions about datapath components that are embedded in the HLS models and algorithms. Most approaches towards HLS assume a simple scheme for mapping behavioral operators to RT components. Synthesis tools often assume an abstract operator directly maps to a single RT operation which in turn can be performed by a component. For example, an abstract addition, such as + in a hardware description language (HDL), will map to an *add* operation which can be performed by a RT-component such as an Adder. This model of RT component functionality works with digital signal processing (DSP) applications because typical DSP datapath components conform to this model. Consequently, there has been some initial success in using behavioral synthesis for DSP applications. However, for applications using a wider range of components, this model does not support the full functionality of commonly used components. A simple examination of existing RT-components from standard databooks reveals many interesting databook RT-components are multi-functional and have multiple outputs. That is, there are databook components that can perform several functions simultaneously,

producing several outputs in a single state. Because of the simplified component models used for synthesis, current HLS systems typically ignore some of the the functionality of such components, making reuse of such components inefficient. Figure 2.1 illustrates this problem. Figure 2.1a is a 4-bit adder with a carry output that a designer wishes to use during synthesis. The behavior of the adder is described abstractly in Figure 2.1b. If this description were given to a typical HLS system, it would likely generate the circuit shown in Figure 2.1c rather than use the adder in Figure 2.1a.

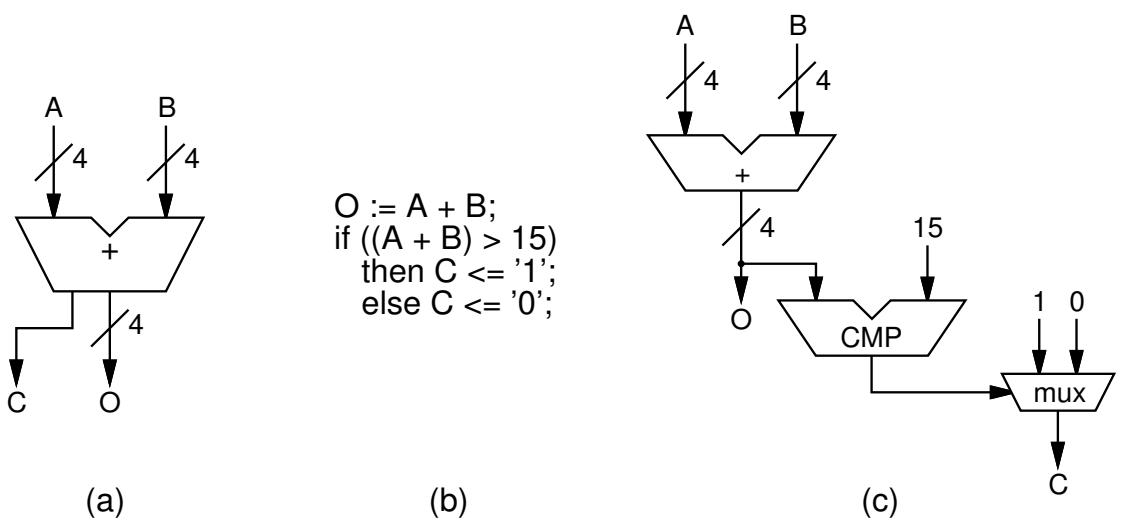


Figure 2.1: Adder: (a) RT component, (b) behavior in VHDL, (c) circuit synthesized from VHDL.

Much of the well known work in HLS was based on the assumption that RT components could perform one operation per function unit per clock cycle. Works based on this assumption include representations for synthesis [LGCP91, Thom90, CaTa89, Rose86], function unit selection and allocation algorithms [GCDM93, RaKu92, RuGB90], operation scheduling algorithms [Camp91, PLNG90, PaKn89,

PaGa87], and entire synthesis systems [MiLD92, CaWo91, Marw86, GrKP85]. These HLS algorithms and representations are constructed on the assumption that all RT component functions are represented by an operator symbol in an HDL. But this is not true. For example, there is no symbol to represent the function of a multiply-accumulator. There exist datapath components with complex functions that cannot be described with a single HDL operator. Consequently, algorithms and representations built with the "one operator per RT component per state" assumption cannot utilize such components.

A few works have recognized the limitation that the "one operator per RT component per state" assumption places on datapath component use in synthesis. [GCDM95] and [LMD94] describe formulations to utilize complex RT component functions. [GCDM95] uses a graph-based representation and graph topology matching to map complex behavior to RT structures, while [LMD94] uses implication rules as the mechanism to map complex expressions to RT units using integer programming. Both of these works address using RT components with complex functions such as a multiply-accumulator. However, these works still assume that each RT component produces only a single data output (i.e., performs a single complex function) per state. But this assumption is not true for databook components. Consider a combinatorial comparator that compares two values and produces "less than", "equal to", and "greater than" outputs. All these outputs can be produced in a single state for a given pair of values. In effect, this comparator can do three simultaneous comparisons of the input values. Thus far, no other work has addressed this type of RT component concurrency at the behavioral level of synthesis. [KnWi92] and [WPAV92] describe representations that recognize the multi-I/O nature of RT components. With these representations, it is possible to

map a set of interconnected operations that produce multiple outputs to an RT component. But these works have focused on RTL synthesis and have not been extended to HLS. The likely reason for this is that the multi-I/O structure of operations in these representations are not well-suited for describing the semantics of HDLs such as VHDL and Verilog. The input to the algorithms using these representations is an RTL description of operations, which is more closely related to the result of HLS rather than the behavioral input to HLS.

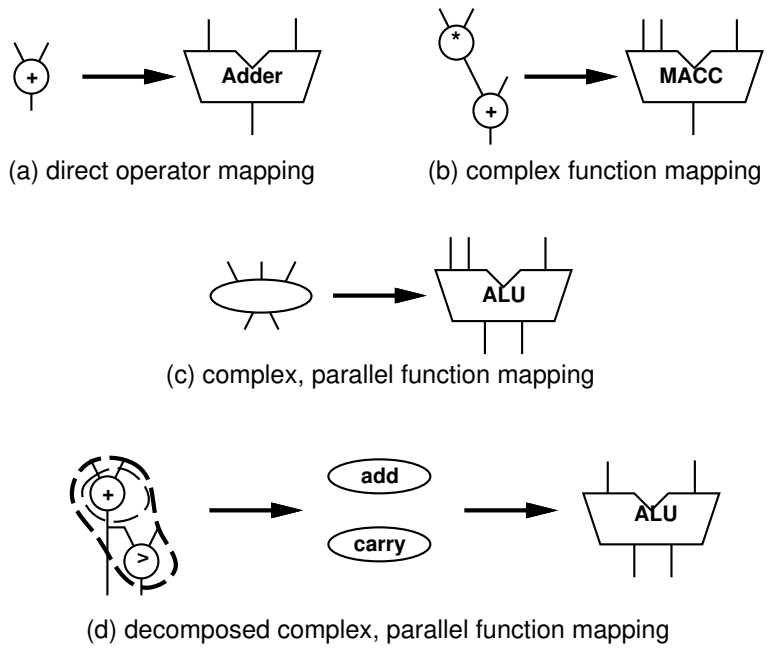


Figure 2.2: Synthesis RT models.

Figure 2.2 illustrates the conceptual differences between the various models of RT component functionality used in synthesis. Figure 2.2a is the basic model the bulk of synthesis work is based on: 1 operator per function unit. Figure 2.2b illustrates the complex function model [GCDM95] and [LMD94] are able to deal with: multiple HDL operators mapping to an RT unit that produces a single output. Figure 2.2c represents the model of RT component functionality in [KnWi92]

and [WPAV92]: a design description with multi-I/O operators which map directly to RT operations. Figure 2.2d is the model my formulation of HLS is based on: multiple operators mapping to concurrent RT functions of a function unit.

At a conceptual level, the problem of mapping an HDL model to a set of RT units has similarities to mapping a programming language to a microprocessor. Therefore, I adopted an approach analogous to the approach described in [GlGr78]. This approach described a method that used a table to map intermediate code to a specific microcode implementation on a fixed, single processor architecture. In effect, [GlGr78] defined an interchangeable "back-end" for software language compilers that could be changed for each specific processor the compiler was implemented on. This approach has been applied to HLS in [LMPa94], which describes instruction-set matching and selection for retargetable code generation for DSP and Application Specific Instruction-set Processors. However, the problem of reusing datapath components is larger in scope than retargetable code generation. For HLS, the problem would be analogous to performing the same code generation tasks for an architecture with multiple processors, where each processor may have a different instruction set.

For synthesis, [Kahrs86] expanded the idea of using a table to map instructions to RT structure. Instead of a table, [Kahrs86] proposes a library of "parts." Each part may be able to perform several functions and each function has a subgraph representation. The algorithm proposed in [Kahrs86] used graph matching to map behavior to generic parts to be generated later. Along this same line of thinking, [RuGB90] described a "unit table" which is used to map mutually exclusive operators to multi-function units. [SMZG96] used a variation of this library

model using syntactic expression matching to map behavior to specific implementations of datapath components. [RaKu92] proposed a "template" library and a regularity extraction algorithm that may be used for synthesis. This idea of "template" mapping, usually formulated as graph matching, has also been used in HLS in various ways. For example, [CKPR93] and [LMPa94] used templates for mapping instruction sets to specialized processors, and [GCDM93] and [Mar93] used templates to map algorithms to predefined RT structures. But in all of these methods, the libraries and models are not sufficiently defined to deal with components that generate multiple data outputs. In my work, like [Kahrs86], I use the idea of having a component library with behavior templates for the components. Similar to [SMZG96], this library is customized to the component set that is available for the specific implementation technology. But I define a library representation for HLS for multi-function, multi-output components, i.e., components with complex, concurrent behavior.

Taking these works into consideration, my work in this thesis is an evolutionary advance for HLS, consolidating various features of some of the previous works applied in other areas and adding the necessary definitions and mechanisms to integrate the features into a cohesive formulation for HLS. The formulation of HLS that I present in the next chapter is constructed to:

- represent the complex, concurrent behavior of datapath RT components,
- bridge the disparities between HDL operator semantics and RT component functionality, and
- provide the means for synthesis tools to recognize the complex, concurrent behavior of datapath RT components.

Chapter 3

Models and Representations

In this chapter, I describe the models and representations used for my formulation. First, I describe my overall formulation of HLS. Then I present the component database model that abstracts the functionality of RT datapath components. I describe the Register-Transfer Data Flow Graph (RTDFG), that represents behavior in terms of RT component functionality and unambiguously links the input behavior with the RT structure being developed during synthesis. Finally, I describe the *template parse tree*, a mechanism to efficiently match behavior to RT component functionality. This chapter describes in brief the representation work presented in [AnDu92].

3.1 High-Level Synthesis Model

One of the main ideas of my formulation of synthesis is to use detailed models of the components in a specific RT component library rather than generic models. These component models can capture the complex, concurrent behavior available in the specific RT components. By using such models, my HLS formulation better utilizes these components and avoids the need for complex technology mapping

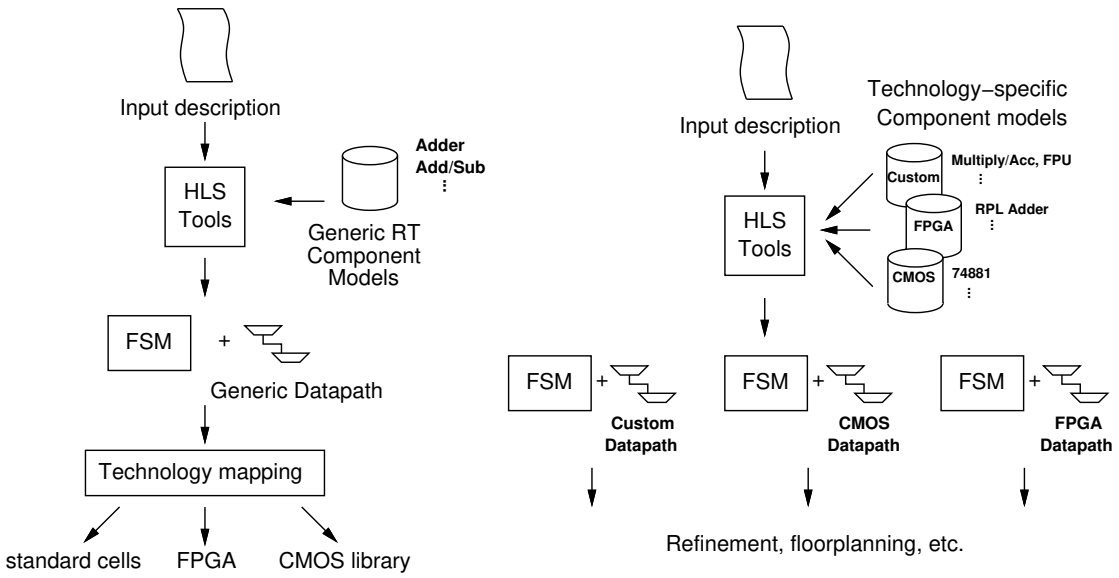


Figure 3.1: (a) Traditional HLS with generic components, (b) Synthesis for component reuse.

after Synthesis Figure 3.1 illustrates how my overall approach differs from traditional HLS. In traditional synthesis, the generic component models are loosely related to the actual implementation of the datapath. That is, HLS assumes there is some implementation of the components but does not concern itself with the actual components. Instead a post-HLS technology-mapping step associates each generic component with an actual implementation. In my synthesis approach, HLS is tightly coupled to the components available for implementing the datapath. The component library used in HLS directly reflects the actual available datapath components, in effect, doing technology-mapping during HLS.

Figure 3.2 outlines the interaction and structure of my HLS formulation. In my system, all information concerning RT component characteristics and behavior is maintained outside of the synthesis tools. The synthesis tools are constructed

without assumptions about the functionality of RT components. The tools query the component database and template parse tree to obtain information about components or to determine the bindings of HDL operators to RT component functions. The synthesis tools are constructed to operate using the RTDFG to build and maintain the bindings of HDL behavior to RT component functionality.

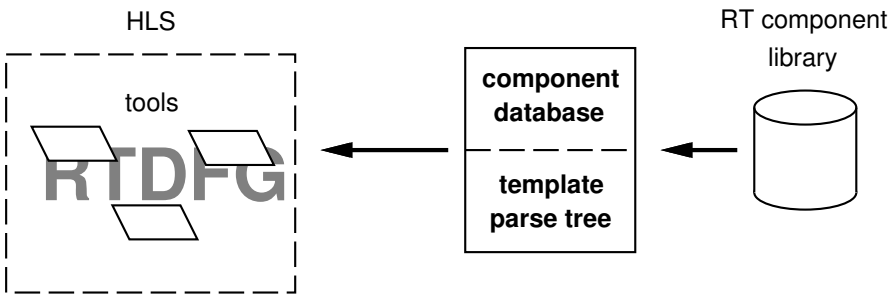
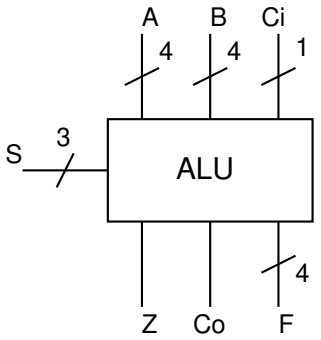


Figure 3.2: Interaction of HLS with models and representations.

3.2 Component Database Model

The component database organizes the physical and behavioral information about all available components. The database also defines the abstraction of the RT components' functionality. I illustrate the semantics of the component database through an example specification of a unit. The unit used for this example is the ALU shown in Figure 3.3, a variant of an ALU described in [Mano88]. This ALU is capable of performing various operations as shown. To develop a model of this ALU, I need to determine:



S	operation
000	F <– A+Ci, Co <– (A+Ci) > 15, Z <– (F=0)
001	F <– A+B+Ci, Co <– (A+B+Ci) > 15, Z <– (F=0)
010	F <– A+$\overline{B}$+Ci, Co <– (A+$\overline{B}$+Ci) > 15, Z <– (F=0)
011	F <– A+$\overline{Ci}$, Co <– (A+$\overline{Ci}$) > 15, Z <– (F=0)
100	F <– A and B
101	F <– A or B
110	F <– A xor B
111	F <– $\overline{A}$

Figure 3.3: Typical ALU.

- the various modes of operations (RT operations) of the ALU, i.e., each control line configuration of the ALU,
- the input and output pin characteristics, e.g., bit widths, and
- the relationships of data inputs to outputs for each RT operation.

From this information, I develop the RT operations for this ALU (see Figure 3.4). For each RT operation the ALU can perform there is an associated set of functions. In each set, there is an RT function associated with each output that produces meaningful data for that operation. Each RT function has parameters which correspond to inputs of the ALU. RT functions in the same set can be performed simultaneously, while those not in the same set must be performed in separate time steps. The names for the RT functions in this example are arbitrary and need not be unique to this component. A named RT function can be used by any number of components, as long as its parameters and behavioral templates are consistent with all components with that RT function.

With these sets, I have an abstraction of the functionality for this specific ALU which is modeled in the database representation In the database, the behavior of each component is decomposed into RT operations and RT functions.

(S : 000)	(S : 001)	(S : 010)	(S : 011)
INC(A,Ci) –> F INC_CARRY(A,Ci) –> Co INC_ZERO(A,Ci) –> Z	ADD(A,B,Ci) –> F CARRY(A,B,Ci) –> Co ZERO(A,B,Ci) –> Z	SUB(A,B,Ci) –> F SCARRY(A,B,Ci) –> Co SZERO(A,B,Ci) –> Z	DEC(A,Ci) –> F DEC_CARRY(A,Ci) –> Co DEC_ZERO(A,Ci) –> Z
(S : 100)	(S : 101)	(S : 110)	(S : 111)
AND(A,B) –> F	OR(A,B) –> F	XOR(A,B) –> F	NOT(A) –> F

Figure 3.4: RT-function sets describing simultaneous operations for ALU.

Each component can have a set of RT operations. Each RT operation represents a distinct mode of operation that the RT component can be configured for. Each RT operation can have a set of RT functions that can be performed concurrently by the component under this particular mode of operation. Each RT function is associated to a *behavior template*. For this discussion, I assume behavior templates are fragments of data flow graphs (DFGs). Figure 3.5 illustrates the hierarchical decomposition of RT component functionality represented in the component database.

The purpose of this database is to define the relationship between three different abstractions:

1. RT level specification of components describing the input and output pins and the input-to-output functionality of each component.
2. *RT function* sets — abstractions of the RT functionality of the components.
3. Intermediate behavioral representation — the abstraction of the input specification.

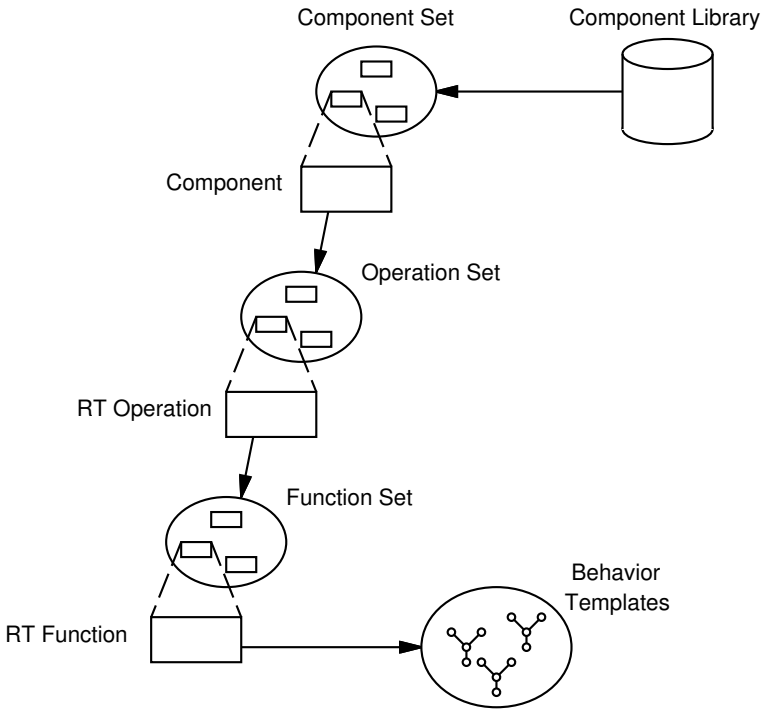


Figure 3.5: Representation hierarchy in component database.

To use the RT functions in HLS, a mapping from behavior to the RT functions is needed. This is the purpose of the behavior templates. Figure 3.6 shows some templates I specified for the RT functions of the ALU. These templates define the behavioral meaning of the RT functions and provide a mechanism to map portions of behavior to RT functions and, consequently, to RT components. How much of the ALU's functionality and the efficiency of ALU function usage depends on how extensive the templates are. Through these templates, the library can be customized for the particular way a designer wishes to use the components. The RTL functionality a synthesis system supports is represented by the templates present for the RT functions of the components. Information such as the number representation of data, the mapping of data values to component ports, etc., is

also described with the templates. In this manner, the templates allow the library to be customized to the HDL style/guidelines the designer uses for synthesis.

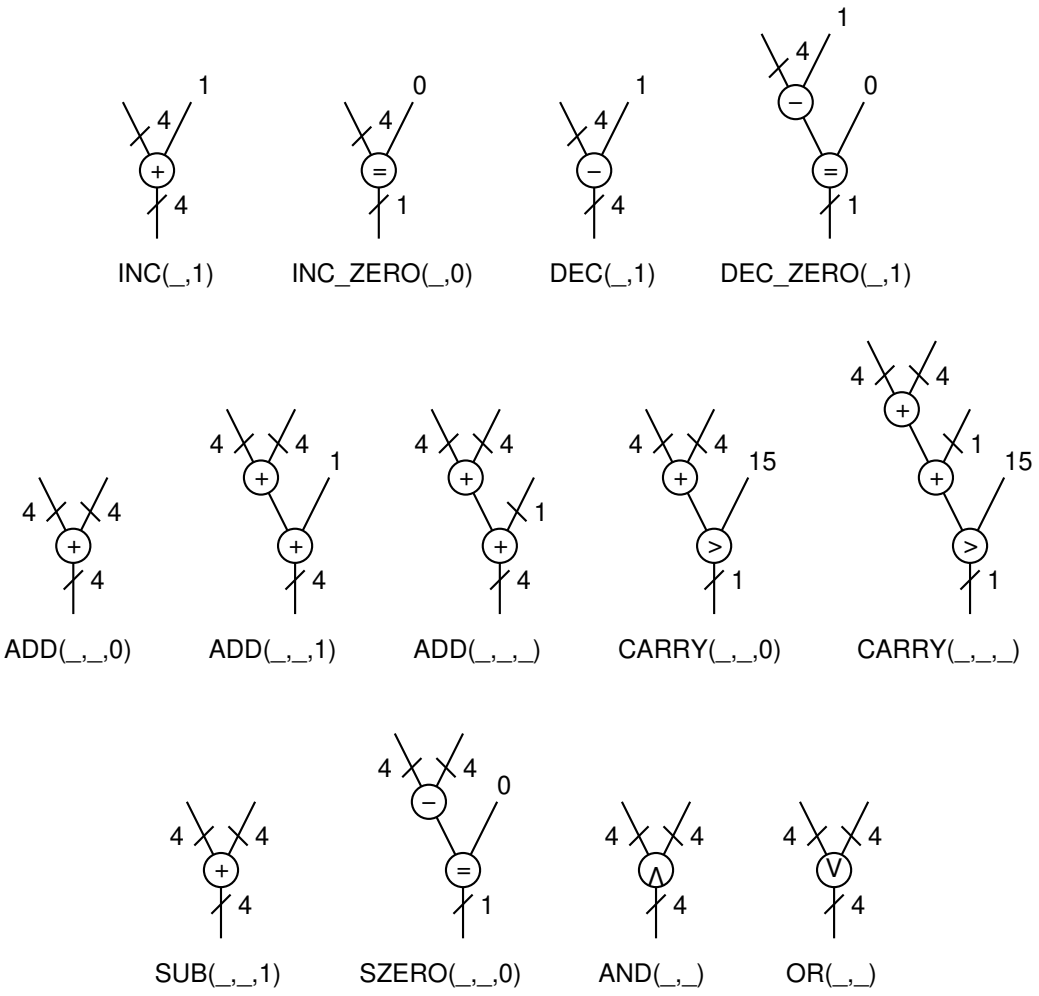


Figure 3.6: DFG templates for ALU RT functions.

3.3 Representation of Behavior Bindings

With the specification of the ALU in the previous section, I now illustrate the representation of the binding of behavior to the ALU. I start with the two

lines of VHDL shown in Figure 3.7a. These can be parsed into a DFG shown in Figure 3.7b. Function unit allocation using the previously described templates for the ALU will find the DFG can be "covered" as shown in Figure 3.8, indicating the ALU can be used in this instance. Scheduling based on this covering and allocation will find that these two behavioral operations can be performed in a single time step since the RT functions they are bound to belong to the same RT operation. At this point, the RT Data Flow Graph comes into use.

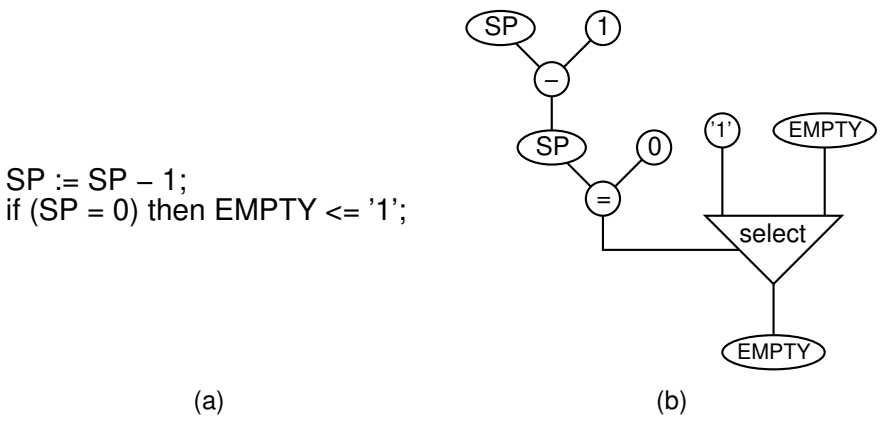


Figure 3.7: Example VHDL and DFG.

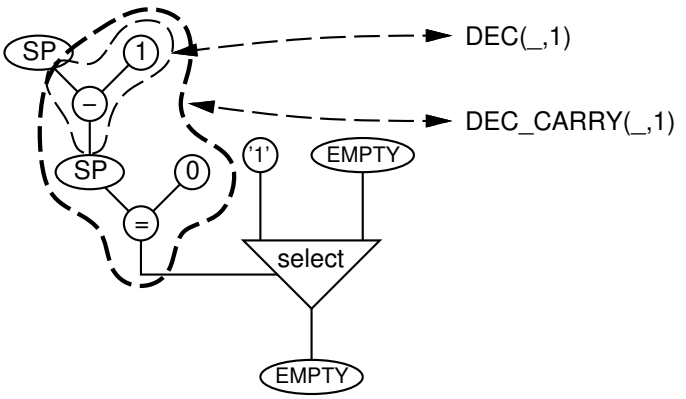


Figure 3.8: Mapping DFG to RT functions.

To construct the RTDFG, global data flow analysis is needed as well as construction of a state-transition graph. The key features of the RTDFG, illustrated in Figure 3.9, are:

- State transition graph — provides a "timeline" through the control paths of the design.
- Data lifetimes — based on the standard compiler concept of *use-definition chains* [AhSU86]. For every data carrier (variable) each *use* (variable as an operand) is linked to every applicable *definition* (assignment to that variable). A definition is "live" in a state if a control path to an applicable use goes through that state.
- Transformation nodes — combined with the data lifetimes, these form a representation of global data flow for the specification, and provide *explicit functional mappings* of inputs to outputs. These nodes indicate how data is transformed and the type of transformation done. For each node, there is a set of data lifetimes that are inputs of the transformation, a data lifetime that is the output of the transformation, and a mapping to an RT function specifying the type of function and relating how the inputs and output correspond to RT function parameters. In addition, these nodes maintain links to the behavior and RT structure to indicate RT unit bindings to behavior. The link to behavior specifies the portion of behavior covered by the RT function, and the link to the RT structure indicate the FU that will perform the RT function.

Figure 3.10 shows how the RTDFG data lifetimes relate to the DFG of our example. The last use of SP ends a lifetime, while the assignment to SP begins another

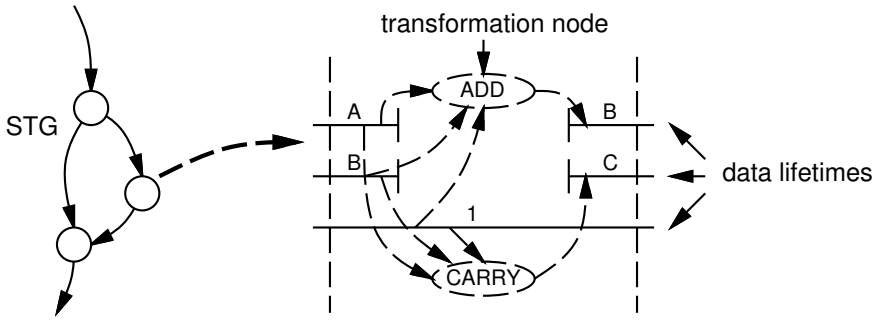


Figure 3.9: RTDFG

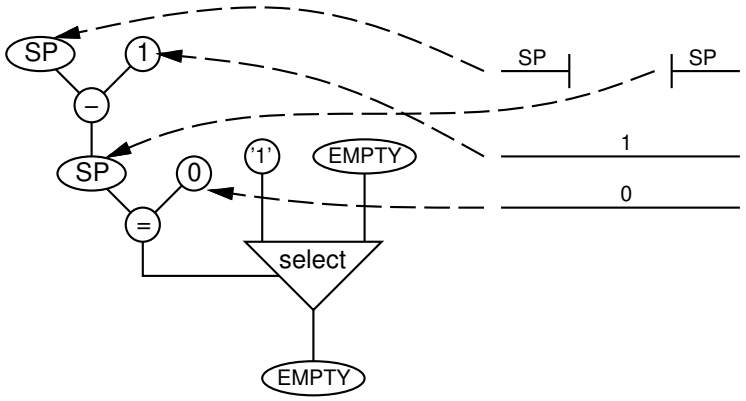


Figure 3.10: Relationship between behavior and RTDFG data lifetimes.

lifetime. Figure 3.11 illustrates the use of the RTDFG transformation nodes for a single state in a design. These nodes represent instances of RT functions to be performed. The solid arcs to the nodes indicate input data lifetimes while the arcs away from the nodes indicate output data lifetimes. Figure 3.11 also shows how links to the RTDFG transformation nodes are used to represent binding of the behavior to the RT structure.

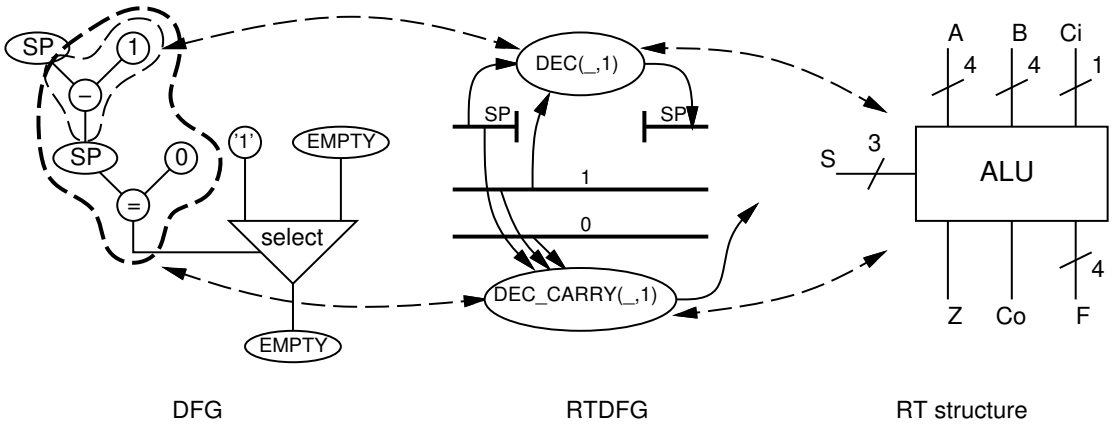


Figure 3.11: Relationship between behavior, RTDFG transformation nodes, and links to RT structure.

The RTDFG provides an additional level of abstraction between the abstract HDL behavior and the RT structure. It bridges the difference between the semantics of intermediate behavioral representations such as DFGs and RTL functionality. The RTDFG provides an unambiguous mapping between the HDL model and the RT structure while enabling greater usage of the functionality of realistic components.

3.4 Template Parse Tree

A basic task in many HLS algorithms is determining how behavior can be mapped to RT components. With my database representation, this task is equivalent to trying to find a behavior template that matches to a given segment of the intermediate behavioral representation of the design description. Obviously, linearly comparing each behavior template of a library to a particular segment of the design is impractical. To efficiently do this search, I construct a *template parse tree* structure from all the behavior templates of the components in a library database. I illustrate the construction and semantics of the template parse tree with an example.

The top of Figure 3.12 shows some example templates and the RT functions they represent: the addition of two 4-bit numbers, addition of two 4-bit numbers and a carry-in bit, the "carry" function for the addition of two 4-bit numbers, decrementing of a 4-bit number, and a "is-zero" function for the decrementing of a 4-bit number. At the bottom of Figure 3.12 is the parse tree constructed from the templates. The first level of internal nodes of the tree are write nodes that describe the outputs of the templates, i.e., the output bitwidth. There will be only one write node for all templates with output of a particular bitwidth. The other internal nodes of the tree are operators. The leaf nodes of the trees are read nodes which describe the inputs to the template. Edges (going down) from an internal node are alternative input sets; in this case, sets of input pairs. This arrangement maintains the context of the templates, i.e., the particular combinations of inputs and operations. Input pairs that have leaf nodes are labeled with the RT functions that can perform that template. Figure 3.13 illustrates how a match is done. First

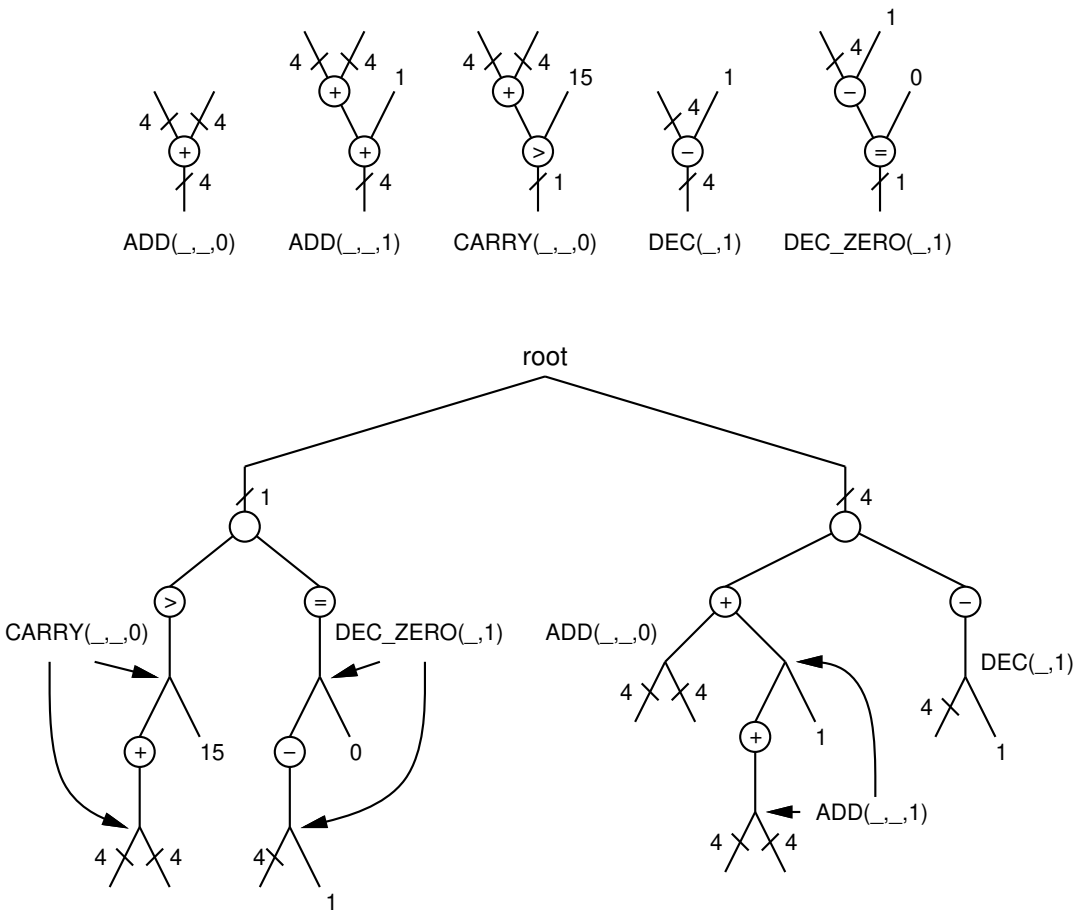


Figure 3.12: Example behavioral templates and parse tree

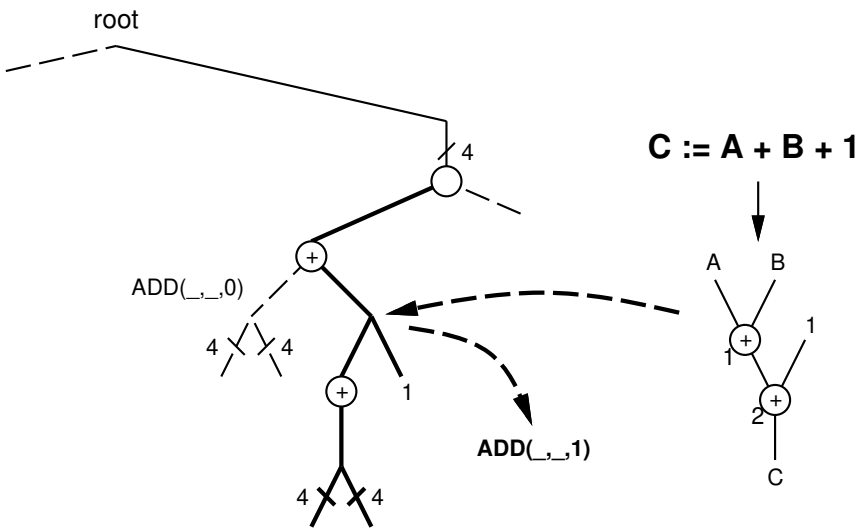


Figure 3.13: Example parse tree match.

the appropriate write node is found from the root. Then the operator that outputs to the write node is matched. Now for operators, the set of inputs to the operator is matched. In this case, $+_2$ has as inputs $+_1$ and a constant **1**. Matching of input sets is recursively done for each subexpression. In this case, we continue by matching the two 4-bit variable inputs of the $+_1$ subexpression. The component database is used in conjunction with the template parse tree to determine how behavior can map to components. Given some design, the template parse tree is searched to find templates that match segments of the design. Querying the component database determines what components can perform each template, and which templates can be performed concurrently on a single component.

3.5 Summary

I have described my overall approach to synthesis and how it differs with traditional synthesis. The three key elements that make my approach to reuse combinatorial datapath components possible are the component database model, the RTDFG, and the template parse tree. The component database represents a hierarchical abstraction of the functionality of the RT components in a particular library. The RTDFG is a representation to be used during synthesis to model the input behavior in terms of the RT component functionality. The template parse tree is the mechanism to perform the mapping of input behavior to RT component functions. In the next chapters, the example algorithms will illustrate how these elements are used for synthesis.

Chapter 4

Function Unit Allocation Algorithm

I describe in this chapter an algorithm to allocate function units to a behavioral description. This algorithm illustrates the use of some of the structures described in Chapter 3. It explores how algorithms can exploit the complex, concurrent functionality in RT components. It also illustrates some of the algorithmic complexity involved in using this model of RT component functionality. This chapter summarizes the work described in [AnDu93].

4.1 Approach and Problem Definition

The problem of allocating RT components for a given description is equivalent to mapping operators from the input description to a set of RT components. In my formulation, this becomes the task of binding the semantics of an imperative HDL to the semantics of the RT functions of components from a given library. I approach the problem of binding behavior to realistic functional units as a 2-phase task. In the first phase, abstract behavior is mapped to an equivalent set

of RT functions. The template parse tree is used for this purpose. For the set of RT functions, which RT functions can be performed simultaneously is determined in part by data flow analysis and mutual exclusiveness of conditions. Mutual exclusiveness of conditions is determined using condition vectors as described in [JCG94]. In the second phase, these RT functions are assigned to RT units. The algorithm I present here focuses on the second phase of this binding process. For the first phase, all possible mappings of behavior to RT functions are generated. Each possible mapping is then passed as input to the algorithm described below. Figure 4.1 shows one possible mapping of an example piece of behavior to a set of RT functions.

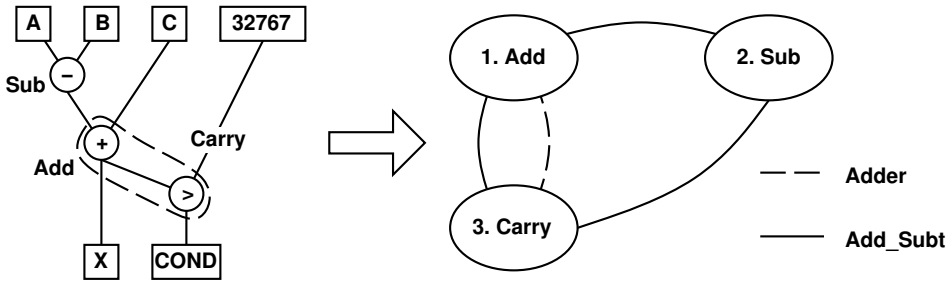


Figure 4.1: Example behavior to RT function mapping.

Here is a more formal definition of the allocation problem I have just described. Given:

L : a user-specified component library consisting of:

- **C** : a set of RT components
- **O** : a set of RT operations, each component in **C** can perform one operation at any time
- **F** : a set of RT functions, multiple RT functions can be performed in a single RT operation in **O**.

This library defines the mappings between RT components, RT operations, and RT functions. For example, $\mathbf{C} \times \mathbf{O} \rightarrow \mathbf{F}$.

I : an input graph. The first phase of binding will produce this graph which describes which RT functions of a set, $\mathbf{IF} \subseteq \mathbf{F}$, can be performed simultaneously. I define this graph such that:

- for each RT function in **IF** there is an associated node in the graph,
- and there is an edge between two nodes if
  1. both RT functions can be performed simultaneously on the same type of RT unit,
  2. or the RT functions are not to be performed simultaneously and both can be performed by the same type of RT unit.

  This is determined from the information in the library **L**, and the edge between the nodes is labeled with the name of the component type that can perform the RT functions.

The input design description can be scheduled or not scheduled. For a description that is not scheduled, the above edges can be determined from data dependencies.

The allocation algorithm will produce a graph **B** where each node in **B** has an associated RT unit, and a set of RT functions. That is, this algorithm performs the mapping: $\mathbf{L} \times \mathbf{I} \rightarrow \mathbf{B}$ | each node in **B** has

- an RT unit, $\mathbf{c} \in \mathbf{C}$.
- a set of RT functions, $\mathbf{f} \subseteq \mathbf{IF}$, bound to **c**.

4.2 Algorithm Overview

The input graph **I** is a compatibility graph describing how RT functions can share RT units from the library **L**. The algorithm does a heuristic search for possible solutions, recursively transforming the input graph to find possible bindings of the RT functions **IF** to RT units **C** . The algorithm will try each node in the graph as a starting point. For each node in the graph, the algorithm will either bind the node to an already allocated RT unit, or try an RT unit allocation according to component-based cost functions. If a partial solution is larger than or redundant to an already found solution, searching of that partial solution is abandoned. The pseudo-code in Figure 4.2 describes the search algorithm for possible allocations/bindings of RT units.

FORM_START_LIST() orders the nodes into a list, START_LIST, so that nodes considered good starting points (or most critical) are searched first. The nodes are rated according to the cost function:

$$\text{Cost} = \text{max. operation cost} \times \text{output bitwidth}$$

Each function associated with a node in the graph can be performed by one or more units in **C**. Each component must be in a particular mode of operation, $\mathbf{o} \in \mathbf{O}$, to perform that function, e.g., shift right, increment, count up. The costs of each RT operation in **O** is given by the user in the form of a table. These costs indicate the relative complexity of various classes of functions (e.g., arithmetic functions vs. logic functions). If a function can be performed by different RT operations, the most expensive operation is considered for the cost function. These operation costs are independent of the actual bitwidth of the data used in the function, so

```
a: START_LIST = FORM_START_LIST();
b: for all n ∈ START_LIST do BRANCH_ON_BIND(n);

   BRANCH_ON_BIND(n) {
      if RT_FUNCTION(n) can't bind to an available RT unit then
         find possible bindings and order by RT unit rating
c:       for each possible binding do
            possible_solution = ALLOCATE_AND_BIND_NEW_UNIT(n);
            if AREA(possible_solution) > AREA(best_solution) then go to c; /* next */
            if REDUNDANT_SOLUTION(possible_solution) then go to c; /* next */
            elseif no RT functions left then
               if AREA(possible_solution) < AREA(best_solution) then
                  save possible_solution as best_solution;
               endif
            else there are RT functions left,
               n = FIND_BINDABLE_FUNCTION();
               if an RT function can bind to an RT unit then BRANCH_ON_BIND(n);
               else no RT function can bind to an RT unit
d:                for each remaining RT function, n do BRANCH_ON_BIND(n);
               endif
            endif
         endfor
      else RT_FUNCTION(n) can bind to an available RT unit m
e:       possible_solution = BIND_RT_FUNCTION_TO_UNIT(n, m);
         if AREA(possible_solution) > AREA(best_solution) then return; /* stop */
         if REDUNDANT_SOLUTION(possible_solution) then return; /* stop */
         elseif no RT functions left then
            if AREA(possible_solution) < AREA(best_solution) then
               save possible_solution as best_solution;
            endif
         else there are RT functions left,
            n = FIND_BINDABLE_FUNCTION();
            if an RT function can bind to an RT unit then BRANCH_ON_BIND(n);
            else no RT function can bind to an RT unit
               for each remaining RT function, n do BRANCH_ON_BIND(n);
            endif
         endif
      endif
   }
```

Figure 4.2: Allocation algorithm pseudo-code.

the output bitwidth of the function is also factored in. This cost function is used to provide a measure of the potential difficulty to implement a given function using a given component library, and the START_LIST is ordered so that the most costly functions are considered first in the search.

BRANCH_ON_BIND() is the main routine that recursively builds possible solutions. RT_FUNCTION(n) refers to the RT function in **IF** that is associated with node n. ALLOCATE_AND_BIND_NEW_UNIT(n) transforms the graph to allocate an instance of an RT unit in **C** and bind RT_FUNCTION(n) to it. BIND_RT_FUNCTION_TO_UNIT(n, m) binds the RT function of node n to an RT unit m allocated to the graph. AREA(`solution`) is the estimated area cost for the RT unit allocation for `solution`. REDUNDANT_SOLUTION(`solution`) determines if the partial solution `solution` is a subset of a solution already found, or a partial solution already abandoned. SELECT_BINDABLE_RT_FUNCTION() returns a node with an RT function that can bind to an already allocated unit.

4.3 Algorithm

Initially, all RT functions are unbound and no RT units have been allocated, so all nodes in the graph have an associated RT function (see Fig. 4.3a). The algorithm begins on a selected node and recursively applies BRANCH_ON_BIND(). The algorithm will try each node as a potential starting point to find alternative solutions. In BRANCH_ON_BIND(), if the RT function for the selected node cannot bind to an available RT unit, or no RT units are allocated yet, a list of

units that can be bound to that function is made. This list is ordered by *RT unit ratings*, which rates how many of the remaining functions in the graph a unit can perform.

$$\text{Unit rating} = \frac{\text{sum of operation costs of nodes that can bind to unit}}{\text{area of unit}}$$

ALLOCATE_AND_BIND_NEW_UNIT() is applied to each of these possible bindings to produce a partial solution (see Fig. 4.3b). If the RT function for the selected node can bind to an available RT unit, BIND_RT_FUNCTION_TO_UNIT() is applied (see Fig. 4.3c). In either case, the partial solution produced is abandoned if it is larger than the best solution found so far, or is redundant to another solution.

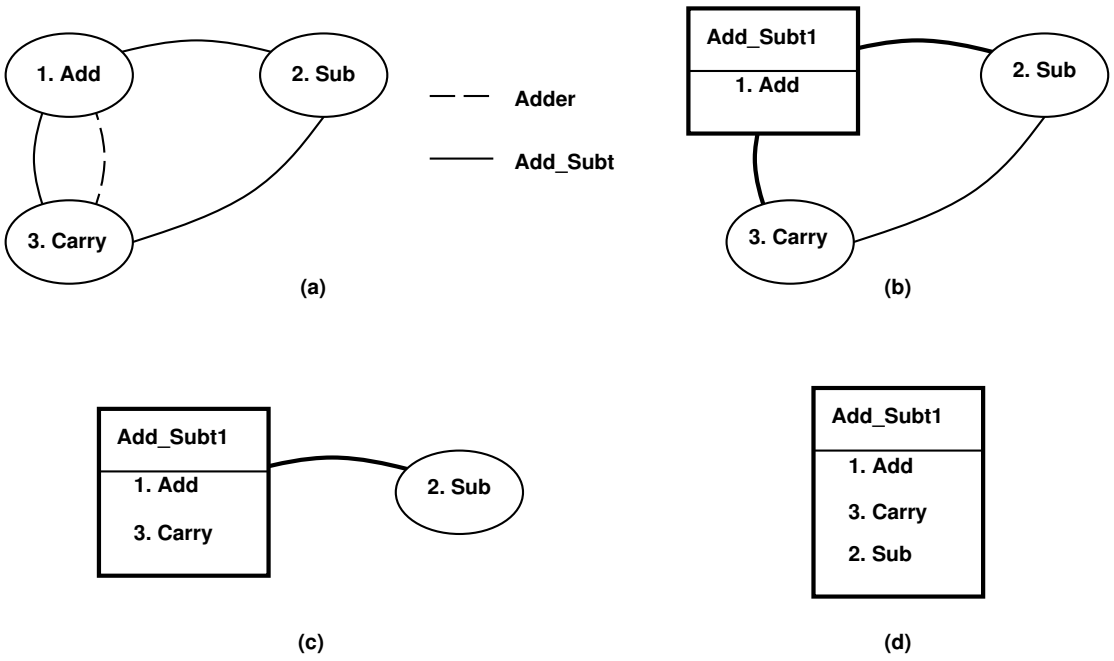


Figure 4.3: Example allocation binding: (a) initial graph. (b) first possible binding of node 1. (c) binding node 3 to Add_Subt1. (d) binding node 2 to Add_Subt1 to form solution.

If no suitable binding or additional allocation is found, the current partial solution is abandoned. If all the RT functions have been bound, then the current solution is saved if it is smaller than the best solution found so far. Otherwise, FIND_BINDABLE_RT_FUNCTION() selects which, if any, RT function should be bound to an available unit. If an RT function can bind to an available unit, BRANCH_ON_BIND() is applied to the node for the selected function. Else, BRANCH_ON_BIND() is applied to each of the remaining function nodes in the graph.

Fig. 4.3 illustrates the execution of the algorithm in Figure 4.2. From the initial graph, Fig. 4.3a, the first iteration of line **b** of the algorithm will apply BRANCH_ON_BIND() on node 1. Since there are no RT units allocated yet, BRANCH_ON_BIND() will form a list of possible bindings for node 1. In this case, node 1 can bind to either an Adder/Subtractor or Adder. The first iteration of line **c** of the algorithm will apply ALLOCATE_AND_BIND_NEW_UNIT() to allocate an Adder/Subtractor and bind node 1 to it, producing the graph in Fig. 4.3b. Then, FIND_BINDABLE_FUNCTION() will select node 3 for the next recursion of BRANCH_ON_BIND(). When BRANCH_ON_BIND() is applied to node 3, node 3 will be bound to the Adder/Subtractor by BIND_RT_FUNCTION_TO_UNIT(), producing the graph in Fig. 4.3c. Then, FIND_BINDABLE_FUNCTION() will select node 2 for the next recursion of BRANCH_ON_BIND() which will bind node 2 to the Adder/Subtractor (see Fig. 4.3d).

4.4 Algorithm Complexity

For the following discussion, let n be the number of nodes, or RT functions, in the graph and m be the number of components in the user-specified library.

The fastest case for BRANCH_ON_BIND() is when all the nodes can only map to a single RT component type. During the call to BRANCH_ON_BIND() in the first iteration of line **b**, there will only one iteration of line **c** for the first node. Subsequent nodes will be recursively bound to the same unit at line **e**. So for the first iteration of line **b**, there will be n calls to BRANCH_ON_BIND(). Following iterations of line **b**, will produce redundant solutions which will be abandoned. That is, the remaining $n-1$ iterations of line **b** will make only $n-1$ calls to BRANCH_ON_BIND(). So, for this case, the algorithm will make $2n-1$ calls to BRANCH_ON_BIND(), i.e., the algorithm runs in $\Omega(n)$ time.

The worst case for BRANCH_ON_BIND() is when each node in the graph can map to every type of RT unit but cannot share a unit with any other node. In other words, there are no edges in the graph though each node can bind to any unit. This problem can be reformulated as having n nodes which can be colored with m different colors. Obviously, there are m^n different solutions. The algorithm would find all these solutions during the first iteration of line **b** if no "pruning", abandoning search of a solution, occurs. That would mean all solutions have the same area. In such a case, there would be m^n calls of BRANCH_ON_BIND(), since each call "colors" a node. The following iterations of line **b** will produce redundant solutions which will be abandoned. That is, the remaining $n-1$ iterations of line **b** will make only $n-1$ calls to BRANCH_ON_BIND(). So theoretically, the algorithm

runs in $O(m^n + n)$ time. However, the assumptions made for this scenario are unrealistic. Intuitively, it is obvious that this worst case situation is unrealistic for two reasons:

1. All RT components do not have the same functionality, bit width, area, etc. I expect that a designer will wish to explore the possible use of a variety of RT components with varied functionality. Consequently, it is a more reasonable assumption that each RT function can map to only a small subset of the components in a library.

2. If all allocations for a model were of identical area, there would be no need to apply an algorithm such as this. Consequently, some pruning of the search is expected.

From these expectations, some assumptions can be made about the expected performance of this algorithm. First, regardless of the size of the component library, each RT function can typically only bind to a small number of components. For example, a 4-bit addition RT function can only map to the set of 4-bit adders, ALUs, etc., regardless of how many different components of different bitwidths are in the library. So for a realistic library, the worst case performance is $O(c^n + n)$ where c is a constant. In addition, while the affects of the pruning in the search are unpredictable, they have a great affect on the amount of search done. Preliminary experiments leads me to believe the average case run-time complexity of this algorithm is n^c, where c is a constant. For my experiments, the elapsed real-time for this algorithm running on an example of about size $n = 40$ was about five minutes.[1]

[1]The algorithm was written in C under Unix running on a SPARCStation 5

4.5 Summary

I have described an algorithm to generate an allocation of RT components from an existing library for a given behavioral description. The allocation of components is minimized for the area of the components. However, the purpose of this algorithm is not to generate a highly-optimized solution. There is room to improve the algorithm and provide more complex cost functions for the results. In retrospect, a different method of solution, e.g., integer programming, could produce improved results without excessive run-time requirements. But this algorithm served two key purposes in my overall synthesis research:

1. It exercised the component database model, illustrating the usefulness of the hierarchical abstraction of RT component functionality.
2. It accomplished the first step of my HLS trajectory — providing a component allocation that will be used in the next step of synthesis, operation scheduling.

Chapter 5

Operation Scheduling Algorithm

In this chapter, I describe a scheduling algorithm to reuse pre-defined components. The information in this chapter is derived from [AnDu95]. First, I will give a very general description of how the algorithm works. I will illustrate the general workings with simple examples. Then I will explain in more detail the metrics and method of the scheduling algorithm. For scheduling, I assume that an allocation of components is given. Also, I assume a delay value is specified for the maximum allowed data propagation delay through the datapath components. This delay estimates the clock period length though it does not account for delays related to registers, interconnect and wiring, or memory accesses. In my formulation of HLS, operation scheduling is performed after allocation of function units but before the allocation of registers.

It must be emphasized that my formulation synthesis is fundamentally different from previous approaches. I build upon previous assumptions about component behavior in HLS in order to consider components with complex, concurrent behavior. Consequently, this algorithm considers a greater variety of ways that operators can be mapped to component types. Figure 5.1 illustrates the various ways an example piece of behavior can be mapped to components for scheduling.

If a multiplier, adder, and comparator have been allocated to the design, both of the mappings in Figure 5.1a and Figure 5.1c would be considered for scheduling. This means that determining how a given operator can possibly map to a set of allocated components becomes a more complicated search than previous scheduling algorithms had formulated.

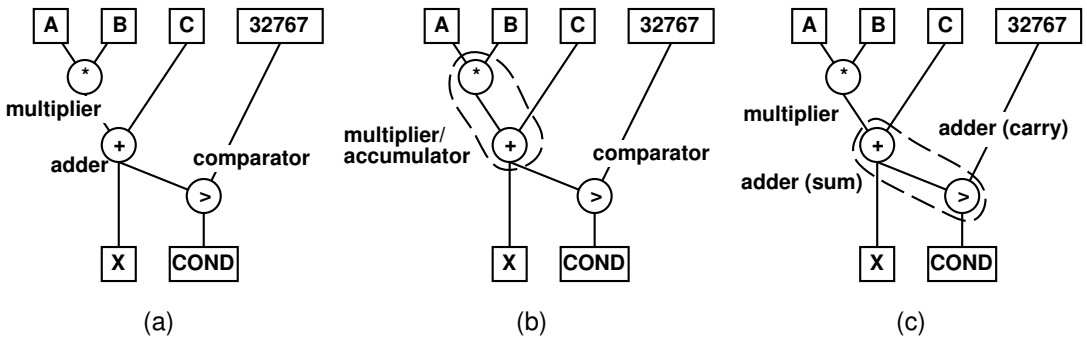


Figure 5.1: Example of alternative ways to map operators to RT units that scheduler considers.

5.1 Algorithm Overview and Examples

Obviously, there are many approaches that can be used to solve this formulation of the scheduling problem. But in analyzing the relationship between the typical control and data flow constructs in intermediate behavior representations, I realized that scheduling of control and data flow is represented by only three situations: unconditional assignments, conditional assignments, and loops. Also, I realized the bulk of the scheduling problem deals with basic blocks of assignments, while control flow constructs require little scheduling. Consequently, my approach is to first deal with the minor scheduling transformations needed for control flow constructs. These will divide the design into a set of basic blocks, each of which can

be treated as a scheduling subtask. I use a depth-first search with various search reduction heuristics to solve each subtask. So for each basic block, the algorithm:

1. estimates the critical path through the block. This involves a "quick-and-dirty" mapping of operators to components so that component input-to-output delays are used for the operators.
2. schedules all operators related to the critical path. A more thorough search of mapping of operators to components is done in this phase. The algorithm will be given a constraint on the clock period length. This constraint will be used to determine chaining and multicycling of RT unit operations.
3. schedules the off-critical path operators. The algorithm determines whether the remaining unscheduled operators can be assigned to an already scheduled RT unit (i.e., searches for concurrent operations).

I will illustrate the operation of the scheduling algorithm on three simple examples that respectively deal with unconditional assignments, conditional assignments, and loops. For these examples, three RTL components will be used: a 16-bit Adder/Subtractor, a 16-bit Multiplier, and a 16-bit Subtractor/6-function Comparator (see Table 5.1). The area and delay characteristics of these components were based on the 0.8 micron CMOS VTI library [VTI92].

Table 5.1: Components used for scheduling examples.

Component	delays	
16-bit Adder/Subtractor (DPASB001S)	input to sum	11 ns
	input to carry/overflow	7 ns
16-bit Multiplier (DPMLT010H)	input to product	25 ns
16-bit Subtractor/Comparator (DPSUB001S + DPZDT001S)	input to difference	11 ns
	input to Greater-or-Equal/Less	12 ns
	input to Less-or-Equal/Greater	14 ns
	input to Equal/Not-Equal	14 ns

5.1.1 Example 1: Unconditional Assignments

Example 1 illustrates how the scheduling algorithm estimates the critical path and schedules operations. The description for this example is shown in Figure 5.2. In this example, **COND** describes the overflow output for the addition $\mathbf{A + B}$. A 16-bit Adder/Subtractor and a 16-bit Multiplier were allocated for this example.

```
entity example1 is
  port (A: in BIT_VECTOR(15 downto 0);
        B: in BIT_VECTOR(15 downto 0);
        C: in BIT_VECTOR(15 downto 0);
        COND: out BIT;
        X: out BIT_VECTOR(15 downto 0));
end example1;

architecture BEHAVIOR of example1 is

begin

process(A,B,C)
  variable TEM1: BIT_VECTOR(15 downto 0);
begin
  TEM1 := A + B;
  COND := TEM1 > B"0111111111111111";
  X := TEM1 * C;
end process;
end BEHAVIOR;
```

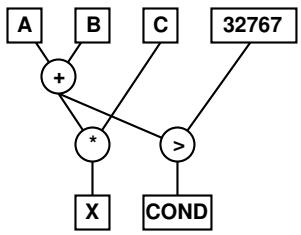


Figure 5.2: Example 1 - unconditional assignments: VHDL and data flow graph.

The first step in scheduling a set of assignments is to estimate the critical path. To do this, the algorithm calculates the *delay to output* (DTO) for data

flowing through this basic block. The operator delays are needed for this and are found by mapping each operator to an allocated component. The matching for this example is shown in Figure 5.3. Note how the match for the **overflow** overlaps with the match for the **add** and how the **overflow** function encompasses more than one operator. Once the delays for the operators are found, the DTOs are calculated by the formula:

$$\text{DTO} = \text{delay of operator} + \text{max. DTO of dependent operators}$$

where a dependent operator is an operator that uses the data output of this operator. Once the DTOs have been calculated, the critical path is found by starting at the operator with the highest DTO and proceeding to the dependent operator with the highest DTO. Once an operator that outputs to a write node is reached, the bottom of the critical path has been found.

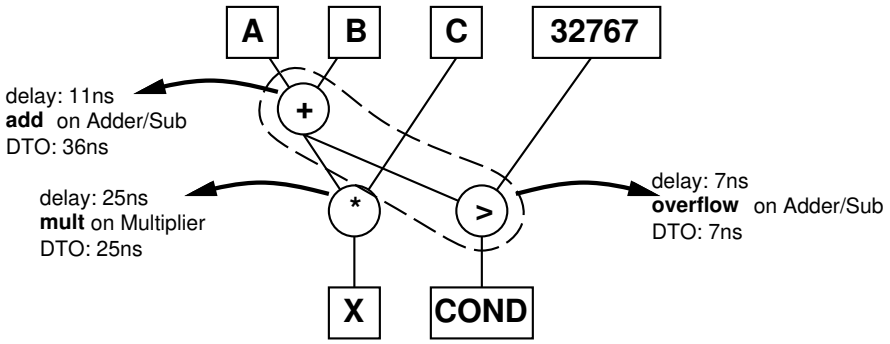


Figure 5.3: Operator mapping for calculating delay.

Now that the critical path has been found, the algorithm will try to schedule the critical path operators. At this point, it is necessary to describe the scheduling options available in the scheduling algorithm. Since delay information about RT functions is available, the scheduling algorithm makes no assumptions about the number of cycles needed to perform each operator. Consequently, the algorithm

can be configured to do various combinations of RT operation chaining and multicycling. I define operation chaining as linking the input of one RT component to the output of another so that data is propagated from one unit to the other within a single state. I define operation multicycling as allowing an RT operation to be executed on a component for more than one clock cycle. For operation multicycling, I assume that appropriate latching of inputs will be implemented, either built into the RT component or added to the design by other synthesis tools.

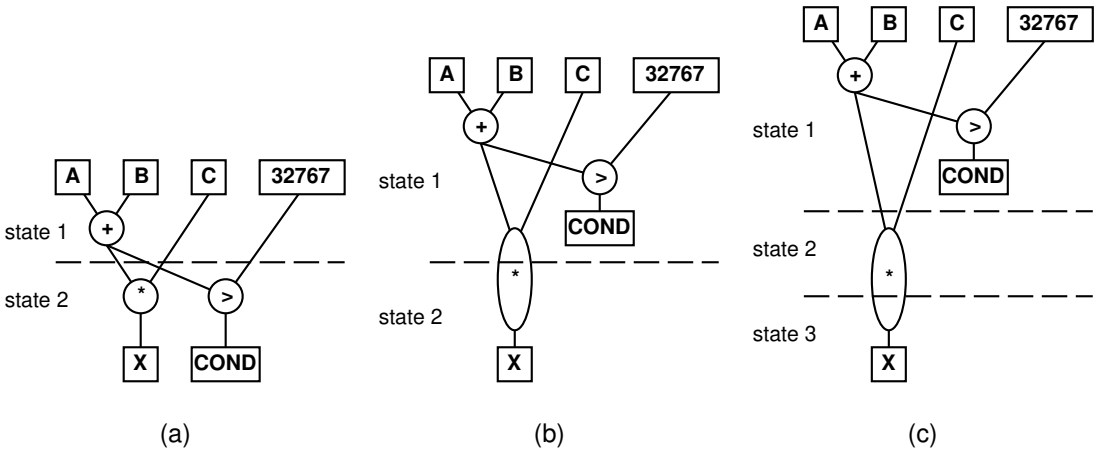


Figure 5.4: Scheduling options: (a) no operator chaining or multicycling. (b) combined operator chaining and multicycling. (c) exclusive operator chaining and multicycling.

Some of the scheduling options are illustrated in Figure 5.4. As shown in Figure 5.4b, chaining the multiplication and addition results in a propagation delay that is greater than the clock period, so the multiplication is multicycled. In *exclusive chaining/multicycling*, operations are chained only if the operations can fit within a single state. So for exclusive chaining/multicycling in Figure 5.4c, the multiplication starts execution in state 2. As Figure 5.4 illustrates, by using available delay information and not making assumptions about component delays

within the algorithm, the scheduling algorithm is able to explore a variety of schedules based on the configured options and the specified clock period. Of course, this can produce infeasible combinations. For this example, if the clock period is only 20 nanoseconds and no multicycling is allowed, there is no way to execute the multiplication that takes 25 nanoseconds. In addition to scheduling operators to components, the algorithm also binds the appropriate operator inputs and outputs to the component. This information is maintained in the RTDFG and is needed when the algorithm tries to schedule other operations to the same component in the same state (i.e., schedule concurrent functions).

Once the critical path has been scheduled, the algorithm will try to schedule the rest of the description in an As-Soon-As-Possible (ASAP) manner. For this example, I will now illustrate some of the key advantages of my models. For the Adder/Subtractor, a template for the overflow output of an addition matches the portion of the graph shown in Figure 5.3 for the **overflow** function. The scheduling algorithm is able to find this template match and query the component database to find that the allocated Adder/Subtractor can perform this template concurrently with the already scheduled **add**. To confirm this concurrency, the algorithm uses the RTDFG to check that the pin bindings for both templates would be compatible, i.e., the pins of the Adder/Subtractor will be bound to the same inputs and outputs for both functions. Once the compatibility of the pin bindings has been confirmed, the **overflow** is scheduled to the Adder/Subtractor in the same state as the **add**. This is why the comparison in Figure 5.4b and Figure 5.4c is scheduled to the same state as the addition.

5.1.2 Example 2: Conditional Assignments

```
entity example2 is
  port (A: in BIT_VECTOR(15 downto 0);
        B: in BIT_VECTOR(15 downto 0);
        C: in BIT_VECTOR(15 downto 0);
        D: out BIT_VECTOR(15 downto 0));
end example1;

architecture BEHAVIOR of example2 is

begin

process(A,B,C)
  variable TEM1, TEM2: BIT_VECTOR(15 downto 0);
  variable COND: BIT;
begin
  COND := C > 16;
  TEM1:= A * B;
  if COND then
    TEM2 := TEM1 – C;
  else
    TEM2 := A – B;
  end if;
  D := TEM2;
end process;
end BEHAVIOR;
```

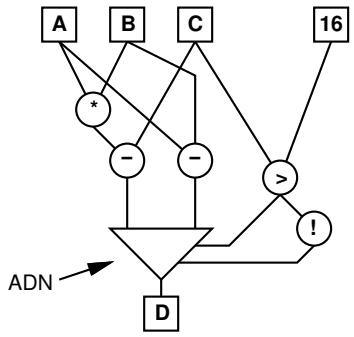


Figure 5.5: Example 2 - conditional assignments: VHDL and equivalent ADD.

This second example will illustrate how the algorithm handles conditional assignments. The description of this example is shown in Figure 5.5. For this example, a multiplier and a subtractor/comparator were allocated. At this point, discussion of some control flow characteristics of the internal representation of the input description is necessary. Conditional branches generated from "if-then-else" statements represent execution paths through the description that are performed mutually exclusive to each other. However, for synthesis, the mutually exclusiveness of operators in these paths is not apparent unless some form of global condition vectors for each operator is available. Consequently, for scheduling without condition vectors, such conditional branches are usually translated into branching state transitions (see Figure 5.6). If condition vectors such as those described in [JCG94] are available, such state transitions are not necessary. The condition vectors in

[JCG94] are useful not only for determining the mutual exclusiveness of operators but they can also relate the *conditional dependencies* between operators, i.e., which operators produce the condition vector of a particular operator. By having the scheduling algorithm analyze the data dependencies and the conditional dependencies, the scheduling algorithm can treat a description that consist of only unconditional or conditional assignments as a basic block.

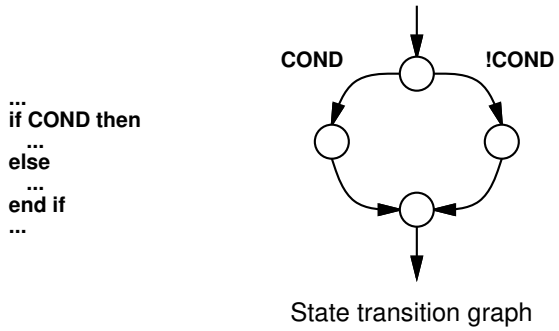


Figure 5.6: CDFG conditional branch translated to state transitions.

For this example, the input description is represented by an Assignment Decision Diagram (ADD) [ChGa92] (Figure 5.7). For conditional assignments, the ADD has an Assignment Decision Node (ADN) which serves a function similar to the *select* in the Value-Trace representation [McFar78]. An ADN has data inputs, each of which has an associated condition input. Each input can be an ADD graph for an expression. Each data input of the ADN must be the same bit-width as the ADN's output. Each condition input must evaluate to a Boolean value and must be mutually exclusive to the other condition inputs. That is, only one condition input should true at any time. The value of a data input is transferred to the output of an ADN when the appropriate condition input becomes true.

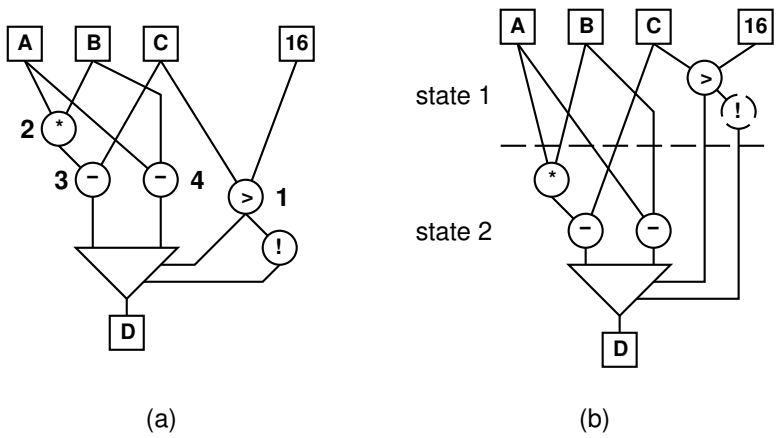


Figure 5.7: Scheduling conditions: (a) order operators are scheduled. (b) schedule for example with allocation of a multiplier and subtractor/comparator.

The scheduling algorithm makes certain assumptions to deal with conditional assignments. Conditional assignments do not affect the way delays or DTOs are calculated. But for determining the critical path, only one data or condition input to an ADN will be considered critical. Also, for scheduling the critical path, the algorithm needs to consider all operators related to the critical path, i.e., operators that have data or conditional dependencies with operators on the critical path. For this example, scheduling the critical path means not only scheduling the multiplication and a subtraction but also the "greater than" comparison, since it determines the assignment condition of the critical path. To ensure that the corresponding condition input is ready when a data input arrives at a ADN, the algorithm schedules the condition vector operators first, then schedules the data input operator to a state after the condition vector operators are executed. For this example, this means it first tries to schedule the "greater than," then the multiplication, then the subtraction, and the subtraction must be scheduled to a state after the "greater than" is performed. Figure 5.7a shows the order in which

the operators are scheduled. Likewise for the other operators in the description, the condition vector operators are scheduled first. So to complete the schedule for this example, the "not" operator should be scheduled before subtraction **4**. However, I assume that some basic logic functions, such as 1-bit AND, OR, NOT, are better implemented as random logic or can be incorporated into the control logic. Consequently, for my experiments and examples, the algorithm does not schedule them because I assume they will be implemented as random logic or will be part of the control unit. When the algorithm attempts to schedule operators to the allocated components, it not only checks for data and conditional dependencies but also for mutual exclusiveness of condition vectors. In this example, the algorithm schedules the two subtractions to the same state on the same component. Figure 5.7b shows a complete schedule for this example. Note, that regardless of the allowed clock period, at least two states are required for this example because of the assumptions I make for scheduling conditions.

5.1.3 Example 3: Conditional Loops

This example illustrates the scheduling of a loop. The description of this example is shown in Figure 5.8. For conditional loops, states must be introduced into the description. For loop constructs like "while" loops, I concluded that three separate blocks of operations were needed: one before the loop, one for the body of the loop, and one after exiting the loop. This would require state transitions like those shown in Figure 5.9, assuming a Moore finite state machine model. For scheduling, each of these states can be treated as a basic block, So, scheduling a description with loops can be done by first introducing the necessary

```
entity example3 is
  port (A: in BIT_VECTOR(15 downto 0);
        B: in BIT_VECTOR(15 downto 0);
        X: out BIT_VECTOR(15 downto 0));
end example1;

architecture BEHAVIOR of example3 is

begin

process(A,B)
  variable TEM1, TEM2: BIT_VECTOR(15 downto 0);
begin
  TEM1:= A–B;
  TEM2 := 0;
  while (TEM1 > 0) loop
    TEM1 := A–B;
    TEM2 := TEM2 + 1;
  end loop;
  end if;
  X := TEM2;
end process;
end BEHAVIOR;
```

Figure 5.8: Example 3 - conditional loop: VHDL.

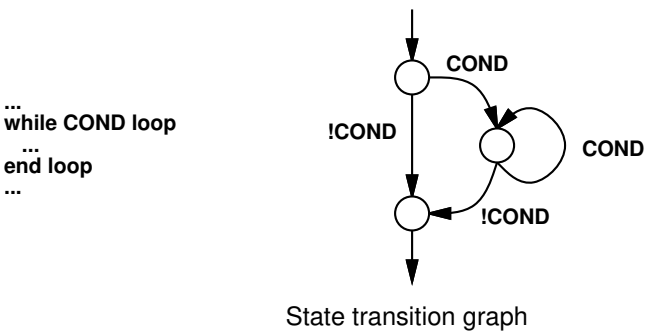


Figure 5.9: Conditional loop translated to state transitions.

state transitions then scheduling the individual blocks. Thus, a description with loops simply becomes a set of scheduling tasks similar to the two previous examples, and the scheduler will "split" each state into a set of states. Figure 5.10 illustrates the scheduling of this example for an allocation of one Adder/Subtractor and one Subtractor/Comparator.

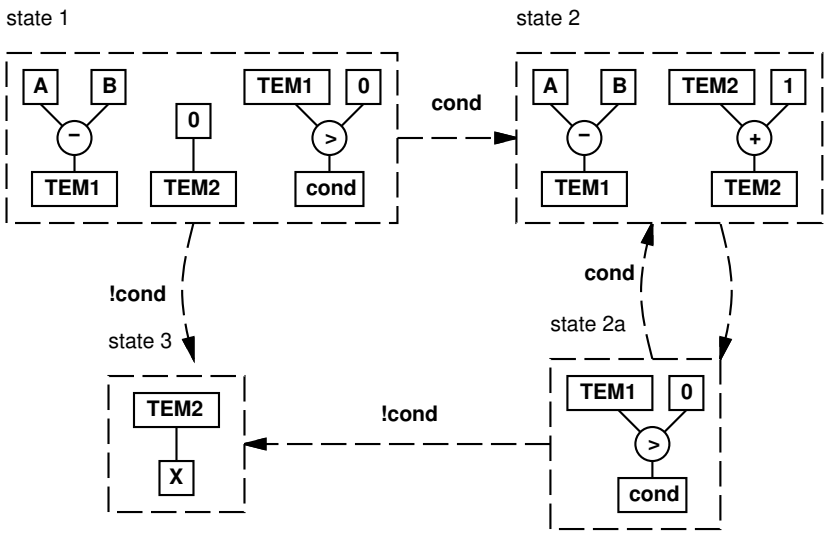


Figure 5.10: Schedule for Example 3 with one Adder/Subtractor and one Subtractor/Comparator.

5.2 Basic Block Scheduling In Detail

Now that I have explained how control-flow constructs are dealt with by the scheduling algorithm, I will go into a more detailed discussion of how the algorithm schedules individual basic blocks. I will describe in detail the three phases of basic block scheduling: estimation of the critical path, scheduling of operators related to the critical path, and scheduling of the remaining operators.

5.2.1 Estimation of Critical Path Delay

For identifying the critical path through a basic block, the *delay to output* (DTO) is calculated for each operator. To do this, the algorithm first must find the input-to-output delay of each operator. This involves mapping the operators to the RT components allocated from the library. For this phase, the algorithm does a "quick and dirty" mapping, trying to map each operator directly to an allocated component. An operator may map to more than one type of component, so the fastest is selected, i.e., the component with the minimum input-to-output delay. But there is a side-effect of this approach that could lead to a false critical path: since only individual operators are considered, the best mapping of operators to components may not be used. For example, if a multiply-accumulator has been allocated and the description has multiplication outputting to an addition (e.g., **A*B+C**), both operators can be mapped to a single RT operation (a *multiply-accumulate*), but for estimating the DTOs, the algorithm will try to map each to separate RT operations (*multiply* and *addition*). The delays for a *multiply* plus an *addition* could be greater than a *multiply-accumulate*. For this first pass, the scheduling algorithm is simply checking if there exists at least one allocated component that can execute each part of the description. For each operator, the algorithm finds the delay by locating which allocated components can perform this operator and checking the pin-to-pin delays.

Once the delays for the operators are found, the DTOs are calculated by propagating the delays to their inputs. For each operator, this formula is used:

$$\text{DTO} = \text{delay of operator} + \text{max. DTO of dependent operator}$$

A dependent operator is one that uses the data output of this operator match. For example, in **A*B+C**, the addition is dependent on the multiplication. Once the DTOs have been calculated, the critical path is found by starting at the operator with the highest DTO, then proceeding to the dependent operator with the highest DTO. Once an operator that outputs to a write node is reached, the bottom of the critical path has been found. During the entire scheduling algorithm, the set of operators with calculated delays is maintained as the set of all operators that need to be scheduled. When operators are scheduled, they are marked but remain in the set.

Note, as previously mentioned, I assume that some basic logic functions, such as 1-bit AND, OR, NOT, are better implemented as random logic or can be incorporated into the control logic. During calculation of delays, when a 1-bit AND, OR, NOT is encountered that cannot be mapped to an allocated component, it is ignored assuming that subsequent synthesis will implement it as random logic or as part of the control logic.

5.2.2 Scheduling the Critical Path

The second phase of basic block scheduling will schedule the operations related to the critical path. These include the operators that have conditional or data dependencies with the critical path operators. This portion of the algorithm will assign the operations to a sequence of states, starting with the operators at the top of the critical path (the ones receiving data that is coming into this basic block) and working down to the output operators. In the process, the algorithm determines if multicycling or chaining of the operations is possible or needed.

The scheduling of the critical path is described in procedure Sched_CP() in Figure 5.11. To handle condition and data dependencies between operators, the algorithm builds a "critical path stack" prior to calling this procedure. For each critical path operator, operators with data or conditional dependencies are placed above it on the stack. During scheduling of the critical path, the procedure pops operators off the stack to ensure that the dependencies for an operator have already been scheduled. The internal looping and recursive procedure calls of Sched_CP() implement a "branch and bound" search, trying various schedules and stopping on alternatives that become longer than the shortest complete schedule found so far. During scheduling of the critical path, the algorithm does not try to evaluate all possible combinations of template matches for the entire critical path. Instead, it evaluates only a portion of the critical path at any time. In effect, the algorithm schedules a small "window" of the critical path, sliding the window down the critical path at each step of the scheduling process. In Figure 5.11, the size of this window is the variable *depth*.

The procedure begins by finding an unscheduled critical path operator on the stack (line 1). If the stack has been exhausted, all operators related to the critical path have been scheduled and this procedure exits. Otherwise, the algorithm first tries to add this operator to the existing schedule at line 3, i.e., schedule it to a state before the current end of the schedule. If that is possible it goes on to the next operator at line 4. Regardless of whether it can schedule the operator to a previous state, the procedure will still explore the possibility of scheduling the operator to the end of the current schedule, i.e. lines 5 and on will still be executed whether line 3 fails or not. At line 6, the procedure tries different size "windows" of the critical path. The current critical path operator to be scheduled is always

```
Sched_CP()
1   Pop an unscheduled node from the stack
2   if there is an unscheduled operator on the critical path
3     if can ASAP schedule node to existing schedule.
4       Sched_CP() next node on stack
5     /* Try other schedules of node */
6     for depth = max. downto 0 do
7       find covers of node of size depth
8       for each cover of node do
9         for each RT function cover can bind to do
10          for each RT unit RT Function can bind to do
11            if RT function needs to be multicycled,
12              add states and schedule
13            if RT function can be chained, schedule.
14            if schedule not shorter than best schedule found,
15              cancel this search.
16            if could not schedule RT function,
17              if could not schedule RT function to an empty state,
18                cancel this search.
19              else maybe no RT units were available,
20                add a state to the schedule and retry
21            else could schedule RT function
22              Sched_CP() next node on stack
```

Figure 5.11: Scheduling of critical path related operators

at the top of this "window" and each iteration of line 6 shrinks the window. All possible template matches in this window that "cover" the critical path operator are examined at lines 7 and 8. For each template match, the algorithm tries to schedule the template to all possible RT functions and RT units (lines 9 and 10). If the selected function cannot be performed within the current time step, it is scheduled to multiple time steps at lines 11 and 12, i.e., scheduled as a multicycle operation. If the function can be performed within the time left for the current step, it will be scheduled to this step, i.e., it will be chained to other operators at line 13. Chaining is possible because the algorithm maintains the amount of delay the needed for the operations already scheduled to the current step. I also assume an additive delay model (i.e., the delay to perform two operations is equal to the sum of each operation's delay). Of course, multicycling and chaining is only done when these options are allowed. If the function was successfully scheduled, the procedure goes on to the next operator at line 22. If the function was not scheduled, the algorithm will add a state (line 20) and attempt to schedule the function again. If the algorithm fails to schedule an operator when given an empty state, then the algorithm returns a failure to find a feasible schedule for the critical path (lines 16-18). This can be due to a combination of no multicycling and the allowed maximum propagation delay being too short for an operation.

5.2.3 Scheduling Remaining Operators

After the critical path is scheduled, the algorithm will schedule the remaining operators. For these operations, the algorithm will try to share functional units. That is, the algorithm will schedule an operation to a state when a unit is free or

when other operations scheduled to that unit have mutually exclusive condition vectors. The algorithm will also try to schedule operations concurrently, i.e., determine if an operator can be scheduled to a function that can execute concurrently with other functions on an RT unit.

The scheduling of remaining operators is described in Figure 5.12. In procedure ASAP_nCP(), the algorithm will first attempt to schedule the operators ASAP, without additional time steps. If this fails, the algorithm will schedule the operators ASAP with time steps added to the schedule. During the ASAP scheduling of an operator condition and data dependencies are checked as well as exploration of concurrent RT functions.

Lines 1 and 2 in ASAP_nCP() check if the current partial schedule for the critical path is shorter than the best solution found so far. If not, there is no point in finishing this schedule. Line 3 will call ASAP_schedule() to try to schedule the remaining operators into the existing schedule. If that was not possible at line 4, the loop through lines 6 to 14 will alternately add states to the end and beginning of the schedule to try to make room for the remaining operators. Lines 8 and 13 check that the current partial schedule created in the loop is shorter than the best solution found so far. As line 5 indicates, the procedure will add states and attempt to schedule operators until a schedule is found or the total number of states added is greater than or equal to 3 times the number of operators to be scheduled. This limit on the number states is arbitrary and was derived on the assumption that efficient designs will not have a large number of operators multicycled for 4 states or more.

```
ASAP_nCP()
1    if not shorter than best schedule found,
2       cancel this search.
3    ASAP_schedule()
4    if failed
5       while not successful or num. added states < 3× num. unscheduled ops. do
6          add state to end of schedule
7          ASAP_schedule()
8          if not shorter than best schedule found,
9             cancel this search.
10         if successful exit while loop.
11         add state to front of schedule
12         ASAP_schedule()
13         if not shorter than best schedule found,
14            cancel this search.
15   if a schedule found
16      if shorter than best schedule found,
17         save as best schedule found

ASAP_schedule():
18   do
19      find unscheduled operator with highest DTO.
20      check if for unscheduled op with data/condition dependency
21      ASAP schedule operator to available units
22      if fails return failed.
23   while there is an unscheduled node.
```

Figure 5.12: Scheduling of remaining operators

The procedure ASAP_schedule() attempts the actual scheduling of each of the remaining operators. Since this procedure tries to schedule operations as early as possible in the schedule, it makes sense to try to first schedule the operators at the top of long expressions. Consequently, at line 19, ASAP_schedule() looks for the unscheduled operator with the highest DTO. But before any operator can be scheduled, the operators that it is dependent on, for its data or condition, must be scheduled first (line 20). If the data and condition dependencies are already schedule, then the procedure will try to insert the operator into the schedule at line 21. It is here that the algorithm will search for concurrent functions and try to map the operator to them. If for any reason the operator cannot be scheduled, line 22 will cause the procedure to quit and return that a feasible schedule could not be found.

5.2.4 Search Reduction

The search space for this scheduling algorithm can be envisioned as a search graph that is a tree. The first branches from the root determine which critical path operators are scheduled to RT units for the first state of the schedule. Subsequent branches add more operators and states to the schedule. Obviously, the size of the search space grows proportional to the number of operators, and the number of different ways each operator can be mapped to an allocated RT unit.

The scheduling algorithm conducts a depth-first search of this tree. It prunes off searches that will obviously not find a better solution than the current solution. That is, it will abandon a schedule that is not shorter than the current complete schedule already found. However, this still could leave a very large space to search.

From experiments I have conducted, I find that the final solution returned is most often found very early in the search. Consequently, I have two additional strategies to limit the search space further: progressive backtracking and a maximum search limit.

Backtracking is an inherent part of depth-first search where the search "backtracks" up one level of the search tree when searching of a branch completes or is pruned. With only one level of backtracking, each schedule the search explores is only slightly different from the previous schedules that have been searched. This is desirable early in the search process when few or no solutions have been found. But as the search progresses, I want to look at schedules that differ greatly from previous schedules in order to explore a wider variety of schedules faster. Consequently, in the algorithm, I progressively increase the number of levels of backtracking as the search progresses. This is done by maintaining the number of attempts to schedule operators, i.e., the number of edges the search has traversed. Whenever a search of a branch fails, the amount of backtracking is calculated.

$$\text{num. levels to backtrack} \sim \frac{\text{num. attempts to schedule operators}}{\text{num. operators in design}}$$

Further, since the best solution found is usually discovered early in the search of the entire tree, I allow setting a limit on the number of attempts to schedule operators. Potentially, the limit could be set so that no solution is found but for my experiments the limit was set so that at least several dozen schedules for very complex designs would be searched. This search limit was implemented with the idea that the algorithm would be used in an interactive environment where deriving a "good" solution in a short amount of time was desirable.

5.3 Summary

The scheduling algorithm I have described here fully illustrates the use of all the structures and representations I have defined for synthesis to reuse RT components. It also illustrates some of the issues in dealing with complex, concurrent RT functionality that is not considered in other HLS algorithms and systems. Like the allocation algorithm in the previous chapter, this algorithm is an initial attempt at synthesis for component reuse and there are many ways this algorithm could be improved. Yet, this algorithm still demonstrates the benefits of my component reuse methodology. In the next chapter, I describe how this algorithm is used in the synthesis system implemented for my experiments, and present the experiments and results that prove the utility of component reuse in HLS.

Chapter 6

System Implementation and Experiments

In this chapter, I describe the system that served as the testing environment for my experiments, and describe the experiments I conducted. I describe how the various parts of my work were implemented and interfaced with the Interactive Synthesis Environment (ISE) [Had95]. I also describe the individual examples that were used for the experiments, present the results I obtained with these examples, and discuss the results. This system is used to demonstrate the benefits of improved combinatorial datapath component reuse, enabled by using a HLS methodology based on improved integration of HLS algorithms with existing RT libraries.

6.1 Implementation & Experiment Design Flow

All algorithms used for my experiments, including the allocation and scheduling algorithms described in the previous chapters, were implemented in C on Sun SPARCstations in a Unix environment. The implementation used the underlying data structures available in the Interactive Synthesis Environment (ISE). However,

additional data structures were implemented to construct the hierarchical component database, and the behavior template parse tree.

With all needed data structures in place, one of the first tasks in preparing for my experiments was constructing a set of RT components. I used the graphical interface of ISE to capture the RT components used for my experiments, including the area and delay of each component as well as the behavioral templates for the functions of each component. Figure 6.1 shows the graphical interface in ISE for capturing component information. The component information for this interface can be saved to and loaded from an external file.

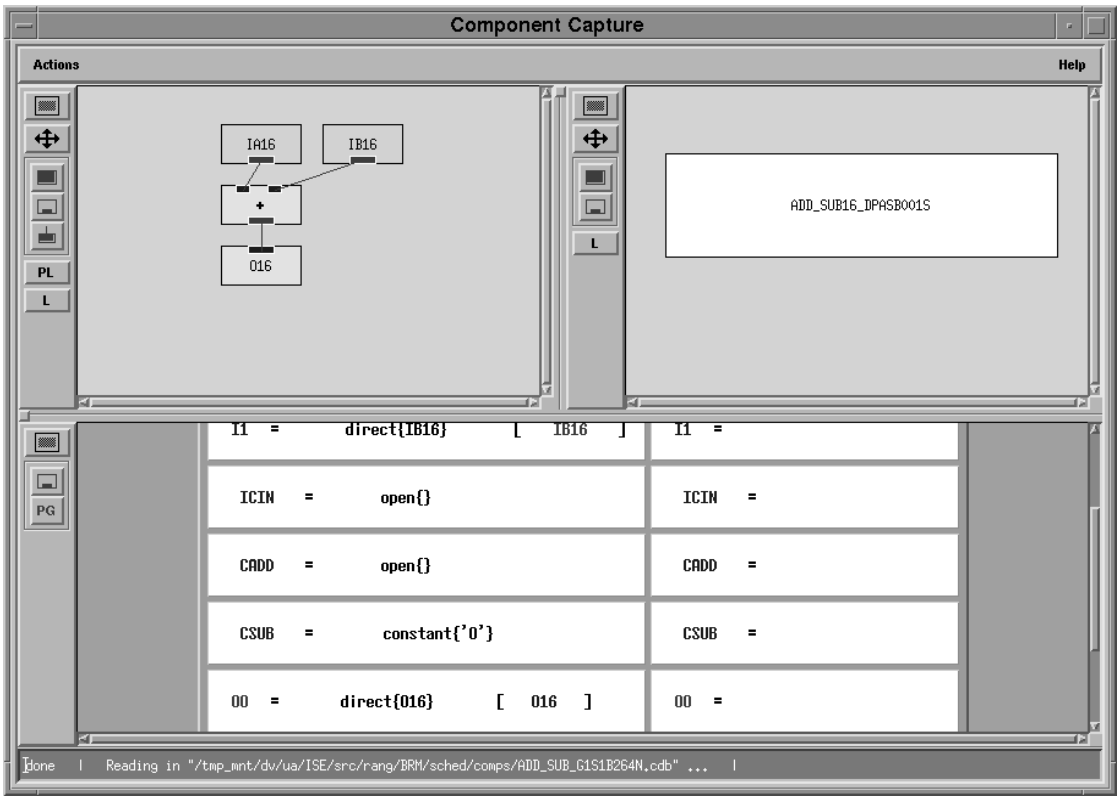


Figure 6.1: ISE component capture interface.

The upper right of the display is primarily used to capture physical information about the component. To describe an RT component in ISE, the designer can specify the height and width of the component in microns. This yields a rectangular box representing the physical shape of the component. A set of ports for the component can be specified. Ports can be classified as input, output, input/output, or control. A port's bitwidth and placement on the edge of the component box can also be specified. Delay values between pairs of ports can be specified. The value can be in picoseconds, nanoseconds, or microseconds. In this way, this portion of the ISE interface captures the area and delay information for a component.

The upper left portion of the display in Figure 6.1 displays behavioral information about a component. Each component in ISE can have an associated set of *behavioral templates*. Each template is an Assignment Decision Diagram (ADD) describing, in a data-flow like manner, a function the component can perform. A template typically is an interconnected set of operations, read nodes, and a single write node. The table in the lower part of Figure 6.1 describes how a template is mapped onto the component. Usually, read nodes are mapped to a component's input ports, and a write node is mapped to a component's output port. However, simple logic operations can also be involved in the port mappings (e.g. ANDing two values to obtain the input value to a port) or ports can be set to a binary constant value. The ADD for the behavioral template is input from an external file. The port mapping table is entered manually through the ISE interface.

For my model of component behavior, I am interested in describing concurrent functions of an RT component. For a template, I interpreted the constant values assigned to the control ports of the component as the mode of operation for

that component. Consequently, for a given component, templates that have the same values on the control ports are interpreted to be concurrent functions. This information is maintained in the component data base.

Using the ISE interface, I captured the components in the library for my experiments, one file for each component. I implemented a separate tool to combine the component information in each of these files into a single file that becomes the hierarchical component database described in Chapter 3. Addition or deletion of components requires that this database file be rebuilt with each change. This same tool also generated a file for the template parse tree for the components in the database file.

For each experimental example, I again used ISE to capture the unscheduled design descriptions. VHDL models of the experimental examples were available, but VHDL input into ISE had not yet been implemented at the time of the original experiments. However, ISE did have facilities to output VHDL simulation models. Consequently, for the experimental examples that had accompanying VHDL simulation test vectors, I generated a simulation model from ISE and tested it with the test vectors to confirm the functionality of the ISE models. Once the ISE model was entered, I used my allocation algorithm to suggest possible datapath component allocations. Using the suggested possible allocations, I then used the ISE interface to add a specific allocation of component instances to the design. I created multiple versions of each example, each with a different allocation of components. After the component allocation had been added to each example, I used my scheduling algorithm with various constraints to produce a variety of schedules for each design. These constraints were used to emulate the assumptions about

the RT components which were discussed in Chapter 2. Three sets of assumptions were used:

1. simple behavior mapping: each HDL operator was mapped to a component assuming each operation would require 1 clock cycle to complete.
2. simple behavior mapping with operator chaining and multicycling: each HDL operator was mapped to a component; component operation delays were used to determine whether a component operation needed to be multicycled, or whether multiple component operations could be chained together in a single clock cycle.
3. complex behavior mapping with operation chaining and multicycling: a set of HDL operators could be mapped to a component; component operation delays were used to determine whether a component operation needed to be multicycled, or whether multiple component operations could be chained together in a single clock cycle; and component behavior was checked to determine if component operations could be executed concurrently.

Assumptions 1 and 2 reflect the current state of HLS while assumption 3 describes my model. Schedules for each example were generated with various limits on the clock period length.[1] During scheduling, the binding of HDL operators to RT component instances is also done. However, due to some difficulties in ISE, binding of multicycled operations is not incorporated. With a complete schedule, not only are operators assigned to states, but temporary variables have been inserted, and reads and writes of variables have been assigned to states in a manner that preserves the originally functionality of the description.

[1] As explained in Chapter 5, this value does not include delay associated with memory/register access or interconnect units.

To derive a complete RT net list for my experiments, two additional tasks needed to be done: allocation and binding of registers, and allocation and binding of multiplexors. Allocation of registers was done using a simple greedy algorithm that essentially merged variables with disjoint data lifetimes and then assigned each resulting variable to a dedicated register. Allocation and binding of multiplexors was done using the interconnection binding tools already available in ISE. This tool resolved any component input ports that had multiple connections by inserting appropriately sized multiplexors before the port. The tool also generated the appropriate conditions for the select lines of the multiplexors.

6.2 Experiments

The purpose of the experiments was to compare the results of using different assumptions about RT components in HLS algorithms. The set of components used for the experiments included component types common to many RT libraries, e.g. adder/subtractors, shifters, etc. But the set also included unusual or custom components such as 0-detectors and subtractor/comparators. The delay and area characteristics of the datapath function units were based on the 0.8 micron CMOS VTI library [VTI92]. The area of registers and multiplexors were also based on this library.

The examples used for the experiments were descriptions of an IEEE standard 32-bit floating point addition/subtraction algorithm [PaDu95], a Fifth Order Elliptical Wave Filter [PaBu87], an integer divider algorithm [ErLa85], a greatest common divisor algorithm [BrBr88], and an experimental example, shown in

Figure 6.2. The results that follow were obtained from these example designs, using the pre-defined set of components, and applying the various assumptions about RT component functionality, described in the previous section, to the synthesis methodology. As mentioned in the previous section, three sets of assumptions were used for the selection of component types for allocation, and the scheduling of operators to component instances. In the component allocation tool, the assumptions resulted in two different types of allocations: allocations that mapped each HDL operator to an operation of a function unit, and allocations that could map a set of HDL operators to an operation of a function unit. The allocations derived for each of the experimental examples is shown in Table 6.1. Using the component types suggested by the allocation tool, versions of each example were created, each with a different allocation of components as shown in Table 6.2. It should be noted that most logic gates are not considered in this synthesis process because the methodology is focusing on reuse of RT function units.

As stated in the previous section, schedules for each allocated version were generated with various limits on the clock period length. For the allocated versions of the floating point addition/subtraction algorithm, the integer divider algorithm, the greatest common divisor algorithm, and the experimental example, the clock period was ranged from 10 to 50 nanoseconds at 5 nanosecond increments, i.e., 10, 15, 20, ..., 45, 50. For the Elliptical Filter, the clock period was ranged from 10 to 100 nanoseconds at 1 nanosecond increments to examine the effects of the heuristic-based approach on the consistency of the resulting designs. The three sets of assumptions used for scheduling produced three different sets of schedules for some of the allocated designs. For the graphs shown in the rest of this chapter, Table 6.2 is referred to for the allocation of components.

```
entity EXAMPLE is
  port (x: in BIT_VECTOR(7 downt 0);
        f: in BIT_VECTOR(2 downto 0);
        parity_in: in BIT;
        enable, odd: inBIT;
        done, error, parity_out: out BIT;
        r: out BIT_VECTOR(7 downto 0)
  );
end EXAMPLE

architecture BEHAVIOR of EXAMPLE is
begin
  process (enable)
    variable dataA: BIT_VECTOR(7 downto 0);
    variable err, parity, par oddP: BIT;
  begin
    if (enable = '0') then
      done <= '0';
      error <= '0';
    else
      -- initialize, load operands
      parity := parity_in;
      oddP := odd;
      err := '0';
      dataA := x;

      -- check parity (generate even parity bit)
      par := dataA(7) xor dataA(6) xor dataA(5) xor
        dataA(4) xor dataA(3) xor dataA(2) xor
        dataA(1) xor dataA(0);

      if (oddP = '1') then
        err := (parity = par);
      else
        err := (parity = not(par));
      end if;

      if (err = '0') then
        -- do selected function
        if (f(1 downto 0) = "00") then
          dataA := dataA + "00000001";
          err := (('0'&dataA) > "001111111");
        end if;

        if (f(1 downto 0) = "01") then
          dataA := dataA – "00000001";
          err := (('1'&dataA) < "110000000");
        end if;

        if (f(1 downto 0) = "10") then
          -- rotate up
          dataA := dataA(6 downto 0) & dataA(7);
        end if;

        if (f(1 downto 0) = "11") then
          -- rotate down
          dataA := data(0) & dataA(7 downto 1);
        end if;

        if (f(2) = '1') then
          -- (re)generate parity for result
          par := dataA(7) xor dataA(6) xor dataA(5) xor
            dataA(4) xor dataA(3) xor dataA(2) xor
            dataA(1) xor dataA(0);
          parity : = (oddP xor par);
        end if;
      end if;

      parity_out <= parity;
      r <= dataA;
      error <= err;
      done <= '1';
    end if;
  end process;
end BEHAVIOR;
```

Figure 6.2: VHDL description of experimental example.

Table 6.1: Allocations of component types for experiments.

design example	assumption	component types	design example	assumption	component types
integer divider	complex	17–bit adder/subtractor 17–bit shifter 16–bit shifter 16–bit 0–detector 4–bit inc/dec 4–bit comparator	FP adder	complex	27–bit adder/subtractor 27–bit shifter 27–bit 0–detector 23–bit sub/comparator 23–bit 0–detector 8–bit adder/subtractor 8–bit inc/dec 8–bit sub/comparator 8–bit comparator
		17–bit adder/subtractor 17–bit shifter 16–bit shifter 16–bit 0–detector 4–bit adder/subtractor 4–bit comparator			27–bit adder/subtractor 27–bit shifter 27–bit 0–detector 23–bit sub/comparator 23–bit comparator 8–bit adder/subtractor 8–bit inc/dec 8–bit sub/comparator
		17–bit adder/subtractor 17–bit shifter 16–bit shifter 16–bit comparator 4–bit adder/subtractor 4–bit comparator			
	simple	17–bit adder/subtractor 17–bit shifter 16–bit shifter 16–bit comparator 4–bit adder/subtractor 4–bit comparator			27–bit adder/subtractor 27–bit shifter 27–bit 0–detector 23–bit sub/comparator 23–bit comparator 23–bit 0–detector 8–bit adder/subtractor 8–bit sub/comparator
example	complex	8–bit inc/dec 8–bit shifter XOR gate		simple	28–bit adder/subtractor 28–bit comparator 27–bit adder/subtractor 27–bit shifter 27–bit comparator 24–bit comparator 23–bit comparator 9–bit comparator 8–bit adder/subtractor 8–bit comparator
		8–bit inc/dec 8–bit shifter 8–bit parity generator			
		8–bit inc/dec 8–bit adder/subtractor 8–bit shifter XOR gate			
		8–bit inc/dec 8–bit adder/subtractor 8–bit shifter 8–bit parity generator	elliptic filter	complex	16–bit multiplier 16–bit adder/subtractor
	simple	9–bit adder/subtractor 8–bit adder/subtractor 8–bit shifter 9–bit comparator XOR gate			16–bit multiplier 16–bit multiply/accumulator
GCD	complex	16–bit sub/comparator			16–bit multiplier 16–bit adder/subtractor 16–bit multiply/accumulator
		16–bit adder/subtractor 16–bit comparator			
	simple	16–bit adder/subtractor 16–bit comparator		simple	16–bit multiplier 16–bit adder/subtractor

Complex assumption: a set of operators can map to a function unit operation.
Simple assumption: each operator maps to a function unit operation.

Table 6.2: Allocations of component instances for experiments.

design example		component allocation	design example		component allocation
integer divider	**1**	1 17–bit adder/subtractor 1 17–bit shifter 1 16–bit shifter 1 16–bit 0–detector 1 4–bit inc/dec 1 4–bit comparator	FP adder	**1**	1 27–bit adder/subtractor 1 27–bit shifter 1 27–bit 0–detector 1 23–bit sub/comparator 1 23–bit 0–detector 1 8–bit adder/subtractor 1 8–bit inc/dec 1 8–bit sub/comparator 1 8–bit comparator
	2	1 17–bit adder/subtractor 1 17–bit shifter 1 16–bit shifter 1 16–bit 0–detector 1 4–bit adder/subtractor 1 4–bit comparator		**2**	1 27–bit adder/subtractor 1 27–bit shifter 1 27–bit 0–detector 1 23–bit sub/comparator 1 23–bit comparator 1 8–bit adder/subtractor 1 8–bit inc/dec 1 8–bit sub/comparator
	3	1 17–bit adder/subtractor 1 17–bit shifter 1 16–bit shifter 1 16–bit comparator 1 4–bit adder/subtractor 1 4–bit comparator		**3**	1 27–bit adder/subtractor 1 27–bit shifter 1 27–bit 0–detector 1 23–bit sub/comparator 1 23–bit comparator 1 23–bit 0–detector 1 8–bit adder/subtractor 1 8–bit sub/comparator
example	**1**	1 8–bit inc/dec 1 8–bit shifter 7 XOR gate		**4**	1 28–bit adder/subtractor 1 28–bit comparator 1 27–bit adder/subtractor 1 27–bit shifter 1 27–bit comparator 1 24–bit comparator 1 23–bit comparator 1 9–bit comparator 1 8–bit adder/subtractor 1 8–bit comparator
	2	1 8–bit inc/dec 1 8–bit shifter 1 8–bit parity generator	elliptic filter	**1**	1 16–bit multiplier 1 16–bit adder/subtractor
	3	1 8–bit inc/dec 1 8–bit adder/subtractor 1 8–bit shifter 7 XOR gate		**2**	2 16–bit multiplier 2 16–bit adder/subtractor
	4	1 8–bit inc/dec 1 8–bit adder/subtractor 1 8–bit shifter 1 8–bit parity generator		**3**	1 16–bit multiplier 1 16–bit multiply/accumulator
	5	1 9–bit adder/subtractor 1 8–bit adder/subtractor 1 8–bit shifter 1 9–bit comparator 7 XOR gate		**4**	2 16–bit multiplier 2 16–bit multiply/accumulator
GCD	**1**	1 16–bit sub/comparator		**5**	1 16–bit multiplier 1 16–bit adder/subtractor 1 16–bit multiply/accumulator
	2	1 16–bit adder/subtractor 1 16–bit comparator		**6**	2 16–bit multiplier 2 16–bit adder/subtractor 2 16–bit multiply/accumulator

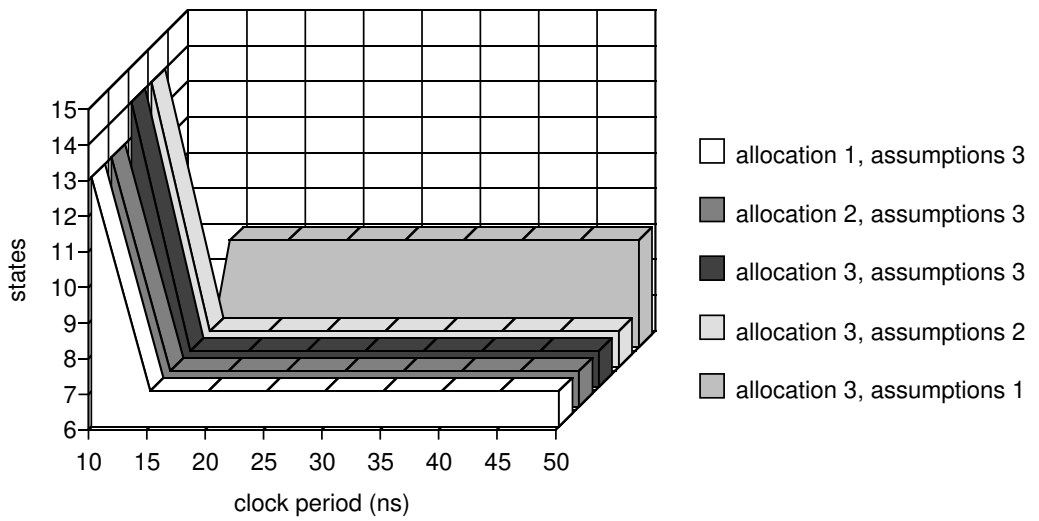


Figure 6.3: Scheduling results for Integer Divider.

Figures 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, and 6.9 show the results of scheduling the allocations for the experimental designs. As these graphs illustrate, in many cases, an allocation could not be scheduled with the simpler assumptions because the allocation uses components that a tool based on simpler assumptions would not expect or know how to use (e.g., 0-detector, subtractor/comparator, multiply-accumulator). For example, this is why Figure 6.3 has no schedules of the integer division algorithm for allocation 1 using assumption sets 1 or 2. In some cases, shorter schedules were generated by the complex set of assumptions (assumption set 3). In Figure 6.8, schedules with fewer states were generated for allocation 4 using assumption set 3 for clock periods of 10, 15, and 20 nanoseconds as compared to schedules using assumption set 2 for the same allocation. In all cases, scheduling with the complex assumptions always produced equivalent or better results than scheduling with assumption set 2, which logically follows since assumption set 3 is

conceptually a superset of assumption set 2. These experiments demonstrate that HLS can explore a wider variety of design implementations and generate improved results when using an existing set of RT components with my reuse methodology. These advantages become more apparent when the performance and area of the resulting designs are analyzed.

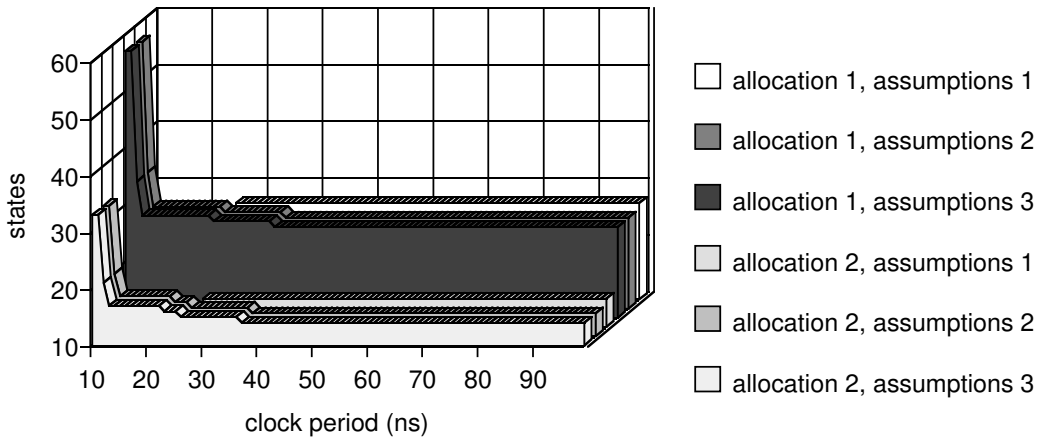


Figure 6.4: Scheduling results for Elliptic Filter, allocations 1 and 2.

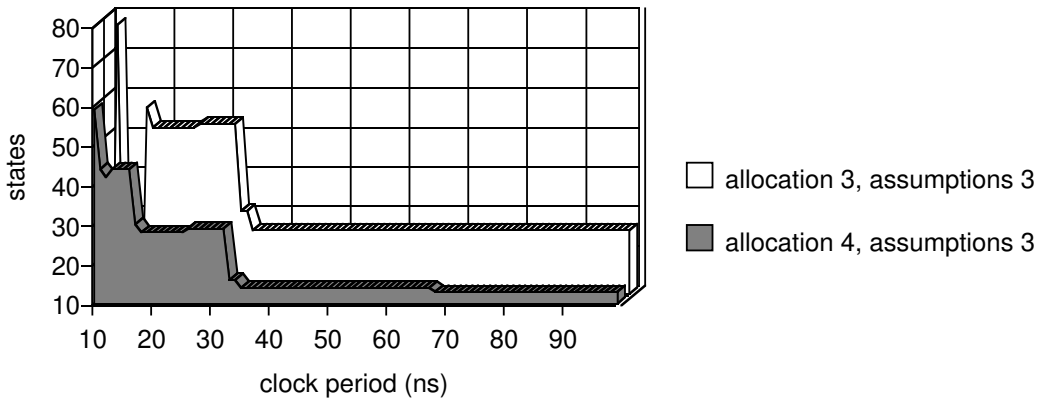


Figure 6.5: Scheduling results for Elliptic Filter, allocations 3 and 4.

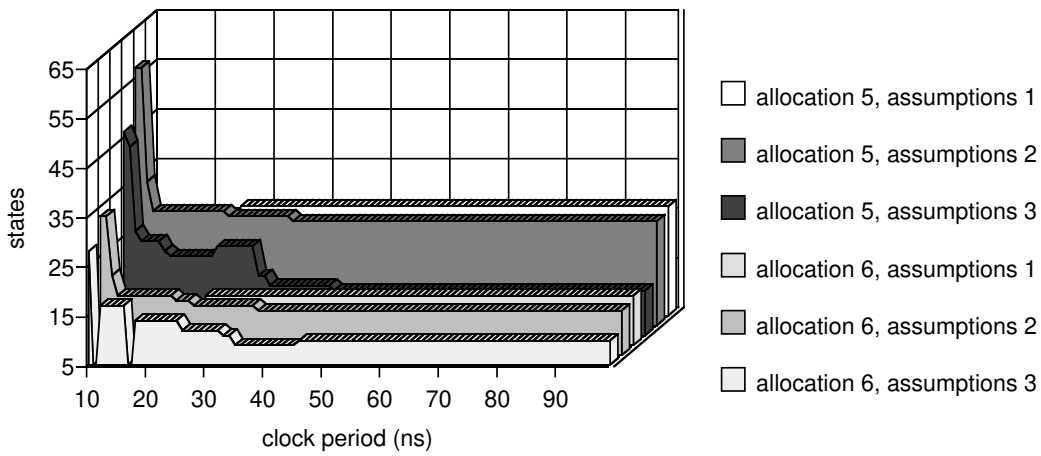


Figure 6.6: Scheduling results for Elliptic Filter, allocations 5 and 6.

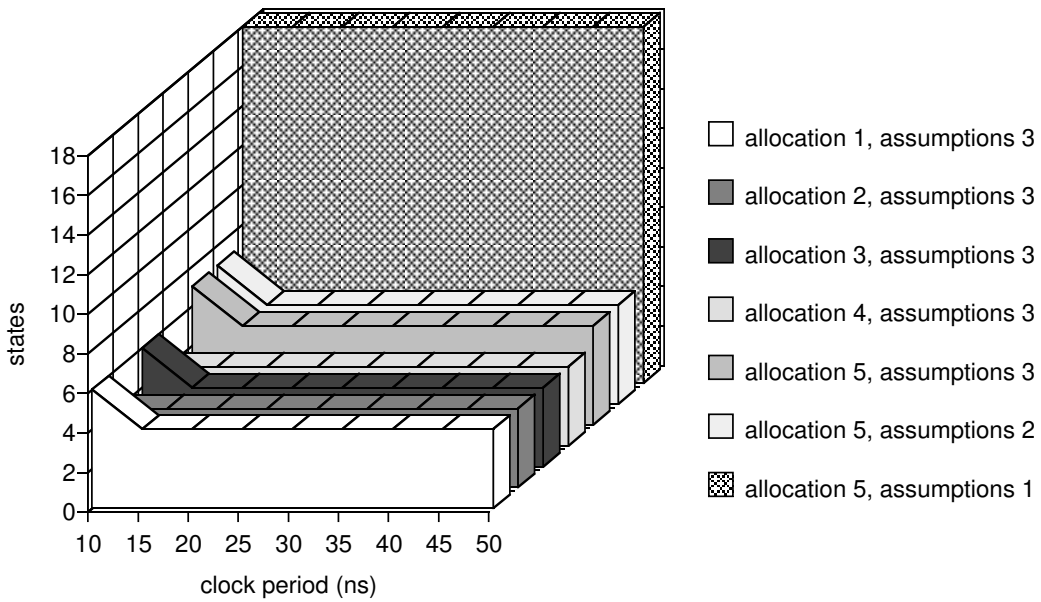


Figure 6.7: Scheduling results for Experimental Example.

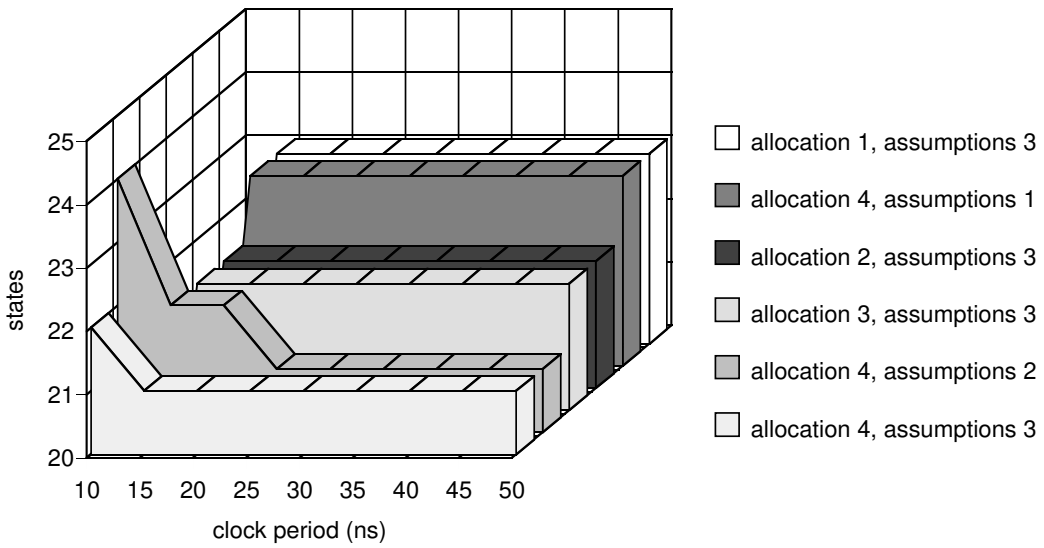


Figure 6.8: Scheduling results for Floating Point Adder/Subtractor.

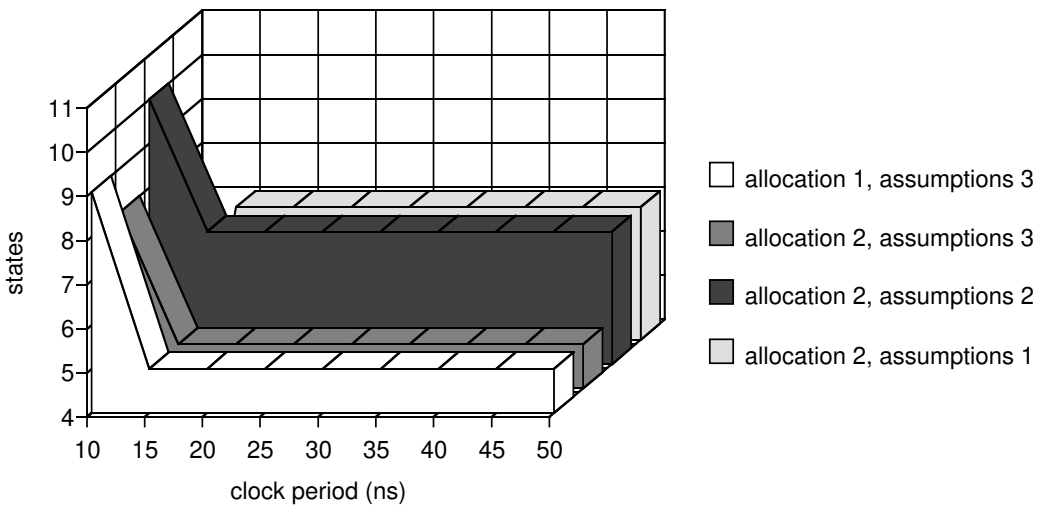


Figure 6.9: Scheduling results for Greatest Common Divisor.

Once scheduling is completed, register and multiplexor allocation and binding is performed to complete the datapath net list. I use this completed datapath net list to estimate the area cost of the design implementation. For the area cost of a design, I factor in the area of the functional units, registers, and multiplexors. Wiring area is not factored in since that can not be known until placement and routing are performed. Also, the area of random logic gates in the datapath is not factored into the area cost. Such gates will be optimized and mapped to technology-specific gates by subsequent logic optimization. The area cost considered in these experiments is the area of the datapath that will remain invariant through the later stages of synthesis, i.e., the parts of the datapath composed of pre-existing components. Consequently, the area costs presented in the figures that follow represent a strict lower bound on the area of the datapath for each design implementation. Performance is calculated by multiplying the clock period length by the total number of states in the scheduled design.

Figures 6.10, 6.11, and 6.12 compare the area and performance ratings of the design implementations derived for the integer divider algorithm, the greatest common divisor algorithm, and the experimental example. As these figures illustrate, my models and assumptions enable synthesis to explore component allocations not feasible with traditional methods and can produce smaller and better performing implementations over existing models and assumptions.

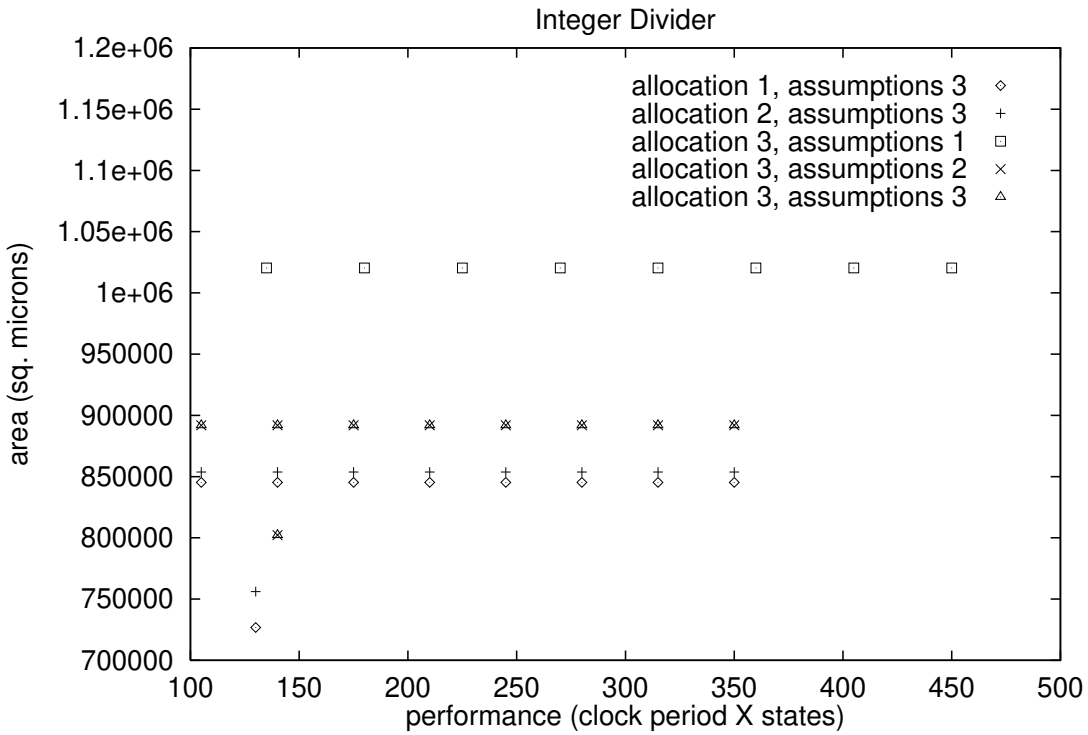


Figure 6.10: Area and performance of Integer Divider implementations.

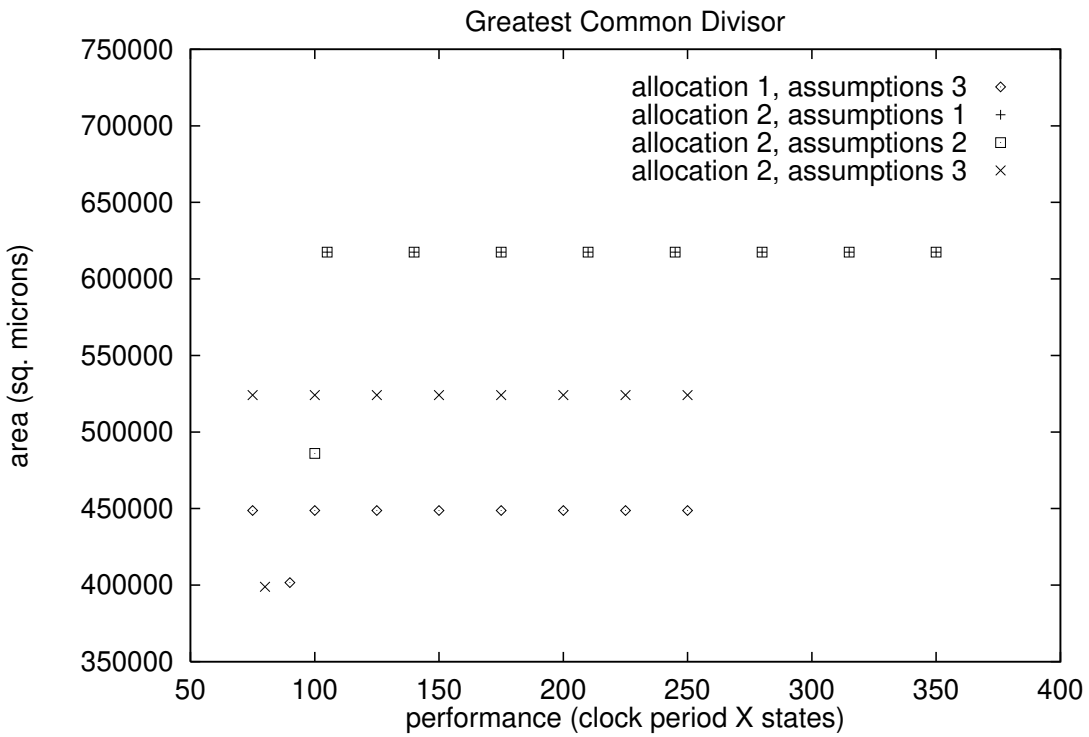


Figure 6.11: Area and performance of Greatest Common Divisor algorithm implementations.

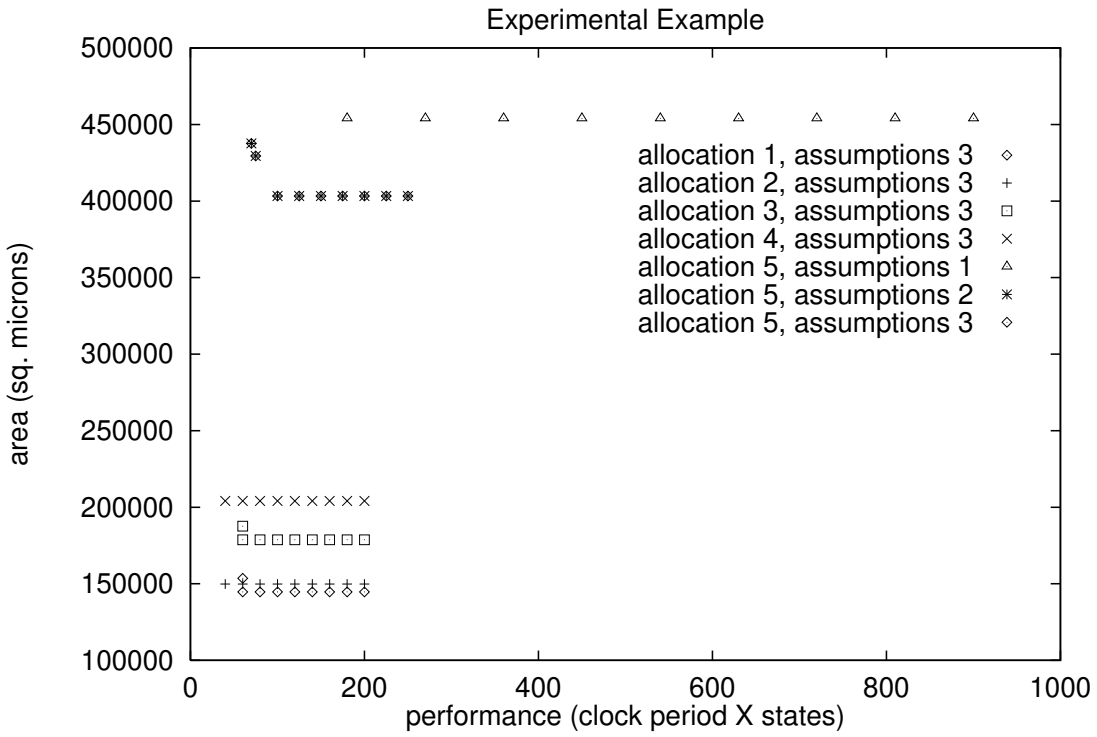


Figure 6.12: Area and performance of Experimental Example implementations.

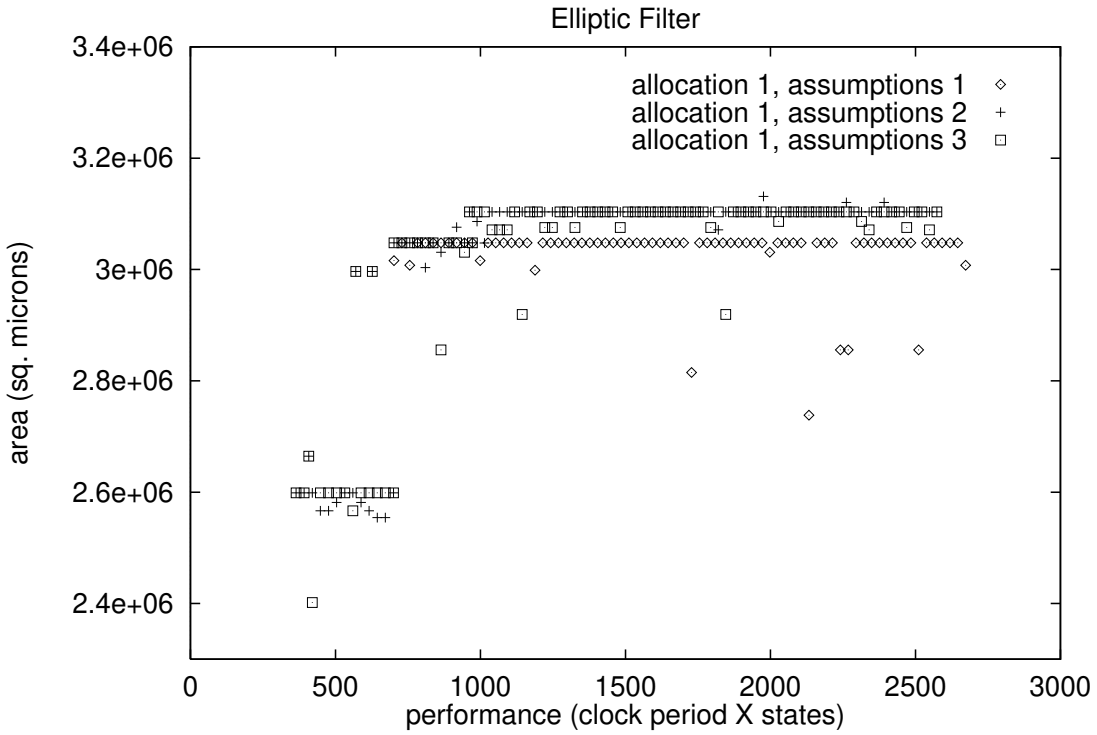


Figure 6.13: Implementations of Elliptic Filter, allocation 1.

Figures 6.13, 6.14, 6.15, 6.16, and 6.17 shows the results generated with the Elliptic Filter. Since the Elliptic Filter is a well-known algorithm for DSP use, I expected the simplified assumptions to perform well. Thus, I expected my methodology to be able to do little improvement, but also I expected it in the worst case to be comparable to other methods. But the other purpose of this example was to generate numerous design points to examine the effects of the heuristics in the algorithms. In particular, I wanted to examine the effects of register and multiplexor allocation and binding.

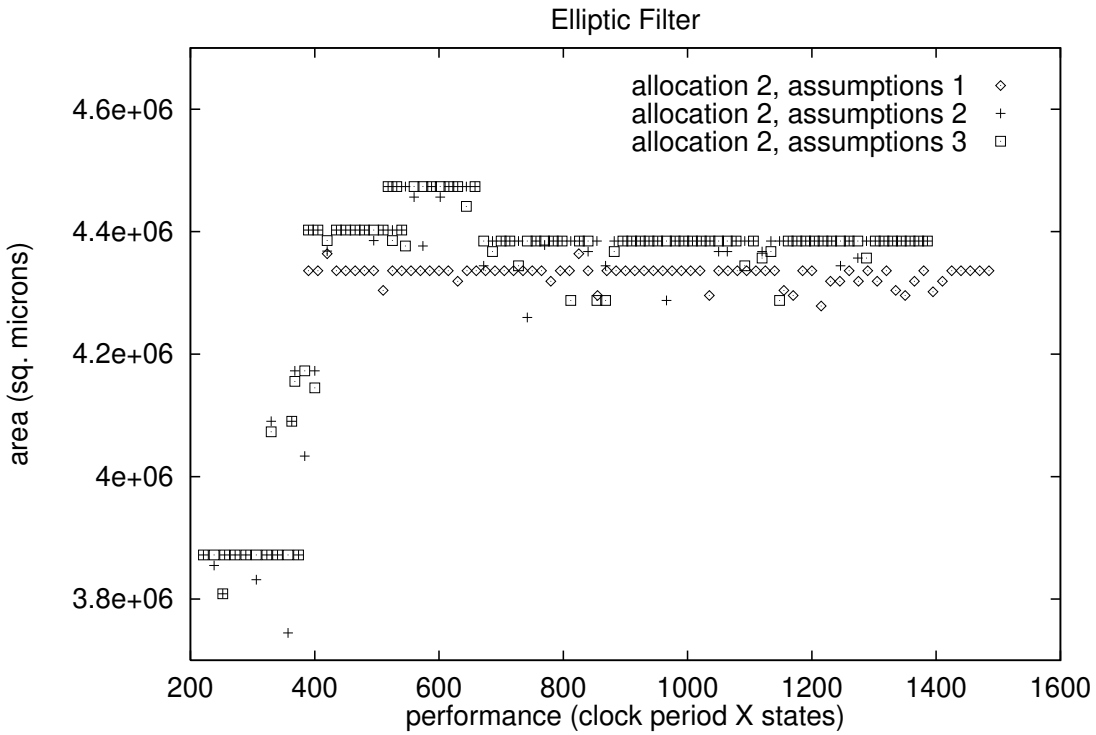


Figure 6.14: Implementations of Elliptic Filter, allocation 2.

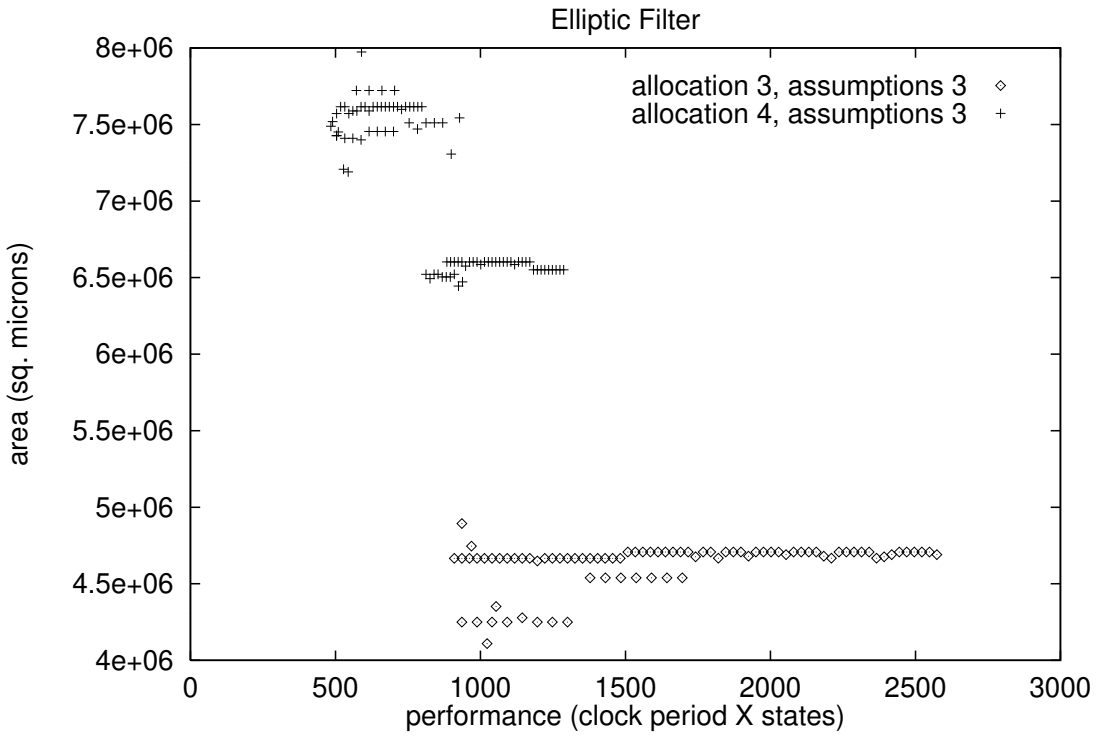


Figure 6.15: Implementations of Elliptic Filter, allocation 3 and 4.

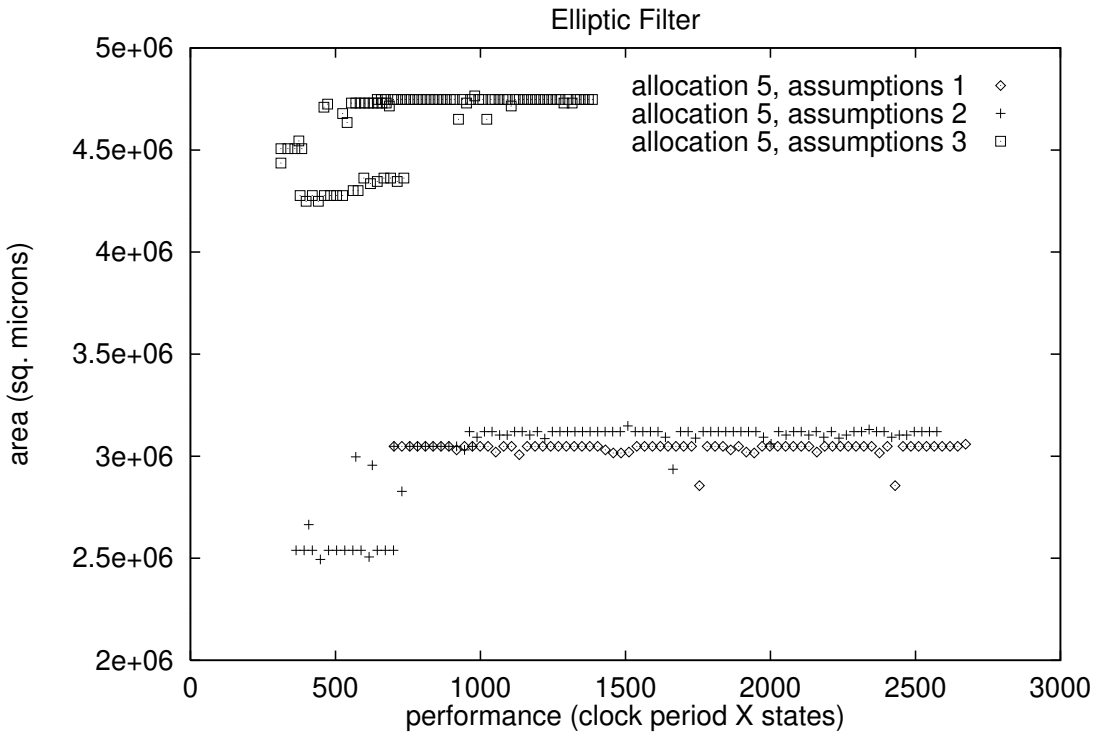


Figure 6.16: Implementations of Elliptic Filter, allocation 5.

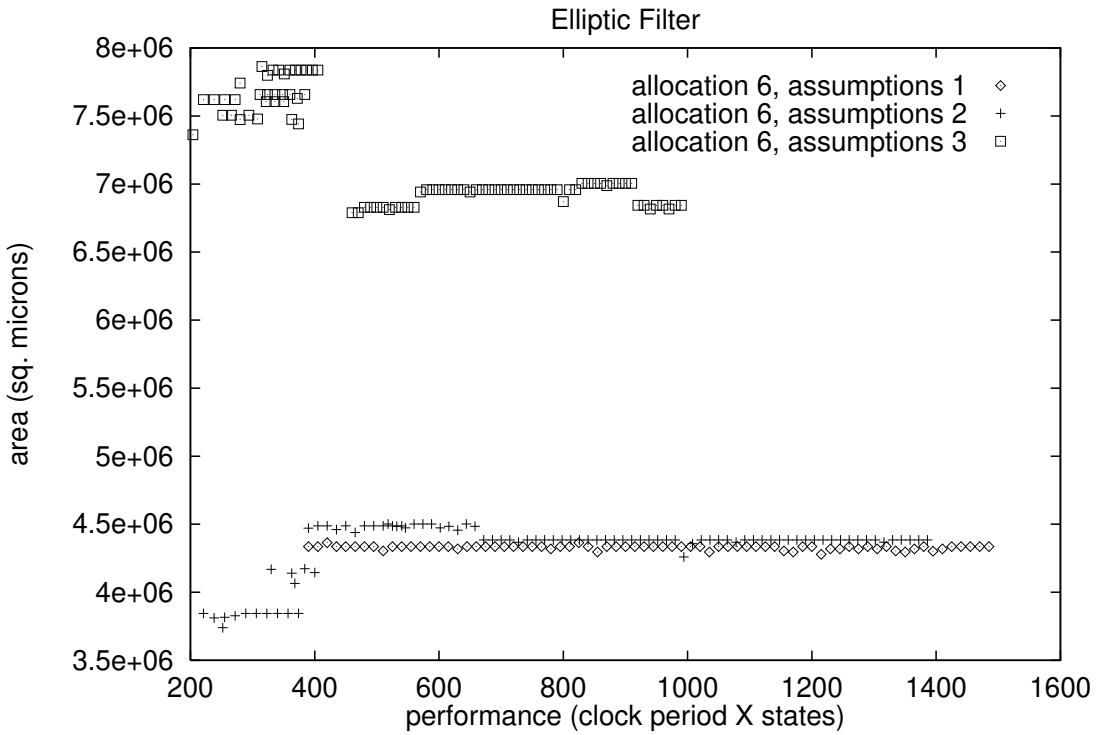


Figure 6.17: Implementations of Elliptic Filter, allocation 6.

Note the design points for a particular allocation and set of assumptions that have similar area but much different performance ratings. These design points have different performance ratings because each design point represents a scheduling of the algorithm with different clock period lengths, but the resulting schedules are similar. The design points have similar areas because the final implementations were similar in the number and types of registers and multiplexors used. In general, these similar designs vary in area by less than $\pm 5\%$ though the registers and multiplexors accounted for as much as 30% to 50% of the design area. This accounts for the design points clustering at particular levels vertically on the graph.

Design points with large differences in area (design points in different clusters) are indicative of vastly different implementations, even with the same allocation of components. In these experiments, greatly different implementations using the same allocation and assumptions were the result of the heuristic scheduler only using a subset of the available components in the allocation, and operation chaining and multicycling. The scheduling algorithm tries to bind as many operations as possible to the same datapath unit. As a result, it is possible for the algorithm to derive a schedule for the design that is minimized for the given constraints but does not need to utilize all component instances of the allocation. In effect, the scheduler derived schedules with different bound component allocations as the clock period changed. Figure 6.18 illustrates how the combination of different clock periods and the heuristics in the scheduling algorithm generated two different clusters of designs for the given allocation: a cluster that used 2 multiply/accumulators and 2 multipliers, and another that used 2 multiply/accumulators and 1 multiplier. Also, for allocations 5 and 6, the simpler assumption sets, 1 and 2, do not recognize the functionality of multiply/accumulators and ignore them. Consequently, the

implementations derived using allocations 5 and 6 under assumption sets 1 and 2 are similar to those derived with allocations 1 and 2. Operation chaining and multicycling can greatly effect the scheduling of the design which in turn also effects the registers and multiplexors in the final implementation. This is the reason for the cluster of design points indicated in Figure 6.19. The execution time for these design points is low because the multiplication operations have been multicycled, producing relatively short schedules with very low clock periods. However, the area for these design points is artificially low because insertion of latches for the inputs to the multicycled function units and the multiplexors to load these latches has not been done. The area for these design points with appropriate latching and additional multiplexors should be comparable to the other design points.

Figure 6.20 presents the results generated with designs for the floating point addition/subtraction algorithm. As with the other examples, my models and assumptions for behavioral synthesis can produce smaller and better performing implementations, and explore more of the design space. However, because of the complexity of the floating point adder/subtractor algorithm, the heuristic allocation tool only suggested a few of the possible component allocations for this design. Using the results shown in Figure 6.20, I created an additional allocation of components shown in Figure 6.21. Comparison of the results of synthesis on this allocation versus the other automatically generated design implementations is shown in Figure 6.22. These results suggest that my proposed synthesis methodology is useful in a two iteration design process where an initial iteration with the automatic tools can be used to generate results for a human designer to modify for improvements in the second iteration through the system.

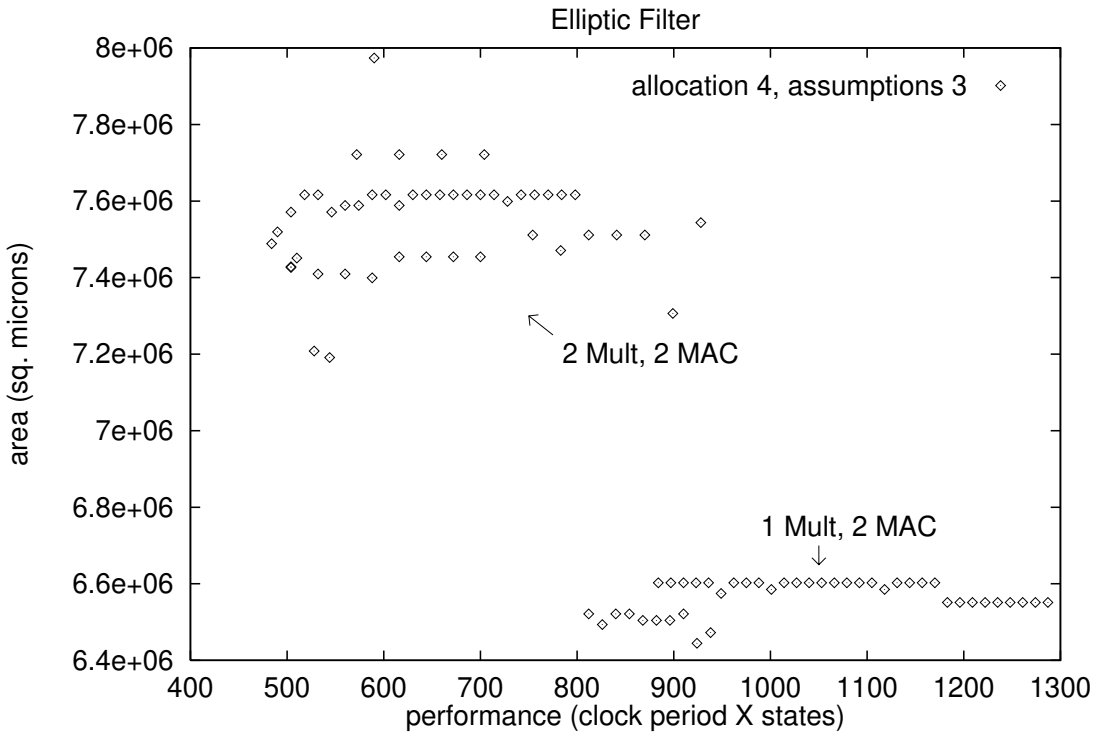


Figure 6.18: Implementations with partial use of the component allocation.

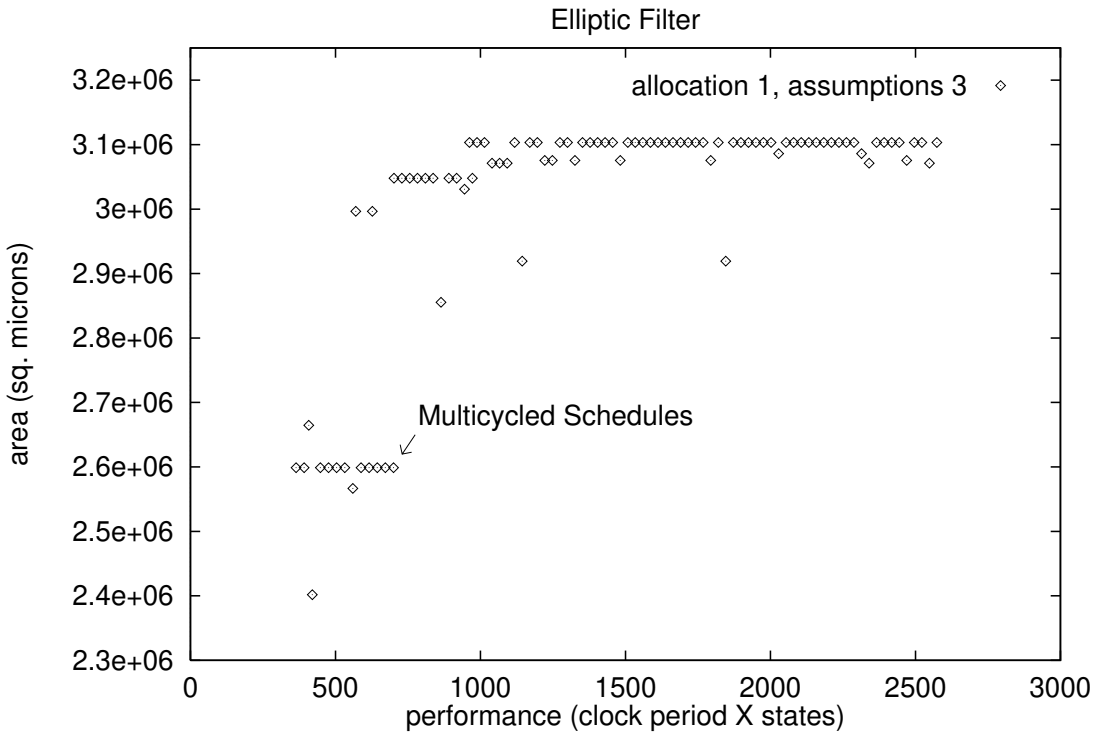


Figure 6.19: The effect of operation multicycling.

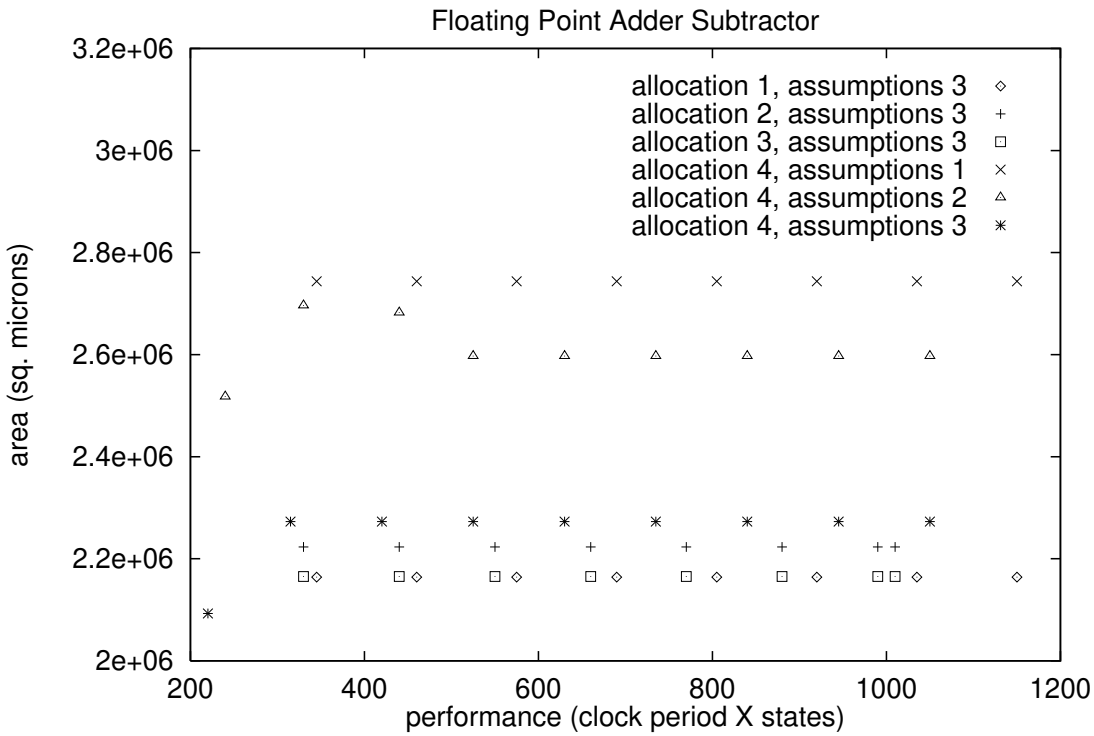


Figure 6.20: Design implementations for Floating Point Adder/Subtractor.

1 27–bit adder/subtractor
1 27–bit shifter
1 27–bit 0–detector
1 23–bit sub/comparator
1 23–bit 0–detector
1 23–bit 1s–detector
1 8–bit adder/subtractor
1 8–bit inc/dec
1 8–bit sub/comparator
1 8–bit comparator
1 8–bit 0–detector
1 8–bit 1s–detector

Figure 6.21: Component allocation A for the floating point adder/subtractor.

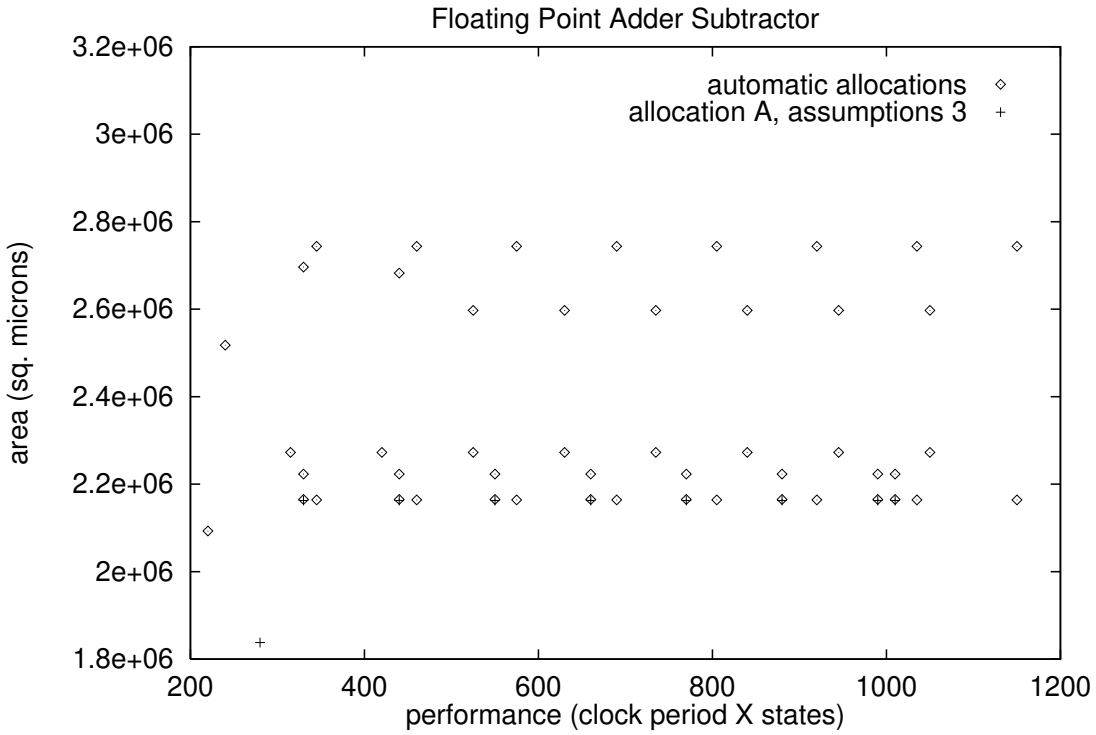


Figure 6.22: Comparison of Floating Point Adder/Subtractor implementations using allocation A.

For the set of examples presented, the results clearly indicate that the first assumptions used by early behavioral synthesis algorithms, one operation on one function unit per state, are inefficient. Because of the wide range of delays associated with operations of various types, all behavioral synthesis algorithms should incorporate operation chaining and multicycling to produce design implementations with efficient performance and area characteristics. But to truly make HLS practical for industrial use requires the ability of HLS tools to use custom sets of components to produce a greater variety of designs, and improve the designs produced. These results show that the models and methodology proposed here do that: generate more design implementations, and produce improved designs.

6.3 Summary

In this chapter, I described the experimental system implemented with ISE to demonstrate my methodology. I also provided an example design flow for component reuse in HLS. By no means do I propose that this methodology replace other HLS methodologies for all applications. As I stated at the beginning of this dissertation, current methods have been useful for synthesis to logic and for DSP applications. But as my experiments have proven, other methods and models do not perform as well as my methodology when synthesis attempts to reuse RT components on non-DSP applications.

Chapter 7

Conclusion

7.1 Summary of Dissertation

In this dissertation, I have described a formulation for reusing libraries of existing datapath RT components in high-level synthesis. Reuse of existing datapath RT components enables high-level synthesis to produce high quality implementations of a circuit design more quickly than a design process that relies completely on logic synthesis for the final implementation. My formulation of component reuse for HLS allows synthesis to explore a wider variety of implementations than previous methods would consider. It also allows the synthesis system to be customized to the individual designer's RT libraries, making HLS more practical for a wider variety of applications.

I have described the role of component reuse in synthesis and discussed related work in HLS. I discussed the aspects of other formulations that inhibit efficient reuse of RT components: simplified models of RT component functionality, rigid sets of component types used in synthesis, and the embedding of assumptions about component types and functionality within algorithms. I described how my formulation addresses these issues and differs in its overall approach to HLS.

I described the various aspects of my work: the overall model of the HLS process; the abstraction of RT component functionality; the structures for synthesis to match RT component functionality, link abstractions, and maintain component library information. I demonstrated the use of these models and representations in two example algorithms: an algorithm to allocate function units to a behavioral description, and an algorithm to schedule operations of a behavioral description to a given set of components. These algorithms illustrated some of the algorithmic complexity of efficiently reusing datapath components in HLS.

Finally, I described the experimental system implemented with ISE to demonstrate my methodology. With these experiments, I provided an example design flow for component reuse in HLS. I presented experimental results for several well known examples, and discussed the advantages and limitations of this approach to synthesis as illustrated by the results. These results demonstrate how this formulation of synthesis for reuse can explore a greater variety of design implementations, and generate improved results.

7.2 Summary of Contributions

The research described in this dissertation makes several fundamental contributions to HLS. I described a more technology-specific synthesis approach for HLS that can be customized to individual RT component libraries. I define an abstraction of RT components to recognize the complex, concurrent functionality of combinatorial RTL components, and describe a mechanism to efficiently map the input description of HLS to RTL behavior. I describe a synthesis representation

to model abstract input behavior in terms of RT component functionality. Finally, I did some initial exploration into the algorithmic complexities of performing HLS for component reuse.

In order to effectively reuse existing RT components, I defined a synthesis design process that integrates technology specific libraries of RT components with the HLS system. Unlike the traditional synthesis that assumed a single, inflexible library of models that represented generic RT components, the synthesis system in my methodology is designed so that different libraries can be interchanged for different implementation technologies, applications, designer custom component sets, etc. Also, instead of generating a result that must go through technology mapping to the final implementation of the datapath components, the result of the methodology I proposed produces a datapath RT structure of existing RT components.

Key to my methodology for RT component reuse is the component database model that abstracts the complex, concurrent functionality of combinatorial RT components. This model of RT component functionality has not been considered in HLS by any other work. The component database hierarchically organizes the functionality of RT components for use with HLS as well as maintains physical information about the components. Used in conjunction with the component database, the template parse tree provides a mechanism to map input behavior to RT component functionality. The component database and the template parse tree together provide HLS tools with an interface to any combinatorial component library to be used for synthesis.

During synthesis, the RTDFG represents the design behavior in terms of RT component functionality. The RTDFG can unambiguously maintain the binding or mapping of input behavior to the RT structure being developed. The RTDFG is important in bridging the semantics of HDL behavior and the operation of RT components.

The algorithms and design flow presented in this dissertation explored issues related to using a model of component functionality with complex, concurrent behavior. These issues included metrics to evaluate mappings of behavior to component functions, search methods to discover complex and concurrent functions in the input behavior, and experimentation with the effects of the clock period length, operation chaining and multicycling. This initial synthesis work for RT component reuse gives a hint of the additional complexity of dealing with complex, concurrent component behavior.

7.3 Future Directions

Since the thrust of this dissertation is to define a new HLS methodology for RT component reuse, there are many possibilities for continuing work in applying this methodology. The algorithms presented here used simple search strategies to find solutions to the various synthesis tasks. There are many ways to improve the search strategies in these algorithms. Alternatively, different methods to perform the synthesis tasks for RT component reuse could provide better algorithm performance, e.g., integer programming, graph partitioning methods, etc.

The experiments presented in this dissertation were demonstrated on a single target technology and architecture: a datapath with multiplexor-based interconnection using the VTI 0.8 micron CMOS datapath library for a bit-slice layout. Exploration should be done with different datapath architectures, e.g., multiple bus interconnection, and with libraries for different target technologies, e.g., macro-block libraries for FPGAs.

My model of RT component functionality only deals with combinatorial function units. Extensions or additional models of functionality are needed for memories and RT components exhibiting sequential behavior, such as counters. The time related aspect of the functionality of these types of components would likely require significantly more complexity in both the representation of the component's behavior, and in the algorithms to perform synthesis using such components.

In the overall scope of HLS, it would of great interest to conduct further experimentation with an iterative, interactive synthesis process that incorporates the component reuse approach I have described here. Intuitively, it is easy to imagine designs with portions of the datapath that will be implemented by reusing components from a library, while other parts of the datapath are implemented for the specific application by using logic synthesis. An iterative synthesis approach could be effective for these types of designs by doing a "first-pass" with a reuse methodology to discover which portions of the design can be implemented with components from a library, then partially implementing the design with library components, i.e., partially binding the design to a partial implementation. A subsequent iteration can use a tradition synthesis approach to implement the remainder of the design with application-specific logic. To guide this process, it would be useful to

incorporate designer interaction with the synthesis tools so that the partial implementation, and the binding of the design to the implementation, can be specified manually. This would require metrics to provide feedback to the designer. In this work, datapath function unit area and delays were considered. Metrics using component information concerning other characteristics such as power consumption and testability should be explored. In addition, a more complex model of area and delay would be useful for refinement of the design at lower levels.

Bibliography

[AhSU86] A. Aho, R. Sethi, J. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

[AnDu92] R. Ang and N. Dutt, "On Linking RT Component Functionality to Abstract HDL Behavior," Technical Report 92-97, University of California at Irvine, revised July 1993.

[AnDu93] R. Ang and N. Dutt, "Allocation of Functional Units from Realistic Component Libraries," Technical Report 93-47, University of California at Irvine, Oct. 1993.

[AnDu95] R. Ang and N. Dutt, "Scheduling for Reuse of Datapath Components in the Interactive Synthesis Environment," Technical Report 95-36, University of California at Irvine, Aug. 1995.

[BrBr88] G. Brassard and P. Bratley, *Algorithmics: Theory and Practice*, Prentice-Hall, New Jersey, p. 15, 1988.

[Camp91] R. Camposano, "Path-Based Scheduling for Synthesis," *IEEE Transactions on Computer-Aided Design for Integrated Circuits and Systems*, vol. 10, no. 1, pp. 136–141, January 1991.

[CaTa89] R. Camposano and R.M. Tabet, "Design Representation for the Synthesis of Behavioral VHDL Models," *Proceedings of the 9th International Symposium on Computer Hardware Description Languages and their Applications*, 1989.

[CaTr89] R. Camposano and L. Trevillyan, "The Integration of Logic Synthesis and High-Level Synthesis," *Proceedings of the International Symposium of Circuits and Systems*, pp. 744–747, 1989.

[CaWo91] R. Camposano and W. Wolf, Editors, *High-Level VLSI Synthesis*, Kluwer Academic Publishers, Boston, 1991.

[ChGa90] G. Chen and D. Gajski, "An Intelligent Component Database for Behavioral Synthesis," *Proceedings of the 27th Design Automation Conference*, pp. 150-155, 1990.

[ChGa92] V. Chaiyakul and D.D. Gajksi, "Assignment Decision Diagrams for High-Level Synthesis," Technical Report 92-103, University of California at Irvine, Dec. 1992.

[CKPR93] M. Corazao, M. Khalaf, L. Guerra, M. Potkonjak, and J. Rabaey, "Instruction Set Mapping for Performance Optimization," *Proceedings of the International Conference on Computer-Aided Design*, pp. 518–521, 1993.

[ErLa85] M. Ercegovac and T. Lang, *Digital Systems and Hardware/Firmware Algorithms*, John Wiley & Sons, Inc., New York, pp. 714–715, 1985.

[GiCa93] E. Girczyc and S. Carlson, "Increasing Design Quality and Engineering Productivity through Design Reuse," *Proceedings of the 30th Design Automation Conference*, pp. 48–53, 1993.

[GlGr78] R. Glanville and S. Graham "A New Method for Compiler Code Generation," *Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages*, pp. 231–240, 1978.

[GCDM93] W. Guerts, F. Catthoor, H. De Man, "Heuristic Techniques for the Synthesis of Complex Functional Units," *Proceedings of the European Conference on Design Automation*, pp. 552–556, 1993.

[GCDM95] W. Guerts, F. Catthoor, and H. De Man, "Quadratic Zero-One Programming-Based Synthesis of Application-Specific Data Paths," *IEEE Transactions on CAD*, vol. 14, no. 1 pp. 1–11, Jan 1995.

[GrKP85] J. Granacki, D. Knapp, and A. Parker, "The ADAM Advanced Design Automation System: Overview, Planner and Natural Language Interface," *Proceedings of the 22nd Design Automation Conference*, pp. 727–730, 1985.

[Had95] T. Hadley, "A System for Interactive High-Level Synthesis," *Ph.D. Dissertation, University of California at Irvine*, April, 1995.

[JCG94] H.-P. Juan, V. Chaiyakul and D.D. Gajski, "Condition Graphs for High-Quality Behavioral Synthesis," *Proceedings of the International Conference on Computer-Aided Design*, 1994.

[Jha95] P. Jha, "High-Level Library Mapping for RT Components," *Ph.D. Dissertation, University of California at Irvine*, 1995.

[Kahrs86] M. Kahrs, "Matching a parts library in a silicon compiler," *Proceedings of the International Conference on Computer-Aided Design*, pp. 169–172, 1986.

[Kipp91] J. Kipps, "An Approach to Component Generation and Technology Adaptation," *Ph.D. Dissertation, University of California at Irvine*, December 1991.

[KnWi92] D. Knapp and M. Winslett, "A Prescriptive Formal Model for Data-Path Hardware," *IEEE Transactions on Computer-Aided Design for Integrated Circuits and Systems*, vol. 11, no. 2, pp. 158–184, February 1992.

[LGCP91] D. Lanneer, G. Goossens, F. Catthoor, M. Panwels, H. De Man, "An Object-oriented Framework Supporting the Full High-level Synthesis Trajectory," *Proceedings of the 10th International Symposium on Computer Hardware Description Languages and their Applications*, pp. 281–301, 1991.

[LMD94] B. Landwehr, P. Marwedel, R. Domer, "OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming," *Proceedings of EURO-DAC*, 1994.

[LMPa94] C. Liem, T. May, and P. Paulin, "Instruction-Set Matching and Selection for DSP and ASIP Code Generation" *Proceedings of the European Conference on Design Automation*, pp. 31–37, 1994.

[McFar78] M. C. McFarland, "The Value Trace: A Data Base for Automated Digital Design," Ph. D. Dissertation, Dept. of Electrical Engineering, Canegie-Mellon University, 1978.

[Mano88] M. Mano, *Computer Engineering: Hardware Design*, Prentice Hall, 1988.

[Marw86] P. Marwedel, "A New Synthesis Algorithm for the MIMOLA Software System," *Proceedings of the 23rd Design Automation Conference*, pp. 271–277, 1986.

[Mar93] P. Marwedel, "Tree-Based Mapping for Algorithms to Predefined Structures," *Proceedings of the International Conference on Computer-Aided Design*, pp. 586–593, 1993.

[MiLD92] P. Michel, U. Lauther and P. Duzy, *The Synthesis Approach to Digital System Design*, Kluwer, 1992.

[OrGa86] A. Orailoglu and D. Gajski, "Flow Graph Representation," *Proceedings of the 23rd Design Automation Conference*, pp. 503–509, 1986.

[PaBu87] T.W. Parks, C.S. Burrus, *Digital filter design*, Wiley, New York, 1987.

[PaDu95] P. Panda and N. Dutt, "1995 High-Level Synthesis Design Repository," Tech. Report 95-04 University of California at Irvine, Feb. 1995. *available for anonymous FTP from ftp.ics.uci.edu*.

[PaGa87] B. Pangrle and D. Gajski, "Slicer: A State Synthesizer for Intelligent Silicon Compilation," *Proceedings of the International Conference on Computer Design*, pp. 42–45, 1987.

[PaKn89] P. Paulin and J. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's," *IEEE Transactions on Computer-Aided Design for Integrated Circuits and Systems*, vol. 8, no. 6, pp. 661–679, June 1989.

[PLNG90] R. Potasman, J. Lis, A. Nicolau, D. Gajski, "Percolation Based Synthesis," *Proceedings of the 27th Design Automation Conference*, pp. 444–449, 1990.

[Rose86] W. Rosenstiel, "Optimizations in High Level Synthesis," *Microprocessing and Microprogramming (18)*, pp. 543–549, 1986.

[RaKu92] D. S. Rao and F. J. Kurdahi, "System Modeling for High-Level Synthesis Using Regularity Extraction," *Proceeding of 6th Workshop on High Level Synthesis*, pp. 267–272, 1992.

[RuGB90] E. Rundensteiner, D. Gajski, L. Bic, "The Component Synthesis Algorithm: Technology Mapping for Register Transfer Descriptions," *Proceedings of the International Conference on Computer-Aided Design*, pp. 208–211, 1990.

[SMZG96] B. Sharma, M. Mahmood, N. V. Zanden, and A. Ginetti, "Flexible Datapath Synthesis Using Parameterized HDL Components," *Proceedings of the 3rd Asia-Pacific Conference on Hardware Description Languages*, pp. 82–85, 1996.

[Thom90] D. Thomas et al., *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench*, Kluwer, 1990.

[VTI92] *0.8 Micron CMOS VCC4DP3 Datapath Library*, VLSI Technology, Inc., 1992.

[WPAV92] A. van der Werf, M. Peek, E. Aarts, J. Van Meerbergen, P. Lippens, W. Verhaegh, "Area Optimization of Multi-Functional Processing Units," *Proceedings of the International Conference on Computer-Aided Design*, pp. 292–299, 1992.