UNIVERSITY OF CALIFORNIA
Irvine

High-Level Library Mapping for RT Components

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Information and Computer Science

by

Pradip Kumar Jha

Committee in charge:
Professor Nikil D. Dutt, Chair
Professor Daniel D. Gajski
Professor Fadi J. Kurdahi

1995

©1995
PRADIP KUMAR JHA
ALL RIGHTS RESERVED

The dissertation of PRADIP KUMAR JHA is approved,
and is acceptable in quality and form for
publication on microfilm:

_____

_____

_____

Committee Chair

University of California, Irvine

1995

Dedication

To my mother,
Late Nunumani Jha

Contents

List of Figures vii

List of Tables ix

Acknowledgement x

Curriculum Vitae xi

Abstract xiii

Chapter 1 Introduction **1**
1.1 Motivation 1
1.2 Thesis Contributions 4
1.3 Thesis overview 5

Chapter 2 High-Level Library Mapping **6**
2.1 Library Mapping 6
2.1.1 Logic level library mapping 7
2.1.2 RT level library mapping 10
2.1.3 System level mapping 12
2.2 High-Level Library Mapping 13
2.2.1 HLLM versus LLLM 15
2.2.2 Domain of HLLM 16
2.2.3 Design scenario with HLLM 17
2.2.4 HLLM tasks 18
2.3 Summary 19

Chapter 3 RT Component Libraries **21**
3.1 Generic RT Library Definition 23
3.1.1 Combinational components 24
3.1.2 Sequential components 26
3.1.3 Miscellaneous components 27
3.2 Generic component coverage 28
3.3 Generic component effectiveness 30
3.4 Summary 33

Chapter 4 HLLM for ALUs 34
4.1 Introduction 34
4.2 Problem Definition 35
4.2.1 Assumptions 36
4.2.2 Universal ALU representation 37
4.2.3 Canonical ALU functions 39
4.2.4 Representation of library components 39
4.2.5 Logic function representation 40
4.2.6 Mapping rule representation 42
4.2.7 The cost function 44
4.3 ALU Mapping Approach 46
4.4 An ALU Mapping Algorithm 50
4.4.1 The search space 53
4.4.2 Dynamic programming algorithm 54
4.5 Experimental results 57
4.5.1 Comprehensiveness 57
4.5.2 Goodness 61
4.6 Summary 67

Chapter 5 HLLM for Memories 68
5.1 Introduction 68
5.2 Previous Work 69
5.3 Problem Definition 72
5.3.1 Memory specification 74
5.3.2 Memory mapping 75
5.3.3 Port mapping 77
5.3.4 Bit-width mapping 80
5.3.5 Word mapping 85
5.3.6 The cost function 91
5.4 Memory Mapping Approach 93
5.4.1 Assumptions 93
5.4.2 Memory mapping algorithm 95
5.5 Experimental results 99
5.5.1 Analysis of experimental results 102
5.6 Summary 105

Chapter 6 GENUS and HLLM Environment 106
6.1 GENUS Environment 106
6.2 Model Generation 108
6.3 Technology Projection 109
6.4 HLLM User Interface 112
6.4.1 Setting design parameters 114
6.4.2 Performing HLLM 116
6.4.3 Displaying mapped design 116
6.5 Summary 116

Chapter 7 Conclusion 118
7.1 Summary of Dissertation 118
7.2 Summary of Contributions 120
7.3 Future Directions 121

List of Figures

2.1 Library mapping example (a) Source design (b) Target design . . . 7
2.2 An example for logic level library mapping (a) Target library (b) Subject graph (c) Match set (d) A cover 8
2.3 An example for FPGA mapping 10
2.4 Logic-level mapping of an ALU 10
2.5 Functional decomposition of an ALU 12
2.6 Comparing HLLM with LLLM 15
2.7 Domain of High Level Library Mapping 17
2.8 Design scenario with HLLM 18

3.1 Coverage of generic components across different libraries and parameters 29
3.2 Study of overhead incurred with generic library 31
3.3 Experimental results of generic component overhead study 31

4.1 High-level library Mapping of an ALU 35
4.2 The Universal ALU 37
4.3 A library ALU example: (a) Port description (b) Function description 40
4.4 Implementation of logic unit with two functions: OR and XOR . . 41
4.5 Port names : (a) Source canonical ALU (b) Target canonical ALU 43
4.6 Worst case delay for an ALU design 45
4.7 Overall system for ALU mapping 47
4.8 Example source component: (a) Port description (b) Function table 48
4.9 Example target component: (a) Port description (b) Function table 48
4.10 Mapping of source component (S) to canonical component (C) . . . 49
4.11 Mapping of source component (S) to target component (T) 50
4.12 Partial solution example: (a) Pictorial representation (b) Textual representation 52
4.13 The search space for ALU mapping example 54
4.14 Execution of dynamic programming algorithm on an ALU example 56
4.15 Experimental results for area-efficient mapping 58
4.16 Experimental results for delay-efficient mapping 59
4.17 Experimental setup for comparing HLLM with logic synthesis . . . 62
4.18 Source and target ALUs for comparative study 62

4.19 Performance metric for Area-optimized designs 64
4.20 Run-time for Area-optimized designs 64
4.21 Performance metric for Delay-optimized design 65
4.22 Run-time for Delay-optimized design 65

5.1 Classification of memory mapping works 70
5.2 Sample high-level library mapping for memories: (a) Source and target modules (b) Mapping result 73
5.3 Three degrees of freedom in memory mapping problem 75
5.4 Port map example (a) Source and target memory components (b) Port map between source s and target t. 79
5.5 An example for bit-width mapping (a) Source and target memory components (b) Target memory module set (c) Enumeration of memory compositions (d) List of best compositions for different bit-widths (e) Data input-output connection between the source and the target modules 84
5.6 Word mapping example (a) Source and target memory modules (b) Intermediate results from word mapping algorithm (c) Final design 90
5.7 Cost of a memory design 91
5.8 Two types of memory composition (a) Regular composition (b) Irregular composition 94
5.9 A memory mapping example with *linear* algorithm (a) Source and target modules (b) Design with word mapping followed by bit-width mapping (c) Design with bit-width mapping followed by word mapping 97
5.10 Complete design for the memory mapping example 98
5.11 Memory mapping result I 99
5.12 Memory mapping result II 100
5.13 Memory mapping result III 101
5.14 Memory mapping result IV 102

6.1 GENUS library structure 107
6.2 Behavioral VHDL simulation models 108
6.3 Aggregate error profile as compared to DTAS 110
6.4 Aggregate error profile as compared to LAST/TELE 111
6.5 HLLM system 113
6.6 Top level window for ALU mapping 113
6.7 Top level window for memory mapping 114

List of Tables

3.1 Generic combinational components 25
3.2 Generic combinational components (continued) 26
3.3 Generic sequential components 27
3.4 Generic miscellaneous components 28

4.1 Canonical arithmetic functions 38
4.2 Canonical logic functions 38
4.3 Canonical comparison functions 39
4.4 Minterms for logic functions 41
4.5 Sample mapping rules 43
4.6 Sample mapping rules for logic functions 44
4.7 Selected set of rules for mapping example 49

Acknowledgement

I am greatly indebted to Professor Nikil D. Dutt, my advisor, for accepting me as a graduate student under his guidance five years back and staying with me since then. He has always been there to listen to my ramblings often at unearthly hours, yet keeping me focussed on my PhD topic. Thanks to his understanding personality, my five year stay at UC Irvine has been quite rewarding and a pleasant experience. I am also thankful to him for his constructive comments on improving my technical writing and presentation skills.

I would like to thank Professor Daniel Gajski and Professor Fadi Kurdahi for their guidance and insight related to my research, and also for serving on my committee. I have been lucky to get an opportunity to work with them; it was a brief yet quite a rewarding experience. Professor Gajski's constructive comments during Friday meetings as well on Saturday lunches have been very helpful to keep me focussed in my research. I am also grateful to Professor Yu-Chin Hsu and Dr. Youn-Sik Hong for serving on my candidacy examination committee.

I would like to express my gratitude to the members of the CADLAB for creating a stimulating work environment at the lab as well as for being wonderful companions in extra-curricular activities. Special thanks are due to Jie Gong, Sanjiv Narayan, Viraphol Chaiyakul, Lognath Ramachandran, Champaka Ramachandran, Alfred Thordarson, Smita Bakshi, Tedd Hadley, Nels Vander Zanden, Frank Vahid, Preeti Panda, Roger Ang, Min Xu, Erica Juan, Nancy Holmes, Sri Parameswaran and Tadatoshi Ishii. I am also grateful to Tedd Hadley for his help in getting me accustomed to the GENUS software as well as assisting me with the computing environment problems. I would also like to thank my friend Raghava Kondepudy for stimulating late night discussions about life and people.

I am also grateful to my family members for providing the moral support. Special thanks are due to my father Kameshwar Jha, my uncle A. K. Mishra and my brother P. C. Jha for their encouragement and to my dear wife Neena for the loving companionship.

Finally I would like to thank SRC for their support of this work.

Curriculum Vitae

July 2, 1968	Born Bhagalpur, India
1990	B.Tech. in Computer Science & Engineering, Indian Institute of Technology, New Delhi, India
1990-1991	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
1994	M.S. in Information and Computer Science, University of California, Irvine, USA
1995	Ph.D. in Information and Computer Science, University of California, Irvine, USA Dissertation: *High-Level Library Mapping for RT Components*

Publications

Journal papers

1. "High-Level Library Mapping for Arithmetic Components," Pradip Jha and Nikil Dutt, *To Appear in IEEE Transactions on VLSI Systems,* March 1996.
2. "Reclocking Controllers for Minimum Execution Time," Pradip Jha, Sri Parameswaran and Nikil Dutt, *To Appear in IEICE Transactions on Fundamentals of Electronics, Communications and Computer Science.*
3. "Rapid Estimation for Parametrized Components in High-Level Synthesis," Pradip Jha and Nikil Dutt, *IEEE Transactions on VLSI Systems,* Vol. 1, No 3, September 93.
4. "RT Component Sets for High-Level Design Applications," Nikil Dutt and Pradip Jha, *To appear in VLSI Design Journal, 1995.*

Book chapters

1. "Towards Better Accounting of Physical Design Effects in High Level Synthesis," Pradip Jha, Champaka Ramachandran, Fadi Kurdahi and Nikil Dutt, *Novel Approaches in Logic and Architecture Synthesis,* A. Mignotte and G. Saucier, Editors, Chapman and Hall, 1995.

Selected conference papers

1. "Design Reuse through High-Level Library Mapping," Pradip Jha and Nikil Dutt, *Proc. of The European Design and Test Conference,* March 1995.
2. "An Empirical Study on the Effects of Physical Design in High-Level Synthesis," Pradip Jha, Champaka Ramachandran, Nikil Dutt and Fadi Kurdahi, *Proc. of 7th Int. Conf. on VLSI Design,* January 94.

Selected technical reports

1. "The GENUS User Manual and C Programming Library," Pradip Jha, Tedd Hadley and Nikil Dutt, *Technical Report 93-32, University of California at Irvine,* April 93.
2. "An Evaluative Study of RT Component Libraries," Pradip Jha, Nikil Dutt and Daniel Gajski, *Technical Report 93-11, University of California at Irvine,* March 93.

Abstract of the Dissertation

High-Level Library Mapping for RT Components

by

PRADIP KUMAR JHA

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1995

Professor Nikil D. Dutt, Chair

We present **High-level library mapping** (HLLM), a novel library mapping technique for RT level components that supports the current day design methodologies using *High-level design* and *Design reuse*. HLLM uses RT level functional behavior to perform mapping of a RT component onto another RT component of similar complexity. The technique is well-suited for mapping regularly-structured datapath and memory components. In this dissertation, we first introduce high-level library mapping and distinguish it from other library mapping approaches. Next, we define a generic library of reusable RT parts that provides the functional basis of HLLM. We demonstrate the HLLM approach on two classes of components, namely ALUs and memories. Our experimental results demonstrate the comprehensiveness and efficacy of the HLLM approach in mapping RT components. Finally, we describe the GENUS library environment and a user-interface to the HLLM system for performing RT level design. The HLLM approach we describe uses functional behavior to elevate library mapping from the logic to the RT level.

Chapter 1

Introduction

1.1 Motivation

Current VLSI designs have reached complexities of millions of gates; it is expected to grow even higher in the coming years. Systems of such complexity are very difficult to design by handcrafting each transistor or by defining each signal in terms of logic gates, since human designers can not do an effective job for problems involving a large number of objects. For systems of such complexities, even the traditional design automation tools such as logic synthesis and physical design fail to provide good solutions in reasonable amounts of time. There is a need to develop design methodologies that can handle systems of higher complexity.

Another requirement for the current system design methodology is to reduce the design time. For certain time critical applications, if the product is delayed by six months, one could lose the market by as much as half. With increasing system complexity, the design time reduction requirement becomes even more difficult to achieve.

Industry and academia have been very actively working on finding solutions to the problem of increased complexity and the problem of reducing the time-to-market for VLSI systems. The two prominent techniques to handle the current day design problems that have emerged out of these works are: *High-Level Design* (HLD) and *Design Reuse*. High-level design [GDWL92] suggests that the design

process should be elevated to higher levels of abstraction where design functionality and tradeoffs are easier to comprehend. Furthermore, higher levels of abstraction make the design process tractable by reducing the number of objects involved in the design process. Design reuse, on the other hand, suggests reusing previously design circuits as much as possible in a new design, as opposed to redesigning it from scratch which is time-consuming and cost-ineffective.

Specifically, high-level design [GDWL92] refers to the task of realizing a high-level behavior of a design (written as an algorithm) with a netlist of register transfer (RT) level components. The high-level design task could be performed by a designer (manual mode) in which he/she refines a behavior into an RT level design. High-level synthesis (HLS), the automatic mode, refers to a design automation approach that synthesizes a behavior with RT level components. HLD often uses generic RT components to specify the result of the design. There is a need to map these generic RT components onto actual RT components from a library, so that the design can be physically realized through fabrication.

Design reuse manifests itself in various design scenarios, a few of which are listed below:

- **Design phase :** Typically, system design proceeds through various phases, from prototypes to large volume implementations. Initially, the designer might make a prototype of the design in order to verify the functionality of the design within its environment. Design prototypes and low-volume designs are often manufactured using programmable devices such as FPGAs. These prototype designs are often migrated to gate arrays when the volume of demand goes above a threshold. In a high-volume environment, the same design may get implemented using a custom methodology. Design reuse is applicable in all of these cases.
- **New Features :** A new design could be generated by upgrading an existing design with additional features. The functionality of the old design is enhanced with those of the new features.

- **Technology Retargetting :** Newer technologies with smaller feature sizes result in circuits with better performance and smaller area. Hence there is a need to migrate existing designs to newer technologies. In some instances, the older libraries may no longer be supported, forcing the need for technology or library migration.

In all these scenarios, a designer would like to replace a component with a new component, insert a new component, or retarget the whole design onto components from a new library, as opposed to redesigning it from scratch which is time consuming and cost ineffective. With the growth in design complexity, a need for design reuse techniques at higher levels has emerged. Current day design methods build and store complex parts in libraries; techniques to reuse these parts need to be developed.

The system design process usually refines an input behavior into a netlist of components from a library. *Component libraries* are essential for design refinement; a well-characterized library plays a pivotal role in establishing a design methodology. For example, existence of well-defined logic level libraries has been instrumental for the acceptance of logic level design in the design community. Moving the designs to higher levels of abstraction necessitates raising the libraries to higher levels as well. There is a need for characterizing the libraries at higher than the logic level.

Library mapping refers to the task of transforming a design using cells from one library to a design using cells from another library. Library mapping is required for coupling the output of a synthesis tool to technology libraries as well as for supporting design reuse by retargetting a design across different libraries. Library mapping at the logic level has been the subject of active research for quite some time; well-defined library mapping techniques exist at the logic level. With the trend towards higher levels of abstraction, there emerges a need for higher-level library mapping techniques, where the mapping is conducted at levels higher than the logic level, such as RT-level.

In this thesis, we present a novel library mapping technique that supports design reuse at RT level as well as links the output of HLD to real RT libraries. **High-level Library Mapping** (HLLM) maps a RT level component onto another RT level component of the same complexity. Our approach can *reuse* components coming from various sources such as standard libraries, datapath generators and handcrafted components. HLLM can also realize the RT netlist generated by *high-level design* by mapping each component in the netlist onto one or more RT components from a physical library.

1.2 Thesis Contributions

In this dissertation, we claim to make the following contributions:

- **Novel RT level library mapping technique :** We present a novel library mapping technique at RT level based on the functional behavior of RT level components and based on a reusable generic library of RT components. A distinguishing feature of our approach is that we can map a RT level component onto other RT level component of the same complexity.
- **Efficient formulation for a datapath component :** We define HLLM for a representative datapath component (an ALU) and present an efficient HLLM formulation based on dynamic programming. Experimental results establish the comprehensiveness as well quality of designs produced by our approach for regularly structured datapath components.
- **Efficient formulation for Memory :** We also present an efficient formulation for memory mapping based on domain specific knowledge for memories. Experimental results demonstrate the efficacy of HLLM for memories as well.

1.3 Thesis overview

The rest of thesis is organized as follows.

- **Chapter 2: High Level Library Mapping**
  We first briefly describe the library mapping techniques at various levels and then introduce High-level library mapping (HLLM), a novel library mapping technique. Next, we compare and contrast HLLM with other library mapping approaches and describe the domain and design scenarios supported by HLLM. Finally, we describe the three subtasks for HLLM.
- **Chapter 3: RT Component Libraries**
  We first motivate the need for an RT library and then describe a generic RT library. We then present results to demonstrate its comprehensiveness and effectiveness.
- **Chapter 4: HLLM for ALU**
  We formulate HLLM for ALUs and present a dynamic programming algorithm to perform ALU mapping. We also describe experimental results to demonstrate its comprehensiveness and effectiveness.
- **Chapter 5: HLLM for Memory**
  We formulate HLLM for memories and present an efficient algorithm to perform memory mapping. We also describe experimental results to demonstrate the efficacy of HLLM for memories.
- **Chapter 6: GENUS and HLLM Environment**
  We first describe the GENUS environment that implements the generic library presented in Chapter 3 along with a set of model generators and technology projectors. Next we describe a user-environment for the HLLM system.
- **Chapter 7: Conclusion**
  We summarize the results and contributions of this dissertation, and conclude with future directions for this research.

Chapter 2
High-Level Library Mapping

2.1 Library Mapping

Library mapping refers to the task of transforming a design with cells of one library to an equivalent design with cells from another library. For example, library mapping can transform a combinational circuit built using AND, OR and NOT gates to a circuit that uses only NAND gates. Figure 2.1 shows an example that transforms a design with AND and OR gates to an equivalent design with NAND gates. Figure 2.1(a) shows the source library cells, source design and its function, whereas Figure 2.1(b) shows target library cells, target design and its function. The source design could use technology independent generic cells or cells from a specific vendor library. However, the target cells are usually from a technology library, the reason why library mapping is often refered to as technology mapping.

Library mapping enables a design process to be broken into two phases: technology independent optimization and technology dependent optimization. The technology independent optimization phase generates a design using generic components void of any technology information. The generic design then can be mapped to a technology library using library mapping. This delayed binding to a specific library helps in retargetting the same design onto different technologies, thus supporting design reuse. Moreover, the technology independent optimizing tools can be developed independent of variations in technologies over time as well as across different vendors giving the tool longer a lifetime.

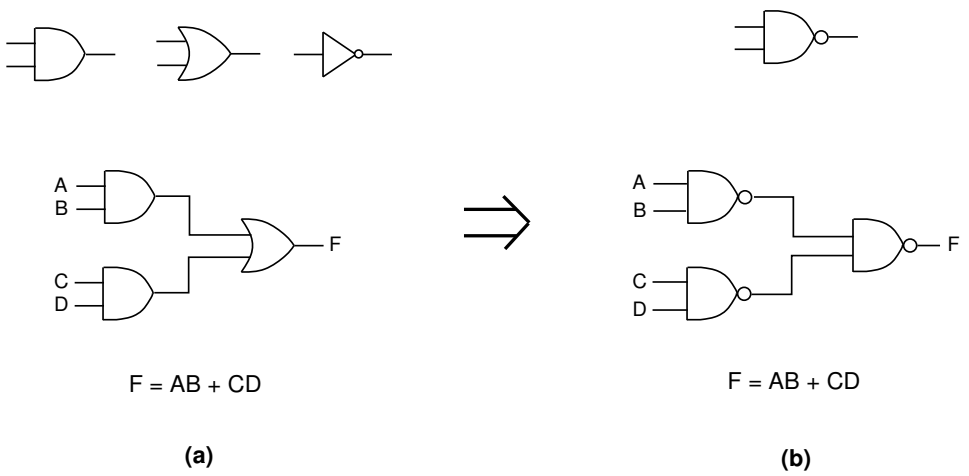


Figure 2.1: Library mapping example (a) Source design (b) Target design

Based on the complexity of cells used in mapping, the library mapping could be categorized into different levels. At the logic level, library mapping implements a logic level design using logic level cells from a library. Similarly, library mapping could be applied on RT level by implementing a RT level design with RT level or lower level components from a library. This mapping scheme could be applied all the way up to system level where the behavior of a system is mapped on to system level parts. Next we briefly discuss library mapping techniques at each level with some examples.

2.1.1 Logic level library mapping

Logic level library mapping (LLLM) has been very actively used in the design process. The existence of very well-defined logic level cells, both technology specific as well as technology independent ones, has helped in developing various library mapping techniques at logic level. LLLM is defined in terms of a source design with logic level cells (generic or technology specific), a logic level target cell library and an optimizing cost function (usually area or delay). Given the logic level source design, the aim of LLLM is to transform the source design into

a functionally equivalent target design using target cells that performs well with respect to the user given cost function. Figure 2.1 shows an example of logic level library mapping.

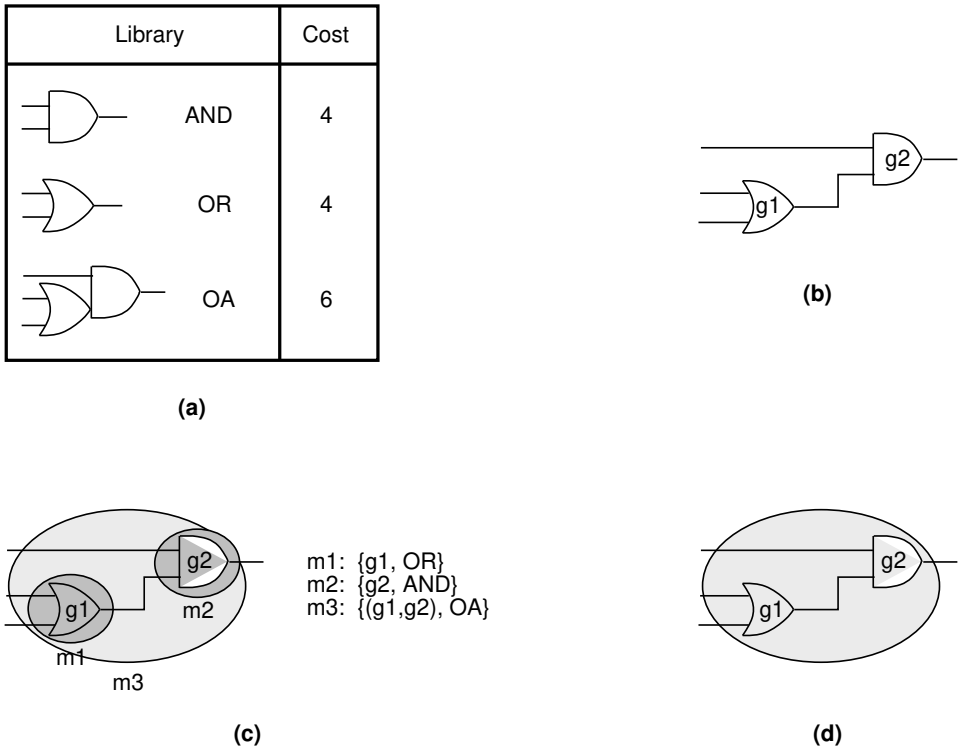


Figure 2.2: An example for logic level library mapping (a) Target library (b) Subject graph (c) Match set (d) A cover

The logic level library mapping typically applies these four steps in order to transform a source design to a target design:

Decomposition breaks the source design into base cells such as two-input gates (e.g., AND and OR).

Partitioning extracts combinational portion of the circuit and partition the circuit into smaller pieces called subject graphs.

Matching finds templates in subject graphs that are functionally equivalent to a target cell.

Covering replaces templates in the subject graph with matching library cells to complete the mapping.

Figure 2.2 shows an example of a complete match set and a cover for a subject graph. Note that there are three matches in this example, one using each library cell as shown in Figure 2.2(c). The cover shown in Figure 2.2(d) represents the area optimal cover for this example.

The logic level library mapping could be broadly classified into two groups : heuristic algorithms and rule-based approaches [Mich92]. Heuristic algorithms themselves could be classified based on techniques to solve the matching problem. In the Boolean approach [MaMi93] [STMF90], the library cells and the subject graphs are described by Boolean functions. In the structural approach [Keut87] [BRSW87], graphs representing algebraic decompositions of Boolean functions are used instead. The rule-based approach[DJBT94] uses a set of rules to transform a source design into a target design. [Mich92] discusses each of these approaches in detail.

The library mapping technique for for field programmable gate arrays (FPGA) also fall into logic level library mapping category, since these techniques uses logic level properties of the source design. However, in contrast to traditional LLLM, FPGA libraries are represented implicitly. This is because FPGAs cells are capable of performing a huge set of functions, often all functions for a fixed number of inputs; thus we can not apply LLLM techniques discussed above. We briefly describe a technique for look-up table (LUT) based FPGAs. The library mapping for LUT based FPGAs first partitions the subject graph into subgraphs satisfying the input-output constraints of the target FPGA cells, followed by covering each subgraph with an FPGA cell. Figure 2.3 shows an FPGA mapping example using cells that can perform any function for 4-inputs. Note that the cover uses two cells. [Mich92] details library mapping approaches for two classes of FPGAs.

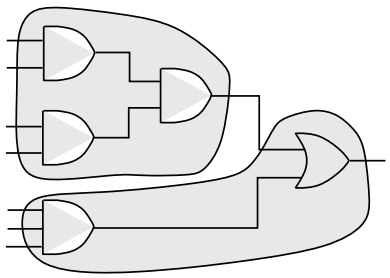


Figure 2.3: An example for FPGA mapping

2.1.2 RT level library mapping

RT level library mapping maps a source RT level component or design onto a target library. We can classify RT level mapping into three approaches, based on the levels of building blocks used to realize the source component :

- **Logic-level Mapping** An RT component's functionality can be described using Boolean equations for the transformation of the inputs into outputs. These equations can then be mapped to logic level cells using logic-level library mapping techniques discussed above. For example, the ALU in Figure 2.4 can be described with Boolean equations for each output (O0, OCOUT and OZERO) in terms of the inputs I0, I1, ICIN and C. Each of these equations can be mapped to components from a logic-level technology library using an LLLM technique.

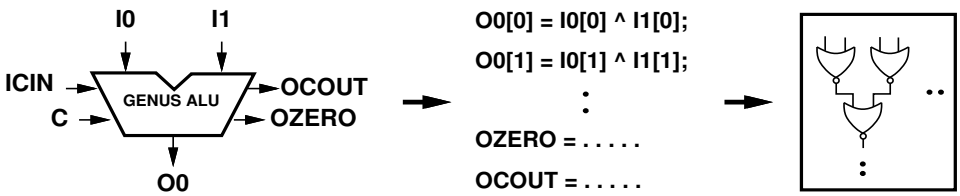


Figure 2.4: Logic-level mapping of an ALU

At the logic level, we have well-characterized primitive cells and technology mapping provides good results for small and random logic designs. As soon as

the complexity of the circuit grows, the run-time of logic level tools becomes prohibitive. [CaTr89] presents an investigation of the relationship between logic-level and high-level synthesis and presents some basic tradeoffs. It is commonly known that designs produced by logic synthesis for regularly-structured datapath components are often of poor quality, indicating the need to apply mapping techniques at higher levels of abstraction. MILO [VaGa88] is one approach that combines logic-level mapping techniques with microarchitectural optimization to realize a netlist of RT-level components.

- **Functional Decomposition** A RT-level regular-structured datapath components can be mapped to MSI-level blocks from a technology library. Each component can be functionally and/or structurally decomposed into smaller building blocks based on well-defined techniques for building datapath components of larger sizes. For instance, an ALU can be implemented as separate AU and LU blocks that are MUXed at the output. Alternatively, an ALU can be built using replicated bit-slices of one-bit ALUs.

  The choice of such construction schemes leads to a design space of alternative implementations, where the RT-level component is represented as a hierarchical tree of alternative decompositions using library primitives. The root of the tree represents the source component (i.e., the one to be mapped), while leaves of the tree consist of the MSI/SSI-level blocks from the technology library. Figure 2.5 shows a sample decomposition tree for an ALU. This ALU is realized by composing the leaf cell blocks (such as 4-bit adders, FAs, MUX2, gates) from a technology-specific component library. The DTAS system [Kipp91] and [BrMR93] follows this mapping approach. The functional decomposition approach is useful when the target component bit widths are much smaller than that of the source component.

- **High-level Library Mapping** High-level Library Mapping (HLLM) can be viewed as a source-to-target component mapping approach where the source and target components have overlapping functionality and are of approximately equal size and complexity. In this approach, the source component

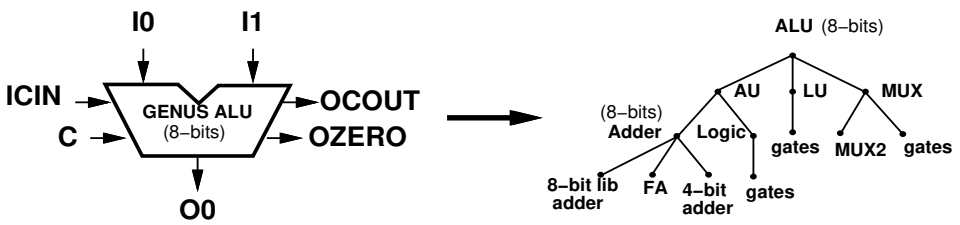


Figure 2.5: Functional decomposition of an ALU

is implemented using the target component, with a minimal amount of glue logic to satisfy the design constraints. We will discuss the high-level library mapping for RT component in more detail later in this chapter.

RT library mapping techniques have also been integrated in some of the RT-level and high-level synthesis works. These works combine two steps of the high-level design process (design synthesis and library mapping), and provides a direct approach to the reuse of RT-level library components. Although this approach can yield good results for a specific library, it requires a lot of effort in tuning the synthesis and refinement tools to accommodate variances in RT-level technology components. Furthermore, changes in the RT-library may necessitate a complete rewrite of the core synthesis algorithms that implement HLS with RT-level components. [Marw93] [AnDu94] [RuGB93] [GCDM93] [LoWM93] follow this strategy.

2.1.3 System level mapping

At the system level, mapping can be performed between system-level components such as processors, memories and interface units. MICON [BiGS89] is one approach that tries to reuse off-the-shelf system-level parts such as processors, memories and peripherals to build a single-board computer system. The input to MICON is a set of system-level specifications that describe the functionality of the required computer in terms of the type of processor, amount and type of memory,

etc., along with the design constraints (board size, cost, etc.). MICON generates a design (netlist of the above components) that satisfies the requirements given to the system.

In this work, we focus on library mapping techniques at RT level. Specifically, we present High-level library mapping (HLLM), a novel library mapping technique at RT level. Next we define High-level library mapping for RT components and mention its distinguishing features. We then compare HLLM with logic level library mapping and illustrate the domain of HLLM along with design scenarios supported by HLLM.

2.2 High-Level Library Mapping

High-level library mapping is a library mapping technique based on the higher level functional behavior of components. In this work, we concentrate on HLLM for RT level components. RT components are often very regular in their structure; human designers and some customized layout generators (e.g., datapath compilers) make use of the regularity of these components and can generate highly optimized designs. Since these designs are usually repetitive in nature, designers first optimize the base cells and then replicate the base cells to generate the complete design. For example, for generating a ripple carry adder, we first generate the optimized layout for a full-adder cell and then abut this full-adder cell n-times to generate an n-bit adder. Similarly, a memory module is designed by first optimizing a single-bit cell and then replicating these cells. A general purpose synthesis tool (e.g., traditional logic synthesis tools) can not make use of such domain specific knowledge; designs generated by these tools are of either inferior quality or require prohibitively longer run-time. Our experimental results show that for moderately sized ALUs, the designs generated by the traditional logic synthesis tools are inferior with respect to all the three design metrics, namely area, delay and runtime. High-level library

mapping provides a mechanism to use these highly optimized RT components from a library.

A distinguishing feature of HLLM for RT components is that it maps a RT component onto other RT component(s) of the same complexity. This is in contrast to logic and other RT level library mapping techniques that break down the source RT component into lower level library cells. Decomposing the source component into lower level blocks may not result into good designs, specially if the library contains highly optimized higher level parts. HLLM tries to use these higher level parts that are of the the same complexity as of the source component.

The target components differ from the source component in terms of port names, size, or set of RT functions they perform. In the HLLM approach for RT components, we study differences between the source and target components with respect to size, ports and functionality and then formulate a scheme to implement the source component behavior with target component and some glue logic. The glue logic accounts for the differences between the source and target component behaviors. The goal of high-level library mapping is to use the target component functionality as much as possible with a minimal amount of glue logic.

RT components have well-defined behavior and can be described very concisely using RT level functions. [Kipp91] shows that the functional description of RT components are smaller as compared to lower level description using boolean logic by orders of magnitude. Furthermore, functional descriptions are independent of the size of the component; the description does not increase with the size of the component. For example, the functional description size of an ALU performing two functions namely ADD and SUB is independent of the bit-width of the ALU. This is in contrast to a logic-level description of an ALU, whose size increases linearly with the bit-width for a ripple carry implementation of the ALU. Similarly, the functional description of a memory component is independent of its word-count and bit-width. High-level library mapping uses this functional description to compare and contrast the source and the target components and finally to perform the

mapping. Since HLLM is based on the functional description, its run-time usually independent of the size of the component.

2.2.1 HLLM versus LLLM

Figure 2.6 summarizes the salient features of high-level library mapping and compares HLLM with logic level library mapping (LLLM). Specifically, the figure compares HLLM and LLLM with respect to the source component domain, the target component domain, base function used for mapping, the approach itself and the application domain :

	LLLM	HLLM
Source comp	Any circuit described with logic level gates	RT level complex comps e.g., ALU, Memory mod.
Target library	Simple logic level cells	RT level complex comps e.g., ALU, Memory mod.
Base function	2–input logic function e.g., NAND, NOR	RT functions e.g., ADD, SUB, READ
Approach	Replacement of a pattern with library cells	Implementation of a RT fn from another RT fn
Applicability	Small random logic	Datapath components

Figure 2.6: Comparing HLLM with LLLM

- **Source component** Logic level library mapping is targeted towards combinational circuits described with logic equations. High-level library mapping is targeted towards towards RT level datapath component such as ALUs, memories, etc.
- **Target library** The target library in LLLM consists of simple logic level cells such as NAND, NOR, etc. In contrast to LLLM, the target components

in HLLM approach are once again RT components of the source component's complexity (e.g., ALUs and memories).

- **Base functions**

  High level library mapping is based on a higher level of abstraction in terms of using RT level functions based on 2's complement arithmetic. The base functions in LLLM consists of simple 2-input logic functions such as AND, OR, etc.

- **Approach**

  The mapping process in LLLM uses simple 2-input logic functions such as NAND, NOR, etc. Our approach uses RT level functions such as ADD, SUB, READ etc. The basic mapping approach in LLLM is to replace a pattern in the design with a library cell with an equivalent behavior. HLLM implements a source RT function using RT functions from the target library.

- **Applicability**

  Finally, because of the above characteristics, logic level library mapping is suited for small random logic. High level library mapping, on the other hand, is suitable for regularly structured datapath components such as ALUs, memories etc.

In summary, high level library mapping is based on a higher level of abstraction in terms of using RT level functions as compared to the Boolean functions used by the logic synthesis approach.

2.2.2 Domain of HLLM

High-level library mapping is applicable to RT level designs, where the basic architectural model is the control unit and datapath with memory, representing a finite state machine with a datapath (FSMD) [GDWL92]. Figure 2.7 shows the three major components of an RT design : a datapath to perform RT operations,

a memory unit to provide major storage requirements and a controller to sequence the activities of the datapath and the memory unit. The datapath consists of combinational components that perform arithmetic, logic and comparison operations and sequential components to provide temporary storage capability. The memory unit has a set of memory modules that provide foreground as well as background storage capability.

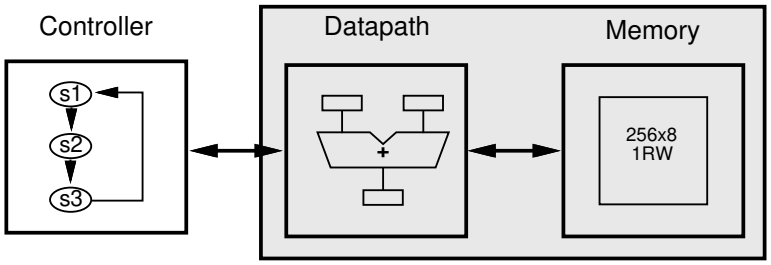


Figure 2.7: Domain of High Level Library Mapping

As illustrated by the shaded region in Figure 2.7, HLLM is well-suited for components in the datapath unit as well as memory unit. The controller is typically built using random logic; hence traditional logic synthesis tools work well for the controller. However, datapath components such ALU, counters, shift-registers are regularly-structured components that could be hand-optimized. Similarly, memory modules are also typically hand-optimized and placed in the library. HLLM is therefore suitable for components in the datapath and the memory units. HLLM will provide good results for regularly structured components with relatively larger sizes.

2.2.3 Design scenario with HLLM

Figure 2.8 shows the design scenario supported by high-level library mapping approach. Given an RT level design, HLLM maps this design using RT level components from a library and generates a library specific design. Specifically, it considers each component in the RT level design one by one and then maps the

component onto one or more components from the given library. The final design consists of a netlist of library specific components.

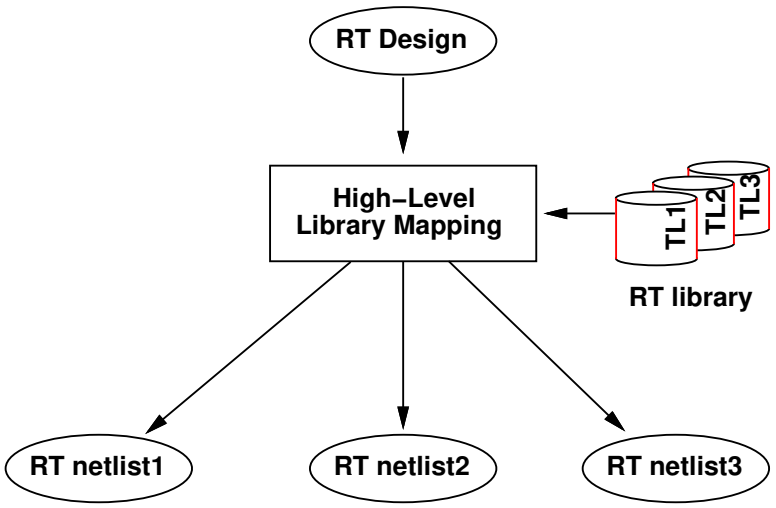


Figure 2.8: Design scenario with HLLM

The above design scenario supports the two current day design methodologies: *high-level design* and *design reuse*. High-level design generates a RT level design as its output. HLLM can map this RT design onto components from a real library. HLLM also supports design reuse by retargetting a RT design across various libraries. Given an RT design (using components from one library), HLLM can retarget the design onto another library by mapping each RT component in the design onto RT components of the target library, thus achieving design reuse. In summary, by providing a library mapping scheme at RT level, HLLM facilitates current day design methodologies using high-level design as well as design reuse.

2.2.4 HLLM tasks

In order to perform HLLM, we need to do the following :

1. We need to define a library of RT level components that will provide a reference point for comparing the source and target components and allow mapping of the source component behavior on the target component. The library should be representative of existing RT level component libraries and provide precise semantic definitions for each component.
2. We need to formulate the HLLM approach for the datapath components shown in Figure 2.7.
3. We need to formulate the HLLM approach for the memory modules shown in Figure 2.7.

In the following chapters, we will address each of these tasks and present our formulations.

2.3 Summary

In this chapter, we introduced library mapping and summarized library mapping techniques at various levels. Specifically, we summarized the library mapping techniques at logic level, RT level and system level. We then introduced high-level library mapping, a novel library mapping technique at RT level. We compared high-level library mapping with logic level library mapping and defined the domain of high-level library mapping. We also demonstrated how HLLM design scenarios can support high-level design and design reuse methodologies. Finally, we listed the three problems that need to be solved in order to perform the HLLM task.

Having given a general overview of HLLM, we next address each of the three subtasks for HLLM problem. In the next chapter, we define a generic RT library that forms the basis of HLLM. Next we apply HLLM to two classes of components. Chapter 4 describes HLLM formulations for a representative datapath component,

the ALU. Chapter 5 describes HLLM for memories. These two formulations along with a generic library definition completes the overall HLLM formulation.

Chapter 3

RT Component Libraries

Component sets and libraries play an important role in the present-day design methodologies that use schematic capture as well as high-level design. These methodologies output designs that are interconnections of components drawn from a vendor's library. The components can vary in their level of complexity from simple logic gates, to sequential components such as counters and registers, to arithmetic blocks such as ALUs, and all the way up to complex components such as CPU cores. However, the register-transfer (RT) level is a common design entry point that is supported by most of the existing CAD tools on the market. The RT-level has had a long history of use as a design entry point, as evidenced by the frequent use of TTL databook component names by designers, as well as in digital system design courses outlined in standard textbooks and taught at universities. We also note that most data sheets for product specifications (either being designed, or after they have been designed) are often composed of register-transfer schematics typically drawn up by system level designers.

A well-defined component library is also critical for the successful realization of a synthesis tool. We typically use *generic* components to specify the input or intermediate results of synthesis, and follow with a phase of *technology mapping* to realize the design with a set of components from a technology library [GDWL92]. For instance, logic synthesis uses generic components such as simple logic gates (e.g., AND, OR, INVERT) at the input and for intermediate synthesis steps, but the last step of logic synthesis involves technology mapping of the generic design

into components drawn from a technology library (e.g., complex CMOS gates, or a different logic gate family such as NOR-NOR)[Mich92]. Generic component sets facilitate technology independence, and allow the capture of a design in a standard form that can be retargetted to different libraries (or technologies) without changing the input description. Of course, technology independence needs to be coupled with good technology mapping strategies that can effectively map generic designs to target library components with low overhead.

High-Level Design (HLD) also relies on a library of well-defined, parameterized RT component generators to simplify the mapping of behavioral variables and operators to physical components. This mapping of the abstract design into an interconnection of RT components involves design space exploration by selecting and allocating a proper set of RT components, guided by design metrics (e.g., area and delay). Each component is customized by parameterized attributes such as the required bit-width and functionality. A well-characterized RT library also serves as a repository of reusable parts and thus supports design reuse. A design using components from such a library is well-suited for retargetting across various technology libraries using library mapping. Similar to logic level library mapping, a well-defined RT-level component set would form the basis for RT level library mapping.

Although RT-level components are commonly used in specifying, documenting, refining and synthesizing designs, there is a lack of standardized RT component sets that can facilitate unambiguous documentation, communication and design reuse. The existing RT-level vendor libraries in the domain of gate-arrays [Tosh90], module generators [Casc92][VTI91] and FPGAs [XBLO92] provide a limited coverage for the RT component domain and are tuned to specific backend technologies. There has been an initial work by Dutt [Dutt88] that defines a set of RT-level generic components. Recently there has been another effort spearheaded by the FPGA community (LPM)[LPM93] to characterize RT-level generic components. This is in contrast to the logic-level, where the designs can be expressed

as netlists of well-understood standard components such as the equivalent 2-input NAND or NOR gate. [CaTr89] [Wolf89] are representative of approaches that provide component characterization and module databases at the layout and logic levels.

With increasing interest in high-level design methods, the need has thus evolved for a well defined generic RT component set, along with estimators and mappers to allow technology projection and mapping respectively into different libraries. In this chapter we briefly describe an RT-level library of reusable parts and discuss its comprehensiveness by comparing it with various technology libraries. We then present the results of experiments on some high-level synthesis benchmarks to evaluate the amount of overhead incurred by using generic components. This library provides the basis for high-level library mapping.

3.1 Generic RT Library Definition

We have defined a generic RT library based on the GENUS library described in [Dutt88] and an examination of commonly used RT level parts. The library contains a set of parametrized component generators based on GENUS [Dutt88], and expanded further to make it comprehensive. These component generators are defined using RT-level functionality and are grouped into classes based on functional similarity. A component instance is generated by specifying the parameters for a corresponding generator. For example, an ALU generator is characterized by the following parameters: *(bit-width, set-of-functions, implementation-style)*, whereas a specific ALU instance is generated by specifying values for these parameters. The grouping of similar components into classes of generators makes the task of library management simpler and more efficient since the resulting number of generators (approximately 50) is much less than the virtually infinite number of possible component instantiations.

Besides the parameters that are used to instantiate a specific component, each generator is characterized by a well-defined interface and associated semantics. Components derived from a generator can perform a specific set of RT level functions and each generator's specification includes the set of these functions. RT-functional mappings specify the exact relationship of each output with respect to the inputs.

The generic RT library components could be broadly classified into three groups: combinational, sequential and miscellaneous. Next, we briefly describe components in each group; [JhD94a] provides detailed description each of these components including their semantic definitions.

3.1.1 Combinational components

A combinational component directly manipulates input data; it either performs a computation or routes the input data to the output. Combinational components do not have have any storage capability; their functionality could be described using Boolean equations.

Table 3.1 and 3.2 show the list of generic combinational component generators, along with their set of functions and parameters. The ALU performs the five arithmetic, six comparison and sixteen logical operations. The *adder-subtractor* component performs three functions: ADD, SUB and RSUB (Reverse subtract). The *logic unit* can perform all the sixteen logic functions on two operands; the *comparator* can perform 6 comparison functions. The *multiplier* and *divider* are self-explanatory. The *barrel-shifter* and *shifter* shifts the input data; the barrel-shifter is more general in the sense that it has another input that specifies the number of bits to be shifted. The *multiplexer* and *selector* route one of the input data to the output based on the control signal. A *parity generator* component can generate two types of parities : odd and even. The library also includes single output as well as bitwise *logic gates*. The *concat* and *extract* components are used

Name	*Functions*	*Parameters*
ALU	5 arith, 6 comps 16 logic fns	input-width, style, num-fns, fn-list
Adder-subtractor	ADD, SUB, RSUB	input-width, style, num-fns, fn-list
Logic unit	16 logic functions	input-width, num-fns, fn-list
Comparator	EQ, NEQ, GT, LT GEQ, LEQ	input-width, num-fns, fn-list
Multiplier	*	left-input-width, style, right-input-width
Divider	/	input-width, style
Barrel-shifter	SHR, SHL, ROTL, ROTR	input-width, num-fns fn-list, shift-distance
Shifter	SHR, SHL, ROTL, ROTR	input-width, num-fns, function-list
Mux	Select input i	num-inputs, input-width
Selector	Select (on guard value)	input-width, num-guards guard-list, else-flag control-width
Parity generator		input-width, type
Logic gates (Single output)	GAND, GXOR, GNOT, GNOR GOR, GNAND, GXNOR	input-width
Bitwise logic gates	AND, OR, NAND, NOR XOR, XNOR	num-inputs, input-width

Table 3.1: Generic combinational components

Name	*Functions*	*Parameters*
Concat		num-inputs, input-width-list
Extract		input-width left-index, right-index
Decoder		input-width
Encoder		input-width

Table 3.2: Generic combinational components (continued)

to concat a set of inputs into the output and to extract a set of bits from the input data respectively. Finally, the *decoder* and *encoder* are used to perform binary decode and encode respectively. [JhD94a] provides a detailed information for each of these components.

3.1.2 Sequential components

Sequential components provide storage capability; their operations are usually activated by a clock signal. In addition to providing the storage capability, these components may perform some computation. Table 3.3 shows the list of sequential components in the generic library. The *register* component can store an input data and shift the data to the left or right. The *counter* component can count up or down, besides performing a synchronous load operation. The *stack* and *FIFO* are used to store and access the data in a specific order. The *register-file* and *memory* have the same generic structure and are used to store large amounts of data.

Name	*Functions*	*Parameters*
Register	LOAD, SHL, SHR	input-width, num-fns, fn-list, type, invert-out, set-value
Counter	LOAD, COUNT-UP, COUNT-DOWN	input-width, num-fns fn-list, type, style
Stack/Fifo	PUSH, POP	input-width, num-words
Register-file/ Memory		input-width, num-words, num-input-ports, num-output-ports, num-inout-ports

Table 3.3: Generic sequential components

3.1.3 Miscellaneous components

Table 3.4 shows the list of components that perform miscellaneous operations. The *buffer* and *tristate* are interface components. The *bus* and *wired-or* components are similar, except that the *bus* component has tristate drivers at each input of the bus. The *clock-generator* generates a clock with its duty cycle specified by the clock-period and clock-high parameters. The *oneshot* component generates a pulse of the specified width. Read-only memories (*ROMs*) are used to store a list of constants. The *delay* component delays the input data by a specific duration. The *port* component models the design's ports that are used to encapsulate the design and communicate with other designs. Finally, the *black box* is a generic component that a user can customize to define a component not included in this library.

The generic components along with their functional behavior could serve as a RT level intermediate form for design documentation, exchange as well as library mapping. We could use these definitions of RT components as the RT-level counterpart for the logic level generic functions defined by 2-input NAND and

Name	*Functions*	*Parameters*
Buffer		input-width
Tristate		input-width, invert-flag
Bus		input-width, num-inputs
Wiredor		input-width, num-inputs fan-out
Clock generator		clock-period, clock-high
Oneshot		pulse-width
ROM(constant)		input-width, num-words, num-output-ports, core-file
Delay		input-width, delay-value
Port		input-width, mode
Black box		comp-key

Table 3.4: Generic miscellaneous components

NOR gates. We use generic component definitions as a basis for high-level library mapping to compare and contrast the source and target RT components as well as to perform mapping between them.

Although generic components are appealing in concept, we have to address some important issues while defining a standardized generic component set. First, how comprehensive is the generic library set? That is, how well do generic components cover various components across different technology libraries? Second, how much penalty do we pay in using generic components as compared to directly using the technology components? Next we discuss these two issues.

3.2 Generic component coverage

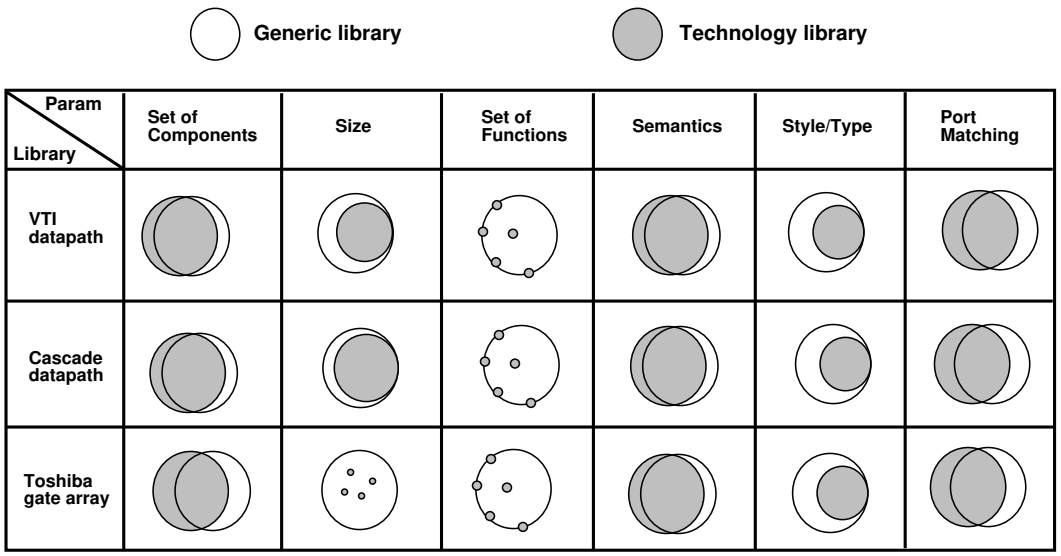


Figure 3.1: Coverage of generic components across different libraries and parameters

We present the results of our survey of library coverage with respect to different technology libraries that use varying layout styles for component implementation. In particular, we examined the following layout styles: Standard cell, Bitslice, Gate array and Field programmable gate array (FPGA). The first two layout styles typically result in more compact designs at the cost of longer design cycles, while the gate array and the FPGA styles provide a quick method for prototyping designs. We considered the following technology libraries in our survey: VTI Datapath Compiler [VTI91], Cascade Digital Library [Casc92], Toshiba Gate Array Library[Tosh90] and the XBLOX FPGA library [XBLO92]. We summarize the results here; further details can be found in [JhDG93].

Figure 3.1 pictorially illustrates the coverage of generic components across various parameters relative to different technology libraries. [JhD94a] contains a description of the set of parameters associated with various components. Each column in Figure 3.1 represents a parameter type or component attribute and each row shows a technology library. In this figure, the white and shaded circles represent the domain (with respect to a specific parameter) of generic and a target

library respectively. The size of these circles represents the domain size; the relative position specifies the domain overlap. For example, the entry in the first row and the first column specifies that the generic library and VTI cover approximately the same number of components and that most of the components are common between the two libraries. On the other hand, the entry in the first row and third column specifies that VTI supports a much smaller set of functions and that some of these are different from the set of functions supported by the generic library.

From the table in Figure 3.1 we observe that the technology libraries that are parametrized (e.g., Cascade and VTI) provide fairly good coverage for generic components. The Toshiba gate array library provides components of specific sizes; components of other sizes have to be built from the available components. The other major difference was the availability of a specific set of functions in realm of multifunction components. We also observed a common problem with mismatches in the port names. This survey indicates that generic components provide good coverage for several technology libraries.

3.3 Generic component effectiveness

In order to further evaluate the usefulness of generic components, we performed some experiments with various designs derived from the HLSW92 benchmark suite [DuRa92]. The goal was to test our approach on designs of various sizes, and that encompass different sets of components so as to exercise the major component types in the generic component set. In our experiments, the designs varied in complexity from a few hundred gates to a couple of thousand gates. The mapping experiments were performed with respect to two different technology libraries: the VTI Datapath Compiler [VTI91] and the Toshiba Gate Array library [Tosh90]. We chose these libraries since they had published gate counts for their databook components, thereby allowing us to compare the effectiveness of mappings for different designs.

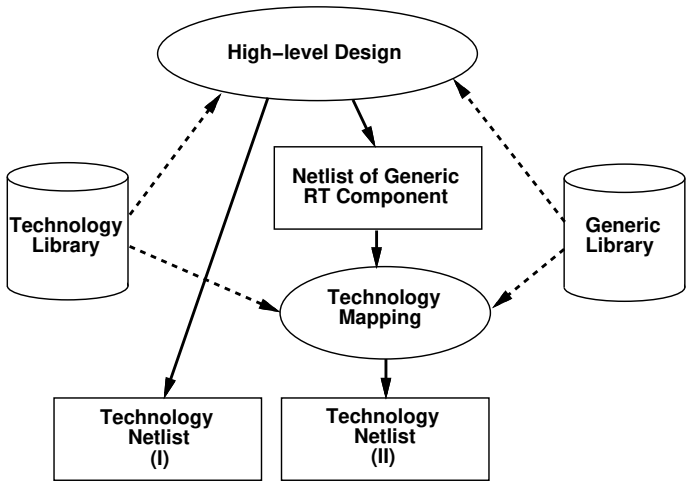


Figure 3.2: Study of overhead incurred with generic library

Design	VTI Library			Toshiba Gate Array		
	Direct (I)	Tech Map(II)	% Overhead	Direct (I)	Tech Map(II)	% Overhead
AM2901	391	435	11.25%	1523	1628	6.89%
AM2910	832	869	4.44%	1528	1557	1.89%
SRT Interface	735	747	1.63%	674	674	0.00%
CB Interface	1832	1836	0.22%	1979	2227	12.53%

Figure 3.3: Experimental results of generic component overhead study

We considered the following RT-level designs derived from different categories such as processors, DSP and interface circuits: the AM2901 Microprocessor [Am2901], the AM2910 Microprogram Controller [Am2910] an SRT Interface [Li93], and a Circular Buffer(CB) Interface [Li93].

In our experiments, we designed each of these circuits using two different paths, as shown in Figure 3.2. First, we designed the circuit using technology library components only (labeled I). Next, we designed the circuit using only generic components, and then mapped each of these generic components to the technology library components (labeled II). The goal was to examine the penalty incurred by designing with a generic component library, followed by technology mapping, as opposed to directly implementing the designs with technology-specific components. For each of these designs, we calculated the total gate count. For our experimental results, a gate is equivalent to the layout area of 4-transistors.

Figure 3.3 tabulates the gate-counts for various designs across different libraries. For each design and each technology library, we present the gate-count for the two methodologies (I, II). We also present the percentage difference (in terms of the gate-count) between implementing a component in the technology library (I) and mapping the generic component design to the technology library (II). This percentage gives a measure of how much overhead is incurred by using generic components.

In Figure 3.3, we observe that the percentage difference between the two design methodologies (I and II) varies from 0.00% to 12.53%. For the SRT interface, the two designs (I and II) are very close in gate-count. This is because the SRT circuit is fairly simple and primarily uses lower-level components such as logic gates and flip-flops that have good coverage relative to the technology libraries. The lack of complex RT components enables a very simple and effective mapping with a resulting overhead that is very low.

On the other hand, consider the mapping of the 2901 microprocessor and the CB interface to the Toshiba gate array library. There is a significant difference in gate-counts for the two methodologies (I and II). These designs use higher-level components such as ALUs and register-files which have a larger mismatch with respect to the technology library components.

Based on these experiments, we observe that not much penalty (generally $\leq 10\%$) is incurred in using generic components first and then performing high level library mapping. This supports our hypothesis that high-level mapping is feasible and practical for the designs we examined.

3.4 Summary

With increasing interest in tightly coupling high-level design techniques with physical design, the need for a well-characterized RT-library has emerged. In this chapter, we motivated the need for such generic RT component libraries, and described a specific well-characterized generic RT component library. We then surveyed the relative coverage of of this library with respect to some technology libraries and performed experiments on some high-level synthesis benchmarks for testing the usefulness of the generic library. We believe that the benefits of using a standard component set such as the one described in this chapter greatly outweigh the small penalty that may be incurred during technology mapping to target libraries. In Chapter 6, we present the GENUS environment that implements this generic RT library.

The generic library forms the basis of high-level library mapping. HLLM uses the definitions of generic components as a reference point to compare the source and target components and finally to map the source component onto the target component. In the next chapter, we formulate HLLM for ALU.

Chapter 4
HLLM for ALUs

4.1 Introduction

In this chapter we describe the high-level library mapping technique for arithmetic-logic units (ALUs). An ALU is a representative component from the datapath unit and is commonly used in RT designs. ALUs are often handcrafted and stored in libraries for future use; different versions of ALUs exist in databases or design groups within a company. Due to the regularity in arithmetic structures and the fact that logic synthesis and logic-level technology mapping techniques do not work well for regularly-structured components such as ALUs, we address the issue of ALU mapping. We consider ALUs that perform any subset of arithmetic, logic and comparison functions, using a well-defined set of operations for the ALUs. Hence it is possible to formulate mapping schemes that map an ALU onto another ALU based on the sets of functions they perform.

In this chapter, we define high-level library mapping problem for ALUs, describe an algorithms to solve the problem, and provide experimental results to validate its effectiveness. Specifically, we demonstrate the versatility of this approach by applying HLLM to ALUs drawn from different libraries. We also compare the HLLM approach versus a traditional logic synthesis approach and demonstrate the advantages of HLLM for complex datapath components.

This chapter is organized is as follows. Section 4.2 defines high-level library mapping for ALUs based on their functional behavior. Section 4.3 discusses the overall approach to the ALU mapping problem. Section 4.4 presents an algorithm to map a source ALU onto a target ALU. Section 4.5 demonstrates the comprehensiveness and quality of designs produced by our approach. The chapter concludes with a summary.

4.2 Problem Definition

The high-level library mapping problem for ALUs is based on functional specifications of the source and target components, which are compared with respect to a "canonical" functional representation to derive an effective mapping result. When the functionality of the target component does not exactly match that of the source component, the target component may need to be padded with additional (glue) logic. Figure 4.1 illustrates this high-level mapping approach between a source and a target ALU.

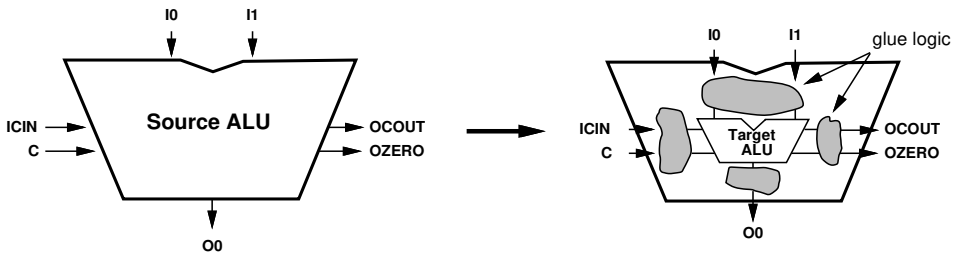


Figure 4.1: High-level library Mapping of an ALU

We therefore define the high-level library mapping problem in terms of a *Source component (S)*, a *Target component (T)* and a set of *Mapping rules (R)* that maps the source component onto the target component. In order to establish equivalence between the source and the target components, each component is described in terms of a set of RT-functions that are defined using a canonical representation of the component. A mapping rule in R describes an alternative

for implementing a function in S using a function in T; each source function can potentially be implemented by different target functions. The task of high-level library mapping, then, reduces to selecting a set of rules, one for each function in source component S, that realizes the best mapping of S on T with respect to the cost function (e.g., area or delay).

In the rest of this section we discuss our assumptions, the canonical representation used for components and functions, and the mapping rules along with cost function to be used in our approach.

4.2.1 Assumptions

Our formulations of HLLM for ALUs make the following assumptions :

- All data and arithmetic use the 2's complement representation.
- A source component can perform only one function at a time. For example, a comparator can implement several RT-functions (e.g., EQ, NEQ, GT, LT, etc.), but only one function is performed at a time.
- We restrict ourselves to arithmetic, logic and comparison functions. These functions are defined using a universal ALU that performs a set of canonical arithmetic, logic and comparison functions (as described in the next section).
- Each of the target component's RT-functions should either be a canonical RT-function or a simple negation of a canonical RT-function.
- The source (S) and the target (T) components have the same bit-widths.

These assumptions are made in order to make the problem size tractable and also to facilitate illustration of the high level library mapping approach. We believe that these assumptions can be relaxed once the basic HLLM approach is defined and well understood.

4.2.2 Universal ALU representation

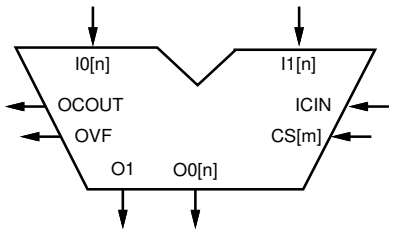


Figure 4.2: The Universal ALU

In order to provide a reference model for HLLM, we define a universal ALU (U) that performs a canonical set of ALU functions: 5 arithmetic functions (ADD, SUB, RSUB, INC, DEC), 16 logic functions (all Boolean functions of 2 variables), and 6 comparison functions (EQ, NEQ, GT, GEQ, LT, LEQ). Using these canonical ALU functions, we can build any other ALU including library-specific ALUs. The canonical ALU and its functions are based on the ALU generator definitions provided by the GENUS library. The canonical ALU functions are described in more detail later in this section.

Figure 4.2 shows an n-bit universal ALU with the following ports:

- **I0[n]:** primary left input.
- **I1[n]:** primary right input.
- **O0[n]:** primary output for arithmetic and logic functions.
- **ICIN:** carry input used by the arithmetic functions.
- **OCOUT:** carry output used by arithmetic functions.
- **OVF:** secondary output that is used by arithmetic functions.
- **O1:** comparison output.
- **CS[m]:** control input.

Function	Operation
ADD	(OCOUT, O0[n], OVF) = I0[n] + I1[n] + ICIN
SUB	(OCOUT, O0[n], OVF) = I0[n] + $\overline{I1[n]}$ + ICIN
RSUB	(OCOUT, O0[n], OVF) = $\overline{I0[n]}$ + I1[n] + ICIN
INC	(OCOUT, O0[n], OVF) = I0[n] + 1
DEC	(OCOUT, O0[n], OVF) = I0[n] - 1

Table 4.1: Canonical arithmetic functions

Function	Operation
ZERO	O0[n] = $0[n]$
ONE	O0[n] = $1[n]$
AND	O0[n] = $I0[n] \wedge I1[n]$
NAND	O0[n] = $\overline{I0[n] \wedge I1[n]}$
OR	O0[n] = $I0[n] \vee I1[n]$
NOR	O0[n] = $\overline{I0[n] \vee I1[n]}$
XOR	O0[n] = $I0[n] \oplus I1[n]$
XNOR	O0[n] = $\overline{I0[n] \oplus I1[n]}$
LID	O0[n] = $I0[n]$
RID	O0[n] = $I1[n]$
LNOT	O0[n] = $\overline{I0[n]}$
RNOT	O0[n] = $\overline{I1[n]}$
LINHI	O0[n] = $\overline{I0[n]} \wedge I1[n]$
RINHI	O0[n] = $I0[n] \wedge \overline{I1[n]}$
LIMPL	O0[n] = $\overline{I0[n]} \vee I1[n]$
RIMPL	O0[n] = $I0[n] \vee \overline{I1[n]}$

Table 4.2: Canonical logic functions

Function	Operation
EQ	$O1 = (I0[n] = I1[n])$
NEQ	$O1 = (I0[n] \neq I1[n])$
GT	$O1 = (I0[n] > I1[n])$ †
LT	$O1 = (I0[n] < I1[n])$ †
GEQ	$O1 = (I0[n] \geq I1[n])$ †
LEQ	$O1 = (I0[n] \leq I1[n])$ †

†Assumes that the data is in 2's complement form.

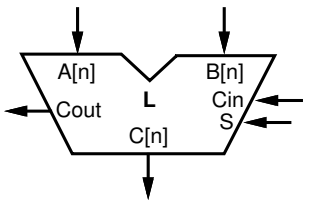
Table 4.3: Canonical comparison functions

4.2.3 Canonical ALU functions

Each canonical ALU function defines a functional mapping between the inputs and the outputs of the universal ALU. Note that an ALU function need not use all the ports of a universal ALU. Table 4.1 presents the canonical representation for 5 arithmetic functions. The arithmetic functions make use of the following ports: I0[n], I1[n], O0[n], ICIN, OCOUT and OVF. The canonical representation for 16 logic functions are shown in Table 4.2. These functions use only primary inputs (I0[n], I1[n]) and the primary output (O0[n]). Table 4.3 describes the canonical representation of 6 comparison functions. These comparison functions use primary inputs (I0[n] and I1[n]) and the primary output O1. The universal ALU therefore has a total of 27 canonical ALU functions (5 arithmetic, 16 logic and 6 comparison).

4.2.4 Representation of library components

We need a representation of library components (e.g., S or T in our problem) that not only captures the functionality but also facilitates the comparison between them. A representation based on canonical functions supports both these needs. Specifically, an ALU library component is described by its port lists and the set of functions it performs. A function in the library component is a variant of one



(a)

Source Function	Canonical Function	Input			Output		Control
		I0	I1	ICIN	C	Cout	S
$\overline{A}$+B+Cin	ADD	$\overline{A}$	AB	Cin	O0	OCOUT	0
$\overline{A \vee B}$	OR	A	B	–	$\overline{O0}$	–	1

(b)

Figure 4.3: A library ALU example: (a) Port description (b) Function description

of the 27 canonical ALU functions. It is described by providing the corresponding canonical function and the Boolean relationship between the ports of the library component and the ports of the universal component. The Boolean relation between the library component ports and the universal component ports is described in a tabular fashion. The table contains a row for each function in the library component; there is a column for each input port in the universal ALU, a column for each output port in the library component and a column for the control configuration. The control configuration lists the values of control lines for each function. An entry in this table consists of a Boolean expression showing relationship between the ports of a library component L and the universal ALU U. For example, a library component (L) that performs the functions ($C = \overline{A} + AB + Cin$) and $C = \overline{A \vee B}$ is shown in Figure 4.3(a), and the corresponding tabular representation is shown in Figure 4.3(b).

4.2.5 Logic function representation

Each logic function of the ALU is described using a standard minterm representation of two primary inputs. Note that we have four minterms with two inputs A and B, namely $\overline{A}\overline{B}, \overline{A}B, A\overline{B}$ and AB. A specific logic function selects a subset of these four minterms. As an example, the OR function is given by the following minterms: $\overline{A}B, A\overline{B}$ and AB. In other words, when one or more of these three minterms are active, output of the OR function is 1. Table 4.4 lists minterms for

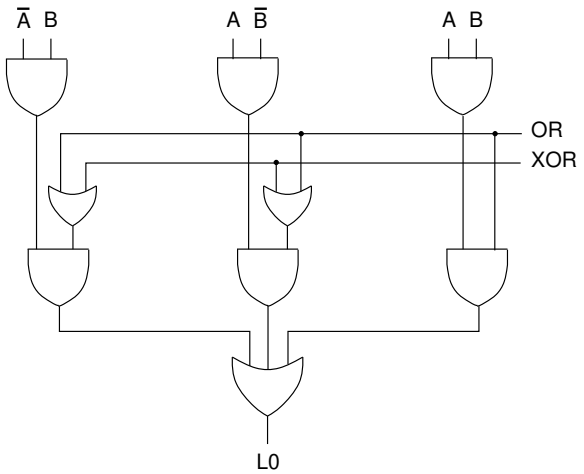


Figure 4.4: Implementation of logic unit with two functions: OR and XOR

Function	$\overline{A}\overline{B}$	$\overline{A}B$	$A\overline{B}$	AB
ZERO	0	0	0	0
ONE	1	1	1	1
AND	0	0	0	1
NAND	1	1	1	0
OR	0	1	1	1
NOR	1	0	0	0
XOR	0	1	1	0
XNOR	1	0	0	1
LID	0	0	1	1
RID	0	1	0	1
LNOT	1	1	0	0
RNOT	1	0	1	0
LINHI	0	1	0	0
RINHI	0	0	1	0
LIMPL	1	0	1	1
RIMPL	1	1	0	1

Table 4.4: Minterms for logic functions

all the 16 logic functions. A set of logic functions is implemented by ANDing the minterms for each function with the corresponding control lines and feeding the output to an OR gate. As an example, Figure 4.4 shows an implementation for two logic functions OR and XOR.

We use an ordered vector (logic vector or LV) of length 4 for representing the implementation of one or more logic functions. The first entry in the logic vector represents the first minterm ($\overline{A}\overline{B}$), the second entry represents ($\overline{A}B$), the third entry represents ($A\overline{B}$) and the fourth entry represents the last minterm (AB). For example, the logic function OR is represented by the vector "0111", since it does not need minterm $\overline{A}\overline{B}$ but requires all the other minterms $\overline{A}B, A\overline{B}$ and AB. A logic vector (LV) for a set of logic functions is obtained by adding the vectors (entry wise) for each function. Thus, the logic vector for the two logic functions OR ("0111") and XOR ("0110") is "0221".

4.2.6 Mapping rule representation

A mapping rule describes how to implement a canonical function from another canonical function. Given a set of mapping rules, we can implement all the functions in the source component including the ones that are not present in the target component. Let SF and TF be the source and the target canonical functions respectively. Let us define port names for the source and target canonical component (CS and CT) as shown in Figure 4.5. A mapping rule describes the mapping of CS ports onto CT ports such that SF is implemented using TF.

A rule is described in a tabular fashion similar to the library component representation. Table 4.5 lists some sample rules. Each row of this table contains a rule name, a source function (SF), a target function (TF), and the port mapping information. The first set of port mappings expresses the target component's inputs in terms of the source component's inputs. The second set of port mappings

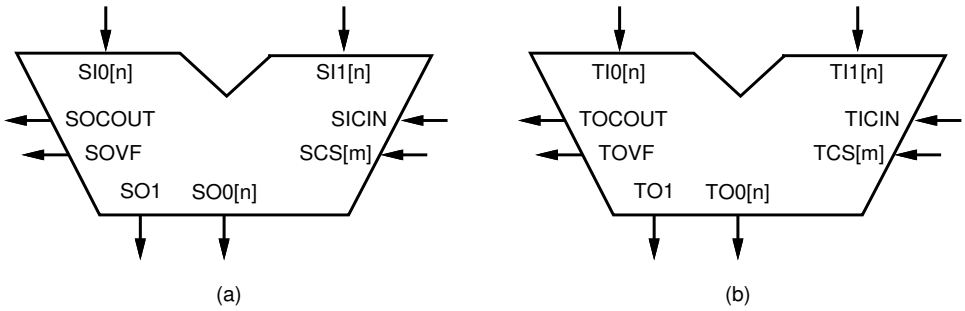


Figure 4.5: Port names : (a) Source canonical ALU (b) Target canonical ALU

describes the the source component's outputs in terms of the target component's outputs.

Each rule describes the implementation of a source function using a target function and indicates the port mappings required to implement the mapping. Note that each source function can be implemented using several alternative target functions. For example, the source *ADD* function in Table 4.5 could be implemented using target *ADD* function (rule "AA1"), or with the target *SUB* function (rule "AS1"). The input and output port entries indicate the connectivity and additional logic required to implement the mapping rule. For instance, the second rule "AS1" in Table 4.5 implements the source function *ADD* with the target function *SUB* by inverting the right input ($\overline{SI1}$).

For each source logic function, there is a rule that implements the function from scratch without using any target function. To this rule, we add another entry

Rule	*Source*	*Target*	*Input ports*			*Output ports*			
name	*fn*	*fn*	*TI0*	*TI1*	*TICIN*	*SO0*	*SOCOUT*	*SOVF*	*SO1*
AA1	ADD	ADD	SI0	SI1	SICIN	TO0	TOCOUT	TOVF	–
AS1	ADD	SUB	SI0	$\overline{SI1}$	SICIN	TO0	TOCOUT	TOVF	–
IS1	INC	SUB	SI0	"1..1"	'1'	TO0	TOCOUT	TOVF	–
EX1	EQ	XOR	SI0	SI1	–	–	–	–	nor(TO0)

Table 4.5: Sample mapping rules

Rule name	*Source fn*	*Target fn*	*Input ports*		*Output ports*	*LV*
			TI0	*TI1*	*SO0*	
ANAN	AND	AND	SI0	SI1	TO0	–
AN_	AND	–	–	–	TL0	"0001"
ANNA	AND	NAND	SI0	SI1	$\overline{TO0}$	–
XO_	XOR	–	–	–	TL0	"0110"

Table 4.6: Sample mapping rules for logic functions

in the table: a logic vector (LV) as discussed previously. The primary output for a logic rule is given by the logic output (LO). Table 4.6 lists some sample logic rules. For example, the source *AND* logic function could be implemented from scratch (rule "AN_") by adding the minterm corresponding to the LV "0001", i.e. AB. [JhD94b] contains an extensive list of rules for all the ALU functions.

4.2.7 The cost function

A good cost function captures the important characteristics of an efficient source-to-target component mapping. We use a cost function based on two criteria: (a) an area metric, represented by the *gate-count* of the hardware overhead, and (b) a delay metric, represented by the worst case delay or *max-delay* of the generated design.

Gate-count

Mapping S on T requires extra hardware that could arise due to:

- Routing data from the inputs of S to the inputs of T and from the output of T to the output of S.
- Mapping a function in S onto some other function on T.

- Generating a function (for example, a logic function) of S from basic gates.
- Mismatch between the canonical functions and the functions in S and T.
- Mapping the control lines of S onto the control lines of T.

In our current formulation, we use the *gate-count* (GC) of the extra hardware as a measure of the hardware cost. Specifically, we use the number of equivalent 2-input gates as the cost function to guide our algorithm. Note that the actual cost of a design should also include the cost of the target component. However, since this cost function is used just to compare two designs, it does not matter if we exclude the component cost in our cost function, because this portion of the cost will be present in each design. Therefore, we use the gate-count of the hardware overhead as an area optimization criterion.

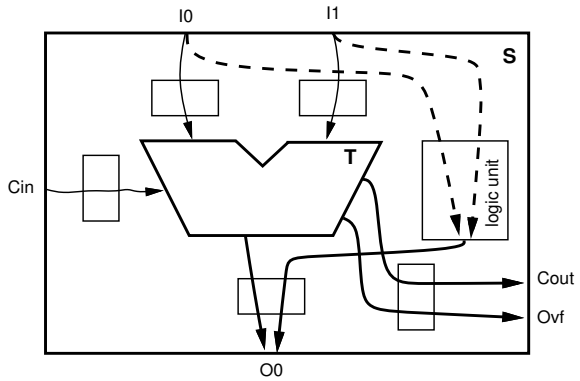


Figure 4.6: Worst case delay for an ALU design

Max-delay

We use the worst-case delay of the design as a delay metric. The worst-case delay, *max-delay* (MD), for a design is given by the maximum delay through all paths of the design. It is an approximation of the delay of the resultant design.

As an example, consider the design shown in Figure 4.6, that shows a sample source component (S) mapped to a target component (T). Let (MD_t) be the

maximum delay through the component T. We can calculate MD for the design S in the following way:

1. Calculate the max-delay to the inputs(MD_i) of T. It is given by the maximum of delays through all the paths shown by light lines in Figure 4.6.
2. Calculate the max-delay to the output(MD_o) of T. It is given by the maximum of delays through all the paths shown by thick lines in Figure 4.6.
3. Calculate the worst case delay for logic-circuit(MD_l) of T. It is given by the maximum delay through all the paths shown by shaded lines in Figure 4.6.
4. The MD of the design, then, is given by maximum of the two figures:

$$Max(MD_i + MD_o + MD_t, MD_l + MD_o)$$

Note that unlike the previous area metric (gate-count) calculation, we cannot ignore the delay of the target component T. This is because MD for all the designs may not include the delay of T; the worst case path might pass through the logic unit.

4.3 ALU Mapping Approach

Figure 4.7 illustrates our overall approach to the ALU mapping problem. The inputs to the system consist of the source component (S), the target component (T) and the mapping rule database (R). As mentioned before, both S and T are library components and they are described as source canonical and target canonical components using the representation discussed in the previous section. The mapping rule database (R) contains all the rules required to map one RT-function onto another RT-function. The rule database is also represented in a tabular fashion. In the first step, the mapping algorithm implements the source component using the target canonical component. In the second step, the target canonical function is mapped to the actual target component. The output of the system is

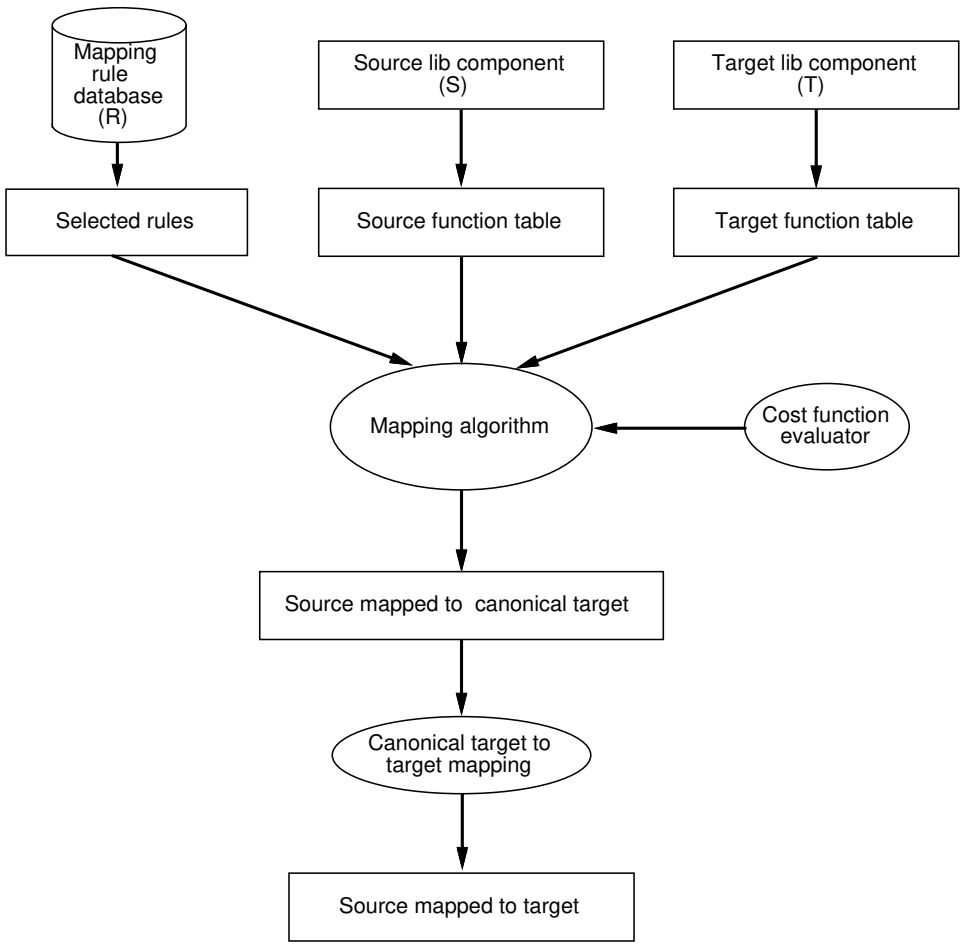
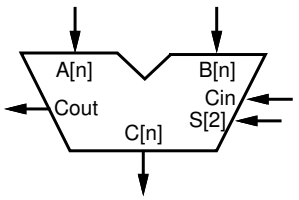


Figure 4.7: Overall system for ALU mapping

an implementation of S on the target component T with some additional (glue) logic surrounding T. We focus primarily on the first mapping step and describe a mapping algorithm for it. The second step consists of simple tasks such as the matching of port names and is relatively trivial.

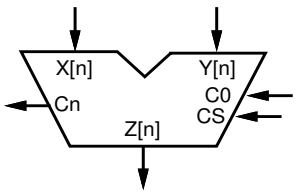


(a)

Function	Input			Output		Control	
	I0	I1	ICIN	C	Cout	S[1]	S[0]
ADD	A	B	Cin	O0	OCOUT	0	0
SUB	A	B	Cin	O0	OCOUT	0	1
RSUB	A	B	Cin	O0	OCOUT	1	0

(b)

Figure 4.8: Example source component: (a) Port description (b) Function table



(a)

Function	Input			Output		Control
	I0	I1	ICIN	Z	Cn	CS
ADD	X	Y	C0	O0	OCOUT	0
SUB	X	Y	C0	O0	OCOUT	1

(b)

Figure 4.9: Example target component: (a) Port description (b) Function table

We illustrate each of the steps in the approach by walking though an example. Let S be an arithmetic unit that can perform three functions: ADD, SUB and RSUB. Let T be another arithmetic unit that can perform two functions: ADD, SUB. These two components are shown in Figures 4.8 and 4.9 respectively. Note the differences in the port names of these two components. From these component descriptions, we extract the function tables for S and T as shown in Figures 4.8(b) and 4.9(b).

The mapping rule database (R) contains an extensive set of rules to map one RT-function onto another. From this database, we extract rules that map a source function onto a target function. Table 4.7 shows some interesting rules that have been extracted for our example.

Rule name	*Source fn*	*Target fn*	*Input ports*			*Output ports*			
			TI0	*TI1*	*TICIN*	*SO0*	*SOCOUT*	*SOVF*	*SO1*
AA1	ADD	ADD	SI0	SI1	SICIN	TO0	TOCOUT	TOVF	–
AS1	ADD	SUB	SI0	$\overline{SI1}$	SICIN	TO0	TOCOUT	TOVF	–
SA1	SUB	ADD	SI0	$\overline{SI1}$	SICIN	TO0	TOCOUT	TOVF	–
SS	SUB	SUB	SI0	SI1	SICIN	TO0	TOCOUT	TOVF	–
RA1	RSUB	ADD	$\overline{SI0}$	SI1	SICIN	TO0	TOCOUT	TOVF	–
RS	RSUB	SUB	SI1	SI0	SICIN	TO0	TOCOUT	TOVF	–

Table 4.7: Selected set of rules for mapping example

In the next step, the mapping algorithm implements S onto a canonical ALU (C). This canonical ALU uses only those functions that are present in T. This mapping is achieved by finding a set of rules, one for each source function, such that cost of extra hardware (i.e., gate count) is minimized. The set of selected rules provides the connectivity between the ports of S and C. Figure 4.10 shows one such solution in terms of the selected set of rules and the canonical implementation.

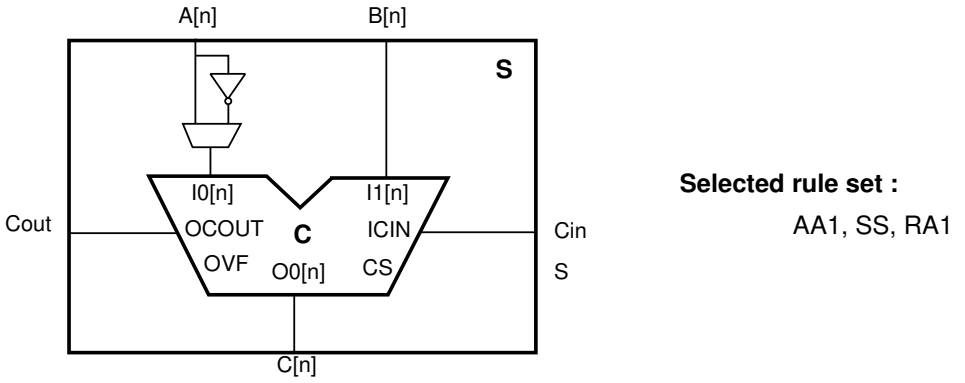


Figure 4.10: Mapping of source component (S) to canonical component (C)

The final step involves mapping the canonical ALU (C) onto the target component (T). This is usually a simple process since we restrict T to be very close to C (see Section 4.2.1). This step connects the ports of C to the ports of T. Figure 4.11 shows the final implementation.

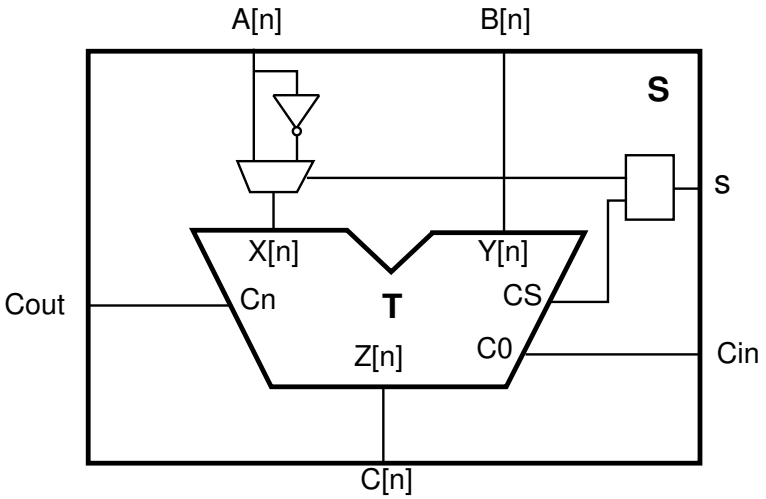


Figure 4.11: Mapping of source component (S) to target component (T)

Note that generation of the final design requires solving many other subproblems such as bit-width mapping, control mapping, secondary input and output mapping, port name mapping, etc. Again, it is important to note that the work described here focuses on an algorithm for the functional mapping of the source to the target component, which is the heart of the ALU mapping problem (the first mapping step in Figure 4.7).

4.4 An ALU Mapping Algorithm

The task of mapping a source component (S) onto target component (T) is accomplished by selecting a set of mapping rules, one for each function in source component S, that realizes the best mapping of S on T with respect to the cost function (e.g., area or delay). Recall that not only are there multiple mapping rules for each source function, but the selection of mapping rules for various source functions are also interdependent. For example, consider the rules presented in Table 4.7. If we decide to use the rule "AS1" for mapping the source function

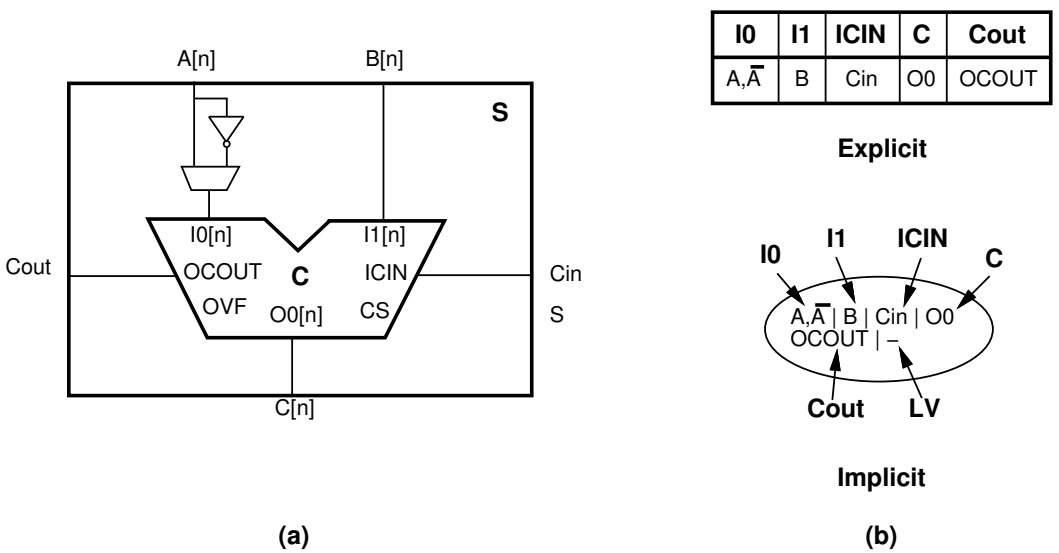
ADD, the rule "SA1" for source function SUB would lead to an efficient implementation, since it shares the factor $\overline{SI1}$ for the right primary input. Thus a strategy is required to select a mapping rule out of multiple alternatives for each source function. In this section, we present a mapping algorithm that performs this selection of mapping rules in an efficient manner. This algorithm takes as input the function tables of T and S along with the selected rule set, and maps a source component (S) onto the canonical component (C). This corresponds to the first (and major) mapping step in Figure 4.7. The algorithm generates as output the required mapping in terms of the set of rules and port connectivity.

Our mapping algorithm employs a constructive approach where an initial (possibly null) partial solution is refined into the final solution. While working through an algorithm, we need to keep track of the partial mapping. A partial mapping is represented by storing the list of inputs to the input ports of C and the output ports of S. Eventually, all these inputs are multiplexed based on the control signals. Specifically, a partial solution keeps the following list of inputs to each of these terminals:

- **I0 :** Left input, given by Boolean expressions with primary inputs of S and constants.
- **I1 :** Right input, given by Boolean expressions with primary inputs of S and constants.
- **ICIN :** Carry input, given by Boolean expressions with carry input and constants.
- **Primary output of T :** given by Boolean expressions with primary output of C, logic output (LO) and constants.
- **Comparison output of T :** could be generated with the comparison output of C and some functions of O0 and L0.
- **Secondary output of T :** e.g., carry out and overflow signal.

- **Logic vector(LV) :** used to keep track of the implementation of logic functions from scratch (as discussed in Section 4.2.5). This is required only if logic functions are to be synthesized.

A textual representation of the partial solution could be described in a tabular fashion just like the mapping rule. In fact, each mapping rule is a partial mapping with only one function. Since we know the ordering of input and output ports (e.g., I0, I1, ICIN, Primary output, Comparison output, Secondary outputs and LV), we need not be explicit in associating the terminal to its input list. Instead, an ordered arrangement of list of inputs for each terminal should suffice. As an example, the output of the mapping algorithm in the previous section (Figure 4.10) is shown in Figure 4.12. It shows both the explicit tabular and the implicit representation.



I0	I1	ICIN	C	Cout
A,$\overline{A}$	B	Cin	O0	OCOUT

A,$\overline{A}$ | B | Cin | O0
OCOUT | –

Figure 4.12: Partial solution example: (a) Pictorial representation (b) Textual representation

Note that we need not keep track of all the above input lists. Depending on the requirements (set of functions to be mapped) some of the port mappings might not be required and could be omitted. For example, if we are not dealing with logic functions, LV could be omitted. If we are using the implicit representation,

we have to be more careful. Note that it is very easy to compute the hardware cost(gate-count) and the implementation from this partial solution representation. This is evident from the example shown in Figure 4.12 as we see that there is a one-to-one correlation between the hardware and the representation.

4.4.1 The search space

Since many feasible mappings exist and since we use a constructive approach, we need to define the search space for our mapping problem. The search space is built by applying different sets of valid mapping rules for the mapping problem. Each of the algorithms mentioned in this section goes though the partial solutions and finally leads to a complete solution. We introduce the search space for the mapping example discussed in the last section. It is described by the tree shown in Figure 4.13. The leaves of this tree represent a complete solution whereas internal nodes represent partial solutions. At the root of the tree, we have a null partial solution. At each level, a source function is selected and all the mapping rules for this function are explored. An internal node represents the partial solution using the rules that are on the path from the root to this node. Figure 4.13 shows some of the partial solutions along with a few complete solutions. Some of the algorithms (1-greedy and m-greedy) could be illustrated using this tree of the search space. Note that a trivial way of finding an optimal solution is the exhaustive method that generates all the leaf nodes and selects the best mapping. Of course, this is a very time-inefficient solution.

We have formulated a suite of 4 algorithms to solve the ALU mapping problem, namely *1-greedy*, *k-greedy*, *dynamic programming* and *graph clustering*. These algorithms are independent of the cost functions and that these algorithms could be applied with either of the cost functions discussed before. We focus on the dynamic programming algorithm here; [JhD94b] describes all the four algorithms.

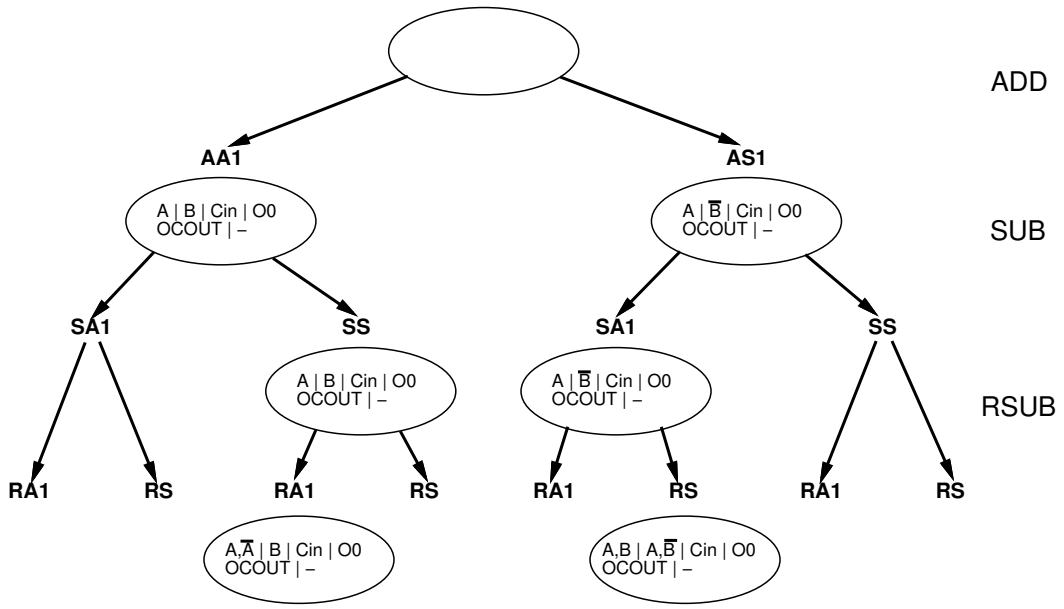


Figure 4.13: The search space for ALU mapping example

4.4.2 Dynamic programming algorithm

The Dynamic Programming (DP) algorithm is based on dynamic programming technique of building the partial solutions in the bottom-up fashion [CoLR90]. The algorithm starts with the mapping for subsets of single functions, followed by mapping for subsets of two functions and so on till it has the mapping for the entire set of source functions. Each of the partial mapping for a subset of functions is stored in a table and subsequently used for building the partial solutions for subsets of bigger size. Refer to [CoLR90] for details on the concepts behind the dynamic programming algorithm.

Algorithm 4.4.1 is a dynamic programming algorithm that keeps track of k (bucket size) best partial solutions. Note that number of partial solutions increases exponentially with the size of function set. We restrict ourselves to a limited number(k) of partial solutions for each subset. Of course, by doing so we may sacrifice optimality with the advantage of requiring bounded storage space.

Algorithm 4.4.1 : Dynamic programming (DP) algorithm
INPUT: f(S), f(T), r(ST), k.
OUTPUT: A set of rules, one for each function in f(S).
1. Order f(S).
2. **for** i = 0 to nS-1 **do**
   2.1 Table[i,i] = k-best way of mapping f_i.
3. **end for**
4. **for** i = 1 to nS-1 **do**
   4.1 **for** j = 1 to (nS-i) **do**
      4.1.1 CreateEntry(j-1, i+j-1).
   4.2 **end for**
5. **end for**

As mentioned before, the dynamic programming algorithm builds up a table of partial solutions in a bottom-up fashion. This table is indexed by the number of source functions(nS) for both row and column. An entry $table(i, j)$ represents a set of partial solutions for source function i to source function j. Figure 4.14 shows the table with the bucket size of 2 for our walk-through example. Note that the table is upper-triangular.

Algorithm 4.4.2 : CreateEntry(row,col)
1. $min_k = 0$.
2. **for** i = 0 to (col-row-1) **do**
   2.1 min_k = k-best of combine(table(row,row+i), table(row+i+1,col)).
3. **end for**

This algorithm iteratively fills up the table with partial solutions. It starts by filling diagonal entries by generating the mapping for single functions. Each diagonal entry represents a set of partial solutions that map exactly one source function. Then it fills up the entries corresponding to two function sets and so on. The final solution is given by the top-row and right-most column. For our example shown in Figure 4.14, $table(0, 2)$ lists few solutions that represent the complete mapping. Function $CreateEntry$ creates the list of partial solutions for a set of source functions using the partial solutions generated so far.

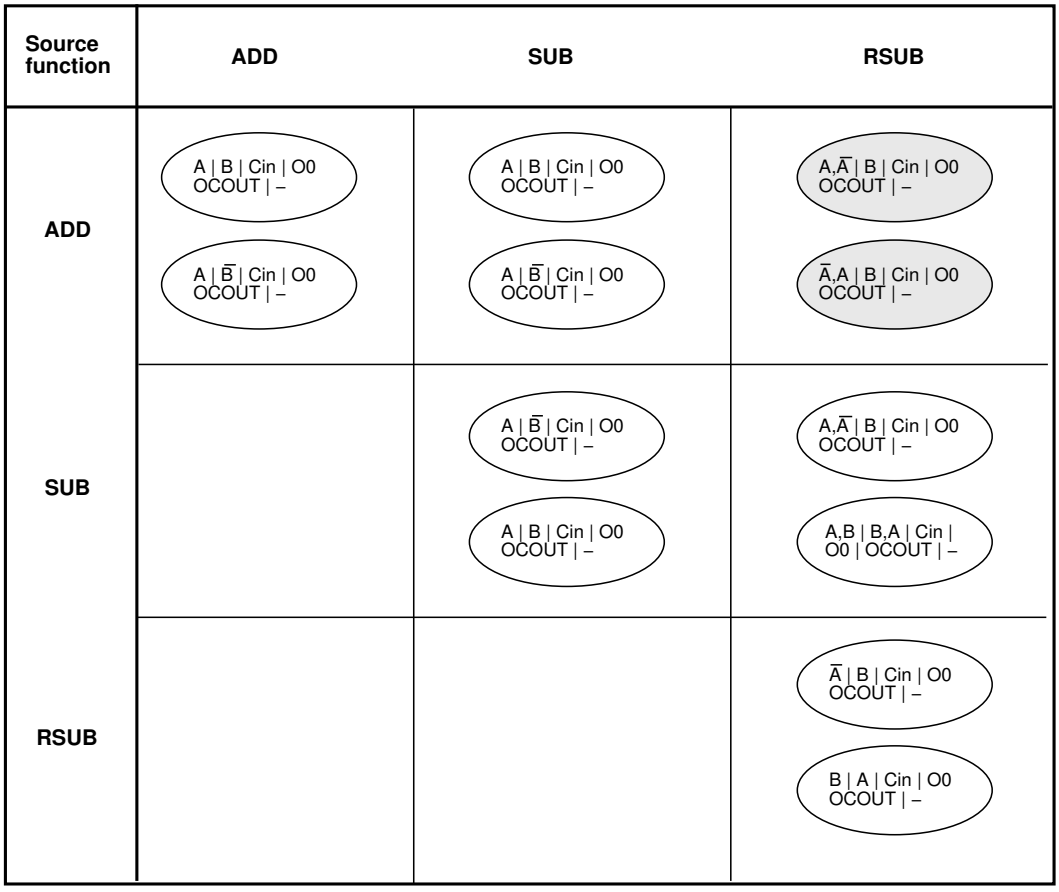


Figure 4.14: Execution of dynamic programming algorithm on an ALU example

Complexity analysis

The complexity of the algorithm is $O(n^3k^2)$, where n is the number of source functions and k is the bucket size. Algorithm 4.4.2 has a loop with worst case iteration count of n. Each iteration of this loop takes $O(k^2)$ time. Algorithm 4.4.1 has a doubly nested loop at statement 4. This loop runs for $O(n^2)$ times and each iteration requires $O(nk^2)$ time. Thus, the complexity of the whole algorithm is $O(n^3k^2)$.

4.5 Experimental results

We performed two sets of experiments to validate our approach. The first set of experiments tests the comprehensiveness of the approach in terms of mapping different arithmetic components. The second set of experiments compares the metrics of the designs generated by our approach against the ones generated by previous approach, namely the logic synthesis approach. First, we demonstrate the comprehensiveness of our approach, followed by the comparative study of design quality.

4.5.1 Comprehensiveness

In this section, we present experimental results that establish the generality of our approach across different source and target libraries using algorithms mentioned in the last section. Specifically, we considered design metrics for sample ALUs mapped by two algorithms: Greedy and *Dynamic programming*. These algorithms were run with two cost functions: gate-count(GC) and max-delay(MD) as optimization criteria. Recall that gate-count is an approximation of extra hardware required to implement the source ALU, whereas max-delay represents the

maximum delay though all the ports of the generated design. We will be summarizing mapping results generated by the dynamic programming algorithm; [JhD94b] provides detailed results, including the ones generated by the *Greedy* algorithm.

Algo : Dynamic programming **Bucket size :** 2 **Cost fn :** Gate count

Example	Source component		Target component		Result		Run–time
	Library	Fns	Library	Fns	Gate–count (2–input)	Delay (ns)	(seconds)
1.	GENUS	2 arith, 2 comp	VTI	2 arith	**37**	**56.40**	**5.5**
2.	GENUS	5 arith	CASCADE	1 arith	**389**	**24.50**	**5.5**
3.	GENUS	2 arith, 11 logic, 2 comp	VTI	3 arith, 9 logic	**630**	**72.10**	**80.6**
4.	AMD	3 arith, 5 logic	VTI	2 arith	**432**	**33.20**	**11.1**
5.	GENUS	2 arith, 11 logic, 2 comp	CASCADE	5 arith, 16 logic	**134**	**31.40**	**63.6**
6.	CASCADE	16 arith, 16 logic	VTI	3 arith, 9 logic	**453**	**36.10**	**5322.1**
7.	CASCADE	16 arith, 16 logic	AMD	3 arith, 5 logic	**757**	**–**	**5500.7**

Figure 4.15: Experimental results for area-efficient mapping

We considered a variety of ALUs, both source and target, in our experiments. These ALUs vary in terms of the library they come from, number and the set of functions they perform. We present the mapping results for ALUs from four libraries : GENUS[JhD94a], CASCADE[Casc92], VDP300[VTI91] and AMD[Am2901]. GENUS contains an ALU generator parametrized by the set of functions, bit-width, etc. The ALUs in CASCADE and VDP300 have a fixed set of functions. The AM2901 ALU is a commonly used 8-function ALU.

We covered a wide range of ALUs in terms of the number of functions they perform. Starting from a simple uni-function adder, the most complex ALU had

32 functions. Note that an ALU, as we have defined, can have only 27 distinct canonical functions. Thus, some ALUs in our experiments have variants of the canonical functions repeated in the function set. For example, one of the ALUs in our experiments has 9 ADD functions, each with different port configurations. The ALUs in our experiments perform different sets of functions, covering different functional categories: arithmetic, logic and comparison. We also chose ALUs of different bit-widths.

Algo : Dynamic programming **Bucket size :** 2 **Cost fn :** Max delay

Example	Source component		Target component		Result		Run–time
	Library	Fns	Library	Fns	Gate–count (2–input)	Delay (ns)	(seconds)
1.	GENUS	2 arith, 2 comp	VTI	2 arith	**37**	**56.40**	**5.4**
2.	GENUS	5 arith	CASCADE	1 arith	**389**	**24.50**	**5.6**
3.	GENUS	2 arith, 11 logic, 2 comp	VTI	3 arith, 9 logic	**1110**	**68.10**	**73.6**
4.	AMD	3 arith, 5 logic	VTI	2 arith	**464**	**32.40**	**11.1**
5.	GENUS	2 arith, 11 logic, 2 comp	CASCADE	5 arith, 16 logic	**278**	**28.20**	**64.6**
6.	CASCADE	16 arith, 16 logic	VTI	3 arith, 9 logic	**485**	**36.20**	**4558.4**

Figure 4.16: Experimental results for delay-efficient mapping

Note that the examples in our experiments were restricted by the availability of design metrics and not by the limitations of our approach. For example, the delay for target components in a library were available only for specific bit-widths such as 8, 16, 48. Thus, all our experiments are for ALUs with one of the above bit-widths. Recall that our approach is independent of bit-widths and that it will require same amount of computation for all bit-widths. Similarly, we had to restrict ourselves to only those libraries that provide design metrics. Even though

we considered other libraries such as XBLOX[XBLO92], LSI[LsiLogic], Toshiba gate array[Tosh90] etc., we could not run our algorithms due to lack of metric data (gate counts, performance) for these libraries. We also had to restrict our mapping examples to ALUs with fixed sets of functions, since these are the only ALUs supported by some of these libraries.

Figures 4.15 and 4.16 summarize the area-efficient and delay-efficient designs respectively generated by the dynamic programming algorithm for seven examples. These tables are in two parts; the left part contains the component descriptions and the right part describes the mapping result. The component description includes example number, followed by the description of the source and the target component. Each of the component description includes the library name and the set of functions that a given ALU can perform. The result section (right part) describes the design metrics and the run time for each mapping result. The gate-count in the result section represents the approximate number of extra 2-input gates that will be required to build the source component using target component. Delay represents max-delay in nanoseconds, i.e., this is the maximum delay through all the input-output port combination of the resultant design. The run time column shows execution time (user + system) for the given example on Sparc 2 (sun-670-mp for examples 6 and 7). We also show the name of the mapping algorithm, the bucket size and the optimizing cost function on the top of the table.

As mentioned before, the seven examples in our experiments are from different libraries and are of varying complexity. Example 1 maps a GENUS ALU with 2 arithmetic and 2 comparison functions onto a VDP ADD-SUB component. Example 2 implements a GENUS ALU with all the arithmetic functions on CASCADE adder. The third example maps another GENUS ALU with functions covering all the three categories (arithmetic, logic and comparison) onto a VDP ALU that covers some of the arithmetic and logic functions. The next example maps the AM2901 ALU with 3 arithmetic and 5 logic functions onto a VDP adder-subtracter. The fifth example maps the same source component as in Example 3,

but this time the target component is a CASCADE ALU that covers some of the arithmetic and all the 16 logic functions. Examples 6 and 7 use same source component: a complex ALU from CASCADE that has 32 functions with many repetitions of same canonical function. In Example 6, the target component is a VDP ALU whereas in Example 7 the target component is the AM2901 ALU. [JhD94b] describes each of the output designs for these examples.

Figures 4.15 and 4.16 summarizes the mapping results generated by the dynamic programming algorithm for a specific bucket size (= 2). We ran dynamic programming algorithm with different bucket sizes (e.g., 1 and 4) and other mapping algorithms (e.g., greedy). [JhD94b] shows the complete results for these runs. From the results shown in Figures 4.15 and 4.16 and others reported in [JhD94b] we observe that our approach is quite versatile in the sense that it can map ALUs from one library onto another. We have demonstrated the versatility of our approach by applying it on ALUs from wide variety of libraries. Also, our approach can handle ALUs with diverse complexity in terms of bit-width, number of functions and set of functions. Our algorithms generate designs of high quality, often optimal designs.

We conclude this section with the following comment. Note that we have used a restricted set of rules to map a source function onto a target function. [JhD94b] shows the list of rules that has been used in our experiments. These set of rules are not complete, particularly with respect to rules for mapping logic functions. Our algorithms may generate even better results, if provided with a comprehensive set of rules.

4.5.2 Goodness

Now we present the results our experiments that compare metrics of the designs produced by our approach against the ones produced by a commercial logic synthesis tool[Syno92]. Figure 4.17 describes the experimental set-up. For

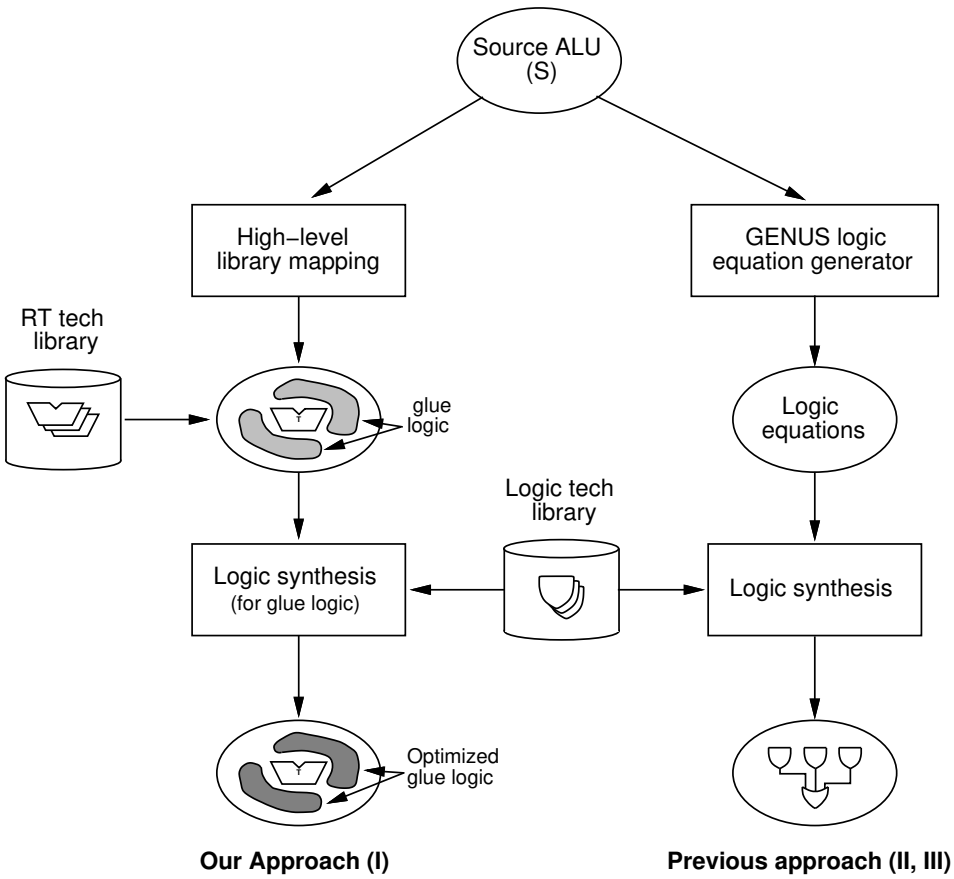


Figure 4.17: Experimental setup for comparing HLLM with logic synthesis

Example		Source component		Target component	
#	BW	Library	Fns	Library	Fns
1.	32	GENUS	3 arith	Designware	add,sub
2.	27	GENUS	5 arith	Designware	add
3.	32	GENUS	3 arith, 5 logic	Designware	add,sub
4.	16	GENUS	2 arith, 11 logic, 2 comp	Designware	add,sub

Figure 4.18: Source and target ALUs for comparative study

each source component, we first map this component onto a macro in the Synopsys DesignWare library[Degn92], with additional glue logic using our approach. Then we pass the resultant design through logic synthesis to optimize the glue logic. We present metrics for the designs (I) arrived by this path (our approach). The left portion of Figure 4.17 illustrates this path. In the next phase of the experiment, we generate the logic equations for the input source component and directly map onto gate-level cells using the logic synthesis approach. Logic synthesis is tried with two levels of optimizations: low and medium. The designs generated by the low optimization script are refered to as (II) and those generated by the medium optimization script are refered to as (III). For each of these designs, we present design metrics: gate-count, max-delay and run-time and compare them against the metrics for designs generated by our approach (I).

For each input source component, we perform two sets of experiments. In the first, we generate designs that are optimized for area (minimum gate-count). For this set of experiments, both the logic synthesis tool and our algorithm are configured to generate best area designs. In the other set of experiments, we configured our algorithm and logic synthesis tool to optimize the worst case delay. We present designs metrics for each of these experimental sets.

Figure 4.18 shows the source and target ALUs used in this experiment. The table in this figure shows the bit-width for each example, followed by the source component and the target component description. The table specifies the library name and the set of functions for each source and the target component used in this experiment. Specifically, the we have used four examples. The source components are from GENUS library and are of varying complexity both in terms of the bit-width and the set of functions they perform. All these components are mapped onto the adder or adder-subtractor macros from the designware library.

Figures 4.19 through 4.22 graphically present the experimental results for the four examples shown in Figure 4.18. Detailed results for this experiment are

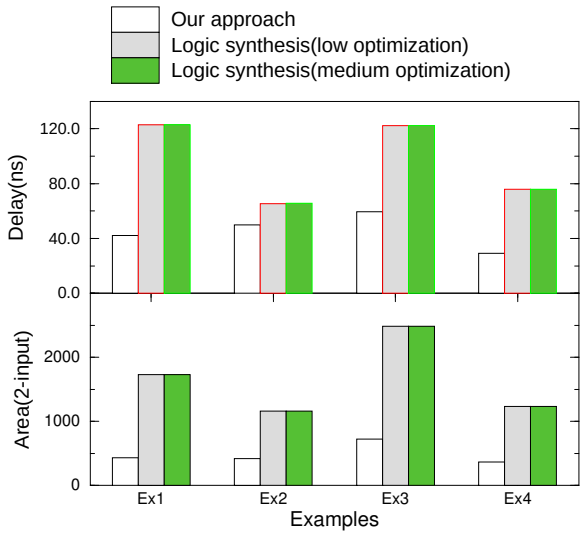


Figure 4.19: Performance metric for Area-optimized designs

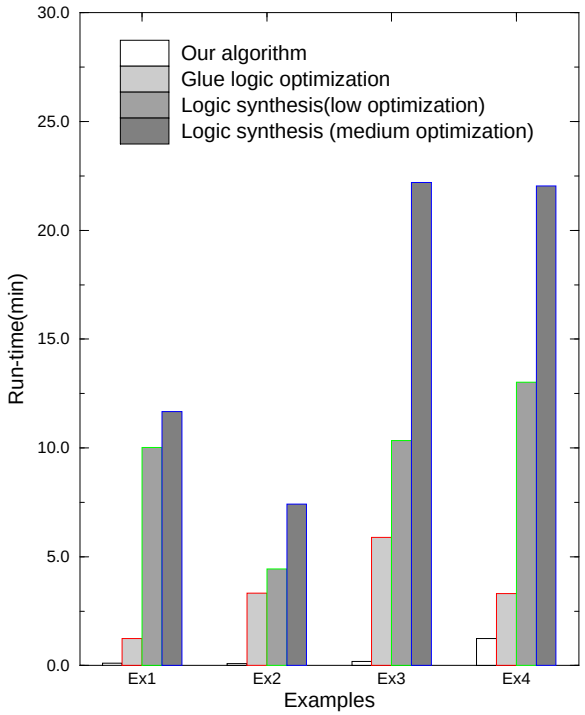


Figure 4.20: Run-time for Area-optimized designs

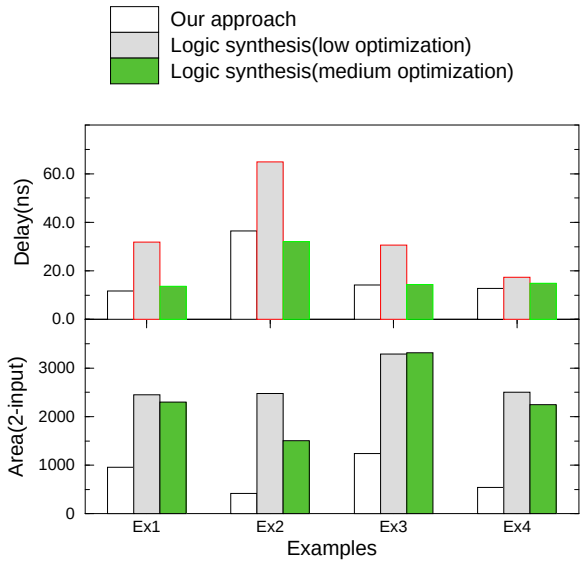


Figure 4.21: Performance metric for Delay-optimized design

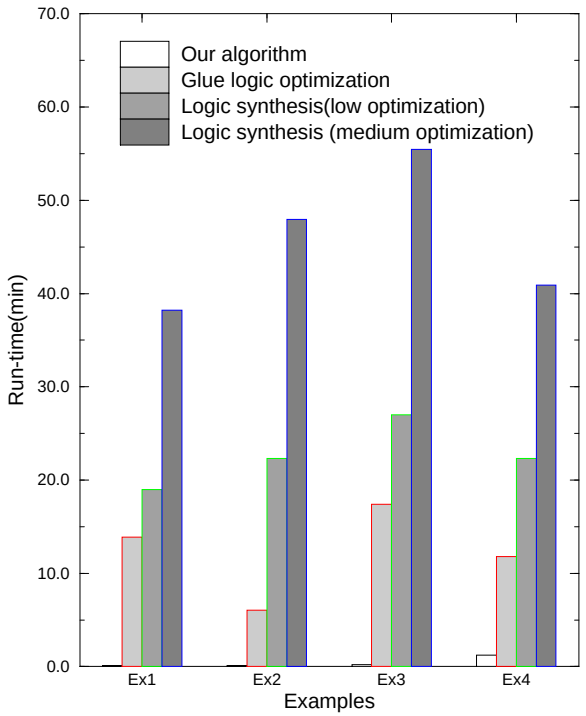


Figure 4.22: Run-time for Delay-optimized design

available in [JhD94b]. Figures 4.19 and 4.21 illustrate area and delay for area-optimized and delay-optimized designs respectively. There are three columns for each design in these two figures. These three columns represent performance metrics for the designs generated by the three approaches: our approach (I), logic synthesis approach with low (II) and medium (III) effort. Figures 4.20 and 4.22 depict run-time for area-optimized and delay-optimized designs respectively. Note that there is one extra column for each design in these two figures. The runtime for generating designs with our approach is split into two columns. The first column represents runtime for mapping algorithm and the second column represents runtime for optimizing the glue logic.

From Figures 4.19 through 4.22, we observe that **designs generated by our approach(I) are better with respect to all the three metrics: gate-count, delay and runtime.** Designs generated by our approach are not just better, but substantially better than the ones generated by the logic synthesis tools. For all these example, the delay of the designs produced by our approach is less than the delay of designs produced by logic synthesis approach (except for one example). The area of designs from our approach is less than the half the area of the designs produced by the logic synthesis approach. With respect to runtime, results are even more dramatic. Specifically, designs generated by logic synthesis tools are inferior in area by 139% to **487%**, in delay by -12% to **191%** and in runtime by 37% to **834%**.

Note that these percentage differences for runtimes considers the sum of the runtimes for mapping algorithm and glue logic optimization. If we consider the time for just mapping algorithm, we outperform logic synthesis by orders of magnitude. This is quite apparent by the bar-charts shown in Figures 4.20 and 4.22. Note that the bars for mapping algorithm in these figures are barely visible for the first three examples, indicating how relatively small they are.

We conclude this analysis with two comments. First, note that in this experiment, we have compared metrics from the netlist of generic gates. The effects

of regularity are more pronounced when we map these designs onto layout; designs from our approach would perform even better. Second, the reason we have been able to outperform logic synthesis tools is that these tools are designed for optimizing random or control logic. These tools cannot exploit the regular structures inherent in data-path components. The logic equations for a moderately size component (32-bit ALU in our example), are too big to be handled by these tools. Thus, there is need for a RT level library mapping technique such as ours to handle these regularly structured datapath components.

4.6 Summary

In this chapter, we formulated a high-level library mapping technique for ALUs. We presented an efficient polynomial time ALU mapping algorithm based on a dynamic programming formulation. The algorithm could be used for generating either area-optimized or delay-optimized designs.

In our experiments, we demonstrated the versatility of our approach by applying HLLM on ALUs drawn from a wide variety of libraries. We also demonstrated the superiority of our approach over logic synthesis for complex ALU components in all the three metrics: area, delay and runtime. These experimental results establish the necessity of a RT level library technique such as ours.

Chapter 5
HLLM for Memories

5.1 Introduction

Memory modules are commonly used in RT level designs and are explicitly represented in the design model at the RT level (Figure 2.7, Chapter 2). With the increasing importance of high-speed data intensive applications in the fields of speech, image and video processing that require significant amount of storage capability, the memory subsystem becomes an important focus of design. For such applications, the area cost of memory components could be as high as 80% of the complete design. Hence, there is a need for efficient implementations of memory elements in these designs.

In this chapter, we present High-level library mapping for memories, a technique to implement a source memory module from a set of memory modules from a target library. HLLM for memory is based on higher levels of abstraction for memories: given the high-level specification of the source and the library modules in terms of word-count, bit-width and the port configurations, the high-level library mapping for memories implements the source memory module using target memory modules in an efficient manner so as to optimize a user-given cost function. This approach is applicable to the synthesis of the on-chip as well as the off-chip memory modules.

This chapter is organized as follows. Section 5.2 briefly describes related work. Section 5.3 defines high-level library mapping for memories and decomposes it into three subproblems. Section 5.4 combines the three subproblem formulations into an efficient memory mapping algorithm. Section 5.5 describes sample memory mapping results to demonstrate the efficacy of our approach. The chapter concludes with a summary.

5.2 Previous Work

Research in the use of memories for design automation systems at RT and HLS domain has recently gained importance. These works could be broadly classified into two groups. The first group of work concentrates on translating the storage requirements in the input behavior onto logical memories. The second group of work maps these logical memories onto physical memories from a library. HLLM for memories fits in the second group. [JhD95b] describes related work in the first group; we summarize works in the second group in this section.

[KiLi93] packs a set of logical memories (result of scalar variable clustering) into a set of memory modules from a library. They model this process as a two dimensional bin-packing problem where the number of ports and number of registers in the modules constitute the two dimensions. [BaGa95] applies a sequence of simple memory expansion steps to build a memory organization that satisfies the logical memory requirements. These steps include bit-width expansion (when the required bit-width is larger than the bit-width of the library memory modules), word-count expansion (when the required word-count is larger than the word-count of the library memory modules), interleaving (to increase the access rate) and port multiplexing (to increase the number of ports or to decrease the access delay)[1].

[1] A set of n words can be accessed serially from a port and be stored in n buffer registers and subsequently be read out in parallel, virtually increasing the number of ports at the cost of increasing the access delay. Conversely, n ports can be grouped together to provide n words

Another work by Schmit and Thomas [ScTh95] first groups arrayed variables using a set of basic moves such as horizontal concatenation (that increases the bit-width), vertical concatenation (that increases the word-count), array widening (consecutive words are placed in a single wider word) and array narrowing (a word is split and placed into consecutive words) then binds (maps) each of these grouped variables onto one or more instances of the same physical memory module. They use simulated annealing to select a set of basic moves (array clustering steps, array binding, etc.) that leads to an efficient memory design. [KaRo94] packs a set of logical memories into a fixed set of physical memories to be used in a Field Programmable System. Each logical memory is first broken into smaller pieces that can fit into a physical memory. Next they use a Branch and Bound algorithm to map these logical memory pieces onto physical memory modules.

Work	Logical memory		Physical memory		Memory parameters		
	Number	Type	Number	Type	Words	Bits	Ports
[Kili93]	Multiple	Multiple	Multiple	Multiple	Yes	No	Yes
[BaGa95]	Single	Single	Multiple	Single	Yes	Yes	Yes
[ScTh95]	Multiple	Multiple	Multiple	Single	Yes	Yes	?
[KaRo95]	Multiple	Multiple	Multiple	Single	Yes	Yes	?
Ours	*Single*	*Single*	*Multiple*	*Multiple*	*Yes*	*Yes*	*Yes*

Figure 5.1: Classification of memory mapping works

Most of these works perform the task of memory mapping as a backend to their behavioral synthesis system. [KiLi93] emphasizes grouping of the scalar variables into logic memory modules and then packs these logic memory modules into physical memory modules. [BaGa95] performs the task of memory selection, and synthesizes the logical memory if the required memory is not available in the library. [ScTh95] incorporates the memory mapping scheme directly into their

which can be accessed serially at the rate of n-times the memory access rate, virtually decreasing the memory access delay by utilizing multiple ports.

behavioral synthesis system. However, in this work we focus on the task of post-synthesis memory mapping. The work of [KaRo94] falls in this domain; they concentrate mainly on mapping logical modules onto physical modules.

Memory mapping works can also be classified on the basis of the number and the type of logical and physical memory modules considered simultaneously. Figure 5.1 illustrates this feature for the above mentioned memory mapping works. [KiLi93] performs the most general mapping in terms of mapping multiple logical memory modules of different types onto multiple physical memories of different types. [ScTh95][KaRo94] realize multiple logical modules with multiple physical memories of the *same* type. We and [BaGa95] focus on realizing a *single* logical memory at a time. However, in contrast to [BaGa95], we pack different types of physical memory modules to realize a logical module.

With respect to realizing a *single* logical memory, these works have limited scope in the sense that either they do not consider all the degrees of freedom associated with memories (word-count, bit-width and port) or their mapping scheme is tuned for a specific memory module set. The last three columns in Figure 5.1 (labelled *Bits, Words and Ports*) compares the comprehensiveness of these works with respect to the three memory parameters[2]. Specifically, [KiLi93] does not consider bit-width expansion, [BaGa95][ScTh95] consider only simple realizations of a source memory module that does not use multiple memory types; [KaRo94]'s system is tuned to a fixed set of physical modules (4 instances of 32Kx8 SRAM). Our approach is comprehensive; it considers all degrees of freedom associated with memory modules (word-count, bit-width and port) and can pack multiple memory types together to realize a required memory module. Furthermore, our approach is not tuned to any specific system or the source or the target memory module

[2]In Figure 5.1 the "Yes" and "No" entries specify that the corresponding work considers or ignores a specific memory parameter, whereas a "?" entry in the *Ports* column specifies that the work does not clearly state how they handle port mismatches between the logical and physical memory modules.

set, and can therefore be used as a backend to most existing behavioral memory synthesis approaches.

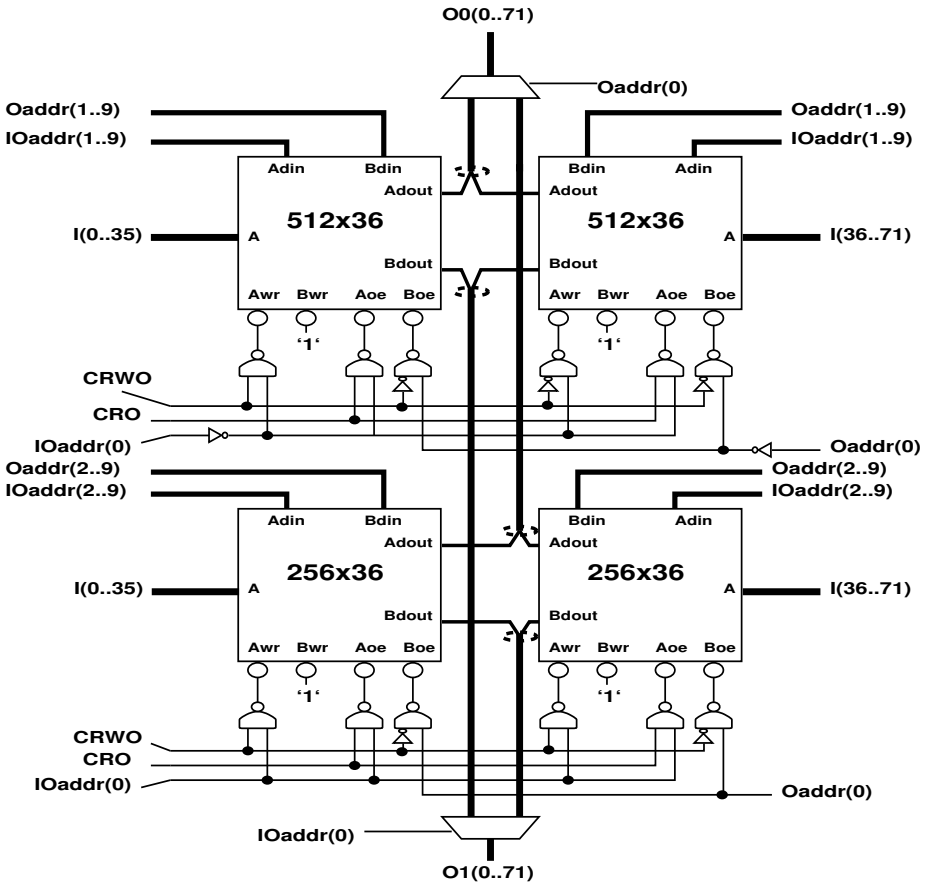
5.3 Problem Definition

We define high-level library mapping for memories (or simply *memory mapping*) as the task of realizing a source memory module with a set of target memory modules from a library. The memories can be on-chip macros or off-chip components. If the size (number of words or bit-width) of a target memory module is greater than the required size, the source memory module can be realized with a single target memory module; otherwise the source memory is realized with a set of target memory modules. Figure 5.2 shows a memory mapping example where a 768-word, 72-bit memory (the source) is implemented using two instances of a 512-word, 36-bit and two instances of a 256-word, 36-bit target modules from a library. Besides the target memory modules, the mapping also requires address decode logic that translates the source module address into the target module addresses, as well as the multiplexers to steer the data output from the target modules. The goal of the mapping process is to achieve a feasible target implementation that satisfies a user-given cost function.

In this section, we first describe the parameters used to specify a memory. Next we define and present algorithmic formulations to the three memory-mapping subproblems, namely port mapping, word-count mapping and bit-width mapping. The section ends with a brief description of the cost functions used in our approach.

Parameters	Source	Target	
	s	t1	t2
Word–count	768	512	256
Bit–width	72	36	36
Ports	1 RW,1R	2R,2W	2R,2W

(a)



(b)

Figure 5.2: Sample high-level library mapping for memories: (a) Source and target modules (b) Mapping result

5.3.1 Memory specification

A library memory module is typically specified by the following parameters : type (SRAM, DRAM, EPROM, etc.), clocking mechanism (asynchronous or synchronous), size (number of words and bit-width for each word), number and type of ports, protocol timing diagram, etc. At higher level of abstraction, a memory module m can be characterized by its size (number of words and bit-width for each word) and the amount of data access parallelism (number of ports). We enumerate the following list of parameters required to characterize a memory component m. For each parameter, we define a function that returns the value of the parameter for a memory module.

- **Ports :** Data is accessed in and out of memory through ports. Based on the direction of transfer of data, ports are categorized into three types:

  Read ports support data transfer in only one direction, **from** the memory. Let $R(m)$ be the number of read ports for m.

  Write ports support data transfer only **to** the memory. Let $W(m)$ be the number of write ports for m.

  Read-write ports support data transfer in **both** the directions, i.e., in and out of the memory component. Let $RW(m)$ be the number of read-write ports for m.

- **Word-count :** The storage capacity of a memory is characterized by specifying its number of words. Let $N(m)$ be the number of words for m.
- **Bit-width :** specifies the width of each word in the memory. Let $B(m)$ be the bit-width for each word in the memory m.

The cost function

We define three cost measures for a memory module. These cost measures are used to guide the mapping algorithms.

- **Area measure :** represents the area used by a memory module; this can be represented by the silicon area, the transistor-count, the gate-count or the actual area occupied on a PCB by the off-chip memories. Let $A(m)$ be the area measure for m.
- **Price measure :** represents the price of a memory component. Let $P(m)$ be the price measure for a memory.
- **Delay measure :** The timing diagram for a memory component is described by a set of access times, set-up times, hold times as well as cycle times. In this work, we use the worst case access time (read or write) to characterize the timing behavior for a memory component and denote it by $D(m)$.

5.3.2 Memory mapping

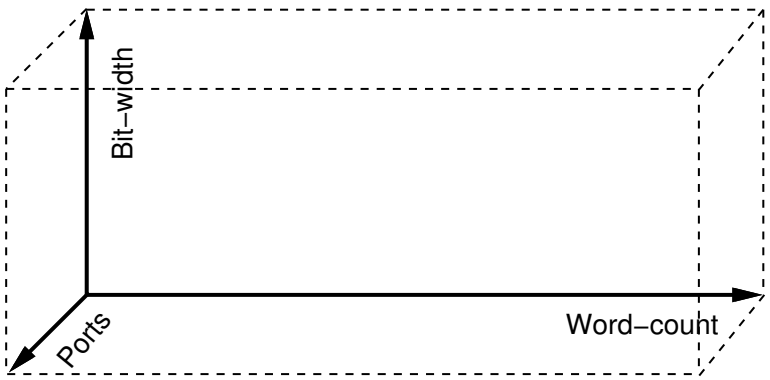


Figure 5.3: Three degrees of freedom in memory mapping problem

The *memory mapping* problem is defined in terms of a source memory module s, a set of target memory modules T from a library and a user-given cost function C. The source and target modules are characterized by specifying the three parameters, namely the ports, the word-count and the bit-width. The aim of memory mapping is then to implement the source s with one or more target modules from the set T in such a way that the realized design performs well with respect to the user-given cost function C.

The memory mapping problem addresses the mismatches in the source and target modules with respect to the three memory parameters namely the ports, the bit-width and the word-count. Figure 5.3 shows these three degrees of freedom for a typical set of memory modules from standard databook libraries.[3] The length of an axis in this figure represents the relative variance in the values for the corresponding parameter. We observe maximum variance in the word-count of memory modules. For instance, a memory component could have its word-count vary from 4-word register-files to 16 Megaword memories. Bit-width has relatively lower variance, usually in the range of 1 to 128. The number of ports in a memory component are usually very small, ranging from 1 (single port ROMs) to 6 (Multiport RAMs from [Casc92]).

Note that these three parameters are orthogonal to each other, i.e., variation of one parameter is independent of another within a library of memory modules. However, the cost measure for a memory module typically is a function of all these three parameters. Thus, we need to consider all these parameters together in selecting a set of target memory modules to find a good implementation. In our mapping formulation, we first consider each of these parameters separately; the rest of this section describes formulations for the port mapping, bit-width mapping and word mapping problems. In Section 5.4, we combine these formulations to achieve a global solution to the complete memory mapping problem.

For the rest of the chapter, s refers to the source memory component, T to the target memory set and t_i to a specific target memory component. We use the terms *memory component* and *memory module* interchangeably. Furthermore, the word-count and the bit-width for a memory layout in a figure are shown in the vertical and horizontal directions respectively.

[3] See [JhDG93] for a survey of RT-level libraries.

5.3.3 Port mapping

The port mapping formulation first specifies the necessary and sufficient conditions that target modules need to satisfy in order to meet the data access requirements of the source memory. We then present a simple scheme to associate a source memory port with a distinct target memory port.

Port constraints

Each of the target memory modules should have enough data access parallelism in terms of number of ports required to support the data bandwidth for the source memory component. A source read port can be realized either by a target read or a target read-write port; likewise a source write port can be realized by a target write or a read-write port. A source read-write port can be realized using a target read-write port or a target read and a target write port. The following three equations establish the necessary and sufficient conditions that a target memory module t needs to satisfy to meet the port requirements for a source memory module s:

$$R(s) + RW(s) \leq R(t) + RW(t) \tag{5.1}$$

$$W(s) + RW(s) \leq W(t) + RW(t) \tag{5.2}$$

$$R(s) + W(s) + RW(s) \geq R(t) + W(t) + RW(t) \tag{5.3}$$

The *read constraint* in Equation 5.1 ensures that t has enough ports to read out the data; the *write constraint* in Equation 5.2 ensures that t has enough ports to write in the data. The *read-write constraint* shown in Equation (5.3) ensures that each port of t can get assigned to only one port of s. These three constraints together are called *port constraints*. Note that at this point we can not improve the data access parallelism by interleaving memory modules, since we do not know that data access patterns; we can use only those memory modules that meet the above port constraints.

Port assignment

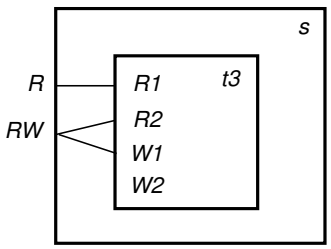
Once we have selected a target memory module that can satisfy the access rate requirements of s, we need to assign each port of a source memory component to a distinct port of a target memory component. Algorithm 5.3.1 describes a simple scheme to perform a one-to-one port assignment. The algorithm first tries to perform a simple one-to-one mapping in terms of mapping a source read port to a target read port, a source write port to a target write port and a source read-write port to a target read-write port. Each of the remaining read and write ports is realized by a single read-write port. Similarly, each of the remaining source read-write ports is realized as a read port together with a write port of t. Note that the port constraints ensure that t has a sufficient number of ports to perform this assignment.

Algorithm 5.3.1 : Port assignment
INPUT: Source memory module (s), Target memory module (t).
OUTPUT: Assignment of ports of s to the ports of t
1. **if** $(R(s) \leq R(t))$ **then**
   1.1 assign $R(s)$ read ports of s to $R(s)$ read ports of t;
2. **else**
   2.1 assign $R(t)$ read ports of s to $R(t)$ read ports of t;
   2.2 assign $R(s) - R(t)$ read ports of s to $R(s) - R(t)$ read-write ports of t;
3. **if** $(W(s) \leq W(t))$ **then**
   3.1 assign $W(s)$ write ports of s to $W(s)$ write ports of t;
4. **else**
   4.1 assign $W(t)$ write ports of s to $W(t)$ write ports of t;
   4.2 assign $W(s) - W(t)$ write ports of s to $W(s) - W(t)$ unused read-write ports of t;
5. **if** $(RW(s) \leq RW(t))$ **then**
   5.1 assign $RW(s)$ read-write ports of s to $RW(s)$ unused read-write ports of t;
6. **else**
   6.1 assign $RW(t)$ read-write ports of s to $RW(t)$ unused read-write ports of t;
   6.2 assign $RW(s) - RW(t)$ read-write ports of s to $RW(s) - RW(t)$ unused read ports of t and $RW(s) - RW(t)$ unused write ports of t;

Besides specifying the port constraints and port assignment, port mapping also involves name mapping and control mapping. Name mapping refers to the task of connecting the data inputs of the source memory component to the data inputs of the target memory component. Control mapping refers to matching the enable lines of s to the enable lines of t. We assume that these tasks are performed by the user. We focus on the high-level task of port constraints specification and port assignment.

	Source	Target		
	s	*t1*	*t2*	*t3*
R	1	0	3	2
W	0	0	0	2
RW	1	1	0	0

(a)



(b)

Figure 5.4: Port map example (a) Source and target memory components (b) Port map between source s and target t.

An example

Figure 5.4 illustrates an example for port mapping. Figure 5.4(a) lists the port configurations for a source component s and three target components $t1$, $t2$ and $t3$. The source component s is a 1R-1RW register file used in the AM2901 microprocessor slice [Am2901]. The target memory modules are instantiated from the Cascade [Casc92] module generator.

We observe that module $t1$ can not be used to implement s, since it violates the read constraint (Equation 5.1) as well as the read-write constraint (Equation 5.3). Also, $t2$ fails to satisfy the write constraint. $t3$, on the other hand, fulfills all the port constraints. Figure 5.4(b) shows the result of port assignment between s and $t3$ generated by Algorithm 5.3.1. The read port (R) of s has been assigned to

the first read port (R1) of $t3$, whereas the read-write port (RW) is realized with a read (R2) and a write (W1) port of $t3$.

5.3.4 Bit-width mapping

Bit-width mapping refers to the task of achieving the bit-width requirement of the source memory component s using a set (one or more) of target memory modules from a library. In order to find a good realization, we often have to compose a set of target memory modules to meet the bit-width requirements of s. For example, the memory realization in Figure 5.2 composes two memory module each with bit-width=36 to implement a source memory module of bit-width=72.

Next we formulate the bit-width mapping problem in terms of the bit-width of the source ($B(s)$) and the target ($B(t_i)$) memory modules. Based on the characteristic distribution of bit-widths, we then present an enumeration scheme to select an optimal set of target memory modules (for a user-given cost measure), whose composition satisfies the bit-width requirement of s.

An ILP formulation for bit-width mapping

We present a simple integer linear programming formulation for an area-based bit-width mapping problem. Given a source memory module s and a set $T = \{t_1, t_2, ..., t_m\}$ of target memory modules from a library, we have to find the number (x_i) for each of the target memory module t_i that minimizes the following expression:

$$\sum_{i=1}^{m} x_i A(t_i) \tag{5.4}$$

and satisfies the following linear constraint:

$$\sum_{i=1}^{m} x_i B(t_i) \geq B(s) \tag{5.5}$$

Recall that the functions A and B refer to the area measure and bit-width respectively for a memory module. The above bit-width mapping problem is NP-complete, since another NP-complete problem, namely the subset-sum problem [CoLR90] can be reduced to the bit-width mapping problem.

Finding an optimal solution to an unrestricted NP-complete problem is a computationally expensive process. However, the bit-width mapping problem domain for practical applications are restricted to a smaller range of bit-widths (the bit-width of a memory module typically lies between 4 and 128). Thus, we can apply an exhaustive search (enumeration) scheme to find the optimal solution.

An algorithm for bit-width mapping

Algorithm 5.3.2 describes an enumeration scheme for all possible compositions of each bit-width in a systematic fashion. The input to the algorithm is the source memory module s, the target memory module set T and the cost measure C (area, delay or price). The algorithm returns an optimal set of target memory modules (with respect to the given cost measure C) that performs the bit-width mapping for the source s.

In this algorithm the arrayed variable *best_sol* keeps track of the best composition of target memory modules for each bit-width. The set Ts stores one or more instances of each target memory module. For each target memory module t_i, we include $\lceil \frac{B(s)}{B(t_i)} \rceil$ instances, since a good mapping would require at most these many instances of t_i. Finally, L_i enumerates all possible bit-widths that can be composed using the first i memory modules from Ts.

Algorithm 5.3.2 : Bit-width mapping
INPUT: Source memory module, s; Target memory module set, T;
Cost measure, C.
OUTPUT: Bit-width mapping of s.
1 **for** i = 1 to B(s) **do**
1.1 $best_sol[i] = \phi$;
2 **end for**;
3 $Ts = \phi$;
4 **for** each $t_i \in T$ **do**
4.1 $best_sol[\mathrm{B}(t_i)] = t_i$;
4.2 **For** j = 1 to $\lceil \frac{B(s)}{B(t_i)} \rceil$ **do**
4.2.1 $Ts = Ts + t_i$;
4.2 **end for**;
5 **end for**;
6 $L_0 = \phi$;
7 **for** i = 1 to $|Ts|$ **do**
7.1 L_i = Expand-list(L_{i-1}, Ts_i);
8 **end for**;
9 **return** $best_sol[\mathrm{B}(s)]$;

The algorithm first initializes array variable *best_sol*, Ts and L_0 (Step 1–5). It then successively enumerates all possible bit-widths that can be composed using a subset of Ts (Steps 6–8). The algorithm finally returns the best composition for the bit-width of s stored at $best_sol[\mathrm{B}(s)]$. The function *Expand-list* builds a new list of bit-width compositions using pre-existing compositions along with a new target memory module: Algorithm 5.3.3 describes the steps involved to perform this task. The function *Expand-list* composes each element of L with a new target memory module and if the resulting composition is not a suboptimal one, the function stores the compositions in the new list and updates the global best solution array *best_sol*. The above algorithm uses the user-given cost function C to determine the quality of the bit-width mapping compositions.

Algorithm 5.3.3 : Expand-list(L,t)
INPUT: Sorted list of bit-width mappings, L; Target memory module, t;
OUTPUT: Sorted list of bit-width mappings composed of members of L and t.
1 $L_{new} = L$;
2 **for** i = 1 to $|L|$ **do**
2.1 new_map = Compose(L_i, t);
2.2 **if** B(new_map) $\leq$ B(s) **or** new_map does not have a t_j such that B(new_map - t_j) $\geq$ B(s) **then**
2.2.1 insert new_map to L_{new};
2.2.2 Update-best-sol(new_map);
2.3 **end if**;
3 **end for**;
4 **return** L_{new};

An example

Figure 5.5 walks through the bit-width mapping algorithm (Algorithm 5.3.2) on an example. The source memory module in this example is a part of an industrial example from [KaRo94]. The target modules are instantiated from the Toshiba gate array library [Tosh90]. The source and target memory modules are shown in Figure 5.5(a).

The bit-width of the source module is 24. There are three target memory modules, namely *t1, t2* and *t3* with bit-widths equal to 4, 8 and 16 respectively. Note that [Tosh90] contains a wide variety of memory modules; we selected a smaller set to illustrate the essence of the algorithm. We use the area measure represented by the gate-count as the optimizing cost function.

Figure 5.5(b) lists memory module instances that are used in the enumeration steps (Steps 6–9) of the Algorithm 5.3.2 for this example. Note that there are 6 instances of t1, 3 instances of t2 and 2 instances of t3 in Ts. Figure 5.5(c) shows sample lists of bit-width compositions generated by the example. The first list L1 contains a single composition for 4-bits. List L6 has 6 elements each with a

	Source	Target		
	s	t1	t2	t3
Bit–width	24	4	8	16
Word–count	64	64	64	64
Gate–count		3172	4182	6636

(a)

S = {t1, t1, t1, t1, t1, t1, t2, t2, t2, t3, t3}

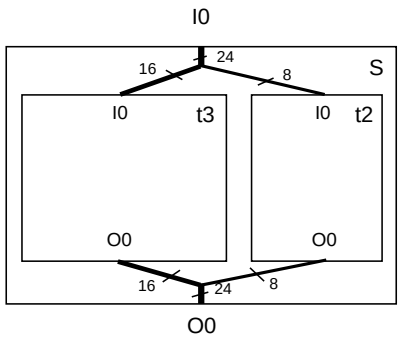
(b)

L1 = {(t1)}
L2 = {(t1), (t1, t1)}
L3 = {(t1), (t1, t1), (t1, t1, t1)}
L6 = {(t1), (t1, t1), (t1, t1, t1), ..., (t1, t1, t1, t1, t1, t1)}
L7 = {(t1), (t1, t1), (t2), (t1, t1, t1), (t2, t1), ..., (t2, t1, t1, t1, t1)}
L9 = {(t1), (t1, t1), (t2), (t1, t1, t1), (t2, t1), ..., (t2, t2), ..., (t2, t2, t2)}
L11 = {(t1), (t1, t1), (t2), (t1, t1, t1), ..., (t1, t1, t1, t1, t1, t1), (t2, t2, t2), **(t3, t2),** (t3, t3)}

(c)

Bit–width	Composition	Gate count
4	{t1}	3172
8	{t2}	4182
12	{t1,t2}	7354
16	{t3}	6636
20	{t3, t1}	9808
24	**{t3, t2}**	**10818**

(d)



(e)

Figure 5.5: An example for bit-width mapping (a) Source and target memory components (b) Target memory module set (c) Enumeration of memory compositions (d) List of best compositions for different bit-widths (e) Data input-output connection between the source and the target modules

different number of instances of *t1*. L7 uses a single instance of *t2* with multiple instances of *t1*. The last list L11 contains 20 compositions including the one that uses two instances of *t3*. The best mappings for each bit-width are shown in Figure 5.5(d). The optimal mapping for the source component is given by the mapping *(t3, t2)* with a gate-count of 10818. Figure 5.5(e) shows the connection between data inputs and outputs of s, $t2$ and $t3$.

We conclude this example with two comments. In this example the bit-width of the optimal solution is exactly equal to the bit-width of the source component (24). This may not be true in general. Secondly, for this example the cost function per bit decreases with increasing bit-width. For such cases, the optimal mapping can be achieved with a linear time algorithm. However, in general the cost function may not necessarily follow the above behavior; our algorithm provides an optimal solution for the unrestricted bit-width mapping problem. The algorithm considers mappings with bit-widths equal or greater than the source bit-width and is independent of the cost distribution for the target modules.

5.3.5 Word mapping

Word mapping, in the context of memory mapping, refers to the task of accomplishing the word-count requirement of the source memory component s using a set (one or more) of target memory modules from a library. As in bit-width mapping, we often have to compose a set of target memory modules to meet the word-count requirement of the source memory module. Refering back to Figure 5.2, the design composes two memory modules with 512 and 256 words to realize the source memory module with 768 words. Recall that the word-count of the resultant design has to be greater than or equal to the word-count of s. The word mapping problem is very similar to the bit-width mapping problem discussed in the last section. As before, we first present an ILP formulation for a simplified

word mapping problem. Next, based on the problem domain characteristic we present an efficient algorithm to perform the task of word mapping.

An ILP formulation for word mapping

The ILP formulation for a simplified area based word mapping problem is similar to the ILP formulation for bit-width mapping. Given a source memory module s and a set $T = \{t_1, t_2, ..., t_m\}$ of target memory modules from a library, we have to find the number(x_i) of each of the target memory module t_i that minimizes the following expression:

$$\sum_{i=1}^{m} x_i A(t_i) \tag{5.6}$$

and satisfies the following linear constraint:

$$\sum_{i=1}^{m} x_i N(t_i) \geq N(s) \tag{5.7}$$

Recall that function N for a memory module refers to the number of words in the module. The above ILP formulation is a simplified version of the actual word mapping problem in the sense that Equation 5.6 does not capture the complete cost of the resultant design. It does not account for the cost of the multiplexers required to select the data output from the various modules used in the design. Refering to Figure 5.2, the design requires two 2-input 72-bit multiplexers (one for each output). Furthermore, Equation 5.6 does not include the cost of the address decoding logic. In the worst case, the address decoding logic may require adders incurring a significant increase in the total cost of the design.

The word mapping problem as defined (similar to the bit-width mapping problem) is an NP-complete problem. However, unlike bit-width mapping, the domain of word mapping is quite large, since the number of words in a memory module varies in a wide range. Thus, a simple enumerative scheme would lead to

a time inefficient solution. However, the number of words in a memory module is typically a power-of-two. A memory with word-count equal to a power-of-two provides a regular structure and leads to an efficient design. Based on this assumption, we present an efficient linear time algorithm to perform word mapping. Note that a few generator based libraries ([VTI91][Tosh90][Casc92]) do provide memory modules with a number of words not equal to the power-of-two. We can approximate these memory word-counts to the largest power-of-two less than given memory word-count. We are making the above assumption only for the target memory modules; the source memory word-count is unrestricted.

An algorithm for word mapping

Algorithm 5.3.4 : Word mapping
INPUT: Source memory module, s; Target memory module set, T;
Cost measure, C.
OUTPUT: Word mapping of s.
1 **for** each t $\in$ T **do**
1.1 N(t) = 2^n, where n is the largest integer such that $2^n \leq N(t)$;
2 **end for**;
3 sort T in decreasing word-count of its elements;
4 delete redundant elements from T;
5 *best_map* = ϕ;
6 *partial_map* = ϕ;
7 *word_left* = N(s);
8 **while** *word_left* $\neq$ 0 **do**
8.1 *curr_mem* = next memory from T;
8.2 *curr_word* = N(*curr_mem*);
8.3 *curr_map* = $\lceil \frac{word_left}{curr_word} \rceil$ *curr_mem*;
8.4 **if** C(*partial_map*+*curr_map*) < C(*best_map*) **then**
8.4.1 *best_map* = *partial_map* + *curr_map*;
8.5 **end if**;
8.6 **if** *curr_mem* is the last element in T **or** S(*curr_map*) == *word_left* **then**
8.6.1 *word_left* = 0;
8.7 **else**
8.7.1 *partial_map* = *partial_map* + $\lfloor \frac{word_left}{curr_word} \rfloor$ *curr_mem*;
8.7.2 *word_left* = *word_left* - $\lfloor \frac{word_left}{curr_word} \rfloor$ *curr_mem*;
8.8 **end if**;
9 **end while**;
10 **return** *best_map*;

Algorithm 5.3.4 describes a scheme to perform the task of word mapping efficiently. The input to the algorithm consists of the source memory module s, the target memory module set T and the user-given cost function C. The global variables *best_map, partial_map,* and *word_left* in this algorithm keep record of the current best complete mapping, current partial mapping and the number of words yet to be mapped respectively. The local variables in the **while** loop at Statement 8, namely *curr_mem*, *curr_word* and *curr_map* store the current target memory module, its word-count and the partial mapping achieved using this *curr_mem*.

The algorithm begins with approximating the target memory word-count to the largest power-of-two that is less than or equal to the memory word-count. After sorting these memory modules in the decreasing order of their word-counts, the algorithm deletes the redundant modules from T. A target memory module t_i is *redundant* if it could be composed using other modules in T with smaller cost. The major computation for the word mapping is performed in the **while** loop at Statement 8 of the algorithm. The algorithm selects *curr_mem*, the next largest memory module from T and generates two mappings:

- A complete mapping using the current *partial_map* and *curr_mem* (Statement 8.4). If the cost of this complete mapping is smaller than the current *best_map*, then *best_map* is updated.
- A new partial mapping using *partial_map* and the maximum number of complete *curr_mem* that can fit in *word_left*.

If *curr_mem* is the last module in T or *curr_map* provides an exact fit for *word_left*, the loop terminates by assigning zero to *word_left*. Otherwise, *word_left* is updated with the number of words yet to be mapped. Finally, the algorithm returns the current best mapping stored in *best_map*.

The run-time complexity of the algorithm is $O(m * log(m))$, where m is the number of elements in T. The *sort* procedure at statement 4 requires $O(m*log(m))$ iterations. The redundancy removal at Statement 4 can be performed in $O(m)$

time. Finally, the **while** loop at Statement 8, in the worst case, iterates for m times. The computation in each iteration can be performed in constant amount of time. Thus the complexity of the whole algorithm is $O(m * log(m))$. Note that the algorithm would require *linear* run-time for sorted T.

An example

Figure 5.6 shows an application of the word mapping algorithm on an example. The source component in this example is a memory module from an industrial design [KaRo94]. The target memory modules are from the Toshiba library [Tosh90]. The word-count, bit-width and gate-count for each memory module are shown in Figure 5.6(a). We use an area measure approximated by the gate-count as the cost function for the example. Furthermore, we approximate the cost of the complete mapping with the sum of gate-counts of the composing target modules. Note that each target memory module in this example has a word-count that is a power-of-two and that all these modules are irredundant.[4]

Figure 5.6(b) shows intermediate results *after* each iteration of the **while** loop in the Algorithm 5.6. In the first iteration we use $t1$ and generate a complete mapping *(t1, t1)* with cost 39816 and a partial mapping *(t1)* with cost 199008. The next iteration generates another complete mapping *(t1, t2)* with a lower cost 30968. In the following iterations, we successively use the remaining target memory modules. The process ends after the fifth iteration when the remaining words to be mapped become zero. The optimal mapping is given by *(t1, t2)* with cost 30968 (shown in bold). Note that the number of iterations for an example is bounded by the number of the target memory modules. Figure 5.6(c) shows the implementation corresponding to the mapping *(t1, t2)*. Along with the two modules $t1$ and $t2$, the implementation uses a 2-input 16-bit multiplexer and some small address decoding logic (two inverters + an AND gate).

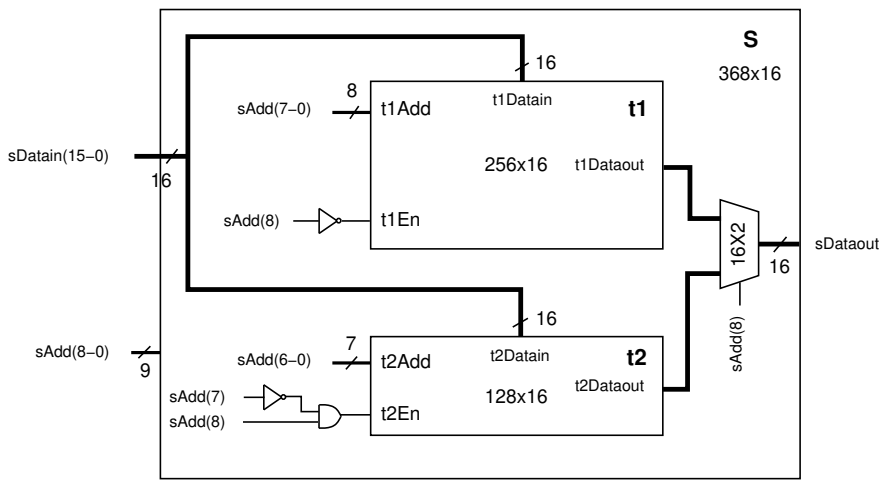
[4] A memory module is irredundant if it can't be synthesized using other memory modules with smaller cost.

Parameters	Source	Target					
	s	t1	t2	t3	t4	t5	t6
Word–count	368	256	128	64	32	16	8
Bit–width	16	16	16	16	16	16	16
Gate count		19908	11060	6636	4434	2408	1300

(a)

Iteration count	*Curr_mem*		*Best_map*		*Partial_map*		*Size_left*
	name	word	mapping	cost	mapping	cost	
1	t1	256	(t1, t1)	39816	(t1)	19908	112
2	t2	128	(t1, t2)	30968	(t1)	19908	112
3	t3	64	(t1, t2)	30968	(t1, t3)	26544	48
4	t4	32	(t1, t2)	30968	(t1, t3, t4)	30966	16
5	t5	16	**(t1, t2)**	**30968**	(t1, t3, t4, t5)	33374	0

(b)



(c)

Figure 5.6: Word mapping example (a) Source and target memory modules (b) Intermediate results from word mapping algorithm (c) Final design

We conclude this section with the following observations. If all the target memory modules have number of words equal to a power-of-two then:

- Our word mapping algorithm generates an *optimal* solution.
- Our algorithm is very efficient (*linear* time for target module set sorted by size).
- Our scheme implements the address decoding logic with very little logic without using the "adder" logic.

5.3.6 The cost function

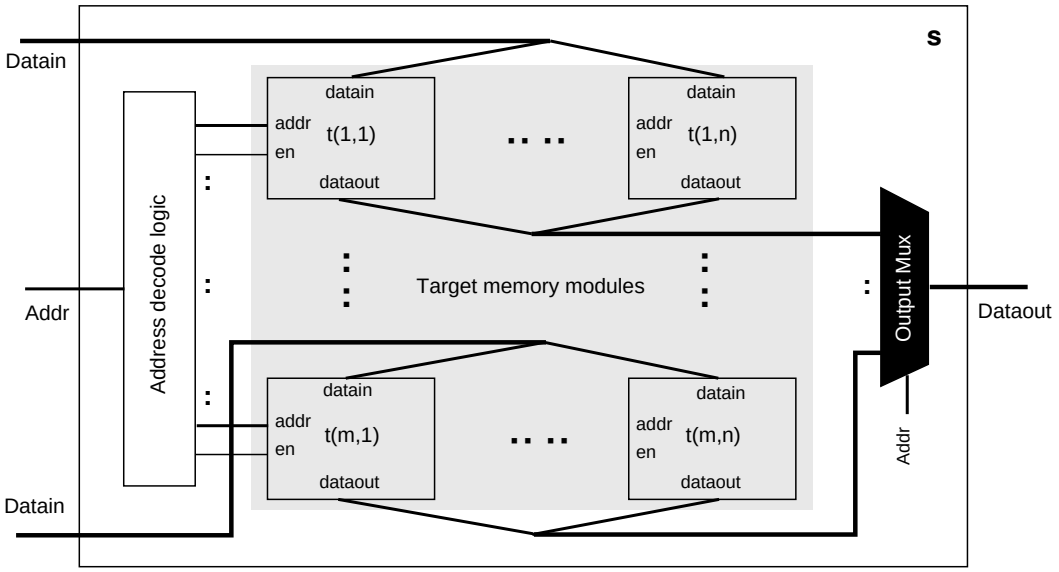


Figure 5.7: Cost of a memory design

The memory mapping algorithms are guided by the cost of the generated design. The cost of the synthesized source memory module is given by the combined cost of the various elements used in the design. We illustrate these elements through an example shown in Figure 5.7, where the design consists of three components:

- **Address decode logic,** shown in the white box, consists of the logic that translates the address lines of the source memory module into the address and the enable lines of the target memory modules. In our mapping approach, this logic is usually small, in the order of a few gates.
- **Target memory modules,** shown in the lightly shaded box, consists of an array of target memory modules. These modules together satisfy the word-count and the bit-width requirements of the source memory module. The example in Figure 7 uses m*n modules.
- **Output mux,** shown in the black box, multiplexes the output data from target memory modules. The number of inputs and the bit-width of the multiplexer is given by the number of target memory modules (m) used in the word mapping and the bit-width of the source memory module respectively.

The cost of a memory design is given by the cumulative cost of the constituent elements:

$$C(s) \quad = \quad C(address) + C(target) + C(mux) \tag{5.8}$$

Here, $C(address)$, $C(target)$ and $C(mux)$ refer to the cost of address decoding logic, the target memory modules and the output mux respectively. We refine these terms to generate a specific cost measure.

- **Area measure :** The area measure for the address decode logic and the output mux is given by the approximate area required by these two elements. The area measure for the target memory modules is given by the sum of the area of all the target memory modules used in the design:

  $$A(target) = \sum_{i=1}^{m} \sum_{j=1}^{n} A(t(i,j))$$

  Note that we have to use the same unit for the area measure (e.g., sq-micron or gate-count) for the different elements of a memory design.

- **Delay measure :** The worst case delay path for the synthesized memory goes through all the three components in the design. The delay measure for the address decode logic and the output mux is given by the worst case delay through these modules. The delay measure for the target modules is given by the maximum access delay for all the modules used in the design:

$$D(target) = Max_{i=1}^{m} Max_{j=1}^{n} D(t(i,j))$$

Note that the cost of a memory design would also include the port mapping and data routing cost. For the sake of simplicity, we ignore these costs in our cost measure.

5.4 Memory Mapping Approach

Now we present the overall approach to the memory mapping problem. The approach combines the various schemes presented in the last section to solve the comprehensive memory mapping problem. Specifically, it uses the port mapping, the bit-width mapping and the word mapping routines to build a complete memory mapping algorithm. In this section, we first describe the basic assumptions underlying our approach. Next we present a memory mapping algorithm and illustrate the algorithm with an example.

5.4.1 Assumptions

Our memory mapping formulations make the following assumptions.

1. Each target memory module is of size equal to a power-of-two.
2. Only regularly structured memory composition is considered. In a regularly structured memory composition all the target modules in a row have the same size and all the modules in a column have the same bit-width (Figure 5.8).

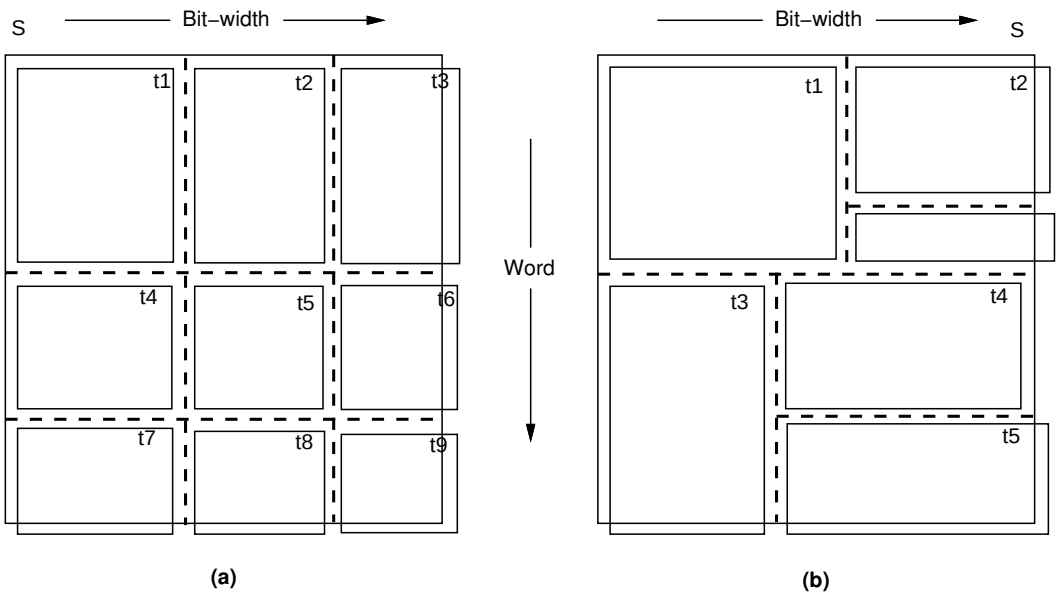


Figure 5.8: Two types of memory composition (a) Regular composition (b) Irregular composition

3. The bit-width of the source memory module is small.
4. The port name mapping for the source and the target modules are perform ed by the user.
5. The cost of a memory design does not include the port mapping and the data routing cost.
6. The timing diagrams of the source and the target modules are compatible .
7. Delay of a memory module is given by the worst case access time.

These assumptions can be classified into three categories:

- Assumptions that are induced by current day design methodologies (e.g., 2, 3). Our algorithms are general; they perform well for the restricted problem space induced by these assumptions.
- Assumptions that simplify the memory mapping problem with the assistance of a user (e.g., 4).

- Assumptions that need to be addressed in order to have a comprehensive solution to the general memory mapping problem (e.g., 2, 5, 6, 7).

Note that as a first step towards solving the complete memory mapping problem, these assumptions are reasonable.

5.4.2 Memory mapping algorithm

Algorithm 5.4.1 describes the steps in the memory mapping algorithm. The inputs to the system are the source memory module (s), the target memory module set (T) and the cost function (C). The algorithm generates a mapping of s using modules from T. The target memory module sets in this algorithm, namely T_p, T_{size} and T_{bit} store the list of modules after port constraint satisfaction, word mapping and bit-width mapping respectively. Variables *common_bit, common_word* keep record of the list of common bits and word-counts for a set of target memory modules. Variable T_{common} lists modules with common bit-widths or word-counts; variables *sol_size, sol_bit* and *sol* store complete solutions.

Algorithm 5.4.1 : Memory mapping algorithm
INPUT: Source memory module, s; Target memory module set, T; Cost measure, C.
OUTPUT: Mapping of s with T.
1 T_p = Port constraints(s, T);
2 T_{word} = Word mapping(s, T_p, C);
3 *common_bit* = Common bit-widths for modules in T_{size};
4 $T_{common} = \{t_i | t_i \in T_p$ and B(t_i) $\in$ *common_bit*$\}$;
5 *sol_word* = Bit mapping(s, T_{common}, C);
6 T_{bit} = Bit mapping(s, T_p, C);
7 *common_word* = Common sizes of modules in T_{bit};
8 $T_{common} = \{t_i | t_i \in T_p$ and S(t_i) $\in$ *common_word*$\}$;
9 *sol_bit* = Word mapping(s, T_{common}, C);
10 *sol* = Min_cost(*sol_word, sol_bit*);
11 *sol* = Port assignment(s, *sol*);
12 **return** *sol*;

The algorithm considers two solutions. In the first solution, it performs the word mapping first followed by the bit-width mapping. In the second solution, it performs the bit-width mapping followed by the word mapping. Finally, it chooses the best out of these two solutions. The algorithm starts by selecting the subset of modules from T that satisfies the port constraints (Step 1) specified by Equations 5.1, 5.2 and 5.3. Then it performs word mapping (Step 2). The first solution is achieved by performing the bit-width mapping on the set of modules returned from the last step that have common sets of bit-widths (steps 3-5). Similarly the algorithm generates the second solution by performing the word mapping and the bit-width mapping in the reverse order (Steps 6-9). The best solution is given by the design with the minimum cost. The mapping is completed by performing the port assignment on the best solution. The Algorithm 5.4.1 is log-linear with respect to the number of modules used for word-count expansion and exponential with respect to the number of the modules used for bit-width expansion.

An example

Figure 5.9 shows an example for memory mapping. The source memory module is a part of a medical image reconstruction [BaCM94] algorithm. There are eight target memory modules *(t1-t8)* from the Toshiba [Tosh90] library. Figure 5.9(a) shows the word-count, the bit-width, the ports and the gate-count for these memory modules. The source module and each of the target modules have single read-write ports. The cost function for this example is an area measure approximated by the equivalent gate-count.

Figures 5.9(b) and 5.9(c) trace the two paths through the memory mapping algorithm. Figure 5.9(b) shows the intermediate results for the path that perform word mapping followed by bit-width mapping. All the target modules in this example satisfy the port constraints, as shown by the variable T_p in this figure. The result of the word mapping on T_p is a two module *(t1, t3)* solution. The

Parameters	Source	Target							
	s	t1	t2	t3	t4	t5	t6	t7	t8
Size	384	256	256	128	128	128	64	64	64
Bit–width	12	8	4	12	8	4	12	8	4
Ports	1 RW	1 RW	1 RW	1 RW	1 RW	1 RW	1 RW	1RW	1RW
Gate count		11562	7564	8680	6642	4636	5166	4182	3172

(a)

Tp = {t1, t2, t3, t4, t5, t6, t7, t8}

Tword = (t1, t3)

Common_bit = {8, 4}

Tcommon = {t1, t2, t4, t5, t7, t8}

sol_word = (t1, t2, t4, t5)

Gate–count = 2+30404+39 = 30445

Tp = {t1, t2, t3, t4, t5, t6, t7, t8}

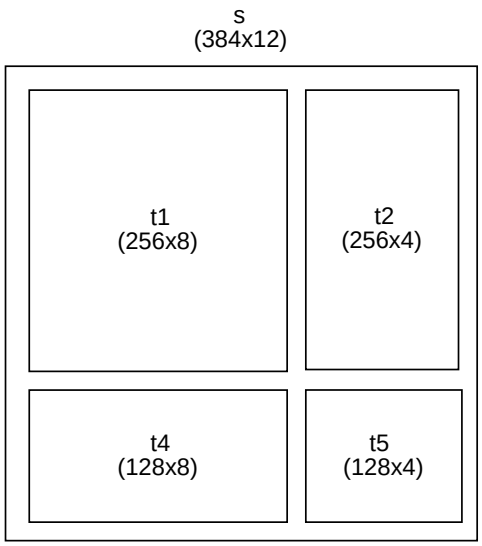
Tbit = (t3)

Common_bit = {12}

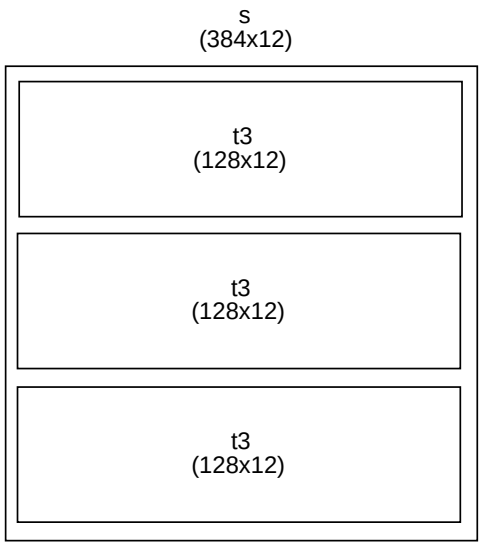
Tcommon = {t3, t6}

sol_bit = (t3, t3, t3)

Gate–count = 2+26040+42 = **26084**



(b)



(c)

Figure 5.9: A memory mapping example with *linear* algorithm (a) Source and target modules (b) Design with word mapping followed by bit-width mapping (c) Design with bit-width mapping followed by word mapping

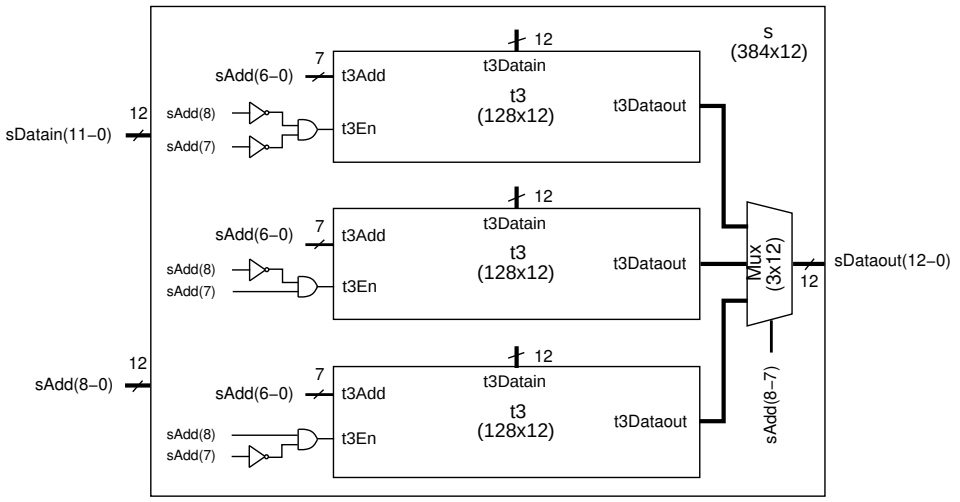


Figure 5.10: Complete design for the memory mapping example

common set of bit-widths for the modules with word-count equal to the word-count of *t1* or *t3* is {8,4}. Thus, the following modules {*t1, t2, t4, t5, t7, t8*} participate in the bit-width mapping. The final solution with this path is given by the composition of (*t1, t2, t4, t5*) and the layout is shown in the bottom portion of the Figure 5.9(b). The cost of this design is 30445 gates.

Figure 5.9(c) walks through the other path that performs the two tasks in the reverse order: bit-width mapping followed by word mapping. The memory design generated by the bit-width mapping algorithm uses a single module *t3*, since it has an exact match to the bit-width (12) of the source module. There are only two modules, namely {*t3, t6*} with bit-width equal to 12. The word mapping on this set results in a design with three instances of *t3*. The layout of the resultant design is shown at the bottom of Figure 5.9(c). The cost of this design is 26084 gates. Thus, the second design is preferred. The complete structure for this design that includes the address decode logic as well as the output mux is shown in Figure 5.10. The port assignment task is straight forward in this example, since there is a one to one match between the ports of the source and the target memory modules.

5.5 Experimental results

Library : Toshiba gate array **Cost fn :** Area

Example			Mapping result				
#	Name	Size	Design	Area		Run–time	
				(2–nand)	% Diff wrt Exhaustive	(sec)	% Diff wrt Exhaustive
1	Differential Heat Release (Ramachan,EDAC 94)	469x16	256x8, 256x8 256x8, 256x8	46331	**6.02**	1.1	–60.71
2	Neural Network Chip (Rose, ICCAD 94)	160x8	128x8 32x8	9035	**18.40**	1.2	–14.28
3	Interlace Scan (Philips, CICC 91)	360x16	256x8, 256x8 128x8, 128x8	36491	**4.32**	1.4	–26.31
4	Medical Image Reconstruction (Balasa, ICCAD 94)	384x12	256x6, 256x6 128x6, 128x6	30319	**3.06**	2.0	+17.64
5	MPEG I (Thordarson, TR 95–8)	54x18	64x6, 64x6, 64x6	11017	**18.14**	2.2	+29.41
6	MPEG II (Thordarson, TR 95–8)	128x17	128x5, 128x6, 128x6	16383	**0.00**	1.2	–20.00

Figure 5.11: Memory mapping result I

In this section we present a summary of memory mapping results from our algorithm on several examples from the literature and compare these designs against the ones produced by an exhaustive memory mapping algorithm. [JhD95b] presents an extended version of these results.

Our experiments use source memory modules derived from various memory-intensive designs reported in the literature as well as from industrial designs. Specifically, we report mapping results for six examples in Figures 5.11 through 5.13 and for another four examples in Figure 5.14. [JhD95b] describes the source designs for these memory modules. These examples cover wide variety of memory modules both in terms of the source of the design as well as the size of the modules, specifically with respect to the word-count and bit-width variation.

Each of the tables in Figures 5.11 through 5.14 describe the memory mapping results generated by our mapping algorithms on a set of the source memory modules. The first three columns in these tables describe the source memory module.

Library : Xilinx 4000 rams **Cost fn :** Area

Example			Mapping result				
#	Name	Size	Design	Area		Run–time	
				(clb)	% Diff wrt Exhaustive	(sec)	% Diff wrt Exhaustive
1	Differential Heat Release (Ramachan,EDAC 94)	469x16	128x8, 128x8 128x8, 128x8 128x8, 128x8 64x8, 64x8 32x8, 32x8	333	**1.52**	1.8	–92.50
2	Neural Network Chip (Rose, ICCAD94)	160x8	128x8 32x8	55	**3.77**	0.9	–25.00
3	Interlace Scan (Philips, CICC 91)	360x16	128x8, 128x8 128x8, 128x8 64x8, 64x8 32x8, 32x8 16x8, 16x8	257	**0.00**	0.9	–67.85
4	Medical Image Reconstruction (Balasa, ICCAD 94)	384x12	128x4, 128x8 128x4, 128x8 128x4, 128x8	206	**1.98**	0.9	–68.96
5	MPEG I (Thordarson, TR 95–8)	54x18	32x4, 32x8, 32x8 32x4, 32x8, 32x8	50	**0.00**	0.8	–0.00
6	MPEG II (Thordarson, TR 95–8)	128x17	128x4, 128x8, 128x8	107	**8.08**	0.8	–0.00

Figure 5.12: Memory mapping result II

For each source module, we list the name of the design from which the memory module was extracted, the source of the design and the size of the memory module. Columns 4 through 8 describe the mapping result. The fourth column (labeled Design) lists the arrayed configuration of the target memory modules synthesized by our memory mapping algorithm; the fifth column displays the design metric (*Area, Delay or Price*) and the seventh column presents the run-time (in seconds) for our algorithm on a SUN Sparc Station 5. The sixth column reports the percentage difference between the design metric for the results produced by our mapping algorithm and an exhaustive algorithm.[5] Similarly, the last column reports the percentage difference between the run-time of our algorithm and the exhaustive algorithm. In each table we also report the name of the target library, the cost function used to generate the designs.

Figures 5.11 shows the mapping results for area-efficient designs generated by our algorithm. These designs have been synthesized using the memory modules

[5] [JhD95b] describes an exhaustive memory mapping algorithm.

Library : Xilinx 4000 rams **Cost fn :** Delay

Example			Mapping result				
#	Name	Size	Design	Delay		Run–time	
				(ns)	%Diff wrt Exhaustive	(sec)	%Diff wrt Exhaustive
1	Differential Heat Release (Ramachan,EDAC 94)	469x16	32x4, 32x4, 32x4, 32x4 : 32x4, 32x4, 32x4, 32x4 16x4, 16x4, 16x4, 16x4 16x4, 16x4, 16x4, 16x4	31.90	**0.00**	1.0	–83.87
2	Neural Network Chip (Rose, ICCAD94)	160x8	32x8 32x8 32x8 32x8 32x8	28.60	**0.00**	0.6	+20.00
3	Interlace Scan (Philips, cicc91)	360x16	32x4, 32x4, 32x4, 32x4 : 32x4, 32x4, 32x4, 32x4 16x4, 16x4, 16x4, 16x4	31.90	**0.00**	1.3	–23.52
4	Medical Image Reconstruction (Balasa, ICCAD 94)	384x12	32x8, 32x8 32x8, 32x8 : 32x8, 32x8	31.90	**0.00**	0.6	–70.00
5	MPEG I (Thordarson, TR 95–8)	54x18	32x4, 32x8, 32x8 32x4, 32x8, 32x8	21.00	**0.00**	0.9	+50.00
6	MPEG II (Thordarson, TR 95–8)	128x17	32x8, 32x8, 32x8 32x8, 32x8, 32x8 32x8, 32x8, 32x8 32x8, 32x8, 32x8	23.50	**0.00**	0.7	–65.00

Figure 5.13: Memory mapping result III

taken from the Toshiba gate array library[Tosh90]. This library contains single port RAMs with word-count varying in the range of 8 to 256 and bit-width in the range of 4 to 8. We report the area of the mapped designs in terms of equivalent 2-input nand-gates. Figures 5.12 and 5.13 describe the mapping results using the RAM macro modules of the Xilinx 4000 series FPGA[Xili93]. This library contains a set of single port RAM modules with the word-count equal to 16, 32, 64, or 128 and the bit-width equal to 2, 4 or 8. Figure 5.12 reports the area-efficient designs, whereas Figure 5.13 reports the delay-efficient designs generated by our algorithm. We report area in terms of approximate number of CLBs (Configurable Logic Blocks [Xili93]) for the resultant design. The delay for these designs represents the worst case access delay in nanoseconds. Figure 5.14 lists the price-efficient mapping results using the off-chip ROMs provided by Texas Instruments [Ti94]. The library contains the following 6 modules : 32Kx8, 64Kx8, 128Kx8, 64Kx16, 256Kx8 and 512Kx8. We report the dollar cost required to synthesize these designs.

Library : TI roms **Cost fn :** Price

Example			Mapping result				
#	Name	Size	Design	Price		Run–time	
				($)	% Diff wrt Exhaustive	(sec)	% Diff wrt Exhaustive
1	Singular Value Decomposition II (Balasa, ICCAD 94)	146590x16	256kx8, 256kx8	16.36	**0.00%**	0.3	0.00
2	Singular Value Decomposition III (Balasa, ICCAD 94)	103550x16	128kx8, 128kx8	10.90	**0.00%**	0.4	+33.33
3	Image Processing (Lim, Prentice Hall 90)	1Mx16	512kx8, 512kx8 512kx8, 512kx8	65.44	**0.00%**	0.3	0.00
4	MPEG III (Thordarson,TR 95–8)	512kx32	512kx8,512kx8,512kx8,512kx8	65.44	**0.00%**	0.3	0.00

Figure 5.14: Memory mapping result IV

5.5.1 Analysis of experimental results

From the results in Figures 5.11 to 5.14, we observe that our approach to memory mapping is quite comprehensive. Our approach can synthesize source memory modules of varying complexity. These memories are of varying word-count and bit-width. The size of the source memory modules in our examples vary in the range of 972 bits (54x18) to 16Mbits (1Mx16). These memory modules come from designs of varying complexity including image processing applications and an industrial design of MPEG algorithm. In our experiments we have covered three target libraries. These libraries include on-chip modules (Toshiba gate array [Tosh90] and Xilinx 4000 RAM modules [Xili93]) as well as off-chip stand alone memory modules (Texas Instruments ROMs [Ti94]). Each library contains a varying number of memory modules. The memory modules in these libraries are also of varying sizes. Starting from smaller modules of size 8x4 size in the Toshiba gate array, the modules of TI ROMs are as big as 512Kx8. Our approach allows the user to select the optimizing cost function. We currently support three cost functions, namely *Area, Delay* and *Price*. Hence, our memory mapping approach is quite general in the sense that it supports a wide variety of user selectable design parameters such as the source component, the target library, the mapping

algorithm and the optimizing cost function. The mapping results in Figure 5.14 demonstrate that our mapping approach is applicable to off-chip modules as well.

From the tables in Figures 5.11 to 5.13, we observe that our approach generates a wide variety of memory designs. These designs include regular structures, where a memory array is built using a single memory module type (Example 1 in Figure 5.11) as well as irregular structures, where a memory array is built using a set of different memory module types (Example 1 in Figure 5.12). We also note that even for the same memory module, we get different designs as we vary the cost function and the target library. For instance, consider the first example (Differential Heat Release computation) in Figures 5.11 to 5.13. All of these three mapping results are different. In other words, a different memory configuration is generated for different optimizing cost functions and the target libraries.

Also, quite often the resultant designs are counter-intuitive. For instance, consider Example 6 in Figure 5.11. The source memory module is taken from an MPEG design and is of size 128x17. This has been synthesized with a row of the following modules: 128x5, 128x6 and 128x6. We would expect a design that uses modules with bit-widths that are powers-of-two such as 128x8, 128x8, 128x4; however the area of this design is larger than the counter-intuitive design generated by our algorithm. This illustrates the utility of our approach in generating a wide variety of designs, often counter-intuitive ones, based on the cost function and the algorithms.

The sixth column in these figures compares the quality of designs produced by our approach against the ones generated by an exhaustive algorithm. Our designs are quite close to ones generated by the exhaustive algorithm. Designs reported in Figures 5.13 and 5.14 are with equal metric, whereas designs in Figure 5.11 are at most 18.40% worse and designs in Figure 5.12 are at most 8.08% worse in area. The higher difference in the design metric for the results with the Toshiba gate array library in Figure 5.11 is because this library contains few modules with word-count not equal to a power-of-two. The exhaustive algorithm has performed

well by making an effective use of these modules. However, for libraries that have restricted module sets with word-counts equal to a power-of-two (e.g., Xilinx RAMs and Texas Instruments ROMS), our algorithm performs very well; most of the designs have metrics equal to those generated by the exhaustive algorithm (Figures 5.13 and 5.14).

If we analyze the run-time (the seventh column in these tables), we observe that we have been able to generate these designs very quickly, in the order of a few seconds. The run-time is independent of the cost function used for optimizing the design. The last column in these tables compares the run-time of our algorithm with the run-time of the exhaustive algorithm. We observe that the run-time of our heuristic is comparable to the run-time of the exhaustive algorithm for the examples reported. There are two reasons for this phenomenon. First, the figures for run-time include the time to execute the mapping algorithm as well as the I/O time which is dependent on various factors such as network conditions, file-server status, etc. Because of such factors, the run-time varies with different executions of an example on the same machine. This is one of the reasons why our heuristic sometimes requires longer run-times as compared to the run-times required by the exhaustive algorithm. Thus, one should use the run-time figures with caution for smaller examples. Secondly, the source examples presented in these tables are relatively small in size. For examples with bigger word-counts, the exhaustive algorithm would require exponentially longer run-times, as compared to our heuristic that solves the word-mapping problem with linear run-time complexity. In fact, [JhD95b] reports an example that requires approximately 10 hours to generate a design with the exhaustive algorithm, but which requires only 2.5 seconds with our algorithm.

5.6 Summary

In this chapter, we presented a memory mapping scheme that implements a source memory module with a set of target memory modules from a library. The approach has applications in two domains: synthesizing the logical memories generated by high-level synthesis and synthesizing a source memory from one library using modules from another library (e.g., retargetting memory designs). Our approach facilitates design reuse for memory subsystems. High-level library mapping for memories could be used to synthesize on-chip as well as off-chip memory modules.

We have identified and formulated three subproblems associated with the memory mapping problem (port mapping, bit-width mapping and word mapping) and composed these formulations into a complete memory mapping approach. We combined these three formulations into an efficient memory mapping algorithm that can be used for generating area-optimized, delay-optimized or price-optimized designs. Our experimental results run on several industrial and literature-based examples demonstrate that HLLM for memories can generate a wide variety of cost-effective memory designs, often conter-intuitive ones, based on the user-given cost function and the target library.

Chapter 6

GENUS and HLLM Environment

In this chapter, we first present the GENUS environment that implements the generic RT component set presented in Chapter 3. Supplemented by the model generators, technology projectors and technology mappers, the environment presents a comprehensive system for RT design. We then briefly describe a user-interface for the HLLM system.

6.1 GENUS Environment

Figure 6.1 shows the structure of the GENUS environment. The core of the GENUS environment is a set of RT level component generators. The automatic model generators of GENUS system provide three kinds models: Functional models, Delay models and Synthesis models. The *Functional models* and *Delay models* are behavioral models used to validate a RT design using commercial simulators. *Synthesis models* are used to refine the RT design design description into logic or layout-level design. *Technology projectors* provide rapid estimates of generic component implementation in a specific technology to perform design space exploration and tradeoff analysis at higher levels. Finally, the *High-Level Library Mapper* maps a RT component onto technology specific RT components. The GENUS system supports high-level design by providing a mechanism to validate its output as well as to link the output with physical design. By providing a library of reusable parts along with a library mapper, GENUS supports design reuse as well.

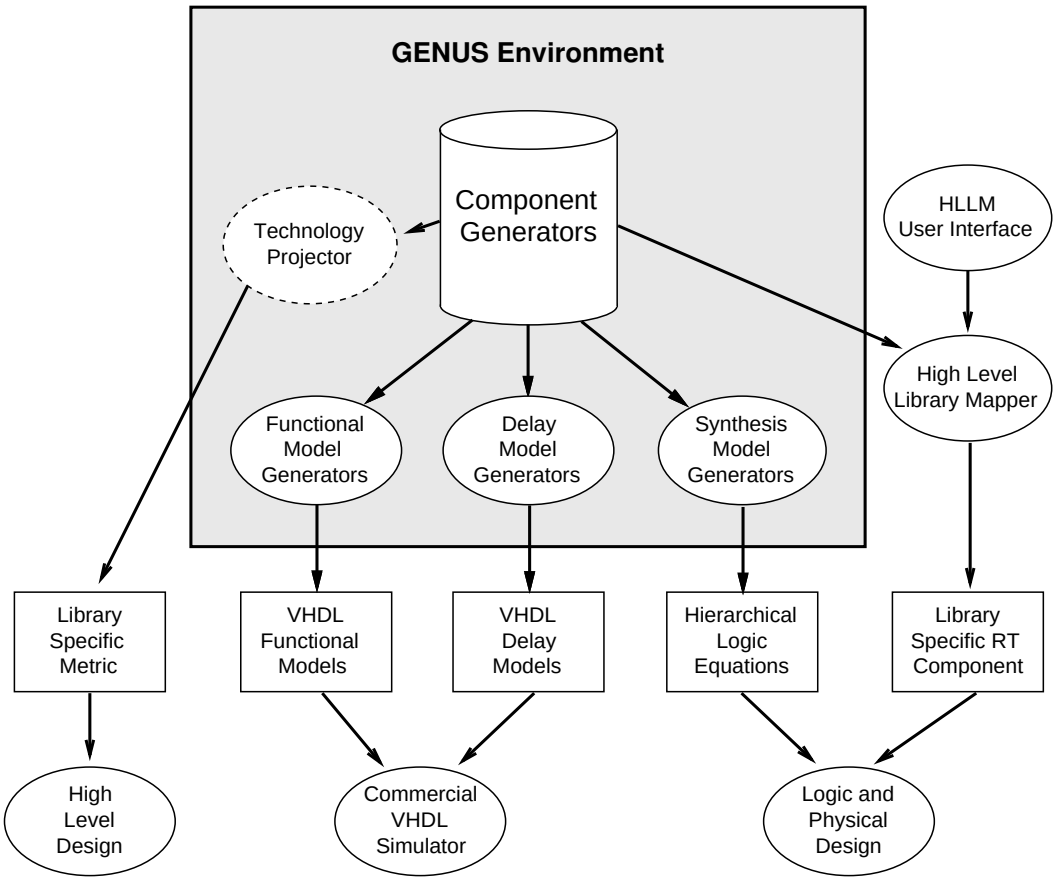


Figure 6.1: GENUS library structure

The GENUS environment has been implemented in "C" and contains approximately 50,000 lines of code. An extensive programming library developed for GENUS provides a set of routines to create, delete and query information regarding a specific component [JhD94a]. As shown by the solid line boxes in Figure 6.1, the component generators and the model generators have been implemented completely. As shown by the dotted box, the technology projectors have not been implemented within the GENUS environment, but the conceptual background as well as the area-delay estimation models for two technologies have been developed. The high-level library mappers form an independent system; they use the RT component definitions provided by the GENUS environment.

6.2 Model Generation

Design	Components	Lines of VHDL code
AM2901	ALU, Reg–file, Reg, Mux	1078
AM2910	ALU, Reg–file, Reg, Mux	854
CB Int	ALU, Reg, Div, Mux	1311
Kalman	ALU, Reg–file, Reg, Mux	1341
Clk–div	ALU, Reg, Mux	476
Timer	ALU, Reg, Mux	605

Figure 6.2: Behavioral VHDL simulation models

The GENUS environment provides both automatic simulation model generation and automatic synthesis model generation for each component. The simulation models correspond to behavioral VHDL processes representing functional behavior of the parameterized RT component. The user can also ask for a block-delay model that generates a behavioral VHDL simulation model with a nominal delay

for each instantiated component. Since the RT components are generic and are typically used in HLS tasks such as scheduling and allocation, this delay model is often sufficient to support both the design task, as well as the ensuing task of functional verification. The synthesis models for each RT component correspond to Boolean logic equations in a standard form (EQN) for combinational components, and sequentialized EQN for sequential components. These Boolean equations can be used to synthesize generic components from logic-level primitives.

The VHDL simulation models are generated in the order of a few seconds; Figure 6.2 shows that, even for small examples, the number of VHDL code lines runs into the thousands. Automatic simulation model generation thus obviates the burdensome task of VHDL code generation and validation, since these model generators have already been validated and tested. The same situation holds for automatic synthesis model generation.

6.3 Technology Projection

We perform technology projection for different libraries using closed form functions to estimate the area and delay for a component in the generic library. These area and delay models are functions of the parameters such as bit-width, set-of-functions, etc. required to instantiate a specific component belonging to the generator.

One significant parameter for a generator is the component's bit-width, which clearly has a significant impact on the estimation models. Tyagi [Tyag90] has developed an information-theoretic model that relates a module's parameters (e.g., bit-width) with its performance metrics. Based on the communication between n-bit slices of a component, he classifies some combinational components into categories and provides a formulation of models for each category. These models

describe the asymptotic behavior of area and delay with respect to major parameters of the generators. For example, both the area and delay of a ripple-carry adder vary linearly with respect to the bit_width of the inputs. We use a similar formulation for the primary factors of the area-delay models that account for the major contribution towards the performance metrics of a component. We then use a least square approximation method on a set of sample design implementations, to determine the coefficients of the formulated estimation equations. The set of components that are to be used for calculating the coefficients should be fairly representative of all the possible components for a generator. We attempt to select the set of components so as to avoid a bias towards any particular parameter or any particular subset of values for a parameter. [JhDu92] presents a detailed description of the technology projection approach.

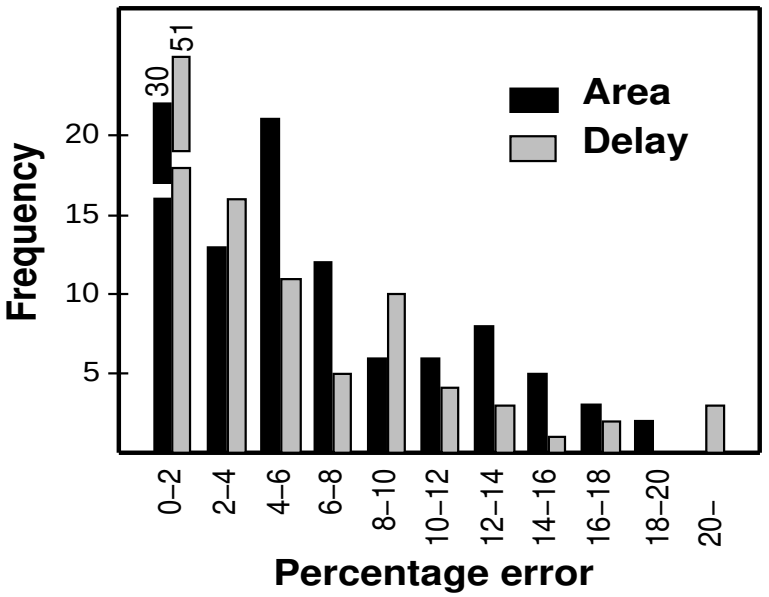


Figure 6.3: Aggregate error profile as compared to DTAS

We ran several experiments to test our models. We compared the estimates generated by our model against the metrics derived from the design structures generated by DTAS [Kipp91], LAST [KuRa91] and TELE [RaKu92]. The area values provided by DTAS counts the number of equivalent two-input NAND gates

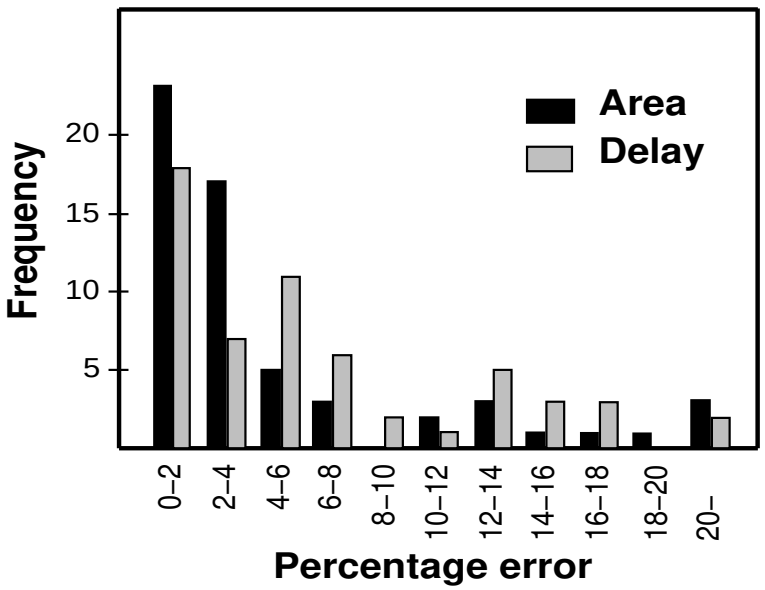


Figure 6.4: Aggregate error profile as compared to LAST/TELE

used to implement the component. For a component's delay (measured in nanoseconds), DTAS returns the worst-case delay for all paths through the design. LAST and TELE provide area and delay values respectively, based on GDT 3μ CMOS standard cell technology.

Our experiments attempted to cover a wide range of possible component implementations. We did this by generating parameter values randomly for each component generator. The number and set of functions (for multi-function components) were also chosen randomly. For each such randomly chosen component, we ran our models, and compared the results with an actual design generated by above mentioned tools. We considered the following generators: Logic gates, Multiplexer, Comparator, LU, Adder, ALU and Shift-register.

Figures 6.3 and 6.4 show the aggregate percentage error profiles across all the generators in consideration; the detailed results for each generators can be found in [JhDu92]. From Figure 6.3 we observe that with respect to DTAS roughly one-third of the area and roughly half the delay data points exhibit an error of less than

two percent. After this huge concentration in the range of 0-2%, the frequency of error tapers off as the error increases. For area, *77% percent of the test points have error less than 10 percent* and *95% test points have errors less than 16 percent.* For delay, the figures are 87% and 94% respectively. Figure 6.4 exhibits a similar trend.

Since our estimation models use only a few additions, multiplications and logarithmic operations, the area-delay estimates are generated very quickly (in the order of microseconds). There is a significant improvement in the run-time as compared to the run-time of DTAS and LAST/TELE. For example, DTAS requires a run-time of approximately 8 minutes to generate and calculate area-delay values for a 64-bit ALU[Kipp91]. The estimation time required by our approach is orders of magnitude lower than the time required by DTAS. Thus, we tradeoff accuracy of the metrics (i.e., $\pm 10\%$) for real-time evaluation of the estimates.

6.4 HLLM User Interface

We now describe the graphical user-interface implemented for the HLLM system. Figure 6.5 shows the structure of the HLLM system. The input to the system are the source component, the target component or the library and the mapping attributes. The mapping attributes include the optimizing cost function, the specific algorithm to be used for mapping and the set of mapping rules. The output of the system is a mapped design that uses one or more target components along with some glue logic.

The user interface to the HLLM system consists of an X window [Youn94] graphical front-end that assists a user to interact with the core of the HLLM system. The interface allows a user to perform mapping for the two classes of components discussed in the last two chapters (ALUs and memories). We briefly

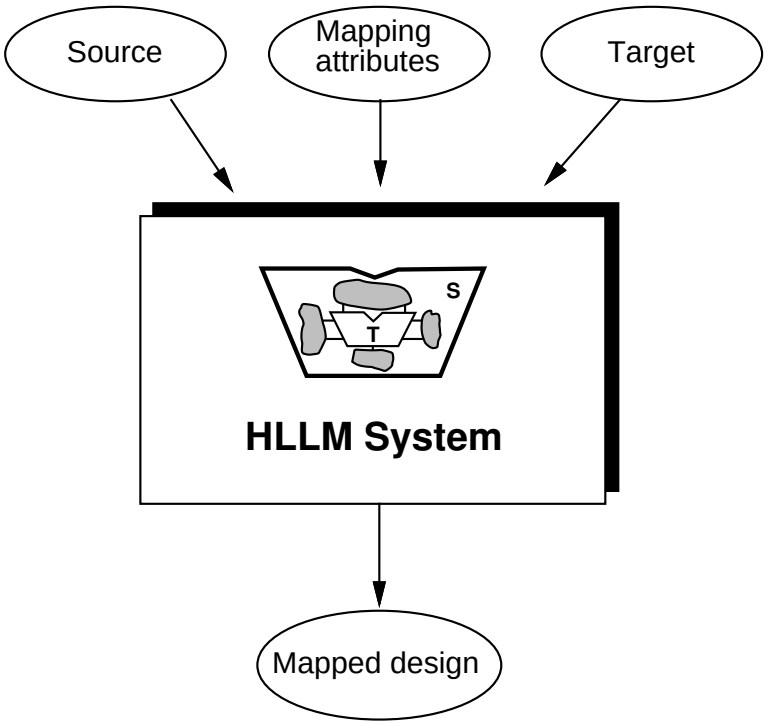


Figure 6.5: HLLM system

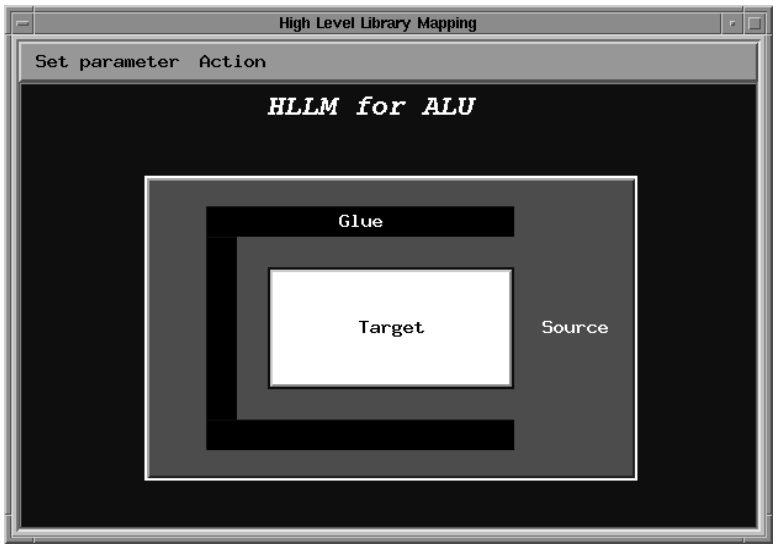


Figure 6.6: Top level window for ALU mapping

illustrate this user-interface with ALU and memory mapping front-end windows. [JhD95a] provides a detailed description of the HLLM user-interface.

Figures 6.6 and 6.7 show the top level windows for ALU and memory mapping respectively. In these windows, the menu bar is at the top and contains two pull-down menus titled: *Set parameter* and *Action*. The *Set parameter* menu is used to set the various design parameters before activating the actual mapping algorithm. The *Action* menu is used to control the overall HLLM system. Figure 6.6 shows a generic ALU mapping design where a source ALU is implemented with a target ALU along with some surrounding glue logic. Similarly, Figure 6.7 shows a generic memory mapping design where a source memory module has been implemented with an array of target memory modules along with address decode logic and multiplexer logic.

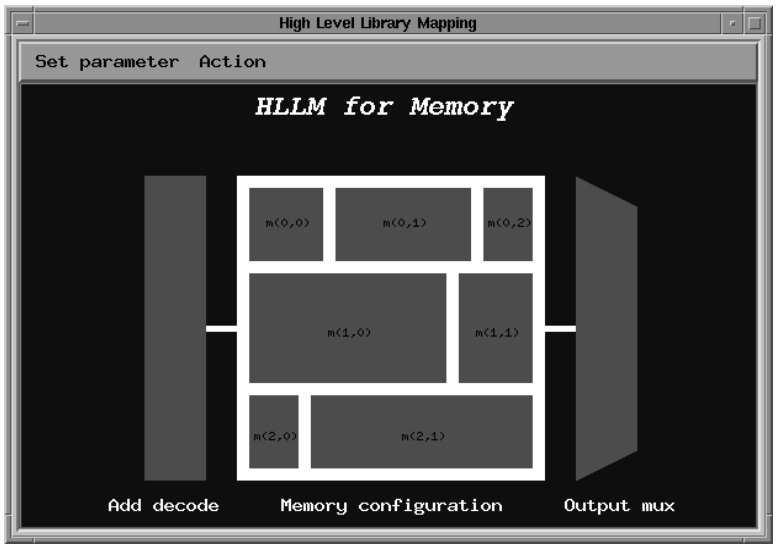


Figure 6.7: Top level window for memory mapping

6.4.1 Setting design parameters

Before activating the actual HLLM mapping algorithm, a user needs to specify values to the following parameters:

- **Component type** A user needs to select a component type out of the two options: *ALU* and *Memory*.
- **Source** A user specifies the filename containing the description of the source component. [JhD95a] describes the format for ALU and memory source components.
- **Target** A user needs to specify the filename for the target component or library. HLLM for ALU maps the source ALU on a target ALU, while HLLM for Memory maps the source memory onto a set of memory modules from a library. [JhD95a] describes the format for each of these with examples.
- **Mapping rules** HLLM for ALU requires a set of mapping rules to implement the missing functions in the source component. A user needs to specify the filename containing the list of mapping rules to be used for ALU mapping. [JhD95a] shows an example of sample mapping rules.
- **Algorithm** A user needs to select a mapping algorithm. Algorithm *greedy* and *dyn-prog* could be applied on ALUs and algorithms *linear* and *exhaustive* on Memories. The *greedy* algorithm generates a run-time efficient mapping, whereas the *dyn-prog* (short for dynamic programming) algorithm generates usually better mappings at the cost of longer run-times. Similarly, the *linear* algorithm generates run-time efficient mappings, whereas the *exhaustive* algorithm generates usually better mappings at the cost of potential exponential increase in run-time. Refer [JhD94b] and [JhD95b] for details of these algorithms.
- **Cost function** Finally, the user needs to specify the cost function to guide the mapping algorithm. He/she could choose either of the two cost functions, namely *area* or *delay* to guide an ALU and memory mapping algorithm; memory mapping supports an additional cost function: *price*.

6.4.2 Performing HLLM

The *Run* button of the *Actions* menu activates the mapping algorithm. It asks for the filename to store the final result. While the system is performing the mapping, it also displays run-time scripts specifying the progress in the mapping process. Besides updating the progress in the mapping process, the script also displays the design metrics for the ALU mapping result.

6.4.3 Displaying mapped design

The *Display* button of the *Action* menu displays the final design. The output design from the ALU mapping is a VHDL netlist of two components, namely the target ALU component and the glue logic component. This VHDL description is compatible with the Synopsys Design Compiler [Syno92] input; [Syno92] can be used to optimize the glue logic. The result of memory mapping is an array of target memory modules along with the address decode logic and the output multiplexers. Finally, the design metric for the mapping result is displayed by clicking on the *Metric* button under the *Display* button of the *Action* menu. The display shows area, delay or price of the resultant design. [JhD95a] shows examples of mapping results and design metrics for ALU and memory mapping.

6.5 Summary

In this chapter, we first presented the GENUS environment that provides a comprehensive system for RT level design. We focused on the automatic model generation technique for simulation and synthesis, as well as a technology projection scheme to link physical design-level information using accurate on-line estimators for the area and delay of the RT component generators. The simulation and synthesis model generators increase designer productivity, since the models

are generated automatically in the order of seconds via component parameters. Furthermore, the model generators reduce design errors as compared to the tedious process of manual model generation, since the generators are pretested and encapsulated. Our estimation models can handle the area/delay contributed by functional blocks as well as the total area/delay including the wiring. We have demonstrated the estimation technique on both combinational and sequential RT components with aggregate errors in the range of $\pm 10\%$. Our model generators are simple, fast and fairly accurate, and have been integrated with an existing high-level synthesis system [RaGa91].

We also presented a user-interface to the HLLM system. The interface includes a graphical environment that assists a user in specifying the parameters for HLLM, activating HLLM and finally displaying the mapping result. The GENUS environment along with the HLLM system supports the demands of current day design methodologies using high-level design as well as design reuse at the RT level.

Chapter 7

Conclusion

7.1 Summary of Dissertation

In this dissertation, we have described high-level library mapping (HLLM), a library mapping technique at the RT level. The technique is specially well-suited to supplement the needs of the current day design methodologies using *high-level design* and *design reuse*. HLLM supports high-level design by mapping its output RT level design onto RT components from a technology specific library. Design reuse is achieved by retargetting a RT level design across various RT libraries using HLLM.

We introduced High-level library mapping, a library mapping technique that is based on RT level functionality. High-level library mapping is well-suited for mapping regularly-structured datapath components as well as memory modules. We also discussed the design scenario supported by the HLLM approach and listed the three tasks for the HLLM problem namely, RT library definition, HLLM formulation for datapath components and HLLM formulation for memory modules.

We described a generic RT library that forms the basis of HLLM. We demonstrated the comprehensiveness of the library by comparing it with various technology libraries and presented experimental results to demonstrate its efficacy. HLLM uses the functional definitions of RT components provided by the the generic library

as a reference point to compare and contrast the source and the target component and finally to map the source component onto the target component.

We applied HLLM technique on a commonly used datapath component class, namely ALUs. We formulated the problem of HLLM for ALUs in terms of a source ALU and a target ALU and presented an efficient algorithm to map the source ALU onto the target ALU. We also presented the experimental results to demonstrate the comprehensiveness as well as the efficacy of designs produced by the HLLM for ALU approach.

We demonstrated the HLLM technique on memory modules as well. We formulated HLLM for memory in terms of a source memory module and a set of target memory modules from a library. We decomposed the memory mapping problem into three subproblems, namely bit-width mapping, word mapping and port mapping and then combined these formulations into a linear time algorithm to solve the complete memory mapping problem. We also presented experimental results to demonstrate the efficacy of high-level library mapping approach for memories.

Finally, we presented the GENUS environment that implements the generic library discussed in Chapter 3. The GENUS environment consists of a set of generic component generators, various models generators, technology projectors and technology mappers. The various models generated by GENUS support validation and synthesis of RT designs. The technology projectors rapidly generate fairly accurate estimates, providing technology specific feedback to higher levels. The High-level library mapper provides a link to lower levels by mapping GENUS components to library specific components. We also presented a graphical user-interface for the HLLM system. The user interface assists a user in specifying the various design parameters, activating a mapping algorithm and finally displaying the mapped design.

7.2 Summary of Contributions

The research described in this dissertation makes three fundamental contributions. First, it defines high-level library mapping, a novel library mapping technique at RT level. Second, it presents efficient formulations of HLLM for a representative datapath component. Finally, it presents efficient formulations of HLLM for memory modules.

High-level library mapping is the first library mapping technique which maps a RT component onto other RT component(s) of the same complexity. This is in contrast to other library mapping approaches which first decompose the source component into lower level cells and then perform technology mapping. Another distinguishing feature of HLLM is that it uses RT level functional behavior of the source and the target component. We defined a library of generic RT components to facilitate this mapping process. Experimental results demonstrate that the HLLM approach for datapath components outperforms the traditional logic level library mapping approach in all the three design metrics : area, delay and runtime.

We used ALUs as a representative datapath component for illustrating HLLM and presented efficient formulations of HLLM for ALUs. Based on the problem characteristic of ALU mapping, we presented a polynomial time algorithm to perform HLLM for ALUs. The experimental results demonstrate that HLLM for ALU is a powerful technique, orders of magnitude better than the existing techniques for reusing ALUs.

We also presented efficient formulations of HLLM for memories. Based on domain specific knowledge of memory modules, we presented a linear algorithm to perform word mapping for memories. Our experimental results demonstrate that HLLM for memories can generate a variety of designs including several counter-intuitive ones quickly.

In summary, the HLLM approach described in this thesis elevates library mapping from the logic to the RT level and facilitates design reuse of datapath components and memories.

7.3 Future Directions

The HLLM approach in its current form could be extended in many directions. Starting from the improvements in the domain of HLLM for ALUs and memories, HLLM could be extended all the way up to include system level parts. We discuss some of these possible extensions :

1. The HLLM formulations for ALUs and memories presented in this dissertation can be refined. Specifically, the assumptions listed in Chapters 4 and 5 can be relaxed. HLLM for ALU could be upgraded to include functions that are not restricted to 2's complement representation as well as making the target ALU more general. HLLM of memory assumes that the access protocol of the source and target memory modules are compatible. The system could be upgraded to consider source and target modules with dissimilar access protocols.

2. HLLM could be applied to other regularly structured datapath components, both in the domain of combinational and sequential components. Counters, shift-registers and barrel shifters are particularly good candidates for HLLM.

3. With respect to a complete RT design, a system that uses a combination of logic level library mapping (LLLM) and HLLM can be developed. HLLM performs efficient mapping for memory modules and regularly-structured datapath components; LLLM can handle the rest of the components in the RT design, including controller circuit and the remaining datapath components.

4. In the long term, efforts could also be directed towards applying HLLM on system level parts such as processor cores. Techniques for representing

system-level parts and effectively mapping between them need to be developed.

Bibliography

[Am2901] "Am2901c: Four-bit Bipolar Microprocessor Slice," *Advanced Micro Devices, Sunnyvale, California,* 1993.

[Am2910] "Am2910A: Microprogram Controller," *Advanced Micro Devices, Sunnyvale, California,* 1993.

[AnDu94] R. Ang and N. Dutt, "An Algorithm for Allocation of Functional Units from Realistic RT Component Libraries," *The 7th International Symposium on High-Level Synthesis,* pp164-169, May 1994.

[BaCM94] F. Balasa, F. Catthoor and H. De Man, "Data-driven Memory Allocation for Multi-dimensional Signal Processing Systems," *International Conference on Computer Aided Design*, pp31-34, 1994.

[BaGa95] S. Bakshi and D. Gajski, "A Memory Selection Algorithm for High-Performance Pipelines," *Technical Report 95-03, University of California, Irvine,* January 1995.

[BiGS89] W. Birmingham, A. Gupta and D. Siewiorek, "The MICON System for Computer Design," *The 26th ACM/IEEE Design Automation Conference,* pp135-139, 1989.

[BrMR93] H. Brand, R. Mueller and W. Rosenstiel, "Design of High Throughput Data Path Components," *The 4th ACM/SIGDA Physical Design Workshop,* April 1993.

[BRSW87] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A. Wang, "MIS : A multiple-level logic optimization system," *IEEE Transaction on Computer-aided Design,* pp1062-1081, November 1987.

[Casc92] "Cascade Design Automation Databook," *Cascade Design Automation, Bellevue, WA,* 1992.

[CaTr89] R. Camposano and L. Trevillyan, "The Integration of Logic Synthesis and High-Level Synthesis," *International Symposium on Circuits and Systems,* 1989.

[CoLR90] T. Cormen, C. Leiserson and R. Rivest, "Introduction to Algorithms," *The MIT Press, Cambridge, Massachusetts* 1990.

[Degn92] "DesignWare Databook, Version 3.0," *Synopsys® Inc., Mountain View,* December 1992.

[DJBT94] J. Darringer, W. Joyner, L. Berman and L. Trevillyan, "LSS: A System for Production Logic Synthesis," *IBM Journal of Research and Development,* vol. 28, No. 5, pp. 537-545, September 1984.

[DuRa92] N. Dutt and C. Ramachandran, "Benchmarks for the 1992 High Level Synthesis Workshop," *Technical Report 92-107, University of California at Irvine,* 1992.

[Dutt88] N. Dutt, "GENUS: A Generic Component Library for High-Level Synthesis" *Technical Report 88-22, University of California at Irvine,* 1988.

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

[GDWL92] D. Gajski, N. Dutt, A. Wu and S. Lin, "High-Level Synthesis: Introduction to Chip and System Design," *Kluwer Academic Publishers,* 1992.

[JhDG93] P. Jha, N. Dutt and D. Gajski, "An Evaluative Study of RT Component Libraries," *Technical report 93-11, University of California at Irvine*, March 31, 1993.

[JhDu92] P. Jha and N. Dutt, "A Fast Area-Delay Estimation Technique for RTL Component Generators," *Technical report 92-33, University of California at Irvine*, April 10, 1992.

[JhD94a] P. Jha and N. Dutt, "The GENUS User Manual and C Programming Library, Revision 4.0," *Technical report 93-32, University of California at Irvine*, August 23, 1994.

[JhD94b] P. Jha and N. Dutt, "High-Level Library Mapping for Arithmetic Components," *Technical report 94-15, University of California at Irvine*, April 10, 1994.

[JhD95a] P. Jha and N. Dutt, "User's Guide to the High-Level Library Mapping System," *University of California at Irvine*, 1995.

[JhD95b] P. Jha and N. Dutt, "High-Level Library Mapping for Memories," *Technical report 95-37, University of California at Irvine*, September 1995.

[KaRo94] D. Karchmer and J. Rose, "Definition and Solution of the Memory Packing Problem for Field-Programmable Systems," *International Conference on Computer-Aided Design*, pp20-26, 1994.

[Keut87] K. Keutzer, "DAGON: Technology Binding and Local Optimization by DAG Matching" *The 24th ACM/IEEE Design Automation Conference,* 1987.

[KiLi93] T. Kim and C. Liu, "Utilization of Multiport Memories in Data Path Synthesis," *The 30th Design Automation Conference,* 1993.

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

[KuRa91] F. Kurdahi and C. Ramachandran, "LAST: A Layout Area and Shape function esTimator for High Level Applications," *The European Design Automation Conference*, February 1991.

[Li93] J. Li, "VHDL Modeling for Silicon Compilation," *M.S. Thesis, University of California at Irvine,* 1993.

[LoWM93] J. Loos, C. Wang and M. Mahmood, "Physical Modelling of Datapath Libraries for Design Automation Applications," *The 4th ACM/SIGDA Physical Design Workshop,* April 1993.

[LPM93] "EDIF 2 0, Library of Parametrized Modules," *Electronic Industries Association, Washington D.C.,* 1993.

[LsiLogic] "Block Synthesis User's Guide," *LSI Logic Corporation, Milipitas*

[MaMi93] F. Mailhot and G. Micheli, "Technology Mapping with Boolean Matching," *IEEE Transactions on CAD,* pp 559-620, May 1993.

[Marw93] Peter Marwedel, "Tree-based Mapping of Algorithms to Predefined Structures," *The International Conference on Computer-Aided Design,* pp. 586-593, 1993.

[Mich92] P. Michel, U. Lauther and P. Duzy (Editors) "The Synthesis Approach to Digital System Design," *Kluwer Academic Publishers,* 1992.

[RaGa91] L. Ramachandran and D. Gajski, "An Algorithm for Component Selection in Performance Optimized Scheduling," *International Conference on Computer-aided Design*, November 1991.

[RaKu92] C. Ramachandran and F. Kurdahi, "TELE: A Timing Evaluator using Layout Estimation for High Level Applications," *The European Design Automation Conference*, March 1992.

[RuGB93] E. Rundensteiner, D. Gajski and L. Bic, "Component Synthesis From Functional Descriptions," *IEEE Transaction on Computer Aided Design,* pp1287-1299, September 1993.

[STMF90] H. Sato and N. Takahashi, "Boolean Technology Mapping for Both ECL and CMOS circuits Based on Permissible Functions and Binary Decision Diagrams," *International Conference on Computer-Aided Design,* pp. 286-289, 1990.

[ScTh95] H. Schmit and D. Thomas, "Array Mapping in Behavioral Synthesis," *Proc. of the Eighth International Symposium on System Synthesis,* 1995.

[Syno92] "Design Analyzer Reference Manual, Version 3.0," *Synopsys® Inc., Mountain View,* December 1992.

[Ti94] "Semiconductor Group Distribution, USA and Canada Suggested Resale Pricing Guide," *Texas Instruments,* 1994.

[Tosh90] "Toshiba ASIC Gate Array Library," *Toshiba Corporation, Tokyo, Japan,* 1990.

[Tyag90] A. Tyagi, "An Algebraic Model for Design Space with Applications to Function Module Generation," *The European Conference on Design Automation,* pp114-118, March 1990.

[VaGa88] N. Vander Zanden and D. Gajski, "MILO: A Microarchitecture and Logic Optimizer," *The 25th Design Automation Conference,* June 1988.

[VTI91] "VDP300 CMOS Datapath Library," *VLSI Technology, Inc., San Hose, California,* November 1991.

[Wolf89] W. Wolf, "How to Build a Hardware Description and Measurement System on an Object-Oriented Programming Language," *IEEE Transaction on Computer-aided Design*, pp. 288-301, March 1989.

[XBLO92] S. Kelem and J. Seidel, "Shortening the Design Cycle for Programmable Logic Devices," *IEEE Design and Test of Computers,* pp40-50, 1992.

[Xili93] "Development System Libraries Guide," *Xilinx®, San Jose, CA* 1993.

[Youn94] "The X Window System Programming and Applications with Xt," *Prentice Hall, Englewood Cliffs, New Jersey,* 1994.