UC Irvine
ICS Technical Reports

Title
High-level library mapping for memories

Permalink
https://escholarship.org/uc/item/94g570n9

Authors
Jha, Pradip K.
Dutt, Nikil D.

Publication Date
1995-09-06

Peer reviewed

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

SLBAR
Z
699
C3
no. 95-37

High-Level Library Mapping for Memories

Pradip K. Jha and Nikil D. Dutt

Technical Report #95-37
Date : September 6, 1995

Dept. of Information and Computer Science
University of California at Irvine
Irvine, CA 92717-3425
Phone: (714) 824-8059
Fax: (714) 824-4056
Email: pradip@ics.uci.edu

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

Abstract

We present High-Level Library Mapping, a technique that synthesizes a source memory module from a library of target memory modules. In this report, we define the problem of high-level library mapping for memories, identify and solve the three subproblems associated with this task and finally combine these solutions into a suite of two memory mapping algorithms. Experimental results on a number of memory-intensive designs demonstrate that our memory mapping approach generates a wide variety of cost-effective designs, often counter-intuitive ones, based on a user-given cost function, the target library and the mapping algorithm used.

Contents

1 Introduction **1**

2 Previous work **2**

3 Problem definition **7**

3.1 Memory specification 7

3.1.1 Cost function 10

3.2 Memory mapping 10

3.3 Port mapping 12

3.3.1 Port constraints 12

3.3.2 Port assignment 13

3.3.3 An example 15

3.4 Bit-width mapping 16

3.4.1 An ILP formulation for bit-width mapping 16

3.4.2 An algorithm for bit-width mapping 17

3.4.3 An example 19

3.5 Word mapping 21

3.5.1 An ILP formulation for word mapping 22

3.5.2 An algorithm for word mapping 25

3.5.3 An example 27

3.6 The cost function 29

4 Overall approach **32**

4.1 Assumptions 32

4.2 Linear algorithm 34

4.2.1 An example 36

4.3 Exhaustive algorithm 38

4.3.1 An example 42

5 Experiments and results **42**

5.1 Summary of experiments 49

6 Conclusion **52**

7 Acknowledgements **54**

List of Figures

1 Classification of memory mapping works 5

2 Sample high-level library mapping for memories: (a) source and target modules (b) mapping result 8

3 Three degrees of freedom in memory mapping problem 11

4 Port map example (a) source and target memory components (b) port map between source s and target t. 15

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 20

6 Memory realization with unrestricted word-count 24

7 Word mapping example (a) source and target memory modules (b) intermediate results from word mapping algorithm (c) final design 28

8 Cost of a memory design 30

9 Two types of memory composition (a) regular composition (b) irregular composition 33

10 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 mapping followed by word mapping 37

11 Complete design for the memory mapping example 38

12 Bit-width relaxed regular composition 39

13 A memory mapping example with *exhaustive* algorithm (a) source and target modules (b) optimal bit-width configuration for each word-count (c) memory configuration for the resultant design 41

14 Memory mapping result I 44

15 Memory mapping result II 45

16 Memory mapping result III 46

17 Memory mapping result IV 47

18 Memory mapping result V 48

19 Memory mapping result VI 49

20 Memory mapping result VII 50

1 Introduction

Digital systems at the RT level are often decomposed into datapaths, controllers and memories. 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 the domain of high-level synthesis (HLS)[GDWL92], the output typically consists of an RT-level netlist of generic components, including logical memory modules. These memory modules can implement both the array variables in the input description, as well as a set of clustered scalar variables to make a cost-effective design. These memory modules are logical in the sense that there may not be a memory in the library that satisfies their size as well as port requirements. We need a memory synthesis scheme to realize these logical memory modules with real memory modules from a library.

A memory mapping scheme is also required to support design resue. Design reuse, in this context, refers to the process of reusing (or migrating) a design across different technology libraries. The existing design may use a memory module with a specific size and port configurations from an old library; such memories may not be available in the new library and therefore have to be synthesized with modules from the new library.

In this work, we present a memory synthesis scheme that implements a source memory module with one or more memory modules from a target library. High-level library mapping

(HLLM) for memory, our approach to memory mapping, 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 report is organized as follows. Section 2 describes related work. Section 3 defines high-level library mapping for memories and decomposes it into three subproblems. Section 4 describes two memory mapping algorithms that implement a source memory module with a set of target memory modules. Section 5 describes sample memory mapping results to demonstrate the efficacy of our approach. The report concludes with a summary and discussion of future work.

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. Our work fits in the second group.

There has been a fair amount of work in mapping the storage requirements of a design's behavior into logical memories. Most of the earlier works have focussed on grouping the scalar

variables to be implemented with multi-port memories. [KiLi93] presents a brief summary of these works. Some recent efforts emphasize on mapping arrayed variables directly onto memory modules. [RaGC93][MaLa94] present algorithms to bind the storage requirements onto memories for general domain problems. The MeSA algorithm [RaGC93] presents a scheme to map the arrayed variables in the input behavior into logical memory modules. The output of the MeSA algorithm is a set of logical memory modules (in terms of size and port configurations) that would be required to implement the arrayed variables that have been clustered based on cost and performance considerations. The TODOS system [MaLa94], on the other hand, tries to incorporate real memories from the library directly into the scheduling and binding phase of synthesis.

Another group of research papers focus on synthesizing the storage requirements for high-throughput applications specific to the domain of speech, image, video and DSP. These applications are often specified using a non-procedural language using data streams that facilitates a designer to make full use of parallelism in the design. The PHIDEO system [LMVW93] presents a memory allocation scheme for hierarchical data streams targeted towards fixed-rate video applications. They allocate background memories for storing the signals that are stored between the clock cycle of production and consumption. [BaCM93][BaCM94] present another memory allocation scheme for real-time multi-dimensional signal processing applications. They partition the multi-dimensional signals into non-overlapping basic sets using a polyhedral data-flow analysis and then pack these basic sets into logical memories using an iterative improvement scheme based on a Branch and Bound algorithm. Our work complements these efforts by providing an efficient scheme to map the logical memory modules

generated by these systems into physical memory modules from a target library.

There are relatively fewer works in the domain of logical memory realization with real physical memories. [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]. 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 lead 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

[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 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.

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

[2]In Figure 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.

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 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.

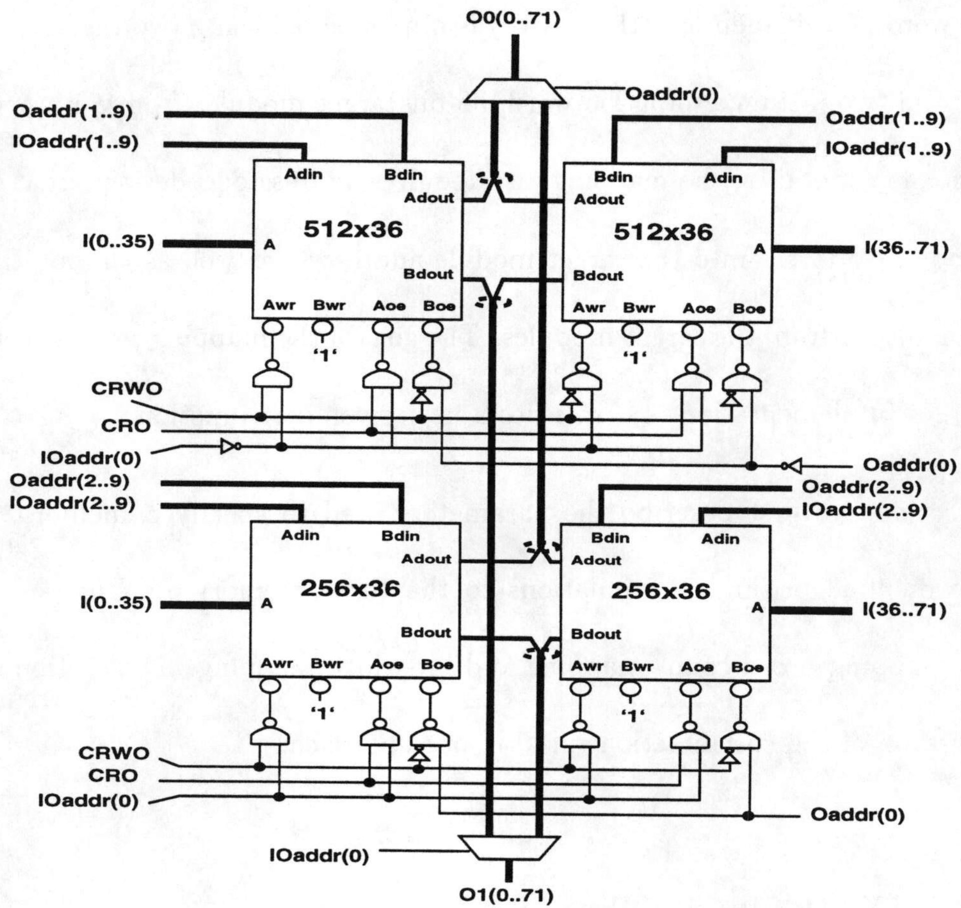
In this section, we first describe the parameters used to specify a memory. Next we define and present algorithmic formulations to the three memory-mapping subproblems, namely port mapping, word-count mapping and bit-width mapping. The section ends with a brief description of the cost functions used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

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

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 2: Sample high-level library mapping for memories: (a) source and target modules (b) mapping result

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.

3.1.1 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 $D(m)$.

3.2 Memory mapping

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.

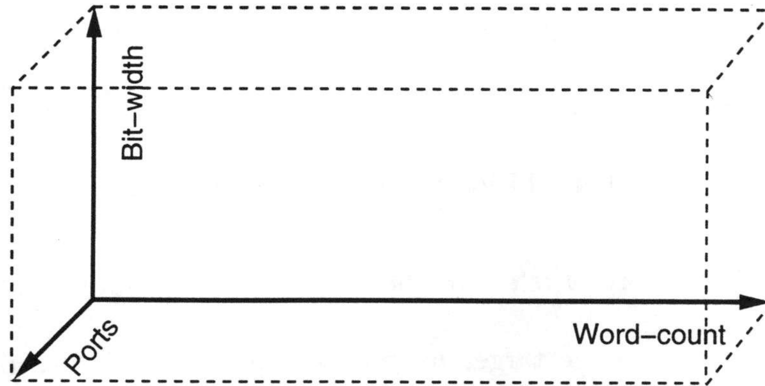


Figure 3: Three degrees of freedom in memory mapping problem

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 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

[3]See [DuJh93] for a survey of RT-level libraries

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 4, we combine these formulations to achieve a global solution to the complete memory mapping problem.

For the rest of the report, 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.

3.3 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.

3.3.1 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:

$$R(s) + RW(s) \leq R(t) + RW(t) \quad (1)$$

$$W(s) + RW(s) \leq W(t) + RW(t) \quad (2)$$

$$R(s) + W(s) + RW(s) \geq R(t) + W(t) + RW(t) \quad (3)$$

The *read constraint* in Equation 1 ensures that t has enough ports to read out the data; the *write constraint* in Equation 2 ensures that t has enough ports to write in the data. The *read-write constraint* shown in Equation (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.

3.3.2 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 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

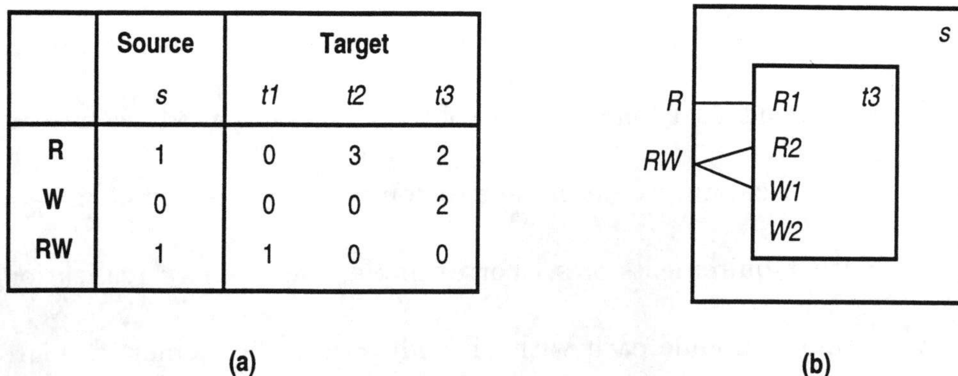
and a source read-write port to a target read-write port. Each of the remaining read and write ports is realized by a single read-write port. Similarly, each of the remaining source read-write ports is realized as a read port together with a write port of t. Note that the port constraints ensure that t has a sufficient number of ports to perform this assignment.

Algorithm 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

Figure 4: Port map example (a) source and target memory components (b) port map between source s and target t.

3.3.3 An example

Figure 4 illustrates an example for port mapping. Figure 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 1) as well as the read-write constraint (Equation 3). Also, $t2$ fails to satisfy the write constraint. $t3$, on the other hand, fulfills all the port constraints. Figure 4(b) shows the result of port assignment between s and $t3$ generated by Algorithm 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 $t3$.

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 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.

3.4.1 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:

$$\sum_{i=1}^{m} x_i A(t_i) \tag{4}$$

and satisfies the following linear constraint:

$$\sum_{i=1}^{m} x_i B(t_i) \geq B(s) \quad (5)$$

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.

3.4.2 An algorithm for bit-width mapping

Algorithm 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 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]* = ϕ;
2 **end for**;
3 $Ts = \phi$;
4 **for** each $t_i \in T$ **do**
4.1 *best_sol*[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*[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*[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 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 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};

3.4.3 An example

Figure 5 walks through the bit-width mapping algorithm (Algorithm 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(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(b) lists memory module instances that are used in the enumeration steps (Steps 6–9) of the Algorithm 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(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

	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}

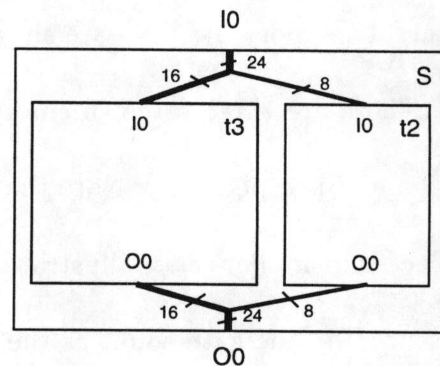
(b)

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: 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

has 6 elements each with a 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(d). The optimal mapping for the source component is given by the mapping *(t3, t2)* with a gate-count of 10818. Figure 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.

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 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.

3.5.1 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:

$$\sum_{i=1}^{m} x_i A(t_i) \tag{6}$$

and satisfies the following linear constraint:

$$\sum_{i=1}^{m} x_i N(t_i) \geq N(s) \tag{7}$$

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 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 2, the design requires two 2-input 72-bit multiplexers (one for each output). Furthermore, Equation 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.

The power-of-two assumption not only makes it possible to devise an efficient algorithm for word mapping, but also obviates the need for adders in the address decoding. If we use memory modules with unrestricted word-counts, we will require complex adder logic to map the address space of the source to the address space of the target memory modules. Figure 6 shows an example. The source memory module (word-count = 64) is realized with three target memory modules *t1, t2* and *t3* of size 28, 20 and 16 words respectively. The base addresses for *t1, t2* and *t3* are 0, 28 and 48 respectively. Figure 6 shows an implementation using two subtractors. Figure 2, on the other hand, illustrates that the address mapping logic

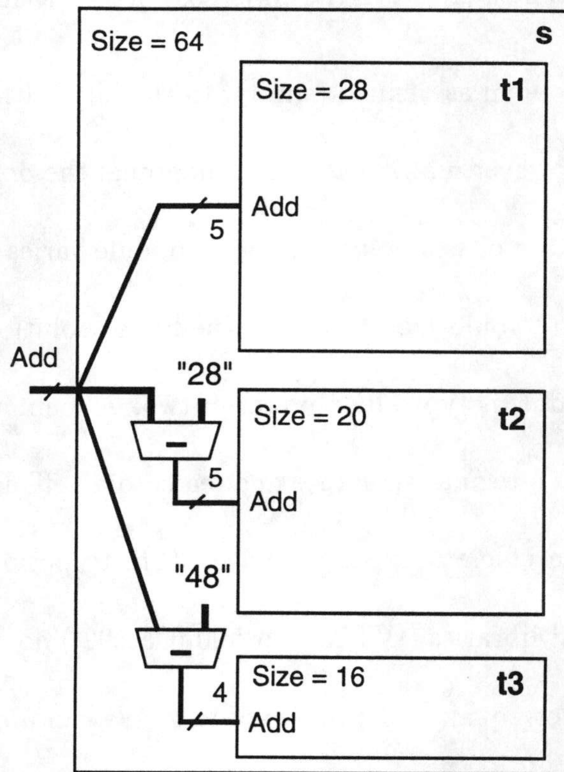


Figure 6: Memory realization with unrestricted word-count

for target memory modules with word-count equal to powers-of-two could be implemented with a couple of gates. Note that [ScTh95] presents a clever scheme for address mapping that obviates the use of adder logic. However, the scheme requires that the word-counts of the source and the target modules satisfy specific conditions; otherwise the source and the target module word-count need to be approximated by next higher word-count satisfying the conditions. This can result in wasted address space of as large as 33%[ScTh95].

3.5.2 An algorithm for word mapping

Algorithm 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 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.

3.5.3 An example

Figure 7 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 7(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 7(b) shows intermediate results *after* each iteration of the **while** loop in the Algorithm 7. 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 7(c) shows the implementation corresponding to the mapping *(t1, t2)*. Along with the two modules $t1$ and

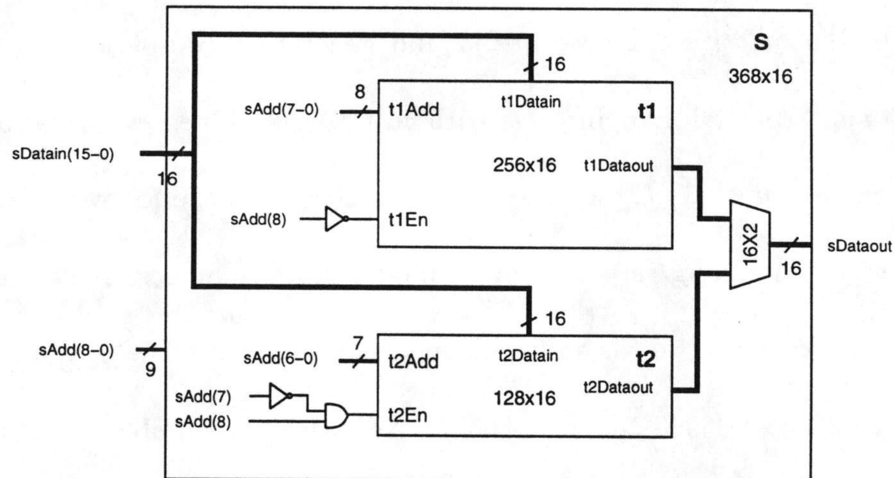
[4] A memory module is irredundant if it can't be synthesized using other memory modules with smaller cost.

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 7: Word mapping example (a) source and target memory modules (b) intermediate results from word mapping algorithm (c) final design

$t2$, the implementation uses a 2-input 16-bit multiplexer and some small address decoding logic (two inverters + an AND gate).

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.

3.6 The cost function

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 8, 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.[5]

[5] Our word mapping algorithm implements the address decoder without using the "adder" logic. However, for a general word mapping algorithm (e.g., *exhaustive* algorithm discussed in Section 4.3), the address decode logic can be of the order of the "adder" logic.

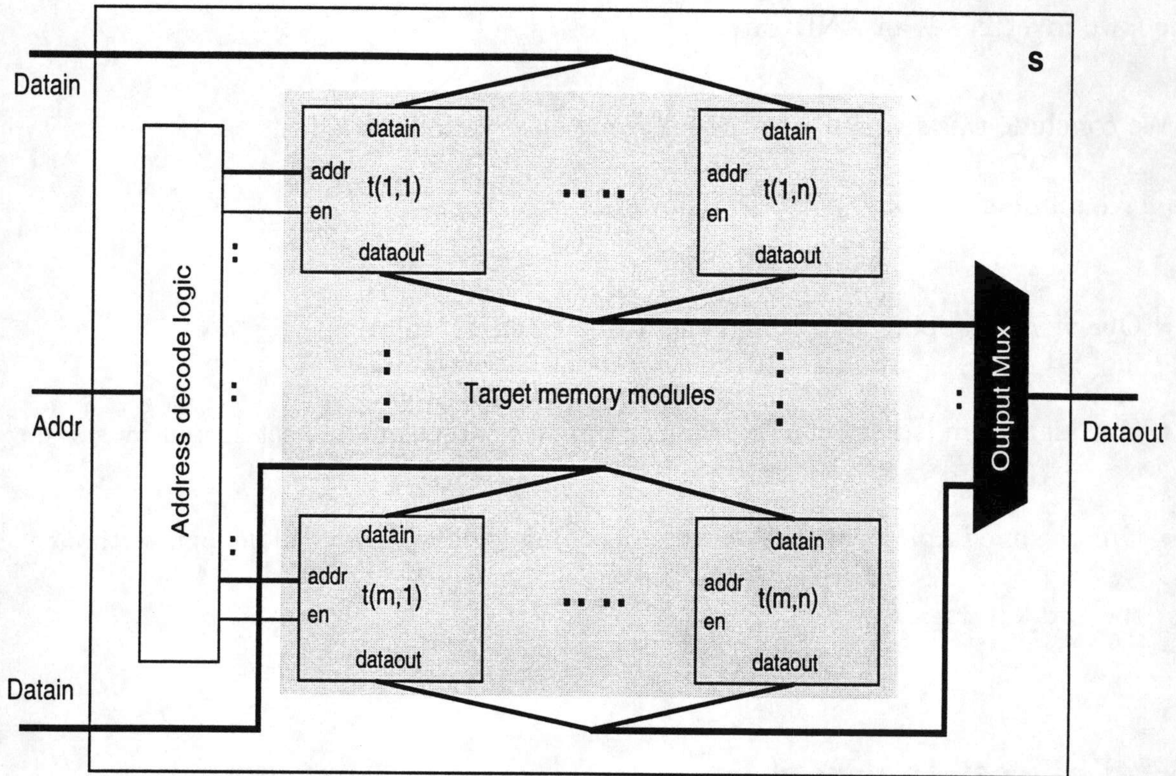


Figure 8: Cost of a memory design

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:

$$C(s) \quad = \quad C(address) + C(target) + C(mux) \tag{8}$$

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.

4 Overall 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 the complete memory mapping algorithm. In this section, we first describe the basic assumptions underlying our approach. Next we present a suite of two algorithms, namely *Linear* and *Exhaustive* to solve the complete memory mapping problem. These two algorithms differ in the way they solve the word-mapping problem: the *Linear* algorithm uses the run-time efficient linear algorithm presented in Section 3.5; the *Exhaustive* algorithm, on the other hand, finds the best mappings by trying out all combinations of word mapping exhaustively. We demonstrate these algorithms with an example.

4.1 Assumptions

In this work, we 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 9).
3. The bit-width of the source memory module is small.
4. The port name mapping for the source and the target modules are performed by the

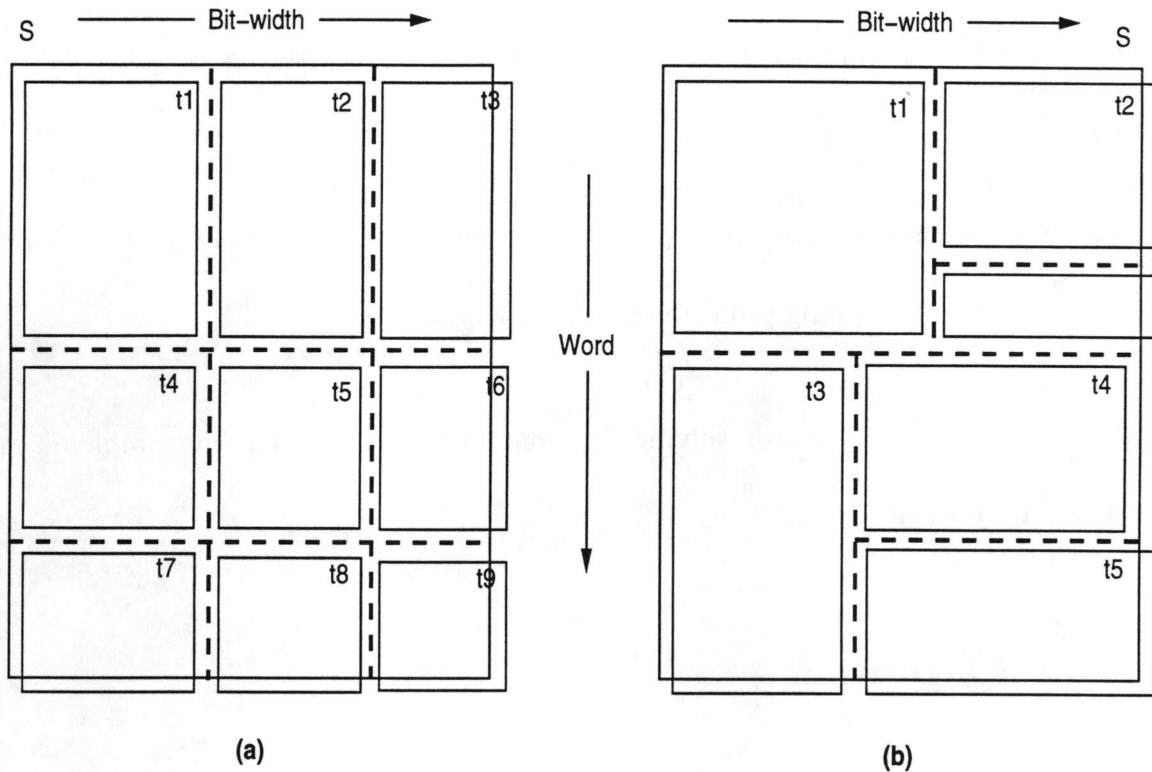


Figure 9: Two types of memory composition (a) regular composition (b) irregular composition

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 have a comprehensive solution to 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.

4.2 Linear algorithm

Algorithm 4.1 describes the steps in the *Linear* 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 4.1 : Linear Memory mapping

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 $\mathrm{B}(t_i) \in common_bit\}$;
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 $\mathrm{S}(t_i) \in common_word\}$;
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 1, 2 and 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 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.

4.2.1 An example

Figure 10 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 10(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 10(b) and 10(c) trace the two paths through the memory mapping algorithm. Figure 10(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 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 10(b). The cost of this design is 30445 gates.

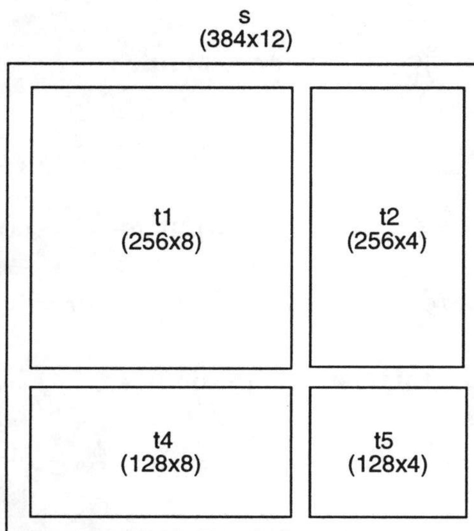
Figure 10(c) walks through the other path that performs the two tasks in the reverse order: bit-width mapping followed by word mapping. The memory design generated by the bit-width mapping algorithm uses a single module *t3*, since it has an exact match to the bit-width (12) of the source module. There 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 10(c). The cost

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

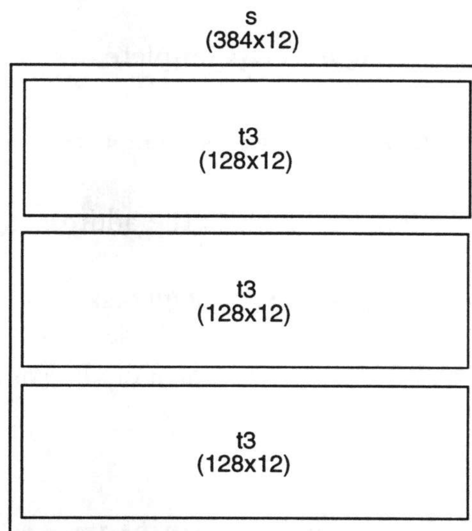
(a)

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 10: 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 mapping followed by word mapping

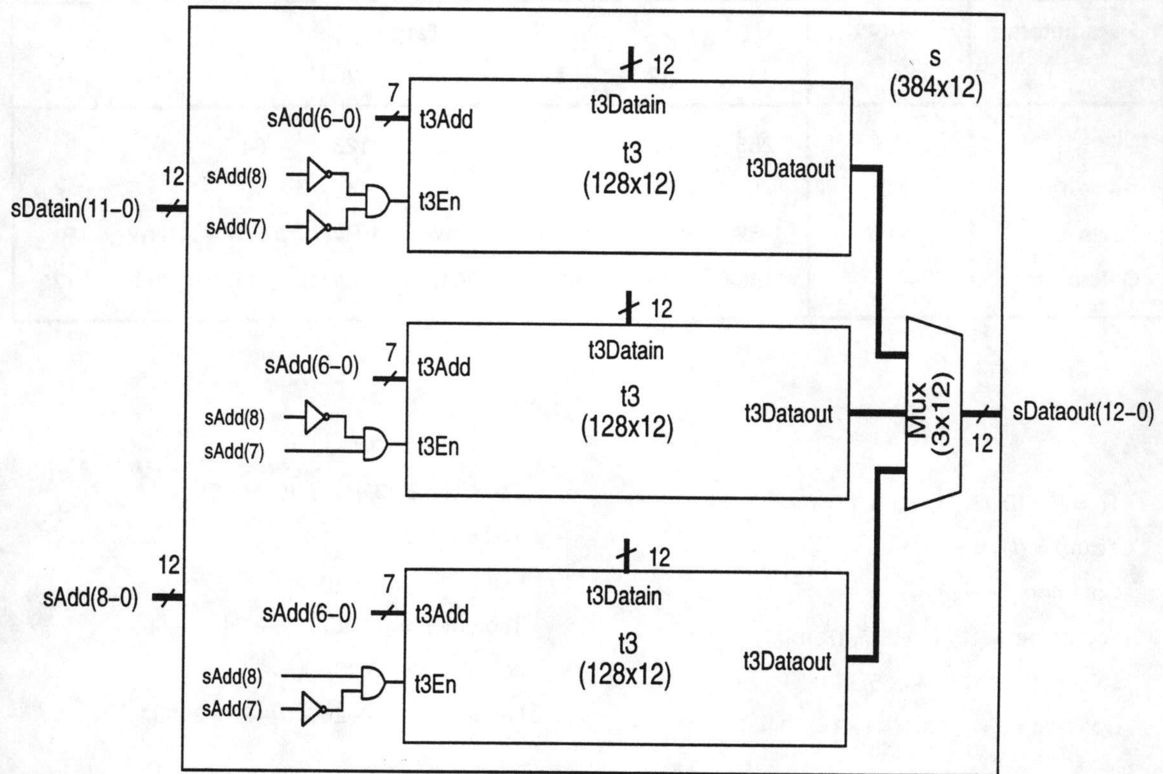


Figure 11: Complete design for the memory mapping example

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 11. 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.

4.3 Exhaustive algorithm

Next we present an exhaustive algorithm that performs the task of word mapping by exhaustively searching all possible combinations. In this algorithm, we relax the first two assumptions mentioned in Section 4.1. The target memory modules can have an unrestricted

word-count. Also, we relax the regularity constraint in the bit-width dimension, i.e., all the memory modules in each column (of the resultant memory composition) need not have the same bit-width. A example of bit-width relaxed regular composition is shown in Figure 12.

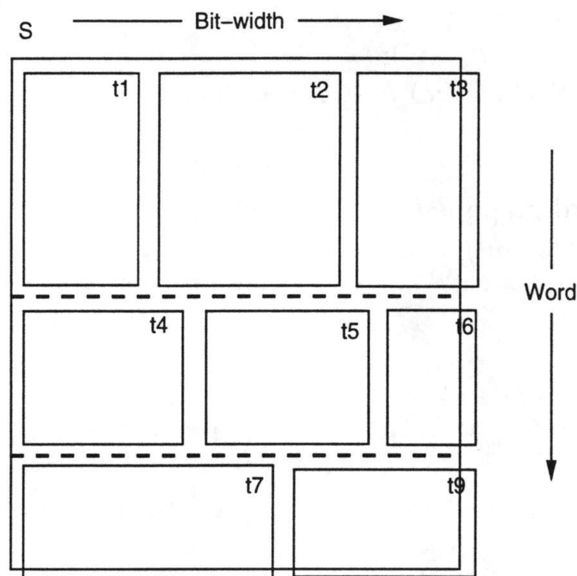


Figure 12: Bit-width relaxed regular composition

Algorithm 4.2 lists the steps in the *exhaustive* algorithm. As before, the inputs to the algorithm are the source memory module (s), the target memory module set (T) and the cost function (C). The target module sets in this algorithm, namely T_p, T_{bit} and T_{word} store the list of modules after port constraint satisfaction, after running bit-width mapping for each word-count and before running the final exhaustive word-mapping algorithm. The temporary variable T_{wc} stores all modules with a given word-count. The variable sol records the final solution.

Algorithm 4.2 : Exhaustive Memory mapping

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 $T_{word} = \phi$;

3 **For** each word-count wc in T_p do

3.1 $T_{wc} = \{t_i | W(t_i) = wc\}$;

3.2 T_{bit} = Bit mapping(s, T_{wc}, C);

3.3 Create a t with $(W(t) = wc)$ & $(C(t) = C(T_{bit}))$;

3.4 $T_{word} = T_{word} + t$;

4 **end for**;

5 sol = Exhaustive word mapping(s, T_{word}, C);

6 sol = Port assignment(s, sol);

7 **return** sol;

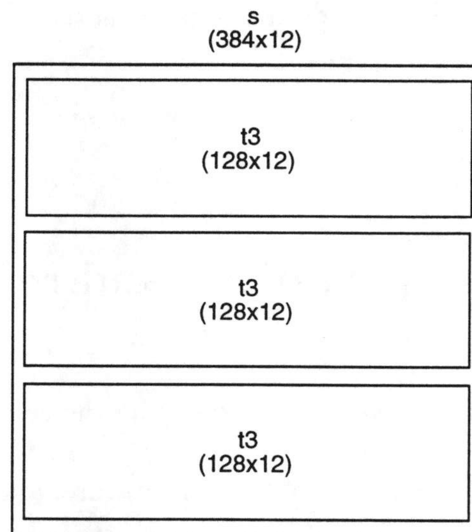
The algorithm starts by selecting the subset of modules from T that satisfies the port constraints (Step 1). In the **For** loop of Steps 3-4, the algorithm generates the best bit-width mapping for each word-count wc and creates a new memory module with the word-count equal to wc and cost equal to the cost of the resultant bit-width mapping. Finally, it performs an exhaustive word mapping on the set of modules created from the last step. The exhaustive word mapping algorithm is similar to the exhaustive bit-width mapping algorithm presented in Section 3.4.2. The result of word mapping provides the the final solution. The mapping is completed by performing the port assignment on the result of the word mapping. Algorithm 4.2 is exponential with respect to the number of modules used for bit-width expansion as well as with respect to the number of modules used for word expansion.

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)

Size	Bit-mapping	Cost
256	(t1, t2)	19126
128	(t3)	8680
64	(t6)	5166

(b)



(c)

Figure 13: A memory mapping example with *exhaustive* algorithm (a) source and target modules (b) optimal bit-width configuration for each word-count (c) memory configuration for the resultant design

4.3.1 An example

Let us reconsider the memory mapping example discussed in section 4.2.1. Figure 13(a) shows the source and target memory modules. We now apply the *exhaustive* algorithm on this example. Figure 13(b) shows the result of bit-width mapping for each word-count. Note that in this example, there are three distinct word-counts: 256, 128 and 64. The source bit-width (12) is synthesized with (t1, t2) for the word-count 256, with (t3) for the word-count 128 and with (t6) for the word-count 64. The table in Figure 13(b) shows the cost of each of these bit-width mappings. Next the algorithm performs an exhaustive word mapping on these bit-width mapping results. The resultant memory configuration is shown in Figure 13(c). The final design is same as the one generated by the *linear* algorithm and is shown in Figure 11.

5 Experiments and results

In this section, we first describe the results of applying our memory mapping algorithm on several examples from the literature, and then summarize 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 14 through 19; Figures 14 and 15 show mapping result for an additional example. The memory module in the first example is from the Differential Heat Release computation algorithm modelling the heat release within a combustion engine[RaGC93]. The second memory module is from a Neural Network Chip design reported

in [KaRo94]. The third module is from a Progressive Conversion algorithm [Lipp91]. The memory modules in the fourth and the seventh examples (in Figures 14 and 15) are taken from the Medical Image Reconstruction and the Singular Value Decomposition algorithms respectively reported in [BaCM94]. The source modules in the fifth and the sixth examples are extracted from industrial designs for an MPEG implementation reported in [ThGa95]. 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 14 through 20 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. 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 6 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 sixth column presents the run-time (in seconds) for our algorithm on a SUN Sparc Station 5. We have an additional column (column 7) in the tables corresponding to the designs produced by the *exhaustive* algorithm shown in Figures 15, 17 and 19. This column reports the percentage improvement in the metrics of the designs produced by the *exhaustive* algorithm over the metrics of the corresponding designs produced by the *linear* algorithm. In each table we also report the name of the target library, the cost function and the algorithm used to generate the designs.

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

Example			Mapping result		
#	*Name*	*Size*	*Design*	*Area* (2-nand)	*Run-time* (sec)
1	Differential Heat Release (Ramachan,EDAC 94)	469x16	256x8, 256x8 256x8, 256x8	46331	1.1
2	Neural Network Chip (Rose, ICCAD 94)	160x8	128x8 32x8	9035	1.2
3	Interlace Scan (Philips, CICC 91)	360x16	256x8, 256x8 128x8, 128x8	36491	1.4
4	Medical Image Reconstruction (Balasa, ICCAD 94)	384x12	256x6, 256x6 128x6, 128x6	30319	2.0
5	MPEG I (Thordarson, TR 95-8)	54x18	64x6, 64x6, 64x6	11017	2.2
6	MPEG II (Thordarson, TR 95-8)	128x17	128x5, 128x6, 128x6	16383	1.2
7	Singular Value Decomposition I (Balasa, ICCAD 94)	1365x16	256x8, 256x8 : 256x8, 256x8 64x8, 64x8 16x8, 16x8 8x8, 8x8	128436	2.5

Figure 14: Memory mapping result I

Figures 14 and 15 show the mapping results for area-efficient designs generated by the *linear* and *exhaustive* algorithms respectively. These designs have been synthesized using the memory modules 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.

From the mapping results in Figures 14 and 15, we observe that the designs produced by the *exhaustive* algorithm are usually different and smaller (except for Example 6) in area as compared to the designs produced by the *linear* algorithm. The area of designs produced

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

Example			Mapping result			
#	*Name*	*Size*	*Design*	*Area* (2-nand)	*Run-time* (sec)	*Improvement over Linear (Fig 14)*
1	Differential Heat Release (Ramachan,EDAC 94)	469x16	224x8, 224x8 256x8, 256x8	43699	2.8	**5.7%**
2	Neural Network Chip (Rose, ICCAD 94)	160x8	160x8	7631	1.4	**15.5%**
3	Interlace Scan (Philips, CICC 91)	360x16	112x8, 112x8 256x8, 256x8	34979	1.9	**4.1%**
4	Medical Image Reconstruction (Balasa, ICCAD 94)	384x12	160x6, 160x6 224x6, 224x6	29391	1.7	**3.1%**
5	MPEG I (Thordarson,TR 95-8)	54x18	56x7, 56x7, 56x4	9325	1.7	**15.3%**
6	MPEG II (Thordarson, TR 95-8)	128x17	128x5, 128x6, 128x6	16383	1.5	**0.0%**
7	Singular Value Decomposition I (Balasa, ICCAD 94)	1365x16	224x8, 224x8 224x8, 224x8 224x8, 224x8 224x8, 224x8 224x8, 224x8 256x8, 256x8	125635	35717.6	**2.2%**

Figure 15: Memory mapping result II

by the *exhaustive* algorithm are less than the area of the corresponding designs produced by the *linear* algorithm by upto 15.5%. Even though the target library contains modules with word-count not equal to a power-of-two, the *linear* algorithm exclusively uses modules with word-count equal to a power-of-two. Recall that the *linear* algorithm approximates the word-count of a memory module that has a word-count not equal of a power-of-two with the next lower power-of-two. However, with this approximation these modules become cost inefficient; hence the algorithm tends to select ones with word-count equal to a power-of-two. The *exhaustive* algorithm has been able to produce better designs mainly because it allows the selection of modules with word-count not equal to a power-of-two. From the last columns

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

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

Figure 16: Memory mapping result III

in these two figures, we observe that most of these mappings have been generated only in a few seconds.

Figures 16 through 19 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. Figures 16 and 17 report the area efficient designs generated by the *linear* and *exhaustive* algorithms respectively. We report area in terms of approximate number of CLBs (Configurable Logic Blocks [Xili93]) for the resultant design. Figures 18 and 19 report the delay efficient designs generated by the *linear* and *exhaustive* algorithms respectively. The delay for these designs represents the worst case access delay in nanoseconds.

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

Example			Mapping result			
#	*Name*	*Size*	*Design*	*Area* (clb)	*Run-time* (sec)	*Improvement over Linear (Fig 16)*
1	Differential Heat Release (Ramachan,EDAC 94)	469x16	32x8, 32x8 32x8, 32x8 : 32x8, 32x8	328	24.0	1.5%
2	Neural Network Chip (Rose, ICCAD94)	160x8	32x8 32x8 32x8 32x8 32x8	53	1.2	3.6%
3	Interlace Scan (Philips, cicc91)	360x16	16x8, 16x8 32x8, 32x8 64x8, 64x8 128x8, 128x8 128x8, 128x8	257	2.8	0.0%
4	Medical Image Reconstruction (Balasa, ICCAD 94)	384x12	32x4, 32x8 32x4, 32x8 64x4, 64x8 128x4, 128x8 128x4, 128x8	202	2.9	1.9%
5	MPEG I (Thordarson, TR 95-8)	54x18	32x4, 32x8, 32x8 32x4, 32x8, 32x8	50	0.8	0.0%
6	MPEG II (Thordarson, TR 95-8)	128x17	32x4, 32x8, 32x8 32x4, 32x8, 32x8 32x4, 32x8, 32x8 32x4, 32x8, 32x8	99	0.8	8.1%

Figure 17: Memory mapping result IV

From these results, we observe that a wide variety of designs are generated as we vary the cost function and the algorithm. As before, designs produced by the *exhaustive* algorithm are often different from the corresponding designs produced by the *linear* algorithm. However, the design metrics from the *exhaustive* algorithm show significantly smaller improvement (with respect to the metrics from *linear* algorithm) using Xilinx library as compared to using the Toshiba gate array library. For example, the area-efficient designs produced by the *exhaustive* algorithm are at most 8.1% smaller than the corresponding designs from the *linear* algorithm (Figure 17). In the domain of the delay-efficient designs, the *exhaustive* algorithm generates designs with the same cost as generated by the *linear* algorithm (Figure 19). This is because all the RAM modules in the Xilinx library have word-count equal to a power-of-two; the *linear* algorithm can make effective use of each of these modules. Once

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

Example			Mapping result		
#	*Name*	*Size*	*Design*	*Delay* (ns)	*Run–time* (sec)
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	1.0
2	Neural Network Chip (Rose, ICCAD94)	160x8	32x8 32x8 32x8 32x8 32x8	28.60	0.6
3	Interlace Scan (Philips, cicc91)	360x16	32x4, 32x4, 32x4, 32x4 : 32x4, 32x4, 32x4, 32x4 16x4, 16x4, 16x4, 16x4	31.90	1.3
4	Medical Image Reconstruction (Balasa, ICCAD 94)	384x12	32x8, 32x8 32x8, 32x8 : 32x8, 32x8	31.90	0.6
5	MPEG I (Thordarson, TR 95–8)	54x18	32x4, 32x8, 32x8 32x4, 32x8, 32x8	21.00	0.9
6	MPEG II (Thordarson, TR 95–8)	128x17	32x8, 32x8, 32x8 32x8, 32x8, 32x8 32x8, 32x8, 32x8 32x8, 32x8, 32x8	23.50	0.7

Figure 18: Memory mapping result V

again these mappings are produced very quickly, often in less than a second.

Figure 20 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. Once again the source memory modules in this experiment are extracted from the designs reported in the literature as well as from the industrial designs. The first two modules in the experiment are from the Singular Value Decomposition algorithm reported in [BaCM94]; the third module is from an Image Processing algorithm reported in [Lim90]; the last example is from the MPEG design reported in [ThGa95]. Note that these modules are relatively larger in size, since these are to be realized by larger off-chip modules. The

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

Example			Mapping result			
#	*Name*	*Size*	*Design*	*Delay* (ns)	*Run–time* (sec)	*Improvement over Linear (Fig 18)*
1	Differential Heat Release (Ramachan,EDAC 94)	469x16	32x8, 32x8 32x8, 23x8 : 32x8, 32x8	31.90	6.2	**0.0%**
2	Neural Network Chip (Rose, ICCAD94)	160x8	32x8 32x8 32x8 32x8 32x8	28.60	0.5	**0.0%**
3	Interlace Scan (Philips, cicc91)	360x16	32x8, 32x8 32x8, 32x8 : 32x8, 32x8	31.90	1.7	**0.0%**
4	Medical Image Reconstruction (Balasa, ICCAD 94)	384x12	32x8, 32x8 32x8, 32x8 : 32x8, 32x8	31.90	2.0	**0.0%**
5	MPEG I (Thordarson, TR 95–8)	54x18	32x4, 32x8, 32x8 32x4, 32x8, 32x8	21.00	0.6	**0.0%**
6	MPEG II (Thordarson, TR 95–8)	128x17	32x8, 32x8, 32x8 32x8, 32x8, 32x8 32x8, 32x8, 32x8 32x8, 32x8, 32x8	23.50	2.0	**0.0%**

Figure 19: Memory mapping result VI

mapping results in Figure 20 demonstrate that our mapping approach is applicable to off-chip modules as well. Our algorithms can synthesize large off-chip modules in seconds.

5.1 Summary of experiments

From the results in Figures 14 to 20, 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.

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

Example			Mapping result		
#	*Name*	*Size*	*Design*	*Price* **($)**	*Run-time* **(sec)**
1	Singular Value Decomposition II (Balasa, ICCAD 94)	146590x16	256kx8, 256kx8	16.36	1.1
2	Singular Value Decomposition III (Balasa, ICCAD 94)	103550x16	128kx8, 128kx8	10.90	1.6
3	Image Processing (Lim, Prentice Hall 90)	1Mx16	512kx8, 512kx8 512kx8, 512kx8	65.44	0.9
4	MPEG III (Thordarson,TR 95-8)	512kx32	512kx8,512kx8,512kx8,512kx8	65.44	1.5

Figure 20: Memory mapping result VII

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.* We also provide a choice of two algorithms: *linear* and *exhaustive.* Depending on the design requirements, a user can select any combination of algorithm and cost function. 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.

From the tables in Figures 14 to 19, 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 17) as well as irregular structures, where a memory array is built using a set of different memory module types (Example 1 in Figure 15). We also note that even for the same memory module, we get different designs as we vary the cost function and the algorithm. For instance, consider the first example (Differential Heat Release computation) in Figures 14 to 19. Out of the six mapping results, we have five different designs for the same source memory module. In other words, a different memory configuration is generated for different optimizing cost function.

Also, quite often the resultant designs are counter-intuitive. For instance, consider Example 6 in Figure 14. 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.

If we analyze the run-time (the sixth column in these tables), we observe that we have been able to generate these designs very quickly, in the order of a few seconds[6]. The run-time is independent of the cost function used for optimizing the design. As far as the quality of the designs is considered, it depends on two factors: the target library and the algorithm. The

[6]The only exception is Example 7 in Figure 15, where the mapping result is generated in 35717.6 seconds.

the designs generated by the *exhaustive* algorithm are often better than those generated by the *linear* algorithm using a library that has modules with unrestricted word-count. However, this comes at the cost of an exponential increase in run-time for larger designs. For instance, the Example 7 in Figure 15 requires approximately 10 hours to generate the result using the *exhaustive* algorithm. The corresponding figure for the *linear* algorithm is just 2.5 seconds. On the other hand, if the target library contains only memory modules with word-count equal to a power-of-two, the *linear* algorithm generates designs that are very close to (< 8%) the designs produced by the *exhaustive* algorithm. Thus, if the design space is small, particularly with respect to word-count expansion, or if the target library is unrestricted, a user should use the *exhaustive* algorithm, since it usually generates better designs without paying too much penalty in the run-time. For designs that involve large search spaces in word-count expansion, the *linear* algorithm generates run-time efficient designs.

We conclude this section with a comment. Our High Level Library Mapping approach for memories is well-suited for synthesizing a source module with target modules coming from standard libraries with macros or pre-built memory modules. It clearly is not suited for memory generators that can generate a memory module given its size (in a limited domain) on demand.

6 Conclusion

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 the 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 presented two implementations of the memory mapping approach: a run-time efficient *linear* algorithm and a cost efficient *exhaustive* algorithm. These algorithms could 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 our mapping approach can generate a wide variety of cost-effective memory designs, often conter-intuitive ones, based on the user-given cost function, target library and the mapping algorithm used. The *exhaustive* algorithm can sometimes generate better designs (than the *linear* algorithm) for an exponential increase in run-time. Hence the exhaustive algorithm is suitable for examples with smaller search space. Our *linear* heuristic generates efficient designs in a matter of seconds, often with the same cost as generated by the *exhaustive* algorithm for restricted (yet realistic) libraries. This is a useful aid for designers, since the search space for these mapping problems are large and very labor intensive.

Future work would involve developing similar high-level library mapping techniques for other sequential datapath components such as counters and shift-registers, in an effort to-

wards a comprehensive mapping scheme for a complete netlist of RT level components.

7 Acknowledgements

This research was supported by SRC contract #94-DJ-146. We are grateful for their support. We also thank Prof. Lueker for discussions on the bin-packing problem.

References

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

[BaCM93] F. Balasa, F. Catthoor and H. De Man, "Exact Evaluation of Memory Size for Multi-dimensional Signal Processing Systems," *Intl. Conf. on Computer Aided Design*, 1993.

[BaCM94] F. Balasa, F. Catthoor and H. De Man, "Data-driven Memory Allocation for Multi-dimensional Signal Processing Systems," *Intl. Conf. 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.

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

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

[DuJh93] N. Dutt and P. Jha, "RT Component Sets for High-Level Design Applications," *Proc. of 1st Asia Pacific Conf. on HDL Standards and Applications,* December 1993.

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

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

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

[Lim90] J. Lim, "Two-Dimensional Signal Image Processing," *Prentice Hall Signal Processing Series*, 1990.

[Lipp91] P. Lippens et. al, "Memory Synthesis for High-Speed DSP Applications," *Proceedings of the IEEE Custom Integrated Circuits Conference*, 1991.

[LMVW93] P. Lippens, J. Meerbergen, W. Verhaegh and A. Werf, "Allocation of Multiport Memories for Hierarchical Data Streams," *Intl. Conf. on Computer Aided Design*, November 1993.

[MaLa94] P. Marwedel and B. Landwehr, "Exploitation of Component Information in a RAM-based Architectural Synthesis System," *Novel Approaches in Logic and Architecture Synthesis,* G. Saucier, Editor, Chapman and Hall.

[RaGC93] L. Ramachandran, D. Gajski and V. Chaiyakul, "An Algorithm for Array Variable Clustering," *The European Design Automation Conference,* February 1994.

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

[ThGa95] A. Thordarson and D. Gajski, "Manual Synthesis of the MPEG Algorithm," *Technical Report 95-8, University of California at Irvine,* March 1995.

[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.

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

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