Library Mapping for Memories[†]

Pradip K. Jha§ and Nikil D. Dutt‡
§IBM EDA Lab, East Fishkill, NY 12533, USA
‡ICS Department, UC Irvine, CA 92717, USA

Abstract

We present a library mapping technique that synthesizes a source memory module from a library of target memory modules. We define the library mapping problem for memories, identify and solve the three subproblems of port, bit-width and size (word) mapping associated with this task and finally combine these solutions into an efficient memory mapping algorithm. 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 and the target library.

1 Introduction

Digital systems are typically 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), 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 [JhDu95a]. 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 paper, we present a library mapping technique that implements a source memory module with one or more memory modules from a target library. Given the specification of the source and the library modules in terms of word-count, bit-width and the port configurations, our memory mapping approach 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 on-chip as well as off-chip memory modules.

The paper is organized as follows. Section 2 describes related work. Section 3 defines the library mapping problem for memories and decomposes it into three subproblems. Section 4 combines the three subproblem formulations into an efficient memory mapping algorithm. Section 5 presents our experimental results that demonstrate the efficacy of our approach. The paper concludes with a summary.

2 Previous Work

Research in the use of memories for design automation systems at the RT and the behavioral levels has only recently received attention. These works can be broadly classified into two groups. The first group of work concentrates on translating the storage requirements in the input behavior onto logical memories. Research efforts described in [KiLi93] [RaGC94] [MaLa94] [LMVW93] [BaCM94] fall into this group. The second group of work maps these logical memories onto physical memories from a library. Our work fits in the second group.

Figure 1 summarizes the works in the domain of library mapping for memories by classifying these works

[†]This research was partially supported by SRC contract #95-DJ-146 and NSF grant CDA9422095.

ED&TC '97 on CD-ROM

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	?
[KaRo94]	Multiple	Multiple	Multiple	Single	Yes	Yes	?
Ours	*Single*	*Single*	*Multiple*	*Multiple*	*Yes*	*Yes*	*Yes*

Figure 1: Classification of memory mapping works

on the basis of the number and the type of logical and physical memory modules considered simultaneously. [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. 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.

3 Problem Definition

We define 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 and 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.

The library mapping problem for memories, thus, 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

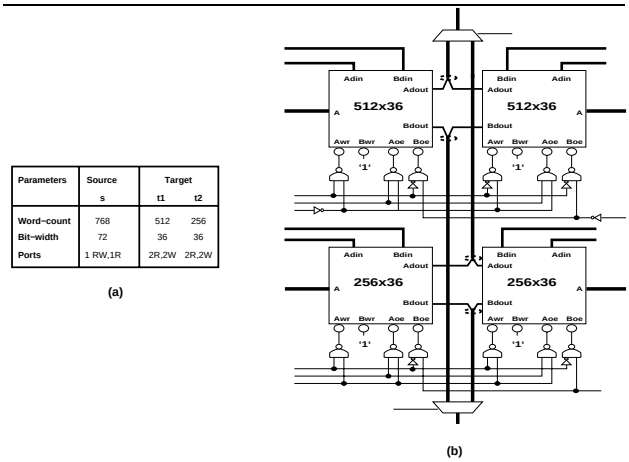


Figure 2: Sample library mapping for memories: (a) Source and target modules (b) Mapping result

are characterized by specifying the three parameters, namely the ports (Read, Write and Read-Write), the Number of 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 cost of a memory design is given by the sum of the cost of the target memory modules, the address decoding logic and the logic for multiplexing the output. The cost function could be Area-based, Delay-based or Price-based. For a memory m, we represent the number of words by $N(m)$, the bit-width by $B(m)$, the number of Read, Write and Read-Write ports by $R(m)$, $W(m)$ and $RW(m)$ respectively and area, delay and price by $A(m)$, $D(m)$ and $P(m)$ respectively.

We decompose the library mapping problem for memories into three subproblems: *port mapping*, *bit-width mapping* and *word mapping*, based on the three parameters used to specify a memory module. Next we discuss each of these subproblems; in Section 4 we combine these formulations to achieve a global solution to the complete memory mapping problem.

3.1 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. 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:

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

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

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

Based on these *port constraints*, we first select a set of target memory modules that satisfy the access rate requirement of the source memory module s and then use a simple one-to-one port assignment technique to map the ports of the source memory module onto ports of target modules. Details of the port mapping algorithm are outlined in [JhDu95].

3.2 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 of (one or more) 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.

The following integer linear programming formulation describes the area-based bit-width mapping problem concisely. 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 expression : $\sum_{i=1}^{m} x_i A(t_i)$, while satisfying the linear constraint : $\sum_{i=1}^{m} x_i B(t_i) \geq B(s)$. 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. 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 use a variation of an exhaustive search (enumeration) scheme from [CoLR90] to find the optimal solution to the bit-width mapping problem; details of the algorithm are described in [JhDu95].

3.3 Word mapping

Word mapping, in the context of memory mapping, refers to the task of realizing the word-count requirement of the source memory component s using a set of (one or more) 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.

The word mapping problem (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.

Algorithm 3.1 describes a scheme to perform the task of word mapping efficiently. 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 5 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 5.4). 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. Note that the algorithm would require *linear* run-time for sorted T.

Algorithm 3.1 : Word mapping

INPUT: Source s; Target T; Cost C.

OUTPUT: Word mapping of s.

1 approximate the target module word-count;
2 sort T in decreasing word-count of its elements;
3 delete redundant elements from T;
4 $best\_map = \phi$, $partial\_map = \phi$, $word\_left = \mathrm{N}(s)$;
5 **while** $word\_left \neq 0$ **do**
  5.1 $curr\_mem$ = next memory from T;
  5.2 $curr\_word = \mathrm{N}(curr\_mem)$;
  5.3 $curr\_map = \lceil \frac{word\_left}{curr\_word} \rceil * curr\_mem$;
  5.4 **if** $\mathrm{C}(partial\_map + curr\_map) < \mathrm{C}(best\_map)$ **then**
    5.4.1 $best\_map = partial\_map + curr\_map$;
  5.5 **end if**;
  5.6 **if** $curr\_mem$ is the last element in T **or**
      $\mathrm{S}(curr\_map) == word\_left$ **then**
    5.6.1 $word\_left = 0$;
  5.7 **else**
    5.7.1 $partial\_map = partial\_map + \lfloor \frac{word\_left}{curr\_word} \rfloor * curr\_mem$;
    5.7.2 $word\_left = word\_left - \lfloor \frac{word\_left}{curr\_word} \rfloor * curr\_mem$;
  5.8 **end if**;
6 **end while**;
7 **return** $best\_map$;

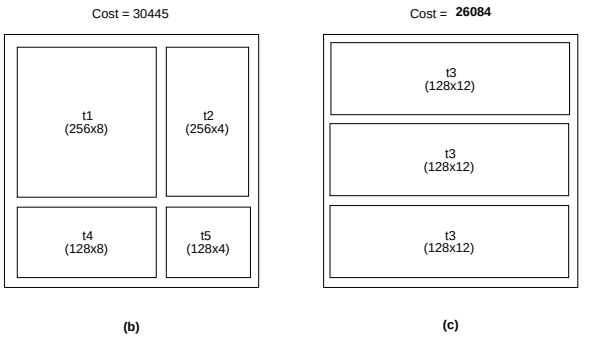
4 Memory Mapping Algorithm

Our memory mapping 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 (see [JhDu95] for the detailed algorithm).

Figure 3 shows an example for memory mapping. The source and the target memory modules for this example are shown in Figure 3(a). The cost function for this example is an area measure approximated by the equivalent gate-count. Figure 3(b) shows the first design that have been generated by performing the task of word mapping followed by bit-width mapping. The figure shows the layout of the design using four target memory modules : *t1, t2, t4 and t5*. The cost of this design is 30445 gates. Figure 3(c) shows another design that has been generated by performing the task of word mapping and the bit-width mapping in the reverse order. The design uses three instances of memory module *t3*. The cost of this design is 26084 gates. Thus, the second design is preferred. 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.

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)



(b) (c)

Figure 3: A memory mapping example (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

5 Experimental results

We applied our memory mapping algorithm on several examples from the literature and copared these designs against the ones produced by an exhaustive memory mapping algorithm. We summarize the mapping results for delay-efficient designs here; [JhDu95] describes the detailed results including the area-efficient and price-efficient designs using other libraries.

Figure 4 summarizes the results for the delay-efficient designs using the RAM macro modules of the Xilinx 400 series FPGA[Xili93]. The first three columns in the result table of this figure describe the source memory module, in terms of its source design and size. 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 delay (worst case access delay in nanoseconds) of the design and the sixth column presents the run-time (in seconds) for our algorithm on a SUN Sparc Station 5. The last column reports the percentage difference between the delay of the designs produced by our mapping algorithm and an exhaustive algorithm.

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

Example			Mapping result			
#	Name	Size	Design	Delay (ns)	Run−time (sec)	Diff wrt Exhaustive
1	Differential Heat Release (RaGC94, EDAC 94)	469x16	32x4, 32x4, 32x4, 32x4 ⋮ 32x4, 32x4, 32x4, 32x4 16x4, 16x4, 16x4, 16x4 16x4, 16x4, 16x4, 16x4	31.90	1.0	**0.00%**
2	Neural Network Chip (KaRo94, ICCAD94)	160x8	32x8 32x8 32x8 32x8 32x8	28.60	0.6	**0.00%**
3	Interlace Scan (Lipp91, CICC91)	360x16	32x4, 32x4, 32x4, 32x4 ⋮ 32x4, 32x4, 32x4, 32x4 16x4, 16x4, 16x4, 16x4	31.90	1.3	**0.00%**
4	Medical Image Reconstruction (BaCM94, ICCAD 94)	384x12	32x8, 32x8 32x8, 32x8 ⋮ 32x8, 32x8	31.90	0.6	**0.00%**
5	MPEG I (ThGa95, TR 95−8)	54x18	32x4, 32x8, 32x8 32x4, 32x8, 32x8	21.00	0.9	**0.00%**
6	MPEG II (ThGa95, TR 95−8)	128x17	32x8, 32x8, 32x8 32x8, 32x8, 32x8 32x8, 32x8, 32x8 32x8, 32x8, 32x8	23.50	0.7	**0.00%**

Figure 4: Memory mapping results

From our experimental results, we observe that our approach to memory mapping is quite versatile. Our approach can synthesize source memory modules of varying word-count and bit-width. These memory modules come from designs of varying complexity including image processing applications and an industrial design of the 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 and size of memory modules. Our approach allows the user to select the optimizing cost function. We currently support three cost functions, namely *Area, Delay* and *Price*. Our algorithm generates a wide variety of memory designs that include the regular structures, where a memory array is built using a single memory module type (Example 2 in Figure 4) as well as irregular structures, where a memory array is built using a set of different memory module types (Example 1 in Figure 4). Furthermore, the resultant designs are often counter-intuitive, generating memory structures that a designer might not be able to generate manually. Examples of such designs are detailed in our technical report [JhDu95].

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. The last column in Figure 4 shows that the delay-efficient designs produced by our algorithm are of the same quality as the ones generated by the exhaustive algorithm. We also observed that the price-efficient designs are of the same quality and the area-efficient designs are at most 18.40% worse as compared to the designs generated by the exhaustive algorithm. Note that the better designs generated by the exhaustive algorithm come at the cost of potential increase in run-time by orders of magnitude. In our experiments, we came across an example that requires approximately 10 hours to generate a design with the exhaustive algorithm and only 2.5 seconds with our algorithm. Our experiments show that the proposed memory mapping approach generates a wide variety of cost-effective designs quickly and based both on a user-given cost function and the given target library.

6 Summary

In this paper we presented a library mapping scheme for memories 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 experimental results run on several industrial and literature-based examples demonstrate the efficacy of our approach.

7 References

[BaCM94] F. Balasa, F. Catthoor and H. De Man, "Data-driven Memory Allocation for Multi-dimensional Signal Processing Systems," *ICCAD 94.*

[BaGa95] S. Bakshi and D. Gajski, "A Memory Selection Algorithm for High-Performance Pipelines," *EDAC 95.*

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

[JhDu95] P. Jha and N. Dutt, "High-Level Library Mapping for Memories," *TR# 95-37, UC Irvine, 1995.*

[JhDu95a] P. Jha and N. Dutt, "Design Reuse through High-Level Library Mapping," *ED&TC-95.*

[KaRo94] D. Karchmer and J. Rose, "Definition and Solution of the Memory Packing Problem for Field-Programmable Systems," *ICCAD 94.*

[KiLi93] T. Kim and C. Liu, "Utilization of Multiport Memories in Data Path Synthesis," *DAC 93.*

[Lipp91] P. Lippens et al. "Memory Synthesis for High-Speed DSP Applications," *CICC*, 1991.

[LMVW93] P. Lippens, et al. "Allocation of Multiport Memories for Hierarchical Data Streams," *ICCAD 93.*

[MaLa94] P. Marwedel and B. Landwehr, "Exploitation of Component Information in a RAM-based Architectural Synthesis System," *Logic and Architectural Synthesis,* G. Saucier (Ed.), Chapman & Hall 95.

[RaGC94] L. Ramachandran, D. Gajski and V. Chaiyakul, "An Algorithm for Array Variable Clustering," *EDAC 94.*

[ScTh95] H. Schmit and D. Thomas, "Array Mapping in Behavioral Synthesis," *ISSS 95.*

[ThGa95] A. Thordarson and D. Gajski, "Manual Synthesis of the MPEG Algorithm," *TR# 95-8, UC Irvine, 1995.*

[Ti94] "Semiconductor Group Distribution, USA and Canada Suggested Resale Pricing Guide," *Texas Instruments,* 94.

[Tosh90] "Toshiba ASIC Gate Array Library," *Toshiba Corporation, Tokyo, Japan,* 90.

[Xili93] "Development System Libraries Guide," *Xilinx, San Jose, CA* 93.