

PRODUCT TERM MODE EMBEDDED  
MEMORY ARRAYS: ALGORITHMS  
AND ARCHITECTURES

by

Ernest Wei-Lang Lin

B.A.Sc, University of British Columbia, 1999

A thesis submitted in partial fulfillment of the  
requirements for the degree of

Master of Applied Science

in

The Faculty of Graduate Studies

Department of Electrical and Computer Engineering

We accept this thesis as conforming to the  
required standard:

---

---

---

---

The University of British Columbia

September 2001

© Ernest Wei-Lang Lin, 2001

## ABSTRACT

### PRODUCT TERM MODE EMBEDDED MEMORY ARRAYS: ALGORITHMS AND ARCHITECTURES

Field-Programmable Gate Arrays (FPGAs) are integrated circuits that can be programmed to implement virtually any digital circuit. Due to the ease with which a user can implement a circuit, FPGAs have become a low-cost, fast-turnaround time alternative to more traditional implementation technologies such as mask-programmed gate arrays and application-specific integrated circuits (ASICs). However, one of the major pitfalls of FPGAs is the area and speed penalty inherent in the technology. In order to help “bridge the gap” between FPGAs and ASICs, intelligent computer-aided design (CAD) tools that use the FPGA as efficiently as possible are needed. In this thesis, we focus on mapping logic to the on-chip memory arrays. In particular, our focus is on the architecture of the product-term mode memory arrays, and the CAD algorithms that target those architectures.

First, we focus on an intelligent technology mapping algorithm that maps a circuit to product term mode memory arrays. We show that the algorithm can pack 22.8% more logic blocks into product term mode memory arrays over using conventional memory arrays alone. Second, we focus on the architecture of the product term mode memory array itself, and present several architectural enhancements for increasing the efficiency of the memory array in implementing logic.

# TABLE OF CONTENTS

<b>Abstract.....</b>	<b>ii</b>
<b>List of Figures.....</b>	<b>vii</b>
<b>Acknowledgements.....</b>	<b>ix</b>
<b>Chapter 1: Overview and Introduction .....</b>	<b>1</b>
<i>1.1 Motivation.....</i>	<i>1</i>
<i>1.2 Research Goals.....</i>	<i>4</i>
<i>1.3 Research Approach .....</i>	<i>6</i>
<i>1.4 Organization of This Thesis .....</i>	<i>7</i>
<b>Chapter 2: Background and Previous Work .....</b>	<b>8</b>
<i>2.1 Programmable Logic: FPGAs and CPLDs .....</i>	<i>8</i>
<i>2.2 An Overview of FPGA Architectures .....</i>	<i>10</i>
<i>2.2.1 Logic Resources .....</i>	<i>10</i>
Lookup-Tables.....	11
Product Terms.....	14
<i>2.2.2 Routing Resources .....</i>	<i>16</i>
Routing Between Logic Blocks.....	17

Routing Inside Logic Blocks .....	18
2.2.3 Memory Resources .....	19
2.3 <i>FPGA Computer-Aided Design Flow</i> .....	20
2.4 <i>Mapping Logic to Memory</i> .....	21
2.4.1 Terminology .....	22
2.4.2 Conventional Mode Memory Arrays .....	25
SMAP .....	25
EMB_Pack .....	26
2.4.3 Product Term Mode Memory Arrays .....	27
Hybridmap .....	29
2.5 <i>Focus and Contributions of This Thesis</i> .....	30
<b>Chapter 3: Technology Mapping to Product-Term Memory Arrays .....</b>	<b>32</b>
3.1 <i>Motivation</i> .....	32
3.2 <i>Problem Description</i> .....	33
3.3 <i>High-Level Description of pMapster</i> .....	34
3.4 <i>Detailed Description of pMapster</i> .....	35
3.4.1 Choosing a Candidate for Collapsing .....	38
Calculating the Number of Resultant Product Terms .....	38
3.4.2 Collapsing a Candidate Fanin Node .....	39
Candidate Fanin with Single Fanout .....	39
Candidate Fanin with Multiple Fanouts .....	40
3.4.3 Ending the Mapping Loop .....	42

3.4.4 Choosing the Next Seed Node.....	43
3.5 <i>Implementation</i> .....	43
3.6 <i>Summary</i> .....	44
<b>Chapter 4: Product-Term Memory Architectural Issues .....</b>	<b>45</b>
4.1 <i>Baseline Architecture</i> .....	46
4.2 <i>Architectural Enhancement Proposals</i> .....	47
4.2.1 Macrocell Granularity.....	47
4.2.2 Sharing Macrocell Outputs.....	49
4.2.3 Multiple Parallel Expanders .....	51
4.3 <i>Memory Configuration Architectural Issues</i> .....	53
4.3.1 Width of the Memory .....	55
4.3.2 Depth of the Memory .....	55
4.4 <i>Changes to pMapster to Support Architectural Enhancements</i> .....	55
4.4.1 Macrocell Granularity.....	56
4.4.2 Sharing Macrocell Outputs.....	57
4.4.3 Multiple Parallel Expanders .....	59
4.5 <i>Summary</i> .....	62
<b>Chapter 5: Results .....</b>	<b>64</b>
5.1 <i>Experimental Setup</i> .....	65
5.2 <i>pMapster Versus Hybridmap</i> .....	65
5.3 <i>Product Term Memory Versus Conventional Memory</i> .....	68

<i>5.4 Effect of Macrocell Granularity</i> .....	72
<i>5.5 Effect of Sharing</i> .....	74
<i>5.6 Effect of Algorithm on Experimental Results</i> .....	76
<i>5.7 Limitations on Experimental Results</i> .....	76
<i>5.8 Summary</i> .....	77
<b>Chapter 6: Conclusions</b> .....	<b>78</b>
<i>6.1 Future Work</i> .....	80
<i>6.2 Contributions of This Work</i> .....	81
<b>References</b> .....	<b>84</b>

## LIST OF FIGURES

<i>Number</i>	<i>Page</i>
Figure 1.1 An FPGA with Memory.....	3
Figure 2.1 Island-Style Architecture .....	11
Figure 2.2 A 4-Input Lookup-Table .....	12
Figure 2.3 Inside a Lookup-Table FPGA Logic Block.....	13
Figure 2.4 Inside a Lookup-Table FPGA Logic Element (Simplified) .....	14
Figure 2.5 Inside a Product Term FPGA Logic Block .....	15
Figure 2.6 Inside a Product Term FPGA OR-Plane .....	16
Figure 2.7 Routing in an FPGA.....	17
Figure 2.8 Structure of a Programmable Switch .....	18
Figure 2.9 FPGA CAD Flow.....	20
Figure 2.10 A Network of Nodes Representing a Circuit.....	23
Figure 2.11 A FFS and the MFFS of the Node Labeled “x” .....	24
Figure 2.12 Collapsing Node x into Node y.....	25
Figure 2.13 Structure of a Product Term Memory Array .....	28
Figure 3.1 Mapping Part of a Network of LUTs to Memory.....	34
Figure 3.2 pMapster Program Flow .....	35
Figure 3.3 Mapping to One Memory Array (pseudocode) .....	36
Figure 3.4 Processing a Seed Node (pseudocode).....	37
Figure 3.5 Shaded Nodes are Single Fanout .....	40
Figure 3.6 Shaded Nodes are Multiple Fanout.....	41
Figure 3.7 Creating a Secondary Seed .....	42

Figure 4.1 Baseline Macrocell Architecture .....	46
Figure 4.2 Macrocell with Granularity of 4.....	48
Figure 4.3 When Output Sharing is Useful .....	50
Figure 4.4 Basic Macrocell with Output Sharing.....	50
Figure 4.5 When Multiple Parallel Expanders is Useful.....	52
Figure 4.6 Basic Macrocell with Multiple Parallel Expanders .....	52
Figure 4.7 Effect of Memory Parameters .....	54
Figure 4.8 Impact of “Fitness” on Number of Macrocells Used .....	59
Figure 4.9 Using the Parallel Expander to Join Macrocells.....	60
Figure 4.10 How Macrocells Limit a Seed’s Ability to Share .....	61
Figure 4.11 Calculating Product Terms and Macrocells (pseudocode).....	62
Table 5.1 Comparing pMapster and HybridMap .....	66
Figure 5.1 Product Term Vs. Conventional Memory (graph) .....	70
Table 5.2 Product Term Vs. Conventional Memory (table) .....	71
Figure 5.2 Effect of Macrocell Granularity (graph).....	73
Table 5.3 Effect of Macrocell Granularity (table) .....	73
Table 5.4 Effect of Macrocell Sharing .....	75

## ACKNOWLEDGMENTS

First of all, I wish to thank my supervisor, Dr. Steven Wilton. Without his great advice and feedback, I would not be presenting this work today. I have really enjoyed working with him and hope I get the chance to collaborate with him on future projects.

This work was supported by the BC Advanced Systems Institute and the Natural Sciences and Engineering Research Council of Canada. Their support is greatly appreciated.

I would also like to thank Peter, Tony, Steve, and maniacal “sisters” Kara and Danna, for sharing their senses of humour with me and for putting up with my antics in the lab for the past two years.

I would also like to thank Alicia. She has been a constant source of motivation and support during the writing of this thesis.

Finally, I dedicate this thesis to my parents, Ran and Jane Lin, for they were the first to introduce me to science and problem solving, and ultimately, if it were not for their efforts, I probably would not be an engineer today.

# *Chapter 1*

## OVERVIEW AND INTRODUCTION

### **1.1 Motivation**

Field Programmable Gate Arrays (FPGAs) are highly specialized integrated circuits that can be used to implement virtually any digital circuit. Current FPGAs have extremely large capacities for logic and memory, making them suitable for implementing entire systems on a single chip. Modern FPGAs also contain a myriad of features, such as low power design, support for a plethora of I/O standards, dual-ported or single-port on-chip RAM, and built-in clock management [1, 2, 13, 28, 29, 30, 31, 32, 33].

Field Programmable Gate Arrays belong to a class of integrated circuits called *programmable logic devices* that also include Complex Programmable Logic Devices (CPLDs) and Mask Programmable Gate Arrays (MPGAs). Programmable devices differ from “traditional” Application-Specific Integrated Circuits (ASICs) in the sense that once a circuit is implemented on an ASIC, it cannot be changed after fabrication. FPGAs and CPLDs can be programmed and reprogrammed indefinitely by the user after fabrication. However, nothing is free; this flexibility comes at a price. There is an area and speed penalty associated with programmable devices. Typically, an ASIC implementation will be several times smaller than an equivalent FPGA implementation, and can operate at clock

frequencies several times faster than the FPGA implementation [6]. An FPGA's reprogrammability, however, makes it an ideal platform for prototyping designs.

As FPGAs are increasingly being used for implementing systems, on-chip memory is becoming an important feature found on most modern FPGAs [1, 2, 13, 30, 31, 32]. Since these memories occupy a large portion of the total chip area, it is important that they are used efficiently. Specifically, if a circuit does not use memory, the chip area devoted to memory is wasted. However, this area need not be wasted if these memories are put to use in implementing logic.

Given that on-chip memories are already part of the FPGA, using them to implement logic is “free”, or of very little to no cost to the end-user. When memories are used to implement logic, a portion of the logic that would normally be implemented by the logic resources on an FPGA will now be implemented by the memory. This means that by simply using the memories to implement a portion of the circuit, fewer logic resources are needed to implement the same circuit. If, by using the memories to implement logic

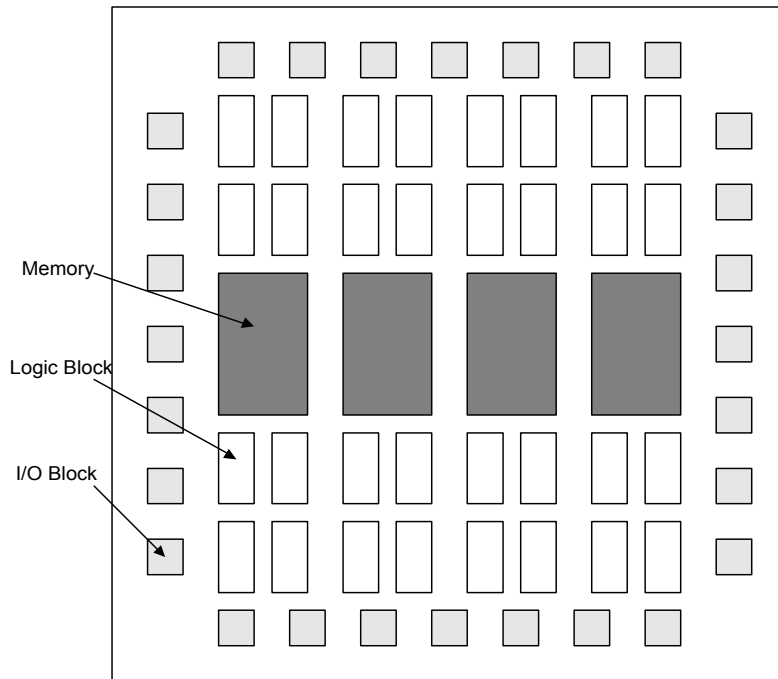


Figure 1.1 An FPGA with Memory

Product term memory arrays are conventional memory arrays with additional circuitry that allows sum-of-products functions to be implemented [15]. Product terms allow more efficient implementations of wide functions (such as state machine transition equations). If implemented using lookup-tables, many lookup-tables would be required to implement the same function. Product term memory arrays are a relatively new invention and therefore require study in order to determine whether these arrays are useful.

As important as the FPGA architecture, are the computer-aided design (CAD) tools used to map circuits to FPGAs. CAD tools are required to convert the user's circuit into the programming bits needed to program the FPGA. The CAD tools take a user's circuit description (usually in a high-level description language such as Verilog or VHDL) and convert it into a netlist of lookup-tables, which are then mapped to the FPGA's resources.

Finally, once FPGA resources are allocated, a programming file is created that, once loaded into the FPGA, sets the FPGA configuration bits and thus implements the user's circuit.

In order to effectively use the resources on an FPGA, the CAD tools must be aware of the target device architecture. If a CAD tool is not aware of the underlying architecture, it cannot make good decisions while compiling a user's design to an FPGA. Consequently, either the design will either take up more on-chip area than it has to, or run slower than it has to, or both.

## 1.2 Research Goals

The goal of this research is two-fold:

1. Provide an intelligent technology mapping CAD tool that targets product term memory arrays.
2. Use the above tool to study methods for increasing the efficiency of the product term memory arrays.

The first goal aims to create a novel algorithm for performing technology mapping on a circuit to a target that has product term memory arrays available for use. Technology mapping tools divide a circuit into parts, where each part is allocated to (or *mapped*) a specific element in the target, for example, a lookup-table or a memory array.

A new technology mapping algorithm is needed because there are very few existing algorithms that target product term memory arrays. We want to create an algorithm that

is different from existing algorithms in the sense that our algorithm will map a previously LUT-mapped circuit to product term memory. Taking a previously LUT-mapped circuit as input means the area benefit from mapping to product terms is easily quantified, since it is just the difference in number of LUTs.

The second goal involves analyzing the current product term architecture, and in particular, the macrocells which combine the product terms, and to determine how they might be used more effectively. From a product term usage point of view, the current architecture is inefficient. Those product terms that are common to more than one function cannot be shared; multiple copies of the product term are generated instead.

A novel architecture is needed for two reasons:

1. There is a drive towards implementing increasingly larger systems on FPGAs. It is important that every bit of on-chip area is used as efficiently as possible. Eliminating unnecessary redundancy (ie. duplicate product terms) is a very important step in reaching this goal of using area efficiently.
2. This avenue of research has never been explored previously. A product term memory architecture that supports the sharing of product terms cannot be found in any commercially available FPGAs today. Research is needed to determine if such an architecture is feasible, both from the CAD tool and the user's points of view.

Our goal is to devise new architectures to alleviate this problem and evaluate the effectiveness of these solutions. As any new architecture is useless without the CAD tools to support it, we will also create a new technology mapping tool to support these product term sharing architectures.

### **1.3 Research Approach**

We use an experiment-based approach to our research. The performance of the new CAD tool we developed in this work is measured in terms of how well it performs on several benchmark circuits. Similarly, the efficiency of the architectures we propose in this work is measured in terms of how well the CAD tools map circuits to them. The best algorithms and architectures are those that lead to the most logic being packed in the product term memory.

We first developed a new technology mapping CAD tool that maps a circuit to a product term memory array (or multiple arrays). Several benchmarks from the Microelectronics Center of North Carolina (MCNC) are used to test and gauge the tool's performance on real-world circuits. We then integrated the new CAD tool into an existing logic synthesis package, SIS [25]. We also developed several architectures that allow for the sharing of product terms. Sharing is an important feature since it reduces the necessity for multiple copies of the same product term to be implemented, thus increasing efficiency. Several modifications were made to our CAD tool in order to allow for mapping to architectures with product term sharing. The modified CAD tool was then used to map MCNC benchmarks to the product term memory array in order to gauge the efficiency of the

different sharing architectures. Finally, we draw conclusions on the effectiveness of product term memories (with and without product term sharing) on implementing logic.

#### **1.4 Organization of This Thesis**

This thesis is organized as follows: Chapter 2 describes previous work related to memory mapping and PLA mapping algorithms, and provides essential background information that puts this work into context. Chapter 3 introduces the problem of mapping logic to product term memories and describes our mapping algorithm for targeting product term memories. Chapter 4 describes various architectural enhancements to the basic macrocell as a way of increasing the efficiency of the macrocells. The modifications to the mapping algorithm that were required to implement mapping to the enhanced macrocell architectures are also described in Chapter 4. In Chapter 5, we present a number of results that compare our algorithm to existing mapping algorithms. We also evaluate the performance of the architectural enhancements described in Chapter 4, and provide discussions of the results. Finally, we conclude the thesis with a brief summary of the work presented, a summary of possible directions for future research, and a list of contributions made.

## *Chapter 2*

### BACKGROUND AND PREVIOUS WORK

In this chapter we present an overview of the basic architecture of an FPGA, and how logic is implemented within this framework. We will also discuss FPGA embedded memory arrays.

#### **2.1 Programmable Logic: FPGAs and CPLDs**

The end user has a number of choices when it comes to implementation technologies. The user can use non-programmable technologies, such as Application-Specific Integrated Circuits and Mask-Programmed Gate Arrays or they can use programmable technologies such as Field Programmable Gate Arrays and Complex Programmable Logic Devices. In this subsection, we focus on the difference between programmable and non-programmable technologies.

In the last fifteen years, FPGAs have come a long way towards becoming the technology of choice for companies to implement digital designs. FPGAs have gone from a mere thousand gates [33] on-chip to over 2.4 million [1, 2, 31, 32] gates on-chip. This has allowed FPGAs to go from being used for strictly small, glue logic-type applications to very large applications, such as entire systems. FPGAs are also getting larger and faster

every year. Roelandts predicts 50 million system gates and 16 Megabits of on-chip RAM in 2005 [24].

FPGAs are becoming the technology of choice for one main reason: FPGAs are programmable. An FPGA is able to implement *any* digital circuit by merely making the appropriate connections inside the FPGA (which are determined by the CAD tools). These connections can be made by configuring a set of SRAM bits (called configuration bits) embedded in the FPGA. Once a design is implemented in an FPGA, changes to the design can be made nearly instantly by simply reprogramming the configuration bits within the FPGA. The same changes made to the same design implemented on a non-programmable technology such as an ASIC involves making an entirely new chip, which can be very expensive and may take several months. Therefore using an FPGA means smaller non-recurring engineering (NRE) costs, and also a faster time-to-market. In today's marketplace, product cycles are often short and a faster time-to-market is a very important advantage over ASICs.

However, ASICs are still the technology of choice for large volume production. The reason is that ASICs still have an area and speed advantage over FPGAs. Circuits implemented on an ASIC will be smaller and faster than the same circuit on an FPGA. Improvements in the quality of FPGA CAD tools help; optimizations in the CAD tools can lead to more logic being packed onto the same FPGA (area optimization), or reduce the length of the critical path (delay optimization). Thus, improvements to the CAD tools are needed to close the area-speed gap between ASICs and FPGAs, so that the area-speed advantage currently enjoyed by ASICs becomes increasingly less significant.

## **2.2 An Overview of FPGA Architectures**

There are many different FPGA architectures available commercially. In this section, we discuss the structure of an FPGA from a general point of view.

Every FPGA on the market has three things in common:

1. Logic resources for implementing logic functions,
2. Routing resources for routing signals between logic blocks, and
3. I/O resources for connecting logic blocks to off-chip signals.

As FPGAs are increasingly being used to implement large systems, on-chip user storage is also a requirement in many applications. In order to meet this demand, most modern FPGAs have large on-chip memory arrays [1, 2, 13, 30, 31, 32]. In the Altera APEX20k, the embedded memory arrays can also function as a PAL (Programmable Array Logic). These memories are called product term mode embedded memory arrays, and will be discussed further in Section 2.4 below.

### **2.2.1 Logic Resources**

There are two primary technologies for implementing logic on an FPGA. Most FPGAs are lookup-table based devices, meaning the logic blocks contain lookup-tables. There are also some FPGAs on the market which are PLA-based devices, meaning the logic blocks implement logic using a product term based approach.

The organization of most modern FPGAs is known as the *island-style architecture*\*. The island-style architecture is illustrated in Figure 2.1. In this type of architecture, “islands” of logic resources (called “*logic blocks*”) are separated by a “sea” of routing channels and interconnect. The input and output pads are located along the perimeter of the chip.

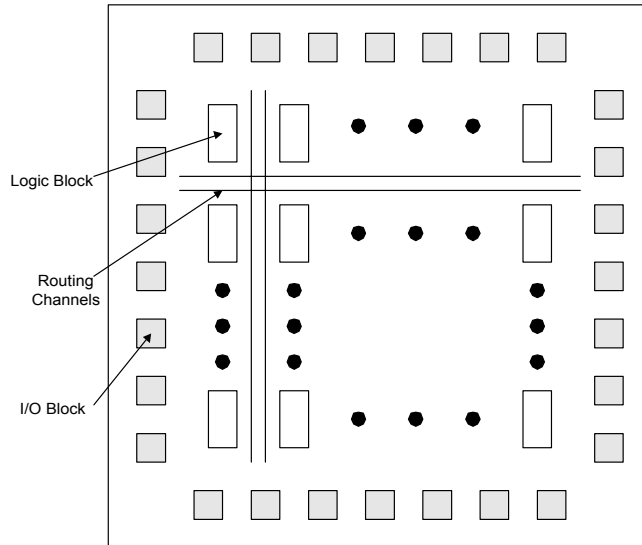


Figure 2.1 Island-Style Architecture

### *Lookup-Tables*

Most FPGAs on the market have logic resources that contain lookup-tables. A lookup-table is actually a small ROM, where the inputs to the lookup-table are the address inputs to the ROM, and the output data line is only one bit wide. This is how logic is implemented in a lookup-table; the ROM essentially stores the desired logic function’s truth table. The number of inputs to the lookup-table defines the size of the ROM; a k-input lookup-table (“k-LUT”) corresponds to a  $2^k$  bit ROM. A vast majority of today’s

---

\* I would like to point out that although the term “FPGA” primarily refers to lookup-table based devices and “CPLD” refers to PLA-based devices, the two terms are often used interchangeably. In this thesis, I will use the term “FPGA”

FPGAs use 4-LUTs as the basic logic resource. Figure 2.2 illustrates a four-input LUT, realized as a 16x1 memory block where the LUT inputs are the address inputs, and the LUT output is the data output.

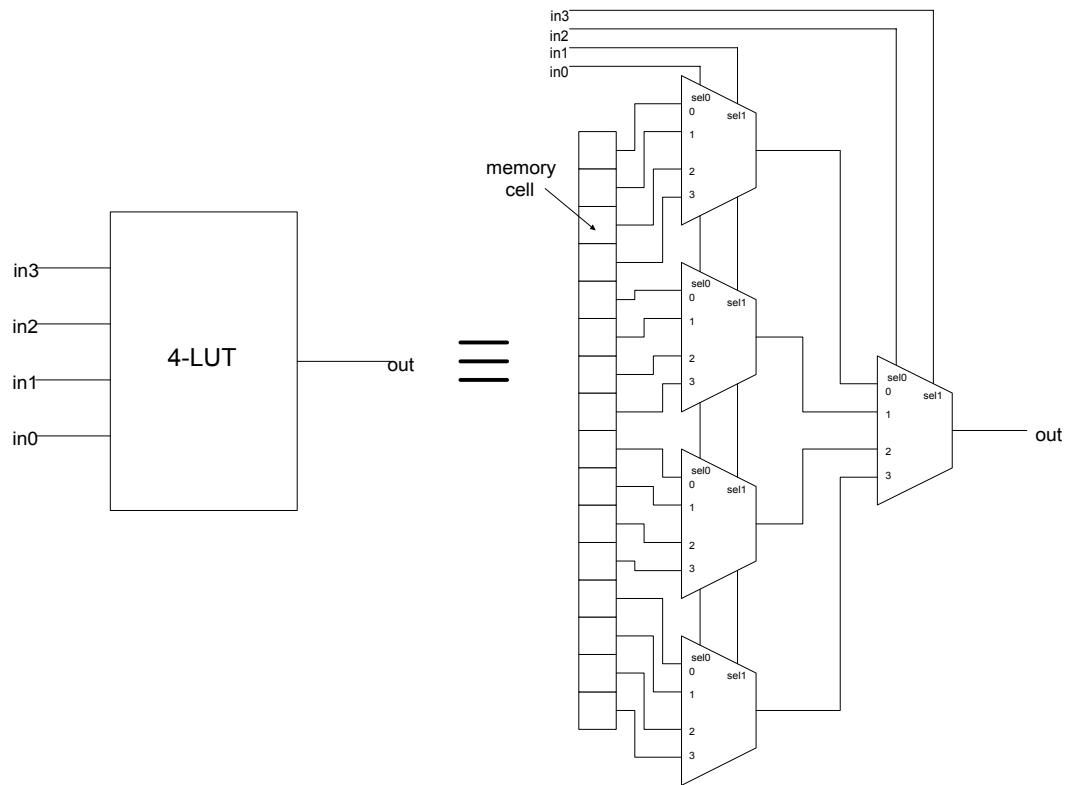


Figure 2.2 A 4-Input Lookup-Table

Lookup-tables can be used to implement logic efficiently because a k-input LUT can implement *any* function of its k inputs.

Each logic block in the FPGA is composed of a number of *logic elements* that are connected by programmable local interconnect. This is shown in Figure 2.3.

---

to describe any programmable logic device, both lookup-table and PLA-based.

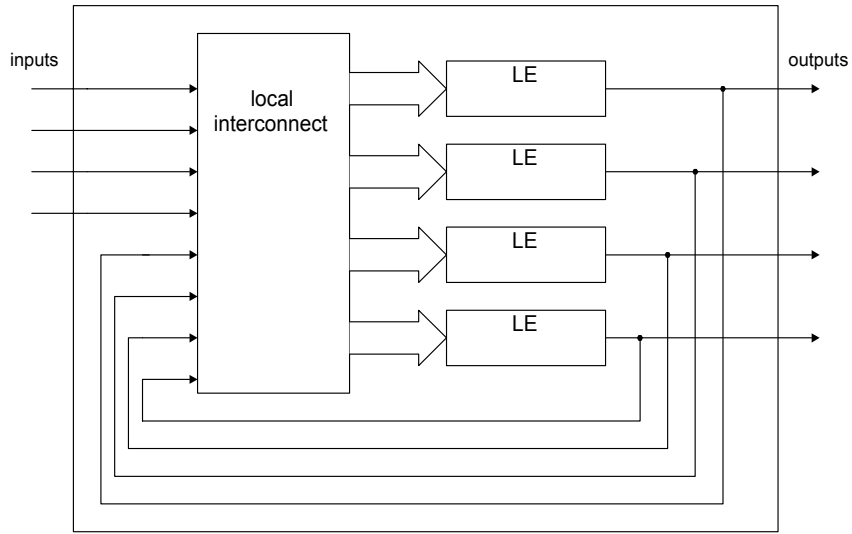


Figure 2.3 Inside a Lookup-Table FPGA Logic Block

Each logic element is in turn composed of a lookup-table and a storage element. The storage element is used when the logic element output is registered, and can be enabled or disabled by the user via a programming bit. Figure 2.4 illustrates the internal structure of a logic element.

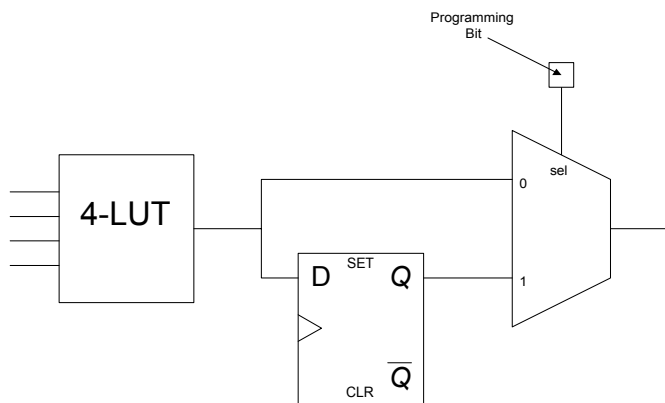


Figure 2.4 Inside a Lookup-Table FPGA Logic Element  
(Simplified)

*Product Terms*

The first programmable devices to use product terms to implement logic were PLAs (Programmable Logic Array) and PALs (Programmable Array Logic). Product term-based devices have two parts:

1. Product term generator (AND plane)
2. Product term distributor / macrocells (OR plane)

Product term-based FPGAs took the PAL concept and extended it to a higher level of integration, so that larger systems could be implemented with one chip. Instead of the lookup-tables found inside lookup-table FPGAs, the logic blocks inside a product term FPGA are essentially small PALs. Figure 2.5 illustrates the structure of a product-term FPGA logic block.

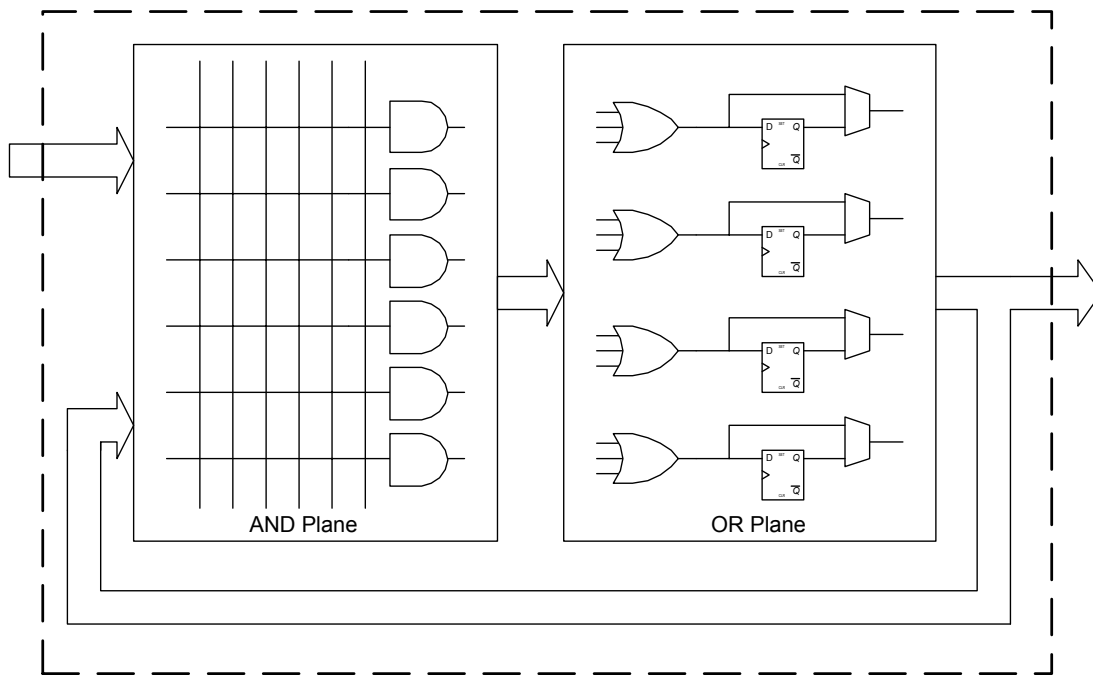


Figure 2.5 Inside a Product Term FPGA Logic Block

Product terms are generated in the product term generator, which is essentially the AND-plane of the PAL-like logic block. The product term generator architectures vary between vendors, but as an example, the Xilinx XPLA3 architecture can generate up to 48 product terms per logic block, from the 36 literals (available in both true and complement forms) that are inputs to the logic block [28].

The product term distributors / macrocells, which essentially comprise the OR-plane of the PAL-like logic block, also contain extra circuitry to allow efficient implementations of specialized functions, such as adders and counters. The macrocell usually also contains a storage element, which allows the output to be registered. The user can enable or disable this via a programming bit. Figure 2.6 illustrates the structure of the OR-plane.

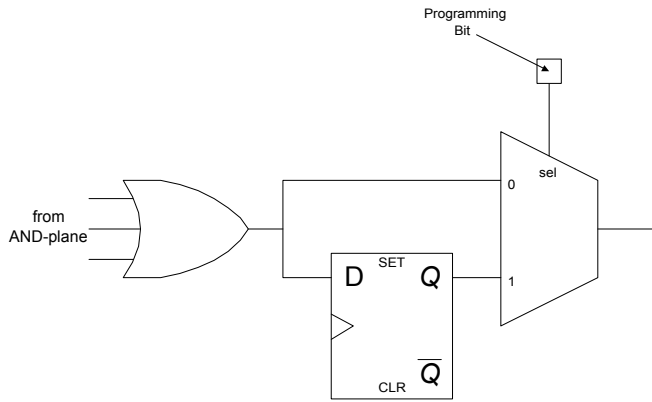


Figure 2.6 Inside a Product Term FPGA OR-Plane

### 2.2.2 Routing Resources

Routing is very important because it is needed to connect signals from one logic block to another. Routing is also needed to connect the logic blocks to off-chip signals via the I/O pads. There are primarily two types of routing in an FPGA:

1. Routing *between* logic blocks.
2. Routing *inside* logic blocks.

Routing elements can be found throughout the FPGA. Figure 2.7 illustrates the different routing elements.

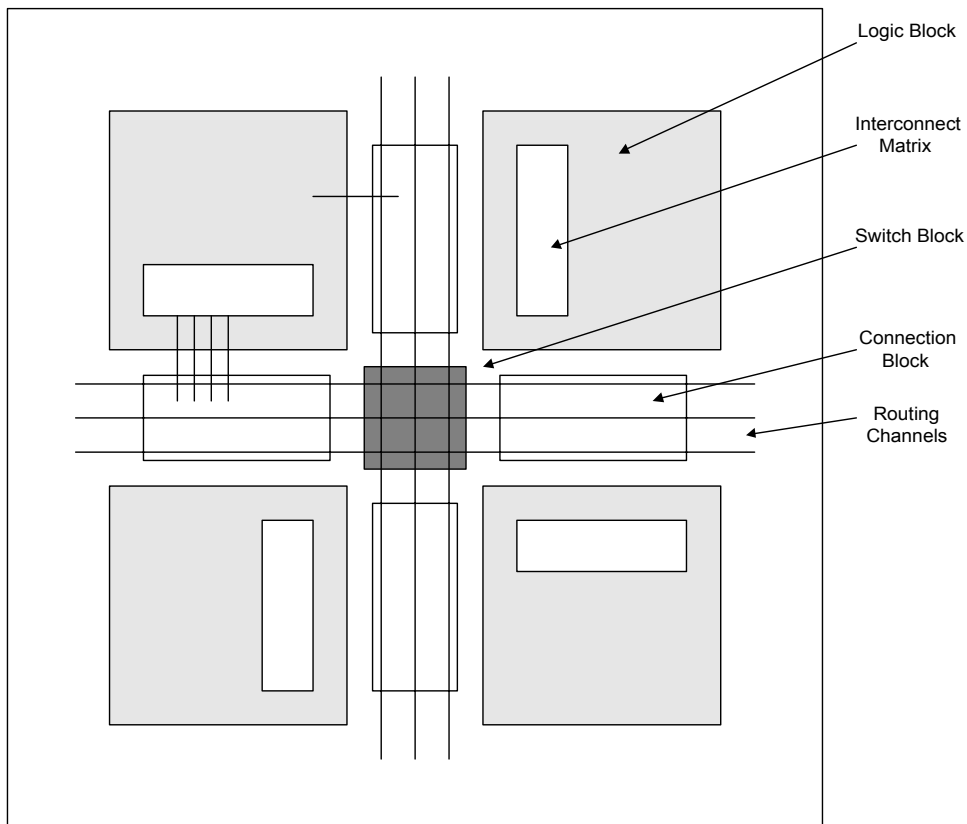


Figure 2.7 Routing in an FPGA

### *Routing Between Logic Blocks*

The routing between logic blocks is composed of several different elements:

1. Routing channels
2. Switch blocks
3. Connection blocks

The routing channels are fixed metal tracks that run the entire length of the chip, horizontally and vertically. There are a number of wires per routing channel; the exact

number depends on the specific architecture. Modern FPGAs also have a *segmented* routing architecture; each wire in the channel is not a single continuous wire from one side of the chip to the other. Instead, the wires are broken into smaller segments, with each segment usually spanning several logic block lengths.

The switch blocks are located at every intersection between the vertical and horizontal routing channels. The switch blocks define the connections between these channels. Several switch block topologies have been studied in detail in [7, 17, 22, 23, 26, 29].

Finally, the connection blocks connect the logic blocks to the routing channels. Both input and output signals connect to the routing channels using the connection blocks. These connections, and the connections between vertical and horizontal routing channels in the switch blocks, are made by programmable switches. Each programmable switch is a pass-transistor, with a user-programmable SRAM cell controlling the gate of the transistor, thus enabling or disabling the connection between source and drain. Figure 2.8 illustrates the structure of a programmable switch.

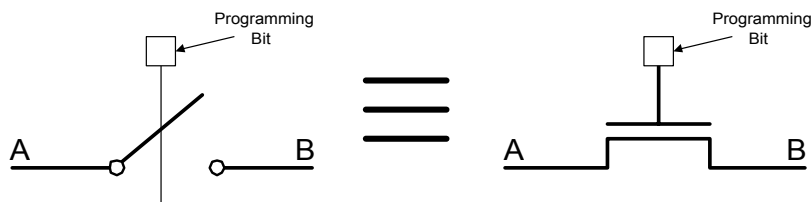


Figure 2.8 Structure of a Programmable Switch

### *Routing Inside Logic Blocks*

Routing inside each logic block is accomplished by a structure called the *interconnect matrix*.

The interconnect matrix connects the inputs to the logic block to the inputs of each logic

element in the logic block. The interconnect matrix also connects feedback paths from the outputs of each logic element to the inputs of the other logic elements. In some FPGAs, the interconnect matrix is fully connected (any of the inputs can be routed to any of the outputs), while in other FPGAs, the interconnect matrix is only partially connected. The architecture of the interconnect matrix is studied in great detail in [18, 19, 22].

### **2.2.3 Memory Resources**

With ever-increasing FPGA gate capacities, integration has reached a level where it is becoming feasible to implement entire systems on an FPGA. Therefore, access to on-chip memory is essential for two reasons. On-chip memory results in a faster circuit since slow, off-chip accesses to external memory are no longer required. On-chip memory also results in relaxed I/O constraints since dedicated memory I/O pins are no longer needed [27].

However, on-chip memory arrays can be problematic if they are not properly utilized, since they occupy a significant amount of chip area. In the APEX20k, on-chip memory arrays take up between 8% and 17% of the total chip area, depending on the specific device in the family [2]. Therefore, the key to having on-chip memory is to use these arrays in the most efficient manner possible.

A recent invention is product term mode memory arrays. These arrays behave like simple PALs and can implement logic as product terms. These arrays will be discussed further in Section 2.4.

### 2.3 FPGA Computer-Aided Design Flow

The computer-aided design (CAD) flow for FPGAs is similar to the ASIC CAD flow. The main difference is that the result of an ASIC CAD flow is a physical layout of transistors, while the result of an FPGA CAD flow is a stream of programming bits used to program the FPGA. The FPGA CAD flow is shown in Figure 2.9.

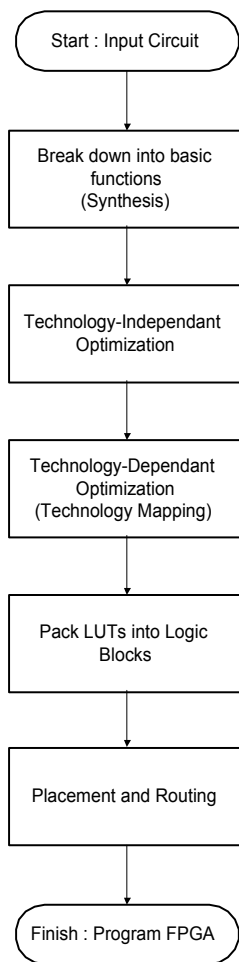


Figure 2.9 FPGA CAD Flow

In the first step of either CAD flow, the user's circuit is transformed from a high-level hardware description language (such as VHDL or Verilog) into Boolean logic equations or a netlist of basic functions. This process is called high-level synthesis [5, 25].

The next step in the CAD flow is called technology-independent optimization. In this step, the netlist or logic equations are optimized (ie. minimized) without knowing anything about the target technology. Many algorithms exist for performing this minimization [5].

After technology independent optimization, the netlist is then mapped to k-input lookup-tables (k-LUTs) in a process called technology mapping. Each gate or groups of gates in the netlist are combined into LUTs and flip-flops [9, 11]. When targeting FPGA architectures where each logic block is composed of many logic elements, the netlist of LUTs is then packed into clusters of LUTs (where each cluster corresponds to a logic block) [21]. The netlist of LUTs is then transformed into a netlist of logic blocks.

In the next phase of the CAD flow, the logic blocks are assigned physical locations on the FPGA. This process is called placement [4]. Placement is followed by the routing phase, in which the connections between logic blocks are made via the FPGA routing channels [4]. Placement and routing are closely related since if an appropriate routing cannot be found, an alternative placement must be found and routing attempted again.

## **2.4 Mapping Logic to Memory**

The problem of mapping logic to unused memory arrays in order to minimize the amount of wasted chip area has been studied extensively. In this section we will give a brief overview of two algorithms for targeting conventional mode memory arrays, SMAP and

EMB\_Pack. We will also discuss product term mode memory arrays and give an overview of Hybrid\_Map, an algorithm that targets product term memory arrays.

### 2.4.1 Terminology

Before we discuss algorithms for mapping logic to memory, we will first describe the terminology used. Much of this terminology has been adapted from [8].

The input to the algorithm is a network of 4-LUTs. A network is a directed graph (collection of directed edges and vertices) that represents the input circuit. Each vertex in the graph is called a *node*, and represents a 4-LUT. An edge is a connection between two nodes, and thus represents a connection between two 4-LUTs. Since each edge is directed, any given node in the network will have a number of incoming edges and a number of outgoing edges. The only exceptions are the inputs to the circuit (called the *primary input nodes*), which have no incoming edges, and the outputs from the circuit (called the *primary output nodes*), which have no outgoing edges. An example network is shown in Figure 2.10. Each node (vertex) in the network represents the output of a lookup-table (and thus can be any arbitrary logic function), and each edge represents a connection between two logic gates.

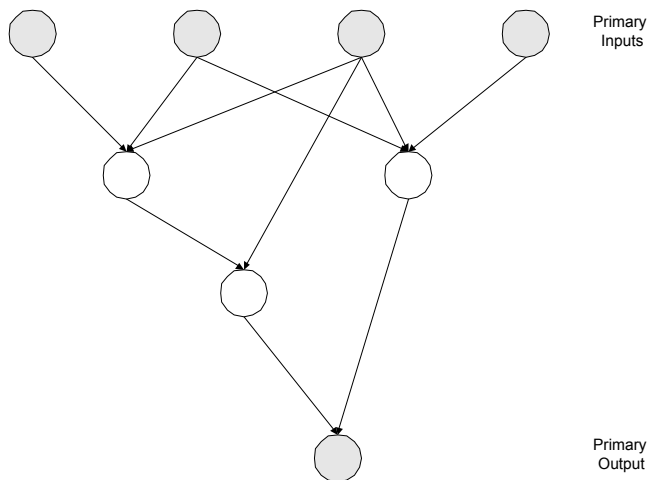


Figure 2.10 A Network of Nodes Representing a Circuit

For any given node  $n$  in a network, the *depth* of  $n$  is given by the maximum number of edges in the path from the primary inputs to  $n$ . Primary inputs have a depth of zero.

Excepting primary input and primary output nodes, every node  $n$  has a number of *fanins* and *fanouts*. The fanin nodes of  $n$  are the set of nodes that have an outgoing edge to  $n$ . The fanout nodes of  $n$  are the set of nodes to which there is an outgoing edge from  $n$ . The number of fanins to  $n$  is equal to the number of incoming edges to  $n$ . Similarly, the number of fanouts from  $n$  is equal to the number of outgoing edges from  $n$ . Finally, the *fanin network* of a node  $n$  is the network containing all nodes  $n'$  for which a path exists from  $n'$  to  $n$ . We say a subnetwork is *rooted* at a root node  $r$  if all nodes in the subnetwork are in the fanin network of  $r$ .

Two terms that are often used when describing mapping algorithms are the *fanout-free subnetworks* (FFS) and *maximum fanout-free subnetworks* (MFFS) of a node  $n$ . A fanout-free subnetwork of  $n$  is a subnetwork rooted at  $n$  such that for every node  $n'$  in the

subnetwork, the fanouts of  $n'$  are only to other nodes in the subnetwork. That means there are no outgoing edges from anywhere inside the subnetwork to anywhere outside the subnetwork, ie. “fanout-free”. A *maximum* fanout-free subnetwork of  $n$  is a FFS of  $n$  such that the FFS encloses the maximum number of nodes, ie. the largest FFS possible of  $n$ . This is illustrated in Figure 2.11.

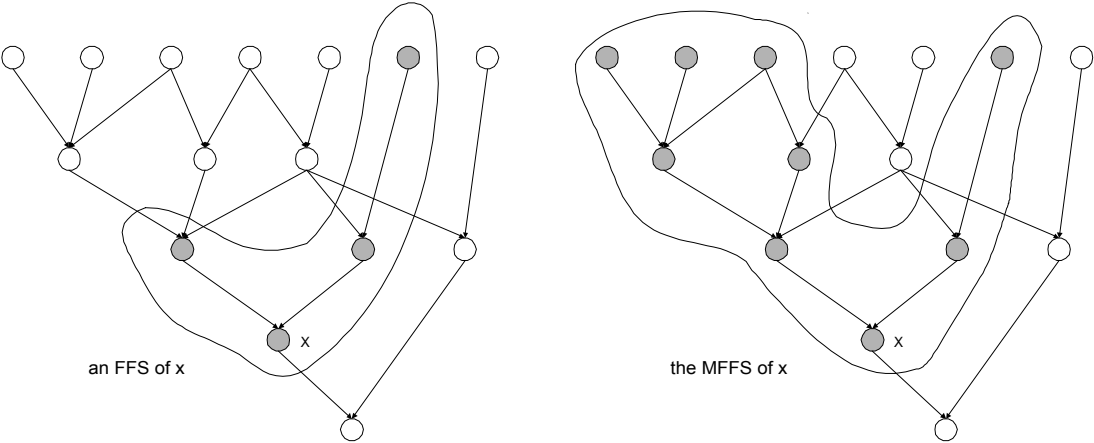


Figure 2.11 A FFS and the MFFS of the Node Labeled “x”

Finally, a concept that is used often in this thesis is *collapsing* a node into another node. Collapsing a node  $x$  into another node  $y$  means that  $y$  is replaced by another node  $y'$ . The logic function of  $y'$  is found by re-expressing  $y$  in terms of the fanins of  $x$ , such that  $y'$  is still logically equivalent to  $y$ . Note that the fanins of both  $x$  and  $y$  have become the fanins of  $y'$ . Also note that  $y$  is replaced, but  $x$  remains (although there is no longer a connection from  $x$  to  $y$ ). This is shown in Figure 2.12. In the figure, node “x” (with function  $ab$ ) is collapsed into node “y” (with function  $x+c$ ), producing a new node “y” (with function  $ab+c$ ).

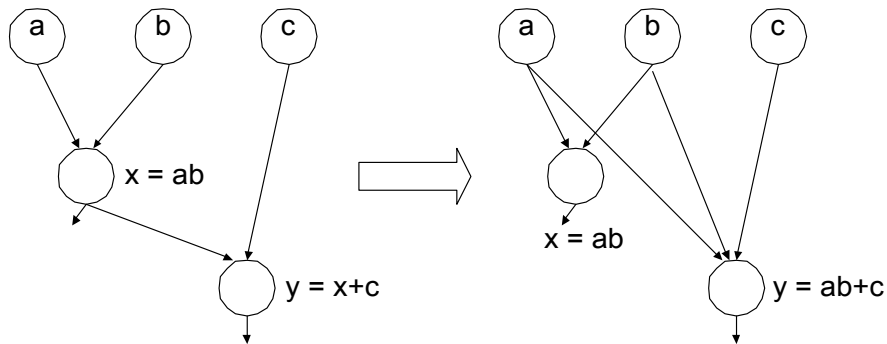


Figure 2.12 Collapsing Node x into Node y

### 2.4.2 Conventional Mode Memory Arrays

Algorithms for implementing logic in unused memory arrays in which the memory array is treated as a large multiple-input, multiple-output lookup-table are presented in [12, 27]. In this section, we give a brief overview of how these algorithms work.

#### *SMAP*

The algorithm proposed by Wilton [27] takes a network of 4-LUTs as input and attempts to pack nodes into unused memory arrays. It involves three steps:

1. Identifying a seed node,
2. Choosing the memory array input signals, and
3. Choosing the memory array output signals.

The goal of the algorithm is to pack as many nodes as possible into the memory; those nodes that could not be packed are implemented using LUTs. In the description of

SMAP below,  $d$  is the number of memory input signals, and  $w$  is the number of memory output signals.

In order to ensure an intelligent seed node choice, the algorithm described below is applied to every node in the network. The seed node that produces the most deleted nodes in the final solution is then chosen.

Given a starting seed node, the memory inputs are chosen. This is done by finding the maximum volume  $d$ -feasible cut of the seed's fanin network. This means that a cut is found such that the number of cut signals is less than or equal to  $d$ , while at the same time the number of nodes below the cut is maximized.

After the memory inputs are chosen, the outputs from the memory are chosen. To do this,  $w$  of the nodes under the cut are chosen such as to maximize the number of nodes that can be implemented in the memory array. For more details, refer to [27].

#### *EMB\_Pack*

The algorithm proposed by Cong and Xu [12] is similar to SMAP in that both algorithms attempt to pack a network of 4-LUTs into unused memory arrays. However, EMB\_Pack takes a different approach to the problem. Rather than applying the algorithm on every possible seed node as SMAP does, the set of seed nodes is chosen first. There are several phases to the algorithm:

1. Calculate seed set,
2. Compute maximum fanout-free subnetworks (MFFSs) of the seed set,

3. Trim infeasible MFFSs, and
4. Select subnetworks for implementation in the memory array.

Once the root set is calculated, the MFFSs of the root set nodes is found. Since no limit on the number of cut nodes has been imposed up to this point (only maximal subnetworks were found), some of the MFFSs that have been calculated are infeasible, because it has an input or output size larger than what is permitted by the memory. These infeasible MFFSs are further cut in order to make them feasible (thereby making them fanout-free subnetworks, since they are no longer maximal). The cut location is chosen such that the number of nodes below the cut is maximized but the FFS is still feasible.

After the feasible MFFSs and now-feasible FFSs have been calculated, they are now proper memory implementation candidates. For every unused memory array, one implementation is chosen from this set of candidates. The candidate is chosen such that the number of packed nodes is maximized while circuit delay is maintained.

### **2.4.3 Product Term Mode Memory Arrays**

A recent innovation in FPGA embedded memory arrays is product term-mode memory [14, 15]. In product term mode, a memory array can implement a basic PAL; thus implementation of a circuit as a sum-of-products is possible using these memory arrays.

The structure of a product term memory array is shown in Figure 2.13. It is composed of several parts:

1. Row and column decode circuitry,

2. RAM block (product term generator), and

3. Macrocell circuitry

Since a product term memory can also function as a regular RAM memory array, signals for both modes of operation exist and need to be multiplexed. A memory can operate in either one mode or the other, but not both simultaneously. For more information, refer to [14, 15].

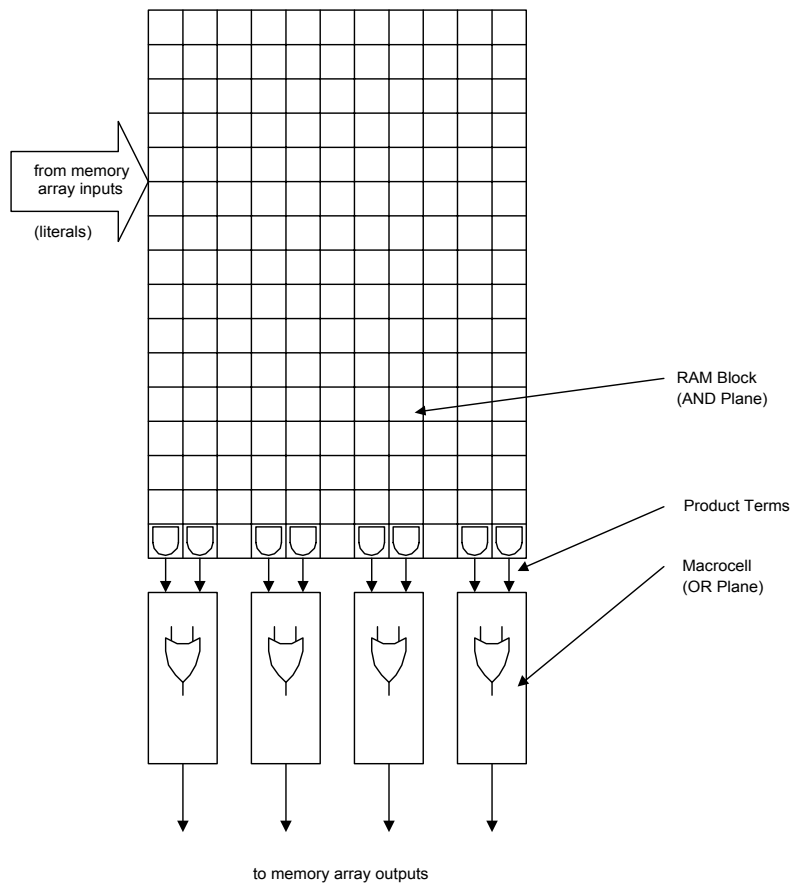


Figure 2.13 Structure of a Product Term Memory Array

### *Hybridmap*

This section describes an algorithm that also targets unused memory arrays. However, whereas the previous chapter described mapping to *conventional* memory arrays, this section is devoted to *product term* memory arrays.

We point out that while technology mapping algorithms that target PLA-like logic blocks exist [3, 10], those algorithms can not be applied to product term mode memory arrays because they do not take the specific memory architecture into account.

An algorithm proposed by Krishnamoorthy, Swaminathan, and Tessier, Hybridmap [16], targets product term mode memory arrays. It involves three steps:

1. Identification of feasible PLA subnetworks,
2. Product term estimation, and
3. Merging subnetworks.

The first step is to search for groups of nodes that have limited fanout which will drive the outputs of the product term memory array. These nodes form a root set and serve as a basis for determining which nodes can be implemented by the memory array. Once the root set is calculated, a search starting at the root set and proceeding backwards through the network (towards the inputs) is made. The backward search proceeds until there are no more nodes to include, or until the fanin to the subnetwork is larger than the number of inputs to the memory array. This step produces a number of subnetworks, all of which meet the input and output requirements of the product term memory array.

Input and output constraints are not sufficient to determine if a subnetwork can be implemented in a product term memory array. The other constraint is the number of product terms. The second step, product term estimation, is a set of heuristics that attempts to estimate, with reasonable accuracy, the number of product terms in each subnetwork in a fraction of the time needed by Espresso. The estimator is used to eliminate subnetworks with too many product terms. For more information on this step, see [16].

All of the subnetworks being considered at this point can be feasibly implemented in a product term memory. The last step is merging multiple, smaller subnetworks together so that the combined subnetworks can be implemented in a product term memory array without violating input, output, and product term constraints. A cost function is used to encourage the merging of wide fanin subnetworks while penalizing subnetwork combinations that have a relatively large number of product terms given the number of inputs and outputs.

## **2.5 Focus and Contributions of This Thesis**

This thesis will focus on the technology mapping aspect of the CAD flow. Technology mapping algorithms for various FPGA architectures exist [3, 9, 10, 11, 12, 16, 27]. However, as product term memory arrays are very recent, technology mapping algorithms that target these arrays are few. In Chapter 3 we propose a new algorithm for mapping a circuit to product term memories.

The product term architecture of a product term memory also requires study. Due to the newness of this feature, architectural enhancements may be in order and such enhancements should be evaluated. In Chapter 4 we propose several enhancements to existing macrocell architectures. In Chapter 5 we present results for these architectures and comment on their performance.

The contributions of this thesis are summarized as follows:

1. A new technology mapping algorithm that is more flexible than Hybridmap.
2. Experimental evaluation of various architectural alternatives for the product term memory arrays.

## *Chapter 3*

### TECHNOLOGY MAPPING TO PRODUCT-TERM MEMORY ARRAYS

In order to use the product term mode of the memory arrays efficiently, an intelligent technology mapping algorithm is required. Area on an FPGA is precious and since memory arrays occupy a large proportion of this area, it is important that the memory arrays be used in the most efficient way.

The algorithm presented in this chapter, pMapster, analyzes a circuit and extracts a subcircuit that, when implemented in a memory array, yields the most efficient implementation.

#### **3.1 Motivation**

On-chip memories consume large amounts of valuable chip area. If this memory is not efficiently utilized, that area is wasted. We need to use area efficiently in order to maximize the size of the circuit that can be implemented on the chip.

Mapping a circuit to memory means the memory is treated like a large multiple-input, multiple-output lookup-table. The technology mapping algorithms that accomplish this mapping exists and are well documented [12, 27]. However, with the advent of product term mode memory arrays, in which the memory behaves like a large PLA, technology mapping algorithms that target these arrays are few.

Very rarely will an entire circuit fit into a product term memory array. Often, a circuit will need to be partitioned into two parts; one part, to be implemented in the product term memory, and the other part, to be implemented in the logic blocks. Thus the mapping algorithm must do this intelligently, otherwise the memory will not be used efficiently.

One will see that just choosing any partition will not do. Clearly, some parts of the circuit are better suited for product term implementation than others. For example, wide fan-in functions such as state machine transition equations are often better for product term implementation. The mapping algorithm must therefore search the entire circuit and choose the best partitions.

### **3.2 Problem Description**

The general goal of technology mapping algorithms is to convert a netlist into one that uses a different technology (or the basic building blocks with which a circuit is implemented). For example, lookup-table technology mapping algorithms convert a netlist of gates into a netlist of lookup-tables. Technology mapping algorithms that target memory arrays such as those discussed in Section 2.4 above convert a netlist of lookup-tables into a netlist of lookup-tables with some of the lookup-tables replaced by one or more memory arrays.

Given a network of lookup-tables and a product term memory architecture as input, the goal is to determine a subnetwork of nodes suitable for implementation in a product term memory array. The goal is area optimization; we want the size of the subnetwork to be maximized while minimizing the size of the memory implementation. The algorithm

outputs the original network of lookup-tables; however, the lookup-tables that the algorithm decided to implement in the product term memory are removed. This is illustrated in Figure 3.1. In the figure, the shaded nodes on the left-hand side (and the combined logic function they implement) are replaced by the memory array on the right-hand side.

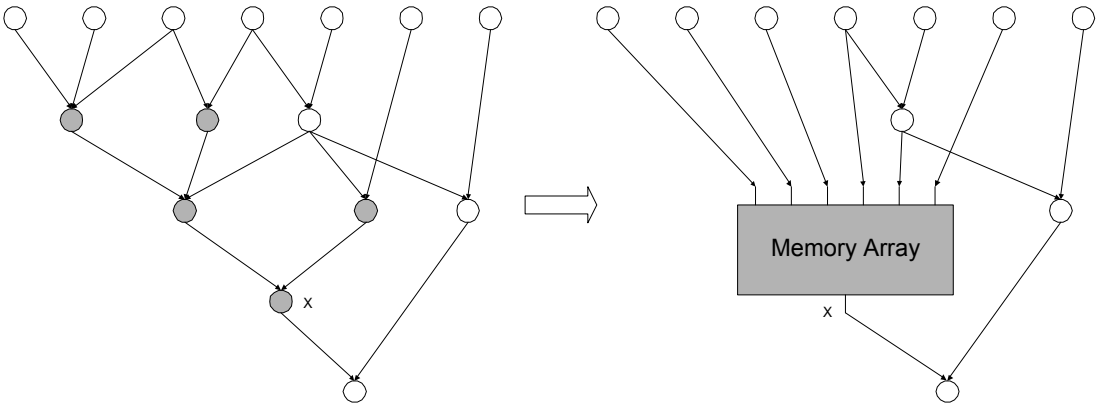


Figure 3.1 Mapping Part of a Network of LUTs to Memory

### 3.3 High-Level Description of pMapster

Our algorithm starts with a network of LUTs as input. Seeds are chosen and a mapping solution is computed for every seed. A comparison is made between the previous mapping solution and the new solution every time a new seed is selected and the corresponding mapping solution found. The better solution is retained. When all seeds have been exhausted, the final solution is the best solution out of all the computed solutions. A flowchart illustrating the process is shown in Figure 3.2.

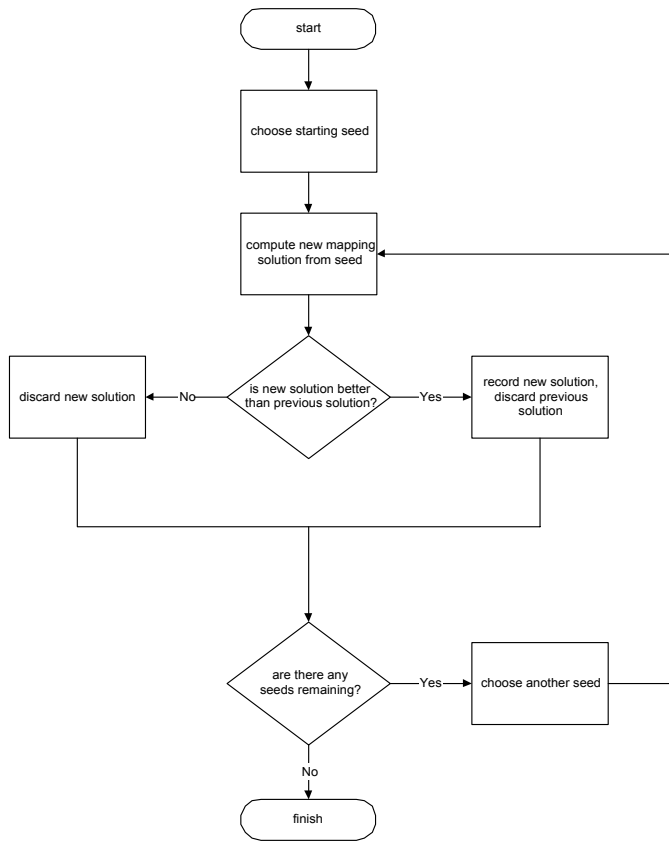


Figure 3.2 pMapster Program Flow

### 3.4 Detailed Description of pMapster

In this section we describe pMapster in greater detail. We elaborate on several important aspects of the algorithm:

1. Choosing a candidate for collapsing,
2. Collapsing the candidate node,
3. Ending the mapping loop, and
4. Choosing the next seed nodes.

The computation of a new mapping solution (given a seed node) is the core of our algorithm. Items 1 through 3 above are integral to this part of the algorithm, and are discussed in Sections 3.4.1 through 3.4.3. Section 3.4.4 discusses seed selection, which follows mapping solution computation in the flow shown in Figure 3.2.

Computing a mapping solution is essentially a loop; on every iteration of the loop, a fanin of the seed is selected and collapsed. If none of the fanins are suitable, then an end condition has been met and the loop is terminated. The calculated mapping solution is returned. The pseudocode that illustrates the process of finding a mapping solution is shown in Figures 3.3 and 3.4.

```
map_one_EAB():

  for each node in the network
  {
    n = numNodesFaninNet(node);
    if(n > numNodes(bestSolution))
    {
      networkCopy = copy(network);
      numDeleted = processSeed(node, networkCopy);

      if(numDeleted > bestSolution)
        bestSolution = numDeleted;

      record cut nodes;
      record deleted nodes;
    }
  }

  remove selected nodes from network;
```

Figure 3.3 Mapping to One Memory Array (pseudocode)

```

processSeed():

    while(!done)
    {
        bestScore = LARGE;
        for each fanin of seed
        {
            calculate newInputs;
            if(newInputs < MAX_INPUTS)
            calculate newPT;
            calculate newMC;

            if(newPT < MAX_PTERMS) &&(newMC < MAX_MACRO)
            {
                calculateScore(fanin);
                if(score < bestScore)
                    bestScore = score;

                record this fanin as candidate;
                atLeastOneFanin = 1;
            }
        }

        if(atLeastOneFanin)
        {
            collapse fanin into all the seeds;
            if(numFanout(fanin) > 1)
                add fanin to seeds;
        }

        calculate usedInputs, usedPT, usedMC;

        if(!atLeastOneFanin)
            done = 1;
    }

    record cut nodes;
    record deleted nodes;

```

Figure 3.4 Processing a Seed Node (pseudocode)

### 3.4.1 Choosing a Candidate for Collapsing

During each iteration of the algorithm, all fanin nodes of the seed are analyzed. The goal is to identify which of the seed's fanins is the most suitable for collapsing into the seed. We collapse fanins into the seed in order to accurately calculate the number of product terms after collapse. This is important because product term counts are used to signal the end of the iterations. We remind the reader that fanins that result in fewer product terms are favoured because it means fewer product term memory array resources are being used and therefore more logic can be packed into the memory array.

A *score* is calculated for each fanin; the fanin with the lowest score (which means the fanin results in the lowest number of product terms used) is collapsed into the seed.

#### *Calculating the Number of Resultant Product Terms*

In order to compute the number of product terms that will result when a fanin is collapsed into the seed, and since collapsing a node into another results in both nodes being changed, a "trial" collapse is computed. A copy of the fanin is collapsed into a copy of the seed and the number of resulting product terms is counted. Since the collapse operation is computationally intensive, it becomes necessary to *prune* the fanin nodes in order to reduce the number of collapse operations that are performed.

We prune by considering the number of literals and number of outputs that will result if a particular fanin is collapsed. Both quantities are easily calculated from information already associated with the fanin and seed nodes.

Product terms in a product term memory implementation are limited to a certain number of literals; for example, product terms in the APEX20k architecture are limited to 32 literals. The number of outputs is limited to the number of macrocells that are available; in the APEX20k architecture, the number of outputs is limited to 16 outputs [15].

If the number of literals or outputs computed for a potential candidate node exceeds the limit given by the given architecture, collapsing that node will result in a mapping solution that cannot be implemented in the given architecture. Therefore, the node should be ignored.

### **3.4.2 Collapsing a Candidate Fanin Node**

In this section we discuss the process of collapsing a candidate node. There are only two types of candidate fanin nodes:

1. Candidates that have one fanout, and
2. Candidates that have more than one fanout.

#### *Candidate Fanin with Single Fanout*

In many cases, the candidate fanin will have only one fanout; the output of the candidate node is connected to the input of only one other node in the network. As an example, the shaded nodes in Figure 3.5 are single fanout.

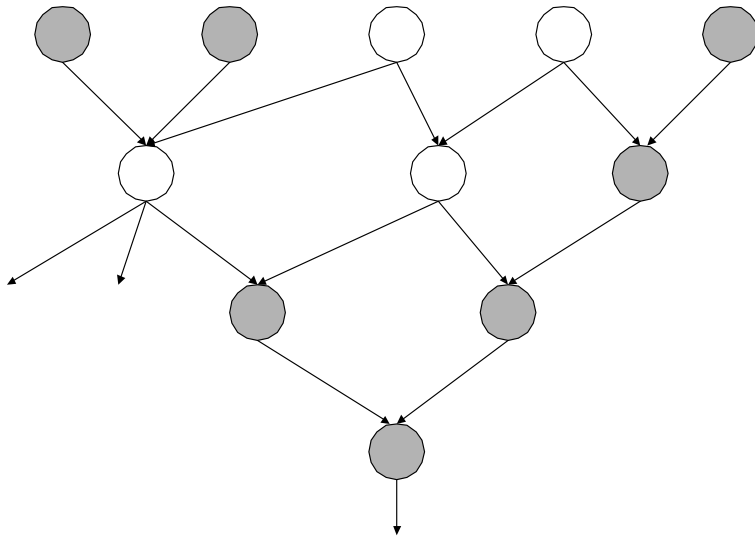


Figure 3.5 Shaded Nodes are Single Fanout

Dealing with collapsing a candidate with a single fanout is simple. Once the candidate is collapsed into the seed, the function implemented by the candidate is no longer needed (it has been integrated into the seed's function through collapsing). The candidate node can be safely removed from the circuit.

*Candidate Fanin with Multiple Fanouts*

Occasionally, the candidate node will have more than one fanout, meaning the output of the candidate node is connected to the inputs of more than one other nodes in the network. As an example, the shaded nodes in Figure 3.6 are multiple fanout.

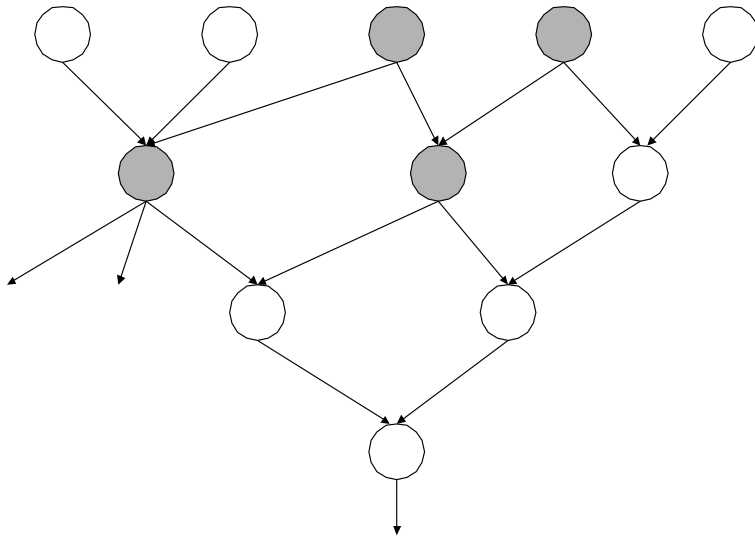


Figure 3.6 Shaded Nodes are Multiple Fanout

Collapsing a multiple fanout candidate node is not as simple as the single fanout case described above. If the candidate is multiple fanout (for example, if the candidate fans out to the seed node and another node), the candidate cannot be deleted after collapsing because it is still needed by the other nodes it fans out to. This poses a small problem: future candidates are possibly fanins of both the seed and the candidate node. When this happens, those future candidates are also multiple fanout. No future nodes will be removed if this is the case.

The only way future nodes can be deleted is if the future nodes are collapsed into both the seed and the candidate. When this happens and the future node has no other fanouts, its logic function has thus been integrated into the nodes it fans out to. Thus it can be safely removed from the network. Therefore, candidates with multiple fanout are made into secondary seed nodes when they are collapsed. All future collapses will then attempt to collapse into the primary seed as well as the secondary seeds. This process is illustrated in

Figure 3.7. In the first collapse operation, an extra output from node 2 is generated (it is made into a secondary seed node), as seen in the figure in the center. In the second collapse operation, node 3 is collapsed into both node 1 and node 2 (the primary and secondary seed nodes), and the result is shown in the figure on the right-hand side.

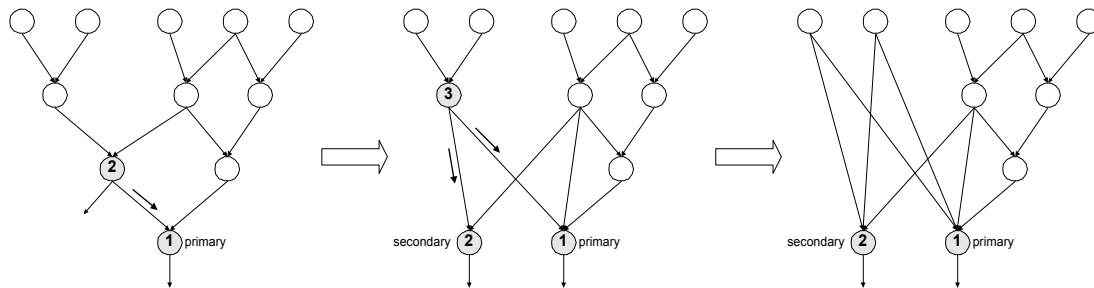


Figure 3.7 Creating a Secondary Seed

The outputs from the primary seed and also the secondary seeds make up what will ultimately become the output signals from the memory array.

### 3.4.3 Ending the Mapping Loop

The end conditions described in this section are important for signaling the end of the mapping loop. At the end of each iteration of the loop, the total number of product terms, literals (memory inputs), and macrocells (memory outputs) are computed. At the beginning of the next iteration, “trial” values for product term, literal, and macrocell usage are calculated for every fanin. If the “trial” values do not exceed limits (which are determined by the particular target architecture), the fanin is processed as a potential candidate node. However, no suitable fanins are found, the mapping loop is terminated and results are recorded.

#### **3.4.4 Choosing the Next Seed Node**

After the mapping loop has ended on the current seed node and a mapping solution has been found, it is compared with the best solution found so far and the better solution is retained. The next step is to choose the next seed node, if one exists. In order to find the best possible seed node (or rather the seed node that yields the best possible mapping solution), every node in the network is tried as a potential seed node. However, since very large run times will result for even medium-sized circuits, it becomes necessary to prune the network and to not consider seeds that will definitely yield a solution not better than the current best.

Pruning the network is a simple operation. The number of nodes in a potential seed's fanin network is counted. If the size of the fanin network is larger than the size of the current best solution, the potential seed node is tried as a seed node. Otherwise, the potential seed is ignored. The reason for this is that the largest possible solution (assuming no limits) that can be generated from any seed node is its fanin network. If the size of the fanin network of the potential seed is smaller than the size of the current best solution, the mapping solution from that potential seed will also be smaller. Therefore it is not necessary to try this potential seed.

#### **3.5 Implementation**

Our algorithm was implemented in approximately 3000 lines of C. It was integrated into the UC Berkeley logic synthesis package, SIS [25]. Integrating the algorithm into SIS allowed the algorithm to use the SIS data types and methods (such as `node_collapse`).

It was also integrated into an existing memory mapping algorithm, SMAP [27]. Product term mode memory arrays found on Altera APEX20k FPGAs can also function as a conventional memory, and thus logic can be mapped to them using SMAP. Thus, by integrating pMapster and SMAP, one algorithm can be used to target two architectures. In this way, logic can be mapped to both conventional and product term memories, which may result in better memory utilization (and thus efficiency) than if logic was mapped to only conventional memories or only product term memories. Experimental results that show the effect of mapping to both conventional and product term memory are presented in Chapter 5.

### **3.6 Summary**

Using on-chip memory arrays efficiently is very important. Implementing logic in unused arrays is one way that this can be accomplished. In this chapter we have presented an algorithm for mapping logic to product term mode memory arrays. We have described in detail several important aspects of our algorithm, pMapster. We have described how our algorithm chooses a candidate fanin on each iteration of the mapping loop, how collapsing a node is accomplished, how the mapping loop is terminated, and also how the next seed nodes are chosen. We have also briefly discussed some issues related to the implementation of the algorithm.

## *Chapter 4*

### PRODUCT-TERM MEMORY ARCHITECTURAL ISSUES

The ability of the algorithm presented in Chapter 3 to pack logic into product term memory arrays depends significantly on the architecture of the memory arrays themselves. The width and depth of the memory dictate how many functions can be packed into each array, as well as how large each function can be. In addition, the architecture of each macrocell within the memory has a significant effect on the amount of logic that can be packed into each array.

In this chapter, we focus on these architectural considerations. We start by reviewing the baseline architecture in Section 4.1, and by describing the role of the macrocells within each memory array. Section 4.2 then presents three proposals for enhancing the architecture of the macrocells. Section 4.3 focuses on the anticipated effects of changing the memory array width and depth on the ability of the memory to implement logic.

The architectural enhancements presented in this chapter are of little use if the CAD tool that maps logic to the memory arrays cannot take advantage of them. This is important – it is meaningless to propose architectural enhancements if they cannot be exploited by the CAD tools. Thus, in Section 4.4, we describe how the CAD tool from Chapter 3 can be modified to take advantage of the proposed architectural enhancements. An early version of some of the work presented in this chapter also appears in [20].

### 4.1 Baseline Architecture

The macrocells are an integral part of any product term device, including product term memory arrays. Macrocells implement the OR plane; without macrocells, sum-of-products implementations of logic are not possible. Macrocells also often contain additional specialized circuitry for implementing adders and counters. However, the “baseline” macrocell architecture that we will consider here is based on a generalization of the macrocell architecture found in FPGAs with product term functionality, such as the Altera APEX20k and Xilinx Virtex-E [2, 31, 32]. Figure 4.1 is a schematic of our baseline macrocell.

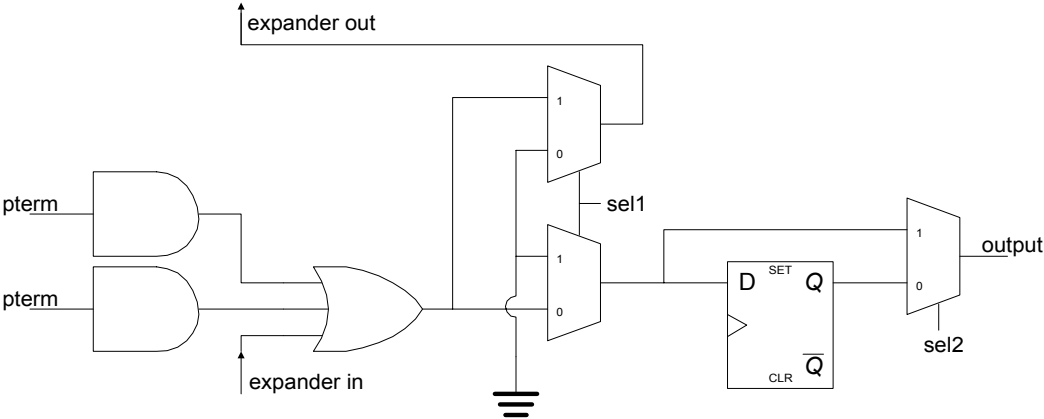


Figure 4.1 Baseline Macrocell Architecture

In the baseline macrocell architecture, the macrocell implements an OR of a fixed number (usually two) of product terms. If more product terms are needed to implement a function, other macrocells can be connected together to provide the additional product terms. This is done through a specialized inter-macrocell connection called a *parallel expander*. In our baseline architecture, the actual product term sum can be routed to the

macrocell output or to the next macrocell via the parallel expander connection, but not both simultaneously.

Finally, our baseline macrocell architecture also includes a storage element that is used when a registered macrocell output is desired. The storage element can be enabled or disabled by the user via a configuration bit.

## **4.2 Architectural Enhancement Proposals**

One of the major pitfalls of current macrocell architectures is the inability to share common product terms. Sharing product terms would lead to more efficient implementations of functions, thus allowing more functionality to be implemented in the memory array. We propose three ways of increasing this efficiency:

1. Macrocell granularity,
2. Sharing macrocell outputs, and
3. Multiple parallel expanders

### **4.2.1 Macrocell Granularity**

When implementing functions of a large number of product terms, many macrocells need to be cascaded (using the parallel expander) to implement the desired number of product terms. However, this incurs an incremental delay for every macrocell that is cascaded. One solution is to increase the number of product terms per macrocell (the *granularity* of the macrocell) so that functions with more product terms can be more efficiently implemented. The problem is that increasing the granularity also decreases the efficiency

with which smaller functions are implemented. Figure 4.2 illustrates a macrocell with a granularity of 4.

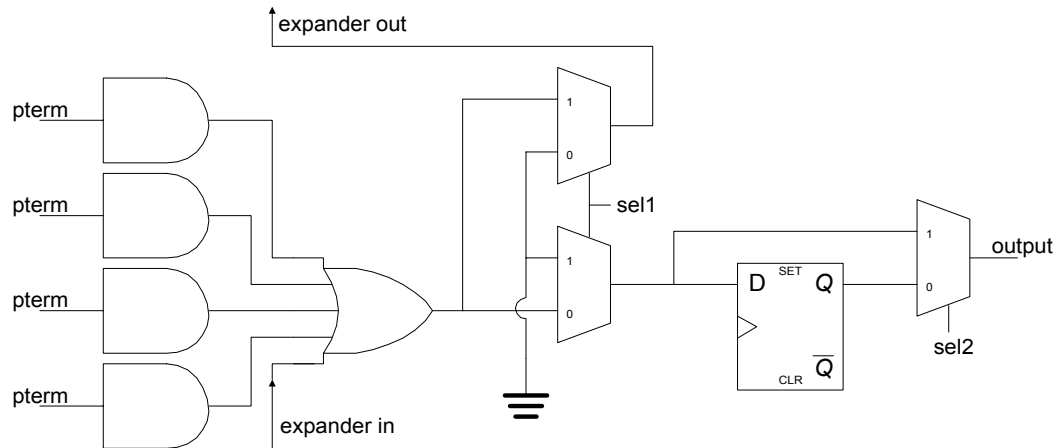


Figure 4.2 Macrocell with Granularity of 4

Varying macrocell granularity will have a slight effect on the area and speed of the memory array. In terms of area, increasing granularity results in logic being removed since there are fewer macrocells (recall each macrocell has one flip-flop storage element and also one set of extra logic for implementing special functions). However, the OR gate in each macrocell is larger; the 2-input OR gate is replaced by a 4 (or more) input OR gate.

Since larger OR gates are being used, the delay of each macrocell will increase slightly. However, since the parallel expander is used less often when implementing functions with many product terms, the overall delay will likely be less.

### 4.2.2 Sharing Macrocell Outputs

In the APEX20k architecture, each macrocell can direct the sum of the product terms to the macrocell output or to the next macrocell (via the parallel expander), but not both simultaneously. This may lead to product terms being used inefficiently.

Consider the following group of *incrementally similar* functions:

$$f = a + b$$

$$g = a + b + c + de$$

$$h = a + b + c + de + abc$$

Note that the product terms of  $f$  are included in  $g$ , and that the product terms of  $g$  are included in  $h$ . Thus we say  $f$ ,  $g$ , and  $h$  are incrementally similar.

In cases where the product term block must implement a number of functions that are incrementally similar, a macrocell that is capable of sharing its output with the next macrocell may be useful. Macrocells can then be chained together to efficiently implement these functions. Figure 4.3 shows how a macrocell with output sharing could be used to efficiently implement the incrementally similar functions  $f$ ,  $g$ , and  $h$ , above. Figure 4.4 is a schematic of a basic macrocell with the enhanced output circuitry.

$$x = a + b$$

$$y = a + b + c$$

$$z = a + b + c + de$$

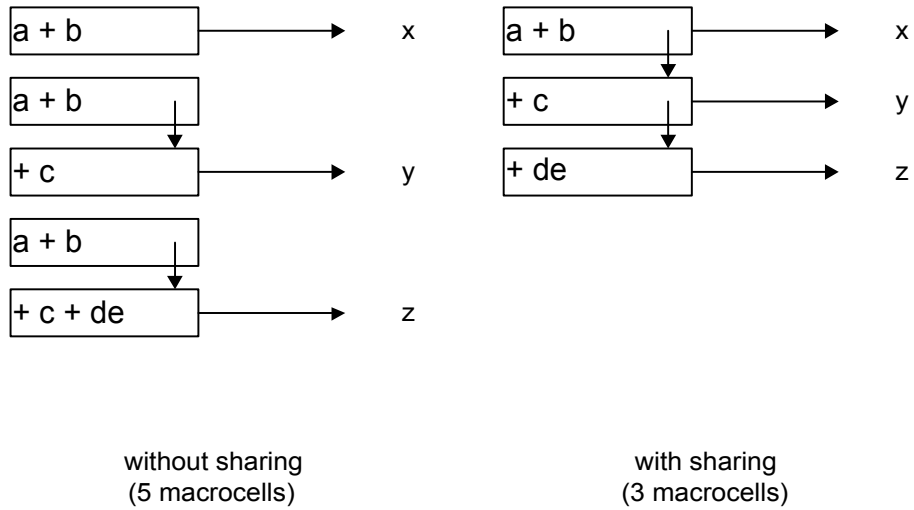


Figure 4.3 When Output Sharing is Useful

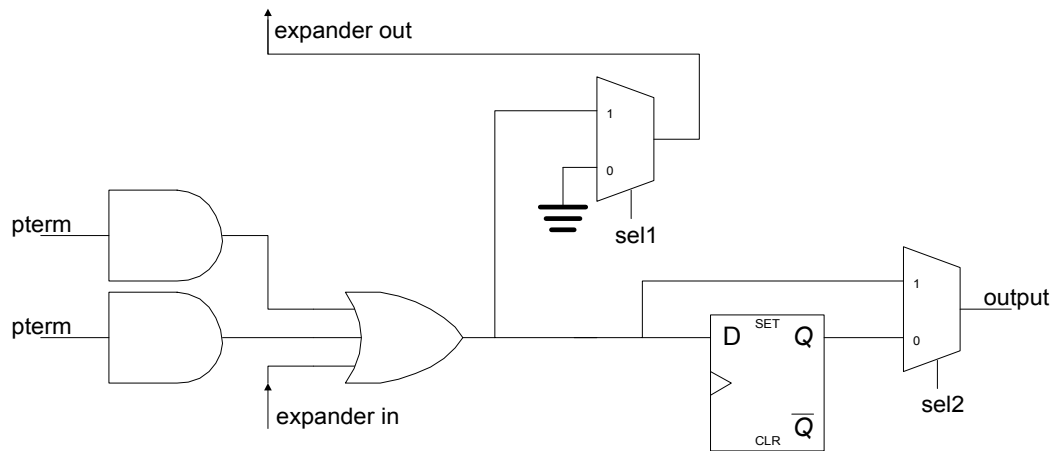


Figure 4.4 Basic Macrocell with Output Sharing

Finally, the area and delay impact of this enhancement is very small. In particular, the only additional area overhead is an extra SRAM configuration bit and pass transistor per

macrocell. Since no extra logic is being added to the macrocell itself, there is virtually no delay overhead.

### **4.2.3 Multiple Parallel Expanders**

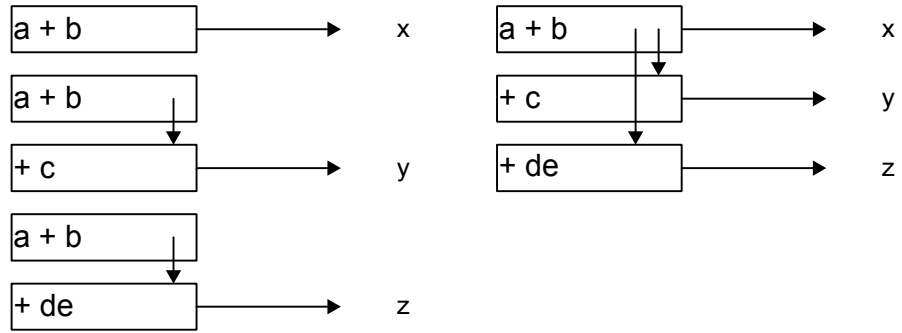
Since each macrocell has only one connection to the adjacent macrocell, product terms generated in one macrocell can only be used by at most two macrocells (the generating macrocell and the macrocell adjacent to it, assuming macrocell output sharing exists). In certain cases, this could lead to an inefficient use of macrocells since certain product terms may be repeated. Increasing the number of parallel expanders could increase product term usage efficiency, since product term repetition can be reduced.

In cases where the product term block must implement a number of functions which are similar, although not necessarily incrementally similar as in the previous example, having a macrocell capable of sharing with the next 2, 4, 6, or more macrocells may be useful. A macrocell can then be used to generate the common product terms (in a set of functions) and share them with several other macrocells. Figure 4.5 shows how this could be useful. Figure 4.6 shows a basic macrocell with enhanced parallel expanders.

$$x = a + b$$

$$y = a + b + c$$

$$z = a + b + de$$



without sharing  
(5 macrocells)

with sharing, 2 parallel expanders  
(3 macrocells)

Figure 4.5 When Multiple Parallel Expanders is Useful

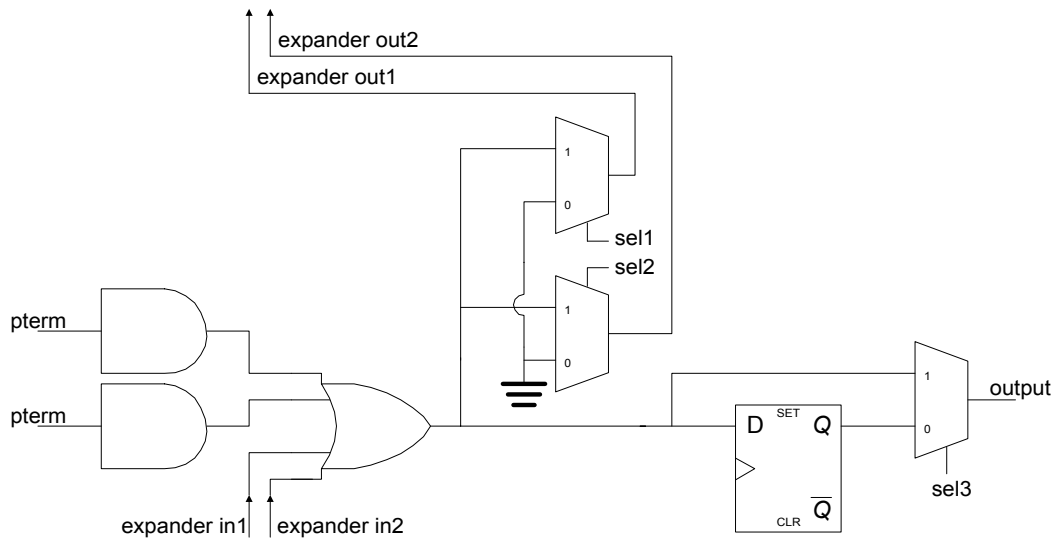


Figure 4.6 Basic Macrocell with Multiple Parallel Expanders

Again, the area and delay implications of this enhancement are small. The area penalty is one additional multiplexer per additional parallel expander, per macrocell. For an architecture with four parallel expanders, this amounts to three extra multiplexers per macrocell. However, there is virtually no delay penalty associated with this enhancement, since there is no extra logic has been added that would increase the amount of delay through the macrocell.

### **4.3 Memory Configuration Architectural Issues**

In the previous section, we described three architectural enhancements, and in doing so, created an architectural “space”. An additional “dimension” in this space is the architecture of the RAM block, which is also the product term generator (AND plane). Specifically, the width and depth of the memory may have a significant effect on its ability to implement logic.

The RAM block generates the product terms; the transistors in each memory cell in the RAM are configured such that each column line implements a wired-AND function of each cell in that column. Thus, each column line can implement a different product term simply by programming the appropriate transistors along that column with the appropriate values.

The dimensions of the memory affect how many product terms can be implemented by the memory array, as well as how many literals can be used for each product term. In particular, each pair of rows of the memory corresponds to a separate literal, with one row each being used for the true and complemented forms of the literal.

Figure 4.7 is an illustration of a memory array and shows how memory parameters affect the product term architecture. The parameters illustrated are width  $w$  and depth  $d$  of the memory.

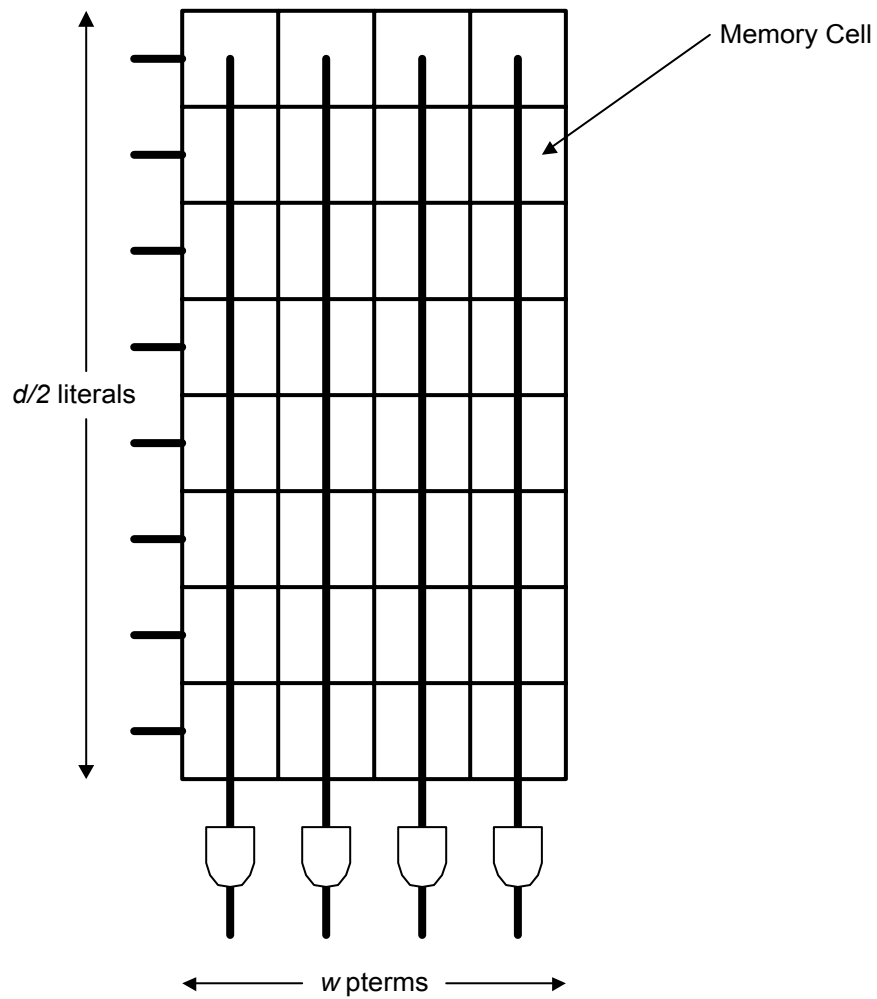


Figure 4.7 Effect of Memory Parameters

### **4.3.1 Width of the Memory**

The width of the memory is important because it determines how many column lines are present in the memory block. This in turn determines the number of product terms the memory block is capable of generating.

### **4.3.2 Depth of the Memory**

The depth of the memory is also important because it determines how many row lines are present in the memory block. This in turn determines the maximum number of literals that are available to be combined into product terms.

## **4.4 Changes to pMapster to Support Architectural Enhancements**

The previous sections described several architectural enhancements and options for the product term memory arrays. This, however, is only half the story. Without CAD tools that support these architectural features, the features will go unused. Very few FPGA users map their circuits to the FPGA architectural components by hand, and this is unlikely to change.

The CAD tool described in Chapter 3 is flexible enough to target arrays with any numbers of rows and columns. However, it is not able to efficiently support the macrocell options described in Section 4.3. In this section, we show how the CAD tool from Chapter 3 can be extended to support these options. In particular, we look at three extensions:

1. We show how a term to the cost function can be modified to reflect the granularity value of the memory array,

2. We show how the algorithm can be modified to support the proposed macrocell output sharing enhancement, and
3. We show how the algorithm can be further modified to target macrocells with multiple parallel expanders.

As in Chapter 3, the experimental evaluation of these algorithmic enhancements will be delayed until Chapter 5.

#### 4.4.1 Macrocell Granularity

Section 4.2.2 defined macrocell granularity as the number of product terms per macrocell. For the Altera APEX20k, this granularity is equal to two; there are two product terms per macrocell. Currently, the granularity value is hard-coded into the algorithm presented in Chapter 3. In this section, we show how our algorithm can be modified to target an architecture with an arbitrary granularity value.

In the APEX20k architecture, the number of macrocells  $M$  needed to implement the  $P$  product terms in a seed is given by:

$$M = \left\lceil \frac{P}{2} \right\rceil \tag{4.1}$$

where  $\lceil x \rceil$  denotes the ceiling function of  $x$ .

As stated above, in the APEX20k architecture, each macrocell can implement two product terms. Recall that if additional product terms are needed, the parallel expander is used and additional macrocells are used to implement the extra product terms. This

means that if one were to implement a function with only one or two product terms, only one macrocell would be required. To implement a function with three or four product terms, two macrocells would be required. Equation 4.1 can thus be derived intuitively.

The next step is to generalize Equation 4.1 for an arbitrary value of granularity, which we will denote by  $G$ . Intuitively, the number of macrocells  $M$  needed to implement the  $P$  product terms in a seed, assuming a granularity of  $G$ , is given by:

$$M = \left\lceil \frac{P}{G} \right\rceil \quad (4.2)$$

Equation 4.2 can be used to calculate the number of macrocells needed to implement the function of a seed for any granularity value. In the algorithm described in Chapter 3, and in particular the mapping process illustrated in Figure 3.7, Equation 4.1 is used to calculate the newMC and usedMC terms in Figure 3.7. Implementing support in our algorithm for different granularity values is done by replacing Equation 4.1 with Equation 4.2.

#### 4.4.2 Sharing Macrocell Outputs

In the baseline algorithm described in Chapter 3, sharing product terms between macrocells is not supported. Thus, in the baseline algorithm, to implement a set of seeds, each seed is implemented independently, since each seed is not “aware” of the other seeds. In particular, the number of macrocells needed to implement a set of seeds is the sum of the number of macrocells needed to implement each seed separately, as shown in Equation 4.3.

$$M_{total} = \sum_{\substack{\text{each} \\ \text{seed}}} M_{seed} \quad (4.3)$$

When sharing is taken into account, the number of macrocells needed to implement a set of seeds is no longer a straightforward sum of the individual seeds. The algorithm must be enhanced with “awareness” that one particular seed can share its product terms with another seed.

This “awareness” is not a trivial problem. In order to implement awareness, our solution is to start with an arrangement of seed nodes, and rearrange the seed nodes many times until the best fit is found.

From the previous iteration of the mapping loop, the seeds start in an already good order. If the current iteration added another seed, then this new seed is the only seed that needs to be put in the order. Otherwise if no seed was added during the current iteration, the order of the seeds should not change from the previous iteration.

The “fitness” of a particular arrangement of seeds refers to how well product terms from one seed are being shared by another seed. Each macrocell can share its product term sum with at most the next macrocell via the single parallel expander. The next section describes the case where multiple expanders exist.

The fitness of an arrangement of seeds is calculated by analyzing seed  $n$  to see if it can share its product terms with the product terms of seed  $n+1$ . This is done by comparing each product term in seed  $n$  with the product terms in seed  $n+1$ . If every product term in seed  $n$  matches a product term in seed  $n+1$ , then seed  $n$  is marked as being a “product

term subset” of seed  $n+1$ . Seeds that are subsets of other seeds are not counted when the number of product terms and macrocells are computed.

Figure 4.8 illustrates the impact of “fitness” on the number of macrocells used by a set of seeds.

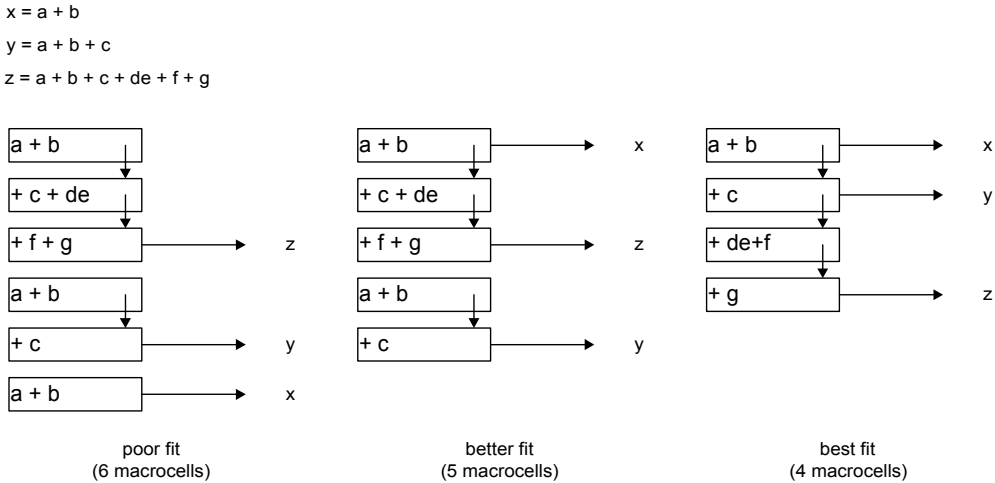


Figure 4.8 Impact of “Fitness” on Number of Macrocells Used

### 4.4.3 Multiple Parallel Expanders

In the base architecture, the parallel expander allows large sum-of-product equations to be implemented. Since each macrocell is only capable of computing a sum of two product terms, the parallel expander links adjacent macrocells. For example:

$f = A + B + C + D + E$  is implemented as  $f = \{ \{ \{ A + B \} + C + D \} + E \}$ , where each  $\{ \}$  represents the function implemented by each macrocell in the link. This is represented graphically in Figure 4.9.

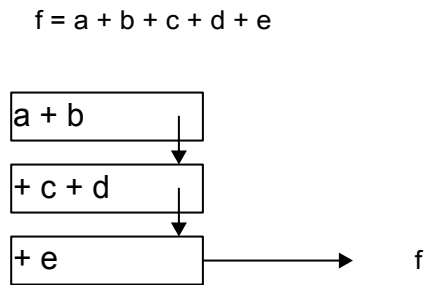


Figure 4.9 Using the Parallel Expander to Join Macrocells

The multiple parallel expander problem is an extension of the output sharing problem described above. Having multiple parallel expanders allows the product terms of one seed to be shared by potentially many other seeds. The algorithm thus requires a more complex awareness than in Section 4.4.2.

The enhanced awareness routine starts with an arrangement of seeds and analyzes each seed in turn. For each seed, the algorithm also analyzes the next few seeds in the arrangement. The depth of this search is limited; after a certain point past the current seed, no more seeds are analyzed. This is described below.

When the function of a particular seed is implemented, it will use a certain number of macrocells. These macrocells, if they do not share product terms with any seeds, will limit previous seeds' ability to share with later seeds. This is shown in Figure 4.10. Thus, as seeds are analyzed for shared product terms, a parameter called the *macrocell distance* is calculated. The macrocell distance represents the number of macrocells between the macrocell implementing the current seed and the seed that is being analyzed for product terms. For each seed that is analyzed, the macrocell distance is incremented by the

number of macrocells required to implement that seed. When the macrocell distance exceeds the depth of the parallel expanders, there is no longer a connection from the current seed to the seed being analyzed. Any shared product terms that exist cannot thus be shared. Thus, the search stops and the process repeats with the next seed in the arrangement.

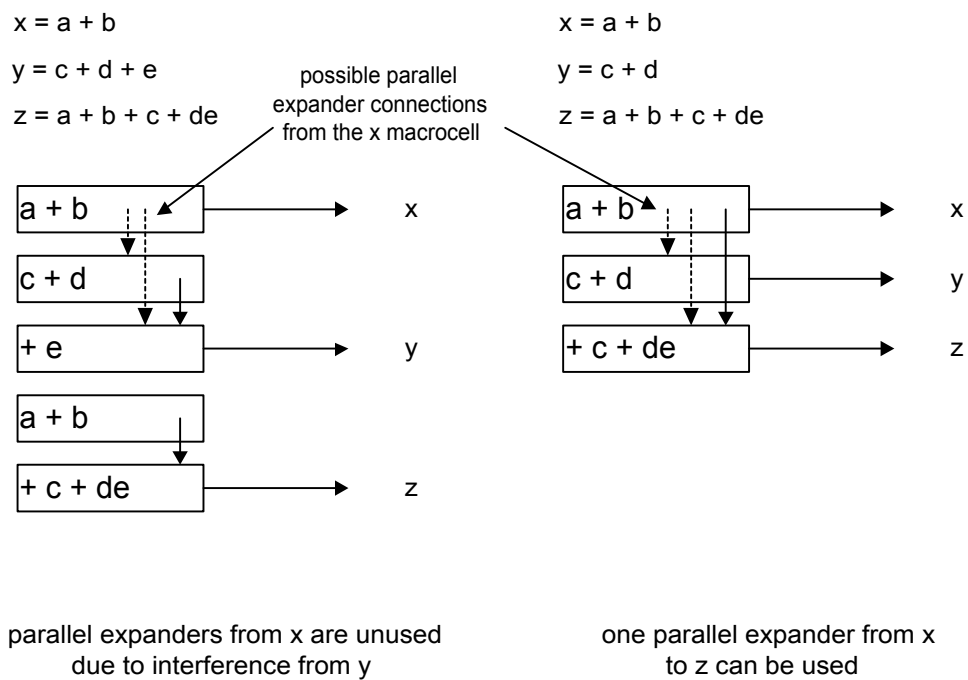


Figure 4.10 How Macrocells Limit a Seed's Ability to Share

As with the output sharing problem, each seed is analyzed. The seeds after the current seed are also analyzed and the number of shared product terms is calculated. Product terms that are shared are not counted in the totals. The number of macrocells required is also calculated.

The pseudocode that implements the sharing part of the pMapster algorithm is shown in Figure 4.11.

```
calc_fitness():

  for i=0 to array_size(seeds)
  {
    cur = seeds[i]; j = 1; distance = 0;

    while(distance < numOfExpanders) && ((i + j) < array_size(seeds))
    {
      dest = seeds[i + j];
      if(cur is shared with dest)
        dest->pterm = numCube(dest)-numCube(cur);
      else
        dest->pterm = numCube(dest);

      distance += numMacrocells(dest->pterm);
      j++;
    }
  }

  for l=0 to array_size(seeds)
  {
    cur = seeds[l];
    ptermsUsedThisArrangement += dest->pterm;
    macrocellsUsedThisArrangement += numMacrocells(dest->pterm);
  }

  return(ptermUsedThisArrangement, macrocellsUsedThisArrangement);
```

Figure 4.11 Calculating Product Terms and Macrocells (pseudocode)

## 4.5 Summary

In this chapter we have presented three architectural enhancements for increasing the efficiency of the macrocells in a product term memory array. Increasing macrocell granularity increases the speed of larger sum-of-products functions, since fewer macrocells are required to implement the same function. Sharing the output of the macrocell with

the next macrocell can allow similar functions to share common product terms, and adding more parallel expanders can allow many functions that have a few common product terms to share them. We have also shown how our CAD tool was modified to support these architectural enhancements.

## *Chapter 5*

### RESULTS

In this chapter, we present our experimental methodology and results. In particular, we do the following:

1. We first validate the algorithm from Chapter 3 by comparing it to a previously published algorithm, Hybridmap [16]. These results are presented in Section 5.2.
2. We then use our algorithm to investigate the ability of product-term memories to implement logic. As explained in Chapter 2, all commercial FPGAs now contain memory arrays; these memory arrays can be configured as ROMs and used to implement logic. The Altera APEX20k architecture, however, has memory arrays that can be configured as either a conventional memory or a product-term array. In Section 5.3, we investigate whether the extra circuitry needed to support these dual-mode memories is worth it. That is, we compare the ability of a conventional memory array to that of a dual-mode array when implementing logic.
3. Finally, in Section 5.4, we quantify the gains obtained using the architectural enhancements and algorithmic enhancements discussed in Chapter 4.

Before these results are presented, however, this chapter will start with a description of the experimental methodology employed.

## 5.1 Experimental Setup

To evaluate the proposed macrocell architectures, we used 15 medium and large benchmark circuits obtained from the Microelectronics Corporation of North Carolina (MCNC). Before the benchmarks were used, they had to be optimized and mapped to lookup-tables. Logic optimization was performed on all the benchmarks using SIS optimization scripts *script.rugged* and *script.algebraic* [25]. The benchmarks were then mapped to 4-input lookup-tables using FlowMap [9].

## 5.2 pMapster Versus Hybridmap

In order to see how much of an improvement our algorithm represents over the previous algorithm, Hybridmap [16], and also to validate our algorithm, we obtained results from Hybridmap and compared them to pMapster. These results are shown in Table 5.1.

Circuit Name	pMapster (post-Quartus)			Hybridmap		
	Initial LUTs	Final LUTs	Difference	Initial LUTs	Final LUTs	Difference
9sym	132	0	132	108	0	108
alu2	205	25	180	140	0	140
alu4	1406	1221	185	1042	888	154
apex2	1506	1369	137	1062	955	107
apex4	1210	1045	165	1011	910	101
apex6	307	213	94	278	120	158
apex7	100	42	58	65	0	65
C880	189	69	120	119	26	93
cps	705	495	210	530	381	149
duke2	234	72	162	166	32	134
pair	692	580	112	545	416	129
<b>Geo. Average</b>			133.9			118.0

Table 5.1 Comparing pMapster and HybridMap

Identical architectures were assumed for each algorithm; 16 macrocells of 2 product terms each, with 32 literals available to the product term generator. Mapping was done over 10 memory arrays, and the numbers reported above represent the number of lookup-tables present in the circuit before mapping to memory and the number of lookup-tables that remain after all memories have been used.

A direct comparison between the algorithms is difficult since the initial number of lookup-tables in the benchmark circuits (before applying the algorithm) do not match. This is because the CAD flow for pMapster and Hybridmap are different. Hybridmap is applied to the circuit before any technology mapping has occurred. Technology mapping in the Hybridmap CAD flow occurs after Hybridmap has extracted the portions of the circuit destined for product term implementation. Altera Quartus is then used to map the circuits to both product term memory and lookup-tables at the same time.

In the pMapster CAD flow, FlowMap [9] is first used to map the circuit to lookup-tables. The mapped circuit is then input to pMapster, which removes the portion of the circuit that it has implemented in product term memory.

In order to try to make the comparison fairer, we exported our benchmark files (both before and after mapping using pMapster) to VHDL. The exported files were then read, compiled, synthesized and fitted to an APEX20k device by Quartus. A direct comparison still cannot be made, since initial lookup-tables counts still do not match, but we wanted to rule out that the differences were not due to optimizations made by Quartus. The fairest comparison we can make with the above data is the number of LUTs each algorithm has removed. The data in Table 5.1 show that pMapster performs better than Hybridmap for most circuits. On average, pMapster is able to remove 13.4% more lookup-tables from the circuit than Hybridmap.

### 5.3 Product Term Memory Versus Conventional Memory

Product term mode memories are more complex than conventional memories. They require specialized circuitry to implement the product term array and macrocells. In the Altera APEX20K, product term is an additional mode; when this mode is turned off, the product term memory operates as a conventional memory [15]. Thus, having product term circuitry is an additional area and delay penalty on the underlying conventional memory array. In this section, we seek to determine if this penalty is worth it.

In order to determine whether having product term mode is worth the area and delay penalties, we attempted to map our benchmarks to memory under three scenarios:

1. Conventional memories available only.
2. Product term memories available only.
3. Both conventional and product term memory are available.

Scenarios 1 and 2 are valuable since these results will directly quantify any benefits of having product term mode memories. Scenario 3 is a valid scenario considering that each memory array can work in either conventional or product term mode; there is no requirement that *all* memory arrays must be in one mode or the other. It is likely that for any memory array, the best solution that the algorithm finds will be different depending on whether the memory is in conventional or product term mode. If both modes are available, the best solution for each mode can be compared and the better of the two selected. An extra degree of flexibility is thus available.

Mapping logic to conventional memory is done using the SMAP algorithm [27], whereas product term mapping is done using the pMapster algorithm. In scenarios 1 and 2, SMAP and pMapster are used, respectively. In scenario 3, SMAP and pMapster solutions are computed for each memory array, and for each memory array, the better solution is chosen.

In this experiment, the architecture was set to the original APEX20k architecture, with none of the enhancements described in Chapter 4. Each memory array has 16 macrocells each capable of computing a sum of 2 product terms, where 32 literals are available to the product term generator. The above product term architecture implies a memory array of 32 columns and 64 rows, or 2048 bits.

Figure 5.1 and Table 5.2 show the results of this experiment, in graphical and tabular forms, respectively. The data in Table 5.2 are geometric averages of the number of logic blocks packed for each number of available memory arrays.

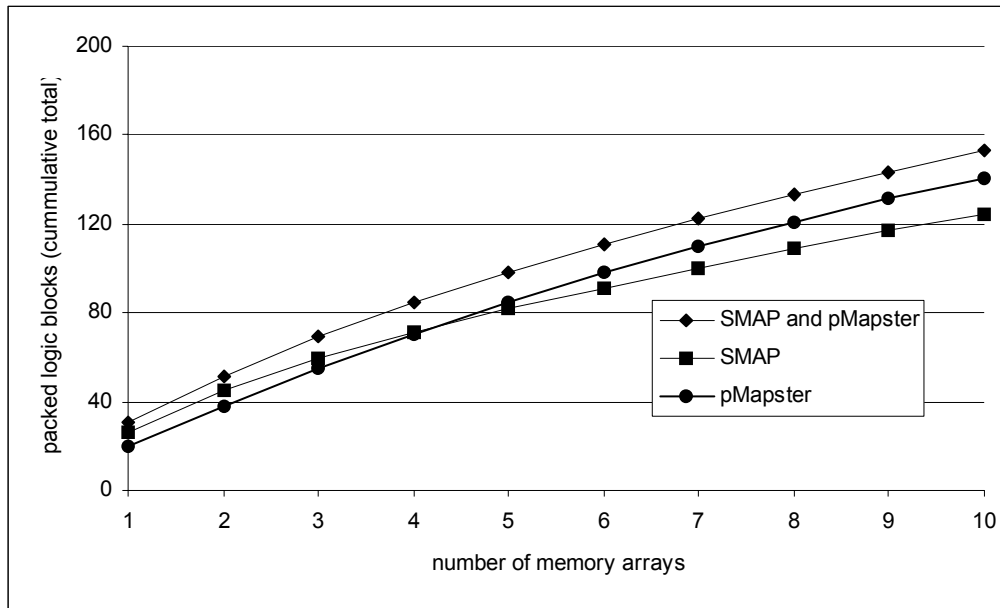


Figure 5.1 Product Term Vs. Conventional Memory (graph)

Circuit Name	pMapster			SMAP			pMapster + SMAP		
	1 array	5 arrays	10 arrays	1 array	5 arrays	10 arrays	1 array	5 arrays	10 arrays
9sym	36	132	144	144	144	144	144	144	144
alu2	27	115	172	76	151	173	76	170	191
alu4	31	153	296	107	241	340	107	261	408
apex2	30	137	259	21	91	175	30	137	259
apex6	17	69	108	16	62	101	17	78	123
apex7	11	41	59	15	48	67	15	52	71
bigkey	12	46	86	19	51	86	19	51	91
C5315	12	54	101	13	60	104	13	61	114
C7552	27	104	171	16	75	128	27	104	178
C880	14	65	107	8	37	68	14	65	107
cps	34	133	249	65	161	243	65	183	302
des	18	81	155	14	68	133	18	81	155
duke2	27	120	184	20	74	109	27	120	184

pair	15	68	127	15	60	107	15	74	137
rd84	25	111	141	157	157	157	157	157	157
s5378	17	68	122	19	73	112	19	82	137
tseng	14	62	105	10	47	82	14	62	109
<b>Geo. Average</b>	20.0	84.9	140.2	26.5	81.6	124.7	30.4	98.6	153.1

Table 5.2 Product Term Vs. Conventional Memory (table)

The data in Figure 5.1 and Table 5.2 show that having product term mode available increases the amount of logic that can be packed into the memory array. In particular, the amount of logic increases the most when both product term and conventional memory modes are available for use.

The reason for this is that when there are only a few arrays, SMAP tends to work better than pMapster. However, as the number of memory arrays is increased, SMAP's efficiency drops off rapidly, mainly because all the "best" nodes have already been chosen and removed, leaving only "bad" seed nodes. pMapster's efficiency is less affected by the number of memory arrays and therefore exhibits more consistent behaviour across all memory arrays. As a result, when both algorithms are combined and work together, the SMAP solutions are chosen for the first few arrays; after that, the pMapster solutions are chosen. The result is a packing efficiency that is better than either SMAP or pMapster working alone.

We can see from the data in Figure 5.1 and Table 5.2 that having product term mode results in a significant increase in the amount of logic that can be packed into a memory

array. On average, 22.8% more logic can be packed when mapped to both product term and conventional memory, and 12.4% more logic when mapped to product term memory only. Since a product term memory array is approximately 16.7% larger than a conventional memory array (one extra transistor per memory cell [15]), and product term memory arrays are able to implement 22.8% more logic than conventional memory arrays, we conclude that the extra area requirement is worthwhile.

#### **5.4 Effect of Macrocell Granularity**

We investigated the effect of macrocell granularity on packing efficiency. Current APEX20k macrocells have a granularity of two product terms. In cases where a large function must be implemented, each extra macrocell that must be used to implement this function (macrocells are joined by the parallel expander) incurs an incremental delay penalty. Macrocells with larger granularity are able to implement larger functions without as much added delay. However, increasing granularity also decreases the efficiency of implementing smaller functions. In this section, we seek to determine the optimal value for macrocell granularity.

In this experiment, the total number of product terms is fixed at 32 product terms (as in original APEX20k architecture). Thus, when the granularity is varied, the number of macrocells varies. For example, a granularity of 2 means there are 16 macrocells, and a granularity of 4 means there are 8 macrocells, and so on. The number of literals available to the product term generator is 32.

Figure 5.2 and Table 5.3 shows the result of this experiment, in graphical and tabular forms, respectively. The data in Table 5.3 are geometric averages of the number of logic blocks packed for each number of available memory arrays.

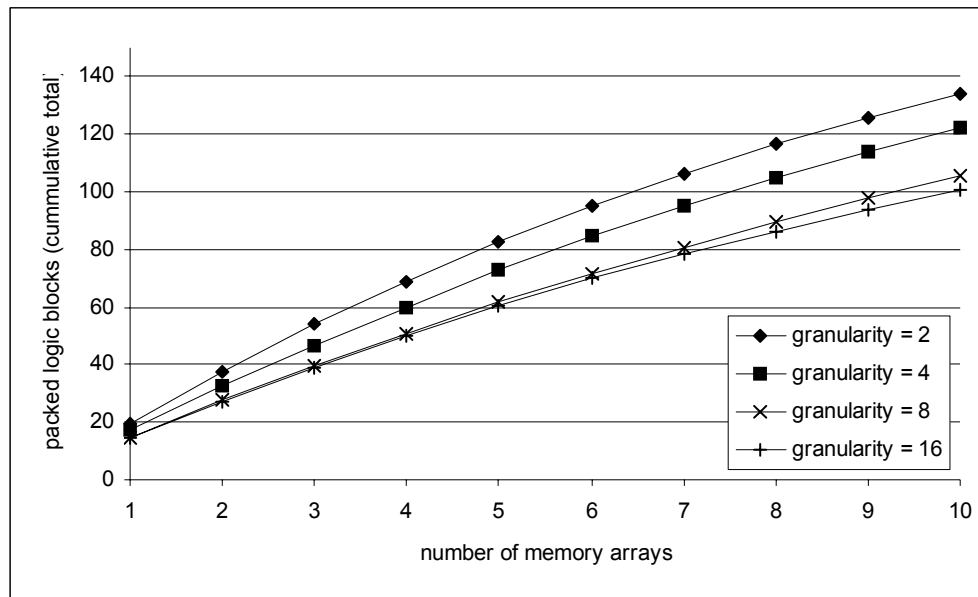


Figure 5.2 Effect of Macrocell Granularity (graph)

Granularity Value	Number of Memory Arrays									
	1	2	3	4	5	6	7	8	9	10
2 (baseline)	19.6	37.4	53.9	68.6	82.5	94.9	106.2	116.4	125.5	133.7
4	17.2	32.6	46.5	59.5	72.7	84.5	95.1	104.6	114.1	122.6
8	14.3	27.7	39.5	50.8	61.5	71.6	80.7	89.7	97.8	105.5
16	14.4	27.2	38.9	50.1	60.6	70.2	78.7	86.4	93.6	100.5

Table 5.3 Effect of Macrocell Granularity (table)

As shown in the above data, as the granularity of the macrocell increases (number of product terms per macrocell increases), the amount of logic that can be packed into the memory decreases. It also shows that the optimal macrocell granularity is two product terms per macrocell.

This is an interesting outcome because it shows that even when granularity is increased from two product terms per macrocell to four, there is a significant decrease in the amount of packed logic (10.8% over ten arrays). The reason for this is that increasing granularity also decreases the efficiency with which small functions are implemented. As a result, the extra product terms that are available in the macrocell are wasted; this limits the amount of logic that can be implemented. Equivalently, when granularity is increased from 2 to 4, the number of outputs available is decreased from 16 to 8. This is also a limiting factor in the amount of logic that can be packed into the memory.

### **5.5 Effect of Sharing**

Finally, we investigated the ability of the parallel expanders to increase the amount of logic that can be packed into each memory array. In the current macrocell architecture, macrocells only have one parallel expander. This means that for any given macrocell, it can be connected to at most one other macrocell. Usually, this connection is used to implement large functions; if a macrocell cannot completely implement a function, several macrocells can be chained together via the parallel expander to implement the large function. However, the parallel expander, with the modifications described in Chapter 4, can allow product terms to be shared between macrocells, thereby increasing the efficiency of the product term architecture. However, it is unclear how well these “sharing

macrocells” benefit from the sharing architecture versus conventional macrocells. Also, if there is a clear benefit from having a sharing architecture, it is unclear what the optimal architecture should look like (for example, how many parallel expanders?).

In this experiment, we assumed the original APEX20k architecture (16 macrocells of 2 product terms each, with 32 literals available to the product term generator). However, the number of parallel expanders was varied from 1 to 8. For the case where there was only 1 parallel expander, output sharing (described in Chapter 4) was both unavailable (original APEX20k architecture) and available (sharing architecture). For all cases where there was more than one parallel expander, the sharing architecture also includes output sharing. Table 5.4 shows the results of this experiment.

Parallel Expanders	Number of Memory Arrays									
	1	2	3	4	5	6	7	8	9	10
1, no sharing (baseline)	19.6	37.4	53.9	68.6	82.5	94.9	106.2	116.4	125.5	133.7
1, sharing	19.7	37.6	53.7	68.4	82.6	95.6	106.6	116.7	125.7	134.1
2, sharing	19.7	37.6	53.7	68.4	82.7	95.8	107.0	117.1	126.3	134.6
4, sharing	19.9	37.7	53.8	68.6	82.8	95.8	106.9	117.2	126.3	134.8
8, sharing	19.9	37.7	53.8	68.6	82.8	95.8	107.1	117.3	126.5	134.8

Table 5.4 Effect of Macrocell Sharing

As the table shows, sharing has a much less effect on the amount of packed logic than expected. Over 10 arrays, the total amount of packed logic increases by less than 1%, even when there are eight parallel expanders.

The most significant reason why this is so is that sharing opportunities do not occur frequently in circuits. A sharing architecture is not useful if there is nothing in the original circuit that can be shared. Closer examination of the output from pMapster reveals that this is true for at least some of the benchmark circuits.

### **5.6 Effect of Algorithm on Experimental Results**

We have used our CAD algorithm in order to obtain experimental results. Clearly, this means that the conclusions we drew from our experimental results depend significantly on the CAD algorithm. This is acceptable because architecture and CAD tool go hand-in-hand; you cannot study an architecture without a CAD tool, and you cannot evaluate a CAD tool without a target architecture.

In order to evaluate an architecture, and draw conclusions on how well it performs with respect to other architectures, we must first know how it performs by itself. This means we must be able to map a circuit to it and extract a metric or measurement from the mapped circuit; for this, we need a CAD tool.

### **5.7 Limitations on Experimental Results**

Although we have tried to present work that is complete, we accept that there are limitations that need to be considered. The first limitation is that these results apply to a

specific architecture. While we believe these results can apply to a broader perspective, it is uncertain that they do. Without obtaining results for a wider range of architectures, it is difficult to know how to interpret these results in a more general context.

The second limitation is that we have not considered the impact of the algorithm on delay. While implementing logic in a product term mode memory array results in a significant decrease in area, we have not studied how the critical path is changed or affected. If the delay penalty is significant, the gains made in terms of area may not be worthwhile.

## **5.8 Summary**

In this chapter we have presented our experimental results. We have compared the performance of our algorithm to that of an existing product term memory mapping algorithm, Hybridmap; we have shown that our algorithm yields a better mapping than Hybridmap. We have also shown that having product term mode memory arrays are worthwhile, as they can pack up to 22.8% more logic than conventional memory arrays alone, when utilized efficiently. Finally, we have presented results that show that the optimal macrocell granularity is two product terms per macrocell, and that macrocells do not benefit very much (less than 1% performance gain) from the ability to share product terms.

## *Chapter 6*

### CONCLUSIONS

In this thesis we have examined FPGA product term memories and how they can be utilized more efficiently. First, we examined the product term memory and proposed an algorithm for mapping logic to the product term architecture. We also examined macrocell architectures that allow the sharing of product terms in order to increase efficiency.

Conventional memory arrays can be used to implement logic; there are several existing algorithms that do this. However, for implementing wide functions, it is better to implement them as product terms. This is the motivation behind having product term-mode memory arrays. We show that implementing logic as product terms increases the amount of packed logic by 12.4% over implementing logic in conventional memory arrays. We also show that even greater gains can be realized when both conventional and product term memory arrays are available for implementing logic; in this case, the amount of packed logic is increased by 22.8% over implementing logic in conventional memory arrays alone.

Being able to pack 12.4% of the circuit into product term mode memory is significant since it implies that a circuit can be made 12.4% larger by simply using the memory to implement the extra logic. Alternatively, since the same mapped circuit now takes 12.4%

less area than the original, unmapped circuit, a smaller FPGA can now be used to implement the circuit. Each successive FPGA in a given device family can usually implement approximately 10% more logic than its predecessor. What this means is that by simply mapping some of the user's logic to product term mode memory, a circuit that once could only fit into a given FPGA can now fit into that FPGA's predecessor. This translates into a savings in cost for the end-user, since a smaller, less expensive FPGA can be used instead of the larger, more expensive one.

In the APEX20K product term architecture, only a limited number of product terms can be generated. In order to maximize efficiency, the generated product terms should ideally be all unique. It would be wasteful if the product term generator had to generate the same product term more than once if it can be avoided. Therefore, we propose a number of architectural enhancements that allow the macrocells to share product terms:

1. Macrocell Granularity
2. Product Term Sharing
  - a. Via Output Sharing
  - b. Via Multiple Parallel Expanders

We show that increasing macrocell granularity has an adverse effect on the amount of logic that can be packed into the memory. Increasing granularity to 4 product terms per macrocell results in a 10% decrease in packed logic. Increasing granularity even further to 8 or 16 product terms per macrocell results in even less logic being packed into the memory array.

We also show that sharing does not increase the amount of packed logic by a significant amount. When the number of parallel expanders is increased to 8 per macrocell, and with output sharing, the amount of logic that could be packed into each memory array only increased by 1%. However, the enhanced parallel expanders and output sharing circuitry add approximately 20% to the area of each memory array. Thus, it appears that having the ability to share product terms is not worthwhile. However, these results are specific to the architecture we investigated.

A technology mapping tool was created to map logic to product terms. The pMapster algorithm takes as input a network of 4-LUTs (which represent a circuit) and outputs the same circuit less the 4-LUTs that are to be implemented in the product term memories. A modified version of the algorithm was also created to allow the sharing of product terms between macrocells.

### **6.1 Future Work**

The exact area and delay information of the macrocell architecture enhancements described in Chapter 4 is still not known. It is important to quantify the exact area and delay penalties imposed by the enhanced macrocell architectures, because without this information, we cannot have exact timing information nor can we say what the exact area penalties will be.

The product term architecture also has a large impact on the amount of logic that can be packed. In particular, varying the number of product terms, product term literals, number of inputs, number of outputs, and sharing architectures, can have a significant effect on

logic packed per bit of memory. Therefore an architecture study is needed to determine the optimum parameters.

While it is clear that the methods for sharing product terms we have described in this thesis do not benefit in terms of improving macrocell efficiency, perhaps other methods of improving the flexibility of the architecture will lead to an increase in efficiency. One suggestion that arose during the course of the research was the possibility of hybrid product term and conventional architectures within the same memory array operating simultaneously. Currently, each memory array can be in one mode or the other, but not in both modes simultaneously. We have seen how combining SMAP with pMapster gives better results than either SMAP or pMapster alone. Extending this concept to the hardware may give even better results.

Our algorithm is currently optimized for area; the solutions it generates are based on maximum number of 4-LUTs removed from the network. Although area is an important optimization goal, delay is also very important, especially if 4-LUTs on the critical path are being packed into the memory array. It would be an important next step to analyze the current algorithm and determine what changes should be made so that unnecessary delay is not added to the packed logic.

## **6.2 Contributions of This Work**

The contributions of this work are summarized as follows:

- i. A novel macrocell architecture that increases macrocell efficiency by allowing product terms to be shared between memory array outputs.

- ii. We quantified the effect of various macrocell granularities on the amount of logic that can be packed into a product term mode memory.
- iii. We quantified the impact of an architecture that supports the sharing of product terms on the amount of logic that can be packed into a product term mode memory.
- iv. A novel algorithm for mapping logic to product term mode memory that can also take product term sharing into account.
- v. We compared the ability of a product term mode memory to a conventional memory in implementing logic.

Most importantly, this research has furthered our understanding of how product term memories can be used efficiently to implement logic.

## REFERENCES

1. Altera Corp., APEX II Programmable Logic Device Family Datasheet, ver. 1.1, May 2001.
2. Altera Corp., APEX20k Programmable Logic Device Family Datasheet, ver. 4.0, August 2001.
3. Anderson, J. and Brown, S., "Technology Mapping for Large Complex PLDs," in Proceedings of the 35<sup>th</sup> Design Automation Conference (DAC'98), 1998.
4. Betz, V. and Rose, J., "VPR: A New Packing, Placement, and Routing Tool for FPGA Research," in Proceedings of the 7<sup>th</sup> International Workshop on Field-Programmable Logic and Applications, 1997, pp.213-222.
5. Brayton, R. K., et al., "Multilevel Logic Synthesis," in Proceedings of the IEEE, vol. 78, no. 2, February 1990, pp. 264-300.
6. Brown, S., Francis, R., Rose, J., and Vranesic, Z., "Field Programmable Gate Arrays," Kluwer Academic Publishers, 1992.
7. Chang, Y. W., Wong, D., and Wong, C., "Universal Switch Modules for FPGA Design," in ACM Transactions on Design Automation of Electronic Systems, vol. 1, January 1996, pp. 80-101.

8. Cong, J. and Ding, Y., "Combinational Logic Synthesis for LUT Based Field Programmable Gate Arrays," in ACM Transactions on Design Automation of Electronic Systems, vol. 1, no. 2, April 1996, pp. 145-204.
9. Cong, J. and Ding, Y., "Flowmap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs," in IEEE Transactions on CAD of Integrated Circuits and Systems, vol. 13, January 1994, pp. 1-12.
10. Cong, J., Huang, H., and Yuan, X., "Technology Mapping for k/m-Macrocell Based FPGAs," in Proceedings of ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Monterey, CA, February 2000, pp. 51-59.
11. Cong, J., and Hwang, Y., "Simultaneous Depth and Area Minimization in LUT-Based FPGA Mapping," in Proceedings of ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Monterey, CA, February 1995, pp. 68-74.
12. Cong, J. and Xu, S., "Technology Mapping for FPGAs with Embedded Memory Blocks," in Proceedings of ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Monterey, CA, February 1998, pp. 179-187.

13. Cypress Semiconductor Corp., Delta39k ISR CPLD Family Datasheet, Document No. 38-03039, July 2001.
14. Heile, F., “Programmable Logic Array Device with Random Access Memory Configurable as Product Terms,” United States Patent #6020759, February 2000.
15. Heile, F. and Leaver, A., “Hybrid Product Term and LUT Based Architectures Using Embedded Memory Blocks,” in Proceedings of ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Monterey, CA, February 1999, pp. 13-16.
16. Krishnamoorthy, S., Swaminathan, S., and Tessier, R., “Area-Optimized Technology Mapping for Hybrid FPGAs,” in Proceedings of the 10<sup>th</sup> International Workshop on Field-Programmable Logic and Applications, August 2000.
17. Lemieux, G. G., and Brown, S. D., “A Detailed Router for Allocating Wire Segments in Field Programmable Gate Arrays,” in Proceedings of the ACM Physical Design Workshop, April 1993.
18. Lemieux, G. G., Leventis, P., and Lewis, D., “Generating Highly-Routable Sparse Crossbars for PLDs,” in Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Monterey, CA, February 2000, pp. 155-164.

19. Lemieux, G. G. and Lewis, D., "Using Sparse Crossbars Within LUT Clusters," in Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Monterey, CA, February 2001, pp. 59-68.
20. Lin, E. and Wilton, S. J. E., "Macrocell Architectures for Product Term Embedded Memory Arrays," to be presented at the 11<sup>th</sup> International Conference on Field-Programmable Logic and Applications, Belfast, UK, August 2001.
21. Marquardt, A., Betz, V., and Rose, J., "Using Clustered-Based Logic Blocks and Timing Driven Packing to Improve FPGA Speed and Density," in Proceedings of ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Monterey, CA, February 1999, pp. 37-46.
22. Masud, M.I., "FPGA Routing Architectures: A Novel Switch Block and Depopulated Interconnect Matrix Architectures," M.A.Sc Thesis, University of British Columbia, December 1999.
23. Masud, M. I., and Wilton, S. J. E., "A New Switch Block for Segmented FPGAs," in Proceedings of the 9<sup>th</sup> International Workshop on Field-Programmable Logic and Applications, Glasgow, UK, September 1999, pp. 274-281.
24. Roelandts, W., "FPGAs Enter the Marketplace," presented at 38<sup>th</sup> Annual Design Automation Conference (DAC'2001), Las Vegas, NV, June 2001.

25. Sentovich, E. M., "SIS: A System for Sequential Circuit Analysis," Technical Report No. UCB/ERL M92/41, University of California at Berkeley, Berkeley, 1992.
26. Wilton, S. J. E., "Architecture and Algorithms for Field Programmable Gate Arrays with Embedded Memory," PhD Thesis, University of Toronto, 1997.
27. Wilton, S. J. E., "SMAP: Heterogeneous Technology Mapping for Area Reduction in FPGAs with Embedded Memory Arrays," in Proceedings of ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Monterey, CA, February 1998.
28. Xilinx Inc., CoolRunner XPLA3 CPLD Datasheet, ver. 1.1, March 2000.
29. Xilinx Inc., Data Book, 1999.
30. Xilinx Inc., Spartan-II 2.5V FPGA Family Datasheet, ver. 2.2, March 2001.
31. Xilinx Inc., Virtex-E 1.8V Extended Memory Field-Programmable Gate Arrays Datasheet, ver. 1.4, April 2001.
32. Xilinx Inc., Virtex-E 1.8V Field-Programmable Gate Arrays Datasheet, ver. 1.3, February 2000.
33. Xilinx Inc., XC4000E and XC4000X Field-Programmable Gate Arrays Datasheet, ver. 1.6, May 1999.



I'm a man I'm a guy  
just a dude and I'm  
hungry

And I'm movin real  
fast cuz I'm goin first  
class for a steak at  
taco bell

Jumpin ford  
Roasted taco  
Steaks like king of  
beats

Cuz I don't got a  
horse to ride there

Grilled steak taco at  
taco bell!

- Serrated bread knife
- All purpose shef  
knife
- Slicer
- All purpose utility  
knife
- 3 inch paring knife

8 quart stock pot  
3 quart sauce pan  
12 inch fry pan  
wok