INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

TECHNOLOGY MAPPING ALGORITHMS OF SEQUENTIAL CIRCUITS USING LUT-BASED FPGAS

BY

QUAN XU

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York

1996

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

May 1, 1996
Date

Chair of Examining Committee

May 1, 1996
Date

Executive Officer

Professor Theodore Brown

Professor Seyed-Ali S. Ghozati

Professor Charles Giardina

Professor Bogong Su

Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

TECHNOLOGY MAPPING ALGORITHMS OF SEQUENTIAL CIRCUITS USING LUT-BASED FPGAS

by

QUAN XU

ADVISER: Professor Stanley Habib

This thesis explores the optimization of technology mapping of sequential circuits using look-up table based field programmable gate arrays (LUT-based FPGAs). The thesis first gives a brief survey of programmable logic devices and field programmable gate arrays, including their architectures in addition to previous research performed on technology mapping of look-up table based FPGAs. The thesis defines terminology used for sequential circuits. A concept of *bell* structure is defined and illustrated in the thesis. A sequential circuit can be represented as a bell network. Each bell consists of a *hook*, which is a series of flip-flops or a primary output node, and a *cone* of combinational logic units. A bell may be followed by another bell. A path from the top node of a cone to its following bell is defined as a *joint*. A joint and its following bell is then called a *bell-bottom*. Combinational logic units in different bells are disjoint. An technology mapping algorithm can then be applied to the combinational logic units in each bell. The thesis presents two groups of technology mapping algorithms for sequential circuits using

LUT-based FPGAs. The first group of mapping algorithms, Hook Map, concentrates itself on hooks and their adjacent parts, that is, flip-flops and their adjacent combinational logic units. Analysis and evaluations of hook mapping are illustrated in the thesis. On the other hand, mapping of a joint, a cluster of combinational logic units between two hooks, is discussed in the second groups of technology mapping algorithms, Joint Map. Examples using the two groups of mapping algorithms show that these algorithms work better than some of existing algorithms. The research in this thesis also shows that optimal technology mapping algorithms for combinational function circuits may not be optimal for sequential circuits.

Table of Contents

LIST OF TABLES

LIST OF ILLUSTRATIONS

Figure Page

INTRODUCTION

This thesis presents a group of technology mapping algorithms for sequential circuits using look-up table based field programmable gate arrays. The purpose of these algorithms is to improve the performance of a sequential circuit using look-up table based field programmable gate arrays with concerns of efficiency of logic area utilization, placement and routing.

Field programmable gate arrays (FPGAs) are a new type of programmable logic device used for semi-custom design. In the past ten years, the products of FPGAs have grown quite rapidly. Comparing it to other logic devices, field programmable gate arrays have many important features. The most important advantage of a field programmable gate array is its short turn around time. Therefore, it attracts the interest of the integrated circuit design industry and computer aided design community. The new technology and architectures used in field programmable gate arrays also brings new challenges to designers and researchers.

A field programmable gate array is often larger than an equivalent custom-designed circuit or other types of semi-custom design circuits. Furthermore, the circuit speed using field programmable gate arrays is slower than an equivalent circuit using other logic devices. Therefore, how to improve the performance (speed) of a circuit using field programmable gate array and minimize the logic area used in a

field programmable gate array or maximize the area use of a field programmable gate array become the two major goals for integrated circuit designers and researchers.

Even though it has many different features in its architecture and implementation, integrated circuit design using field programmable gate array still follows the general procedures used for other integrated circuit design. Technology mapping, placement, and routing are still three main steps between logic synthesis and implementation of the circuit specification. Technology mapping is the first step in a design which concerns itself with the architecture of a specific field programmable gate array technology. Many research reports were published about technology mapping for circuit design using field programmable gate array. Most of them are concerned with combinational circuits, yet only few of them discussed sequential circuits mapping.

Several semi-conductor technologies and circuit architectures are used to manufacture field programmable gate arrays. The look-up table based field programmable gate array using static random access memory is the most popular field programmable gate array used in the current circuit design industry.

This thesis focuses itself on the technology mapping aspect. Several algorithms are presented in this thesis, which are designed to improve the performance of a sequential circuit using look-up table based field programmable gate arrays. These algorithms are also concerned with reduction of logic area used in a chip, and the efficiency of placement and routing.

There are mainly two methods used for technology mapping of sequential

circuits using look-up table based field programmable gate arrays: (1) mapping combinational parts and sequential parts (flip-flops) separately, then combining them with some adjustments; (2) using retiming technology to simplify the circuit [LeiR83]. In algorithms in this thesis, the former method is used. We divide a sequential circuit into two parts: a combinational part and a flip-flop part. Any existing technology mapping for combinational circuits can be applied to the combinational part. Algorithms are designed for flip-flops in this thesis. We improve the final mapping solution using two groups of algorithms. One group of algorithms considers improvements on the mapping of flip-flops and the other group of algorithms refers to addressing the combinational part in the critical path of the circuit.

In the following part of this thesis, we first introduce the field programmable gate array with comparison to other logic devices. The thesis then illustrates three major types of field programmable gate arrays depending on the integrated circuits technology used. Current research on technology mapping of look-up table based field programmable gate arrays is introduced in Chapter 3. Two algorithms are briefly described in this chapter. One of them is a performance oriented mapping algorithm for combinational circuits which is used for combinational parts in our algorithms. The other is a mapping algorithm for sequential circuits which is presented here for a comparison. In Chapter 4, the terminologies used for sequential circuits are introduced. Hook Map, a group of mapping algorithms which is concerned with flip-flops, is discussed in Chapter 5. In Chapter 6, Joint Map, another group of algorithms is presented and evaluated which focuses on

combinational parts in the critical path of the circuit. Several examples are illustrated in Chapter 7 to show how the algorithms work. Our further study concerns placement and routing with technology mapping to make the final configuration solution more efficient.

CHAPTER 1

INTRODUCTION TO FPGAS

Field Programmable Gate Arrays (FPGAs) were first introduced by Xilinx in 1985 and known as *Logic Cell Arrays* (LCAs) [EET85a][EP85][EET85b][CarD86], which are new types of *programmable logic devices* (PLDs). In the past ten years FPGA technology has grown quite rapidly. It had become a $357-million business by 1993 and is excepted to generate over one billion dollars by the end of this century [Act94].

1.1 Programmable Logic Devices

A digital circuit designer can achieve his or her desired circuits by using a full-custom approach or a semi-custom approach. In the full-custom design approach, a circuit is designed and fabricated fully according to the user's specifications. Circuits made in this way can make the most use of the chip. However, laying out this circuitry is very difficult. At present the *computer aided design* (CAD) tools and silicon compilers do not assist this process very much. The layout is mainly done by hand. Full-custom design takes much time to bring the products to market and it costs much, therefore it is used only for producing a large volume of circuit demand, such as watches and calculators.

A so-called semi-custom design does not specify every bit on a chip for a

desired circuit but uses some pre-designed logic elements, known as standard cells, as bricks to build a circuit or may use a fabricated wafer with some functional units already built-in to configure a circuit.

Standard cells used in semi-custom designs implement those digital circuits that are frequently used in many other circuit designs. The standard cells used for semi-custom designs are stored in a database called the *cell library* of the design system. The designer partitions a circuit into several parts that correspond to standard cells available in the cell library and then makes interconnections among these cells. A chip is then fabricated according to this design. Every cell area in a circuit is fully used. No dies are wasted in standard cell technology. However, standard cells may not be practical for some circuit designs (we shall see that later in this thesis). Some circuit designers do not identify standard cells as semi-custom [Ekl81].

Another method for semi-custom design is to use some silicon devices in which there are certain logic elements. The prefabricated logic elements in a wafer can be simple "AND/OR" structures, or a more sophisticated structure of uniformed functional models. The formal devices are used to implement two-level logic. The earlier devices that consist of uniformed functional models are referred to *gate arrays* which have thousands of gates (often referred to "a sea of gates").

Digital circuit devices can be either completely made for a specific function when they are fabricated or configured completely after they are fabricated. If the circuits are made completely when the chips are manufactured, the circuit devices are called fixed devices or with fixed part if only a part or some parts of a chip are fixed.

All full-custom devices are in fixed formats, since once they are made in the factories they can no longer be altered. A digital circuit made by semi-custom design is not in a fixed format. Contrarily, if a chip or a part of it is not completely made until it is configured according to the circuit specification after the wafer is manufactured, then the circuit device is called a "programmable device."

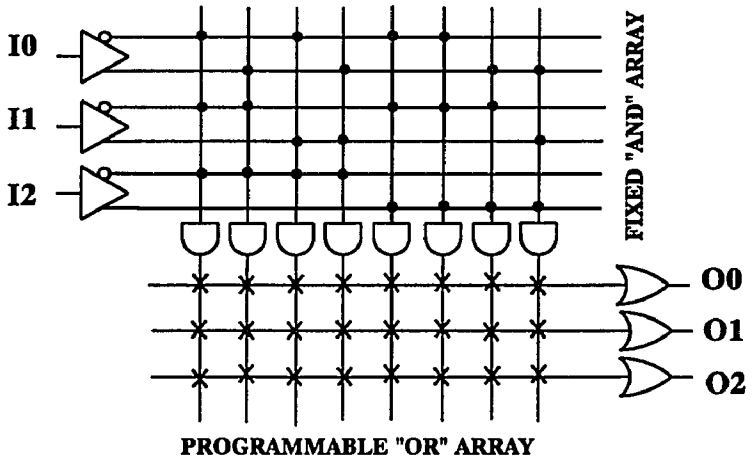


Figure 1.1-1 Logic diagram of PROM

The configuration of a circuit specification on a chip is called "programming." The programming process can be done by the manufacturers in their factories or by the users at their work sites. If it can be programmed by a user, a programmable device is called "field programmable." There are multiple technologies used to program or configure a programmable logic device. Mask-programming is often employed for *Mask Programmable Gate Arrays* (MPGAs). An MPGA consists of an array of fully functional models and routing facilities among them. When an MPGA chip is manufactured, its fabrication is not complete. The metalization layer, the

wiring of functional units, is not yet made. The mask for the wiring is customized by the circuit designer and the wiring is built on the chip by the manufacturer during the final wiring stage.

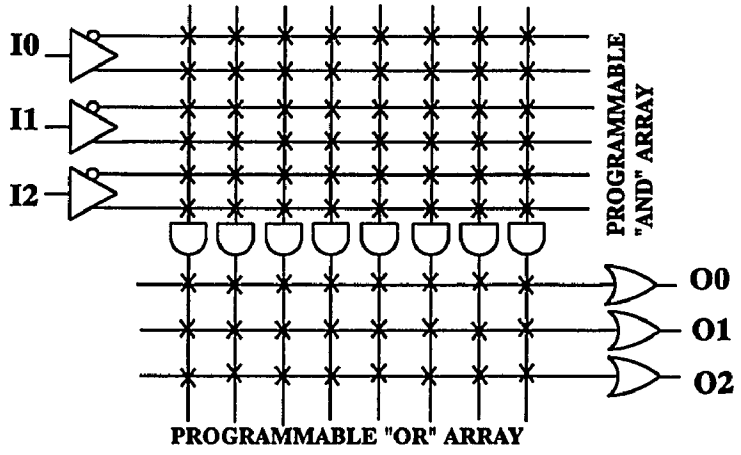


Figure 1.1-2 Logic diagram of FPLA

Other PLDs are often referred to as *Programmable Read-Only Memories* (PROMs) (Figure 1.1-1), *Field Programmable Logic Arrays* (FPLAs) (Figure 1.1-2), *Programmable Array Logic* (PALs) (Figure 1.1-3) and by several other names. Sometimes PLDs mainly refer to these devices in an historical context. We may call them conventional PLDs. In this paper the word PLD is used for any digital device that consists of certain type of logic structure with uniformed format and must be customized to the circuit specification in order to complete the chip design after the chip is fabricated. Technology used for programming logic for early PLDs was known as the fuse configurable diode matrix, developed in the mid-1960s [PelH91]. The matrix was small and programming it was very difficult at the time the

technology was invented. Bipolar technology is used in fuse matrices. A fuse is made of alloy with a low melting point. A fuse matrix is then applied to the device so that each pair of crossing wiring lines in the device is connected by a fuse at the intersection when the device is manufactured. When a specified sequence of inputs and voltages is applied to the device pins, a high currency passes through the target fuse and the fuse melts. As the fuse blows up, the associated crossing signals are disconnected and cannot be connected again.

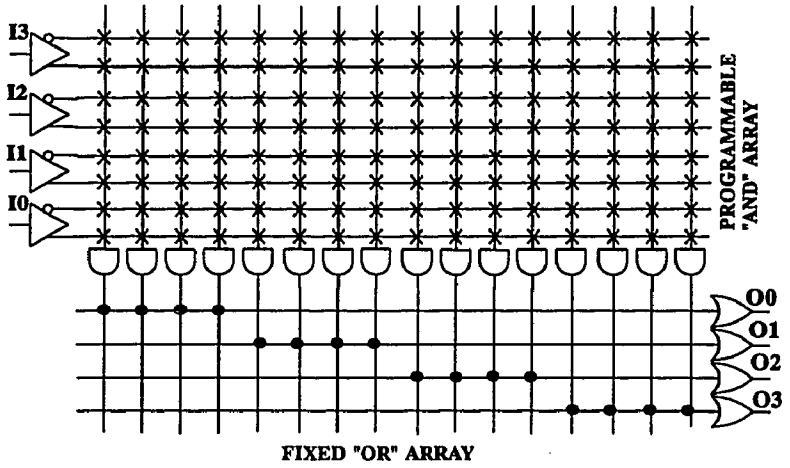


Figure 1.1-3 Logic diagram of PAL

PROMs, FPLAs and PALs consist of an array associated with "AND" gates followed by an array associated with "OR" gates. An entire circuit using PROM, FPLA, or PAL implements a function in a sum-of-product form. Every gate corresponds to either a row or a column in the array. Each row or column then connects with an input signal or an output signal. When the chip is made, the wiring is not yet completed: each cross point or intersection of row and column is linked by a fuse. This kind of device is suitable to circuits with a small amount of logic,

because the routing part occupies most of the chip area. The purpose of programming these devices is to blow out some fuses and have gates, which are necessary to implement the circuit, connected by the intact fuses. This can be done by the users. In a PROM the "AND array" is in a fixed form and the "OR array" is programmable. However, a PAL performs in an opposite way: the "OR array" is fixed and the "AND array" is programmable. With more flexibility the FPLA is manufactured in two programmable arrays: both "AND array" and "OR array" are programmable.

PROMs have two different programming methods: one is mask programming and the other is *antifuse* technology. The antifuse-programming technology was developed by Actel Corporation in 1988, and called a *Programmable Low Impedance Circuit Element*, or PLICE. Unlike the fuse matrix, the PLICE is unconnected in its initial state when the chip is manufactured. It is connected when it is programmed by applying higher voltage to it. Mask-programmed and antifuse programmed PROMs cannot be altered once they are programmed. The mask-programming is made by the manufacturers and antifuse programming is done by the circuit users. FPLAs may also be configured by using antifuse technology. If it is configured by the user, a PLD is called *Field Programmable* (FPLD). A PLD may also be reconfigured, that is, the contents of the circuit once specified can be erased and new contents can be written on the chip. This process may be done repeatedly until the user is satisfied with the circuit. An *Erasable PROM* (EPROM) is such a device. Ultraviolet light is used for the erasure. A PROM also can be erasable in an electrical way. This kind

erasable PROM is called *Electrically Erasable PROM* (EEPROM). We call PROMs, PALs, FPLAs, EPROMs, and EEPROMs conventional PLDs.

Comparing MPGAs with conventional PLDs, we see that MPGAs have high logic density while conventional PLDs are easily configurable or even reconfigurable. The *field programmable gate array*[1] is a new type of programmable logic device somewhere between MPGAs and conventional PLDs. The general structure of an FPGA is an array of fully functional models (or logic blocks) and each part of the device, either logic blocks, routing parts, or input/output parts, can be programmed by the users using multiple configuration technologies depending on the nature of the FPGA chips. Since the functional models are arranged in an array style in an FPGA and are much more sophisticated and have more functionality, an FPGA is more similar to an MPGA than it is to a conventional PLD.

1.2 FPGAs and MPGAs

The common feature of MPGAs and FPGAs is that both contain arrays of functional blocks which can implement multi-level logic whether combinational or sequential. In conventional PLDs either "AND-array" or "OR-array" actually means that a group of "AND" gates or "OR" gates associated with an array of connections of input lines as shown in Figure 1.1-1 through Figure 1.1-3. Either "AND" gates or "OR" gates are in a linear group rather than a two-dimensional array.

---

[1] The word *field programmable gate array* was employed by Signetics in 1977 for its PLD product, which contains a programmable AND-array with programmable-polarity outputs but no OR-array, descriptions of which can be found in [Alf89] and [Lal90]. It is different from what we discuss in this thesis. The word *field programmable gate array* used by most researchers is now referred to the meaning in this thesis.

1.2.1 Advantages of FPGAs

MPGAs are customized by manufacturers using masking and FPGAs are customized by the users themselves using different programming technologies. The major design steps for FPGAs are different from for MPGAs. Figure 1.2-1 compares the design processing for FPGAs and MPGAs [Tri94].

The field programmability of FPGAs reduces each design's test pattern generation, wafer fabrication, packaging, and testing in FPGA's design cycle. It usually takes weeks from timing simulation to system integration for MPGAs, but only from a few milliseconds to a few minutes for FPGAs. Therefore, the most important and greatest advantage of FPGAs over MPGAs is that an FPGA has a shorter design cycle than an MPGA. This fast turn-around feature makes FPGAs more popular than MPGAs for many designers in today's rapidly growing high-

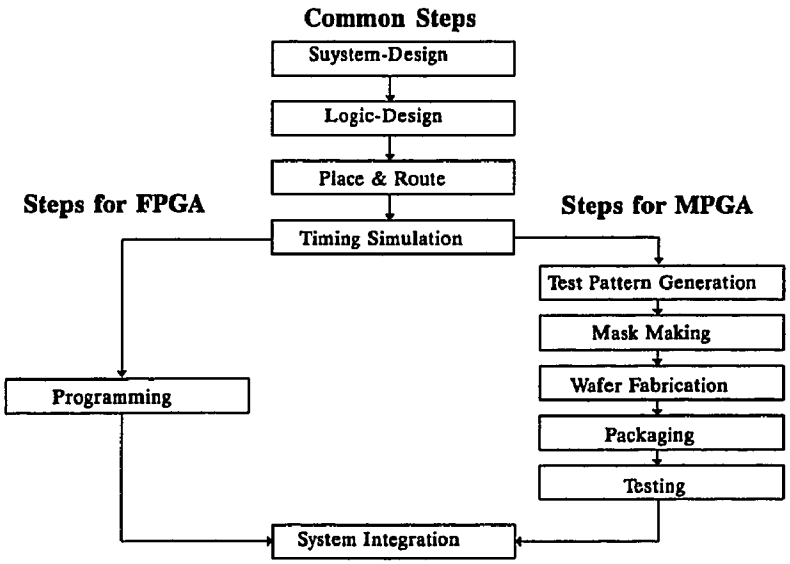


Figure 1.2-1 Design steps for MPGAs and FPGAs

technology environment. An FPGA is therefore an ideal solution for the time-to-market problem for the *Integrated Circuit* (IC) industry today.

Custom masking an MPGA is expensive, costing several thousand dollars for each mask. Every unit manufactured using the mask technique bears the mask charge. To reduce overhead costs the units must be made in large amounts. However, statistics show that not more than 25 per cent of existing designs require

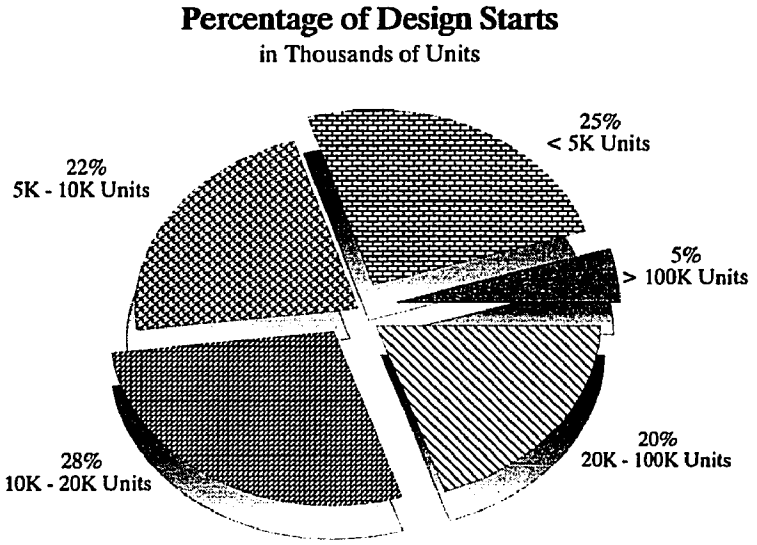


Figure 1.2-2 Percentage of design starts (Source: [Xil92])

more than twenty thousand units [Xil92] (see Figure 1.2-2). When MPGAs are used for low volume or mid-volume designs, the masking charges are significant. However, customizing an FPGA needs much less time and the technology used is much less expensive than masking. Additionally FPGAs are well suited to many mid-volume and low-volume designs. Figure 1.2-3 shows the preferred implementation options in the design comparing FPGAs, MPGAs, conventional PLDs, and standard

and custom cells. Not only are customizations less expensive for FPGAs, but also the testing cost is lower using FPGAs than using MPGAs, since FPGA users are not required to write design-specific tests for their designs. The manufacturers test every FPGA for all possible designs that may be implemented on it before the FPGA chips are delivered to the users. This manufacturer's "one-test" program makes the design process much easier and less expensive.

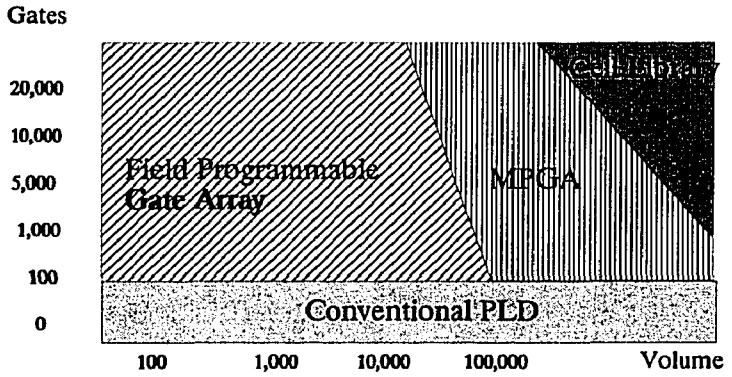


Figure 1.2-3 Gate arrays starts by unit volume (Source: [Xil92])

Once a reprogrammable FPGA is installed in a device, the chip needs only to be reprogrammed by using an upgraded design instead of being entirely replaced by a new one if there is some modification to be made on the circuit. In contrast, an MPGA designer must redesign a new MPGA chip, requiring a whole new design process. This unique feature allows the FPGA users not to worry about products being out of stock while avoiding over-purchasing of unnecessary parts to avoid this situation.

FPGAs are now used in a variety of integrated circuits (ICs). The sales of

FPGAs are increasing steadily. Table 1.2-1 shows the sales figures of FPGAs

US $ Millions	1992	1993	1994	1995	1996	1997	1998
Simple PLD	566.4	576.1	605.3	630.4	644.3	651.7	652.2
Complex PLD	156.4	254.6	358.2	468.5	602.4	739.8	879.2
FPGA	247.6	353.9	472.1	599.4	732.8	861.7	998.6

Table 1.2-1 Programmable logic market, five-year forecast as of April 1994.

(Source: In-Stat, Inc.)

compared to other programmable logic devices. The complex PLD in the table includes EPROMs (*Erasable Programmable Read Only Memory*), which are sometimes classified as a kind of FPGA. This is discussed in Chpater 2. The growing rate of logic device market is illustrated in Figure 1.2-4. Their easy reprogrammability makes FPGAs more useful for researchers to build computing machines for both teaching and experimentation. By August 1995, more than fifty computing machines implemented in FPGAs had been built [Guc95].

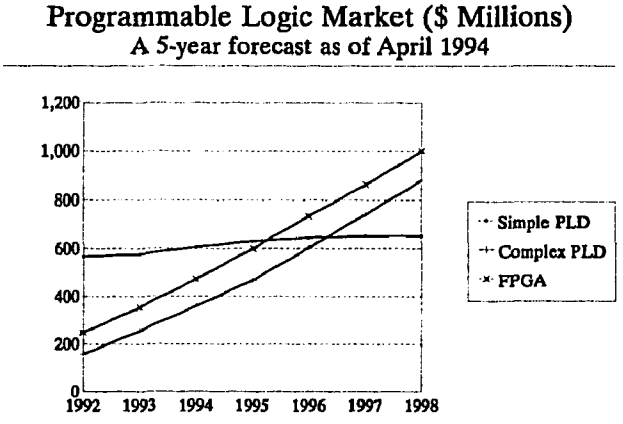


Figure 1.2-4 Five-Year forecast of PLD market

1.2.2 Disadvantages of FPGAs

The fact that FPGAs are customized in the user's sites shortens their time-to-market, but brings extra on-chip programming overhead circuitry to the users. The area of programming overhead stores the configuration software and cannot be used for any desired circuit. Therefore, the FPGA gate density becomes lower compared with MPGAs with same size wafer. The programmable switches and routines in an FPGA are larger than the mask programming area that can be built in an MPGA. Signals pass through the programming switches in an FPGA more slowly than they do through the metal layer in an MPGA. Therefore, FPGAs have lower logic density and greater time delay than equivalent MPGAs

An FPGA is as much as ten-times larger than its equivalent MPGA for the same number of gates. This makes FPGAs more expensive than MPGAs on a per-chip basis. The more logic capacity an FPGA has, the more area is used for programming, and the higher the cost of an FPGA becomes. For this reason an FPGA may contain only as many as 10,000 gates. Currently the largest Xilinx's XC4025 FPGA has approximate 25,000 equivalent gates. If FPGAs are used for large designs, the design must be split onto several FPGAs adding to the difficulty a designer may face. Multi-FPGA systems are used for these cases [Hau95]. Figure 1.2-5 displays costs over volume of products for both MPGAs and FPGAs using 5,000-gate chip. The cost curve of FPGAs approximates a straight line, the curve for

MPGAs levels off as the number of project units grows. When the volume of units increases, the total cost of the FPGAs grows faster than that of the MPGAs. By the volume of 20,000 units, the total cost using FPGAs becomes greater than the total costs of MPGAs. This fact proves that FPGAs should be used mainly for low and mid-volume IC designs (see Figure 1.2-3).

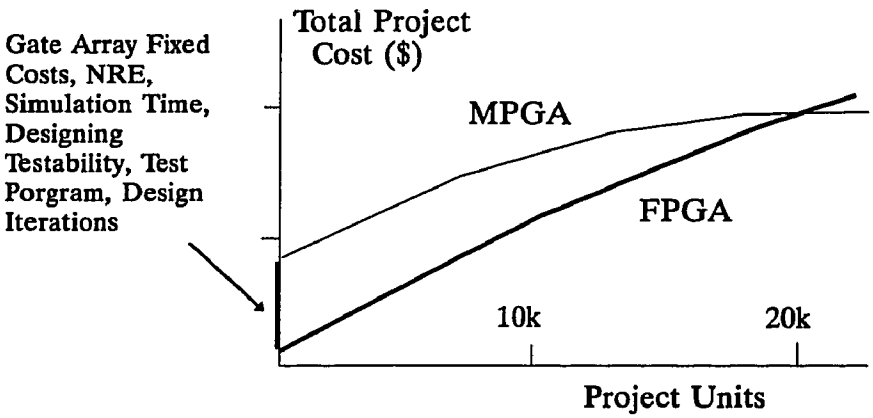


Figure 1.2-5 Typical break-even analysis of 2000 gates
(Source: [Xil92])

MPGA interconnection routines are made of a metal layer, which causes very low resistance. However, in FPGA programmable interconnection parts, programmable points or programmable switches, produce more resistance to interconnection routines due to the nature of these programmable parts. Inside a switch capacitance is raised by the programmable points.

Compared to an equivalent MPGA with the same gate capacity, FPGAs are larger and routines between logic blocks must be longer. This causes more resistance and capacitance within routines. Resistance and capacitance within routines decrease the speed of signals between logic units inside FPGA chips. Time-delay in a current

FPGA is about three-times that of an MPGA. Therefore, FPGA performance is poor.

In summary an FPGA has lower speed and lower density of logic utilization than an MPGA. Therefore, improving the speed and having an FPGA implement more logic are the paramount interests of the VLSI (Very Large Scale Integrated circuit) designers. Table 1.2-2 shows a summary of comparisons between MPGAs and FPGAs.

Name	Time to Market	User Pro-grammable	Repro-grammable	Initial Cost	Price per-Chip	Cost of Test	Speed	Density
MPGA	Slower	No	No	Higher	Lower	High	Higher	Higher
FPG	Faster	Yes	Yes*	Lower	Higher	No	Lower	Lower

* SRAM and EPROM based FPGAs are user reprogrammable.

Table 1.2-2 Summary of Comparisons between MPGAs and FPGAs

CHAPTER 2

ARCHITECTURES OF FPGAS

2.1 Overview of FPGAs' Architectures

Architectures of an FPGA family can be classified by their programming technologies and how their components are arranged on a chip.

FPGA			
SRAM-Programmed		Antifuse-Programmed Channeled	EPROM-Programmed Array (Complex PLDs)
Island	Cellular		
Xilinx LCA AT&T Orca Altera Flex UTFPGA1 Concurrent Logic CLi6000	Toshiba, Plessey's ERA Atmel's CLi family Algotronix CAL Triptych	Actel's ACT-1 and ACT-2 Quicklogic's pASIC Crosspoint's CP20K	Altera's MAX 5000 and MAX 7000 AMD's Mach Xilinx's EPLD Lattice pLSI & ispLSI

Table 2.1-1 Families of FPGAs (Source: [Tri94])

There are three major types of programming technologies used for FPGAs. They are SRAM (*Static Random Access Memory*)-programming, antifuse-programming, and EPROM (Erasable Programmable Read Only Memory)-programming. Corresponding to different types of programming technologies there are two basic types of architecture organizations: array-based and row-based devices. A row-based device is also called channeled device. Array-based devices fall into several sub-styles [Tri94]. Table 2.1-1 shows the families of FPGAs. The names of manufactures and products listed in the table represent only a portion of the products

currently on the market.

2.1.1 SRAM-programmable FPGAs

In an SRAM-programmable FPGA, logic units and connect units are controlled by configuration memory cells. The configuration memory is an SRAM, which is built with the logic units on a same chip. The logic specification and the routine which implement the application circuit are loaded in the configuration memory from an external source.

The configuration memory cells hold the logic information only when the power is on. When the power is turned off, the circuit's programming is lost. An SRAM FPGA must be re-programmed each time the power is turned on. This volatility is an obvious disadvantage of SRAM-programming because it takes time to reload the circuit logic program from outside the FPGA to its configuration memory cells. However, this drawback can be reduced by adding a logic element onto the SRAM FPGA, which automatically reloads the program every time the power is restored. This reloading takes a short period of time (from 2ms to 30ms, shorter than most systems' start-up times). This technology gives SRAM FPGAs a feature called "virtual non-volatility."

Although one might think this volatility a disadvantage since SRAM FPGAs require re-loading circuit specification program every time power applied, it brings benefits to SRAM FPGAs. Since the configuration memory cells must be re-loaded, the FPGAs are reprogrammable. This reprogramming does not add additional costs; a designer only has to change the circuit specification from the external source and re-

load it onto the configuration memory cells, then test it and reprogram it until the circuit has satisfied his or her requirements.

This reprogramming feature is specially useful for VLSI designs, making updating circuits which are implemented using re-programmable FPGAs very easy and less expensive than the alternative. To update such a system the only thing needed is to modify the programming of the FPGA and then re-load the new program onto its configuration memory cells of the FPGA. The updated program can be stored in any type of data storages. It may be convenient to store the program on a floppy disk and deliver it from a circuit designer to an FPGA customer or even transfer the program via network communications. Since the logic specification can be stored in memory and the FPGA can be reprogrammed, there is a possibility that a configuration may be done piece by piece in a system. If a system can be divided into parts that are not used simultaneously, the parts can be designed into separated configurations of the FPGA. Only when a specific part of the circuit is needed is it then programmed; the part not needed now may be erased to make space for a new logic. This is called *time-share of a reprogrammable FPGA*. This feature allows an FPGA virtually to expand its capacity many times that of the chip actually has. Reprogrammable FPGAs are widely used for board-level tests. After a board-level test, the FPGA is programmed with the application logic. Using this design system-level test does not require additional hardware.

Due to this reprogrammability, SRAM FPGAs maintain very high quality. Because each part of an SRAM FPGA can be tested at the factory the test does not

destroy the parts. In contrast to that, programming antifuse cause damages to package pins, programming yield for SRAM FPGAs is always 100%, guaranteeing their quality.

Another advantage of SRAM-programming over antifuse-programming and EPROM-programming is that SRAM-programmed FPGAs use static gates and do not have passive pull up and sense-amplifier circuitry so that they avoid prohibitively high power dissipation for high capacity or high-speed logic and consume lower power.

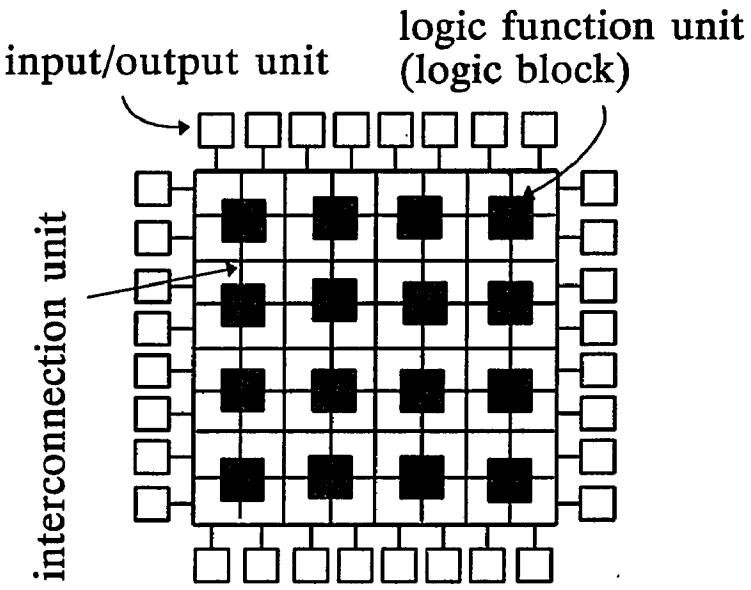


Figure 2.1-1 Block diagram of a logic block

Similarly SRAM-programmed FPGAs also demonstrate advantages over CMOS (*Complementary Metal Oxide Silicon*), because the way to process SRAM-programmed FPGAs is the same as to process CMOS for ASICs (Application Specific Integrated Circuits) and is very similar to processing CMOS memories. Any improvements made for CMOS memories can likewise be applied to SRAM-

programmed FPGAs.

So far all SRAM-programmed FPGAs have similar architectural characteristics in that an FPGA device consists of an array of uniformed logic cells which are interconnected by either vertical, horizontal routines or even diagonal routines, such as routines in Triptych [EbeB91]. The logic cell in an FPGA is not a simple "AND" or "OR" logic, but either a combinational function generator with multiple inputs and one or more outputs, or a combinational function generator followed by flip-flops. This is the main difference in device structures between FPGAs and conventional PLDs.

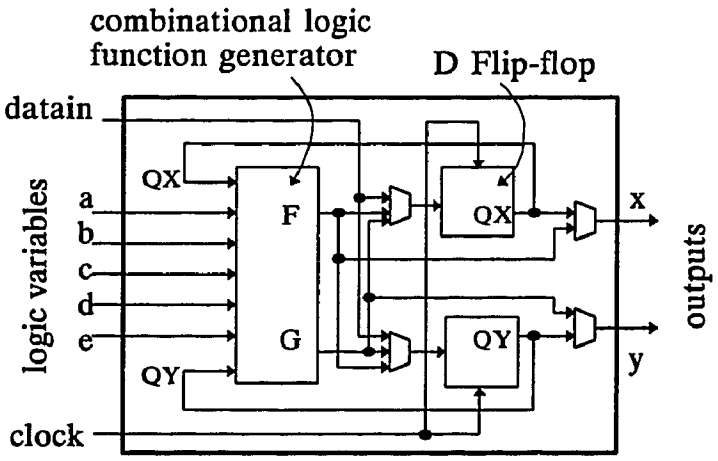


Figure 2.1-2 Logic Diagram of XC3000's CLB

The so called "island-style" architectures in Table 2.1-1, which are sometimes called "Manhattan Architecture" [York93], is a typical SRAM FPGA device structure. Among this category, Xilinx's LCA is the prototype. Logic cells in Xilinx's LCA series products are capable of performing both combinational functions and sequential

functions. Cellular-style SRAM FPGAs consist of smaller logic cells, which may only perform conbinational functions. To compute sequential functions multiple logic cells are applied, as does Triptych [EbeB91]. Figure 2.1-1 shows a block diagram of Xilinx's LCA and Figure 2.1-2 illustrates a logic diagram of LCA's *Configurable Logic Block* (CLB). Details about Xilinx's XC products are discussed in section 2.2.

2.1.2 Antifuse-programmable FPGAs

An *antifuse* is an electrically programmable two-terminal device [HamM88]. An antifuse device can be changed from high resistance to low resistance. Once done however, it cannot be changed back to high resistance. A programming voltage applied across terminals of an antifuse device make this irreversible change. Antifuse-programmed FPGAs are configured by programming antifuses among logic units to implement the circuit design [Cer86] [HamM88] [WhiB90].

Low on-resistance and small size are two main advantages of antifuse-programmed FPGAs. Antifuses have significantly lower on-resistance and capacitance than SRAM-programmable FPGAs. Therefore, the time-delay of antifuse-programmed FPGAs is shorter than of SRAM-programmable FPGAs. The on resistance of antifuse-programmed FPGAs can be as low as 100 to 600 ohms. The size of an antifuse is only the size of a via, which is used to connect metal lines in an MPGA. An Actel antifuse is as small as one fiftieth the size of the SRAM elements and is as small as one-tenth the size of a CMOS EPROM cell. Currently a single FPGA can integrate more than 1,000,000 antifuses, which gives an FPGA more flexibility to implement more logic. Current antifuse-programmed FPGAs can

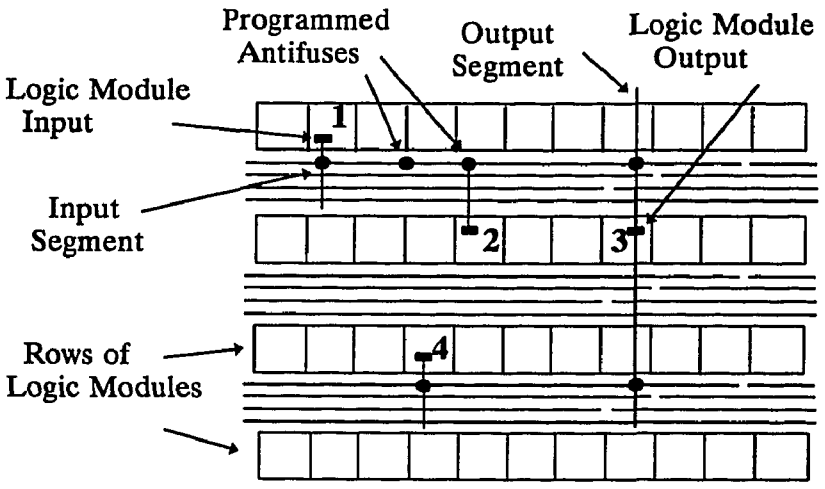


Figure 2.1-3 Actel architecture (Source: [McCW94])

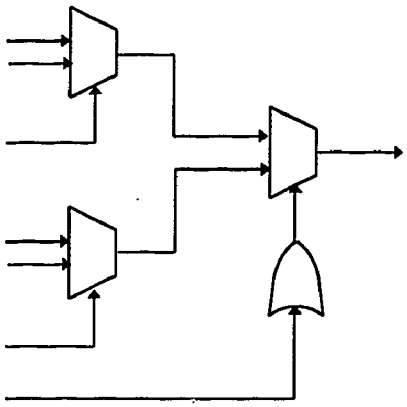
implement logic as equivalent to a 10,000-gate MPGA. The system clock can be as fast as 75MHz.

The main disadvantage of antifuse-programmed FPGAs is that they lack reprogrammability. Once an antifuse is programmed, the circuit cannot be changed any more. The circuit specification is recorded by programming the antifuse, a process very difficult to interpret. Because of this, antifuse-programmed FPGAs are said to be secure.

While Xilinx' FPGAs are representative of SRAM-programmable FPGAs, Actel FPGA products are typical antifuse-programmed FPGAs [ElA89] [Ahr90] [Sch93] [WhiS93].

Figure 2.1-3 illustrates the architecture of Actel FPGAs as a simplified block diagram. Logic modules are grouped row by row. Rows of logic modules are separated by sets of horizontal routing channels. Each channel contains predefined

wiring segments of various lengths and offsets, rather than a complete line. To connect logic modules there are vertical wiring segments passing through the modules and across the channels. A vertical wiring segment and a channel are connected by programming an antifuse at their intersection. Each logic module can accept several inputs and computes a single output function. Each input of a module is connected with a vertical wiring segment. Each output signal is associated with a long vertical wiring segment since an output signal may be used as input for several modules. In the figure, the output of module 3 is used as an input signal for module 1 as well as



2.1-4 Logic diagram of Macrocell

for module 2 and module 4, respectively. The vertical wiring segment which carries the output signal of module 3 connects two horizontal segments by programmed antifuses. The connected horizontal segments then connect with input segments of module 2 and 4 respectively. Two adjacent horizontal segments can be connected by programming an antifuse between them. Using this, the two horizontal segments which connect to module 1 and module 2, respectively, are connected in the top

channel.

In the basic version of Actel's FPGAs, a logic module is simply a two-level two-input multiplexer structure. This general purpose logic model can compute any combinational function of two inputs, any function of three inputs (except the three-input NAND and exclusivity functions), many functions of four inputs, and other functions up to eight inputs. By applying multiple logic modules and connecting them properly flip-flops and other macro functions can be implemented. In more advanced Actel FPGAs, a combinational function unit computes input for a sequence of two D-type flip-flops. This is illustrated in Figure 2.1-4.

2.1.3 EPROM-Programmable FPGAs

EPROM-programmable FPGAs are another type of FPGA, which is a result of an evolution of programmable logic devices. Early PALs were programmed on bipolar fuses. Bipolar fuse-programmed PALs produce high power dissipation and can be programmed only once. In addition, bipolar fuse-programmed PLDs can implement only very limited amount of logic, up to only a few hundred gates. EPROM-programming FPGAs use EPROM or EEPROM instead of a bipolar fuse to control the routines, thereby employing CMOS technology in conjunction with floating-gate programmable transistors. These EPROM (or EEPROM) bits are much smaller than fuses and are also electrically programmable and erasable. The EPROM transistor has two states, unprogrammed (erased) and programmed. The programming of an EPROM bit is realized by applying high voltages and the subsequent charge tapping on the floating gate. By resetting the EPROMs, EPROM-

programmed FPGAs are reprogrammable.

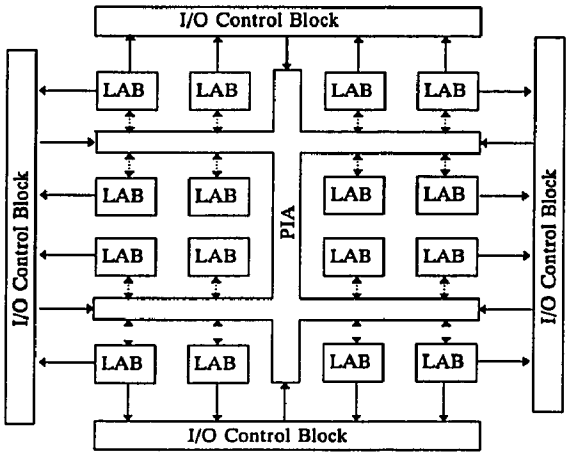


Figure 2.1-5 Block diagram of Altera's EPROM-programmed FPGA (Source: [Tri94])

The CMOS based EPROM-programmable FPGAs greatly increase logic capacity and reduce power consumption compared to traditional PLDs. EPROM-programmable FPGAs have some features similar to SRAM-programmable FPGAs due to their reprogrammability. A unique feature of EPROM-programmable FPGAs is that they all contain a programmable security bit, which controls access to the data programmed into the device. If this feature is used, a proprietary design implemented in the device cannot be copied or retrieved.

The Altera Classic [Har84], MAX 5000, and MAX 7000 families are main members of the EPROM-programmable FPGA family. Figure 2.1-5 shows a typical EPROM-programmable FPGA device architecture of Altera's products. The acronym MAX stands for *Multiple Array matriX*. In the figure, each LAB is a *Logic Array Block* which is an array of macrocells. Each macrocell consists of three parts: the logic array, which is an AND-array combined with "OR" and "XOR" gates and

computes all combinational logic functions (Figure 2.1-6 shows its logic diagram); the programmable register which can be configured to provide a variety of flip-flops, such as D-type, T-type, JK-type, or SR-type flip-flops (of course, the register part can be bypassed); and programmable I/Os, each of which can be configured as a dedicated output, a dedicated input, or as a bidirectional pin. PIA, *Programmable Interconnect Array*, is employed for connecting logic blocks and I/O pins.

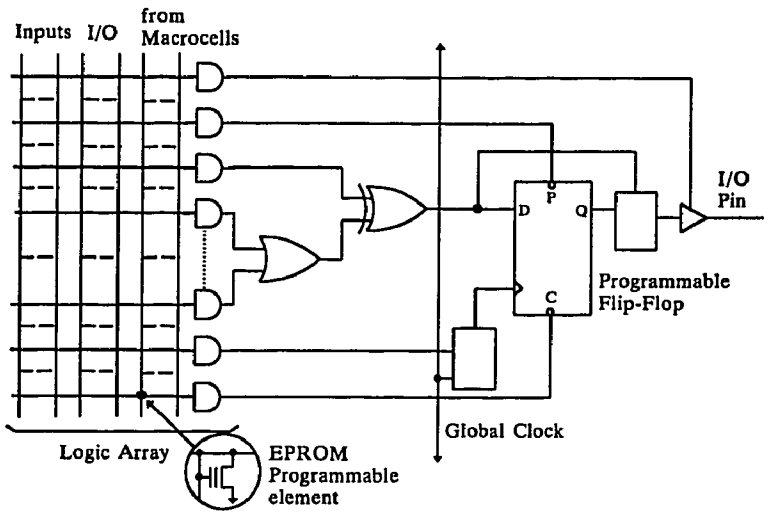


Figure 2.1-6 Logic diagram of Altera LAB's Macrocell
(Source: [Tri94])

This multi-array style of architecture makes devices more flexible. Either antifuse or SRAM bits can be used on this LAB/PIA style architecture. A new kind of FPGA with this multi-array structure is the FLEX (*Flexible Logic Element MatriX*) from Altera. FLEX is built on standard CMOS technology and the architecture is very similar to its predecessor, the EPROM-programmable FPGA; but the programming elements of FLEX devices are SRAM bits rather than EPROM. A logic array block of FLEX consists of *Logic Elements* (LEs). An LE is very similar

to a CLB in Xilinx's LCA. Each LE has a look-up table which generate any four-input combinational function and is linked with a register.

2.2 Look-Up Table Based FPGAs

In SRAM-programmed FPGAs, logic function generators are implemented by static random access memory. A four-input conbinational function generator corresponds to a 16-bit register, which dominates a total of 2^{16}, which equals to 2^{2^4}, different logic functions. This means that a four-input conbinational function generator can compute any logic function with at most four variables. FPGAs with this feature are called *look-up table based FPGAs* (LUT-based FPGAs for short). Xilinx's LCA is typically such a device.

2.2.1 Logic Block

The architecture of Xilinx's XC series FPGAs is called Logic Cell Array (LCA). This name implys that the device is built in an array of logic cells. An LCA device consists of three programmable parts: logic cells, routine resources, and input/output blocks (IOBs). Each of these three parts is user-configurable. The logic cells play a core role in an LCA, which are called *Configurable Logic Blocks* (CLBs). An overall structure of an LCA is shown in Figure 2.1-2. Xilinx has announced three XC series products: XC2000, XC3000, and XC4000. The smallest XC device contains an 8×8 CLB array, while the largest XC device, XC4025, contains a 16×16 CLB array. Each CLB is surrounded by two-level wiring routines. The perimeter of a device is occupied by a number of IOBs.

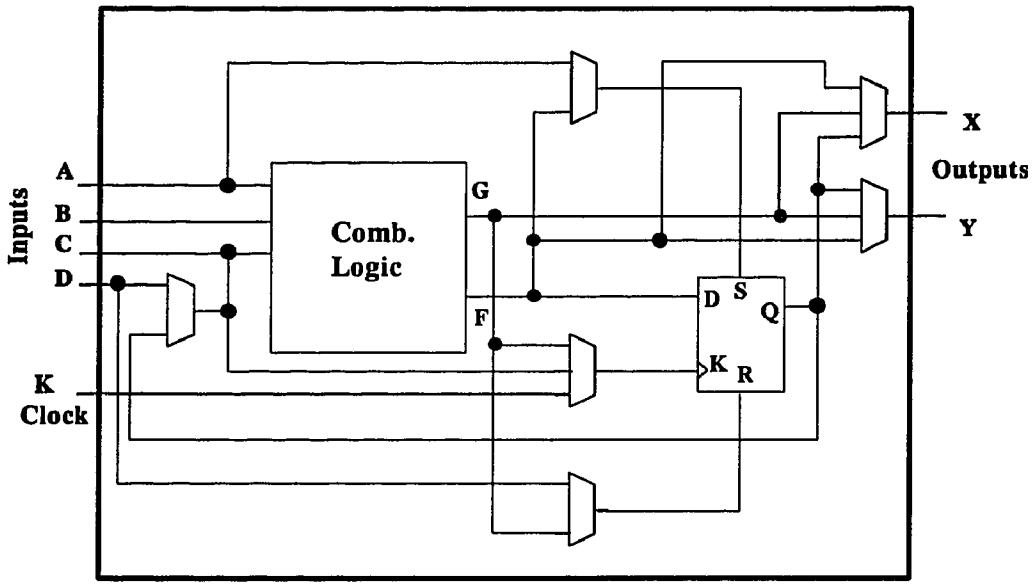


Figure 2.2-1 Logic diagram of XC2000's CLB (Source: [Xil92])

In an LCA device, logic functions are implemented by programmed look-up tables. A CLB consists of combinational function generator(s) and flip-flop(s). A CLB may compute more than one different output function. Multiplexors are employed in a CLB to control functional options. The functionalities are different among XC2000, XC3000, and XC4000. A later series is more powerful and more flexible than its predecessors. A CLB in an XC2000 LCA device can have up to four inputs from outside of the block for its combinational function generator and a clock signal for its D-type flip-flop. The output from the flip-flop can be fed back to the combinational function generator within the block as well. Figure 2.2-1 illustrates this logic scheme.

In an XC3000 device a CLB has one more D-type flip-flop than in an XC2000 device and its combinational function generator may have five inputs from outside of

the block. In addition, the flip-flops can receive an input directly from outside of the block, which is called *Direct In* (DI). Outputs produced by flip-flops also go to the combinational function generator using wiring inside of the block. Figure 2.1-2 shows the logic diagram of a XC3000's CLB. An XC4000 CLB shows much difference from its precursors. Figure 2.2-2 is a simplified logic diagram of XC4000-Families CLB. In an XC4000 CLB, there are still two D-type flip-flops, but there is

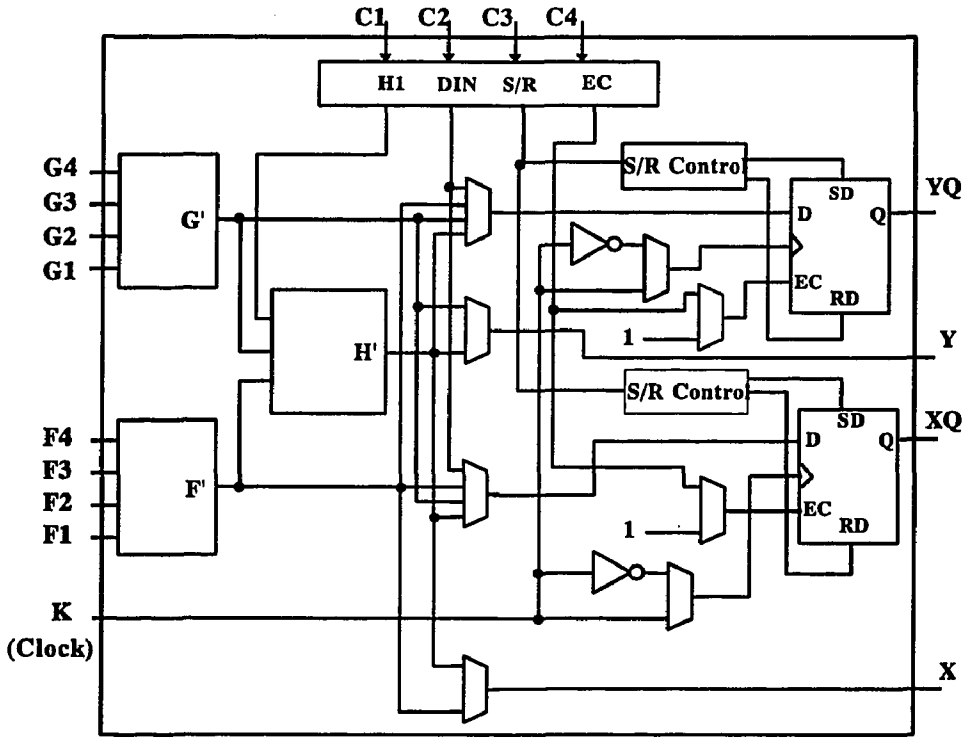


Figure 2.2-2 Block diagram of XC4000-Families CLB
(Source: [Xil94])

no interconnection between any output of flip-flops and input of the combinational function generator inside the block. When it goes to the combinational function generator of the same block, a signal from any flip-flop must travel along wiring

outside the block to reach an input port of the same block. A block may have four outputs, two from combinational function generators and the other two from the flip-flops directly. The entire block has more input ports than CLBs in either the XC2000 and XC3000 products. Furthermore, there are a total of three combinational function generators in an XC4000 CLB. Two four-variable function generators are independent. The rest of the function generator combines the outputs from the two independent function generators with a ninth input.

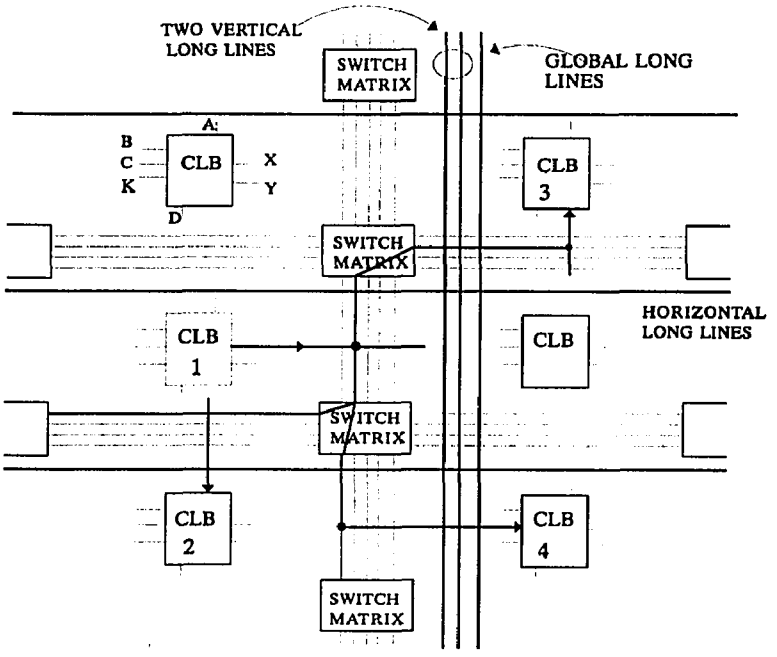


Figure 2.2-3 General-Purpose Interconnect and Long lines of XC2000 (Source: [Xil92])

2.2.2 Routine of LUT-Based FPGAs

The programmable interconnect of an LCA lies in three categories: general

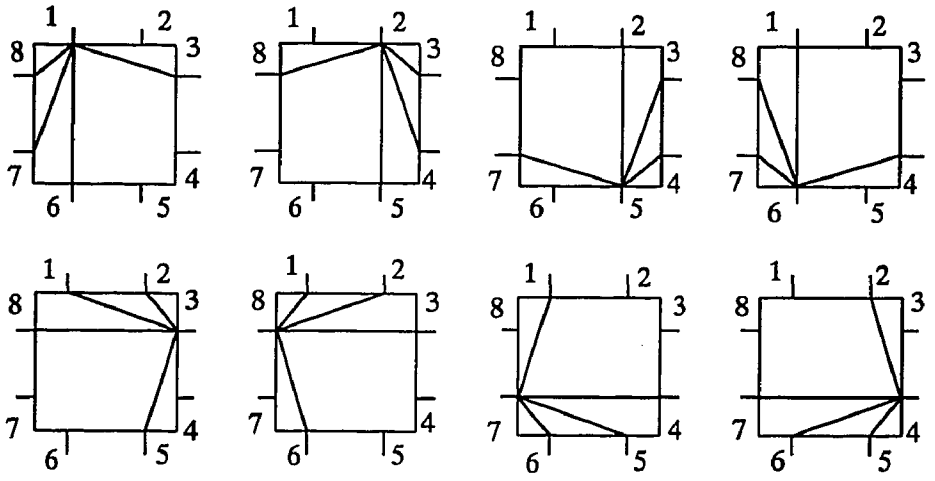


Figure 2.2-4 Switch matrix connections of XC2000
(Source: [Xil92])

purpose interconnect, longlines, and direct connection, as illustrated in Figure 2.2-3. All interconnections are composed of metal segments. In the intersections of wiring lines are programmable switching points or *switch matrices*. General-purpose interconnection is composed of several horizontal metal segments between the rows and several vertical metal segments between the columns of logic and I/O blocks. The number of wires in each group of segments is different in different XC series. Each metal segment is only the height or width of a logic block. Switching matrices are built to connect adjoining rows and columns at the intersections of rows and columns where these segments would cross. Switches in the switch matrices and on block outputs are special transistors. Each such transistor is controlled by a configuration bit. A connection is provided between logic block output switches and adjacent general interconnect segments. A switch matrix can connect an interconnect segment to other segments to form a network. Figure 2.2-3 shows the general

interconnect used to route a signal from logic block 1 to logic block 2, 3, and 4. General-purpose interconnection allows signals passing through one logic block to another by making proper combinations of closed switches in a series of switch matrices. The inputs of the logic or I/O blocks can be programmed with configuration bits to select an input network from the adjacent interconnect segments. Possible combinations of closed switches in a switch matrix are shown in Figure 2.2-4. General-purpose interconnect schemes shown in Figure 2.2-3 and Figure 2.2-4 are for XC2000 series LCAs. Interconnect units in XC3000 and XC4000 are similar to XC2000, but more sophisticated.

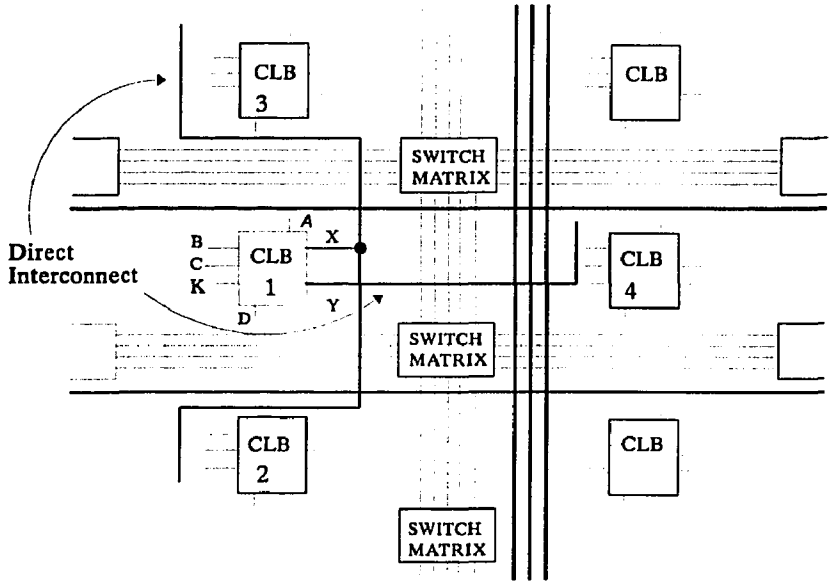


Figure 2.2-5 Direct interconnect (Source: [Xil92])

Long lines shown in Figure 2.2-3 cross the interconnect area both vertically and horizontally. Each vertical interconnection column has long lines and global

lines. The vertical long lines cross over the chip from one side to the opposite and carry signals that must travel a long distance or must have minimum skew among multiple destinations. Global lines are used, as an alternative, for a single signal to all B and K inputs of logic blocks for their clock signals. A horizontal Longline can drive a vertical Longline in each interconnection column by using a buffer.

For a signal from one logic block to the block's nearest neighboring blocks it is not reasonable to travel on either general purpose interconnect or Longlines. Direct Interconnect is designed for passing a signal from one block to its adjacent blocks. As shown in Figure 2.2-5, direct interconnect is the most efficient way to perform the logic network. Output signal X of CLB 1 rides direct interconnect wiring to CLB2, CLB3, and CLB4 without using general purpose interconnect and Longlines. This reduces traffic congestion on both general purpose interconnection and Longlines.

2.2.3 I/O Blocks

User-configurable I/O blocks (IOBs) surround the array of CLBs, which

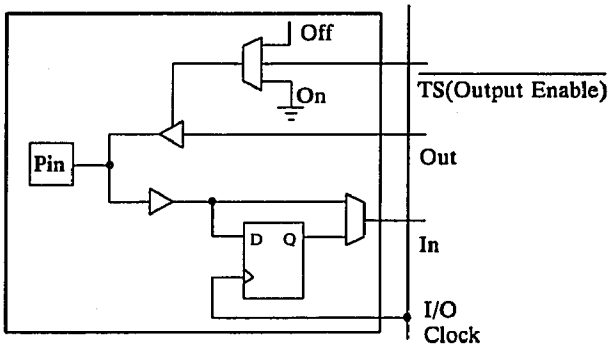


Figure 2.2-6 I/O block (Source: [Xil92])

provide an interface between the external package pins of the device and the internal logic. Both the input path and the output buffer in each I/P block are programmable. Figure 2.2-6 shows the general structure of the I/O block. The input buffer of each I/O block translates external signals applied to the package pin to internal logic levels. The buffered input signal drives both the data input of an edge-triggered D-type flip-flop and one input of a two-input multiplexer. The output of the flip-flop provides the other input to the multiplexer. Selection of either the direct input path or the registered input is controlled by the programmed memory cell. The output buffer gets an output signal from some logic block's output only. The output buffer is controlled by configuration memory cells. The two multiplexers in the figure are for program-controlled multiplexers.

CHAPTER 3

RESEARCH ON TECHNOLOGY MAPPING OF LUT-BASED FPGAS

3.1 Earlier Research on Technology Mapping of LUT-Based FPGAs

Designs of circuits using look-up table based FPGAs follow general CAD circuit design methods. An initial logic specification is first optimized using a logic synthesis tool. Based on this optimized logic network, a technology mapper is used to map logic nodes of the logic network into the logic blocks of a specific FPGA device. After technology mapping, the logic network becomes a *logic-block network*. The following placement phase assigns each mapped logic block to a physical position on the FPGA chip. The final connection of logic blocks is completed during the routing phase. The result from routing can then be used to configure the FPGA devices (see Figure 3.1-1).

Figure 3.1-1 FPGA Design Steps

FPGA technology is more complicated than either conventional PLDs or MPGAs. A logic block in a LUT-based FPGA chip contains both combinational

function generators and flip-flops. This look-up table technology gives a circuit designer more opportunities to implement multiple-input gates, but traditional standard cell library technology is not helpful in LUT. A look-up table can implement various functions depending on the number of its inputs. A 4-input look-up table can compute 2^{2^4} functions. A 5-input look-up table can compute total 2^{2^5} different boolean functions, which is more than 4×10^9 boolean functions. If one function in a cell library needs about 1,024 bytes storage area, then it needs more than 1,000 gigabytes to store all possible 5-input boolean functions in a cell library. The traditional cell library to implement so many logic combinations is not practical in present time. In addition, the position of a look-up table in an FPGA chip is fixed. The connection utility is also limited and much more restricted than general ASIC circuit design. Therefore, FPGA technology attracts many researchers to study and to find new design tools for it.

There have been many papers published concerning technology mapping of LUT-based FPGAs. Some of this earlier research can be found in references [FraR90], [ChuS91], [Cart91], [ErcM91], [Kar91], [MurS91a], [MurS91b], [PedB91], [BhaH92], [CheC92], [ChuR92], [ConD92], [KunD92], [Woo92], [ConD93], [Kar93], [MurB93], [SawT93], [CheW94], [ConD94], [HauB94], [HaqM94], [SchK94], [HabX95], [PanL96] and [XuH96].

Research on LUT-based FPGA technology mapping branches has taken place in three major directions: area-driven, time-driven, and routing-driven algorithms. Area-driven algorithms focus on minimizing the use of FPGA chip area to implement

a given circuit. The measurement of area is counted using the number of logic blocks to implement a circuit. As an intermediate phase, the area may be measured using the number of function units, both look-up tables for combinational parts and flip-flops. Time-driven algorithms consider minimization of time-delay as the most important factor during the design. Since, during technology mapping phase, the information on placement and routing is neither clear nor complete, it is difficult to predicate the actual time-delay on the final configured circuit using FPGA. A usual method used to estimate the time-delay is counting the length of the critical path in the equivalent mapped block network. Routing-driven algorithms consider the routability of the circuits designed using FPGAs. During technology mapping, the mapper considers the reduction of the links between blocks within an FPGA chip to make the final routing simpler and more easily implemented.

3.2 A Depth Optimization Mapping Algorithm

Among many technology mapping algorithms for combinational circuits using LUT-based FPGAs, Cong's Flow Map in [ConD94] is a time-driven algorithm and also the one that has been mathematically proved depth-optimal mapping algorithm. Flow Map can be applied to any K-bound Boolean network[1]. It converts a given Boolean network into a functionality equivalent network that is satisfied with *max-flow min-cut* algorithm [ForF62]. The mapper then use Lawler's clustering labeling algorithm [Law69] to find out the optimal level of the current processing node in the

[1]A *K-bound Boolean network* is a network, in which every node represents a Boolean logic gate with at most K in-degree except the source nodes, which represent primary inputs.

network. Once every node in the equivalent network is processed (labeled), Flow Map packs labeled nodes, according to their level label values, into target K-feasible look-up tables[2].

In an overall view, Flow Map works from the primary input nodes (PIs) in a given Boolean network up to the primary output nodes (POs). If a given network is not a 2-bound network, Flow Map then constructs a functionality equivalent 2-bound network using Huffman's coding algorithm [Huf52]. An auxiliary node s is introduced into the network as a source node, which has outgoing edges to all PIs of the network. All PIs are initially assigned level label 0. Then the algorithm starts to label an unlabeled node. An unlabeled node can be labeled only when all of its predecessors[3] are labeled. The current processing node is then treated as a sink node of an equivalent sub-network. This sub-network is divided by a cut (X, X'), where X' contains the sink node and X contains all the other nodes in this sub-network. The optimal depth of this sub-network depends on the optimal depth of X. A node splitting transformation algorithm is then applied to the current processing network. A node, except PIs and the PO in the processing sub-network, is split into a pair of nodes linked by a new edge. The max-flow min-cut algorithm can be used on this network. The problem to determine an optimal depth then becomes a problem to determine whether there is a maximum flow K between X and X'. The current

---

[2]If the maximal number of inputs of a look-up table is K, this look-up table is called a K-feasible look-up table.

[3]If node v has a directed path to node u, then node v is called node u's predecessor and node u is called node v's successor.

processing node is then determined its level label. The algorithm repeats the procedure until all nodes in the network that is functionality equivalent to the given Boolean network are labeled. The label value of each node is its optimal level value. The optimal depth of the given Boolean network, therefore, is determined, which equals to the level value of the PO.

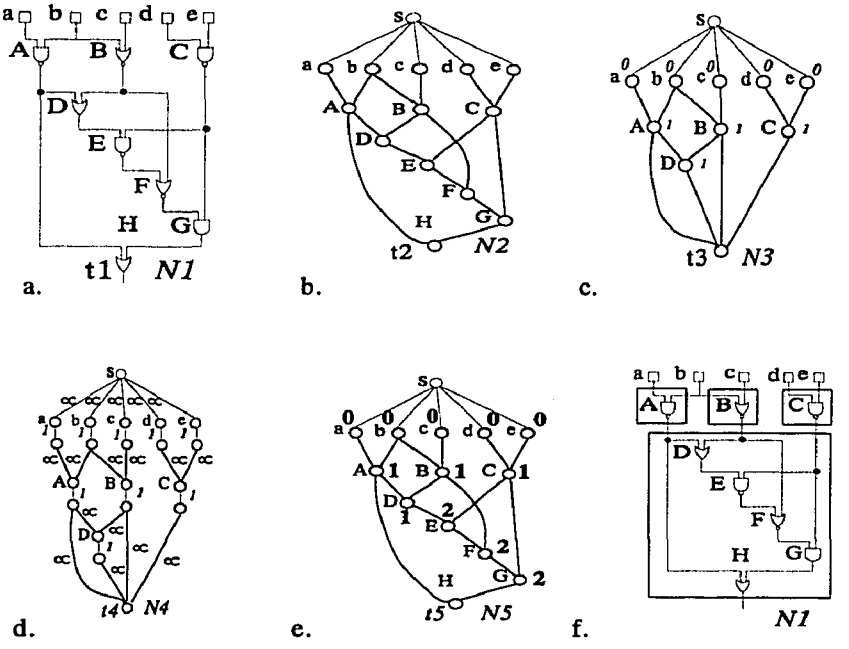


Figure 3.2-1 Labeling process in Flow Map

Figure 3.2-1 illustrates the labeling procedure used in Flow Map. In this example the target FPGA uses a 3-feasible look-up table. The sub-circuit N_t in Figure 3.2-1a is to be computed. An auxiliary node s is introduced into its equivalent network N'_t in Figure 3.2-1b. In Figure 3.2-1c, a node splitting transformation algorithm is used. New edges are assigned weight 1 and original edges are assigned weight infinite.

3.3 A Technology Mapping Algorithm for Sequential Circuits

Woo developed an FPGA technology mapping system, called the ATOM (Another Technology Mapping) [Woo92], which produced FPGA circuits for LUT-based FPGAs including Xilinx 3000 FPGAs.

The ATOM system consists of two basic modules, *reduction* and *packing*. The reduction module produces reduced networks from input networks. The packing module maps elements in the reduced network into logic blocks of a specific FPGA architecture.

The reduction module is based on the concept of "Edge Visibility" [Woo91]. The packing module is used to allocate logic blocks of the target FPGA and to assign combinational nodes and flip-flops in the reduced network to the logic blocks. One packing module works with one type of target FPGA.

The problem of reduction is converted into a problem of assigning a "visibility

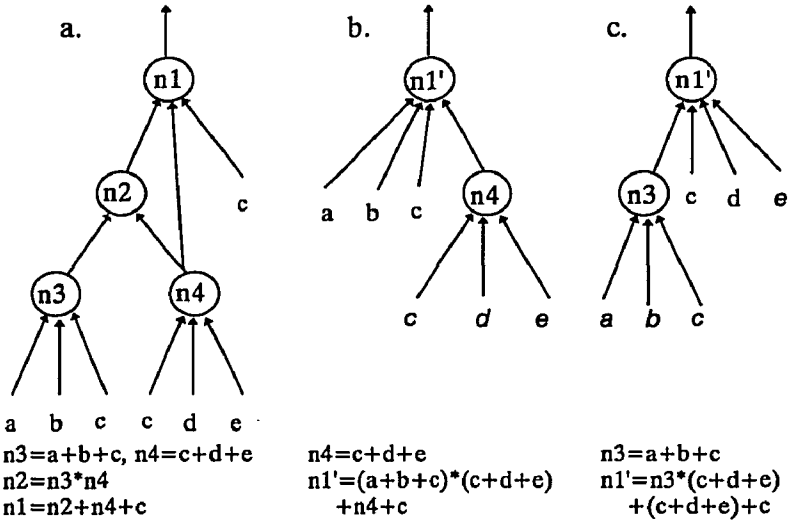


Figure 3.3-1 Example of "Edge Visibility"

value" to each edge. The concept of "edge visibility" is illustrated in the following: assume that $n_0 \xrightarrow{e} n_1$ is a pair of nodes, n_0 and n_1, linked by edge e, where n_0 is called an *fanin node* (or just *fanin*) of n_1 and n_1 is called an *fanout node* (or just *fanout*) of n_0. If e is determined to be "invisible," then this pair of nodes is collapsed into one node n_1. Edge e and node n_0 are eliminated. If any edge goes to n_0, then that edge will go to n_1 directly. If an edge is invisible, then the function computed in the fanin node (n_0) can be realized in its fanout node (n_1) since a look-up table in an FPGA can compute any boolean function depending on the number of its inputs. In Figure 3.3-1a, edges n3→n2, n4→n2, and n2→n1 are all "invisible" and are to be eliminated. Inputs to nodes n3 and n2 are then connected with node n1 directly as shown in Figure 3.3-1b. Figure 3.3-1c shows an alternative solution: edges n4→n2, n4→n1, and n2→n1 are all invisible and eliminated. Functions n1', either in Figure 3.3-1b or Figure 3.3-1c, performs exactly the same logic as the circuit in Figure 3.3-1a.

Basically, an edge linking to a flip-flop is always visible. The reduction module can be applied to the combinational parts of a sequential circuit. A sequential circuit is divided into a set of combinational networks, in each of which the source nodes are either primary inputs or flip-flops, and, fanouts of the sink nodes are either primary outputs or flip-flops. The "edge-visibility" technique is applied to each combinational network. The details of how to determine the visibility of an edge are

presented in [Woo91]. After assigning edge-visibility value to each edge, invisible edges and their associated fanin nodes are eliminated. As the example in Figure 3.3-1 shows a circuit may have multiple results from same edge-visibility scheme. Algorithms in reference [Woo91] are used to determine the best result for each combinational network.

Once the reduction operation is done, the packing operation is applied to the entire circuit. The packing operation is based on the maximum cardinality matching operation [MurN90] for combinational parts of a sequential circuit. "*Seed Circuit*" is introduced during packing to process flip-flops in a sequential circuit. The packing operation is a combination of clustering and matching. A "seed circuit" is a sub-

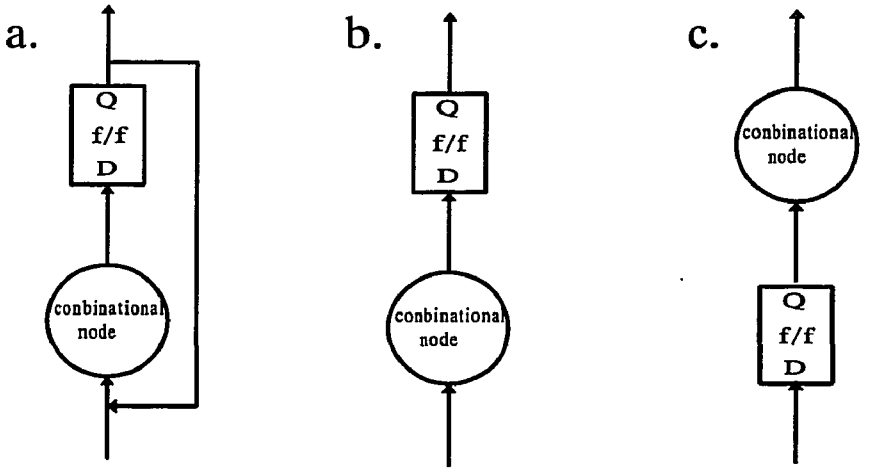


Figure 3.3-2 Types of seed Circuits

operation [MurN90] for combinational parts of a sequential circuit. "*Seed Circuit*" is introduced during packing to process flip-flops in a sequential circuit. The packing

operation is a combination of clustering and matching. A "seed circuit" is a sub-circuit consisting of a flip-flop and its combinational fanin or fanout node. Figure 3.3-2 shows three types of seed circuits used for Xilinx XC3000 packing. The flip-flops used in the packing operation are assumed D-type flip-flops with one input and one output only.

The packing operation first defines all seed circuits in a reduced network, then each seed-circuit is assigned to a new logic block. Each type of seed circuit in Figure 3.3-2 is supposed to be fully mapped into a logic lock without using any routing resource between logic blocks, that is, an entire seed-circuit must be capable to be packed in one logic block. After a seed circuit is assigned a new logic block, any nodes connected to the mapped seed circuit are to be packed into that block if possible. After mapping the fanouts and fanins of seed-circuits into the associated logic blocks, there may be some logic blocks that still have some room for mapping nodes, either combinational nodes or flip-flops. If this happens, the packing operation uses a bipartite matching algorithm in [Law76] to map some unmapped combinational nodes and flip-flops into the logic blocks, which still have spaces for some function units, until either all the blocks, which are invoked for seed-circuits, are full or there retain no more nodes to map. The bipartite matching algorithm is used here to make maximal use of functional resources in allocated blocks, in another words, to minimize the number of blocks. If there are still some unmapped nodes when all blocks associated with seed-circuits are full, new logic blocks are then invoked and the rest of the unmapped nodes are assigned to the new blocks. Finally, the packing

operation merges allocated blocks if possible.

An example in Figure 3.3-3 is used to demonstrate the reduction module and packing module in the ATOM system. It is observed that the final result from the ATOM is more efficient than the result from Xilinx's technology mapping program XNFMAP [Xil91]. Figure 3.3-4 shows the mapping result using ATOM's reduction and packing operations. The mapping result from Xilinx's XNFMAP is shown in Figure 3.3-5. It is observed that the packing operation for Xilinx XC3000 in the ATOM does not consider a seed circuit with a long feedback loop. The algorithm focuses itself on using free routing resources, which are connections between combinational function generators and flip-flops within logic blocks.

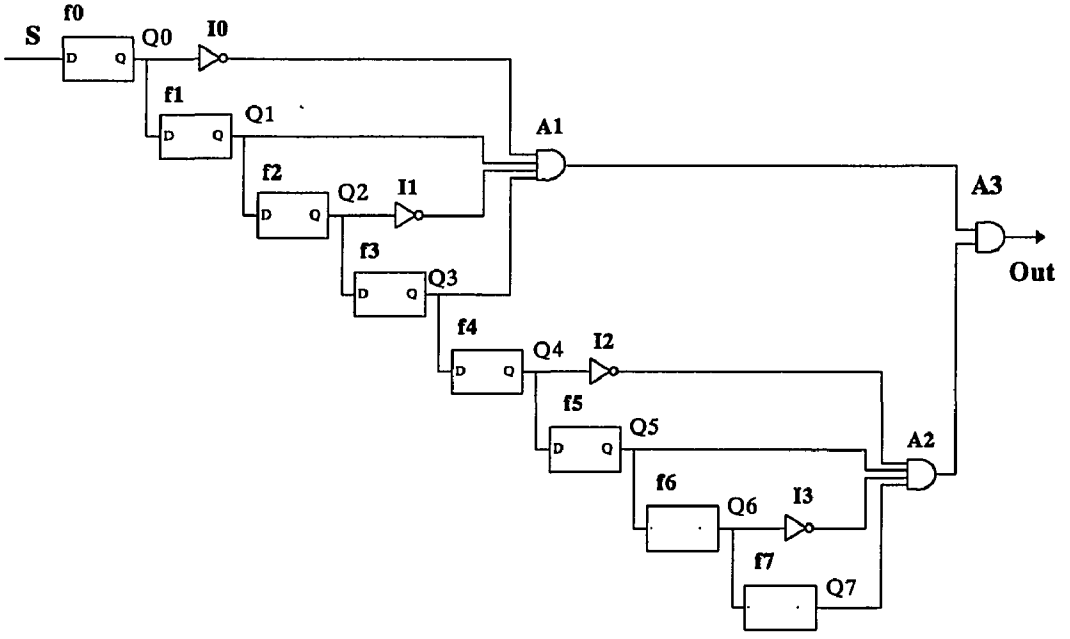


Figure 3.3-3 A sequential circuit with 8 flip-flops

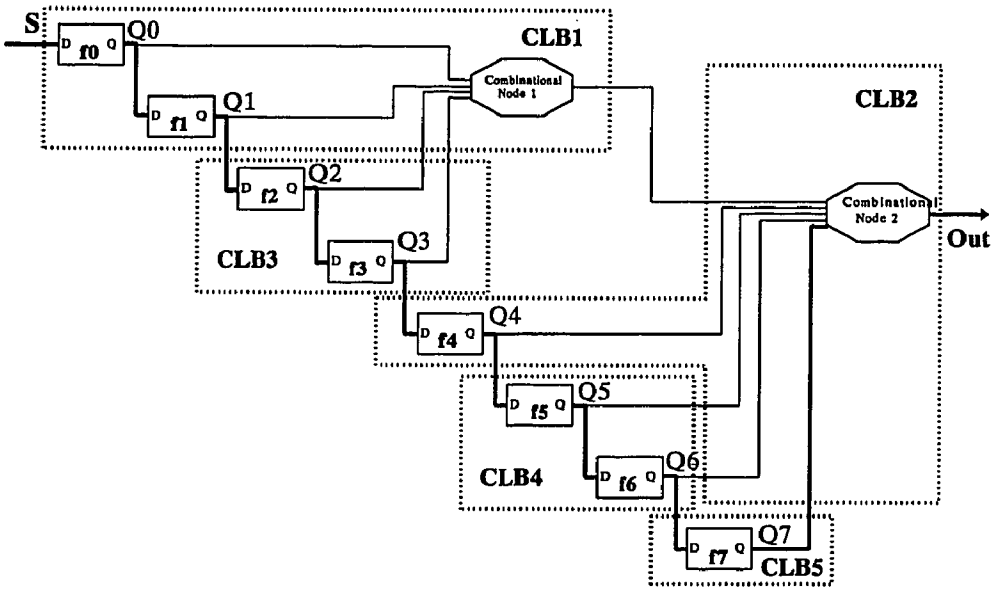


Figure 3.3-4 Mapping result using ATOM

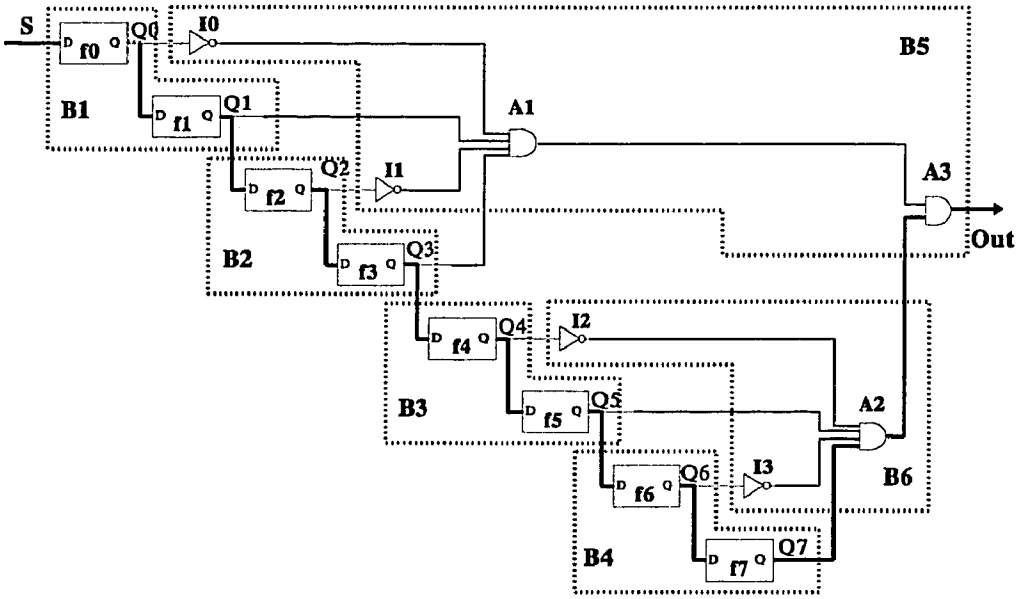


Figure 3.3-5 Mapping result using XNFMAP

CHAPTER 4

OUR VIEW OF SEQUENTIAL CIRCUITS AND PRE-MAPPING ALGORITHMS

4.1 Our view of sequential circuits

A sequential circuit consists of necessary combinational logic gates and flip-flops, which can be viewed as disjoint sets of combinational logic gates that are separated by flip-flops. A sequential circuit is a directed network in which every edge has a direction; ***primary input nodes***(PINs) are sources of the network; ***primary output nodes***(PONs) are sinks of the network and each node, except PONs, has at least one directed path to a sink.

We may call a graph, which represents a sequential circuit, an ***SCG*** (*Sequential Circuit Graph*) network. Source nodes in an SCG network represent primary input signals or input pins, while a sink node represents a primary output signal or an output pin. Any one of other nodes in an SCG is called a functional node that represents either a boolean logic gate or a flip-flop. In this thesis, all flip-flops refer to D-type flip-flops with only one input and one output. An SCG network may have multiple sinks as well as multiple sources. Sources do not have fanins and sinks do not have fanouts within the same SCG. A functional node in an SCG network may have multiple fanins and/or fanouts. Each edge in an SCG network has

a direction. To simplify, we discuss an SCG network with only one sink node. The major difference, from the viewpoint of graphs, between sequential circuits and combinational circuits is that a sequential circuit may have one or more directed circuits which correspond to some feedback signals in the circuit. In order to avoid confusion with the word "circuit", we define a ***loop*** in a sequential circuit as a path in an SCG network that starts from a node and returns back to that same node. Therefore, an SCG network may have several loops, whereas a combinational circuit does not have a loop. It is reasonable to assume that a loop consists of at least two nodes of the network. We define a ***feedback path*** of a loop as a segment of the loop which has the following features: (1) the segment starts from a node from which there is a path to a PON without passing through any other nodes in the loop; (2) the segment ends at a node to which there is a path from a PIN without passing through any other nodes in the loop; (3) all existing nodes in the segment, except the two previously mentioned in (1) and (2), must not have a path either from a PIN or to a PON without passing through any other nodes in the loop.

Figure 4.1-1 shows possible feedback path candidates. In the loop **n0→n1→n2→ n3→n4→ n5→n6→n0**, paths **n0→n1** and **n2→ n3→n4** both satisfy the conditions above.

In the event there are multiple segments in a loop which satisfy the above conditions, we use the following rules to determine a feedback path: (1) We choose the one whose start node has shortest path to the PON as the feedback path. This is shown in Figure 4.1-1. Path **n2→n3→n4** is chosen as the feedback path in the loop.

(2) If case (1) can not be determined, then we choose the path whose end node has a shortest path from a PIN as the feedback path. Figure 4.1-2 illustrates this case. Path **n0→n1** is chosen as the feedback path of the loop. (3) If both case (1) and (2) can not be determined, we arbitrarily choose any one of the segments, which qualify either case as a feedback path of the loop.

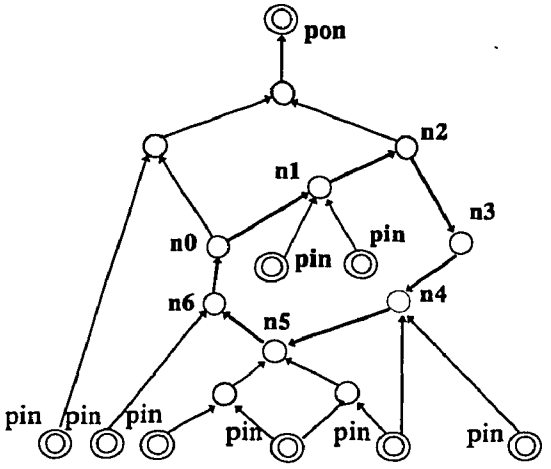


Figure 4.1-1 Loop & its feedback path

If it consists of only one edge, then the feedback path can be called a feedback edge.

Once a feedback path of a loop is defined we can then define the start node of the feedback path as the ***turning point node*** (TPN) of the loop and the end node of the feedback path as the ***starting point node*** (SPN) of the loop. In Figure 4.1-1, node **n4** is the SPN and node **n2** is the TPN of the loop. In Figure 4.1-2, node **n1** is the SPN and node **n0** the TPN.

We break a loop at its turning point by introducing an auxiliary node. The

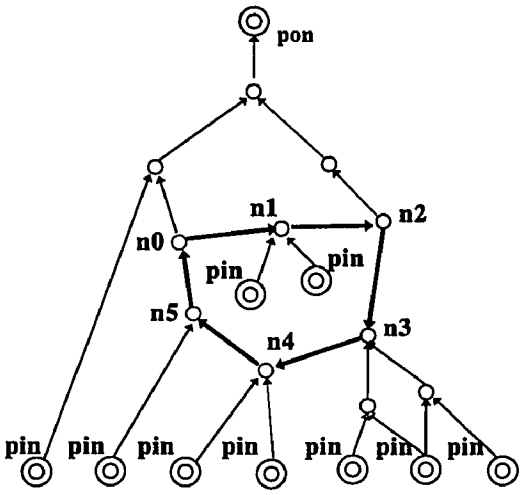


Figure 4.1-2 Determination of a loop's feedback path

auxiliary node maintains the same logic function as the turning point node has and is treated as a primary input node during technology mapping. After mapping, the position of the auxiliary node should be replaced with the original turning point node. This is illustrated in Figure 4.1-3.

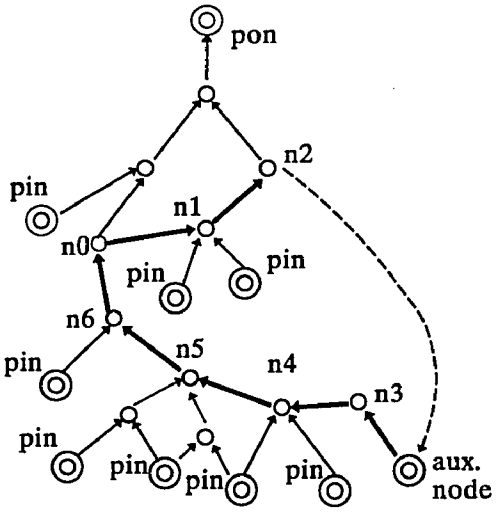


Figure 4.1-3 Loop elimination & auxiliary node

In a loop-free SCG, a ***hook*** is defined as one of the three cases: (1) a single PON, (2) a series of flip-flops (possibly one), (3) a PON and its fanin flip-flops. A network of combinational nodes with only one sink is defined as a ***cone***. The sink node of a cone is called the cone's ***top node***. If the top node of a cone **C** is connected with a hook **H**, then cone **C** is called hook **H**'s cone and hook **H** is called cone **C**'s hook. If node **n** is the top node of a cone, this cone may be called cone **n**. Therefore, if node **n** is a sink of a combinational network, then this network can be called **n**'s cone or just cone **n**. For convenience we need to define a ***dummy hook***, which consists of a null node. Similarly a ***dummy cone*** is used for convenience and likewise is null. A hook and its cone together form a ***bell***. A bell can be identified by the hook's name or the top cone's name if the hook is a dummy. Its usefulness will be seen in section 4.2. The cone of a hook in a bell can also be called the ***bell's cone***. A path from the top node of a cone to any node in the cone is a ***twig***. If a twig reaches another bell, the twig is called a ***joint***. If joint **J** in bell **B** links a bell **B~** that are **B**'s descendants, then we call **J** and **B~** a ***bell-bottom*** of bell **B**. See Figure 4.1-4 for such an illustration. If two cones have a common part, that is, two cones have a same sub-graph, the common part has to be duplicated for each of the two cones. Now a sequential circuit can be viewed as a network of bells. In a circuit with only one PON, a bell whose hook -- whether a dummy hook or not -- contains the PON is the ***top bell*** of the circuit. If this circuit is also a loop-free network, then

all hooks' cones are disjoint. That is, a sequential circuit is partitioned into disjoint parts of combinational sub-circuits. If there is a path from one bell's cone to another bell's cone, this path must pass through at least one hook or a series of flip-flops, including a dummy hook. A bell, say bell **B**, may have multiple fanouts. In such a case Bell B is a common fanin for its fanouts. From Bell B there must be multiple paths that reach some ancestors of its fanouts. Different paths starting from Bell B which meet at another bell form a cycle. To eliminate a cycle the common bell, Bell B, and its following bells must be duplicated for each of its fanouts.

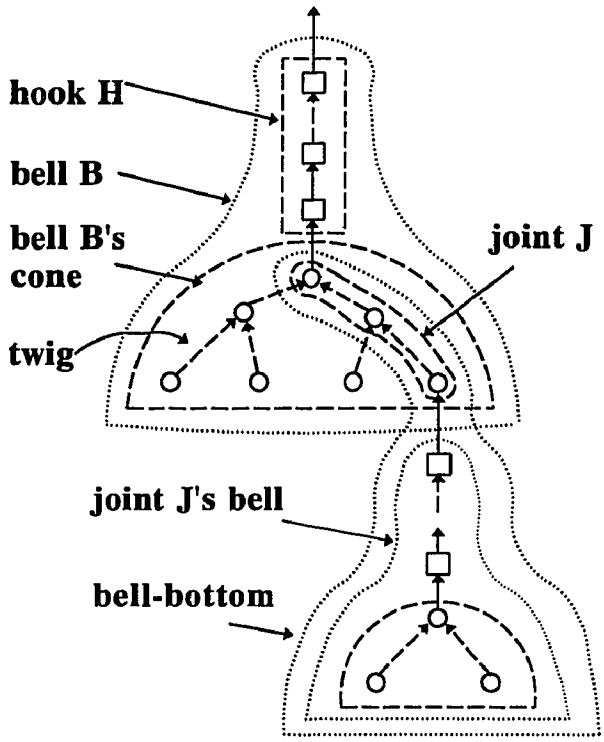


Figure 4.1-4 A bell structure

In order to describe the mapping onto look-up tables we define a ***bell-graph*** as a graph whose vertices are bells. A path in a bell-graph is a sequence of bells which

is connected by edges between bells and their joints. We also define an ***FU-graph***, whose vertices are either flip-flops or combinational logic devices or combinational function units. Here combinational function units are still some form of combinational logic devices which match combinational function units inside the logic blocks of certain FPGAs, but not the logic block itself. Similarly, a ***block-graph*** is a graph whose vertices are mapped onto logic blocks. A mapping converts an SCG network into a block-graph. During mapping, the circuit network may consist of different types of vertices: function units and logic blocks at a same time.

If it has no loops and no cycles and only one PON, a bell-graph is called a ***bell-tree***. Similarly an ***FU-tree*** is an FU-graph with only one PON but no loops and no cycles. Leaves in a bell-tree are also called ***leaf-bells***. Leaf-bells are bells without bell-bottoms in a bell-tree. Similarly a bell which has no bell-bottoms in a bell-graph is called a ***bottom-bell***. A bell in the top level of a bell-tree or a bell-graph is called a ***top-bell***. A path in a bell-graph or a bell-tree may be called a ***bell-path***. A bell-path from a leaf-bell to a top-bell or from a bottom-bell to a top-bell is called the ***major bell-path*** of that bell-graph or bell-tree. The ***length of a bell-path*** is the number of bells on the bell-path. If a bell-graph or bell-tree has several major bell-paths, then the one which has the longest length is called a ***critical bell-path*** (or a ***critical path***) of the bell-graph or bell-tree. In a block-graph, a block which has no fanin blocks is called a ***bottom-block***. The top-most block in a block-graph, which has only a primary output, is called the ***top-block*** of the block-graph. A path in a block-graph is called a ***block-path***. A block-path from a bottom-block to the top-

block is a ***major block-path*** of the block-graph and the longest major block-path is called the ***critical block-path*** of the block-graph (or just critical path if there is no ambiguity). The depth of a circuit in logic blocks is determined by the critical block-path of the block-graph after a mapping.

4.2 Method to Evaluate Circuit Depth

We further evaluate the total depth of a circuit C with multiple PONs using the terminology previously defined in section 4.1.

Assume that for a sub-circuit C_m with PON_m after the function unit mapping every bell's cone is mapped onto a set of combinational function units with optimal depth OpD using a depth optimal algorithm, such as Flow Map, and every series of flip-flops is mapped onto a set of flip-flop units. We define the ***height of bell*** **B**, H(**B**), in logic blocks as the following:

(1) $H(\mathbf{B}) = \max \{OpD, L(\mathbf{BB}_i), _i = 0, 1, \ldots, j\} + NLB$,
where OpD is the optimal depth of bell **B**'s cone, which is the number of function units of the longest path of the cone and $L(\mathbf{BB}_i)$ is the length of bell-bottom$_i$ of bell **B**. Assume that bell **B** has up to j bell-bottoms. If bell **B** does not have a bell-bottom, then $L(\mathbf{BB}_0)$ is 0. NLB is the number of new logic blocks which are introduced by bell **B**'s hook.

Since a depth optimal algorithm is used on pure combinational circuits, and each mapped unit occupies one logic block, the depth in logic blocks is the same as the depth in function units. The optimal depth of sub-circuit $C(\mathbf{B})$ should not be greater than H(**B**), where $C(\mathbf{B})$ is a sub-circuit whose top-bell is bell **B**.

We denote **BB** as a bell-bottom of bell **B** and L(**BB**) as the length of bell-bottom **BB**. We have set the following equation:

(2) $L(\mathbf{BB}) = L(\mathbf{J}) + H(\mathbf{B}_j)$,
where L(**J**) is the length of the joint **J** in bell **B**, which links to the bell-bottom **BB** and $H(\mathbf{B}_j)$ is the height of the bell $\mathbf{B}_j$ of the bell-bottom **BB**, which is linked by joint **J** with the super bell **B**.

Therefore, the height of bell B defined in formula (1) and (2) represents the length of a critical block-path of sub-circuit $c(\mathbf{B}_m)$, where bell $\mathbf{B}_m$ is the top-bell of the sub-circuit c_m. The depth of the sub-circuit c_m with PON_m, $Dp(c_m)$, is:

(3) $Dp(c_m) = H(\mathbf{B}_m)$,
where $H(\mathbf{B}_m)$ is the height of top-bell $\mathbf{B}_m$.

Then the depth of the global circuit c with multiple PONs, $Dp(c)$, is:

(4) $Dp(c) = \max \{Dp(c_m),\ _m = 1,.., n\}$,
where n is the number of total PONs in the circuit c.

The optimal depth of a global circuit, $Dp(c)_{optimal}$ then must satisfy:

(5) $Dp(c)_{optimal} \leq Dp(c)$.

Figure 4.2-1 and 4.2-2 illustrate the method used to evaluate the depth of a circuit. Bells are surrounded by thicker dotted lines, while cones are enclosed in thinner dotted lines. Figure 4.2-1 shows a sub-circuit in a bell format. The small

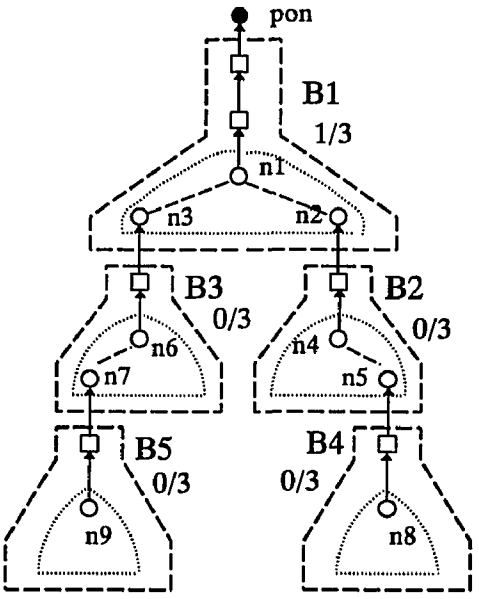


Figure 4.2-1 A bell network of functional nodes

squares in the figure represent flip-fops and the circles the combinational logic nodes. The top-bell **B1** contains two bell-bottoms: one consists of bell **B2** and joint **<n2→...→n1>** and the other consists of bell **B3** and joint **<n3→...→n1>**. While bell **B2** contains a bell-bottom: bell **B4** and joint **<n5→...→n4>** and bell **B3** contains a bell-bottom: bell **B5** and joint **<n7→...→ n6>**. Assume that after using a depth optimization algorithm and block mapping, each bell's cone is mapped onto a block-graph. Except flip-flops in bell **B1**, which use an additional logic block, all flip-flops in other bells do not evoke any new blocks. A pair of numbers shown near each bell indicates the depth of blocks (or number of function units for combinational logic nodes) after mapping. The number in front of the "/" is the value of NLB and the number behind of the "/" is the value of OpD of the bell's cone. The mapping results are shown in Figure 4.2-1.

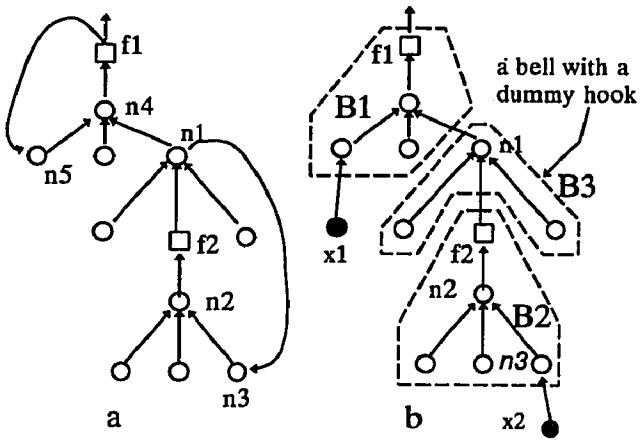


Figure 4.2-2 A bell network of blocks

In Figure 4.2-2, small squares represent mapped logic blocks which are grouped in bells. Block **b1** is the new block evoked by flip-flops in bell **B1**. We evaluate the depth of the sub-circuit from the bottom bells, **B4** and **B5**, up to the top-bell **B1**. Since bell **B4** is a bottom-bell, it does not have any bell-bottom and the length of bell-bottom for bell **B4** is then zero, that is , L(BB) = 0 in **B4**. Its NLB (number of logic blocks evoked by flip-flops) is also zero and its OpD (optimal depth of a cone after using an optimization mapping algorithm) is three. Therefore, the height of bell **B4**, H(**B4**) = 3. Bell **B2** has a bell-bottom: bell **B4** and joint **<b8→b7>**. The length of the joint is two. We denote this bell-bottom $\mathbf{BB}_{\text{b7-BB4}}$. The length of the bell-bottom, L($\mathbf{BB}_{\text{b7-BB4}}$) = 2 + 3 = 5. The OpD of **B2** is three. **B2**'s hook does not use new logic blocks in addition to the blocks evoked by its cone. The largest value between L($\mathbf{BB}_{\text{b7-B4}}$) and OpD is five. Therefore, H(**B2**) = 5. Applying the same method to bell **B5** and bell **B3**, we get the results: H(**B5**) = 3 and H(**B3**) = 6. We can now evaluate H(**B1**). **B1** have two bell-bottoms with different heights:

five and six. The length of the joint in both bell-bottoms of bell B1 is the same, which is four. We pick the largest sum as the height of **B1**. Therefore, H(**B1**) = 10. This is also the minimum up-bound of the optimal depth of the sub-circuit.

If we count the critical block-paths of the block-graph from the bottom-blocks in Figure 4.2-2, we find that from blocks **b22**, **b23**, **b24**, **b26**, and **b27** up to the top-block **b1** respectively the lengths of critical block-paths are the same. Furthermore these critical block-paths are the longest ones in this block-graph compared to other critical block-paths, such as paths from **b14** or **b19** up to the top-block **b1**. This demonstrates that $Dp(C_m) = H(\mathbf{B}_m)$.

4.3 Preparatory Algorithms

To apply mapping algorithms to a sequential circuit, some preparatory work must be done so that we can start a mapping algorithm on a simple network. The following algorithms convert a general sequential circuit, a directed network with multiple primary outputs and loops, into a set of 2-bound network with one primary output. Conventional combinational circuit technology mapping algorithms and mapping algorithms on flip-flops can be applied to each 2-bound loop-free network respectively. To get an entire mapping result we combine mapping results from all these sub-circuits.

4.3.1 The sub-circuit with only one primary output node

A sequential circuit may have more than one primary output. It is necessary for us to detect each sub-circuit which has only a single PON. The following

algorithm is used to find such every sub-network.

```
Algorithm FinSubNet:
While there is any PON_i in SCG network N not marked
        NODE ⇐ PON_i;
        Call procedure ProcessNode (NODE);
        Duplicate a sub-graph of SCG network N_i which consists of all marked
                nodes and edges;
        Remove all marks from the nodes and edges of SCG network N, except
                PONs;

Procedure ProcessNode (nodetype NODE);
If NODE is not marked & NODE is not a PIN then
        Mark N;
        While there is any in-edge_j of NODE not marked
                Mark in-edge_j;
                N ⇐ the node where in-edge_j comes from;
                Call procedure ProcessNode (N);
Else If NODE is a PIN & not marked yet then
        Mark NODE;
        Put NODE in PinList of sub-graph N_i;
Else;   /* do nothin                    */
        /* NODE is marked already      */
```

There may be some parts of the circuit common to several sub-circuits. These parts need to be duplicated for each of the sub-circuits, which includes the common parts. Once every such a sub-circuit is processed, the whole circuit is a combination of the results of sub-circuits.

4.3.2 The Loop-free network

In a one-primary output network, we search for loops. Once a loop is found, we eliminate it. There are algorithms in Johnson [Joh75] and Mateti [MatD90] to determine loops in a network.

To eliminate loops in an SCG network, we introduce an auxiliary node X for loop L. Let X keep track of the loop L's TPN (turning point node). Move the feedback edge or feedback path from the TPN to X. The TPN also must keep track

of X, so that the feedback edge, or feedback path, can be restored with the TPN and the auxiliary node X can be removed after the mapping. During combinational circuit mapping, using the Flow Map algorithm for the example, X is treated as a primary input node in the sub-circuit of each bell's cone. See Figure 4.3-1.

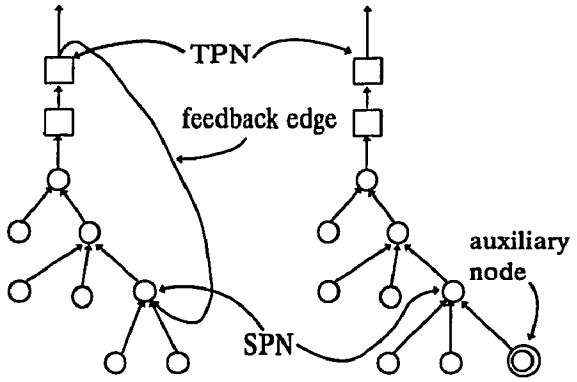


Figure 4.3-1 Elimination of a loop

The following is an algorithm used to find and eliminate a loop in a circuit, simpler than the algorithms used to find loops in a directed graph in [Joh75] and [MatD90]. In the algorithm below a loop is eliminated or broken by simply introducing an auxiliary node to duplicate the first node, which is discovered twice in the searching path, once the loop is found. The pseudo codes of the algorithm to find a loop in a given circuit network is listed in the following:

```
Algorithm -- Loopdriver
        node ⇐ PON;
        Loop(node);

Procedure -- Loop (nodetype node)
        path ⇐ path + {node};
        for each n =node.fanin[k] /* the kth fanin of node */
                if n in path then
                        x ⇐ newnode;
                        node.fanin[k] ⇐ x;
                        totalnode ⇐ totalnode + {x};
                        x.fanout ⇐ node;
```

```
                    x.TPN ⇐ n;                  /* keep track of TPN */
                    nodetab[n].aux ⇐ x;/* keep track of aux. node */
            else Loop (n);
path ⇐ path - {node};
```

The above algorithm does not use the rules for TPN and SPN as defined in section 2. To use the rules to determine TPN and SPN, the following procedure is added into the algorithm once we find a loop and introduce an auxiliary node, x, to replace the determined TPN.

The following is an algorithm to determine the starting point and the turning point of a loop as well as the feedback path of the loop.

```
Procedure DetLoop
DetLoop (pathtype loop)
1.      loopath ⇐ loop;  /* built as working space */
2.      i ⇐ 0;
3.      if loopath is not empty then node ⇐ pop(loopath);
4.              if node has a direct path to the PON then
                        feedback_path(i).start ⇐ node;
                else if node has a direct path from a pin then
                        feedback_path(i).end ⇐ node;
                else;   /* do nothing */
5.              nextnode ⇐ pop(loopath);
6.      if nextnode < > node and next node < > nil then
7.              if nextnode has a direct path to the pon then
                        feedback_path(i).start ⇐ nextnode;
                        goto step 5;
                else if nextnode has a direct path from a pin then
                        feedback_path(i).end ⇐ nextnode;
                        i ⇐ i +1;
                        goto step 3;
                else goto step 5;   (* skip this node *)
8.      /* searched up all nodes in loopath */
        if (distopon(feedback path(j).start)
                        = min {distopon(feefback_path(k).start,
                        0 <= k < i} and j ∈ [0, i), j < > k) then
                goto 11;
9.      if (distopin(feedback path(j).end
                        =min {distopin(feedback_path(k).end,
                        0 ⇐ k < i} and j ∈[0, i), j < > k) then
                goto 11;
10.     else j <= m, where both j and m satisfies
                distopon(feedback_path(t).start) =
                min{distopon(feefback_path(k).start, 0 <= k < i}
```

```
                    and
                    distopin(feedback_path(t).end) =
                    min{distopin(feefback_path(k).end, 0 <= k < i}
                        where t ∈ [0, i);
11.     TPN ⇐ feedback_path(j).start;
12.     STN ⇐ feedback_path(j).end;
```

In a one-PON circuit we examine the paths from the PON down to the PINs, thereby uncovering each bell in the circuit. The purpose of doing this is to divide the circuit into a set of disjoint parts. Two special cases have to be specified to determine a bell. If a combinational node **n** is also the TPN in a loop, node **n** is defined as the top node of a bell's cone no matter what the fanout of node **n** is. If the fanout of node **n** is a combinational node rather than a part of a hook we define a dummy hook, which links node **n** with **n**'s fanouts. In this case cone **n** and the dummy hook form a bell **n**. We do this because node **n** has multiple fanouts, one of which is, or leads to, the SPN of the loop. There must be a port to output the function generated by node **n**. If node **n** is merged with some of **n**'s fanouts to fill a function unit, then there is no way to get the input signal for the feedback path from node **n**.

Figure 4.3-2 is a demonstration of this case. In this figure, there are two loops in the sub-circuit: loop **f1→n5→ n4→f1** and loop **n1→n3→n2→f2→n1**. Node **f1** and node **n1** are two TPN nodes in the loops respectively. We introduce two auxiliary nodes **x1** and **x2** for the two loops respectively, and show them in darkened circles in part b of Figure 4.3-2. Note that node **n1** has two fanouts: **n4** and **n3**, both conbinational nodes. Node **n3** and node **n1** are in a same loop. Furthermore, node **n3** and node **n4** belong to two separate cones, one under hook **f2** and the other is

under hook **f1**. If node **n1** is grouped with node **n4** in the same cone, after mapping it may not be possible to find the function generated by node **n1**, since node **n1** may then be combined into one combinational function generator with its fanouts in a logic block. Therefore, the sub-circuit is defined as a sequence of three bells as shown in the figure: bell **B1** whose hook is flip-flop f1; bell **B2** whose hook is flip-flop **f2**, and; bell **B3** whose hook is a dummy hook and all of its nodes are combinational nodes. A depth optimization algorithm must be applied to each of these three bells separately.

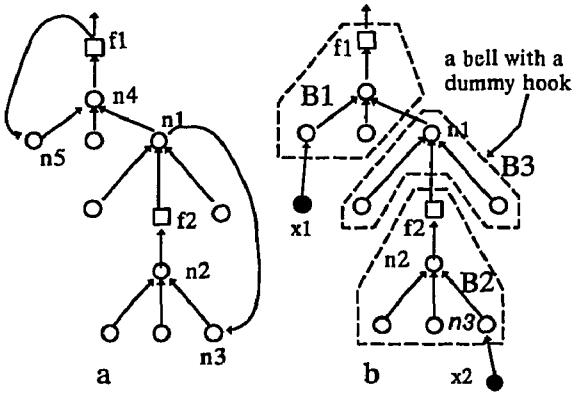


Figure 4.3-2 A bell with a dummy hook

Similarly if a flip-flop **f**, is a TPN within a flip-flop series, say **fs**, **f** must be separated from **fs** when we are redefining a circuit as a bell-graph. Flip-flop **f** will be selected as the top item in a hook. The remainder of **fs**, including flip-flops from the fanout of **f** forward to the top-most flip-flop of **fs**, form a hook of a bell with a dummy cone. A top item of a bell represents the final function of the bell, so in a bell-graph it produces an input signal of its fanout bell. Therefore, this top item and

its corresponding mapped unit is treated as a PIN for its fanout bell when its fanout bell is processed during mapping.

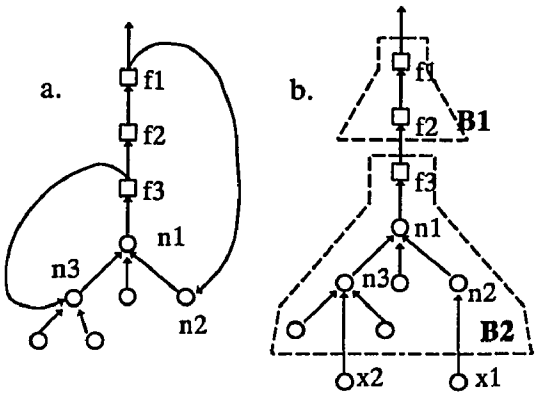


Figure 4.3-3 A bell with a dummy cone

Figure 4.2-3 shows a sub-circuit, having a hook consisting of flip-flops, **f1**, **f2**, and **f3**. Both **f1** and **f3** are TPNs of two overlapped loops: loop **f1→n2→n1→f3→ f2→f1** and loop **f3→n3→n1→f3**. We identify flip-flop **f3** as a top-most flip-flop in the hook of bell **B2**, which guarantees that after mapping there must be an output port of a block which exports the same output function as flip-flop **f3** does. If flip-flop **f3** is processed with flip-flops **f2** and **f1** during mapping, **f3** may not get the output port of the block, which **f3** is mapped onto, to output its function. This is shown in part b of Figure 4.2-3. Bell B1 consists of flip-flops f2 and f1 and a dummy cone.

4.3.3 The 2-bound network

We convert an SCG network into a *2-bound network* (a network in which each

node has at most two fanins). The earlier work on technology mapping used decomposition of nodes with multiple-fanins during mapping process. FPGA technology mapping is actually a packing process and intuitively packing smaller gates (with not more than K fanins) into a K-feasible LUT is easier and less space wasteful. The algorithm presented in Chen [CheC92] decomposes multi-input gates (DMIG) using an idea which can be found in Huffman's algorithm in [Huf52] to construct minimum redundancy codes. It is also proved in [CheC92] that the depth of a 2-bound network obtained by using DIMG algorithm is only a small constant away from the original depth of a given logic network.

Since we use one-input D type flip-flops, the conversion should be performed only on the combinational parts of circuits, that is, the cone parts of all bells. So we focus the conversion on a combinational circuit without loops, that is, a one-sink, loop-free network. A detailed pseudo code of the conversion algorithm is shown in the following:

```
Algorithm--Convert: Converting a Loop-Free one sink Network, N_i, into a 2-
                    bound Network, N_i'
sink ⇐ sink node of N_i;
findlevel (sink);
conv2 (sink).

Sub-algorithm--findlevel (sink)
P_Node ⇐ {p | p = <pin_i→...→Node>, path from a primary input node pin_i to
                    Node};
Node.level ⇐ max { length of p_k, p_k ∈ P_Node; if there is no path from pin_k to
                    Node, length of p_k is − ∞};
Q is a queue,
Q ⇐ [N_0≤N_1≤..≤N_s | N_j ∈ N_i and N_m.level ≤ N_t.level, where 0 ≤ m ≤ t ≤
                    s]
while Q ≠ ∅ do
        Node ⇐ delete(Q);
        if Node is not 2-bounderized then conv2 (Node).
        Sub-algorithm conv2(Node) to convert each interior node with its fanins
```

```
        into a 2-bound network (actually                    a binary tree)
Put all fanins of Node in a queue q, that is
        q ⇐ [n_0≤n_1≤..≤n_s | n_j is a fanin of Node and n_m.level ≤ n_t.level, where
        s + 1 = number of Node's fanins and 0 ≤ m ≤ t ≤ s]
if |q| = 1, return;
while |q| ≠ 0 do
        Anode ⇐ delete(q);
        Bnode ⇐ delete(q);
        Newnode.fanin[0] ⇐ Anode;
        Newnode.fanin[1] ⇐ Bnode;
        Newnode.level ⇐ max {Anode.level, Bnode.level} + 1;
        N_i ⇐ N_i ∪ {Newnode};
        Newnode is a new fanin of Node;
        adjust number of Node's fanins;
        adjust queue q:
        q ⇐ [n_0≤n_1≤..≤n_s | n_j is a fanin of Node and n_m.level ≤ n_t.level,
            where s + 1 = number of Node's fanins and 0 ≤ m ≤ t ≤ s];
Node.fanin[0] ⇐ Anode;
Node.fanin[1] ⇐ Bnode;
Node.number_of_fanins ⇐ 2;
Node is 2-bounderized.
```

The "while loop" inside the sub-algorithm conv2 will be executed n-1 times, where n is the number of fanins of Node of the number of in-edges of Node. For the whole network N_i, the total execution time of "while loop" in conv2 is n - v, where n is the total number of edges and v is the total number of interior nodes and the sink node in conv2.

Once combinational sub-circuits are redefined as a 2-bound network of bells with no loops, we can apply a depth optimization algorithm onto every such sub-circuit. We apply the mapping algorithm to circuits from bottom-bells up to top-bells. Actually the optimization algorithm is only applied to each bell's cone (including dummy cones) in the circuit. Since the Flow Map algorithm guarantees finding the optimal depth of a combinational circuit, finding the optimal depth of the whole sequential circuit is greatly related to the depths of series of flip-flops and their adjacent parts. Therefore, mapping of hooks must now be considered.

CHAPTER 5

ALGORITHMS OF HOOK MAPPING TO FIND OPTIMAL DEPTH OF SEQUENTIAL CIRCUITS

In this chapter, we discuss mapping of hooks in an SCG network in order to minimize the depth of a sequential circuit. We use Xilinx XC3000 series [Xil92] as our target FPGA model.

We use two steps to map function nodes in an SCG network onto logic blocks. First we map SCG network nodes onto function units, either combinational function units or flip-flops, which are equivalent to function units in logic blocks. Then we map these function units onto logic blocks, that is, some mapped function units may share a logic block but in other cases one logic block may be occupied by only one mapped function unit. Mapped combinaitonal function units are equivalent to combinational function generators in our target logic blocks. Mapping of flip-flops in an SCG onto flip-flops in our target logic block is actually performing no mapping at all but rather rewriting the reference. In the second step, mapped function units, both combinational function units and flip-flops, are packed into logic blocks. If continuous mapped function units, either combinational function units or flip-flops, can be packed into one logic block, they must be one of the following: (1) a flip-flop and its fanin or fanout mapped combinational function unit, (2) a flip-flop and its both

fanin and fanout combinational function units, (3) continuous mapped flip-flops, (4) a group of continuous flip-flops and its fanin or fanout combinational function units, and (5) a group of continuous flip-flops and its both fanin and fanout combinational function units. Because each mapped combinational function unit has been mapped onto a maximal number of logic nodes using an optimal depth algorithm, no two continuous function units can be packed into one logic block in the further mapping procedure. Therefore, the first step for combinational logic nodes requires using an optimal depth technology mapping algorithm for combinational circuits. What must be done for flip-flop parts of a sequential circuit is to pack consequential flip-flops as many as possible into the same logic blocks, or to pack flip-flops into a single logic block, which has been mapped onto a combinational function unit, also referred to as the fanout or the fanin of the flip-flop series.

In the following sections, we discuss mapping algorithms on a sub-circuit c_1 with only one primary output. We call the set of algorithms about the mapping of hooks discussed in this chapter HookMap or HMap for short.

5.1 Mapping of Flip-Flops in a Hook

A LUT-based FPGA basically has two types of function units inside a logic block: combinational logic generators and flip-flops. We map nodes onto function units prior to mapping them onto the logic blocks of a given FPGA technology. For a combinational circuit the depth of its equivalent FU-graph is the same as the depth of its equivalent block graph. Generally after an optimal depth mapping, a mergence

of the logic blocks, which represents two sub-circuits within the same bell, will not reduce the depth of the logic blocks. This implies that only mergence of two logic blocks, representing two adjacent function units in two adjacent bells respectively, may reduce the depth of a circuit in logic blocks. Finding the optimal depth of the whole sequential circuit is determined essentially by the depths of a series of flip-flops, that is, the depths of hooks.

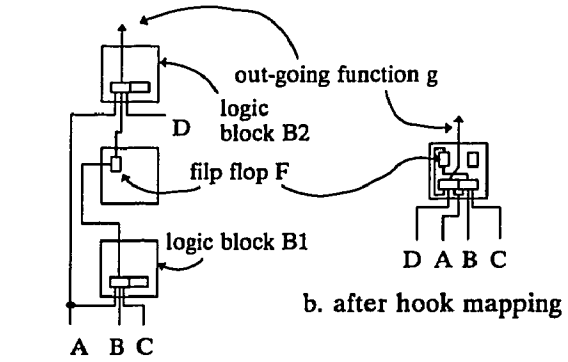
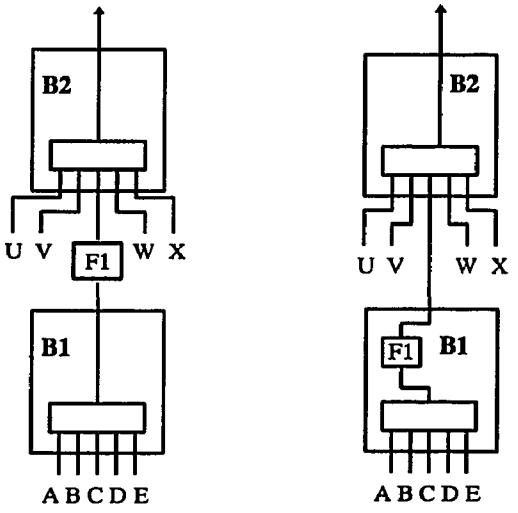


Figure 5.1-1 Hook mapping case 1

Following this, there are three major cases of hook mapping, the ideal being case 1. Illustrated in Figure 5.1-1, a hook and its adjacent mapped combinational function units can be embedded in a single logic block. For XC3000 model this may happen only under the condition that the hook has one flip-flop and both adjacent combinational function units are small enough so that they can share the combinational function generator of a single logic block. In this case, the corresponding logic block is fully filled after mapping. The length of mapped logic blocks, which represent the hook H and its two adjacent combinational function units,

is one rather than two. Hence the number of logic blocks on the path, which contains the hook **H**, is reduced by 1, compared to the number of logic blocks used after combinational function unit mapping. This is illustrated in Figure 5.1-1. In figure 5.1-1a, two logic blocks, **B1** and **B2**, are used after combinational function unit making. After hook mapping in Figure 5.1-1b, these two blocks and the one-flip-flop hook are packed into a single logic block.



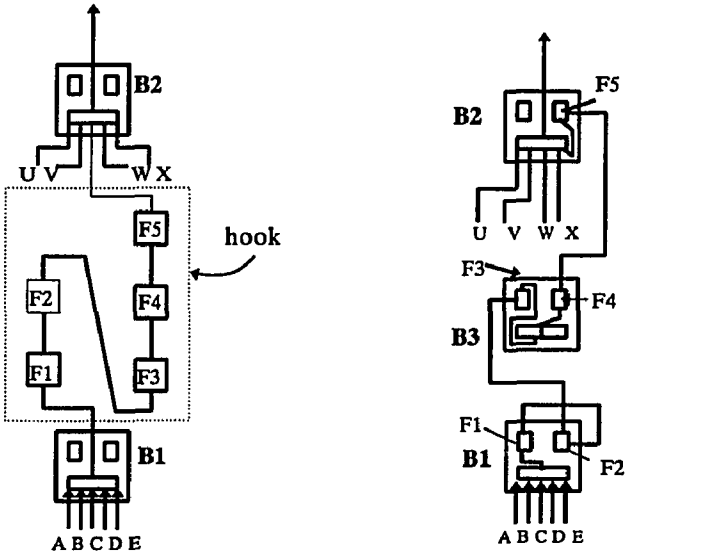
a. before hook mapping b. after hook mapping

Figure 5.1-2 Hook mapping case 2

Figure 5.1-2 demonstrates that in the second case a hook can be completely absorbed in one or both of the adjacent logic blocks. It is therefore not necessary to create a new logic block for any flip-flops of the hook. In this case the number of the logic blocks, including the hook and its adjacent combinational function units, is two, the same as would result from combinational function unit mapping.

In the third case, a hook's flip-flops can not be absorbed completely in its

adjacent logic blocks. Some number of new logic blocks must be introduced for the remainder of the flip-flops. In this case after mapping the hook, the length of logic blocks, including the hook and its adjacent combinational function units, will be increased. Figure 5.1-3 illustrates this situation.



a. before hook mapping b. after hook mapping

Figure 5.1-3 Hook mapping case 3

We use mathematical notation to describe the hook mapping situation. We denote c as the capacity of a single logic block in a target FPGA technology used. It depends on (1) the number of functions which can be implemented in a logic block, (2) the number of distinct inputs for each function, and (3) the maximal number of distinct inputs which the entire logic block can have. The notation $c=u_1+u_2+...u_m$ means that m combinational logic functions, each of which has volume u_i ($i = 1, ..., m$), fit sufficiently in the logic block with capacity c. Notation $c > u$ means that the logic block with capacity c implements a combinational logic function with volume u

and still has some capability to implement other combinational logic function(s). We also denote p as the maximal number of flip-flops, which a single logic block can implement. Assume that a series of flip-flops, **F**, consists of n flip-flops. Its two adjacent combinational function units have volume u_1 and u_2 respectively and are mapped onto logic blocks, $\mathbf{LB}_1$ and $\mathbf{LB}_2$, respectively. Logic block $\mathbf{LB}_1$ can implement k_1 flip-flops and $\mathbf{LB}_2$ can implement k_2 flip-flops. Keep in mind that we defined k_1 and k_2 separately from p because they may be different from each other. Originally k_1 and k_2 are equal to p if logic block $\mathbf{LB}_1$ and $\mathbf{LB}_2$ have not been mapped onto any function units, either combinational functions or flip-flops. However, in some FPGA technology a signal passing from one flip-flop to another must first pass through the combinational function generator in the logic block. When the combinational function generator of a logic block is full, the logic block will not possess enough capacity to implement flip-flops as well. In this case the ability of the logic block to implement flip-flops is less than p. Therefore $k_1 \leq p$ and $k_2 \leq p$. In cases in which logic block $\mathbf{LB}_1$ or $\mathbf{LB}_2$ have been mapped onto a different number of flip-flops, k_1 and k_2 will not be equal. Then the total number of logic blocks required to implement the flip-flop series **F** and its adjacent combinational function units, NLB, is:

(1) Case 1: if $0<n<k_1$ or $0<n<k_2$ and $c=u_1+u_2$ then
NLB = 1;
or
Case 2: if $0<n\leq k_1+k_2$ then
NLB = 2;
or
Case 3: if $n>k_1+k_2$ then

NLB = $\lceil (n-k_1-k_2)/p \rceil + 2$, where $\lceil x \rceil$ is the least non-negative integer not less than x.

If signals between flip-flops inside a logic block pass freely without necessarily passing through some combinational function units and/or these flip-flops are always mapped onto logic blocks which have not been mapped onto other hooks, then both k_1 and k_2 will be equal to p. In such cases, NLB will be:

(2) Case 1: $0 < n < p$ and $c = u_1 + u_2$;
NLB = 1;
or
NLB = $\lceil n/p \rceil$, where if $x < 2$, then $\lceil x \rceil = 2$; otherwise, $\lceil x \rceil$ is the least integer not less than x.

5.2 Algorithm 1 -- Straight Mapping of Flip-Flops

A simple and straight forward way to pack flip-flops onto logic blocks is to pack flip-flops onto their nearby logic blocks, which are already mapped onto their adjacent combinational function nodes. If there are some flip-flops that can not be packed onto the adjacent logic blocks, then new logic blocks are employed for these flip-flops. During the packing, paths from one block to another travels forward to the top adjacent combinational function mapped logic block. This means that an output from one block never goes back to its predecessors and itself. Because such a path will produce a loop in the network. The following is the pseudo code for this algorithm.

Algorithm 1 -- A Straight Mapping of Flip-Flops

1. for each bell
   1.1 mapping the cone onto combinational logic function units using an optimal depth combination function mapping algorithm;
   1.2 mapping the hook onto flip-flops;
2. Packing all Mapped Function Units
   2.1 packing the lower end flip-flops onto their adjacent combinational function mapped logic block as much as possible without generating a path to the block itself;
   2.2 if still some flip-flops unmapped yet, packing the unmapped top end flip-flops onto their adjacent combinational function mapped logic blocks as much as possible without generating a path to the block itself;
   2.3 while there are still some flip-flops unmapped, invoking a new logic block and packing unmapped flip-flops, from bottom to top, as much as possible onto the invoked new logic block;

5.3 Algorithm 2 -- Mapping of Loops

In a sequential circuit with loops even though we eliminate loops by introducing auxiliary nodes, these auxiliary nodes are eventually replaced by the original turning point nodes in the loops. Therefore, in the final result the loops still exist in the circuit. Depending on different mapping strategies the results about loops may differ from each other. The research shows that the time delay between two logic blocks or between a logic block and a connection box is greater than the time

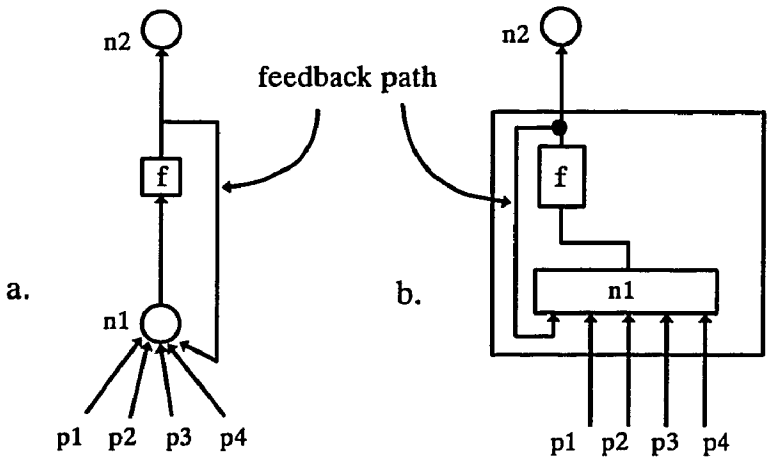


Figure 5.3-1 Simple case of loop mapping

delay within a logic block [BroF92]. Therefore, it is meaningful to pack one loop completely into a single logic block if possible.

It is possible that we can embed flip-flops of a hook in logic blocks which are mapped on combinational function units not adjacent to the hook and still have sufficient room. Even though this may reduce the number of logic blocks to be used, it can not reduce the level of logic blocks during mapping. It would be desirable that an entire loop including all its nodes and its edges be mapped onto a single logic block. If a loop is small so that the TPN is a flip-flop which forms a hook and the SPN is the top node of the hook's cone, it is expected that the adjacent combinational function units of the hook use as small an area as possible in a single logic block. In this way the entire loop can be embedded in the same logic block.

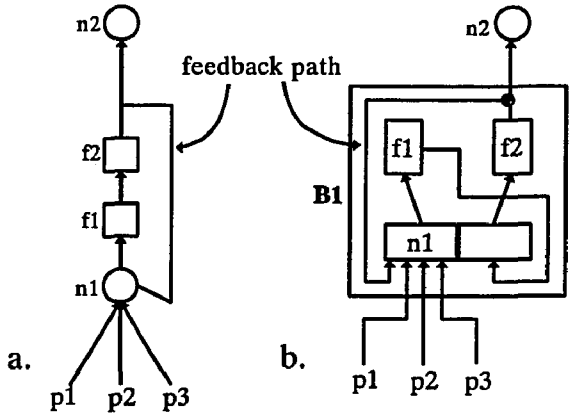
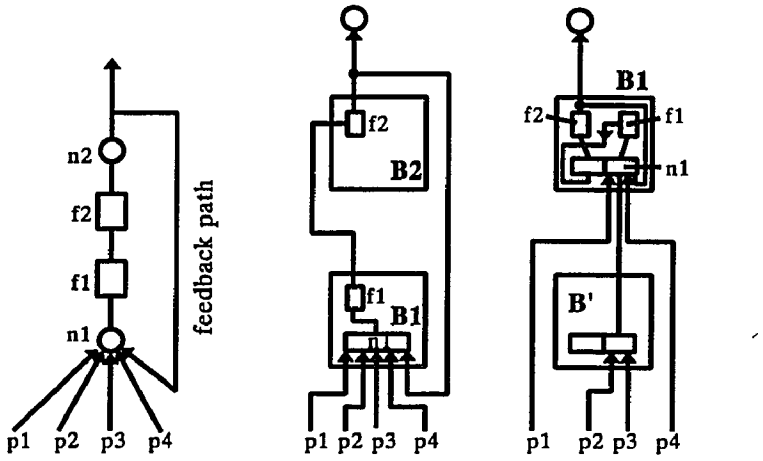


Figure 5.3-2 Two-f/f in loop mapping

The ideal thing is that an entire loop is embedded in a single logic block. Figure 5.3-1 shows a simple case, in which a hook has only one flip-flop and its following top c.f.u. has the full number of fanins. In our examples the maximal

number of fanins of a logic block for its combinational function generator is five. Using algorithm 1 the entire loop can be embedded in one single logic block.

Figure 5.3-2 shows another simple case, in which a hook has two flip-flops and its following c.f.u. has fewer fanins (not the full number of fanins). The loop can still be embedded in one logic block without making any change on any previous c.f.u. mapping using algorithm 1.



a. b. c.

Figure 5.3-3

We consider cases, in which after using algorithm 1 a hook's top flip-flop is the TPN of the loop and its following c.f.u. with full number of fanins is the SPN of the loop. The circuit and the mapping result from algorithm 1 are illustrated in Figure 5.3-3a and Figure 5.3-3b respectively. We can not pack the whole loop, the devices and the wire, in one logic block without changing the following c.f.u. If we want to pack entire loop in one block, then we must make some change on the c.f.u.,

that is, splitting the c.f.u. into smaller c.f.u. units. Figure 5.3-3c shows such a solution that embeds two flip-flops and a feedback path and the starting point node in one single logic block. We observe that this solution breaks down the original c.f.u. into two separate logic blocks, B1 and B2. This does not affect the previous c.f.u. mapping in the parts below fanins p1, p2, p3, and p4. The total number of blocks used is still two, which means that the result does not affect the depth of the circuit. However, the loop now is completely inside a logic block. No any parts of the loop can be found outside the two logic blocks.

Let us examine more details on time delay estimation. Assume that signal propagation in each function unit, either combinational logic or flip-flop, takes m units of time; n units from a port of a logic block to a function unit or vise versa or one function unit to another; q units between ports via connection lines outside logic blocks. We denote by T the time delay from fanins pi to the fanout of the TPN and T' the time delay of a complete loop. Therefore, the time delay for each result in Figure 5.3-3 is the following:

Figure 5.3-3b: $T_1 = 3q + 5n + 3m$, $T'_1 = 2q + 5n + 3m$;
Figure 5.3-3c: $T_2 = 3q + 7n + 5m$, $T'_2 = 4n + 4m$.

Assume that after every q units of time a signal reaches its fanout. In Figure 5.3-3b when the next signal reaches **B1**, it must wait for the feedback signal from the loop, because the loop takes more than $2q$ time delay. Therefore, both Figure 5.3-3c and Figure 5.3-3b use same number of logic blocks. Figure 5.3-3c has however greater advantages than Figure 5.3-3b.

If a loop is too large to be covered by a single logic block, there will be multiple logic blocks required to implement the loop and the loop can not be eliminated in the routing after mapping. After function unit mapping, we can recover a loop by moving its feedback path from the auxiliary node **X** to its original TPN and then delete node **X** from the entire network. In a final mapping solution, if a loop is eliminated, it means that the loop can not be seen between logic blocks but will be found between function units inside a block.

The following algorithm considers mapping loop first, then use almost the same algorithm as Algorithm 1 to deal with the remain of the circuit.

1. Deal with loop first
   - 1.1 mapping each hook onto flip-flops;
   - 1.2 if the top flip-flop of a hook is the TPN of the loop, then pack this flip-flop in a new logic block. And from this flip-flop down to the SPN of the loop if all nodes can be mapped onto this block, then it succeeds; otherwise, it fails. If the top flip-flop of a hook is not the TPN of the loop, then map this flip-flop into a new block and if all ancestors of this flip-flop up to the TPN of the loop can be mapped onto the block, then map them; further more if all descendants of this flip-flop can be mapped onto the same block, then it succeeds; otherwise, it fails. If the loop mapping fails, then release the called logic block and try other loop till all loops tried either succeeds or fails;
2. for each bell repeat the following steps till every element of all bells is mapped:
   - 2.1 if the hook is not mapped onto one logic block, then apply algorithm 1 to this bell;
   - 2.2 if the hook is mapped already, then apply an depth optimization mapping algorithm for combinational circuits to the rest part of the cone of the hook;

5.4 Algorithm 3 -- Special features of certain FPGAs

In some cases two consecutively mapped combinational function units can be packed together in one single logic block. This is true when the XC3000 series is

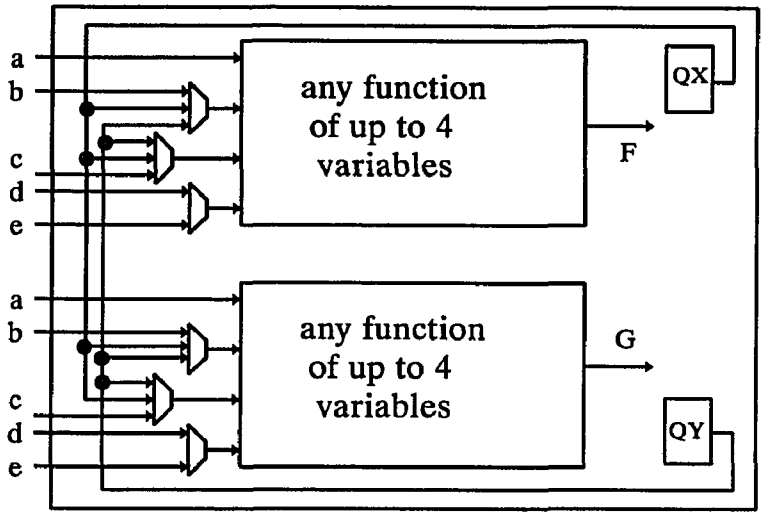


Figure 5.4-1 FG-Mode of XC3000 (Source: [Xil92])

used. When we apply a optimal depth combinational function mapping algorithm to a cone, a K-feasible policy is used. This means that a mapped combinational function unit may have at most K different inputs. For XC3000 the value of k is five. However, if a top-most mapped function unit of a cone can be expanded to a logic block so that the block contains an entire loop (which means the TPN of the loop is the hook of the mapped unit and the SPN of the loop is within the unit), then this logic block may be able to absorb a fanin unit using XC3000 combinatorial logic option FG in [Xil94], which is shown in Figure 5.4-1. Using option FG mode, the combinational function generator in a logic block is divided into two parts, each of which can output a 4-variable function. The entire combinational function generator may have up to seven inputs, five of which come from the outside the block and two from the two flip-flops inside the block. Each half part of the generator may share one variable, which is from outside the block, with the other half part and all the other variables different

from each other.

As discussed in section 4.2, depth of a circuit is determined based on the length of a major path and basically the length of a path is a summary of the heights of bells on the path. A mapping algorithm is applied to circuits from bottom-bells to top-bells. If a top-most mapped unit can have benefits using FG mode, then we consider FG mode prior to hook mapping case 1. Assume that on a major path there are mapped function units from lower level to higher level: **U0**, **U1**, **U2**, **f**, **U3**, **U4**. Here **f** is a flip-flop and all others are combinational function units. Furthermore, **f** is a TPN of a loop and **U2** contains an SPN of the loop. **U1** can not be packed with **U0** in same block and **U3** can not be packed with **U4** in same block. **U2** and **f** must be mapped onto one block in order to pack the entire loop in a single block. If **U2**, **f**, and **U3** satisfy hook mapping case 1, then after the hook mapping the length of the path from **U1** to **U3** is 2 in blocks. If **U1**, **U2**, and **f** satisfy FG mode, then **U1**, **U2**, and **f** can be mapped onto the same block and **U3** occupies separate single block. The length of the path is still two. This shows that using FG mode will not be worse than using hook mapping case 1. Particularly when **f** is also an output function leading to a

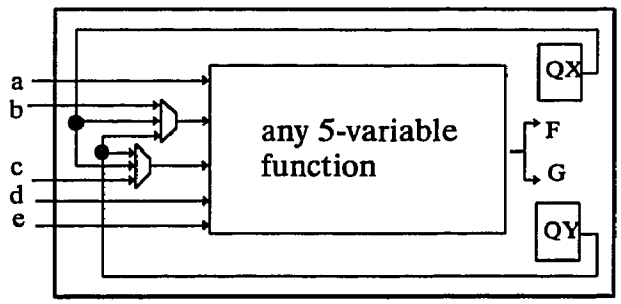


Figure 5.4-2 F-Mode of XC3000 (Source: [Xil92])

PON, we choose FG mode rather than hook mapping case 1. In order to use the benefits from FG mode, we may apply the Flow Map algorithm again to the remaining parts of the SCG network below the top-most mapped unit and use the 4-feasible policy.

Even during the merge of mapped blocks, FG mode may still result in some benefits for reduction of blocks. In some cases a combinational function unit of a block, say **Block1**, is fully used in F mode [Xil94], which is shown in Figure 5.4-2, and there is an additional opportunity to break down the combinational function generator into FG mode and another half of the generator, say Ua, producing the same function in **Block2**, then **Block2** can be deleted, all of whose fanouts are then connected with output port from **Ua** of **Block1**. An algorithm using this feature is in the following:

Algorithm 3 -- Mapping with some Special Features of certain FPGAs
1. For each bell
   1.1 mapping each hook onto flip-flops;
   1.2 packing mapped flip-flops in logic blocks;
   1.3 expanding every logic block which is mapped onto flip-flops to inclose the whole loop if possible;
2. for each bell
   2.1 if 1.3 fails then mapping cone onto combinational logic function units using Flow Map;
   2.2 if 1.3 succeeds then mapping the rest combinational logic nodes in the cone onto combinational logic function units using Flow Map;
3. Packing
   3.1 packing mapped hook and its adjacent combinational logic function units in logic blocks according to the hook mapping rules;
   3.2 checking each mapped logic block to see if it can be broken down to use the advantage of FG mode, if so then remapping and repacking the involved parts;
   3.3 packing rest of the circuit into logic blocks using neighbor rules.

5.5 Algorithm 4 -- Concerns on Placement and Routing

Research related to FPGAs is concerned mostly with FPGA architectures or

design tools. Although function units in an FPGA device are identical, so are interconnection units. Different patterns of units chosen to implement a circuit design may cause very different results in terms of resource consumption and time delay. Therefore, much research concentrates on finding algorithms which reduce the number of units used in an FPGA chip or the internal delay time of a chip. As in other VLSI designs, a process to design a circuit using FPGAs contains technology mapping, placement, and routing as shown in Figure 3.1-1.

Since during technology mapping users have less information about the final routing available, the depth of the directed acyclic graph (DAG) equivalent to the logic network of the circuit is often used as a measurement of time delay. This assumes that the greater the depth of a DAG, the longer the time delay will be. However the path in a graph does not represent the actual routine in a circuit and the actual time delay of a circuit is known only after the routing process. Assume that we have a LUT-based

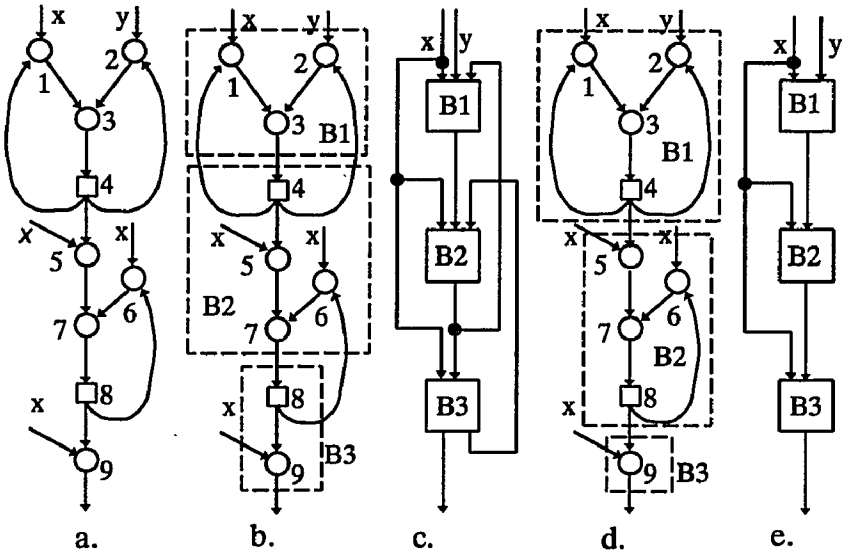


Figure 5.5-1

FPGA technology, in which each logic block can accept up to three different inputs and within the logic block there is a combinational function generator and a flip-flop, each of which can produce a logic function as an output of that logic block and an input of the other function unit in the same logic block. That is, a function unit can output up to two different logic functions. For the circuit shown in Figure 5.5-1a, ignoring the loops in the circuit and just considering the flip-flops as common combinational logic gates, we can get a depth optimal result in Figure 5.5-1b. After we get the optimal result, we replace the loops back to their original functionality (Figure 5.5-1c). Alternatively our research shows that we can pack nodes in logic blocks as shown in Figure 5.5-1d and obtain the result in Figure 5.5-1e, which has the same depth as in Figure 5.5-1c. We observed that although the circuit in Figure 5.5-1e and the circuit in Figure 5.5-1c have the same depth, the circuit in Figure 5.5-1c contains loops and the circuit in Figure 5.5-1e does not. If we consider only the depths in the graphs, the two results have the same effect. However, if we consider the routing, they may differ significantly because one has loops in the final routing arrangement and four connections between logic blocks and the other does not have loops and only two connections between blocks. Brown in [BroF92] pointed out that interconnection units have more percent of area and take longer time delay than function units do. This suggests that the circuit in Figure 5.5-1c may possess longer time delay after the routing process because it has loops and more connections between blocks in the routing and a signal passing between logic blocks may utilize more time than within a logic block. Intuitively, we see that if we can pack the flip-flop node

first in a logic block then expand the block as much as possible to include the fanins of the flip-flop node and the entire loop, then we may eliminate the loop in the final mapping graph.

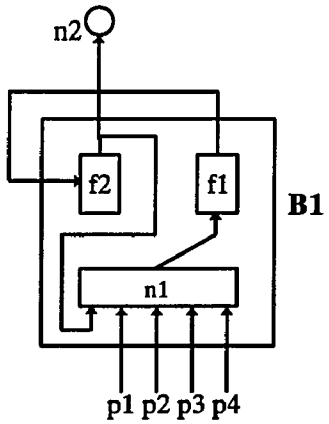


Figure 5.5-2

We noticed that in algorithm 2 embedding a loop in one logic block may not always succeed. In such cases should we still be concerned about loops? In a sequence of combinational function units, if the mapping is depth optimal, then there is no further chance to pack two adjacent combinational function units into one logic block; otherwise, the optimal mapping is violated. This implies again that to reduce the depth of the circuit we can consider only the reduction around the hooks, that is, the flip-flops and their neighboring combinational function units. Consider examples shown in Figure 5.3-3. After the mapping of flip-flops (Figure 5.3-3b) we merge logic blocks. Intuitively we may consider merging two consequential blocks first. Without breaking down the previous c.f.u. mapping, we pack flip-flops **f1** and **f2** and the c.f.u. **n1** together in one block, while the connecting line between **f1** and **f2** travels

outside the block from one of the output ports of the block to the direct data input port of the same block. Figure 5.5-2 illustrates this situation. The time delay for Figure 5.5-2 is the following.

$$T_3 = 3q + 5n + 3m,\ T'_3 = q + 4n + 3m.$$

Comparing with the results in Figure 5.3-3, we still have

$$T'_3 < T'_1 \text{ or } T'_1 - T'_3 = q - n.$$

We also find that $T'_3 - T'_2 = q - n - m$. Actually when signals p_is reach their fanout block, they may not have to wait for $q - n - m$ units of time for the arrival of the feedback signal. For simplicity assume that $n = m = 1$, $q = 20$. Then $T'_3 - T'_2 = 18$. But in Figure 5.5-2, $T'_3 = 20 + 4 + 3 = 27$. After c.f.u. in block **B1** accepts the first set of fanins, 20 time units later the next p_is reaches the c.f.u. again and 27 time units later the feedback signal reaches the c.f.u. Signals p_is wait for 7 time units. While in Figure 5.3-3c every 20 time units signal from **B'** reaches n1, a c.f.u. in **B1**; however, feedback signal reaches n1 every 8 time units. The feedback signal waits for 12 time units for p_is to arrive at the c.f.u., n1. Therefore, the result in Figure 5.5-2 is only 8 time units slower than the result in Figure 5.3-3b. The benefit of using the result in Figure 5.5-2 is that it uses only one block and need not re-map the cone of the hook. If the connection of two ports of a logic block always has a priority during placement and routing, then it may be one of the shortest paths between two ports in one chip. In that case the value of q in T'_3 represents the time of

traveling from one logic block's output port to its input port. It may be much less than an acceptable practical time delay between two different blocks. Therefore, we have $T'_3 \ll T'_1$. Since in Figure 5.5-2 the SPN and TPN of the loop are packed in one block, we call this mapping strategy S-T-1 mapping. The figure also shows that the SPN c.f.u. and its closest two ancestor flip-flops are mapped onto one logic block. From this pint of view we may call this mapping strategy S+2 mapping.

We further consider cases, in which a hook has more than two flip-flops and the top-most flip-flop is the TPN of the loop and the successive c.f.u. of the hook is the SPN of the loop. We apply Algorithm 1, Algorithm 2, S-T-1 mapping, and S+2 mapping individually to the examples shown in Figure 5.5-3 and Figure 5.5-4 respectively.

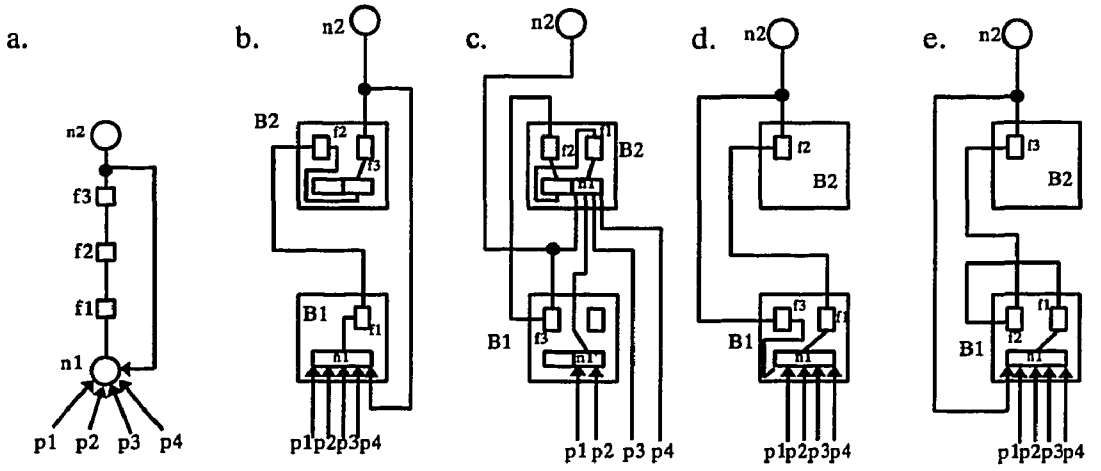


Figure 5.5-3

In Figure 5.5-3a, the hook contains three flip-flops, the c.f.u. n1 has the full

number of fanins (it is five here). The feedback signal is from the top-most flip-flop f3 to the following c.f.u. n1. The mapping resulting from the three algorithms are illustrated in Figure 5.5-3b, Figure 5.5-3c, Figure 5.5-3d, and Figure 5.5-3e accordingly. The time delay for each result is the following:

Algorithm 1: $T_1 = 3q + 7n + 5m$, $T'_1 = 2q + 7n + 5m$;
Algorithm 2: $T_2 = 4q + 9n + 6m$, $T'_2 = 2q + 7n + 5m$;
S-T-1 mapping: $T_3 = 4q + 7n + 4m$, $T'_3 = 2q + 6n + 4m$;
S+2 mapping: $T_4 = 4q + 7n + 4m$, $T'_4 = 3q + 7n + 4m$.

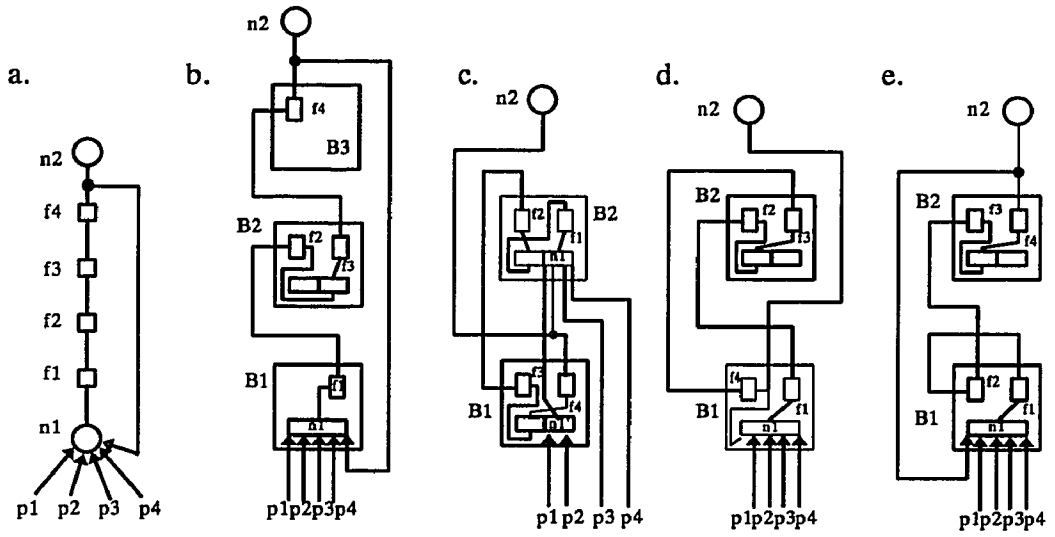


Figure 5.5-4

Figure 5.5-4a shows a hook with four flip-flops. The mapping results are shown in Figure 5.5-4b, Figure 5.5-4c, Figure 5.5-4d, and Figure 5.5-4e respectively. The time delay for each algorithm is listed below:

Algorithm 1: $T_1 = 4q + 9n + 6m$, $T'_1 = 3q + 9n + 6m$;
Algorithm 2: $T_2 = 4q + 11n + 8m$, $T'_2 = 2q + 9n + 7m$;
S-T-1 mapping: $T_3 = 4q + 9n + 6m$, $T'_3 = 2q + 9n + 6m$;
S+2 mapping: $T_4 = 4q + 9n + 6m$, $T'_4 = 3q + 9n + 6m$.

Examples in Figure 5.3-2, Figure 5.3-3, Figure 5.5-2, Figure 5.5-3, and Figure 5.5-4 imply that using logic blocks, such as in XC3000, each of which contains two flip-flops in the case a hook has an odd number of flip-flops and its following c.f.u. is full, the time delays for a loop as a result of different mapping algorithms are slightly different; but in the case a hook has an even number of flip-flops, the results are quite different. In the later case we get the following:

$$T'_3 < T'_2 \ll T'_1 \leq T'_4.$$

These examples also show that packing two flip-flops in a single block may not be the best way, and in some cases a worst way, to reduce time delay of a loop. An algorithm for mapping with concerns of placement and routning is listed in the following:

Algorithm 4 -- Mapping with Concern to Placement and Routing
1. For each hook
   1.1 mapping each hook onto flip-flops;
   1.2 packing mapped flip-flops in logic blocks;
   1.3 expanding every logic block, which is mapped onto flip-flops to enclose the whole loop if possible;
2. For each bell
   2.1 if 1.3 fails, then mapping the cone onto combinational logic function units using Flow Map;
   2.2 if 1.3 succeeds, then mapping the remaining combinational logic nodes in the cone onto combinational logic function units using Flow Map;
3. Packing all Mapped Function Units
   3.1 packing mapped hook and its adjacent combinational logic function units in logic blocks according to the hook mapping rules;
   3.2 packing the remaining mapped function units using neighbor rules.

CHAPTER 6

MAPPING OF JOINT

In the previous chapter, Hook Map considers mapping flip-flops between combinational function units. A hook may share a logic block with its following combinational nodes. In this chapter we discuss mapping combinational nodes between two hooks. Nodes between two hooks form the joint of a bell-bottom. The super hook of the joint may be a dummy hook. Algorithms developed in this chapter focus on the joint of a bell-bottom, and we call them Joint Map algorithms, or JMap for short.

Cong's Flow Map [ConD94] guarantees that every node in a boolean network has only one level value, the maximum level value is the optimal depth of the boolean network. A feature of Flow Map is that one boolean network has only one labeling system, which guarantees to reach the optimal depth of the combinational circuit. However, the result of packing labeled nodes into logic blocks is not unique. This means that there may be different mapping patterns with same optimal depth value. This is true for a connected combinational logic network but this may not be always true when a combinational logic network is only a part of a sequential circuit. Figure 6.0-1 illustrates an example of such a case. In Figure 6.0-1a, the capital letters indicate already mapped bells followed by their bell height values. Nodes are indicated by lower case letters followed by their level values in the Flow Map

labeling system. A simple packing solution is that packing all nodes in a single clique, which have same level values, into one logic block. The result using this method is shown in Figure 6.0-1b. Node a is mapped onto block A~. Node b and its clique members, d and e, are mapped onto block B~. And node c and its clique members, f and g, are mapped onto block C~. New blocks A~, B~, and C~ have their height values 8, 5, and 7 respectively. However, there are alternative ways to packing combinational nodes in Figure 6.0-1a into logic blocks. One of the alternative solutions is shown in Figure 6.0-1c: node a and node c's clique are mapped onto one logic block A~ and node b's clique is mapped onto logic block B~. Not only the second solution uses less logic blocks, but also the topmost block A~ has height value 7, which is one level less than the solution presented in Figure 6.0-1b. This fact shows that the study of joint mapping is meaningful for global depth optimization of sequential circuits.

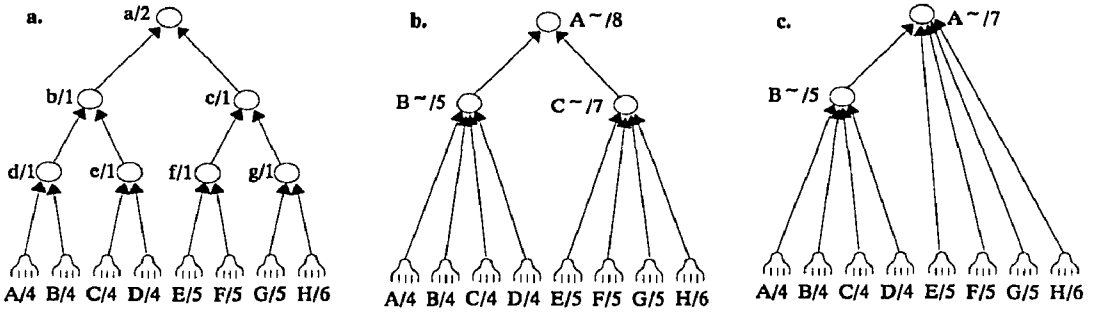


Figure 6.0-1 Optimal depth of a combinational network may not be the global optimal solution

6.1 Mapping with level fit first strategy

Based on Flow Map's labeling system a simple way to pack nodes in blocks without losing the optimal depth feature is packing an entire clique[1] into one logic block. Figure 6.1-1 is the pseudo-code for mapping combinational nodes in a cone depending on their level values, that is, connected nodes with same level values are packed into same combinational function unit. The algorithm listed in Figure 6.1-1 works on a combinational sub-circuit that has been labeled using some optimal depth algorithm, such as Flow Map. This algorithm processes nodes from higher levels down to lower levels and packs cliques into blocks one level by level. We call it **LFF (Level Fit First)** algorithm. Figure 6.1-1 lists the pseudo-code of LFF.

```
Algorithm LFF
Q ⇐ Ø;                  /* a queue of head nodes of cliques */
K ⇐ Ø;                  /* initial structure of a clique */
1.    n ⇐ top node of cone C;     /* start from top node of the cone */
2.    Q ⇐ Q + {n};
3.    while (Q is not empty) do
4.          {head ⇐ gethead(Q);          /* get first node of Q */
5.           if (head is not a source node of C) then
6.                 {searchclique(head, C, K);
7.                  invoke a new function unit F;
8.                  F.output ⇐ K.output;
9.                  F.fanins ⇐ K.fanins;
10.                 Q ⇐ Q + K.fanins; /* add new heads to Q */
                    }
            }
11.   endwhile;
```

Figure 6.1-1 A pseudo-code of LFF algorithm

In this pseudo-code assigning a function unit to an unmapped node means that

[1] A *clique* is a group of nodes in network, in which all nodes are connected and have same level values, in Flow Map's labeling system. A *head node* of a clique is a noe in the clique, to which every other nodes in the clique has a directed path.

this unmapped node can now share one function unit with its clique members, to which the function unit has been assigned. The output function of the function unit will be the output of the head node of the clique which it is assigned to. Function ***searchclique***(haed, C, K) searches all members of head's clique and stores all information about the clique in a data structure K, such as members' references, output function, fanout, and fanins of the clique.

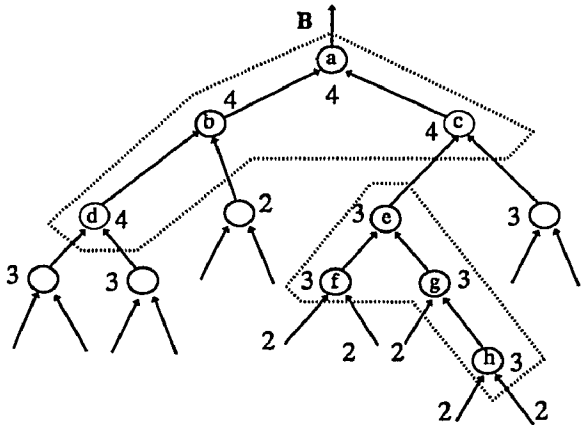


Figure 6.1-2 Example of LFF algorithm

Figure 6.1-2 illustrates the LFF mapping algorithm. Numbers in the figure show the level values of the nodes. Nodes a, b, c, and d are connected and have the same level value 4 on the top of the combinational 2-bound network. They form a clique whose head node is node **a**. These nodes are mapped onto one function unit. Meanwhile nodes e, f, g, and h are in another clique with level value 3. They are mapped onto another function unit. Mapping of other nodes are not shown in this figure.

6.2 Mapping with higher bell-bottom first strategy

The depth of a circuit is determined by its critical path. Assume that we work on bell B and map all nodes in the circuit up to the top combinational node of B's cone. The formula (1) of Chapter 4 should be used in this case:

$$(1) \qquad H(B) = \text{Max } \{OpD, L(BB_i), i = 0, 1, \ldots, j\}$$

In each bell-bottom BB_i when its bell B_i has been computed, the length of bell-bottom BB in formula (2) of Chapter 4, $L(BB) = L(J) + H(B_j)$, is determined by L(J), the depth of a bell-bottom's joint. Based on the labeling system, the length of J can be calculated by adding the number of function units that would be mapped to J using algorithm LFF. If the number of function units that exactly cover the joint J is p_j, then p_j must be less than or equal to OpD. To make a new algorithm better than the previous algorithm LFF we must find a way to map J in less than p_j units. For each bell-bottom B_i, we map its joint into K-feasible function units as less as possible. We start from a bell-bottom with the greatest sum of p_j and $H(B_j)$. We do this from the top node of the joint down to its bottom node. Once computed, a bell-bottom is reduced from the current bell. The original bell then collapses into several smaller disjoint bells. We then apply the same procedure above to each smaller bell until all nodes in the original bell B are mapped onto function units Since this algorithm maps a joint in a critical path first, we call it **Joint Fit First** algorithm, **JFF** for short. Figure 6.2-1 illustrates the process, where K is 5.

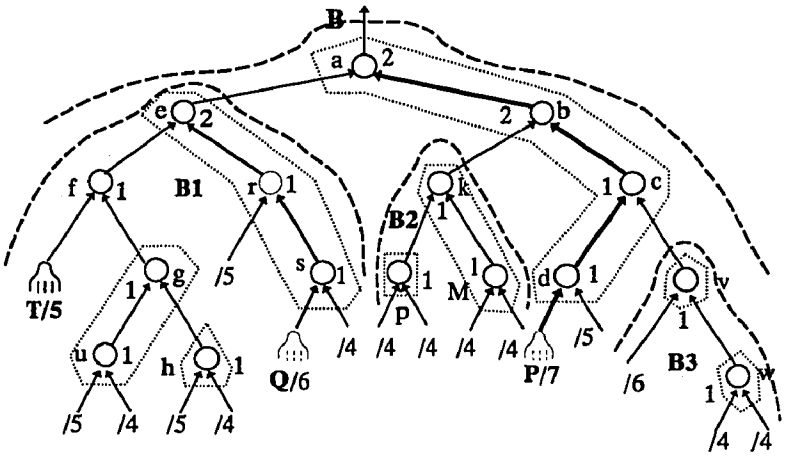


Figure 6.2-1 Example of JFF algorithm

In Figure 6.2-1, numbers nearby nodes are their level values. Numbers after "/" indicate the heights of the following bells. Jont <a-b-c-d> would have length of 2 using algorithm LFF and the associated bell-bottom would be the longest bell-bottom in the bell B. Joint <a-b-c-d> is selected for starting the algorithm JFF. After it is mapped, joint <a-b-c-d> is reduced from bell B. Bells B1, B2, and B3 are new bells defined after this reduction. They are disjoint. In bell B1, joint <e-r-s> now is the crictical path of bell B1. It is selected for starting JFF for bell B1. The algorithm JFF is then applied to the remaining bells until all nodes in bell B is mapped onto a combinational function unit (equivalent to a logic block). The final mapping solution using JFF shows that the height of bell B now is 8 instead of 9:

$$H(B0 = L(BB_Q) = L(J_{<a\text{-}b\text{-}c\text{-}d>}) + H(B_Q) = 1 + 7 + 8$$

Figure 6.2-2 shows the pseudo-code of procedure LengthJoint that computes a

joint J's length. Figure 6.2-3 shows a procedure that determines the length of a bell-bottom. These two algorithms are used when determining the longest bell-bottom in a bell. Algorithm JFF and its driver are listed in Figure 6.2-4 and Figure 6.2-5 respectively.

```
Procedure LengthJoint
1.      newJ ⇐ 0;
2.      while (queue J is not empty) do     /* J holds all nodes of joint J */
3.              { core ⇐ getq(J);
4.               invoke a new function unit U, assign core to U;
5.               while (U is not full) do      /* test K-feasibility */
6.                      {next ⇐ getq(J);
7.                       assign next to U;}
8.               newJ ⇐ newJ + 1;}          /* length of joint J */
```

Figure 6.2-2 Pseudo-code of procedure LengthJoint

```
Procedure LengthBB
1.      for every bell-bottom BB in bell B do
2.              {compute H(Bj), Bj is bell-bottom BB's bell;
3.               pj ⇐ LFF(J);
4.               L(BB) ⇐ pj + H(Bj);}
```

Figure 6.2-3 Pseudo-code of procedure to calculate length of a bell-bottom

```
Procedure JFF
1.      while (J is not empty) do
2.              {next ⇐getq(J);
3.               if (U is full) then do
4.                      invoke a new function unit U;
5.               assign next to U;}
```

Figure 6.2-4 Pseudo-code of algorithm JFF

Procedure RunJFF

```
1.    Bell set ß ⇐ {C};                    /* initial working bell set */
2.    while (ß is not empty) do
3.          {if (B ∈ ß and L(BBm) = max{L(BBi) for every bell-bottom
                                          BBi of B}) then
4.                call JFF(Jm);            /* mapping joint Jm */
5.          B- ⇐ B - BBm;                  /* reduce mBB from bell B */
6.          redefine bell structures of network B-;
                [B-] is the set of all bells of B- after redefining;
7.          ß ⇐ ß + [B-];}                 /* add new bells to bell set */
```

Figure 6.2-5 Pseudo-code of mapping driver of JFF

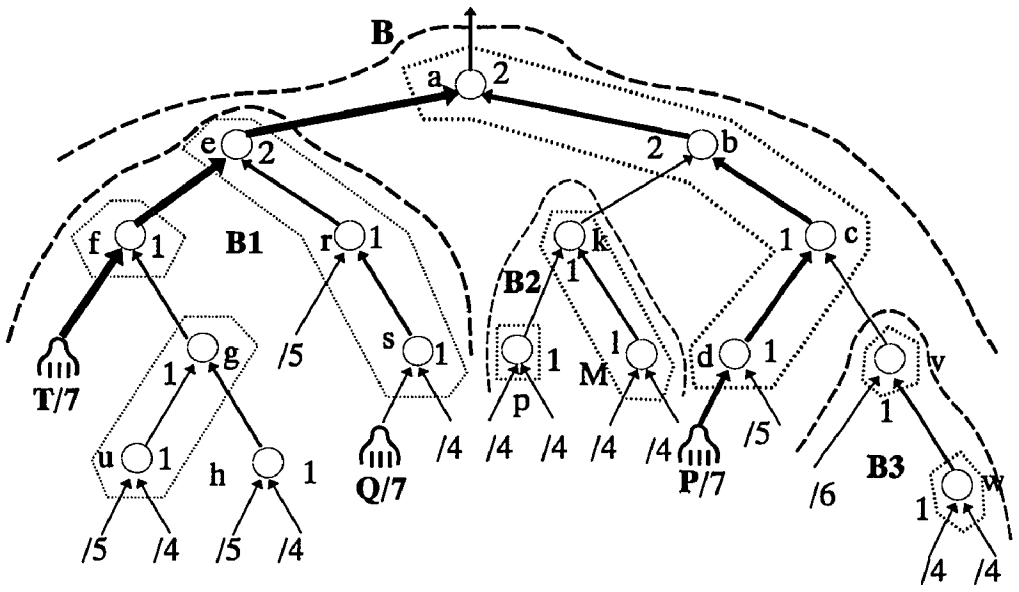


Figure 6.2-6 A worse case using JFF

It is possible that after mapping, the longest joint is longer than p_j and what may be even worse is that the longest L(BB) is longer than the sum of p_j and $L(B_j)$. In Figure 6.2-1 if the height of bell T is 7 instead of 5 and the height of bell Q is 7 either, then the mapping solution may be worse using JFF. We illustrate this in Figure 6.2-6. There are three critical paths in bell B: bell-bottom BB_P, BB_Q, and BB_T. All the three critical paths have length of nine. If joint <a-b-c-d> is first

selected for JFF mapping, then bell-bottoms BB_Q and BB_T are still critical paths in bell B1. If joint <e-r-s> is then selected for JFF mapping in bell B1, then in the rest part of bell B1, joint <f> must be selected for JFF mapping first. The final JFF mapping solution shows that the height of bell B is now ten rather nine:

$$H(B) = L(BB_T) = L(J_{<a\text{-}e\text{-}f>}) + H(B_T) = 3 + 7 = 10$$

6.3 Mapping level fit first then joint fit

A new algorithm is developed to avoid the disadvantages of algorithm JFF shown in the previous section, which break the labeling system of the cone C and results in less advantageous mappings. In the new algorithm the mapper first fits the top node of a joint and its clique members into a unit. If the unit still has some room, then the mapper assigns nodes in the rest of the joint from the top to the

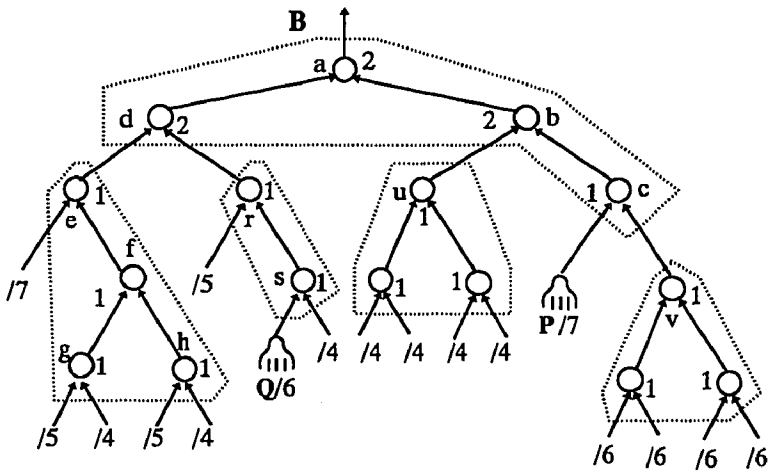


Figure 6.3-1 Example of mapping algorithm using level fit first then joint fit strategy

bottom into the same unit till the unit is full. It then applies algorithm JFF to the

remaining of the same joint, which has not been mapped yet. Since this algorithm maps nodes depending their level values first, then maps the remaining of the joint in the critical path or longest bell-bottom, we call it algorithm LFTJ for Level First Then Joint. An example illustrating this algorithm is shown in Figure 6.3.-1. Figure 6.3-2 shows the pseudo-code of algorithm LFTJ. Figure 6.3-3 shows a pseudo-code of the driver of algorithm LFTJ.

```
Procedure LFTJ
1.      wile (queue J is not empty)
2.          {core ⇐ getq(J);
3.           invoke a new function unit U, assign core to U;
4.           next ⇐ getq(J);
5.           while (level(next) = level(core) and J is not empty)
6.                  {assign next to U;
7.                   next ⇐ getq(J);
8.           if (level(next) = level(core)) then  /* J must be empty */
9.                  {assign next to U;
10.                  while (U is not full)
11.                        if (n is core's descendant & level(n) =
                                  level(core)) then assign n to U;}
12.          else    /* level(next) < > level(core) */
13.                 while (U is not full)
14.                       if (n is core's descendant & level(n) =
                                  level(core))then assign n to U;
15.          invoke a new function unit U, assign next to U;
16.          while (J is not empty)
17.                 {next ⇐ getq(J);
18.                  if (U is full) then invoke a new function unit U;
19.                  assign next to U;
                    }
             }
```

Figure 6.3-2 Pseudo-code of algorithm LFTJ

```
Procedure RunLFTJ
1.      Bell set ß ⇐ {C};
```

```
2.      while (ß is not empty)
3.          {if (B ∈ ß and L(mBB) = max{L(BB_i),
                        for every bell-bottom BB_i of B} then
4.              call LFTJ(BBm);      /* length of bell bottom BBm */
5.           B⁻ ⇐ B - BBm;           /* reduce BBm from bell B */
6.           redefine bell structures of network B⁻;
                /* [B⁻] is the set of all bells of B⁻ after redefine */
7.           ß ⇐ ß + [B⁻];           /* add new bells into bell set */
             }
```

Figure 6.3-4 Pseudo-code of mapping algorithm using LFTJ

Algorithm LFTJ listed in this section does not break the depth optimization system created by an optimal depth algorithm such as Flow Map. It is observed that for each bell, LFTJ first maps the top node of a joint and all its clique members onto one combinational function unit. From that unit down to the node at the bottom of the joint, if a clique is broken and mapped onto two or more function units, then the top part of the broken clique must be mapped onto the function unit in a higher level, which includes the fanout of the clique. As shown in figure 6.3-3, the clique (c, v, t) are broken into two parts: node c is mapped with nodes a, b, and d onto a higher level block; the lower part, v and t, is mapped onto a sigle unit. Therefore, this algorithm will not increase the depth of a bell even it may break a clique of joint nodes into two new groups. This guarantees that the mapping result from LFTJ will not be worse than the result from LFF.

CHAPTER 7

EXAMPLES

We use the Xilinx' XC3000 model as our target technology to illustrate the algorithms discussed in the previous chapters.

7.1 An 4-bit Counter

Figure 7.1-1 shows a logic diagram of a 4-bit binary counter [Xil92]. Since four flip-flops in this circuit have the same clock signal CLK, we omit CLK in future diagrams. This will not affect the mapping result.

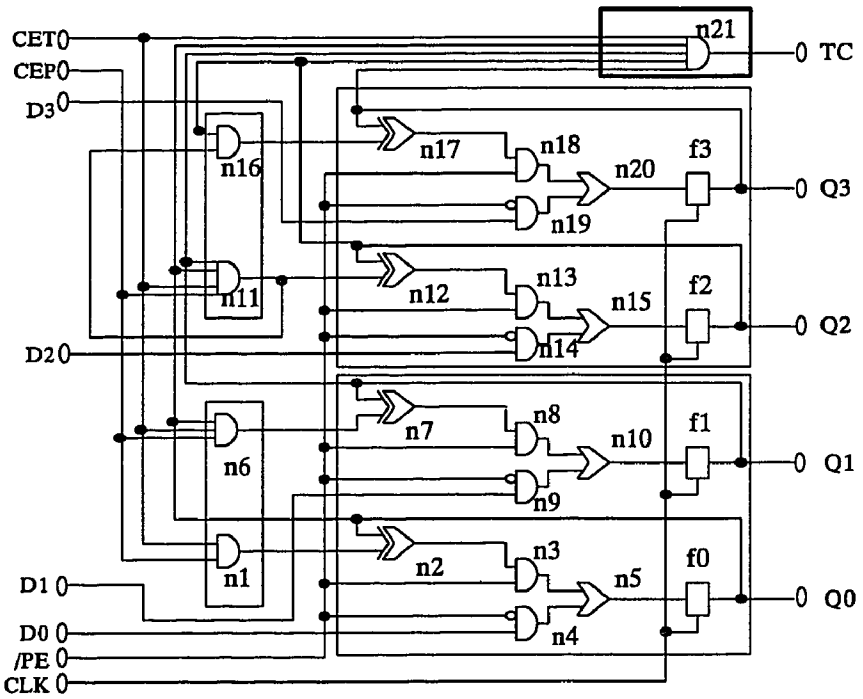


Figure 7.1-1 A 4-bit binary counter

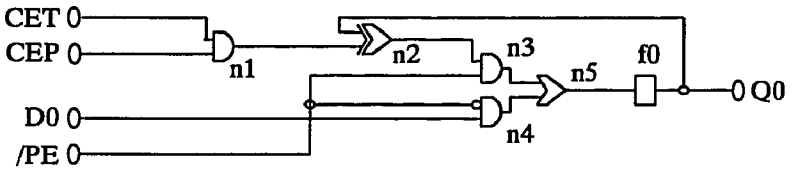


Figure 7.1-2 Sub-circuit Q0

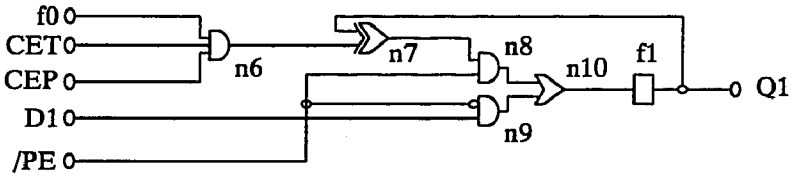


Figure 7.1-3 Sub-circuit Q1

The circuit displayed in Figure 7.1-1 has seven primary inputs (omitting CLK) and five primary outputs. Logic blocks as mapping result from [Xil92] are represented by five bold lined rectangles shown in this figure. Initially we try to find all sub-circuits, each of which has only one PON. Circuit Q0 shown in Figure 7.1-2 is one such sub-circuit. This is a single bell, whose hook consists of a flip-flop f0 and a PON Q0 (We may use same symbol to indicate both logic device and the output function of the device). However, the sub-circuit Q1 contains two bells, one is the top bell, whose hook is the flip-flop f1 and PON Q1, and the other is a bottom-bell f0, which is actually the sub-circuit Q0. The bottom-bell f0 is connected with bell f1 by the joint n6→n7→n8→n10. We deal with a circuit one bell at a time and from lower bells to upper bells. Since bell f0 is a lower bell within a bell-bottom of bell f1, we may use symbol f0 to represent the sub-circuit Q0 instead of displaying the entire sub-circuit Q0 in the diagram of sub-circuit Q1. This is shown in Figure 7.1-3.

Similarly, we find sub-circuit Q2, Q3, and TC, each of which has only one PON. Sub-circuit Q2 has a sub-circuit f1 and sub-circuit Q3 has a sub-circuit Q2. Sub-circuit TC is a bell-graph. The hook of the top bell of TC consists of only a PON TC and has four sub-circuits, Q0, Q1, Q2, and Q3, which are directly connected together at an AND gate with signal CET.

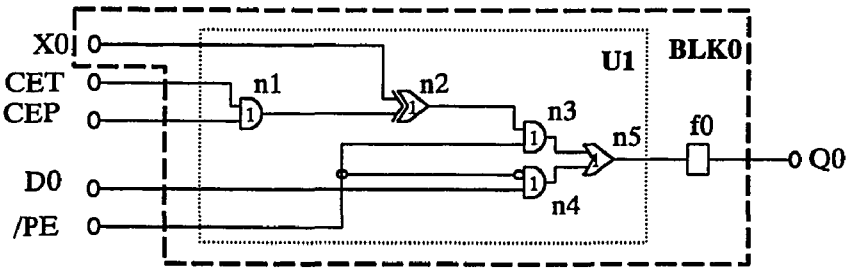


Figure 7.1-4 LUT and mapping of sub-circuit Q0

Sub-circuit Q0 is a common sub-circuit of all other sub-circuits. It is in the lowest position of the entire circuit system. Let us take a look at sub-circuit Q0 first: There are no cycles in this circuit, but there is a loop in sub-circuit Q0. The SPN of the loop is node n2 and the TPN of the loop is node f1. The feedback path is from f1 to n2, that is, a single edge f1→n2. To eliminate the loop, an auxiliary node X0 is introduced. This is shown in Figure 7.1-4. Now sub-circuit Q0 is converted into a 2-bound network and is ready for application of the Flow Map algorithm. Figure 7.1-4 shows the level labeling system of cone n5 in sub-circuit Q0, which is performed by using Flow Map algorithm. The auxiliary node X0 is treated as a PIN during level labeling. Small numbers inside of gates indicate the level of the gates. All gates in

cone n5 have the same value, 1, which means that the optimal depth of this combinational sub-circuit is 1 and all five gates, n1, n2, n3, n4, and n5, can be packed in one full size combinational function unit of a logic block. In Figure 7.1-4 the inner-dotted rectangle (U1) represents a mapped combinational function unit. Flip-flop f0 is the only flip-flop in the hook of bell Q0 and the cone of bell Q0 can mapped onto one logic block. For hook mapping of bell Q0 we combine f0 with U1 in one logic block.

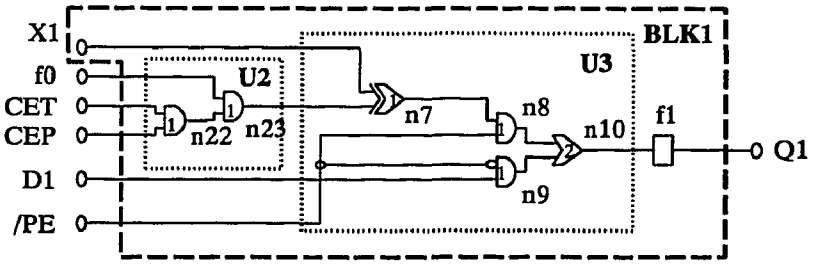


Figure 7.1-5 LUT and mapping of sub-circuit Q1

We now work on sub-circuit Q1. We introduce an auxiliary node X1 to represent the TPN f1 in the loop. There is no cycle in this sub-circuit; however, the circuit is not a binary graph as yet because node n6 has three fanins. We break down n6 into two AND gates, n22 and n23. The output function of n23 is the same as that of n6. Figure 7.1-5 illustrates the equivalent 2-bound network of the sub-circuit Q1. Next we apply the Flow Map algorithm to sub-circuit Q1. Actually the Flow Map algorithm is only applied to the top-most cone of bell Q1. The mapping result is shown in Figure 7.1-5 also. The combinational part of sub-circuit Q1 has two levels. Since the flip-flop f1 is the TPN of the loop and the SPN is node n7, we try to map

them into one logic block according to our mapping strategy. Because the hook of sub-circuit Q1 has only one flip-flop f1, we imply that if node n7 can be mapped onto the top function unit, then we may be able to pack this unit and f1 into one single logic block. There are several mapping configurations of cone n10 even though the optimal depthfrom Flow Map is only two. Now we can estimate the optimal depth of sub-circuit Q1 in logic blocks. The optimal depth from the Flow Map algorithm for the top-most cone of bell f1 is two. The height of bell f0 is one because the entire sub-circuit Q0 can be mapped onto one block BLK0. The optimal depth of cone n10 in bell Q1 is two and f1 can be mapped with the top-most function unit. Therefore the NLB of f1 (the number of new logic blocks invoked by hook f1)is 0. The length of the joint which links bell f0 to bell f1 is 2 since node n23, n7, and n8 have level value 1 and node 10 has level value 2. The height of the bell-bottom n10-n23-f0 is 3. Comparing the height of the bell-bottom n10-n23-f0 to the optimal depth of cone n10, we choose 3 and add it to the NLB of f1. The optimal depth of sub-circuit Q1 must be less than or equal to three.

We try to find a mapping configuration utilizing features of the XC3000 chips, as discussed in section 5.4, so that we may pack all nodes of cone n10 into one block or alternatively all nodes after f0 through f1 on the major path can be packed in one block. We pick node n7 into a mapping function unit as the top unit. We then expand this mapping unit until it has four inputs including one input from X1 (f1) so that this unit contains all nodes in the loop except the TPN f1. Two nodes, n22 and n23 remains after mapping function unit U3. Two inputs, f1 and n23, can be in the

same block BLK1 with U3. The other two inputs, D1 and /PE, must come from the outside of the block BLK1. Since we have already used two inputs from the inside of the block for function unit U3, at most five outside inputs may be used by the entire block, two of which must be D1 and /PE. Actually there are only three nodes left outside the block. We simply map node n22 and n23 onto another combinational function unit U2. The combinational function units, U2 and U3, fit within the XC3000 combinatorial logic generator using operation FG mode. Flip-flop f1 can be mapped together with U2 and U3 onto a single logic block BLK1. We evaluate the optimal depth of sub-circuit Q1. The optimal depth of cone n10 now is one in logic blocks and NLB of f1 is 0. The length of bell-bottom n10-n23-f0 is two. Therefore after the block mapping the optimal depth of sub-circuit Q1 is two only.

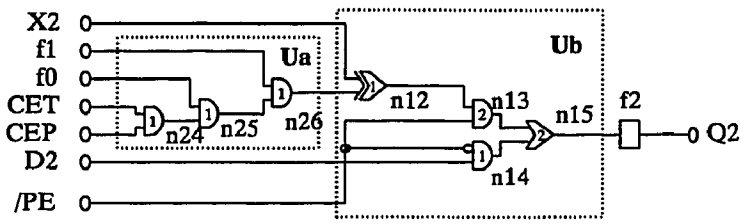


Figure 7.1-6 Level labeling of sub-circuit Q2 and combinational function unit mapping

We use the same procedure for sub-circuit Q2. Figure 7.1-6 shows the Flow Map results. Ua and Ub form a possible mapping resolution. According to this configuration there may be two levels of logic blocks used to implement bell Q2. We try the FG mode again. Nodes n12, n13, n14, and n15 are selected to fill combinational function unit U6. We apply the Flow Map algorithm again to cone n26

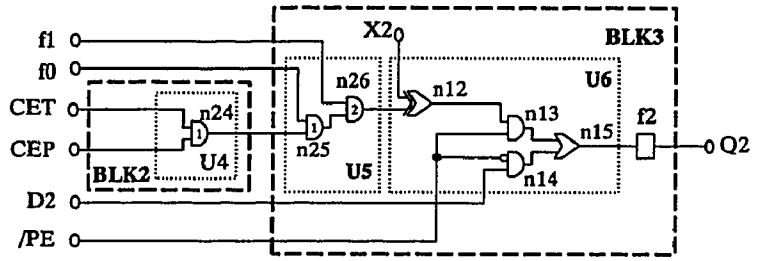


Figure 7.1-7 Modified labeling and mapping of sub-circuit Q2

with 3-feasible policy, because D2 and /PE are not inputs of sub-circuit n26 and one half of the combinational function generator already uses four inputs, two of which are from the inside of the block.

In Figure 7.1-7, the small numbers inside gates, n24, n25, and n26, indicate the new level labeling system of cone n26. Since sub-circuit Q1 has more depth than sub-circuit Q0, node n26 is on the major path. U5 and U4 are the results of this new mapping. U5 and U6 satisfy the FG mode and can be packed in one logic block, BLK3, with flip-flop f2. This shows us that the optimal depth of sub-circuit Q2 is just one greater than the optimal depth of sub-circuit Q1, hence three. U4 can be packed into one logic block, BLK2. This does not affect the depth of sub-circuit Q2 because

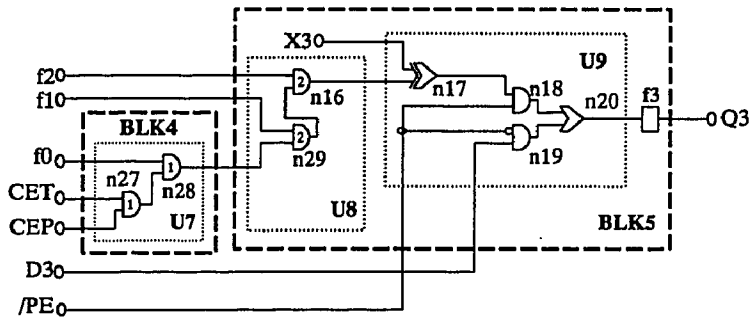


Figure 7.1-8 Modified level labeling and mapping of sub-circuit Q3

the depth of BLK2 is only 1, less than the depth of f1.

We continue to apply the same procedure to sub-circuit Q3. The result is shown in Figure 7.1-8. The optimal depth of sub-circuit Q3 is four, one more than thedepth of Q2. Sub-circuit TC is much simpler. The bell TC is a pure combinational circuit and a full size combinational function generator is used to implement the original gate n21 with five inputs from CET, f0, f1, f2, and f3. This is shown in Figure 7.1-9.

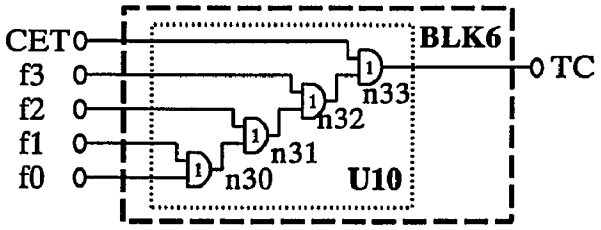


Figure 7.1-9 Level labeling and mapping of sub-circuit TC

We have now mapped all sub-circuits onto logic blocks. Figure 7.1-10 is the mapping result shown in blocks. There are total seven logic blocks arranged in five levels. We next procedurally merge blocks. We find that the function produced from block BLK 4 is the same as the one from U2 in block BLK1. Since U2 occupies an independent section of the combinational function generator and can output its result to fanouts of block BLK1, block BLK4 can be merged with BLK1. This implies that BLK4 can be deleted from the mapping resolution and all its fanouts can be connected with U2 via one of the output ports of block BLK1. There are no other such block pairs. Since blocks BLK1, BLK3, and BLK5 have no units producing the same

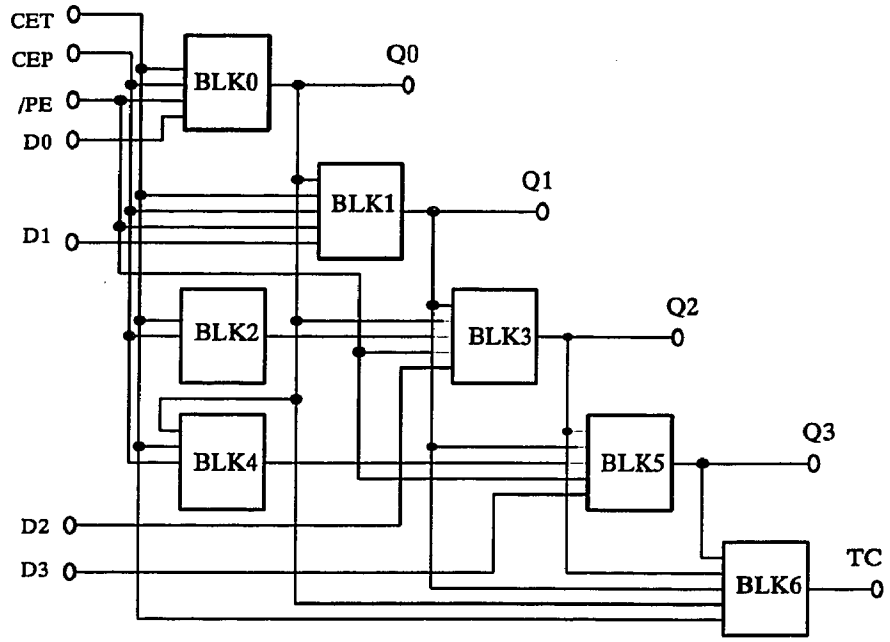


Figure 7.1-10 Block mapping before merge

function as block BLK2 does, we next consider block BLK0 and BLK6. If one ofthese can be decomposed into two parts, one of which does as BLK2 does, then BLK2 may be absorbed by that block. The combinational function generator of BLK1 can be decomposed into two parts using FG mode: node n1 occupies half of the combinational function generator, say U11, and the rest of the nodes occupy the other part, namely U12. Therefore, block BLK2 can be deleted and all of its fanouts are subsequently connected with one of the outputs of block BLK0 from unit U0. After merging, the optimal depth of each sub-circuit is not changed and the total number of blocks used is now only five.

The final results are shown in Figure 7.1-11 and Figure 7.1-12. All gates and mapped combinational function units are shown in Figure 7.1-12. The above

configuration resulting from our algorithm is more effective than the result shown in [Xil92]. Although the two results use same number of logic blocks, our mapping sub-circuits Q0, Q1, Q2, Q3, and TC have depths 1, 2, 3, 4, and 5 respectively while in [Xil92] the depths are significantly larger, which are 2, 4, 6, 8, and 9 accordingly.

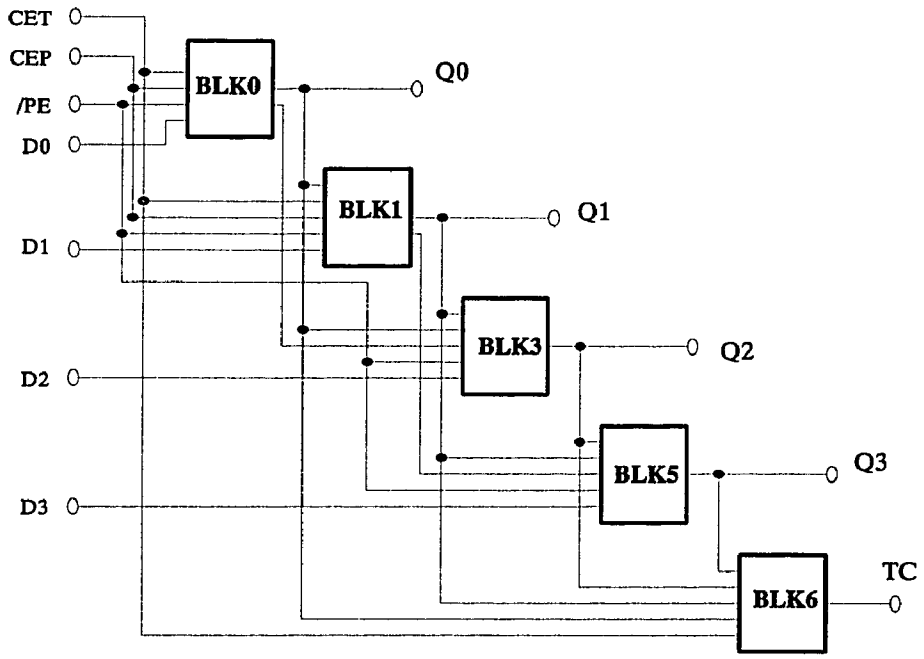


Figure 7.1-11 Block mapping after merge

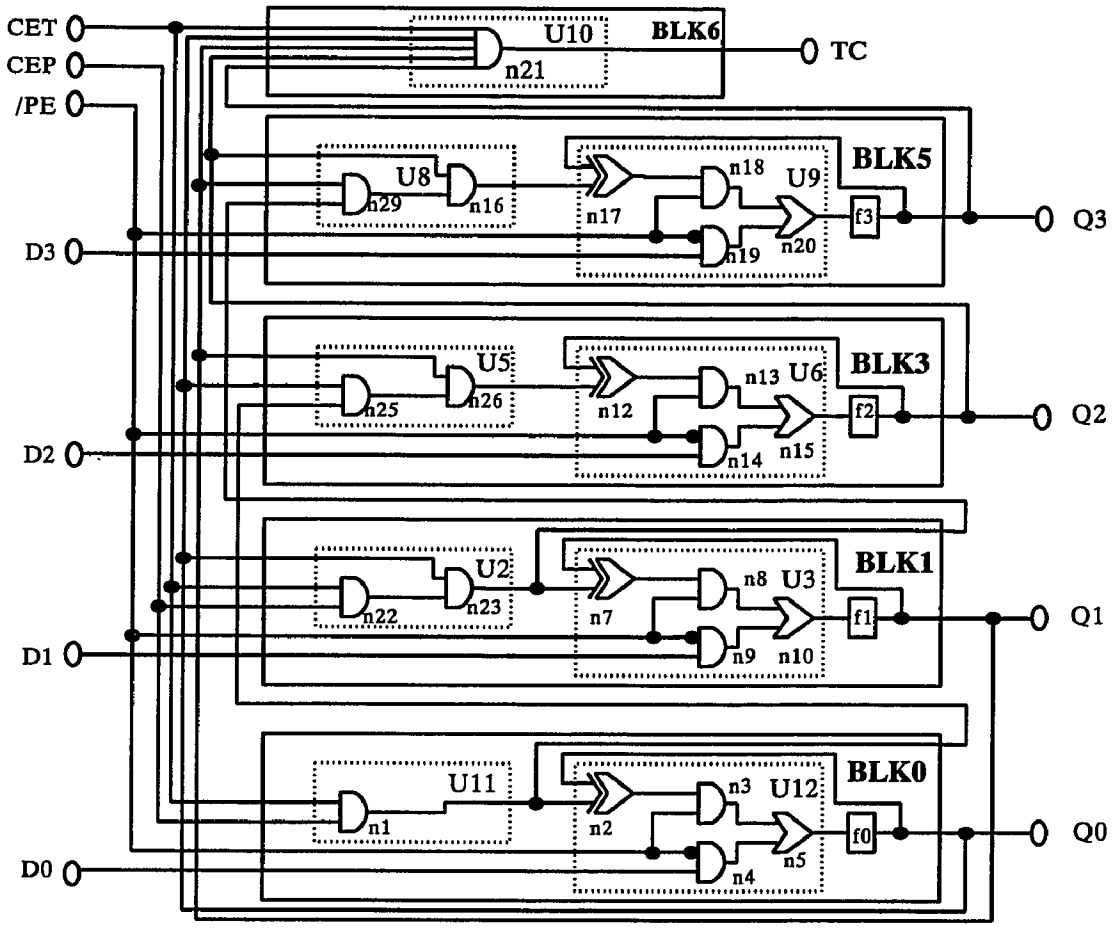


Figure 7.1-12 Final block mapping solution

7.2 Using Special Features

A logic diagram of Example 2 is shown in Figure 7.2-1. Following the mapping of preparation procedures, we get the two sub-circuits: sub-circuit p1 and sub-circuit p2 as shown in Figure 7.2-2 and Figure 7.2-3 respectively. Sub-circuit p1 is also a sub-circuit of p2 because output of f1 is an input of node n7 in p2. We apply the Flow Map algorithm to the two sub-circuits separately. The Flow Mapping results are shown in Figure 7.2-4 and Figure 7.2-5. Note that either units U2 and U3

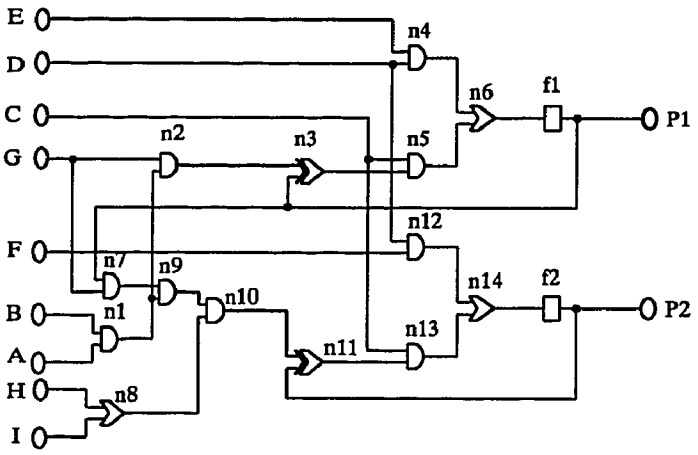


Figure 7.2-1 Logic diagram of example 2

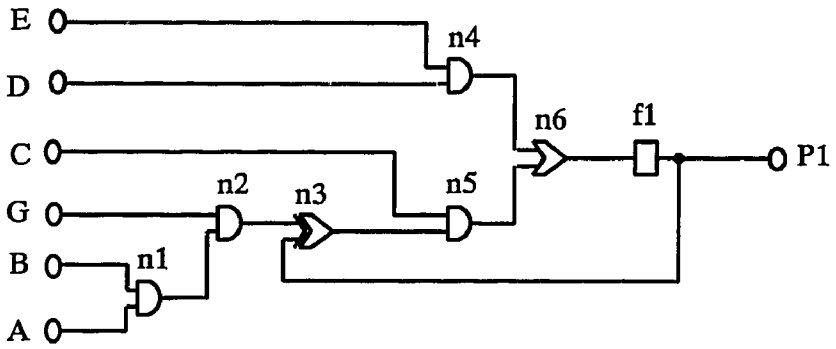


Figure 7.2-2 Level labeling and unit mapping
of sub-circuit P1

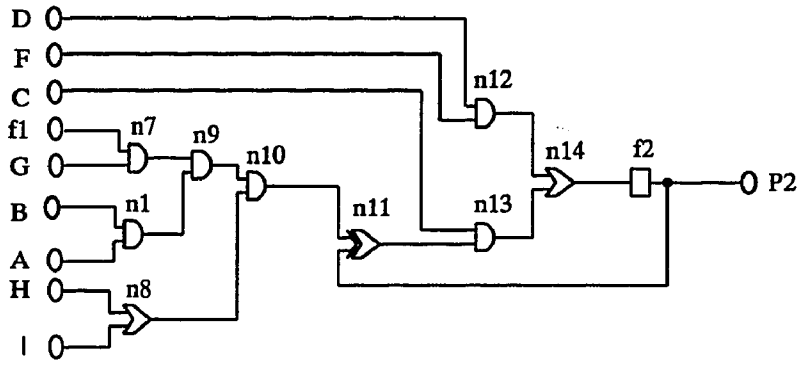


Figure 7.2-3 Sub-circuit P2

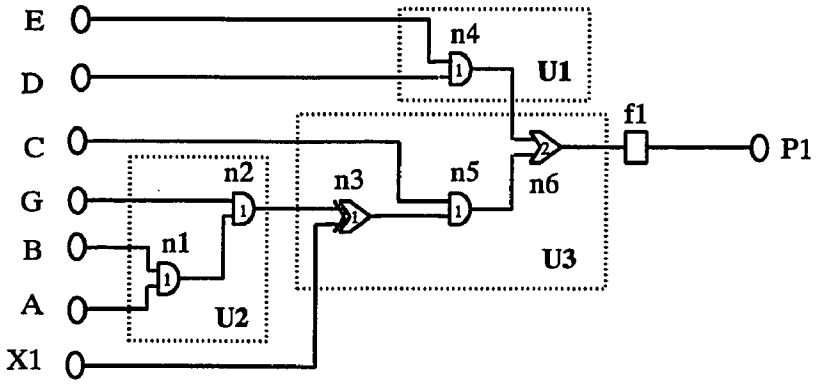


Figure 7.2-4 Level labeling and unit mapping of P1

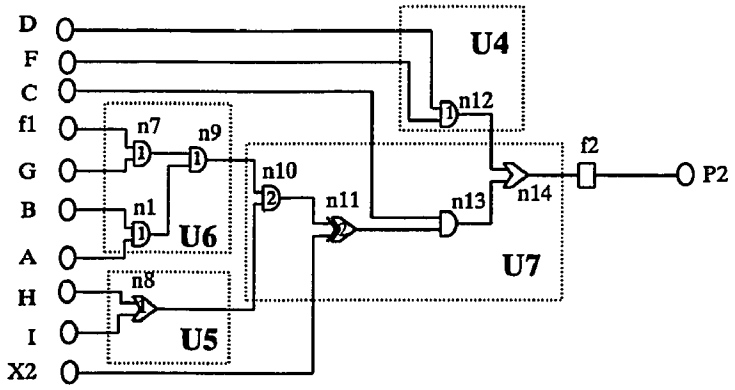


Figure 7.2-5 Level labeling and unit mapping of P2

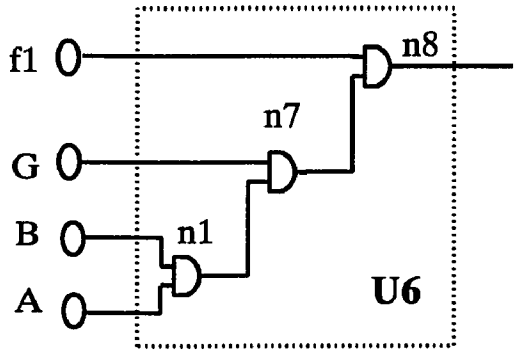


Figure 7.2-6 Redefined logic diagram of a mapped function unit

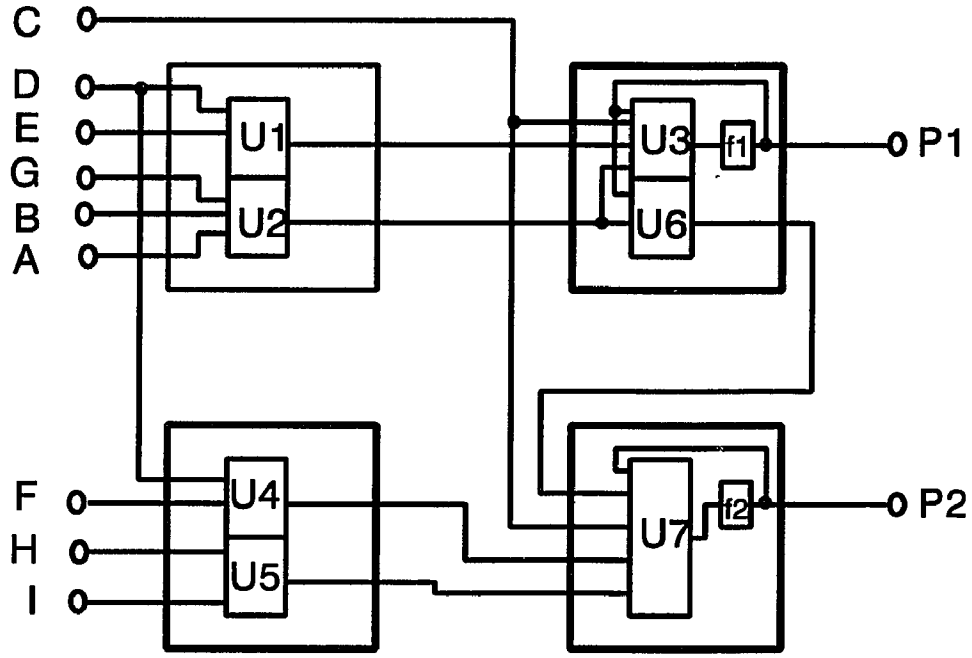


Figure 7.2-7 Final block mapping solution

or units U2 and U1 can be combined together using one unit; however, we did not do so because this procedure will not reduce the depth in blocks for sub-circuit p1.

In sub-circuit p2 we consider the major path: f1→U6 →U7. U7 will occupy an entire combinational function generator of a logic block. It can not benefit from the FG mode. If unit U3, f1, and unit U6 satisfy the hook mapping case 1, then the depth of sub-circuit p2 will be one level deeper than the depth of sub-circuit p1. Figure 7.2-6 shows that U6 is equivalent to f1 AND U2. It is observed that f1 and U2 are two of the four inputs of U3. If we combine U6 with U3 in one block, the rules of total number of input variables of one logic block will not be violated because no new variable will be introduced during the merge. In this example, the total

maximum number of input variables of one block is five. The final block mapping results are shown in Figure 7.2-7. The figure shows each part of a combinational function generator of a logic block and its corresponding mapped function unit. Sub-circuit p1 has depth of two and sub-circuit p2 has depth of three.

7.3 A Circuit with 8 D-type Flip-flops

Section 3.3 Figure 3.3-6 shows a sequential circuit example, which outputs 1 if the input sequence is 01010101 [Woo92].

The technology mapping result of the circuit in Figure 3.3-6, using XC3000 from Xilinx' XNFMAP, is shown in Figure 3.3-8. The total number of CLBs used in this mapping solution is six and the depth of the circuit is six. Flip-flops f0 and f1 share one CLB, B1, which has one input S and produces two outputs Q0 and Q1. Output Q0 has one fanout CLB, B5. Output Q1 has two fanout CLBs, B2 and B5. The entire block B1 has two fanout blocks, B2 and B5. Flip-flops f2 and f3 share one CLB, B2, which has one input, two outputs, and two fanout blocks, B3 and B5. Flip-flops f4 and f5 are mapped onto one block, B3, and similarly has two fanout blocks, B4 and B6. Flip-flops f6 and f7 use one block, B4; however, B4 has only one fanout block B6 even though it has two output functions Q6 and Q7. The critical path of the mapping result is: B1→B2→B3→B4→B6→B5. The number of links between the six CLBs is twelve.

Figure 3.3-7 illustrates the technology mapping result of ATOM [Woo92] for

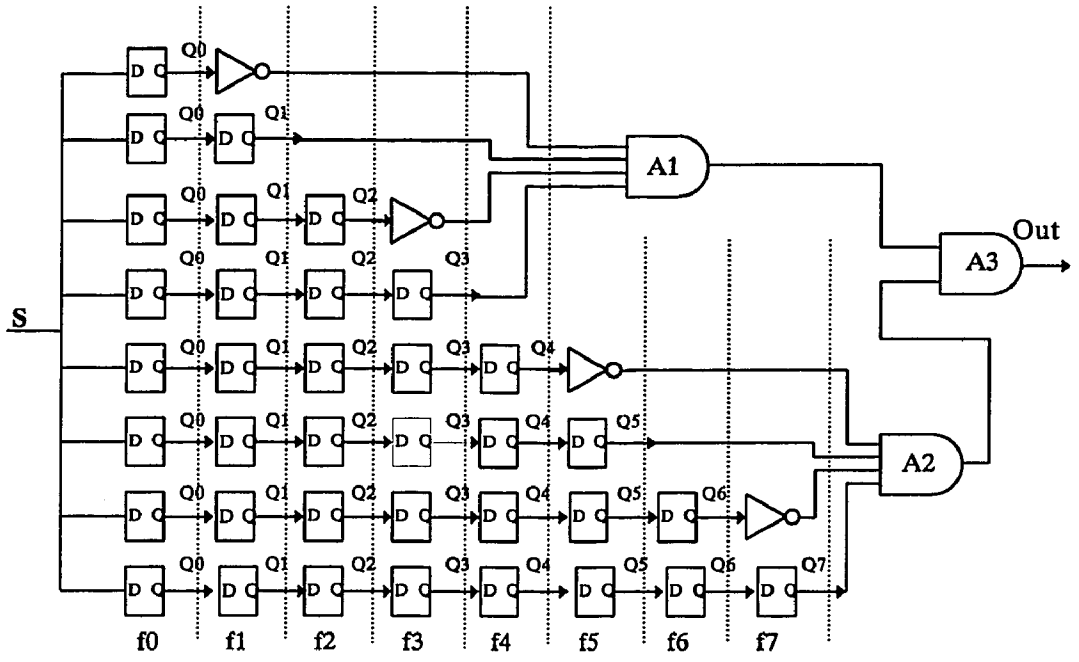


Figure 7.3-1 Pre-mapping: elimination of cycles

he same circuit. Using the ATOM technology mapping algorithm, the same sequential circuit uses five CLBs and the depth of the circuit is six: CLB1→CLB3→ CLB2→CLB4→CLB5→CLB2. The number of links between the five CLBs is ten.

To map the circuit using our algorithms, first we convert the circuit into a bellnet. Figure 7.3-1 illustrates an equivalent circuit of the 8-f/f circuit in Figure 3.3-6 after elimination of cycles. Then we deal with each bell respectively. There are nine bells in this circuit as shown in Figure 7.3-2. Bell B9 consists of combinational nodes only, while the other eight bells consist of flip-flops only. Bell B9 is a cone with top node A3. The remaining eight bells are all hooks without combinational nodes. The Hook Map algorithm is applied to each hook respectively. Flow Map is applied to bell B9. To use Flow Map, bell B9 is converted into a 2-bound tree.

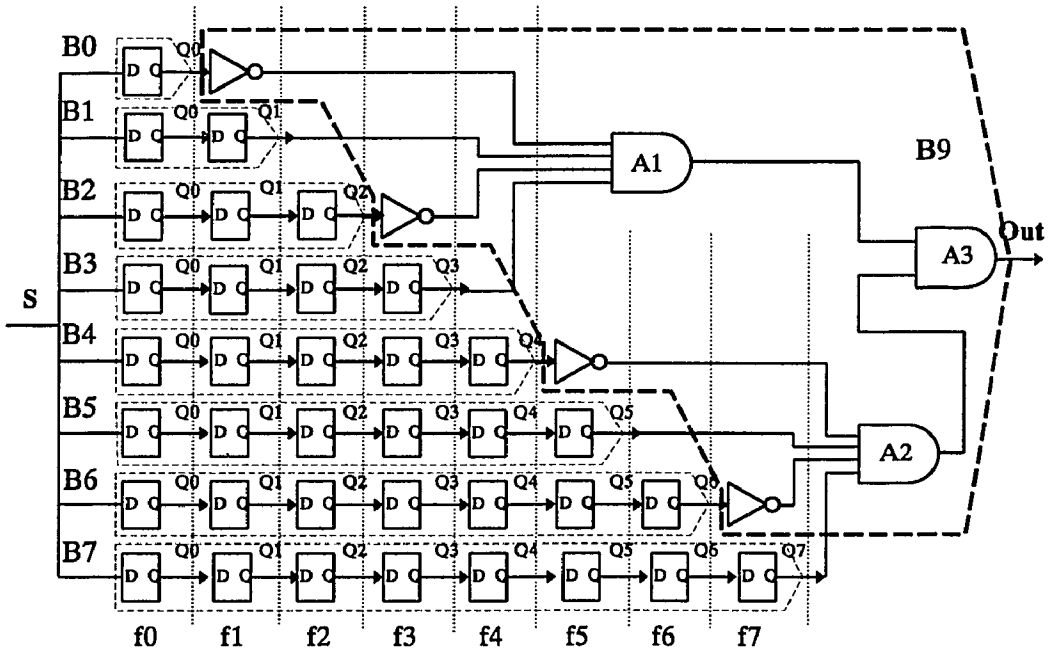


Figure 7.3-2 Pre-mapping: bell structures

Using Flow Map, each node in this tree is assigned a level label. Since combinational function units in the logic blocks of the XC3000 model can implement any kind of boolean logic, an inverter can then be combined with "AND" logic in one gate. The Pre-Mapping result is shown in Figure 7.3-3.

Figure 7.3-4 shows a proposed block mapping solution of the hooks in Figure 7.3-3. It is obvious that there are many proposed blocks that produce same functions due to the duplication of flip-flops during pre-mapping process. The duplicated blocks should be eliminated to reduce the number of blocks in the mapped CLB net. Each time when a new block is being invoked, if there is already a block in the mapped CLB net that computes the same function, then it is not necessary to invoke a new block. This means that by adding one more fanout block to the existing

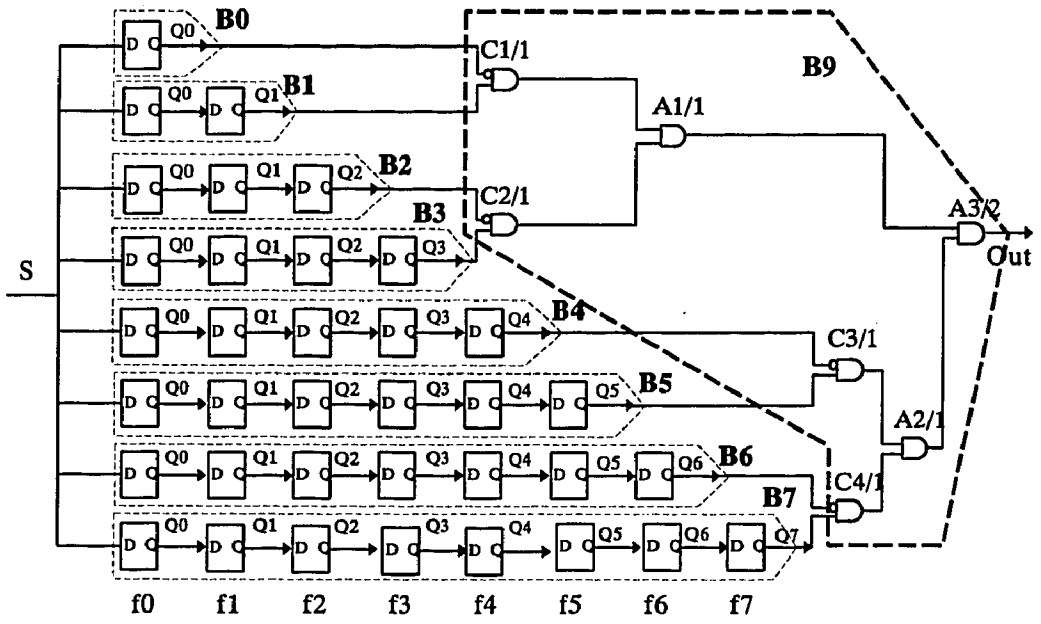


Figure 7.3-3 Pre-mapping: 2-bound net of a cone

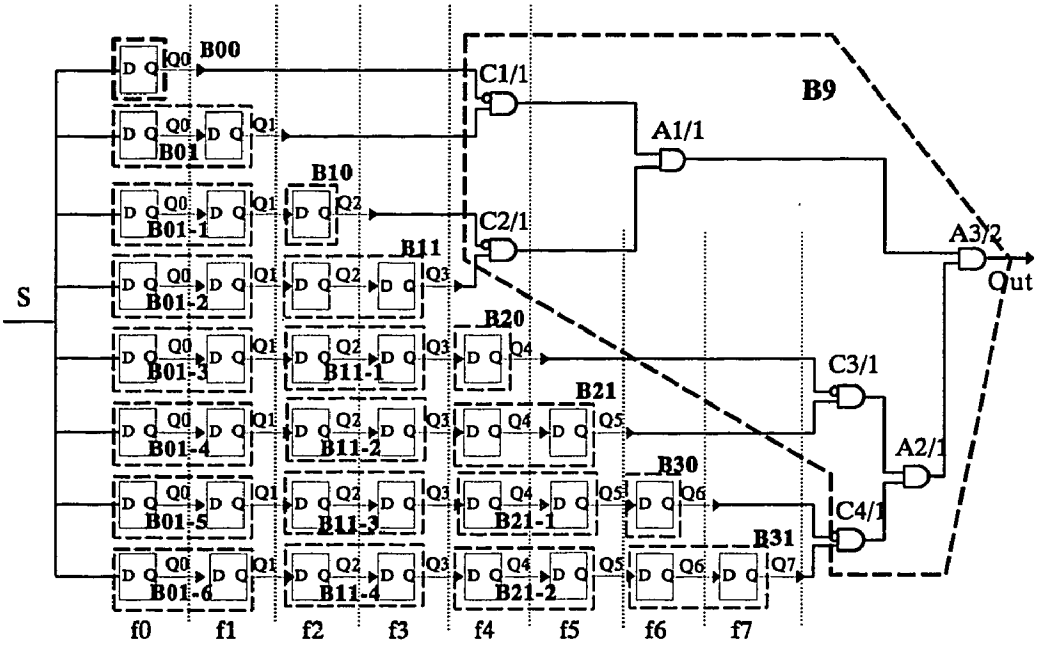


Figure 7.3-4 Block mapping of hooks

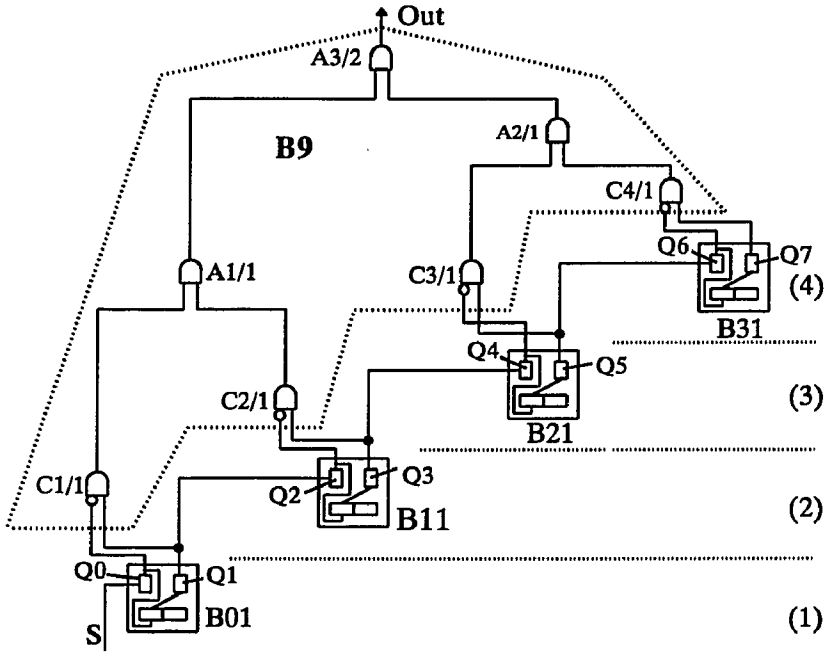


Figure 7.3-5 After eliminating duplicated blocks of hooks

block that computes the same function as an input of the adding fanout block, we eliminate one block, which would be used to compute a same function. Therefore, the block mapping results for hooks contain fewer blocks than what it would have if all blocks that computes same functions were kept in the mapped CLB net. Figure 7.3-5 illustrates the solution. We label the depth of each block in Figure 7.3-5. Block B01 has the lowest depth of 1. Block B31 has the highest depth of 4 in the mapped circuit part. We then apply different mapping algorithms discussed in Chapter 6 to the circuit shown in Figure 7.3-5.

First we pack nodes in B9 from top to bottom according to their level values.

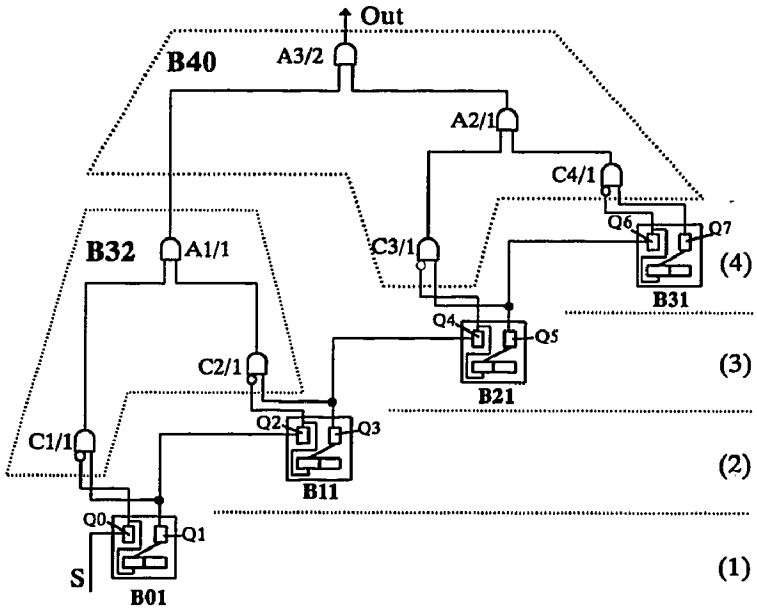


Figure 7.3-6 Cone mapping using algorithm LFF

Flow Map guarantees that the highest level value of a combinational boolean network is optimal depth of the network; but the number of packing solutions varies and the r

method is to pack as many as possible nodes into one block. Based on the circuit in Figure 7.3-5, we start packing from node A3. Since only node A3 has level value 2, the highest level, and the mapping block still have more room available for other nodes to be packed in, we expand the mapping block to include arbitrarily its fanin nodes until either the mapping block has no room for more nodes or the mapping block has no more fanin nodes for packing. Figure 7.3-6 shows packing of all combinational nodes in the cone of A3.

We then consider the merge of blocks that have fanin-fanout relationships (two nodes are connected by one edge). This is a part of the Hook Map algorithm. We

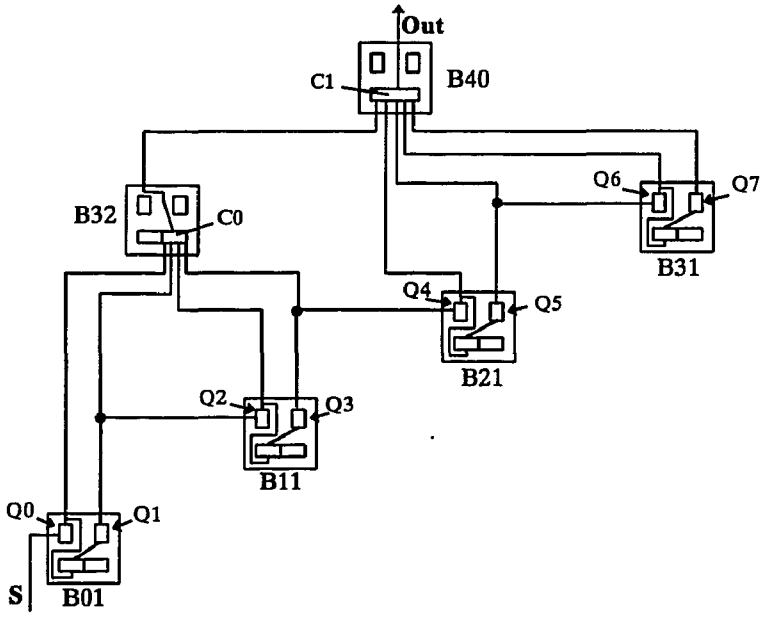


Figure 7.3-7 After cone mapping using LFF (1)

start from blocks with the highest depth value. Blocks B31 and B40 are not qualified to be merged. but blocks B30 and B5 can be combined together. Similarly blocks B32 and B11 can be combined into one block. The result is shown in Figure 7.3-7. After Hook Map has been applied the circuit has five blocks. The mapping result is shown in Figure 7.3-8.

The total number of blocks used in the final circuit is five and the number of links between logic blocks is ten. The circuit now has a depth of five. The result is similar to the result from ATOM's packing algorithm but its depth is one level less.

In Figure 7.3-5, if we arbitrarily expand the top block B40 to the other direction, how will be the result?

Figures from Figure 7.3-9 through Figure 7.3-10 show the similar process to the one we just have done. The final solution uses same number of blocks and links.

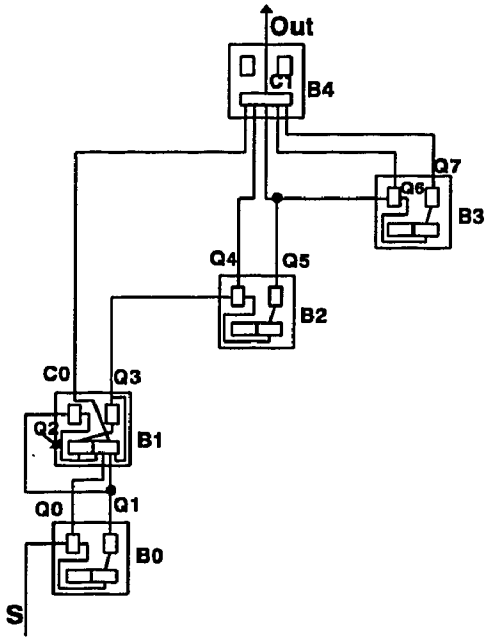


Figure 7.3-8 Final solution using LFF (1)

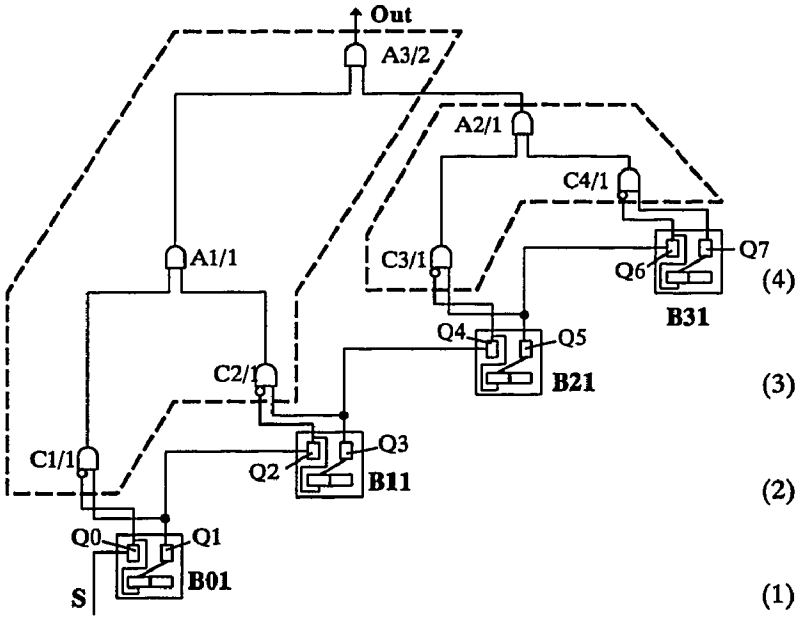


Figure 7.3-9 Cone mapping using LFF (2)

The final circuit depth is also the same.

From formula (1) of Chapter 4 and the block mapping result of hooks in Figure 7.3-8 the longest path of the circuit is: B01→B11→B21→B31→(C4, A2)→A3. The optimal depth of the circuit should be not greater than six. In the previous demonstration, we get result with one level less the maximum.

The depth of a circuit is determined by the depth of its critical path(s). If the depth of a critical path can be reduced, then the depth of the entire circuit may benefit by this. Algorithm JFF, Joint Fit First, is designed for this purpose. We start from the circuit in Figure 7.3-5. The longest bell-bottom of a bell is the critical path of the bell. The top most combinational segment of a critical path is the joint of the longest bell-bottom. If a joint can share blocks with its following bell, which generally starts a series of flip-flops, then the length of a critical path may have opportunity to be reduced. Therefore, we start work from the top node that is not packed in a working bell. We choose this node as a core node and assign it to a new block. We then expand the new block to the chosen critical segment till either the new block reaches the following mapped hook or no more nodes can be packed into the new block. We repeat the process till all nodes in the working bell are packed into blocks.

In Figure 7.3-5, the top node that is not packed into a block is node A3. The critical path is B01→B11→B21→ B31→[C4→A2]→[A3]. The corresponding joint is <A3-A2-C4>. We first pack node A3 into a new block, B40. We expand block B40 to the critical bell-bottom. Nodes A3, A2, and C4 are mapped onto block B40 and the block reaches mapped block B31. Both node A1 and node C3 are now the

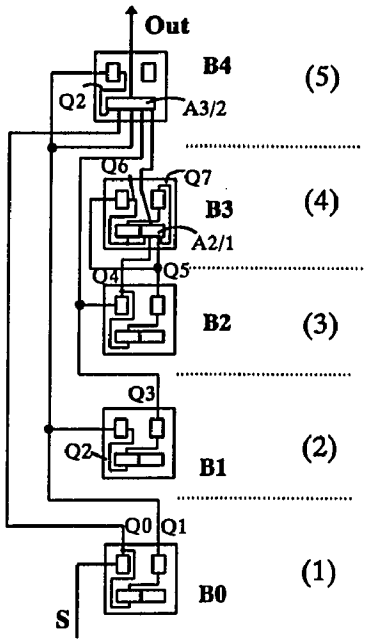


Figure 7.3-10 Final solution using LFF (2)

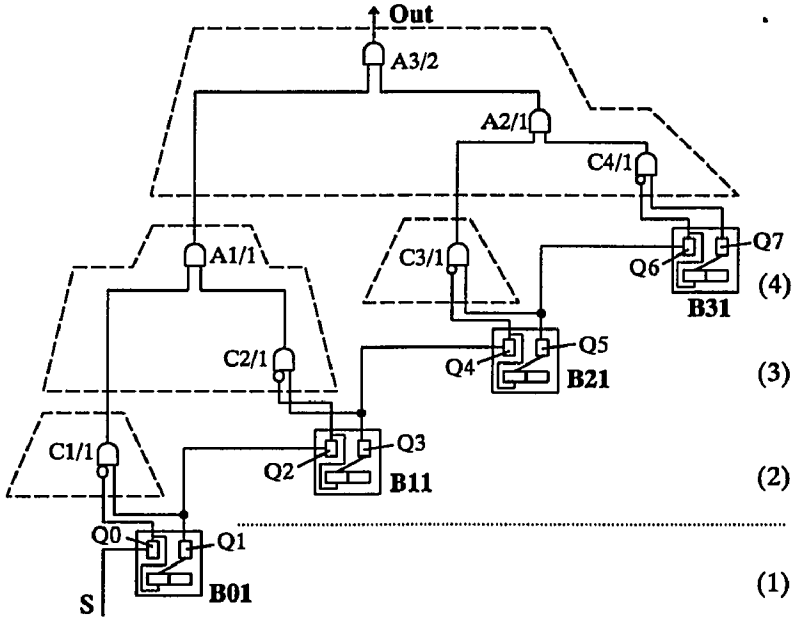


Figure 7.3-11 Cone mapping using algorithm JFF

unpacked top nodes in the rest parts of cone B9. We then apply the procedure to A1 and C3 respectively. After that only one node C1 is left unpacked. Apply the procedure to node C1. Figure 7.3-11 shows the process.

We now consider using the Hook Map algorithms. Blocks B31, B21, B11, and B01 are chosen to be involved in HookMap with their fanout blocks respectively. The result of HookMap is shown in Figure 7.3-12.

Four blocks and six links between logic blocks are used in the final solution. The depth of the final circuit is four. The result is better than the results from ATOM, XNFMAP, and the previous solution using algorithm LFF.

It is observed that when we pack nodes along the top combinational segment of a critical path, we broke down the optimal level structure of a cone. We can see that in Figure 7.3-11 cone A3 is collapsed into a three level network: node C1 is in the lowest level; nodes A1 and C3 are in the second level; while nodes A3, A2, and C4 are in the topmost level. The length of the path from node C1 to node A1 to node A3 becomes three instead of two. This may cause some risk. For example, in Figure 7.3-11 if the proposed block of nodes C1 can not be combined with block B01, block B01 is not a fanin of blocks B11, and the depth of B01 is four, then the final solution would be worse.

To avoid the risk, we use algorithm LFTJ, Level First Then Joint, that is discussed in chapter 6. Using this algorithm we do not break out the optimal level structure of a cone, but still consider the critical path overwhelming on other lower level nodes. We start from the top node of a cone, which is not packed into a block.

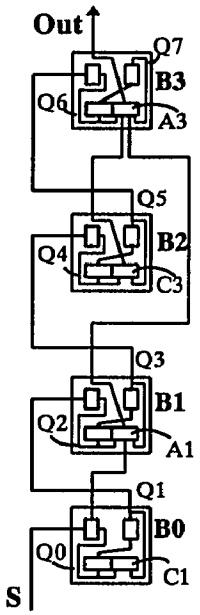


Figure 7.3-12 Final solution using algorithm JFF

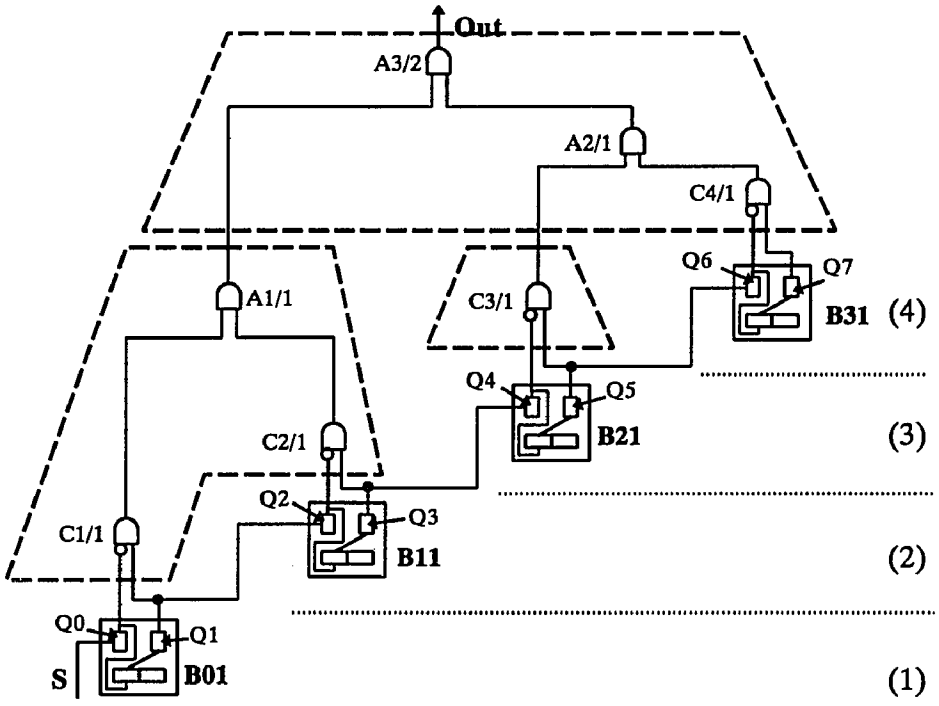


Figure 7.3-13 Cone mapping using algorithm LFTJ

We choose this top node as a core node and pack it into a new block. All clique members of this core node are then packed into the block. Then we expand the new block to the critical bell-bottom from the core node until either the new block has no room for packing more node into it or the new block reaches the following mapped hook.

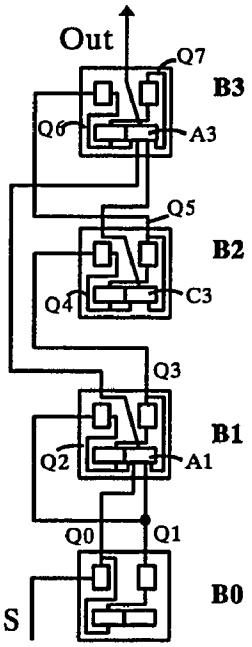


Figure 7.3-14 Final solution using algorithm LFTJ

We apply this procedure to the circuit which is shown in Figure 7.3-5. The whole process is shown from Figure 7.3-13 through Figure 7.3-15. The final solution uses four blocks and seven links between logic blocks. The final entire circuit has a depth of 4. Comparing with the solution from algorithm JFF there is only one more link used, but avoid the risk making more levels. A comparison of solutions using

different algorithms on this 8-flip-flop circuit is listed Table 7.1.

Algorithms	ATOM	XNFMAP	LFF	JFF	LFTJ
Blocks used	5	6	5	4	4
Depth in blocks	6	6	5	4	4
Links between blocks	10	12	10	6	7

Table 7.1 Comparison of mapping solutions

CONCLUSION

Field programmable gate arrays give circuit designers the possibility of a new design approach. More designers are using FPGAs to implement their logic designs, especially for circuit prototyping. Even though hardware costs goes down constantly, improving the performance of FPGAs and making the maximum use of the silicon area is still a main research goal in the FPGA and CAD community. This thesis joins much of the research in this field. The terminology defined in this thesis presents a convenient way to describe the problems discussed in this research work. Even though FPGA vendors have their design tools for sequential circuits, including technology mappers, these technology mappers have never been published. Currently FPGA researchers are moving their attention from combinational circuits to sequential circuits [PanL96]. Our following work extends the use of the algorithms in this thesis to general look-up table FPGA models. Basically the algorithms in this thesis should be applicable to other types of look-up table models, such as Xilinx XC4000 products. Our research indicates that even every combinational part is optimized. For a global optimization of a sequential circuit the optimization in each sub-circuit may have to be subdivided. This implies that we may not have to consider optimization for each combinational part in a sequential circuit. Whether there is an depth optimization mapping solution for a given sequential circuit using a target FPGA or not is still an unsolved problem. Our next goal is to explore this problem and continually to

improve the efficiency of the technology mapping algorithms presented in this thesis to have placement and routing get most benefits from mapping processes.

WORKS CITED

[Act94] Actel Corporation, *Actel FPGA Data Book and Design Guide*, 1994.

[Ahr90] M. Ahrens, et. al., "An FPGA Architecutre Optimized for High Densites and Reduced Routing Delay," *Proceedings of IEEE Custom Integrated Circuits Conference*, July 1990.

[Alf89] Roger C. Alford, *Programmable Logic Designer's Guide*, Howard W. Sams & Company, Indianapolis, Indiana, 1989.

[Aya89] K. El Ayat, et. al. "A CMOS Electrically Configurable Gate Array," *IEEE J. Solid-State Circuits*, Vol. 24, No. 3, June, 1989, pp. 752-762.

[BhaH92] Narashima Bhat and Dright D. Hill, "Routable Technology Mapping for LUT FPGAs," *Proceedings of IEEE International Conference on Computer Design: VLSI in Computer & Processors*, 1992, pp. 95-98.

[BroF92] Stephen D. Brown, Robert J. Francis, Jonathan Rose, and Zvonko G. Vranexic, *Field-Programmable Gate Arrays*, published by Kluwer Academic Publishers, Boston, 1992, p.p. 206.

[CarD86] William S. Carter, Khue Duong, Ross H. Freeman, Hung-Cheng Hsieh, Jason, Ja, John E. Mahoney, Luan T. Ngo, Shelly L. Sze, "A User Programmable Reconfigurable Logic Array," *Proceedings of EEE Integrated Circuits Conference*, May 1986, pp. 231-235.

[Cart91] William S. Carter, "The Evolution of Programmable Logic," 1991 Symposium on VLSI Circuits, 1991, pp.43-46.

[Cer86] L. Cerzberg, *US Patent 4,590,589, 1986.*

[CheC92] Kuang-Chien Chen, Jason Cong, Yuzheng Ding, Andrew B. Kahng, and Peter Trajmar, "DAG-Map: Graph-Based FPGA Technology Mapping for Delay Optimiza-tion," *IEEE Design & Test of Computers*, September 1992, pp. 7-22.

[CheW94] Y. P. Chen and D. F. Wong, "On Retiming for FPGA Logic Module Minimization," *Proceedings of IEEE International Conference on Computer Design: VLSI in Computer & Processors*, 1994, pp. 394-397.

[ChuR92] Kevin Chung and Jonathan Rose, "TEMPT: Technology Mapping for the Exploration of FPGA Architectures with Hard-Wired Connections," *Proceedings of 29th ACM/IEEE Design Automation Conference*, 1992, pp. 361-367.

[ChuS91] Kevin Chung and Satwant Singh, "Using Hierarchical Logic Blocks to Improve the Speed of FPGAs," *FPGAs,* W. R. Moore & W. Luk (eds.), 1991, Abingdon EE&CS Books, Abingdon, England, pp. 102-113.

[ConD92] Jason Cong, Yuzheng Ding, Andrew B. Kahng, and Peter Trajmar, "An Improved Graph-Based FPGA Technology Mapping Algorithm for Delay Optimization," *Proceedings of IEEE International Conference on Computer Design: VLSI in Computer & Processors,* October 1992, pp. 154-158.

[ConD93] Jason Cong and Yuzheng Ding, "On Area/Depth Trade-off in LUT-Based FPGA Technology Mapping," *Proceedings of 30th ACM/IEEE Design Automation Conference*, June 1993, pp.213-218.

[ConD94] Jason Cong and Yuzheng Ding, "Flow Map: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Design," *Transaction on Computer-Aided Design of Integrated Circuits and Systems,* Vol. 13, No. Jan. 1994, pp. 1-12.

[ConD94] Jason Cong and Yuzheng Ding, "On Area/Depth Trade-Off in LUT-Based FPGA Technology Mapping," *IEEE Trans. on VLSI Systems*, Vol. 2, No. 2, June 1994, pp. 137-148.

[EbeB91] Carl Ebeling, Gaetano Borriello, Scott A. Hauck, David Song, Elizabeth A. Walkup, "TRIPTYCH: A New FPGA Architecture," *FPGAs, W. R. Moore & W. Luk (eds.), 1991, Abingdon EE&CS Books, Abingdon, England,* pp. 75-90.

[EET85a] "A simple device to cut your gate array costs by 50%," *Electronic Engineering Times*, November 4, 1985, pp. 18-19.

[EET85b] "Startup Xilinx puts its faith in reconfigurable logic array chips," *Electronic Engineering Times*, December 9, 1985, pp. 1, 12.

[Ekl81] Eklund, M., "Semicustom high density business," *ICECAP Report*, August 28, 1981, Integrated Circuit Engineering Corp., Scottsdale, AZ.

[ElA89] K. El-Ayat, et.al., "A CMOS Electrically Configurable Gate Array," *IEEE J. Solid-State Circuits,* Vol. 24, No. 3, June, 1989, pp. 752-762.

[Ep85] "Logic array reconfigures itself on the fly," *Electronic Products,* November 15, 1985, pp. 27-30.

[ErcM91] Silvia Erociani and Giovanni De Micheli, "Technology Mapping for Electrically Programmable Gate Arrays," *Proceedings of 29th ACM/IEEE Design Automation Conference*, 1991, pp. 234-239.

[ForF62] Lestor R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[FraR90] Robert J. Francis, Jonathan Rose, and Kevin Chung, "Chortle: A Technology Mapping Program for Lookup Table-Based Field Programmable Gate Arrays," *Proceedings of 27th ACM/IEEE Design Automation Conference,* 1990, pp. 613-619.

[Guc95] Steve Guccionel, "List of FPGA-based Computing Machines," World Wide Web file: *http://www.io. com/~guccione/ hw_list.htm.*

[HabX95] Stanley Habib and Quan Xu, "Technology Mapping Algorithms for Sequential Circuits Using Look-up Table Based FPGAs," Proceedings Fifth Great Lakes Symposium on VLSI, The State University of New York at Buffalo, March, 1995, pp. 164-167.

[HamM88] E. Hamdy, J. McCollum, S. Chen, S. Chiang, S. Eltoukgy, J. Chang, T. Speers, Mohsen, "Dielectric Based Antifuses for Logic and Memory ICs," *IEDM Tech. Digest*, pp. 786-789, 1988

[HaqM94] Faisal Haq and Samiha Mourad, "Optimal Logic Blocks for FPGAs, using Fractorical Design Techniques," *Proceedings of ICCD*, October 1994, pp. 470-474.

[Har84] R. Hartmann, "Estimating Gate Compexity of Programmable Logic Devices," *VLSI Design Magazine*, pp. 100-102, May 1984.

[Hau95] Scott Hauck, *"Multi-FPGA Systems,"* Ph. D. Diss., University of Washington, 1995.

[HauB94] Scott Hauck, Steven Burns, Gaetano Borriello, and Carl Ebeling, "An FPGA for Implementing Asynchronous Circuits," *IEEE Design & Test of Computer*, Fall 1994, pp. 60-69.

[Huf52] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the IRE*, Vol 40, No. 9, pp. 1098-1101, 1952.

[Joh75] Donald B. Johnson, "Finding All the Elementary Circuits of a Directed Graph," *SIAM Journal on Computing*, Vol. 4, No. 1, March 1975, pp. 77-84.

[Kar91] Kevin Karplus, "Xmap: A Technology Mapper for Table-Lookup Field-Program-mable Gate Arrays," *Proceedings of 28th ACM/IEEE Design Automation Conference1991*, pp. 240-247.

[KunD92] D. S. Kung, R. F. Damiano, T. A. Nix, and D. J. Geiger, "BDDMAP: A Technology Mapper Based on a New Covering Algorthm," *Proceedings of 29th ACM/IEEE Design Automation Conference*, 1992, pp. 484-487.

[Lal90] Parag K. Lala, *Digital system design using programmable logic devices*, Prentice-Hall, Inc., 1990.

[Law69] Eugene L. Lawler, K. N. Levitt, and J. Turner, "Module Clustering to Minimize Delay in Digital Networks," *IEEE Trans. on Computers,* Vol. C-18, No. 1, Jan. 1969, pp. 47-57.

[Law76] Eugene L. Lawler, *Combinatorial Optimization*, published by Holt, Rinehart and Winston, 1976.

[LeiK94] Samir Lejmi, Bozena Kaminska, Edoward Wagneur, "Retiming for the Global Optimization of Synchronous Sequential Circuits," Proceedings of ICCD, 1994, 398-401.

[LeiR83] Charles e. Leiserson, Flavio M. Rose, and James B. Saxe, "Optimizing Synchronous Circuitry by Retiming," *The CALTECH Conference on Very Large Scale Integration*, Randal Bryant (eds), Computer Science Press, Maryland, 1983, pp. 87-117.

[MurB93] Rajeer Murgai, Robert K. Brayton, and Alberto Sangiovani-Vincenteli, "Some Results on the Compelexity of Boolean Functions for Table Look Up Architectures," *Proceedings of IEEE International Conference on Computer Design: VLSI in Computer & Processors*, 1993, pp. 505-512.

[MurS91a] Rajeev Murgai, Narendra Shenoy, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli, "Performance Directed Synthesis for the Table Look Up Programmable Gate Arrays," *Proceedings of IEEE International Conference on Computer-Aided Design,* 1991, pp. 572-575.

[MurS91b] Rajeev Murgai, Narendra Shenoy, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli, "Improved Logic Synthesis Algorithm for Table

Look Up Architectures," *Proceedings of IEEE International Conference on Computer-Aided Design,* 1991, pp. 564-567.

[PanL96] Peichen Pan and C. L. Liu, "Technology Mapping of Sequential Circuits for LUT-based FPGAs for Performance," from the authors, Dept. of ECE, Clarkson University, NY, 1996.

[PedB91] Massoud Pedram and Narasimha Bhat, "Layout Driven Technology Mapping," *Proceedings of 28th ACM/IEEE Design Automation Conference,* 1991, pp. 99-105.

[PelH91] David Pellerin and Michael Holley, *Practical Design Using Programmable Logic*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1991.

[SawT93] Prashant Sawkar and Donald Thomas, "Performance Directed Technology Map-ping for Look-Up Table Based FPGAs," *Proceedings of 30th ACM/IEEE Design Automation Conference*, June 1993, pp. 208-212.

[Sch93] J. Schlageter, et. al., "An Advanced Sub-Micron Architecture for IO Intensive Applications," *Proceedings of the 1993 Compcon Conference*, 1993, pp. 362-366.

[SchK94] Martine Schlag, Jackson Kong, and Pak K. Chan, "Routability-Driven Technology Mapping for Lookup Table-Based FPGAs," IEEE Trans. on CAD, Vol.13, Jan. 1994, pp. 13-26.

[SchP93] Martine Schlag, Jackson Kong, and Pak K. Chan, "Routability-Driven Technology Mapping for Lookup Table-Based FPGA's," *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 13, No. 1, Jan. 1994, pp. 13-26.

[Tri94] *Field-Programmable Gate Array Technology*, edited by Stephen M. Trimberger, Xilinx, published by Kluwer Academic Publishers, Boston, 1994, pp. 258.

[WhiB90] R. Whitten, R. Bechtel, M. Thomas, H. T. Chua, A. Chan, J. Brikner, *European Patent Application No. 90309731.9*, May 9, 1990.

[WhiS93] T. Whitney and J. Schlageter, "A New High Performance Field Programmable Gate Array Family," *Proceedings of 1993 International Conference on Computer Design*, October 1993.

[Woo91] Nam-Sung Woo, "A Heuristic Method for FPGA Technology Mapping

Based on the Edge Visibility," *Proceedings of 28th ACM/IEEE Design Automation Conference,* 1991, pp. 248- 251.

[Woo92] Nam-Sung Woo, "ATOM: Technology Mapping of Sequential Circuits for Lookup Table-Based *FPGAs*," paper from the author, AT & T Bell Lab., Murray Hill, NJ, 1992.

[Xil91] Xilinx, Inc., *Xilinx User Guide and Tutorials*, San Jose, California, 1991.

[Xil92] Xilinx, Inc., *The Programmable Gate Array Data Book*, San Jose, California, 1992.

[Xil94] Xilinx, Inc., *Xilinx The Programmable Logic Data Book*, San Jose, California, 1994.

[XuH96] Quan Xu and Stanley Habib, "Another Technology Mapping Algorithm for Sequential Circuits Using LUT-Based FPGAs," to be published in Workshop on Academic Electronics in New York State, Sytrcuse, New York, June 1996.

[YanW94] Honghua Yang and D. F. Wong, "Edge-Mape: Optimal Performance Driven Technology Mapping for Iterative LUT Based FPGA Designs," *Digest of Technical Papers of 1994 IEEE/ACM International Conference on Computer Design: VLSI in Computer & Processors*, 1994, pp. 150-155.

[York93] Trevor A. York, "Survey of field programmable logic divices," *Microprocessors and Microsystems*, Vol. 17, No. 7, September 1993, pp. 371-381.