

UNITED STATES PATENT AND TRADEMARK OFFICE

BEFORE THE PATENT TRIAL AND APPEAL BOARD

INTEL CORPORATION,
Petitioner,

v.

MASAHIRO IIDA,
Patent Owner.

Case No. IPR2023-00864
U.S. Patent No. 6,812,737

**DECLARATION OF AGNES O. VILLERO
UNDER 28 U.S.C. § 1746**

I, Agnes O. Villero, do hereby declare as follows:

1. I make this declaration on behalf of Petitioner Intel Corporation in the matter of the *inter partes* review of U.S. Patent No. 6,812,737 (the “IPR Proceeding”). Unless otherwise stated, the facts stated in this declaration are based on my personal knowledge.

2. I am Library Manager at Munger, Tolles & Olson LLP (“MTO”), 350 South Grand Avenue, 50th Floor, Los Angeles, California 90071. I received a master’s degree in Library and Information Sciences from the University of California, Los Angeles, in 1996. Between 1996 and 2001, I worked at UNOVA, an oilfield services company, first as an Information Research Specialist and, following a promotion, as a Research Analyst. From 2001 to 2006, I worked as an Acquisitions Librarian at the law firm of O’Melveny & Myers LLP. In 2006, I joined MTO. I held the title of Librarian until 2020, when I was promoted to Library Manager. As the Library Manager, I provide research and reference services to attorneys and staff located in all of the firm’s offices, train and orient attorneys and staff in research techniques and use of library materials and resources, and provide services to practice groups, including the patent group.

3. I am familiar with library procedures for entering and indexing print and electronic sources in publicly available searchable databases such as

WorldCat, the U.S. Copyright Office, Inspec, Compendex, and the Library of Congress Catalog, among others.

I. Ex.1006: Vaughn Betz et al., *Architecture and CAD for Deep-Submicron FPGAs* (1999 Kluwer Academic Publishers)

4. Ex.1006 in the IPR Proceeding is Vaughn Betz et al., *Architecture and CAD for Deep-Submicron FPGAs* (1999 Kluwer Academic Publishers) (“Betz”).

The cover material of Betz indicates it was published by recognizable and reputable publisher Kluwer Academic Publishers as part of the Kluwer International Series in Engineering and Computer Science.

5. Appendix A.1 to this declaration is the WorldCat database record for Betz, accessed on March 2, 2023, having OCLC reference number 40602916.

Appendix A.1 indicates that the WorldCat database record for Betz was entered into the WorldCat database on January 4, 1999. Appendix A.1 indicates that 115 libraries worldwide have copies of Betz. Betz would have been locatable and available to the public by search in the WorldCat database as of January 4, 1999.

6. Appendix A.2 to this Declaration is the Library of Congress catalog record for Betz, accessed on March 2, 2023, having a catalog number 99011904. Appendix A.2 indicates that Betz is available in the Jefferson or Adams Building Reading Rooms (FLM2) at call number TK7895.G36 B48 1999. Appendix A.2 indicates that Betz was copyrighted in the United States in 1999.

7. Appendix A.3 to this declaration is the Library of Congress MARC Tag record for Betz, accessed on March 2, 2023. Appendix A.3 indicates that Betz is available in the Jefferson or Adams Building Reading Rooms at call number TK7895.G36 B48 1999. The formatted MARC data in Appendix A.3 shows that the original MARC record was entered into the Library of Congress catalog on January 4, 1999, and copyrighted in the United States in 1999.

8. Appendix A.4 to this declaration is a copy of SpringerLink's online record for Betz, accessed on March 2, 2023. Appendix A.4 indicates that Betz was first published on March 31, 1999.

9. Appendix A.5 to this declaration is a copy of Betz purchased by MTO. The copyright date in Appendix A.5 is 1999. I have compared Appendix A.5 to Ex.1006. Ex.1006 is a true and correct copy.

II. Ex.1011: Stephen D. Brown et al., *Field-Programmable Gate Arrays* (1992 Kluwer Academic Publishers)

10. Ex.1011 in the IPR Proceeding is Stephen D. Brown et al., *Field-Programmable Gate Arrays* (1992 Kluwer Academic Publishers) ("Brown-FPGA").

11. Appendix B.1 to this declaration is the WorldCat database record for Brown-FPGA, accessed on March 2, 2023, having OCLC reference number 25675383. Appendix B.1 indicates that the WorldCat database record for Brown-FPGA was entered into the WorldCat database on March 25, 1992. Appendix B.1

indicates that 207 libraries worldwide have copies of Brown-FPGA. Brown-FPGA would have been locatable and available to the public by search in the WorldCat database as of March 25, 1992.

12. Appendix B.2 to this Declaration is the Library of Congress catalog record for Brown-FPGA, accessed on March 2, 2023, having a catalog number 92013785. Appendix B.2 indicates that Brown-FPGA is available in the Jefferson or Adams Building Reading Rooms (FLM2) at call number TK7872.L64 F54 1992. Appendix B.2 indicates that Brown-FPGA was copyrighted in the United States in 1992.

13. Appendix B.3 to this declaration is the Library of Congress MARC Tag record for Brown-FPGA, accessed on March 2, 2023. Appendix B.3 indicates that Brown-FPGA is available in the Jefferson or Adams Building Reading Rooms at call number TK7872.L64 F54 1992. The formatted MARC data in Appendix B.3 shows that the original MARC record was entered into the Library of Congress catalog on March 25, 1992, and copyrighted in the United States in 1992.

14. Appendix B.4 to this declaration is a copy of SpringerLink's online record for Brown-FPGA, accessed on March 2, 2023. Appendix B.4 indicates that Brown-FPGA was first published on June 30, 1992.

15. Appendix B.5 to this declaration is a copy of Brown-FPGA purchased by MTO. The copyright date in Appendix B.5 is 1992. I have compared Appendix B.5 to Ex.1011. Ex.1011 is a true and correct copy.

III. Ex.1012: Stephen M. Trimberger et al., *Field-Programmable Gate Array Technology* (1994 Kluwer Academic Publishers)

16. Ex.1012 in the IPR Proceeding is Stephen M. Trimberger et al., *Field-Programmable Gate Array Technology* (1994 Kluwer Academic Publishers) (“Trimberger”).

17. Appendix C.1 to this declaration is the WorldCat database record for Trimberger, accessed on March 2, 2023, having OCLC reference number 29181710. Appendix C.1 indicates that the WorldCat database record for Trimberger was entered into the WorldCat database on October 1, 1993. Appendix C.1 indicates that 184 libraries worldwide have copies of Trimberger. Trimberger would have been locatable and available to the public by search in the WorldCat database as of October 1, 1993.

18. Appendix C.2 to this Declaration is the Library of Congress catalog record for Trimberger, accessed on March 2, 2023, having a catalog number 93039703. Appendix C.2 indicates that Trimberger is available in the Jefferson or Adams Building Reading Rooms (FLM2) at call number TK7895.G36 F54 1994. Appendix C.2 indicates that Trimberger was copyrighted in the United States in 1994.

19. Appendix C.3 to this declaration is the Library of Congress MARC Tag record for Trimberger, accessed on March 2, 2023. Appendix C.3 indicates that Trimberger is available in the Jefferson or Adams Building Reading Rooms at call number TK7895.G36 F54 1994. The formatted MARC data in Appendix C.3 shows that the original MARC record was entered into the Library of Congress catalog on October 1, 1993 and copyrighted in the United States in 1994.

20. Appendix C.4 to this declaration is a copy of SpringerLink's online record for Trimberger, accessed on March 2, 2023. Appendix C.4 indicates that Trimberger was first published on January 31, 1994.

21. Appendix C.5 to this declaration is a copy of Trimberger purchased by MTO. The copyright date in Appendix C.5 is 1994. I have compared Appendix E to Ex.1012. Ex.1012 is a true and correct copy.

IV. Ex.1021: Narasimha B. Bhat, "Novel Techniques for High Performance Field Programmable Logic Devices," Technical Report No. UCB/ERL M93/80, Electronic Research Laboratory, University of California, Berkeley (Nov. 1993)

22. Ex.1021 in the IPR Proceeding is Narasimha B. Bhat, "Novel Techniques for High Performance Field Programmable Logic Devices," Technical Report No. UCB/ERL M93/80, Electronic Research Laboratory, University of California, Berkeley (Nov. 1993) ("Bhat").

23. Appendix D.1 to this declaration is the WorldCat database record for Bhat, accessed on April 14, 2023, having OCLC reference number 215199002.

Appendix D.1 indicates that the WorldCat database record for Bhat was entered into the WorldCat database on September 6, 1994. Appendix D.1 indicates that the University of California, Berkeley has a copy of Bhat. Bhat would have been locatable and available to the public by search in the WorldCat database as of September 6, 1994.

24. Appendix D.2 to this Declaration is the University of California, Berkeley Library Digital Collections catalog record for Bhat, accessed on April 14, 2023, having a file number ERL-93-80. Appendix D.2 indicates that Bhat was published on November 1, 1993, and is available in the Engineering Library at the University of California, Berkeley, as part of the Electrical Engineering & Computer Sciences Technical Reports collection.

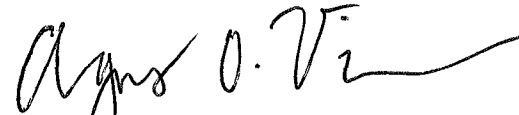
25. Appendix D.3 to this Declaration is the University of California, Berkeley Technical Memorandum Series catalog record for Bhat, accessed on April 14, 2023, having a file number M93/80. Appendix D.3 indicates that Bhat was published in November 1993.

26. Appendix D.4 to this declaration is a copy of Bhat purchased from ProQuest by MTO. The copyright date in Appendix D.4 is 1993. I have compared Appendix E to Ex.1021. Ex.1021 is a true and correct copy.

* * *

27. I declare under penalty of perjury under the laws of the United States of America that the foregoing is true and correct.

Executed on April 17, 2023, at Los Angeles, California



Agnes O. Villero

Appendix A.1

FirstSearch: Libraries that Own Item

 firstsearch.oclc.org/WebZ/FSFETCH

[English](#) | [Español](#) | [Français](#) | [عربي](#) | [日本語](#) | [한국어](#) | [中文\(繁體\)](#) | [中文\(简体\)](#)

LOS ANGELES PUB LIBR

[Libraries that Own Item](#) | [California State Library](#)

This screen shows libraries that own the item you selected.

[Home](#)

[Databases](#)

[Searching](#)

[Results](#)

[Staff View](#) | [My Account](#) | [Options](#)

| [Comments](#) | [Exit](#) | [Hide tips](#)



E-mail



Print



Return



Help

Current database: **WorldCat** Total Libraries: **115**



Title: Architecture and CAD for deep-submicron FPGAs **Author:** Betz, Vaughn **Accession Number:** 40602916

All Libraries that Own Item: "Architecture and CAD for ..."([Record for Item](#) | [Get This Item](#))

<u>Location</u>	<u>Library</u>	<u>Local Holdings</u>
<u>Shared Print Commitment</u>	<u>Code</u>	

US, CA ALBRS	ALIBRIS	
US, CA CIT	<u>CALIFORNIA INST OF TECH</u>	
US, CA CCH	<u>CALIFORNIA STATE UNIV CHICO</u>	
US, CA <u>Availa...</u>	<u>JET PROPULSION LAB</u>	<u>Local Holdings</u>
US, CA MFLFD	SSJ MORRISON & FOERSTER LLP LIBR	
US, CA STF	<u>STANFORD UNIV LIBR</u>	
US, CA CUY	<u>UNIV OF CALIFORNIA BERKELEY LIBR</u>	
US, CA CUV	<u>UNIV OF CALIFORNIA, DAVIS, SHIELDS LIBR</u>	
US, CA CUI	<u>UNIV OF CALIFORNIA, IRVINE</u>	
US, CA CLU	<u>UNIV OF CALIFORNIA, LOS ANGELES</u>	
US, CA CUS	UNIV OF CALIFORNIA, SAN DIEGO	
US, CA CSL	<u>UNIV OF SOUTHERN CALIFORNIA</u>	
US, AL AAA	<u>AUBURN UNIV</u>	
US, AR AFU	<u>UNIV OF ARKANSAS MULLINS</u>	
US, AZ AZS	<u>ARIZONA STATE UNIV TEMPE</u>	
US, CO COF	<u>COLORADO STATE UNIV FT COLLINS</u>	
US, CO COD	<u>UNIV OF COLORADO AT BOULDER</u>	
US, DC DLC	<u>LIBRARY OF CONGRESS</u>	
US, FL EBC	<u>EASTERN FLORIDA STATE COLL LIBR</u>	
US, FL FTU	<u>UNIV OF CENT FLORIDA LIBR</u>	
US, FL FHM	<u>UNIV OF S FLORIDA LIBR</u>	
US, GA GAT	<u>GEORGIA INST OF TECHNOL MAIN CAM LIBR</u>	
US, HI HUH	<u>UNIV OF HAWAII AT MANOA LIBR</u>	

US, IA IWA	<u>IOWA STATE UNIV</u>		
US, IL UIU	<u>UNIV OF ILLINOIS</u>		
US, IN IND	<u>UNIV OF NOTRE DAME</u>		
US, KY <u>Availa...</u>	<u>UNIV OF LOUISVILLE</u>	KLK	<u>Local Holdings</u>
US, MA LIN	MASSACHUSETTS INST OF TECH, LINCOLN LAB		
US, MA MYG	<u>MASSACHUSETTS INST OF TECHNOL LIBRS</u>		
US, MA <u>Availa...</u>	<u>TUFTS UNIV</u> <u>Committed to Retain</u>	TFW	<u>Local Holdings</u>
US, MD JHE	<u>JOHNS HOPKINS UNIV</u>		
US, MD NAG	NASA GODDARD SPACE FLIGHT CTR LIBR		
US, MD <u>Availa...</u>	<u>UNIV OF MARYLAND BALTIMORE CNTY</u> <u>Committed to Retain</u>	MUB	<u>Local Holdings</u>
US, MD <u>Availa...</u>	<u>UNIV OF MARYLAND, COL PARK</u>	UMC	<u>Local Holdings</u>
US, MN <u>Availa...</u>	<u>SAINT CLOUD STATE UNIV</u> <u>Committed to Retain</u>	MST	<u>Local Holdings</u>
US, MO XII	<u>SAINT LOUIS UNIV, PIUS XII LIBR</u>		
US, MO MUU	<u>UNIV OF MISSOURI--COLUMBIA</u>		
US, NC BTCTA	BAKER & TAYLOR INC TECH SERV & PROD DEV		
US, NC NDD	<u>DUKE UNIV LIBR</u>		
US, NC <u>Availa...</u>	<u>NORTH CAROLINA A&T STATE UNIV</u> NQA		<u>Local Holdings</u>
US, NC NRC	<u>NORTH CAROLINA STATE UNIV AT RALEIGH LI</u>		
US, NE LDL	<u>UNIV OF NEBRASKA AT LINCOLN</u>		
US, NH YDX	GOBI LIBR SERVS		
US, NM SNX	SANDIA NAT LAB, TECH LIBR		

US, NM <u>Availa...</u>	<u>UNIV OF NEW MEXICO</u>	IQU	<u>Local Holdings</u>
US, NY <u>Availa...</u>	<u>CORNELL UNIV</u>	C00	<u>Local Holdings</u>
US, NY NYP	<u>NEW YORK PUB LIBR</u>		
US, NY RVE	<u>ROCHESTER INST OF TECHNOL LIBR</u>		
US, OH BAKER	<u>BAKER & TAYLOR</u>		
US, OH GGTST	<u>GREEN GLASS TEST 1</u>		
US, OH CIN	<u>UNIV OF CINCINNATI</u>		
US, OH WSU	<u>WRIGHT STATE UNIV</u>		
US, PA PMC	<u>CARNEGIE MELLON UNIV</u>		
US, PA UPM	<u>PENNSYLVANIA STATE UNIV</u>		
US, TN TKN	<u>UNIV OF TENNESSEE</u>		
US, TX RCE	<u>RICE UNIV, FONDREN LIBR</u>		
US, TX TXA	<u>TEXAS A&M UNIV</u>		
US, TX TXI	<u>TEXAS STATE UNIV-SAN MARCOS</u>		
US, TX ILU	<u>TEXAS TECH UNIV</u>		
US, TX INT	<u>UNIV OF N TEXAS</u>		
US, TX IXA	<u>UNIV OF TEXAS AT AUSTIN</u>		
US, TX ITD	<u>UNIV OF TEXAS AT DALLAS</u>		
US, UT UBY	<u>BRIGHAM YOUNG UNIV PROVO</u>		
US, UT UUM	<u>UNIV OF UTAH</u>		
US, UT UUS	<u>UTAH STATE UNIV</u>		
US, VA VRC	<u>VIRGINIA COMMONWEALTH UNIV LIBR</u>		
US, WI GZQ	<u>MARQUETTE UNIV</u>		
US, WI GZM	<u>UNIV OF WISCONSIN, MADISON, GEN LIBR SYS</u>		

Australia RM0	<u>RMIT LIBR</u>		
Australia LJ1	<u>UNIV OF NEW S WALES</u>		
Australia YQU	<u>UNIV OF QUEENSLAND</u>		
CA, AB UAU	<u>UNIV OF CALGARY LIBR</u>		
CA, BC SFB	<u>SIMON FRASER UNIV</u>		
CA, BC UBC	<u>UNIV OF BRITISH COLUMBIA LIBR</u>		
CA, BC VT2	<u>UNIV OF VICTORIA LIBRS</u>		
CA, NB <u>Availa...</u>	UNIV OF NEW BRUNSWICK	UNBCA	<u>Local Holdings</u>
CA, NL <u>availa...</u>	<u>MEMORIAL UNIV/NEWFOUNDLAND, ELIZABETH II</u> QE2		<u>Local holdings</u>
CA, ON CEF	<u>CARLETON UNIV</u>		
CA, ON CAI	<u>NATIONAL SCI LIBRARY/BIBLIO SCI NAT</u>		
CA, ON CNECS	<u>UNIV OF TORONTO ENG & COMP LIBR</u>		
CA, ON CNUTO	UNIV OF TORONTO GROUP		
CA, ON UWW	UNIV OF WATERLOO LIBR		
CA, QC <u>Availa...</u>	<u>MCGILL UNIV</u>	LGG	<u>Local Holdings</u>
France FRBUJ	BREST BIB INFORMATIQUE		
France FRAHY	CERGY-ENSEA		
France ELYON	ECOLE CENTRALE DE LYON		
France E6L	ECOLE NORMAL AT SUPERIEURE DE LYON		
France FRAQN	LYON-BIU-LSH		
Germany F3U	FREIE UNIV BERLIN		
Germany OHX	HARRASSOWITZ		
Germany DEKAP	HOCHSCHLBIBLIOTHEK PFORZHEIM BG		
Germany DEDDS	<u>SACHSISCHE LANDESBIBLIOTHEK</u>		
Germany DEKAB	UNIV KARLSRUHE		

Germany DEUEN Germany DESTU	UNIVERSITÄT, BIBLIOTHEK ERLANGEN-NÜRNBERG UNIVERSITÄT STUTTGART	
Hong Kong HNK Hong Kong HUA	<u>HONG KONG UNIV OF SCI & TECH, THE</u> <u>UNIV OF HONG KONG</u>	
Japan C6Y	CHUKYO UNIV LIBR	
Malaysia APUMS	UNIV MALAYSIA SABAH	
Mexico I8M	<u>INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUP</u>	
Singapore SINTU Singapore SINAP Singapore SITPL Singapore SIUAL	NANYANG TECHNOLOGICAL UNIV NGEE ANN POLYTECHNIC LIBR TEMASEK POLYTECHNIC LIBRARY UNSW ASIA LIBR SVC	
Taiwan CH4GT Taiwan CKT Taiwan R00 Taiwan NTU	CHANG GUNG UNIV <u>NATIONAL CHENG KUNG UNIV</u> <u>NATIONAL TAIWAN OCEAN UNIV</u> <u>NATIONAL TAIWAN UNIV</u>	
United Kingdom <u>Availa...</u> United Kingdom BRI United Kingdom UKM United Kingdom BLSTP United Kingdom LIP United Kingdom EUM United Kingdom ESU	BOURNEMOUTH UNIV LIS <u>BRITISH LIBR</u> BRITISH LIBR BRITISH LIBR REFERENCE COLLECTIONS IMPERIAL COL LONDON UNIV OF MANCHESTER LIBR THE <u>UNIV OF STRATHCLYDE</u>	<u>Local Holdings</u> B0@

Record for Item: "Architecture and CAD for ..."(Libraries with Item)

--

Get This Item

Access: <http://catdir.loc.gov/catdir/enhancements/fy0813/99011904-t.html>

Availability: Check the catalogs in your library.

- [Libraries worldwide that own item](#): 115 (3 Committed to Retain)
-  [Search the catalog at your library](#).

External Resources: • [Cite This Item](#)

Find Related

More Like This: [Search for versions with same title and author](#) | [Advanced options ...](#)

Find Items About: [Rose, Jonathan](#), (max: 5)

Title:

Architecture and CAD for deep-submicron FPGAs /

Author(s): [Betz, Vaughn](#).
[Rose, Jonathan](#),; 1958-
[Marquardt, Alexander](#).

Publication: Boston : Kluwer Academic,

Year: 1999

Description: xi, 247 pages : illustrations ; 24 cm.

Language: English

Series: The Kluwer international series in engineering and computer science ;; SECS 497; **Variation:** Kluwer international series in engineering and computer science ;; SECS 497.

Standard No: **ISBN:** 0792384601; 9780792384601; **National Library:** 008431346; B9905741; 006468760; **LCCN:** 99-11904

Abstract: Architecture and CAD for Deep-Submicron FPGAs addresses several key issues in the design of high-performance FPGA architectures and CAD tools, with particular emphasis on issues that are important for FPGAs implemented in deep-submicron processes. Three factors combine to determine the performance of an FPGA: the quality of the CAD tools used to map circuits into the FPGA, the quality of the FPGA architecture, and the electrical (i.e. transistor-level) design of the FPGA. Architecture and CAD for Deep-Submicron FPGAs examines all three of these issues in concert.

Contents: Ch. 1.; Introduction --; Ch. 2.; Background and Previous Work.; 2.1.; FPGA Architecture.; 2.2.; CAD for FPGAs --; Ch. 3.; CAD Tools: Packing and Placement.; 3.1.; Logic Block Packing.; 3.2.; Placement: VPR --; Ch. 4.; Routing Tools and Routing Architecture Generation.; 4.1.; Position within the CAD flow.; 4.2.; Architecture Parameterization and Generation.; 4.3.; Routability-Driven Router.; 4.4.; Timing-Driven Router.; 4.5.; Delay Extraction and Timing Analysis.; 4.6.; Router and Placement Algorithm Validation --; Ch. 5.; Global Routing Architecture.; 5.1.; Motivation.; 5.2.; Experimental Methodology.; 5.3.; Experimental Results: Directionally-Biased Routing.; 5.4.; Experimental Results: Non-Uniform Routing --; Ch. 6.; Cluster-Based Logic Blocks.; 6.1.; Motivation.

Access: **Materials specified:** Table of contents<http://catdir.loc.gov/catdir/enhancements/fy0813/99011904-t.html>
Materials specified: Publisher description<http://catdir.loc.gov/catdir/enhancements/fy0813/99011904-d.html>
Materials specified: 1850-9999<http://www.springer.com/gb/>

SUBJECT(S)

Descriptor: Field programmable gate arrays -- Computer-aided design.
Programmable array logic.
Réseaux logiques programmables par l'utilisateur -- Conception assistée par ordinateur.
Logique à réseau programmable.
Programmable array logic.
CAD
Field programmable gate array.
PROGRAMMABLE LOGIC DEVICES.
GATE (CIRCUITS).
ARRAYS.
COMPUTER AIDED DESIGN.
LOGIC CIRCUITS.
Réseaux logiques programmables par l'utilisateur -- Conception assistée par ordinateur.
Réseaux logiques programmables.
CAD.

Note(s): Includes bibliographical references (pages 233-242).

General Info: **National bibliography no:** GB9932743

Class Descriptors: **LC:** TK7895.G36; **Dewey:** 621.39/5

Responsibility: Vaughn Betz, Jonathan Rose, Alexander Marquardt.

Vendor Info: Baker & Taylor Baker and Taylor Otto Harrassowitz Ingram YBP Library Services (BKTY BTCP HARR INGR YANK) 265.00 **Status:** active

Material Type: Internet resource (url)

Document Type: Book; Internet Resource

Date of Entry:	19990104
Update:	20220508
Accession No:	OCLC: 40602916
Database:	WorldCat

 E-mail
  Print
  Return
  Help

Current database: **WorldCat** Total Libraries: **115**



[English](#) | [Español](#) | [Français](#) | [عربي](#) | [日本語](#) | [한국어](#) | [中文\(繁體\)](#) | [中文\(简体\)](#) | [Options](#) | [Comments](#) | [Exit](#)

 © 1992-2023 OCLC
[Terms & Conditions](#)

For information on how your personal data is processed when you use FirstSearch, please see your institution's privacy policy.

Appendix A.2

BookArchitecture and CAD for deep-submicron FPGAs

 catalog.loc.gov/vwebv/search

Links

Links

- Table of contents only <http://www.loc.gov/catdir/enhancements/fy0813/99011904-t.html> opens in a new window
- Publisher description <http://www.loc.gov/catdir/enhancements/fy0813/99011904-d.html> opens in a new window

More Information

LCCN Permalink

<https://lcn.loc.gov/99011904>

Description

xi, 247 p. : ill. ; 24 cm.

ISBN

0792384601 (alk. paper)

LC classification

TK7895.G36 B48 1999

Browse by shelf order

TK7895.G36

Notes

Includes bibliographical references (p. [233]-242).

Series

The Kluwer international series in engineering and computer science ; SECS 497

LCCN

99011904

Dewey class no.

621.39/5

Type of material

Book

Item Availability

CALL NUMBER

- TK7895.G36 B48 1999
- Copy 1

Request in

Jefferson or Adams Building Reading Rooms

Status

Not Charged

CALL NUMBER

- TK7895.G36 B48 1999 FT MEADE
- Copy 2

Request in

Jefferson or Adams Building Reading Rooms - STORED OFFSITE

Status

Not Charged

Appendix A.3

BookArchitecture and CAD for deep-submicron FPGAs

 catalog.loc.gov/vwebv/staffView

- 00001433cam a22003374a 4500
- 0011684340
- 00520140818110559.0
- 008990104s1999 maua b 001 o eng
- 906__ |a 7 |b cbc |c orignew |d 1 |e ocip |f 19 |g y-gencatlg
- 955__ |a pc06 to ja00 01-05-99; jg14 01-04-99; jf08 01-05-99 to SL;jf12 01-06-99 to ddc; cip ver jfo9 3-25-99
- 010__ |a 99011904
- 020__ |a 0792384601 (alk. paper)
- 035__ |9 (DLC) 99011904
- 040__ |a DLC |c DLC |d DLC
- 042__ |a pcc
- 05000 |a TK7895.G36 |b B48 1999
- 08200 |a 621.39/5 |2 21
- 1001_ |a Betz, Vaughn.
- 24510 |a Architecture and CAD for deep-submicron FPGAs / |c Vaughn Betz, Jonathan Rose, Alexander Marquardt.
- 260__ |a Boston : |b Kluwer Academic, |c c1999.
- 300__ |a xi, 247 p. : |b ill. ; |c 24 cm.
- 440_4 |a The Kluwer international series in engineering and computer science ; |v SECS 497
- 504__ |a Includes bibliographical references (p. [233]-242).
- 650_o |a Field programmable gate arrays |x Computer-aided design.
- 650_o |a Programmable array logic.
- 7001_ |a Rose, Jonathan, |d 1958-
- 7001_ |a Marquardt, Alexander.
- 85641 |3 Table of contents
only |u <http://www.loc.gov/catdir/enhancements/fy0813/99011904-t.html>
- 85642 |3 Publisher
description |u <http://www.loc.gov/catdir/enhancements/fy0813/99011904-d.html>
- 920__ |a ** LC HAS REQ'D # OF SHELF COPIES **
- 991__ |b c-GenColl |h TK7895.G36 |i B48 1999 |t Copy 1 |w BOOKS

Item Availability

CALL NUMBER

- TK7895.G36 B48 1999
- Copy 1

Request in

Jefferson or Adams Building Reading Rooms

Status

Not Charged

CALL NUMBER

- TK7895.G36 B48 1999 FT MEADE
- Copy 2

Request in

Jefferson or Adams Building Reading Rooms - STORED OFFSITE

Status

Not Charged

Appendix A.4

Architecture and CAD for Deep-Submicron FPGAS

 link.springer.com/book/10.1007/978-1-4615-5145-4

- Book
- © 1999

1. [Home](#) >
2. Book

Part of the book series: [The Springer International Series in Engineering and Computer Science](#) (SECS, volume 497)

- 3719 Accesses
- 729 [Citations](#)
- 1 [Altmetric](#)

This is a preview of subscription content, [access via your institution](#).

Table of contents (8 chapters)

1. **Front Matter**

Pages i-xi

[PDF](#) 

2. **Introduction**

Vaughn Betz, Jonathan Rose, Alexander Marquardt

Pages 1-9

3. **Background and Previous Work**

Vaughn Betz, Jonathan Rose, Alexander Marquardt

Pages 11-35

4. **CAD Tools: Packing and Placement**

Vaughn Betz, Jonathan Rose, Alexander Marquardt

Pages 37-61

5. **Routing Tools and Routing Architecture Generation**

Vaughn Betz, Jonathan Rose, Alexander Marquardt
Pages 63-103

6. **Global Routing Architecture**

Vaughn Betz, Jonathan Rose, Alexander Marquardt
Pages 105-126

7. **Cluster-Based Logic Blocks**

Vaughn Betz, Jonathan Rose, Alexander Marquardt
Pages 127-149

8. **Detailed Routing Architecture**

Vaughn Betz, Jonathan Rose, Alexander Marquardt
Pages 151-190

9. **Conclusions and Future Work**

Vaughn Betz, Jonathan Rose, Alexander Marquardt
Pages 191-198

10. **Back Matter**

Pages 199-247

[PDF](#) 

[Back to top](#) 

About this book

Since their introduction in 1984, Field-Programmable Gate Arrays (FPGAs) have become one of the most popular implementation media for digital circuits and have grown into a \$2 billion per year industry. As process geometries have shrunk into the deep-submicron region, the logic capacity of FPGAs has greatly increased, making FPGAs a viable implementation alternative for larger and larger designs. To make the best use of these new deep-submicron processes, one must re-design one's FPGAs and Computer-Aided Design (CAD) tools.

Architecture and CAD for Deep-Submicron FPGAs addresses several key issues in the design of high-performance FPGA architectures and CAD tools, with particular emphasis on issues that are important for FPGAs implemented in deep-submicron processes.

Three factors combine to determine the performance of an FPGA: the quality of the CAD

tools used to map circuits into the FPGA, the quality of the FPGA architecture, and the electrical (i.e. transistor-level) design of the FPGA. *Architecture and CAD for Deep-Submicron FPGAs* examines all three of these issues in concert.

In order to investigate the quality of different FPGA architectures, one needs CAD tools capable of automatically implementing circuits in each FPGA architecture of interest. Once a circuit has been implemented in an FPGA architecture, one next needs accurate area and delay models to evaluate the quality (speed achieved, area required) of the circuit implementation in the FPGA architecture under test. This book therefore has three major foci: the development of a high-quality and highly *flexible* CAD infrastructure, the creation of accurate area and delay models for FPGAs, and the study of several important FPGA architectural issues.

Architecture and CAD for Deep-Submicron FPGAs is an essential reference for researchers, professionals and students interested in FPGAs.

[Back to top](#) ↑

Keywords

- FPGA
- Field Programmable Gate Array
- architecture
- computer
- computer-aided design (CAD)
- development
- gate array
- geometry
- logic
- modeling
- transistor
- visualization

[Back to top](#) ↑

Authors and Affiliations

University of Toronto, USA

Vaughn Betz, Jonathan Rose, Alexander Marquardt

[Back to top](#) ↑

Bibliographic Information

- Book TitleArchitecture and CAD for Deep-Submicron FPGAS

- AuthorsVaughn Betz, Jonathan Rose, Alexander Marquardt
- Series Title The Springer International Series in Engineering and Computer Science
- DOI<https://doi.org/10.1007/978-1-4615-5145-4>
- PublisherSpringer New York, NY
- eBook Packages Springer Book Archive
- Copyright InformationSpringer Science+Business Media New York 1999
- Hardcover ISBN978-0-7923-8460-1Published: 31 March 1999
- Softcover ISBN978-1-4613-7342-1Published: 02 November 2012
- eBook ISBN978-1-4615-5145-4Published: 06 December 2012
- Series ISSN 0893-3405
- Edition Number1
- Number of PagesXI, 247
- Topics

[Back to top](#) ↑

[Access via your institution](#) →

Buying options

› eBook USD 269.00

Price excludes VAT (USA)

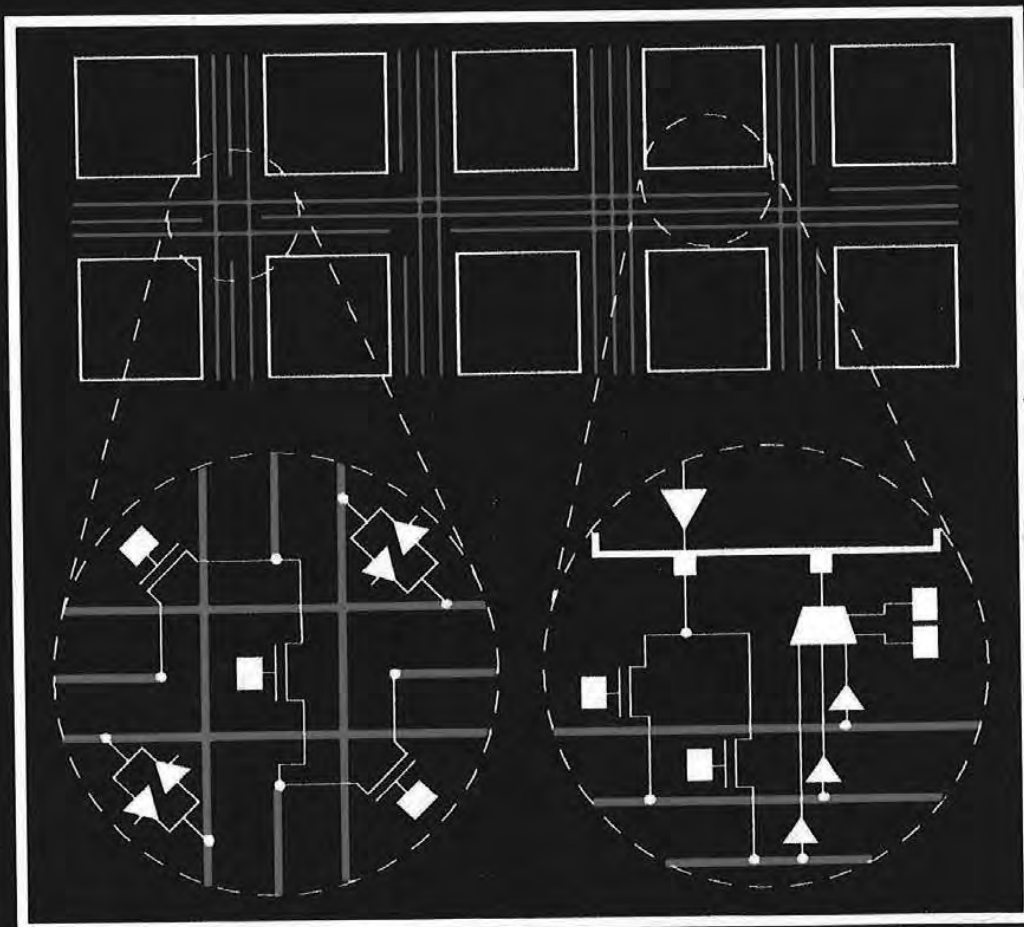
- ISBN: 978-1-4615-5145-4
- Instant PDF download
- Readable on all devices
- Own it forever
- Exclusive offer for individuals only
- Tax calculation will be finalised during checkout

› Softcover Book USD 349.99

› Hardcover Book USD 349.99

[Learn about institutional subscriptions](#)

Architecture and CAD for Deep-Submicron FPGAs



SCIENCE
TK
7895
G36
B48
1999

Vaughn Betz
Jonathan Rose
Alexander Marquardt

ARCHITECTURE AND CAD FOR DEEP-SUBMICRON FPGAs

**THE KLUWER INTERNATIONAL SERIES
IN ENGINEERING AND COMPUTER SCIENCE**

ARCHITECTURE AND CAD FOR DEEP-SUBMICRON FPGAs

**Vaughn Betz
Jonathan Rose
Alexander Marquardt**
University of Toronto



KLUWER ACADEMIC PUBLISHERS
Boston / Dordrecht / London

Distributors for North, Central and South America:

Kluwer Academic Publishers
101 Philip Drive
Assinippi Park
Norwell, Massachusetts 02061 USA
Telephone (781) 871-6600
Fax (781) 871-6528
E-Mail <kluwer@wkap.com>

Distributors for all other countries:

Kluwer Academic Publishers Group
Distribution Centre
Post Office Box 322
3300 AH Dordrecht, THE NETHERLANDS
Telephone 31 78 6392 392
Fax 31 78 6546 474
E-Mail <orderdept@wkap.nl>



Electronic Services <<http://www.wkap.nl>>

Library of Congress Cataloging-in-Publication Data

A C.I.P. Catalogue record for this book is available
from the Library of Congress.

Copyright © 1999 by Kluwer Academic Publishers

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher, Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, Massachusetts 02061

Printed on acid-free paper.

Printed in the United States of America

To Corinne, William, and Isabel,

Betty and Frank Rose,

Marie and Ron

Table of Contents

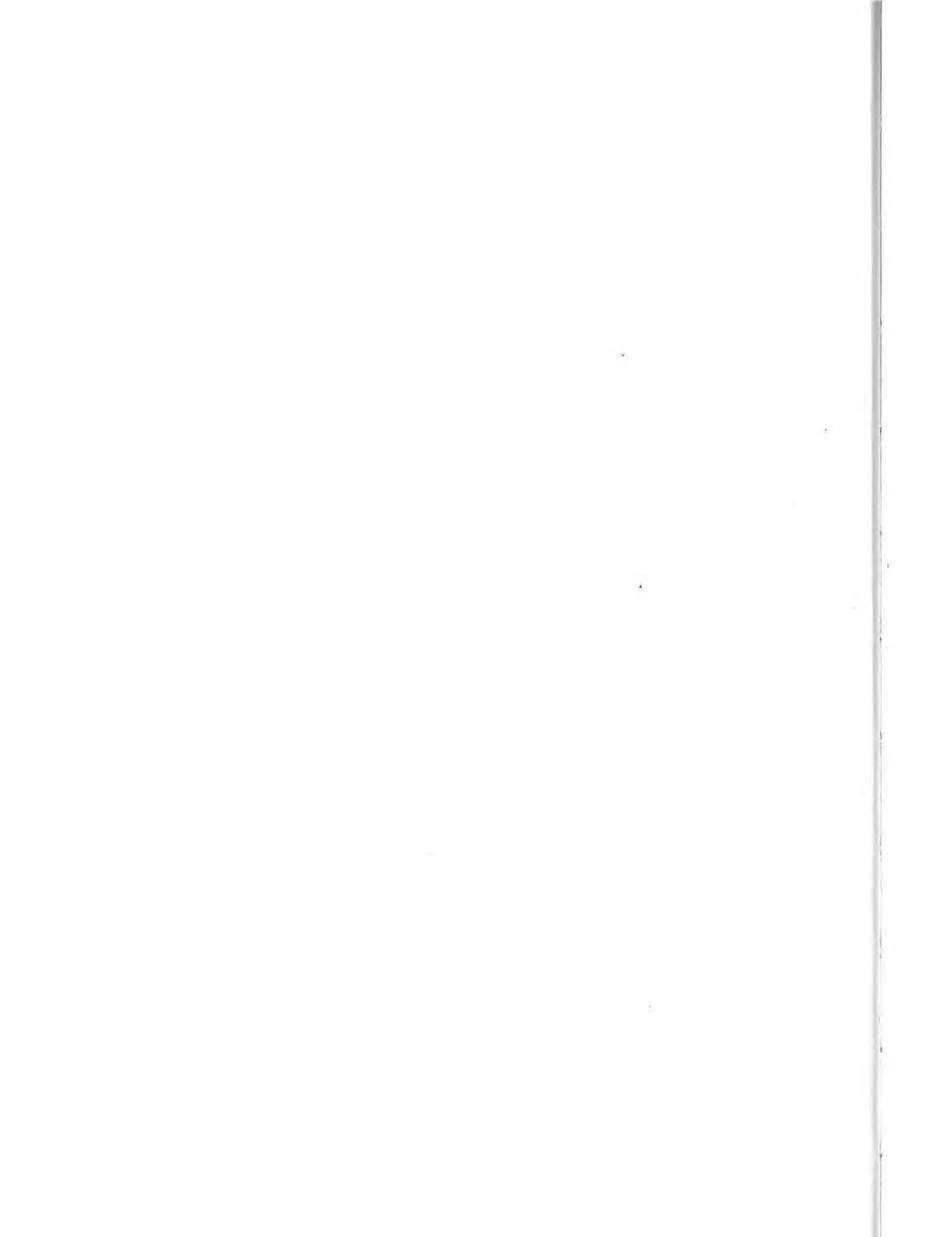
CHAPTER 1	<i>Introduction</i>	1
1.1	Overview of FPGAs	2
1.2	FPGA Architectural Issues	3
1.3	Approach and CAD Tools	7
1.4	Book Organization	8
1.5	Acknowledgments	9
CHAPTER 2	<i>Background and Previous Work</i>	11
2.1	FPGA Architecture	11
2.1.1	FPGA Programming Technologies	11
2.1.2	FPGA Logic Block Architecture	13
2.1.3	FPGA Routing Architecture	14
2.2	CAD for FPGAs	18
2.2.1	Synthesis and Logic Block Packing	19
2.2.2	Placement	22
2.2.3	Routing	25
2.2.4	Delay Modelling	31
2.2.5	Timing Analysis	32
2.3	Summary	34
CHAPTER 3	<i>CAD Tools: Packing and Placement</i>	37
3.1	Logic Block Packing	37
3.1.1	Cluster-Based Logic Blocks	38
3.1.2	Basic Logic Block Packing Algorithm: VPack	39
3.1.3	Timing-Driven Logic Block Packing: T-VPack	43
	Cluster Seed and Attraction Function	43
	Computational Complexity vs. Frequency of Timing Analysis	47

3.1.4	Result Quality of T-VPack vs. VPack	48
3.2	Placement: VPR	50
3.2.1	Overview of the VPR Placement Tool	51
3.2.2	New Adaptive Annealing Schedule	52
3.2.3	New Cost Function: Linear Congestion	55
3.2.4	Incremental Net Bounding Box Updates	58
3.3	Summary	61
CHAPTER 4 Routing Tools and Routing Architecture Generation		63
4.1	Position within the CAD flow	63
4.2	Architecture Parameterization and Generation	64
4.2.1	Architecture Parameterization	65
4.2.2	The Routing-Resource Graph	68
4.2.3	Automatic Architecture Generation from Parameters	70
4.3	Routability-Driven Router	76
4.3.1	Cost Functions and Routing Schedules	76
4.3.2	Speed Enhancements	79
4.4	Timing-Driven Router	80
4.4.1	Superiority of Elmore Delay to the Linear Delay Model	80
4.4.2	Directly Optimizing the Elmore Delay	83
4.4.3	Net Routing Algorithm Complexity	90
4.4.4	Dynamic Base Costs	90
4.4.5	Routing Schedule	92
4.5	Delay Extraction and Timing Analysis	94
4.6	Router and Placement Algorithm Validation	95
4.6.1	Routability-Driven Router and Placement Algorithm	95
	Experimental Results with Input-Pin Doglegs.....	96
	Experimental Results Without Input Pin Doglegs.....	97
4.6.2	Timing-Driven Router	99
4.7	Summary	103
CHAPTER 5 Global Routing Architecture		105
5.1	Motivation	105
5.2	Experimental Methodology	107

5.2.1	CAD Flow	107
5.2.2	Area-Efficiency Metric	109
5.2.3	Significant FPGA Architectural Details	109
5.3	Experimental Results: Directionally-Biased Routing	110
5.3.1	Results for Square Logic Block Arrays	111
5.3.2	Results for Rectangular Logic Block Arrays	113
5.4	Experimental Results: Non-Uniform Routing	115
5.4.1	Center/Edge Capacity Ratio	116
5.4.2	Single Center Channel	120
5.4.3	I/O Channel	122
5.5	Summary	126
CHAPTER 6	<i>Cluster-Based Logic Blocks</i>	127
6.1	Motivation	127
6.2	Experimental Methodology	130
6.2.1	CAD Flow	130
6.2.2	Area Model	132
6.2.3	Delay Model	134
6.2.4	Architecture Evaluation Metric: Area-Delay Product	136
6.2.5	FPGA Architectural Assumptions	136
	Basic Architecture	137
	Routing Architecture	137
	Effect of Cluster Size on the Physical Length of FPGA	
	Routing Segments	138
	Sizing Routing Transistors to Compensate for Different	
	Physical Segment Lengths	138
6.3	Cluster Inputs Required vs. Cluster Size	139
6.4	Flexibility of Logic Block to Routing Interconnect vs. Cluster Size	141
6.5	Speed and Area-Efficiency vs. Cluster Size	142
6.5.1	Discussion of Delay vs. Cluster Size Results	145
6.6	Effect of Cluster Size on Compile Time	147
6.7	Summary	148

CHAPTER 7 Detailed Routing Architecture	151
7.1 Motivation	152
7.2 Experimental Methodology	153
7.2.1 FPGA Architectural Assumptions	154
7.2.2 CAD Flow	155
7.2.3 Delay Model Accuracy	156
7.2.4 Area Model	157
Importance of a Detailed Area Model	158
7.2.5 Experimental Philosophy	159
7.3 Single Wire Length Architectures	159
7.3.1 Switch Block Issues	160
7.3.2 Best Single Wire Length	163
7.3.3 Amount of Connectivity Between Logic Blocks and Channels	165
7.4 Two Types of Wire Segment Architectures	166
7.4.1 Tri-State Buffer Routing Switches Only	166
7.4.2 Length 4 Buffered Wires Plus Pass-Transistor-Switched Wires	167
7.4.3 Length 8 Buffered Wires Plus Pass-Transistor-Switched Wires	172
7.4.4 Length 4 Pass-Transistor-Switched Wires Plus Buffered Wires	172
7.5 Internal Population	175
7.5.1 All Length 4 Buffered Wires	175
7.5.2 Two-Wire-Type Architectures	181
7.6 Wire Spacing for Speed	183
7.7 Overall Architecture Comparison	185
7.8 Summary	189
 CHAPTER 8 Conclusions and Future Work	 191
8.1 Summary and Contributions	191
8.2 Future Work	196
8.2.1 CAD Tool Enhancements	196
8.2.2 Future FPGA Architecture Research	197
 APPENDIX A Graphic Visualization in VPR	 199
 APPENDIX B FPGA Circuitry and Process Modeling	 207
B.1 Transistor-Level Schematics and Assumptions	207

B.1.1	FPGA Routing Structures	208
	Gate Boosting	208
	Buffers	210
	Connection Block to Logic Block Input Pins	212
B.1.2	Logic Block Structures	214
B.2	Delay and RC-Equivalent Circuit Extraction	217
APPENDIX C <i>Sizing of Routing Transistors and Metal</i>		221
C.1	Sizing Pass Transistor Routing Switches	221
C.2	Sizing Tri-State Buffer Routing Switches	225
C.3	Tri-State Buffers in Output Pin Connection Blocks	228
C.4	Metal Width and Spacing	228
References		233
Index		243



Since their introduction in 1984, Field-Programmable Gate Arrays (FPGAs) have become one of the most popular implementation media for digital circuits and have grown into a \$2 billion per year industry. As process geometries have shrunk into the deep-submicron region, the logic capacity of FPGAs has greatly increased, making FPGAs a viable implementation alternative for larger and larger designs. To make the best use of these new deep-submicron processes, one must re-architect one's FPGAs and Computer-Aided Design (CAD) tools. This book addresses several key issues in the design of high-performance FPGA architectures and CAD tools, with particular emphasis on issues that are important for FPGAs implemented in deep-submicron processes.

Three factors combine to determine the performance of an FPGA: the quality of the CAD tools used to map circuits into the FPGA, the quality of the FPGA architecture, and the electrical (i.e. transistor-level) design of the FPGA. In this book, we examine all three of these issues in concert, and we believe this is the first systematic study of FPGA architecture and CAD to do so.

In order to investigate the quality of different FPGA architectures, one needs CAD tools capable of automatically implementing circuits in each FPGA architecture of interest. Once a circuit has been implemented in an FPGA architecture, one next needs accurate area and delay models to evaluate the quality (speed achieved, area required) of the circuit implementation in the FPGA architecture under test. This book therefore has three major foci: the development of a high-quality and highly flexible CAD infrastructure, the creation of accurate area and delay models for FPGAs, and the study of several important FPGA architectural issues.

Throughout this book we compare FPGA architectures by estimating the area each requires and the speed each achieves in a deep-submicron ($0.35\text{ }\mu\text{m}$) process. Our area model is a detailed estimate of the layout area required by an FPGA, and our delay model includes important deep-submicron effects, such as the significant resistance and capacitance of metal lines, that have often been neglected in earlier research. We believe that many FPGA architectural issues can be adequately studied only with detailed models of FPGA circuitry implemented in the target fabrication process, so this book discusses such issues extensively.

Industrial FPGA research and development tends to focus on point solutions — industrial architects make reasonable design choices and educated guesses to get a product to market in a timely fashion. The documentation of industrial devices do not indicate which design choices were carefully researched and which were educated guesses, and they usually do not investigate (or at least do not publish) the effect radically different design choices would have had. In this book we deliver precisely this kind of knowledge — all the information leading to good choices for a broad set of FPGA architectures, and complete discussions of the trade-offs involved. We determine the best choice for various architecture parameters, the sensitivity of FPGA performance to each parameter, and useful “rules of thumb” for designing good FPGA architectures.

1.1 Overview of FPGAs

The key to FPGAs’ popularity is their ability to implement *any* circuit simply by being appropriately programmed. Other circuit implementation options, such as Standard Cells or Mask-Programmed Gate Arrays (MPGAs), require that a different VLSI chip be newly fabricated for each design. The use of a standard FPGA, rather than these custom technologies, has two key benefits: lower non-recurring engineering (NRE) costs, and faster time-to-market.

To implement a circuit in an MPGA or with Standard Cells, one sends the completed design to a silicon foundry which manufactures a chip to implement exactly (and only) that design. The non-recurring engineering (NRE) fees to have the first chip manufactured are typically between \$100 000 and \$250 000; these fees cover the cost of making lithography masks for the circuit and of running a new design through the fabrication plant. On the other hand, a design is implemented in an FPGA simply by programming the FPGA (a standard part) to have the desired functionality, so there are no NRE costs. This makes FPGAs the lowest cost implementation medium for small and medium volume designs.

Time-to-market is the other key advantage of FPGAs. Full fabrication typically takes 6 - 8 weeks. If problems are found in the finished chip it must be thrown away, and one must wait another 6 - 8 weeks to fabricate a corrected design. FPGAs, on the other hand, can be programmed in seconds, and any bugs found once the chip is tested in system can be corrected in minutes by reprogramming the FPGA. With today's short product cycles, the time-to-market advantage this provides is often compelling.

FPGA programmability carries a price, however. In MPGAs and Standard Cells circuitry is interconnected with metal wires. FPGAs, in contrast, must connect circuitry via programmable switches. These switches have higher resistance than metal wires and add significant capacitance to connections, reducing circuit speed. The switches also take up more area than metal wires would, so an FPGA must be considerably larger than an MPGA to implement the same circuit. A circuit implemented in an FPGA is typically 10 times larger and roughly 3 times slower than the same circuit implemented via an MPGA in an equivalent process [1]. The larger size of FPGA circuitry makes FPGA implementations more expensive than MPGAs for high volume designs, and the limited speed of FPGAs precludes their use in very high-speed designs. These differences mandate research into new FPGA architectures in order to reduce these speed and density penalties. In addition, because the FPGA marketplace is highly competitive each FPGA manufacturer is constantly searching for better FPGA architectures in order to gain a speed and density advantage.

1.2 FPGA Architectural Issues

All FPGAs consist of a large number of programmable logic blocks, each of which implement a small amount of digital logic, and programmable routing which allows the logic block inputs and outputs to be connected to form larger circuits. In this book we investigate three different issues in FPGA architecture: two concern FPGA routing design, and one concerns FPGA logic block design.

The first issue we investigate is *global routing* architecture [2, 3]. The global routing architecture of an FPGA specifies the relative width of the various wiring channels within the chip. Figure 1.1 depicts an example global routing architecture in which the channels near the center of the FPGA are wider than those near the edges. In MPGA and Standard Cell implementations, a custom chip is created for each design, so routing channels can easily be made wider in areas of a chip where the demand for routing is greater. In FPGAs, however, all routing resources are prefabricated, so the width of all the routing channels is set by the FPGA manufacturer. Our goal, then, is to find the distribution of routing resources, or tracks, to the various channels that permits their efficient utilization by the largest class of circuits. If there are too few

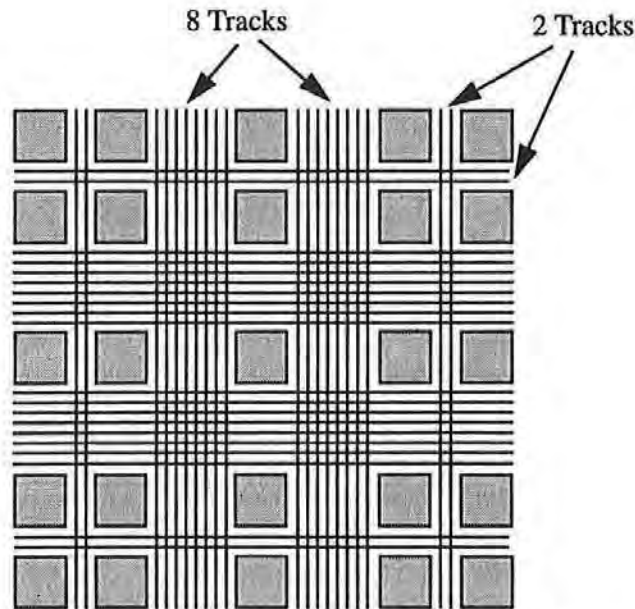


FIGURE 1.1 Example global routing architecture.

tracks in some area of the chip then many circuits will be unroutable, while if there are too many tracks, they may be wasted. There is no agreement amongst commercial FPGAs on the best global routing architecture [4, 5, 6, 7], so this question has clear relevance.

The second issue we explore is the use of *cluster-based* logic blocks in FPGAs [8, 9, 10]. These logic blocks are groups, or clusters, of look-up tables (LUTs) and flip flops along with local routing to interconnect the LUTs within a cluster; Figure 1.2 depicts an example logic cluster. In an FPGA using cluster-based logic blocks, many connections can be made using the local interconnect within a cluster. Since this local interconnect can be made faster than the general-purpose interconnect between logic

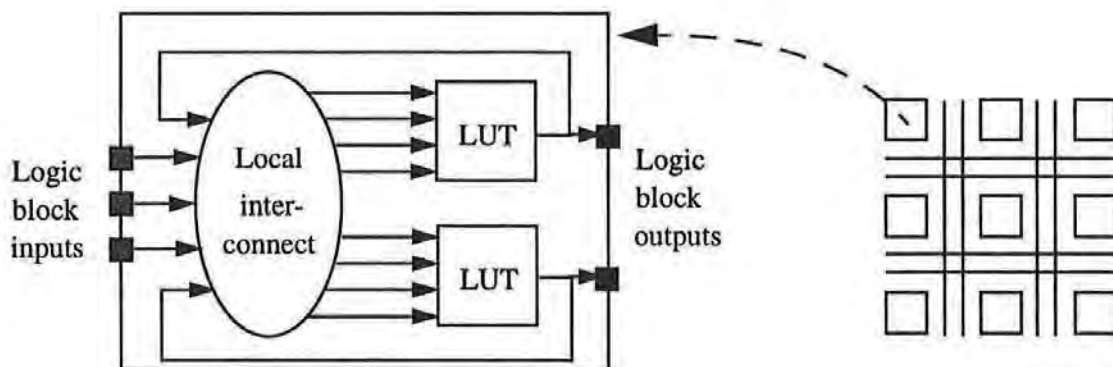


FIGURE 1.2 Example logic cluster containing two LUTs.

blocks, cluster-based logic blocks can improve FPGA speeds. As well, an FPGA in which every logic block contains several LUTs will need fewer logic blocks to implement a circuit than an FPGA in which each logic block is a single LUT. This reduces the size of the placement and routing problem considerably. Since placement and routing is usually the most time-consuming step in mapping a design to an FPGA, cluster-based logic blocks can significantly reduce design compile time. As FPGAs grow larger, it is important to keep this compile time from growing too large or one of the key advantages of FPGAs, rapid prototyping and quick design turns, will be lost.

We will investigate how the number of LUTs per cluster affects FPGA speed, area, compile time and other important FPGA architecture parameters, such as the proper number of inputs to a logic block and the required flexibility of the general-purpose interconnect. The trade-offs among these architectural issues and metrics are quite complex. For example, grouping related LUTs together into a single logic block reduces the number of connections to be routed between logic blocks, saving routing area. Since the general-purpose interconnect consumes most of the die area in SRAM-based FPGAs [1], this is a significant area savings. On the other hand, in the logic clusters we study, the area required by the local routing grows quadratically with the number of LUTs in a cluster. For sufficiently large clusters, the area used by this local interconnect will exceed the area saved in the general interconnect. Similarly, since the local connections within clusters can be made very fast, we expect clusters containing a larger number of LUTs to attain higher circuit speeds. At some point, however, the speed gained by increasing the cluster size will diminish or be lost because increasing the cluster size slows the local cluster routing.

While recent FPGAs from Xilinx [11], Altera [7], Lucent Technologies [5], Actel [12] and Vantis [13] have all grouped several LUTs together into larger logic blocks, there has been little published work investigating the impact of the number of LUTs per logic block on FPGA speed or area.

The final FPGA issue we examine is that of *detailed routing* architecture [14]. The detailed routing architecture of an FPGA defines how logic block inputs and outputs can be interconnected. The style of detailed routing architecture we will investigate is the island-style architecture [1] used by Xilinx, Lucent, and Vantis FPGAs [4, 5, 13]. A simplified detailed-routing architecture is depicted in Figure 1.3. In many ways, detailed routing architecture is *the* key element of an FPGA because:

1. Most of an FPGA's area is devoted to routing;
2. Most of a circuit's delay is due to routing delays rather than logic block delays [15, 1]; and

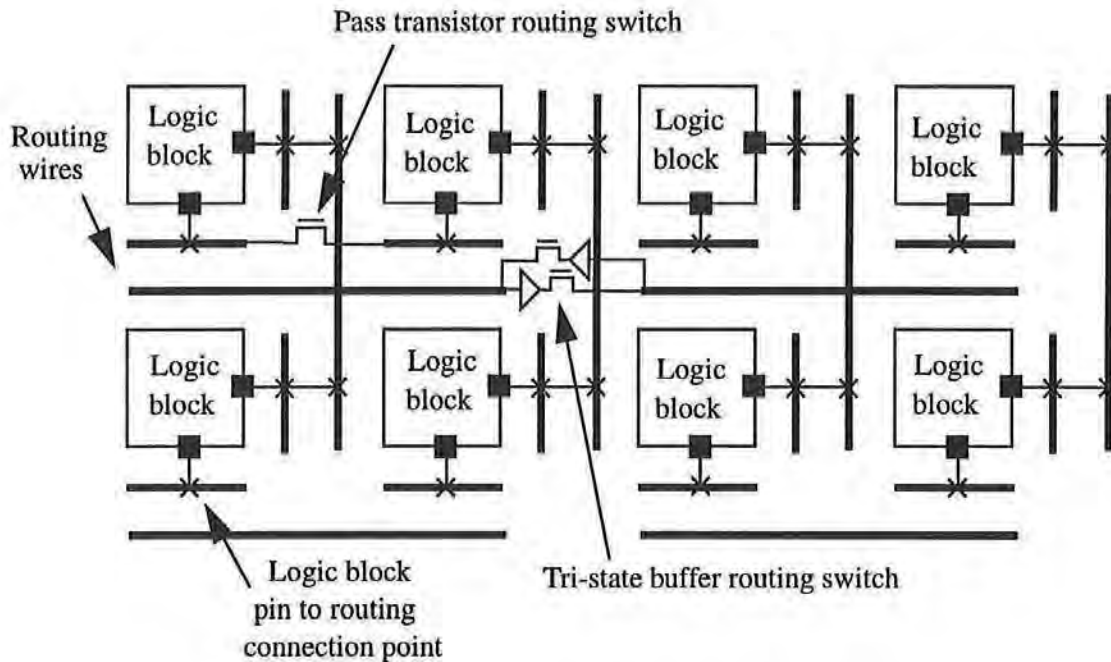


FIGURE 1.3 Example detailed routing architecture.

3. Interconnect delay does not scale as well as logic delay with process shrinks, so the fraction of delay due to routing in FPGAs is increasing with each process generation [16].

The detailed routing architecture defines such features as:

- Which routing wires in the channel adjacent to a logic block input or output can connect to that logic block pin,
- How many logic blocks each routing wire spans before terminating,
- Where routing switches are located and which routing wires they can connect together,
- Whether each routing switch is a pass transistor or a tri-state buffer,
- The sizes of the transistors used to build the various programmable switches, and
- The metal width and spacing of the various routing wires.

The design of a good detailed routing architecture is a very challenging problem, as the best value for each of the parameters above depends on complex trade-offs. For example, if we have too many programmable switches we will clearly waste area, and the increased delay caused by connections passing through many switches will degrade circuit speed. If, on the other hand, we have too few programmable switches, or if they are arranged in a poor topology, circuits will be either wasteful of wires or

completely unroutable. Achieving an appropriate balance between pass transistor based and tri-state buffer based switches is also crucial — pass transistors require less area and are faster for connections that go through only a few switches, but buffers are faster for connections that go through many switches. Similarly, if we have too few long routing wires, long connections will have to pass through many programmable switches and will be slow. On the other hand, too many long wires will result in short connections being made via long wires, wasting area and degrading speed.

It is difficult to look at one detailed routing architecture parameter in isolation because the architectural parameters interact in complex ways to determine an FPGA's speed and area-efficiency. In the research presented in this book, we have tried to take a holistic approach that looks at all the parameters determining a detailed routing together, and assesses both the speed and area of each FPGA we investigate. We believe that this is the first study to examine such a broad spectrum of detailed routing architectures.

1.3 Approach and CAD Tools

Our approach to exploring FPGA architecture is an experimental one. Benchmark circuits are technology mapped, placed and routed into each FPGA architecture of interest and the area and/or delay of each circuit in that architecture is measured. These measurements allow us to determine the best architectural parameters.

In order to evaluate different architectures, the CAD tools used in the empirical approach must be flexible enough to target all the architectures of interest. This precludes the use of commercial FPGA CAD tools, which target only the FPGAs of a specific manufacturer. Accordingly, we have created new CAD tools to pack multiple LUTs into a logic cluster, to place logic blocks within an FPGA, and to route the connections between logic blocks [17, 10]. In addition to making these CAD tools flexible, we also took considerable care to ensure they produced high quality results, as poor CAD tools can lead to inaccurate architecture conclusions. The CAD tools understand the special features of each architecture, and attempt to fully optimize for each architecture they target. They incorporate several new features and enhancements to existing algorithms, so the CAD algorithms are of interest in their own right. As well, these CAD tools are in use in numerous FPGA and CAD studies at the University of Toronto and around the world, and obtain the highest quality results on the set of standard benchmarks used to compare academic place and route tools [17].

Ideally, one would lay out each FPGA architecture of interest to obtain exact area measurements and highly accurate delay values. In this book we study hundreds of

different FPGA architectures, so this is clearly impossible. Accordingly, we have developed more abstract area models and delay estimates that do not require layout of the FPGAs, but which are still sufficiently accurate to allow fair comparison of different architectures.

1.4 Book Organization

Most of the material in this book is adapted from the Ph.D. thesis of Vaughn Betz [18], with important additional information (the T-VPack algorithm described in Chapter 3 and most of the experimental results and analysis concerning logic clusters presented in Chapter 6) adapted from the M.A.Sc thesis of Alexander Marquardt [19].

The next chapter provides background information and details some of the previous work in the relevant areas of both CAD and FPGA architecture. Chapter 3 describes the CAD algorithms we use to pack several LUTs together into a logic cluster, and the algorithms we use to perform placement of logic blocks in an FPGA. In Chapter 4 we detail the algorithms used in our FPGA router.

Chapters 5 through 7 then employ these CAD tools to investigate the three FPGA architectural issues outlined earlier. Each of these three chapters begins by describing the CAD flow and area and/or delay models used in the experimental study, and then presents the architectural results obtained. Chapter 5 evaluates different global routing architectures for FPGAs, and determines the most area-efficient architectures. In Chapter 6 we investigate cluster-based logic blocks and determine the logic cluster types that maximize area-efficiency and speed. In Chapter 7 we ascertain which detailed routing architectures lead to FPGAs with the best combination of speed and area-efficiency. The final chapter summarizes the architectural conclusions and provides suggestions for future work.

The three appendices provide additional useful information. Appendix A outlines the graphic visualization features incorporated in our placement and routing CAD tools, and highlights their usefulness in algorithm and architecture research. Appendix B describes the transistor-level design of an FPGA and our characterization of a deep-submicron IC process. Finally, in Appendix C we investigate how routing transistors and metal routing wires should be sized in order to create an FPGA with the best combination of area-efficiency and speed. The detailed information concerning the electrical design of an FPGA provided by Appendices B and C is essential to create accurate FPGA area and delay models, and hence to properly compare FPGA architectures.

1.5 Acknowledgments

The authors are indebted to Carl Harris of Kluwer Academic Publishers for his help and advice as we prepared this book. We would like to thank the other members of our research group at the University of Toronto — Steve Wilton, Mike Hutton, Mohammed Khalid, Yaska Sankar, Jordan Swartz, Rob McCready, and Paul Leventis — for the many valuable suggestions and insightful discussions they have provided over the years. Special thanks are due to Jordan Swartz for enhancing the VPR FPGA architecture generator to allow targeting of an architecture very similar to that of the Xilinx 4000X series FPGAs. Many other students and faculty in the University of Toronto Electrical Engineering department lent us their expertise on vexing problems; we would particularly like to thank Dave Lewis, Mark Stoodley, Guy Lemieux, Jason Anderson, and Jason Podaima. We also appreciate several interesting discussions on FPGA CAD with Russ Tessier of MIT.

The authors are grateful for many enlightening discussions with people in the FPGA industry. Particular thanks are due to Steve Trimberger, Steve Young, and Bill Carter of Xilinx, Frank Heile and David Mendel of Altera, Sinan Kaptanoglu of Actel, and Tim Lacey of Cypress. We appreciate the helpful suggestions of Jason Cong, Steve Trimberger, Sinan Kaptanoglu, Corinna Lee, Stephen Brown, Dave Lewis, and Derek Corneil concerning the Ph.D. thesis on which the majority of this book is based.

We gratefully acknowledge the financial support of our work provided by the Natural Sciences and Engineering Research Council of Canada (via an NSERC 1967 Science and Engineering Scholarship), the Information Technology Research Centre of Ontario, the Walter C. Sumner Foundation, a V. L. Henderson Research Fellowship, and Xilinx. Finally, we appreciate the 0.35 μm process data provided to us by the Canadian Microelectronics Corporation and the Taiwan Semiconductor Manufacturing Company (TSMC).



Background and Previous Work

The first half of this chapter provides background information about FPGA architectures, and briefly describes the prior work relevant to this book. The second half of the chapter describes the CAD flow used to automatically map circuits into FPGAs and determine their speed, and summarizes some of the prior work in the relevant areas of CAD.

2.1 FPGA Architecture

All FPGAs are composed of three fundamental components: logic blocks, I/O blocks and programmable routing, as Figure 2.1 shows [20]. A circuit is implemented in an FPGA by programming each of the logic blocks to implement a small portion of the logic required by the circuit, and each of the I/O blocks to act as either an input pad or an output pad, as required by the circuit. The programmable routing is configured to make all the necessary connections between logic blocks and from logic blocks to I/O blocks. The following section briefly describes the basic technologies used to make FPGAs programmable. Sections 2.1.2 and 2.1.3 then describe some of the different architectures and prior research for FPGA logic blocks and routing, respectively.

2.1.1 FPGA Programming Technologies

There are three different approaches to making FPGAs programmable [1, 20]. The most popular technology today uses SRAM cells to control pass transistors, multi-

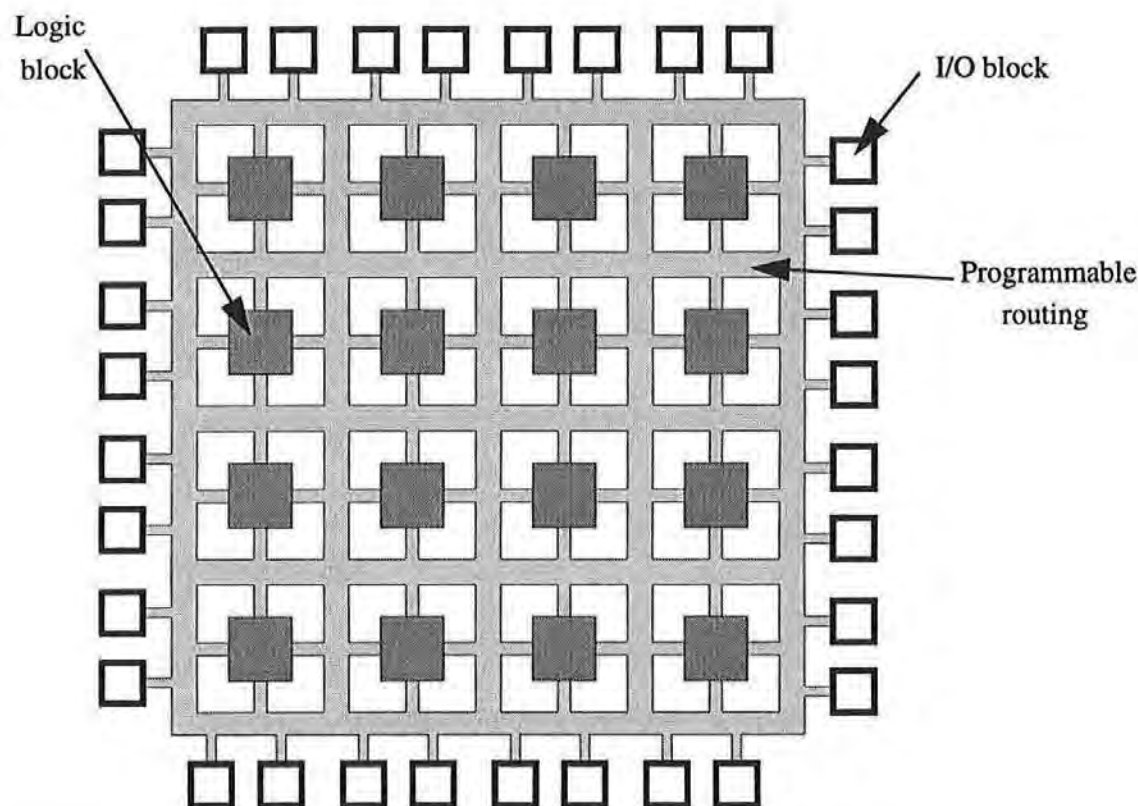


FIGURE 2.1 A generic FPGA (from [20]).

plexers and tri-state buffers in order to configure the programmable routing and logic blocks as required. Figure 2.2 shows these SRAM-based switches. Notice that pass gates are implemented with nMOS pass transistors, rather than complementary transmission gates, as this results in better speed due to the higher carrier mobility in nMOS transistors [21, 22]. Most Xilinx FPGAs [4], the larger Altera devices [7], the latest Actel FPGAs [12], all Lucent Technologies FPGAs [5] and the Vantis VF1 FPGAs [13] are SRAM-based. Alternative programming technologies are antifuses [23] which are used in many Actel FPGAs [6], and floating gate devices (e.g.

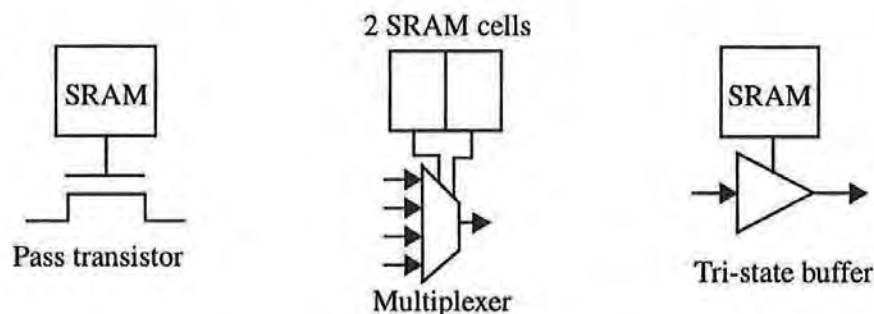


FIGURE 2.2 Three types of programmable switch used in SRAM-based FPGAs.

EPROM, EEPROM and Flash) which are used in many Complex Programmable Logic Devices (CPLDs) [24].

2.1.2 FPGA Logic Block Architecture

The logic block used in an FPGA strongly influences the FPGA speed and area-efficiency. While many different logic blocks have been used in FPGAs [1], most current commercial FPGAs use logic blocks based on look-up tables (LUTs). Accordingly, in this book we will investigate only LUT-based logic blocks. Figure 2.3 shows how a 2-input LUT can be implemented in an SRAM-based FPGA — a k -input LUT requires 2^k SRAM cells and a 2^k -input multiplexer. A k -input LUT can implement any function of k -inputs; one simply programs the 2^k SRAM cells to be the truth table of the desired function.

Previous research has shown that LUTs with 4-inputs lead to FPGAs with the highest area-efficiency [25], and most commercial FPGAs are based on 4-LUTs.

Most modern FPGAs are composed not of a single LUT, but of groups of LUTs and registers with some local interconnect between them, and there has been some research into logic blocks containing groups of LUTs. Jianshe He and Rose studied FPGAs that contained look-up tables of two different sizes [26]. Over a set of benchmark circuits, they found that an FPGA containing two-thirds 4-input LUTs and one-third 2-input LUTs reduced the number of SRAM bits within the LUTs by 22% and the number of logic block pins by 10% vs. an FPGA that contained only 4-LUTs. They never actually placed and routed an FPGA with a logic block composed of two 4-LUTs and one 2-LUT, but these results show that such a logic block should have good area-efficiency. In [27], Chung and Rose investigated logic blocks composed of several LUTs joined by inflexible “hardwired” connections. They found that some of these “hard-wired logic blocks” outperformed single 4-LUT logic blocks. In particular, a logic block composed of four 4-LUTs connected in a specific tree topology led

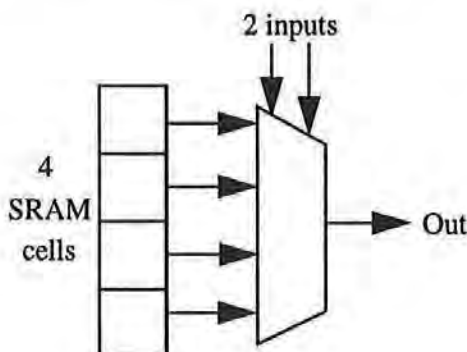


FIGURE 2.3 A 2-input LUT implemented in an SRAM-based FPGA.

to an FPGA 7% more dense and 28% faster than an FPGA using a single 4-LUT as its logic block. The area model used in this study modelled routing area by estimating the number of routing switches needed after global routing. Since this model did not account for the fact that the routing transistors would most likely have to be sized up for the larger logic blocks, these area numbers are probably somewhat inaccurate (overly optimistic) for logic blocks with more than one LUT. Cong and Hwang [28] examined how well four different logic blocks composed of groups of LUTs connected by hardwired connections, and in one case a multiplexer, were able to implement “wide” functions of many variables.

In [29], Aggarwal and Lewis investigated hierarchical FPGA architectures. Since the logic clusters we will investigate in Chapter 6 can be thought of as a logic block with a two-level hierarchy, this is somewhat related to our work.

2.1.3 FPGA Routing Architecture

Commercial FPGAs can be classified into three groups, based on their routing architecture. The FPGAs of Xilinx, Lucent and Vantis are *island-style* FPGAs, while Actel’s FPGAs are *row-based*, and Altera’s FPGAs are *hierarchical* [1]. In this book, we will almost exclusively investigate the island-style routing architecture,¹ so we describe this style of routing architecture below.

Figure 2.4 depicts an island-style FPGA. Logic blocks are surrounded by routing channels of pre-fabricated wiring segments on all four sides. A logic block input or output, which we call a *pin*, can connect to some or all of the wiring segments in the channel adjacent to it via a *connection block* [30] of programmable switches. At every intersection of a horizontal channel and a vertical channel, there is a *switch block* [30]. This is simply a set of programmable switches that allow some of the wire segments incident to the switch block to be connected to others; note that for clarity only a few of the programmable switches contained by switch blocks are shown in Figure 2.4. By turning on the appropriate switches, short wire segments can be connected together to form longer connections. In the FPGA of Figure 2.4, notice that some wire segments continue unbroken through a switch block. These longer wires span multiple logic blocks, and are a crucial feature in commercial FPGAs.

In [30], Rose and Brown developed a useful notation for describing some of the parameters of an FPGA’s routing architecture, and we will use their notation in this book. The number of tracks, or wires, contained in a channel is denoted by W . The number of wires in each channel to which a logic block pin can connect is called the

1. Some of the results in Chapter 5 are also applicable to row-based FPGAs.

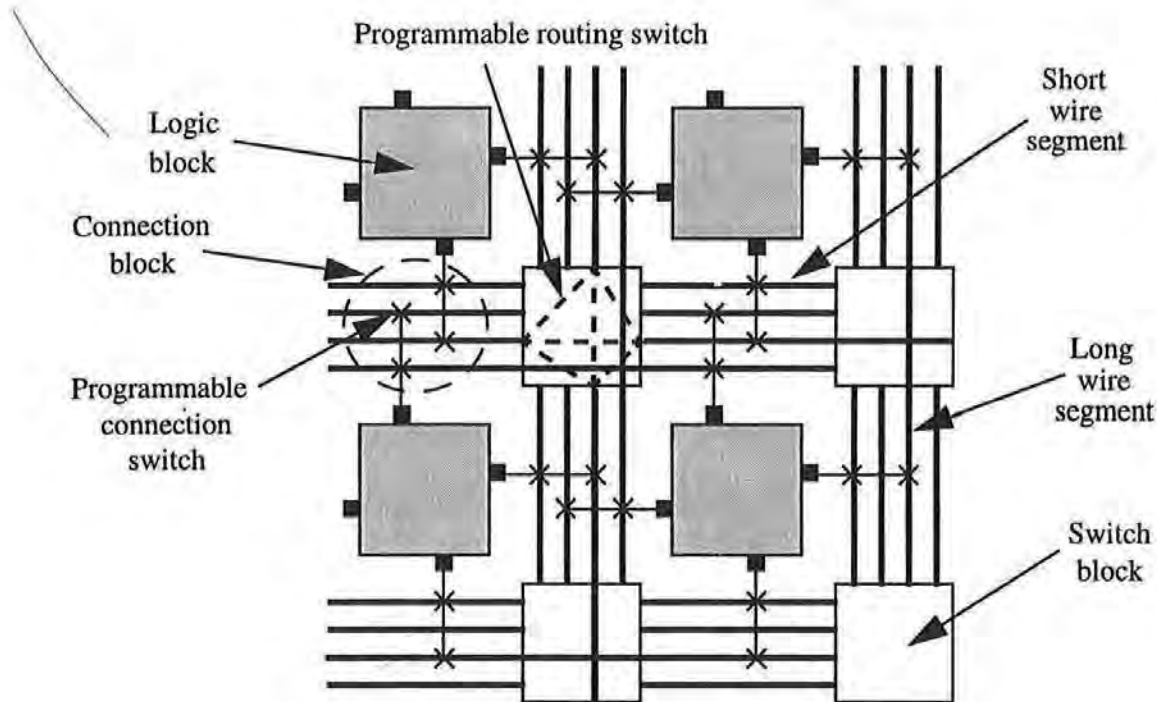


FIGURE 2.4 An island-style FPGA.

connection block flexibility, or F_c . The number of wires to which each incoming wire can connect in a switch block is called the switch block flexibility, or F_s . In the FPGA of Figure 2.4, for example, W is 4 for all channels, F_c is 2 and F_s is 3.

Considering the importance of an FPGA's routing architecture to both its area-efficiency and speed, relatively little prior research has been conducted. The question of how many wires each channel in an FPGA should contain relative to the other channels, which we call the *global routing* architecture of an FPGA has not been studied before. There is some prior work concerning the *detailed routing* architecture of FPGAs, however.

Most prior work has investigated FPGAs in which all wires span only one logic block before terminating at a switch block, and have compared architectures only on the basis of area-efficiency. The area-efficiency metric in these studies has usually been the number of programmable switches contained in the routing. In [30], Rose and Brown investigated FPGAs of this type and found that the most area-efficient architectures had $F_s = 3$ or 4 and $F_c = 0.7W$ to $0.9W$. Tseng et al showed that FPGAs with $F_s = 2$ can get reasonable area-efficiency despite the very low flexibility of their switch block if the CAD tools properly optimize for the architecture [31]. Even then, however, they found that $F_s = 3$ was still 25% more area-efficient than $F_s = 2$. Other

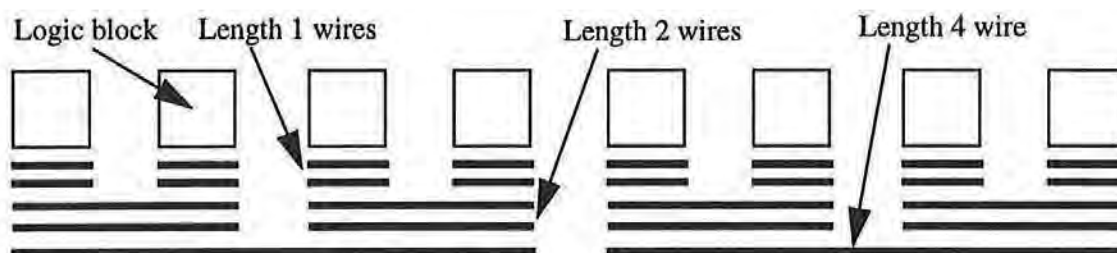


FIGURE 2.5 Example channel segmentation distribution.

researchers have focused on finding better $F_s = 3$ switch block topologies in order to enhance routability and hence area-efficiency [32, 33].

A few studies have looked at routing architectures that include different lengths of wires. Figure 2.5 shows one channel in such an FPGA. The *length* of a wiring segment is the number of logic blocks it spans; Figure 2.5 shows segments of length 1, 2 and 4. The *segmentation distribution* defines what fraction of the tracks in each channel are of each length. In Figure 2.5 for example, 40% of the tracks are of length 1, 40% are of length 2, and 20% are of length 4. When a wire spans the entire width or height of an FPGA, we will use Xilinx's terminology and call it a *long line*.

Greene et al briefly describe a procedure to choose an appropriate distribution of segment lengths for use in an antifuse-programmed row-based FPGA [34]. They looked at the distribution of connection lengths extracted from a set of global routings of circuits, and hand-crafted a segmentation distribution to match these lengths. Roy and Mehendale followed a similar procedure, again to find segmentation distributions for antifuse-programmed row-based FPGAs. They created a tool to automatically generate a segmentation length distribution that closely matched a specified length distribution [35]. They found that a Poisson distribution of segment lengths led to fewer series switches being used to make connections, on average, than a uniform distribution that made all segments span four logic blocks. Consequently, the Poisson distribution should lead to higher speed, but it required more routing area. Notice that both these studies assume that a distribution of connection lengths can be guessed or determined from a global routing. While this is possible in a row-based FPGA, island-style FPGAs usually perform global and detailed routing in one unified step, so these methods of generating segmentation distributions will not work well with island-style FPGAs.

Some prior work has also investigated segmentation for island-style FPGAs. In these studies, all the programmable switches within a switch block were assumed to be pass transistors; the possibility of tri-state buffer routing switches has not been investigated.

Brown et al investigated different segment length distributions in order to optimize the speed and area of an island-style FPGA [36, 37, 38]. They placed circuits, then routed them with a two-step (global then detailed) routing procedure, and evaluated the resulting speed and area for each FPGA architecture of interest. They found that segments of length greater than 3 or 4 were unnecessary, and that segments of length 1 were also not very important. The best segmentation distributions, in terms of speed/area trade-off, had 40 - 70% length 3 wires, 10 - 50% length 2 wires, and 10 - 20% length 1 wires. Their conclusions may have been affected by several factors, however. First, in the two-step routing scheme they use, the global router is unaware of the segmentation distribution, and hence is not attempting to optimize for it. This may make it difficult to use long wires effectively. Second, their area metric was number of tracks per channel required to route, W , while FPGA area is usually determined by transistor area. Third, their speed metric was average net delay, rather than the circuit critical path. Finally, the tools did not use timing analysis to determine which connections needed high-speed routing, and which could use area-efficient routes.

Chow et al [39] performed a similar study of segmentation distributions in an island-style FPGA, using similar tools to Brown et al [36]. They did not evaluate circuit speed, but assumed that longer wires would lead to faster circuits. Hence, they looked for segmentation distributions that incorporated length 2 and 3 wires but required only slightly higher W values to route circuits than a segmentation distribution where all wires were length 1. They also investigated a new architectural parameter, *internal population*, that is relevant to wires which span more than one logic block. All wires are assumed to be able to connect to other wires and to logic blocks at their endpoints. A wire segment is *internally switch-block populated* if it can make connections from its interior to other wire segments (via switch blocks); otherwise it is switch-block depopulated. Similarly, a wire that can connect to from its middle to logic blocks (via connection blocks) is *internally connection-block populated*. Figure 2.6 illustrates the four different internal population possibilities for segments. Notice that a length 3 wire that is connection-block depopulated can connect to only 2 of the 3 logic blocks it passes. A length 3 wire that is switch-block depopulated can connect to wires in only two of the four vertical channels that pass by it. Internally depopulated wire segments are faster than populated wire segments because there are fewer switches, and hence less parasitic capacitance, connected to them. Chow et al found that the use of switch-block depopulated wires resulted in a significant increase in the required value of W to route circuits, while connection-block depopulation caused only a small (~0 - 10%) increase in W . They also found that many architectures with 0 - 60% length 2, 10 - 40% length 3, and the remainder length 1 wires required only about 10% more tracks per channel than an all length 1 architecture.

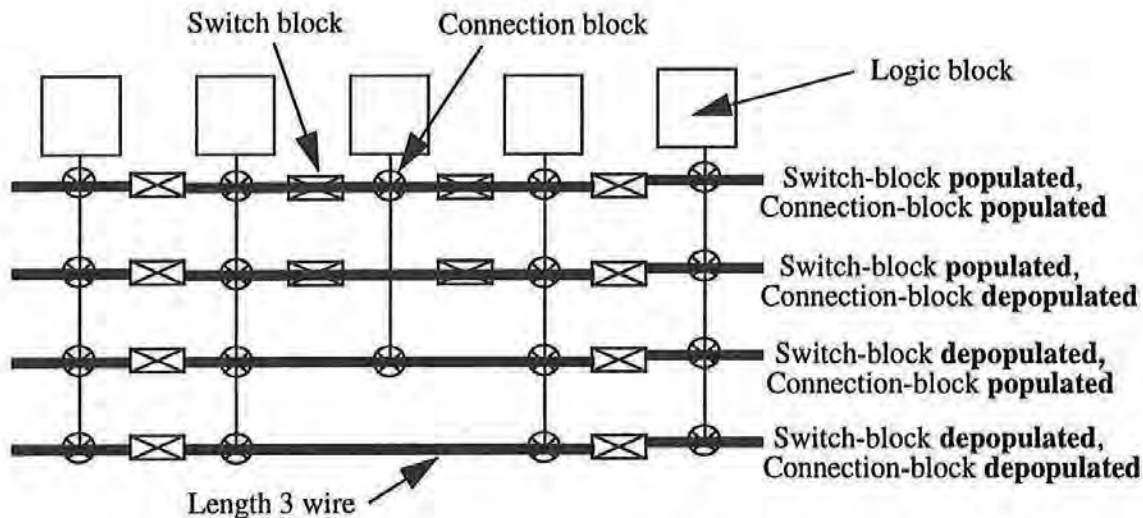


FIGURE 2.6 Internal population and depopulation of wiring segments.

In [40], Sun et al proposed a new type of FPGA routing structure that incorporated longer wires. This new structure is related to but not identical to the routing of an island-style FPGA. Its area-efficiency, as measured by the number of programmable switches in the routing, is 50% worse than that of an island-style FPGA that uses all length 1 wires. The number of active routing switches (switches that are actually turned on to connect wires) is 35% lower in this new structure, however. The authors of [40] therefore conclude that this new FPGA is faster than an island-style FPGA with all length 1 wires. This number of active routing switches delay metric is not very closely related to critical path delay, however, so there is a large margin of error in their delay results.

2.2 CAD for FPGAs

Implementing a circuit in a modern FPGA requires that hundreds of thousands or even millions of programmable switches and configuration bits be set to the proper state, on or off. Clearly if a circuit designer has to specify the state of each programmable switch in an FPGA very few designs will ever be completed! Instead, users of FPGAs describe a circuit at a higher level of abstraction, typically using a hardware description language (such as VHDL) or schematic entry. Computer-Aided Design (CAD) programs then convert this high-level description into a programming file specifying the state of every programmable switch in an FPGA. To keep the complexity of this procedure tractable, the problem of determining how to map a circuit into an FPGA is normally broken into a series of sequential subproblems, as shown in

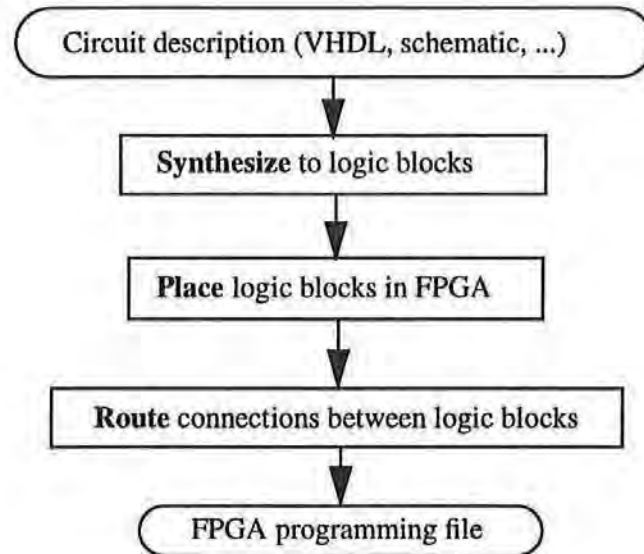


FIGURE 2.7 FPGA CAD flow.

Figure 2.7. In the following three sections we will describe the synthesis, placement and routing problems and briefly outline some of the prior work in each area. Once a circuit is placed and routed, delay models and timing analysis are used to assess its speed; we describe these problems in Sections 2.2.4 and 2.2.5.

2.2.1 Synthesis and Logic Block Packing

The first stage of synthesis converts the circuit description, which is usually in a hardware description language or schematic form, into a netlist of basic gates. Then, the logic synthesis process converts this netlist of basic gates into a netlist of FPGA logic blocks such that the number of logic blocks needed is minimized and/or circuit speed is maximized. Logic synthesis is sufficiently complex that it is usually broken into two or more subproblems [41]; in this book we will use a three-stage synthesis flow as shown in Figure 2.8.

Technology-independent logic optimization removes redundant logic and simplifies logic wherever possible [41, 42]. The optimized netlist of basic gates is then mapped to look-up tables [43, 44, 45, 46, 47]. Both of these problems have been extensively studied and good algorithms and tools capable of targeting the FPGAs we are interested in studying are publicly available, so this book does not study these phases of the synthesis process.

The third synthesis step in Figure 2.8 is necessary whenever an FPGA logic block contains more than a single LUT. Logic block packing groups several LUTs and reg-

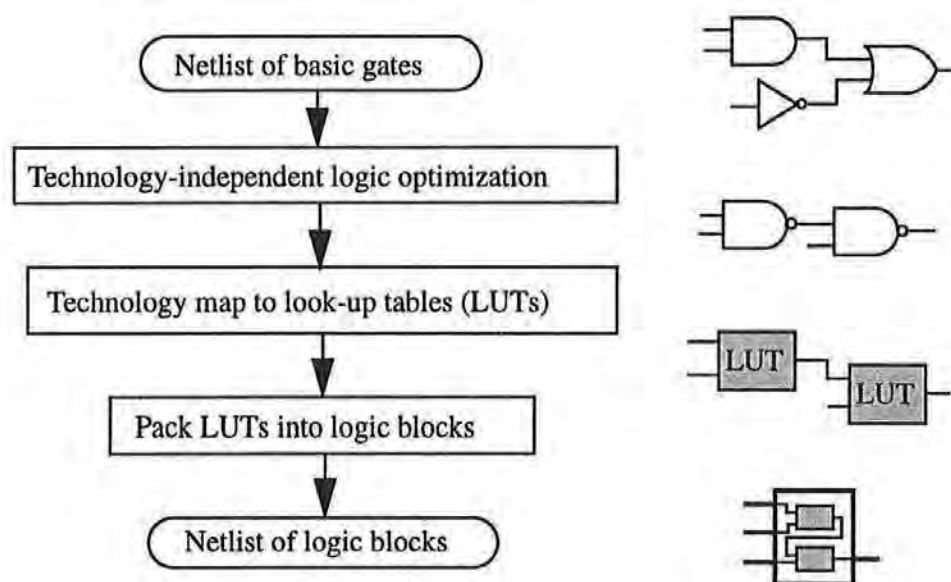


FIGURE 2.8 Details of synthesis procedure.

isters into one logic block, respecting limitations such as the number of LUTs a logic block may contain, and the number of distinct input signals and clocks a logic block may contain. The optimization goals in this phase are to pack connected LUTs together to minimize the number of signals to be routed between logic blocks, and to attempt to fill each logic block to its capacity to minimize the number of logic blocks used.

This problem is a form of clustering. Clustering and partitioning are essentially the same problem; divide a netlist into several pieces, such that certain constraints, such as maximum partition size, are respected, and some goal, such as minimizing the number of connections that cross partitions, is optimized [48]. When a circuit is to be divided into only a few pieces, the problem is called partitioning. When a circuit is to be divided into many small pieces in one step (as opposed to recursively partitioning into a few partitions in each step), the problem is usually called clustering.

Clustering has been extensively studied [48, 49, 50, 51, 52], with spectral [53, 54, 55, 56] and labelling [57, 58, 59, 60] methods being popular. Many of these methods, however, cannot easily incorporate simultaneous constraints on the maximum number of inputs, the number of clocks, and the number of LUTs and registers in a logic block, and these constraints are key in logic block packing. Spectral methods use analytic techniques to produce “natural” groupings of circuit elements, and then enforce constraints on cluster size, etc. in a post-processing step. With many cluster legality constraints, then, it becomes likely that this post-processing step will perturb

the “natural” clustering solution enough to significantly impact the result quality. It is difficult for labelling methods to simultaneously satisfy both a constraint on the number of inputs to a cluster (pin constraint) and the maximum number of items (e.g. LUTs) in a cluster (size constraint). Yang and Wong [60] present a labelling algorithm which can simultaneously satisfy both size and pin constraints, but it tends to produce large amounts of logic replication and to only partially fill clusters. Hence it requires a large number of clusters to cover a circuit.

In [61], Cong et al describe a logic block packing tool capable of targeting several different types of logic blocks. It uses a closeness metric to determine the desirability of putting two LUTs into the same logic block, and it respects any constraints restricting which LUTs can be packed together. In [61], the packing step is performed via maximum weighted matching on a graph representing the circuit. This is an $O(nm)$ algorithm, where n is the number of LUTs, and m is the number of edges in the graph [62, 63]. For the logic clusters we study, the number of edges in this “closeness” graph, m , is $O(n^2)$, which would lead to an $O(n^3)$ algorithm.¹

In [64], Shin and Kim describe a greedy algorithm to transform a netlist of circuit blocks into a netlist of clusters, where each cluster contains approximately the same number of circuit blocks, and the circuit blocks in each cluster tend to be highly connected. Initially each circuit block is a cluster. The “closeness” of two clusters is a function of how many nets the two clusters share and the size of the cluster that would result from merging them. More formally:

$$closeness(C, D) = \frac{NumCommonNet(C, D)}{\min(NumPin(C), NumPin(D))} - \alpha \cdot \frac{ClusterSize(C, D)}{AverageClusterSize} \quad (2.1)$$

where α is a weighting factor controlling how balanced the cluster sizes should be. The two clusters with the largest closeness value are merged, and the closeness values of all the other clusters are updated to reflect the change. The merging process terminates when the number of clusters in the netlist falls below some user-specified value. While this algorithm was not developed for FPGAs, and does not understand the hard constraints on cluster size and number of inputs intrinsic to logic clusters, the general approach of greedily optimizing a closeness function is highly adaptable, and hence of interest.

1. If the edge weights are all integers within a limited magnitude range, algorithms with complexity somewhat worse than $O(n^5m)$, but less than $O(nm)$ exist [63]. With m of $O(n^2)$ this would still lead to an algorithm with complexity worse than $O(n^{2.5})$.

2.2.2 Placement

Placement algorithms determine which logic block within an FPGA should implement each of the logic blocks required by the circuit. The optimization goals are to place connected logic blocks close together to minimize the required wiring (wirelength-driven placement), and sometimes to place blocks to balance the wiring density across the FPGA (routability-driven placement) or to maximize circuit speed (timing-driven placement).

The three major classes of placers in use today are min-cut (partitioning-based) [65, 66, 67], analytic [68, 69, 70, 71, 72, 73, 74] which are often followed by local iterative improvement [75], and simulated annealing based placers [76, 77, 78, 79, 80, 81]. Recall that to fairly investigate architectures we must ensure that our CAD tools are attempting to use all of an FPGA's features — this means the optimization goals of our placer may change from architecture to architecture. It is much easier to add new optimization goals or constraints to a simulated annealing based placer than to a min-cut or an analytic placer, so we focus on this algorithm below.

Simulated annealing mimics the annealing process used to gradually cool molten metal to produce high-quality metal objects [76]. Pseudo-code for a generic simulated annealing-based placer is shown in Figure 2.9. A cost function is used to evaluate the quality of any placement of logic blocks — for example, a common cost function in wirelength-driven placement is the sum over all nets of the half-perimeter

```

S = RandomPlacement ();
T = InitialTemperature ();
Rlimit = InitialRlimit ();

while (ExitCriterion () == False) {      /* "Outer loop" */
    while (InnerLoopCriterion () == False) { /* "Inner loop" */
        Snew = GenerateViaMove (S, Rlimit);
        ΔC = Cost (Snew) - Cost (S);
        r = random (0,1);
        if (r < e-ΔC/T) {
            S = Snew;
        }
    } /* End "inner loop" */
    T = UpdateTemp ();
    Rlimit = UpdateRlimit ();
} /* End "outer loop" */

```

FIGURE 2.9 Pseudo-code of a generic simulated annealing-based placer.

of their bounding boxes. An initial placement is created by assigning logic blocks randomly to the available locations in the FPGA. A large number of moves, or local improvements, are then made to gradually improve the placement. A logic block is selected at random, and a new location for it is also selected at random. The change in cost function that would result from moving the selected logic block to the proposed new location is computed. If the cost would decrease, the move is always accepted and the block is moved. If the cost would increase, there is still a chance of the move being accepted, even though it makes the placement worse. This probability of acceptance is given by $e^{-\Delta C/T}$, where ΔC is the (positive) change in cost function the move causes, and T is a parameter called temperature that controls the likelihood of accepting moves that make the placement worse. Initially, T is very high so almost all moves are accepted; it is gradually decreased as the placement is refined so that eventually the probability of accepting a move that makes the placement worse is very low. This ability to accept hill-climbing moves that make a placement worse allows simulated annealing to escape local minima in the cost function.

The rate at which temperature is decreased, the exit criterion for terminating the anneal, the number of moves attempted at each temperature (InnerLoopCriterion), and the method by which potential moves are generated are defined by the annealing schedule. A good annealing schedule is crucial to obtain good results in a reasonable amount of CPU time. Many proposed annealing schedules are “fixed” schedules that have no ability to adapt to different problems. Such schedules can work well within the narrow application range for which they were developed, but their lack of adaptability means they are not very general [82]. Accordingly, we discuss below only “adaptive” annealing schedules that determine an annealing schedule based on statistics computed during the anneal itself.

Huang, Romeo, and Sangiovanni-Vincentelli [82], developed an annealing schedule and applied it to a standard-cell placement problem. Huang’s schedule performs a set of random moves on the initial placement, and sets the initial temperature (InitialT) to 20σ , where σ is the standard deviation of the cost over these moves. New temperatures (UpdateT) are computed via:

$$T_{new} = T_{old} \cdot e^{\frac{-\lambda T_{old}}{\sigma}} \quad (2.2)$$

where λ is typically set to 0.7 and σ is the standard deviation of the moves accepted at T_{old} . The InnerLoopCriterion of Figure 2.9 is fairly complex for this schedule; it involves monitoring the fraction of new states generated that have their costs within a certain range of the average cost at that temperature, and there are several special cases and fall-back cases defined. Finally, the anneal terminates (ExitCriterion), when the difference between the maximum and minimum costs accepted at that tem-

perature equals the maximum cost change caused by any one move at that temperature.

The schedule of Lam and Delosme [83], employs feedback control to set the annealing schedule. It monitors the standard deviation of the cost, the average cost, and the fraction of proposed moves that were accepted, α , over the past τ moves. Typically τ is 100. These values are inputs to a sophisticated feedback system that determines a new temperature. In this schedule, a new temperature is computed every move — that is, the “inner loop” in Figure 2.9 executes only one iteration each time control reaches it. The anneal terminates when there has been no change in the average cost for the last $k \cdot \tau$ moves, where k is typically 5. This annealing schedule also employs a range limiter to control the move generation process. The R_{limit} parameter in Figure 2.9 controls how close together blocks must be to be considered for swapping. Initially, R_{limit} is fairly large, and swaps of blocks far apart on a chip are likely. Throughout the anneal, R_{limit} is adjusted to try to keep the fraction of moves accepted at any temperature close to 0.44. If the fraction of moves accepted, α , is less than 0.44, R_{limit} is reduced, while if α is greater than 0.44, R_{limit} is increased.

One disadvantage of the Lam schedule is its complexity. Fortunately, Swartz and Sechen have developed an annealing schedule incorporating some of the key ideas of the Lam schedule, and which achieves equivalent quality, but is much less complex [84]. In this schedule, the number of moves attempted in the “inner loop” is $10 \cdot N_{\text{blocks}}^{1.33}$. The range limiter, R_{limit} is updated according to a fixed (hard-coded) schedule; the exact form of R_{limit} is not specified in [84], but it likely initially spans the entire chip, and gradually shrinks to a small region. The “outer loop” is executed 150 times, and then the anneal terminates. The temperature is controlled by the fraction of moves accepted:

$$T_{\text{new}} = \left[1 - \frac{\alpha - 0.44}{40} \right] \cdot T_{\text{old}}, \quad (2.3)$$

where 0.44 is desired acceptance rate, and 40 is a damping coefficient to prevent wild oscillations in temperature. While this schedule is much simpler than Lam’s, it has sacrificed some of the adaptability of the Lam schedule, since the range limiter variation and the number of outer loop iterations are now hard-coded.

Since the amount of routing in FPGAs is limited and set by the manufacturer when the FPGA is fabricated, some FPGA placement tools attempt to optimize not just the wirelength of a placement, but also its routability. In [85], Ebeling et al describe a simulated annealing based placer that targets the Triptych FPGA developed at the University of Washington. Its cost function incorporates not only a bounding-box wirelength term, but also a “porosity” term that monitors the fraction of logic blocks

in a local area that are being used. Since the Triptych FPGA is usually unroutable in regions where the logic blocks are completely used, maintaining a porosity of 50% or so across the FPGA is essential. In [86, 87], Nag and Rutenbar describe a simulated annealing based tool that performs placement and routing simultaneously in one combined step. After any swap of blocks, all the affected nets are re-routed via a maze router. To keep the CPU time reasonable, this maze router is constrained to look at only a small number of potential routes when the temperature is high; at lower temperatures, the router is allowed to spend more time looking for routes. If no suitable route is found among the allowed candidates, the net is marked as currently unroutable, and the placement cost is increased. The result quality of this tool is high, but the CPU time required is very large — a circuit containing only 461 Xilinx 4000 logic blocks required 11 hours of CPU time to place and route, and the complexity of this algorithm appears to be approximately $O(n^3)$, where n is the number of logic blocks in a circuit. (Alexander et al have created a partitioning-based algorithm, FPR [88], that performs placement and global routing simultaneously, again in an attempt to maximize circuit routability.)

Some work has also been done in routability-driven standard cell placement that is applicable to FPGAs. Cheng [89] describes a simulated annealing placer that divides a chip into many subregions and estimates the demand for wiring in each region. This demand is compared to the supply of routing in each region, and when the expected demand outstrips the routing supply in some regions the placement is penalized by having its cost increased. Dividing the chip into subregions and estimating wiring demand makes localized congestion visible, and merely estimating the wiring demand in a region is faster than actually routing each placement proposed during the anneal.

2.2.3 Routing

Once locations for all the logic blocks in a circuit have been chosen, a router determines which programmable switches should be turned on to connect all the logic block input and output pins required by the circuit. In FPGA routing, one usually represents the routing architecture of the FPGA as a directed graph [86, 85]. Each wire and each logic block pin becomes a node in this *routing-resource graph* and potential connections become edges. Some prior research has represented FPGAs as undirected graphs [90], but a directed graph representation is needed if directional switches, like tri-state buffers and multiplexers, are to be modelled correctly.

Routing a connection corresponds to finding a path in this routing-resource graph between the nodes representing the logic block pins to be connected. To avoid using up too many of the limited number of wires in an FPGA, one wants this path to be as short as possible. As well, it is important that the routing for one net not use up rout-

ing resources another net needs, so most FPGA routers have some kind of congestion avoidance scheme to resolve contention for routing resources. An additional optimization goal is to make nets on or near the critical path fast by routing them using short paths and fast routing resources. Routers that attempt to optimize timing in this way are called timing-driven, whereas delay-oblivious routers are purely routability-driven. Since most of the delay in FPGAs is due to the programmable routing, timing-driven routing is crucial to obtain good circuit speeds.

FPGA routers can be divided into two groups. Combined global-detailed routers [85, 90, 91, 92, 93, 94, 95, 96] determine a complete routing path in one step, while two-step routing algorithms first perform global routing [97, 98] to determine which logic block pins and channel segments each net will use, and then perform detailed routing [99, 100, 101, 34] to determine the wire(s) each net will use within each of the specified channel segments. A channel segment is the length of routing channel that spans one logic block — a channel that spans M logic blocks contains M channel segments. The task of an FPGA detailed router is often difficult or impossible because FPGA routing has limited flexibility and the detailed router is highly constrained by the decisions the global router made about which channel segments each net must use. Combined global-detailed routers have the potential to more fully optimize the routing, since they are free of such constraints.

Of the routers listed above, only those of [85, 95, 94] use timing analysis (see Section 2.2.5) to determine which nets are on, or “almost” on, the critical path so they can be given priority for fast routing paths. Since much of our work is concerned with timing-driven routing, we will focus on these three algorithms. At their core, all these routers use variants of maze routers [102] to connect the terminals of each net. A maze router essentially consists of running Dijkstra’s algorithm [103] to find the shortest (lowest total cost) path between a net source node and a net sink node in a routing-resource graph. All of these algorithms perform multiple routing iterations in which some or all of the nets are ripped-up and rerouted by different paths to resolve competition for routing resources or improve circuit speed. Both [94] and [95] use timing analysis only to help identify good nets to rip-up and re-route — nets which are likely to lead to a circuit speed-up if they can be rerouted using a faster path. Ripping-up and re-routing these nets only changes the net ordering, however; they are all routed by the same maze routing algorithm, regardless of how timing-critical they are. The PathFinder negotiated congestion-delay algorithm [85] uses a more sophisticated technique in which the congestion-delay trade-off of each connection is controlled by how timing critical it is. In other words, a timing-critical connection will be routed by a minimum delay path even if it is congested, while a non-timing-critical net will take a longer, uncongested path. This algorithm produces excellent results and incorporates several important ideas, so we describe it in detail below.

Pathfinder repeatedly rips-up and re-routes every net in the circuit until all congestion is resolved — this idea is due to Nair [104]. Ripping-up and re-routing every net in the circuit once is called a *routing iteration*. During the first routing iteration, every connection is routed for minimum delay, *even if this leads to congestion*, or overuse, of some routing resources. A circuit routing in which some routing resource is overused, such as a wire being used by two different nets, is not a legal routing. Consequently, when overuse exists at the end of a routing iteration, another routing iteration (or more) must be performed to resolve this congestion. After each routing iteration the cost of overusing a routing resource is increased, so that the probability of resolving all congestion increases. At the end of each routing iteration we have a complete, but potentially somewhat illegal, routing. We can therefore determine the net delays from this routing, and perform a full timing analysis to compute the slack (see Section 2.2.5) of each source-sink connection. These slack values are used in the next routing iteration to control how much attention each connection pays to delay, and how much is paid to congestion-avoidance. Pseudo-code for the algorithm is given in Figure 2.10.

The criticality of the connection from the source of net i to one of its sinks, j , is:

$$Crit(i, j) = 1 - \frac{slack(i, j)}{D_{max}} \quad (2.4)$$

where D_{max} is the delay of the circuit critical path, and $slack(i, j)$ is the amount of delay that could be added to this connection before it affected the circuit's critical path. $Crit(i, j)$ is therefore between 0 and 1.

The cost of using a routing resource node, n , as part of connection (i, j) is

$$Cost(n) = Crit(i, j) \cdot delay(n) + [1 - Crit(i, j)] \cdot [b(n) + h(n)] \cdot p(n). \quad (2.5)$$

The first term in (2.5) is the delay sensitive term — the criticality of the connection times the intrinsic delay of the node. The second term is the congestion sensitive term. $b(n)$ is the base cost of node n , and is set equal to $delay(n)$ in [85]. $h(n)$ is the historical congestion of node n ; it is increased after every routing iteration in which node n is overused and gives the router “congestion memory.” $p(n)$ is the present congestion cost of node n ; it is 1 if using this node to route the current connection will not cause any overuse, and increases with the amount of overuse of the node. $p(n)$ is also a function of the number of routing iterations that have been performed. In early iterations, $p(n)$ grows slowly with the current overuse of node n ; in later iterations, $p(n)$ goes up very rapidly with overuse of node n .

```

Let:  $RT(i)$  be the set of nodes,  $n$ , in the current routing of  $net(i)$ .

 $Crit(i,j) = 1$  for all nets  $i$  and sinks  $j$ ;
while (overused resources exist) { /* Illegal routing? */

    for (each net,  $i$ ) {
        rip-up routing tree  $RT(i)$  and update affected  $p(n)$  values;
         $RT(i) = NetSource(i)$ ;

        for (each sink,  $j$ , of  $net(i)$  in decreasing  $crit(i,j)$  order) {
            PriorityQueue =  $RT(i)$  at  $PathCost(n) = crit(i,j) \cdot delay(n)$  for
                each node  $n$  in  $RT(i)$ ;
            while (sink( $i,j$ ) not found) {
                Remove lowest cost node,  $m$ , from PriorityQueue;
                for (all fanout nodes  $n$  of node  $m$ ) {
                    Add  $n$  to PriorityQueue at  $PathCost(n) =$ 
                         $Cost(n) + PathCost(m)$ ;
                }
            }

            for (all nodes,  $n$ , in path from  $RT(i)$  to sink( $i,j$ )) { /* Backtrace */
                Update  $p(n)$ ;
                Add  $n$  to  $RT(i)$ ;
            }
        }
    }

    Update  $h(n)$  for all  $n$ ;
    Perform timing analysis and update  $Crit(i,j)$  for all nets  $i$  and sinks  $j$ ;
} /* End of one routing iteration */

```

FIGURE 2.10 Pseudo-code of the Pathfinder routing algorithm.

The excellent performance of Pathfinder is due to two innovations: allowing overuse of routing resources, and using the cost function of (2.5) to allow congestion to gradually be resolved, and timing to be directly optimized. By slowly increasing the cost of congestion, via $p(n)$ and $h(n)$, as more routing iterations are performed, connections that are on or near the critical path tend to take the fastest paths and stay there, while less timing-critical connections are gradually forced off any overused nodes onto slower paths.

Notice that the router of Figure 2.10 uses a breadth-first search through the routing resource graph to connect net terminals. The creators of Pathfinder [85] also describe

an enhancement to the basic algorithm that uses an A*, or directed, search [105] to speed execution. In Figure 2.10 a node is added to the priority queue with a PathCost equal to the sum of all the node costs along the path up to and including it. To convert this to an A* search, one simply adds this term to a lower bound on the sum of the node costs needed to reach the target sink from the current node; the result is then used as the sort value when the node is added to the priority queue.

Finally, Ebeling et al [85] also describe a purely routability-driven variant of the Pathfinder router, which they call the Pathfinder negotiated congestion algorithm. This algorithm simply sets the criticality of every net, $Crit(i,j)$, to 0 so that the cost of a node is given solely by the congestion-sensitive term in (2.5). As well, this router connects the current routing tree to the first net sink found, rather than a pre-determined target sink, during maze expansion.

The router used in the simultaneous placement and routing tool developed by Nag and Rutenbar [86] has one unique feature of interest. This router is again maze-router based, and it is targeted at Xilinx XC4000 series FPGAs. When routing a multi-terminal net, a maze router will typically route to the closest sink, and then use this partial routing as the source (start point) when it attempts to route to the next closest sink, and so on. This can cause problems when routing high-fanout nets on FPGAs that contain some long wire segments. High-fanout nets typically span most of the FPGA, but the distance from one sink to the next closest sink is usually only a few logic blocks. Consequently, a traditional maze router, that looks only at how to connect the partial routing to the next closest sink, will tend to use short wires to build the routing trees for high-fanout nets, even though using longer wires would be more efficient and result in faster nets. Nag solved this problem by dramatically reducing¹ the cost of using a long line when the net bounding box spanned most of the FPGA. The router then saw the cost of using a long wire to be less than that of even one short wire, so it would use long lines to connect even to nearby sinks and construct a routing tree of long wires for high-fanout nets. This idea of varying the cost of resources depending on the type of connection being routed, or *dynamic weighting*, is a powerful one; in some sense Pathfinder's varying the weighting of delay and congestion according to the criticality of the connection being routed is another example of this idea.

One shortcoming of current non-commercial timing-driven FPGA routers is that they are designed to optimize only the linear delay model, in which every routing resource has a fixed delay.¹ Most FPGAs contain at least some pass transistor switches in their routing, so the delay of a routing resource actually depends on the topology of the net

1. SEGA can use a more advanced delay model, but [36] showed that SEGA achieved better speed with a cost function that emphasized recombining the two-point nets passed to it by the global router into multi-terminal nets than it did with this delay-based cost function.

using it. As well, since the FPGAs we study in Chapter 7 include buffers in the routing, the router must understand when to use a buffered switch and when to use a pass transistor. Unfortunately, no non-commercial FPGA router is “buffer-aware.” The Xilinx commercial router, described by Trimberger in [106], is buffer-aware and uses the Penfield-Rubinstein [107] delay model during routing. It is likely that the routers of other FPGA companies whose products contain a mix of pass transistors and buffers are also buffer-aware, but to our knowledge the algorithms used by these other companies have never been made public.

Considerable work has been done in the standard cell and MPGA routing areas on routing under more accurate delay models [108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121] and buffer insertion [122, 123, 124, 125, 126]. However, much of this work is not easily applicable to the FPGA routing problem because:

1. MPGA and standard cell routers can choose exactly where they want wires, how long these wires should be, and where buffers should be placed. Since all FPGA routing resources are prefabricated, however, FPGA routers are constrained to choose from a set of prefabricated wires and switches. The flexibility of FPGA routing is low enough that if the router decides to connect two wires together it usually has no further choice about whether to insert a buffer or not at the join. There is usually only one switch that can connect these two wires, and whether it is a buffer or a pass transistor was determined when the FPGA was fabricated. In graph theoretic terms, FPGA routing consists of finding Steiner trees embedded in a graph, while MPGA routing consists of finding Steiner trees embedded in the Manhattan plane [90].
2. The complexity of many MPGA and standard cell routing algorithms is quite high — $O(k^3)$ to $O(k^4)$ is common, where k is the number of terminals on a net. Since we will be routing circuits with thousands of nets, and a few of these nets have hundreds of terminals, we must use algorithms with reasonably low (ideally linear) complexity [127].

Nonetheless, some ideas from the MPGA world are relevant to our work. For example, in the absence of congestion, PathFinder attempts to greedily optimize a combination of wirelength and (linear delay model) delay to a net’s sinks, which is similar to the approach of Alpert et al in [110]. In a similar vein, [109] and [111] attempt to greedily optimize a combination of the Elmore delay and wirelength; it should be possible to adapt these algorithms to FPGAs. The time complexity of these algorithms is quite high, however. The algorithm of [111] is $O(k^4)$, and while the authors do not give the complexity of their algorithm in [109], it appears to be at least $O(k^3)$. Since adapting these algorithms to FPGAs involves routing within a large routing-resource graph, their complexity may increase even further and make them impractical.

2.2.4 Delay Modelling

One must compute the delay of a route from a net source to any of its sinks in order to:

1. Determine the speed of a circuit after it has been routed, and
2. Determine the delay of different net topologies during routing.

Ideally, one would use a circuit simulator such as SPICE to obtain highly accurate delay estimates, but the CPU time required to run SPICE on the thousands of nets in a typical circuit is prohibitive. Instead, previous researchers have modelled pass transistors as linear resistors and wires as an RC pi-network, so that a net's routing may be modelled as an RC-tree [22]. In [22], the Penfield-Rubinstein delay model [107] was then used to determine an upper and lower bound on the delay of the RC-tree to each of the net sinks. An alternative to the Penfield-Rubinstein model is the Elmore delay [128], which is the most widely used delay estimate in routing research [108]. The Elmore delay was originally defined only for RC-trees, but it has been combined with a common model of buffer delay to allow its use with circuits that contain buffers, as well as resistors and capacitors [112]. Each buffer is modelled by a constant delay and a resistor. The constant delay accounts for the intrinsic delay of the buffer, while the resistance accounts for the load-dependence of the buffer delay. Figure 2.11 shows the RC-model for each of the three elements of FPGA routing. Note that pass transistors and buffers attached to a wire add parasitic capacitance regardless of whether they are on or off.¹

The Elmore delay of a source-sink path is then [112]:

$$\sum_{i \in \text{Source-sink path}} R_i \cdot C(\text{subtree}_i) + T_{d,i} \quad (2.6)$$

where $T_{d,i}$ is the intrinsic delay of a buffer if element i is a buffer, and 0 otherwise. R_i is the equivalent resistance of element i (R_{wire} , R_{buf} , or R_{pass}). In (2.6), $C(\text{subtree}_i)$ is the total capacitance of the dc-connected subtree rooted at element i — that is the total downstream capacitance which is not isolated by buffers.

1. Notice that we model the capacitance of both “on” and “off” pass transistors as being purely due to diffusion capacitance. In fact the capacitance of an “on” pass transistor is larger than that of an “off” pass transistor, since the channel created when a transistor is “on” has capacitance to the gate and to the substrate. Since relatively few pass transistors are “on” at any time and most of the capacitance in an FPGA is metal capacitance, the error in total capacitance caused by this approximation is small (~1% to 2%).

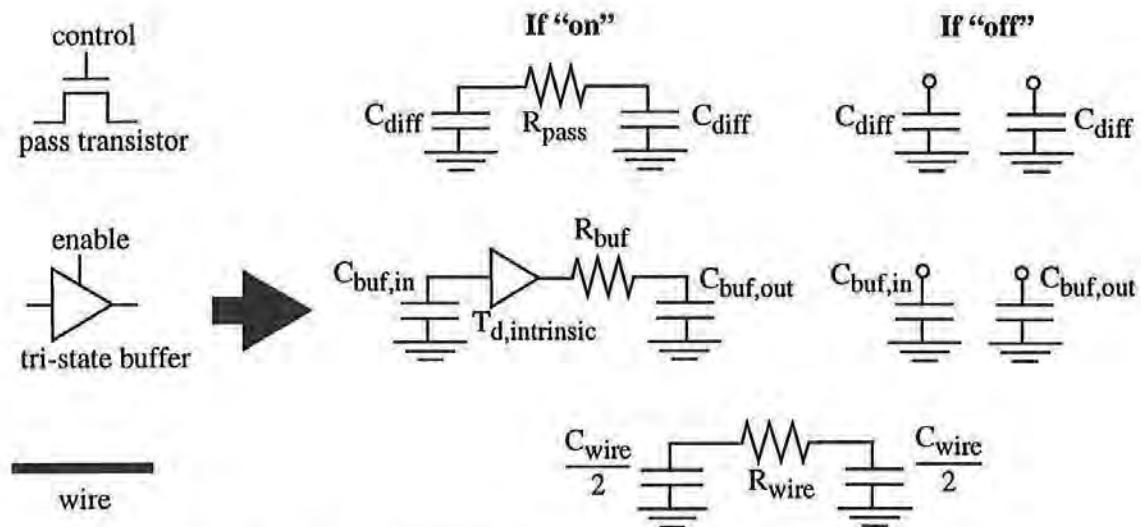


FIGURE 2.11 Equivalent circuits for FPGA routing elements.

The Elmore delay of an RC-tree can be computed in linear time, and we will show in Section 7.2.3 that its accuracy is quite good. As well, [129, 130] showed that even when the Elmore delay is somewhat inaccurate it still tends to have high *fidelity*. That is, it still tends to rank the speed of different routing topologies correctly. This is a useful feature, as it means that the relative comparisons of topologies made by a router using it will correctly rank the alternatives, and that comparisons of FPGA architecture speed made via Elmore delay will correctly rank the different architectures.

2.2.5 Timing Analysis

Timing analysis [131] is used for two basic purposes:

1. To determine the speed of circuits which have been completely placed and routed, and
2. To estimate the *slack* of each source-sink connection during routing (or other parts of the CAD flow) in order to decide which connections must be made via fast paths to avoid slowing down the circuit.

Normally one performs timing analysis on a directed graph representing the circuit structure [131]. Nodes represent the input pins and output pins of basic circuit elements, such as registers and LUTs, and edges are added between the inputs of combinational logic blocks (e.g. LUTs) and their outputs and between pins which the circuit netlist specifies are connected. Each edge is annotated with the delay required to pass

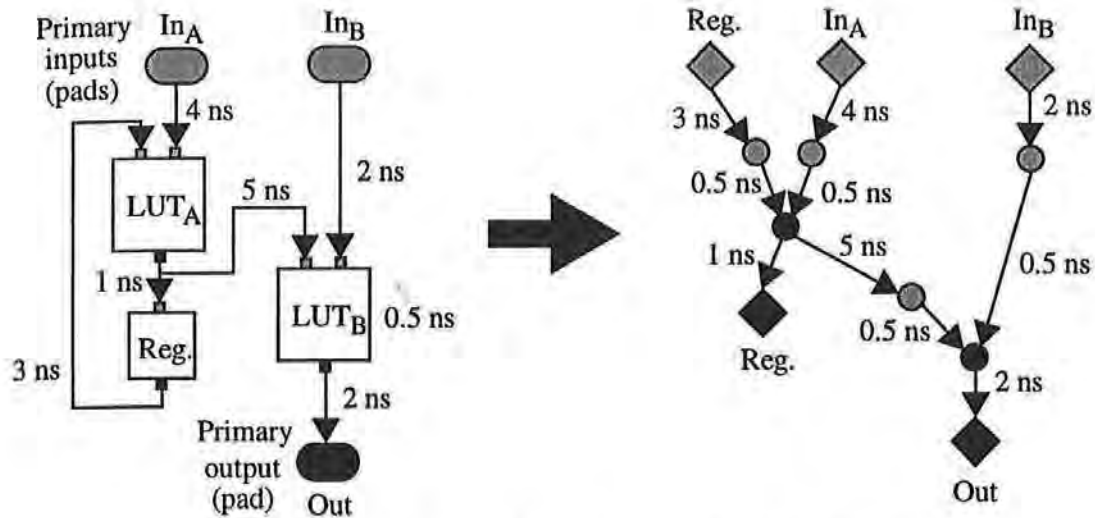


FIGURE 2.12 The timing graph for a simple circuit.

through the circuit element or routing. Register input pins are not joined to register output pins — register outputs have no edges incident to them, and register inputs have no edges leaving them. Similarly, primary inputs (input pads) have no incident edges, and primary outputs (output pads) have no exiting edges. Figure 2.12 shows a simple circuit implemented via 2-input LUTs and registers, and the corresponding timing graph.

One can determine the minimum required clock period with $O(n)$ computation for a timing graph with n nodes via a breadth-first traversal of the timing graph. The traversal begins at nodes with no incident edges (primary inputs and register outputs) and labels each with a signal *arrival time*, T_{arrival} , of 0. Each node which has incident edges only from labelled nodes is then labelled with its arrival time according to

$$T_{\text{arrival}}(i) = \text{Max}_{\forall j \in \text{fanin}(i)} \{T_{\text{arrival}}(j) + \text{delay}(j, i)\} \quad (2.7)$$

where node i is the node being labelled, and $\text{delay}(j, i)$ is the delay value marked on the edge joining node j to node i . This procedure continues until every node in the graph has been labelled. The node with the largest arrival time, which will always be a primary output or a register input, then defines the maximum delay, D_{max} , (= minimum clock period) through the circuit. In Figure 2.12, for example, the arrival time at node *Out* is 12 ns, and this is the largest arrival time, and hence the maximum delay, in the circuit.

The above procedure is sufficient to determine the speed of a circuit, but during timing-driven placement or routing we also want a metric indicating how important each source-sink connection is in terms of its effect on the circuit's delay. The *slack* of a

connection is defined as the amount of delay which could be added to this connection without increasing the minimum cycle time of the circuit (if the delays of all other connections remained constant). To compute the slack, we perform a second breadth-first traversal of the timing graph. This time each node with no outward edges has its *required time*, $T_{required}$, set to D_{max} and a backward breadth-first traversal of the graph is performed to label the remaining nodes. The required time of any node with fanout (outward edges) is

$$T_{required}(i) = \text{Min}_{j \in \text{fanout}(i)} \{T_{required}(j) - \text{delay}(i, j)\} \quad (2.8)$$

The slack of the connection from node i to node j is then

$$\text{slack}(i, j) = T_{required}(j) - T_{arrival}(i) - \text{delay}(i, j) \quad (2.9)$$

Connections with a slack of zero are on the circuit critical path — any increase in the delay of such a connection will lead to a corresponding increase in the circuit's delay. Connections with large slacks, on the other hand, could be routed via significantly slower paths without affecting a circuit's delay. In Figure 2.12, for example, the slack of the connection between In_A and LUT_A is 0, while the slack between In_B and LUT_B is 7.5 ns.

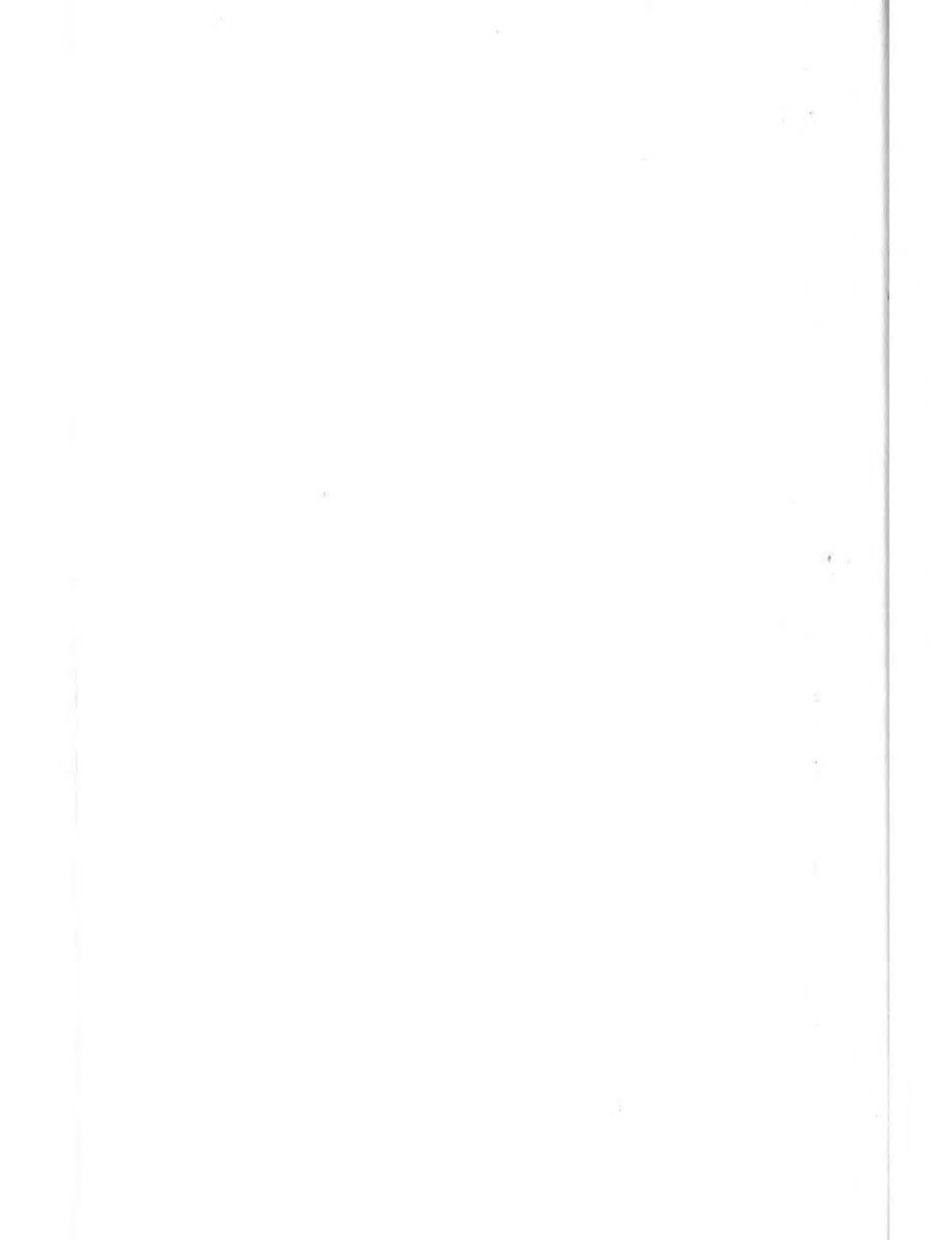
Slack allocation [132, 133, 95] is used to determine a set of upper bounds on connection delays from a set of slacks, such that the circuit speed is guaranteed not to decrease as long as each connection is routed with delay less than its upper bound. Some timing-driven placement and routing tools require such upper bounds to guide them, while other tools work with the net slack values directly.

2.3 Summary

In this chapter we first reviewed the basics of FPGA architecture. We then detailed the terminology used to describe FPGA logic blocks and FPGA routing architecture, and presented some of the important prior research into both areas. In the second half of the chapter, we described the CAD flows used with FPGAs. We focused particularly on algorithms for logic block packing, placement and routing, and described some of the strengths and weaknesses of prior approaches. Finally, we described the delay modelling and timing analysis procedures used both to determine how fast a circuit will operate, and to help guide timing-driven CAD tools.

The next chapter describes the CAD tools we developed to perform logic block packing and placement within an FPGA. Chapter 4 describes the new FPGA router we have developed. Our placement tool is simulated annealing based, and our router makes use of many of the ideas of the Pathfinder algorithm. Since both simulated annealing and Pathfinder were described in this chapter, Chapters 3 and 4 will focus on the new ideas and enhancements in these tools.

Chapters 5, 6 and 7 then apply these CAD tools to investigate three different aspects of FPGA architecture: global routing architecture, logic block clusters, and detailed routing architecture.



CAD Tools: Packing and Placement

This chapter describes the logic block packing tool we developed to target cluster-based logic blocks and the new and novel parts of our FPGA placement tool.

A key goal for these tools, and of the router discussed in Chapter 4, is to be able to target a wide variety of FPGA architectures in order to enable architectural exploration. Furthermore, these tools should be responsive to the variation of the FPGA architectural parameters — they should directly optimize for the different features of the FPGAs we investigate. Finally, we require the result quality of the tools to be high, since low-quality tools can lead to inaccurate architectural conclusions.

3.1 Logic Block Packing

Recall that the logic block packing problem (introduced in Section 2.2.1) consists of packing the LUTs and registers generated by a LUT-based technology mapping program into larger logic blocks. The next section describes the exact structure of the “cluster-based” logic blocks we wish to target. Section 3.1.2 details the first algorithm we developed to pack LUTs and registers into these logic blocks, while Section 3.1.3 describes enhancements to this basic algorithm to make it timing-driven and improve its performance. While Altera has an in-house tool targeting cluster-based logic blocks and Xilinx has an in-house tool targeting the “cluster-like” logic blocks of the 5200 FPGA, we know of no published work describing algorithms that target cluster-based logic blocks.

3.1.1 Cluster-Based Logic Blocks

The structure of a cluster-based logic block, which we also call a *logic cluster*, is shown in Figure 3.1 [8, 9]. The cluster-based logic blocks we investigate have a two-level hierarchy: the overall block is a collection of *basic logic elements* (BLEs). As shown in Figure 3.1a, our basic logic element consists of a LUT and a register, and the BLE output can be either the registered or unregistered version of the LUT output. This is how many commercial FPGAs combine a LUT and a register to create a structure capable of implementing either combinational or sequential logic. The complete logic block contains N BLEs and local routing to interconnect them, as shown in Figure 3.1b.

A logic cluster is described with the following four parameters:

1. the size of (number of inputs to) a LUT (K),
2. the number of BLEs in a cluster (N),

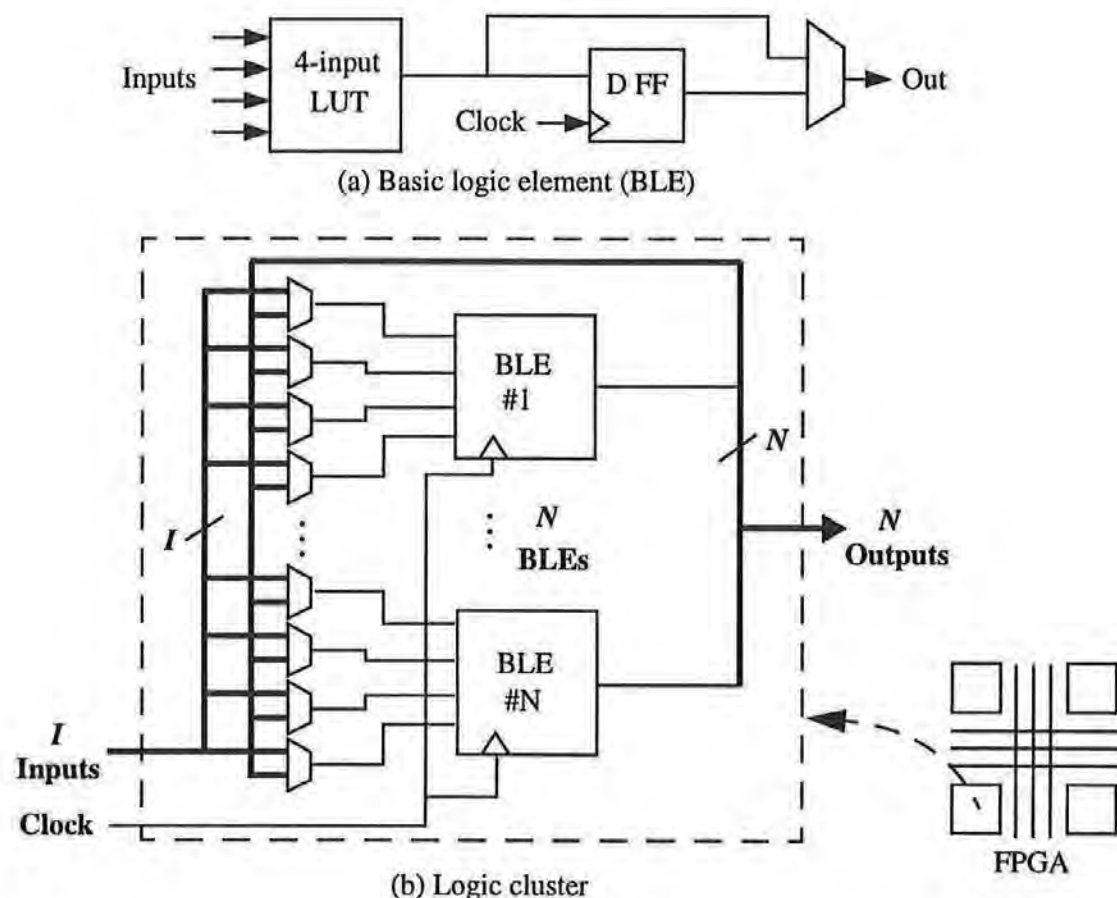


FIGURE 3.1 Structure of (a) basic logic element (BLE) and (b) logic cluster.

3. the number of inputs to the cluster for use as inputs by the LUTs (I), and
4. the number of clock inputs to a cluster (for use by the registers), M_{clk} .

In this book, we focus on logic clusters in which the LUT size, K , is 4 and the number of clock pins on a cluster, M_{clk} , is 1 — this is the case shown in Figure 3.1. Nonetheless, our logic block packing tools are capable of targeting logic clusters with different values of K and M_{clk} [134].

As Figure 3.1 shows, not all $K \cdot N$ look-up table inputs are accessible from outside the logic cluster. Instead, only I external inputs are provided to the logic cluster — multiplexers allow arbitrary connections of these cluster inputs to the BLE inputs. The same multiplexers also connect to each of the N BLE outputs, allowing the output of any BLE within the cluster to be connected to any of the BLE inputs. All N outputs of the logic cluster can be connected to the FPGA routing for use by other logic clusters.

Notice that each of the BLE inputs can be connected to any of the cluster inputs or any of the BLE outputs. Logic clusters are therefore *fully-connected*. This is a useful feature, as it simplifies our CAD tools considerably. For example, determining if all the appropriate signals can be connected to the BLE inputs reduces to counting the number of different input signals needed by the BLEs in a cluster that are not generated within the cluster, and comparing it to the number of cluster inputs, I . The logic cluster used in the Altera 8K and 10K FPGAs is fully connected [7], and the logic block used in the Xilinx XC5200 FPGA is nearly fully connected [11].

3.1.2 Basic Logic Block Packing Algorithm: VPack

In this section we describe our basic logic block packing tool, VPack. VPack takes as input a netlist of LUTs and registers, and outputs a netlist of logic clusters. The parameters describing the exact logic cluster to be targeted (N , I , K , and M_{clk}) are all specified on the command line; VPack can target a logic cluster corresponding to any combination of these parameters. VPack groups the LUTs and registers into logic clusters in two stages: first, it packs LUTs and registers together into BLEs, and then it packs multiple BLEs into logic clusters. The entire algorithm is outlined via pseudo-code in Figure 3.2, and is described in detail below.

The first stage of the algorithm uses a simple and optimal pattern matching algorithm to pack a register and a LUT together into one BLE whenever possible. Figure 3.3 shows that when the output of a LUT fans out only to a single register, both the LUT and register can be packed into a single BLE. Any other interconnection pattern requires that a LUT and register be assigned to two separate BLEs.

```

Let: UnclusteredBLEs be the set of BLEs not contained in any cluster
C be the set of BLEs contained in the current cluster
LogicClusters be the set of clusters (where each cluster is a set of BLEs)

UnclusteredBLEs = PatternMatchToBLEs (LUTs, Registers);
LogicClusters = NULL;

while (UnclusteredBLEs != NULL) { /* More BLEs to cluster */
    C = GetBLEwithMostUsedInputs (UnclusteredBLEs);
    while (|C| < N) { /* Cluster is not full */
        BestBLE = MaxAttractionLegalBLE (C, UnclusteredBLEs);
        if (BestBLE == NULL) /* No BLE can be added to cluster */
            break;
        UnclusteredBLEs = UnclusteredBLEs - BestBLE;
        C = C  $\cup$  BestBLE;
    }

    if (|C| < N) { /* Cluster not full — try hill-climbing */
        while (|C| < N) {
            BestBLE = MinClusterInputIncreaseBLE (C, UnclusteredBLEs);
            C = C  $\cup$  BestBLE
            UnclusteredBLEs = UnclusteredBLEs - BestBLE;
        }
        if (ClusterIsIllegal (C))
            RestoreToLastLegalState (C, UnclusteredBLEs);
    }

    LogicClusters = LogicClusters  $\cup$  C;
}

```

FIGURE 3.2 Pseudo-code for VPack.

The second, and more complex, stage of VPack packs these BLEs into logic clusters. The optimization goals are twofold. First, we want to fill each logic cluster to its capacity, N , in order to minimize the number of logic blocks needed. Second, we want to minimize the number of inputs to each cluster in order to reduce the number of connections to be routed between clusters, and hence enhance routability. To be packed into a single logic cluster, a group of BLEs must satisfy three conditions:

1. The number of BLEs must be less than the cluster size, N ;
2. The number of distinct signals generated outside the cluster and used as inputs to the cluster BLEs must be less than or equal to I ; and
3. The number of different clocks used by the BLEs must be less than M_{clk} .

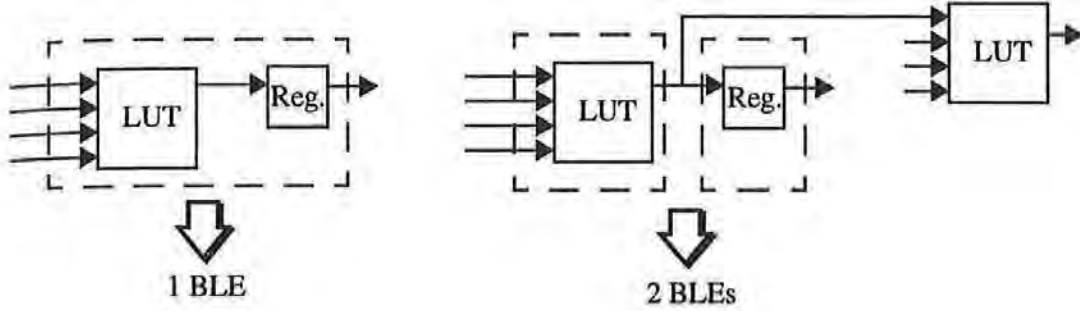


FIGURE 3.3 Packing LUTs and registers into BLEs.

VPack constructs each cluster sequentially. The algorithm has two phases: an initial phase that greedily packs BLEs into a cluster, and a hill-climbing phase that is invoked only if the greedy phase is unable to completely fill a cluster. The greedy phase begins by choosing a seed BLE for the current cluster. We have found that the best way to choose this seed is to select the unclustered BLE with the most used inputs, as such BLEs use the most cluster inputs, which are a scarce resource. Next, VPack selects the BLE with the highest *attraction* to the current cluster which could legally be added to the current cluster, and adds it to the cluster. The attraction between a BLE, B , and the current cluster is the number of inputs and outputs they have in common:

$$\text{Attraction}(B) = |\text{Nets}(B) \cap \text{Nets}(C)| \quad (3.1)$$

where C denotes the current cluster. This attraction function tends to group related BLEs into clusters, and minimize the number of distinct inputs to cluster. A BLE with a high attraction to the current cluster will not be added to the cluster if it would result in an illegal cluster — a cluster that needs more than I inputs, or more than M_{clk} different clocks.

This procedure of greedily selecting a BLE to add to the current cluster continues until either the cluster is full or adding any unclustered BLE would make the current cluster illegal. If the cluster is full, we select a new seed BLE and begin packing BLEs into a new cluster. If however, the cluster occupancy is less than N and we cannot add any BLEs because of a lack of cluster inputs, a second, hill-climbing, phase of VPack is invoked.

Since we know that any clusters that reach this second phase will be difficult to pack to capacity, VPack now selects BLEs to add to the cluster in order to minimize the increase in the number of cluster inputs required. The number of additional cluster inputs required when a BLE, B , is added to a logic cluster is simply

$$\Delta_{\text{cluster inputs}}(B) = |\text{Fanin}(B)| - |\text{Nets}(B) \cap \text{Nets}(C)| \quad (3.2)$$

In this hill-climbing phase, VPack also allows BLEs to be added to a cluster even if it results in a cluster that is infeasible because it needs more than I inputs (but it does not allow the number of clocks to exceed M_{clk}). Note that adding a BLE to a cluster in which all of its inputs are already present, and in which the output of the BLE is used by some other BLE already in the cluster *decreases* the number of distinct inputs to the cluster by one. Figure 3.4 shows this situation. This is the key to the hill-climbing phase; while adding one BLE to a cluster makes it infeasible, it may become feasible again when additional BLEs are added. The hill-climbing phase terminates when the cluster is full; if the cluster is still infeasible VPack backs up to the last point at which the cluster was feasible. VPack then selects a seed BLE for the next cluster and invokes the first phase again, as before.

The hill-climbing phase leads to only a small improvement in the number of BLEs packed into each cluster, which we call logic utilization, vs. using the greedy phase alone. Many circuits show no improvement, while others show only 1 - 2% gains in logic utilization. The overall average logic utilization improvement is under 1%. A likely reason for this small improvement is that the greedy first phase of VPack works very well for clustering problems of this type. It is possible, however, that the small improvement is because of the limited type of hill-climbing we are allowing. Logic replication moves could be added to the hill-climbing phase in a very natural manner, and this would likely improve the utility of hill-climbing.

This packing algorithm is very efficient. The largest benchmark circuit used in this book contains 8381 four-input LUTs and 33 registers — packing these LUTs and registers into logic clusters requires 3 seconds of CPU time on a 300 MHz UltraSparc workstation. The complexity of the algorithm is $O(k_{\max} \cdot K \cdot n)$, where $k_{\max}$ is the maximum number of terminals on a net, K is the number of inputs to each LUT, and n is the number of LUTs plus the number of registers in the circuit.

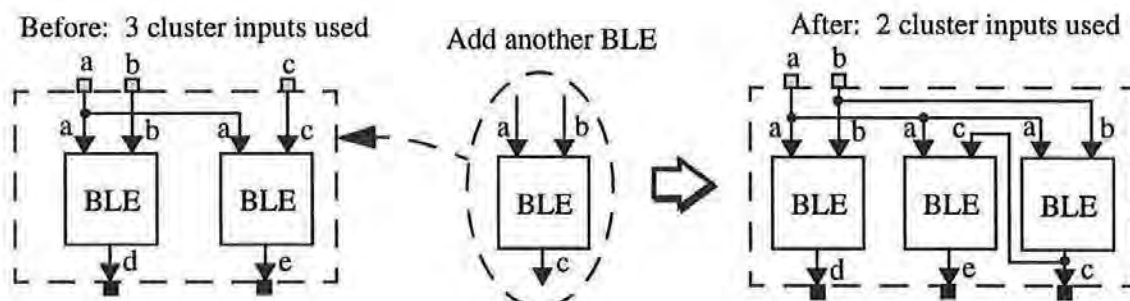


FIGURE 3.4 Adding a BLE to a cluster can decrease the number of used cluster inputs.

3.1.3 Timing-Driven Logic Block Packing: T-VPack

Our timing-driven logic block packing algorithm, T-VPack [10, 19], attempts not only to pack each logic block to capacity, but also to minimize the number of inter-cluster (between cluster) connections on the critical path. The local routing within clusters is faster than the general-purpose routing between logic clusters, so reducing the number of inter-cluster connections on the critical path reduces circuit delay. The basic operation of the algorithm is the same as that of the VPack algorithm described in the previous section: T-VPack first chooses an unclustered BLE as the seed of a new cluster, and then sequentially adds unclustered BLEs with the greatest attraction to the current cluster until the cluster is full. The T-VPack algorithm differs from VPack in how it chooses the seed BLE for a new cluster and in the attraction function it uses.

Cluster Seed and Attraction Function

In order to minimize the number of inter-cluster connections on the critical path, T-VPack first needs to determine which connections are on the critical path. Accordingly, T-VPack performs timing analysis (described in Section 2.2.5) to determine the slack of each connection between BLEs. The timing analyzer within T-VPack models three types of delay: the delay through a BLE, or *LogicDelay*, the connection delay between blocks within the same cluster or *IntraClusterConnectionDelay*, and the connection delay between blocks that are in different clusters, or *InterClusterConnectionDelay*. The delay of a connection between two BLEs in different logic clusters is not known until after a circuit has been placed and routed, so T-VPack approximates the delay between clusters as a constant *InterClusterConnectionDelay*. Note that this leads to some inaccuracy in T-VPack's estimate of where the critical path lies, so that sometimes T-VPack will be attempting to shorten a path which will not be part of the post-place-and-route critical path. The performance of T-VPack is not very sensitive to the exact values chosen for these three delay parameters. Throughout this work we set *LogicDelay* to 0.1, *IntraClusterConnectionDelay* to 0.1 and *InterClusterConnectionDelay* to 1.0.

We define the criticality of a connection, i , to be

$$\text{ConnectionCriticality}(i) = 1 - \frac{\text{slack}(i)}{\text{MaxSlack}} \quad (3.3)$$

where *MaxSlack* is the largest slack amongst all connections in the circuit.

The first (seed) BLE packed into a new cluster is the unclustered BLE with the highest *Criticality*. We explain exactly how the Criticality of a BLE is computed below; basically, BLEs attached to connections with high *ConnectionCriticalities* have high

criticalities. Once a seed is chosen, the attraction function used to determine the next unclustered BLE, B , to be added to the current cluster, C , is:

$$Attraction(B) = \lambda \cdot Criticality(B) + (1 - \lambda) \cdot \frac{|Nets(B) \cap Nets(C)|}{MaxNets} \quad (3.4)$$

Notice that the second term in (3.4) is essentially the attraction function from the original VPack algorithm. The MaxNets factor in the denominator of this term is the maximum number of nets that could connect to any BLE ($I + N + M_{clk}$) and simply normalizes the magnitude of the second term. The first term in (3.4) promotes the grouping of BLEs with high criticalities (defined below) into the current cluster in order to minimize delay. λ is a parameter that controls the trade-off between net sharing and delay minimization. If λ is 0, we have an algorithm that focuses solely on minimizing the number of used inputs to a cluster, and is equivalent to the basic VPack algorithm of Section 3.1.2. If λ is 1, the algorithm focuses solely on minimizing the delay of a circuit, with no regard for how many nets are shared by the BLEs within a cluster. We have experimentally determined that any λ value in the range from 0.4 to 0.8 produces packings of essentially the same quality in terms of both post-place-and-route delay and routing area. Throughout this work we set λ to 0.75.

The *Criticality* of a BLE is computed as follows. Let us first define the base criticality of an unclustered BLE, or *BaseBLECrit*(B). *BaseBLECrit* is defined slightly differently depending on whether we are choosing a seed BLE for a new cluster or computing the attraction of a BLE to the current cluster:

1. When we are choosing a seed BLE, *BaseBLECrit*(B) is the maximum ConnectionCriticality value amongst all of BLE B 's connections; and
2. When we are computing the attraction of a BLE to the current cluster, *BaseBLECrit*(B) is the maximum ConnectionCriticality value amongst all the connections joining BLE B to BLEs within the cluster currently being packed, C . If a BLE does not have any connections to C then its base criticality score is zero.

In Figure 3.5 we illustrate how the *BaseBLECrit* values are assigned when we are computing the attraction of a BLE to the current cluster. Each connection between unclustered BLEs and BLEs within the cluster C is labelled with its ConnectionCriticality value. Notice how the base criticality of each BLE is set to the highest criticality amongst the connections between it and the cluster being packed.

During packing, multiple BLEs often have the same base criticality value. In this case, we use a tie-breaker mechanism to select which BLEs are the most beneficial to pack. This mechanism is designed to choose (from the BLEs tied with the highest base criticality value) the BLE whose packing would reduce the length of the largest number of critical paths. This is best illustrated by an example.

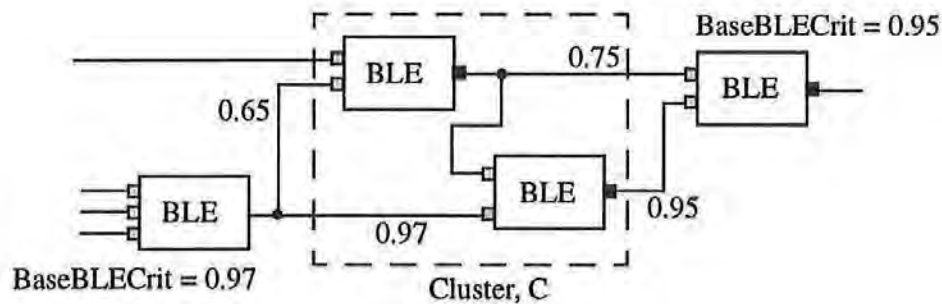


FIGURE 3.5 Determining BaseBLECrit from connection criticalities.

In Figure 3.6 we have darkened connections and BLEs on the critical paths. Notice that when selecting which BLEs to place into a cluster, it is more beneficial to absorb certain critical BLEs over other critical BLEs. In this case, absorbing BLEs X, Y, and Z would be much more beneficial than absorbing BLEs Q, T, and V. We can see that absorbing X, Y, and Z affects three partially-overlapping critical paths, and will reduce the criticality of seven other BLEs (Q, R, S, T, U, V and W). On the other hand, absorbing Q, T, and V affects only one critical path, and will not reduce the criticality of any other BLEs, since all the other critical BLEs would still lie on a critical path after the packing of Q, T, and V into a single cluster. Clearly it is best to cluster BLEs that reduce the criticalities of the most other BLEs.

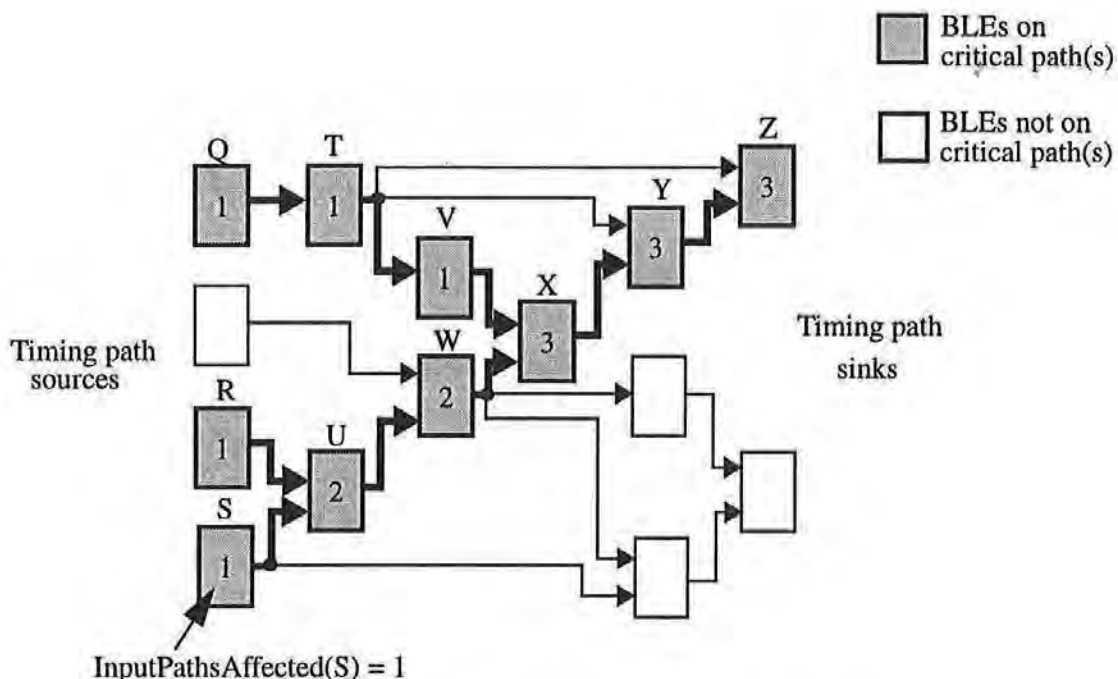


FIGURE 3.6 Example of first criticality tie-breaker.

We define three variables that keep track of the number of critical paths that each BLE in the circuit affects. First we define *InputPathsAffected* as the number of critical paths between timing path sources (primary inputs or register outputs) in the circuit and the BLE currently being labelled. Next we define *OutputPathsAffected* as the number of critical paths between the BLE currently being labelled and the timing path sinks (primary outputs or register inputs). Finally, we define *TotalPathsAffected* as the sum of the previous two variables. The calculation of these variables is explained below.

In Figure 3.6 each BLE is labelled with its *InputPathsAffected* value. We assign any timing path source nodes that are on the critical paths an *InputPathsAffected* value of one, and all other timing path source nodes are assigned an *InputPathsAffected* value of zero. Then we perform a breadth-first traversal of the circuit starting at the sources, and define the *InputPathsAffected* value as

$$InputPathsAffected(B) = \sum_{\forall D \in critical\ inputs(B)} InputPathsAffected(D) \quad (3.5)$$

where *critical inputs(B)* refers to the BLE(s) driving the connections on *B*'s input(s) that are on the critical path. The *OutputPathsAffected* variable is calculated in the same manner, but it starts at timing path sink nodes and works back toward the timing path sources.

$$OutputPathsAffected(B) = \sum_{\forall D \in critical\ outputs(B)} OutputPathsAffected(D) \quad (3.6)$$

TotalPathsAffected is then simply

$$TotalPathsAffected(B) = InputPathsAffected(B) + OutputPathsAffected(B) \quad (3.7)$$

When two BLEs have the same *BaseBLECrit*, we break the tie by choosing to insert the BLE with the higher *TotalPathsAffected* value in the cluster. While this breaks most ties, it does not resolve all of them. Consider the simple circuit shown in Figure 3.7, and the selection of a seed BLE for the first cluster. There is only one path through the circuit, so all 4 BLEs are on the critical path, and have a *BaseBLECrit* value of 1. Similarly, all 4 BLEs have a *TotalPathsAffected* value of 2, so we have a four-way tie for the best BLE to use as a seed. Figure 3.7(a) shows a potential outcome if we randomly choose one of the four tied BLEs as the cluster seed when the cluster size is 2. If we choose BLE F as the cluster seed, and then BLE G is chosen as the second BLE in this cluster, we have “marooned” BLEs E and H — it is not possible to pack either E or H with its fan-in or fan-out. If instead, we choose BLE H as the seed of the first cluster (as Figure 3.7(b) shows), the first cluster would contain G

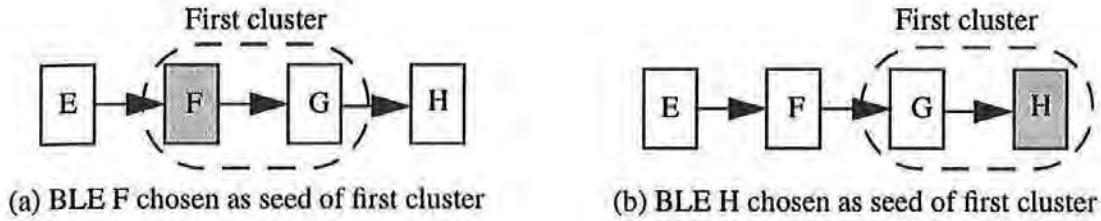


FIGURE 3.7 Example of second criticality tie-breaker.

and H, and E and F could still be packed together into one cluster. Clearly, the clustering shown in Figure 3.7(b) is preferable to that of Figure 3.7(a).

We use a second tie-breaker mechanism to break ties not resolved by the first tie-breaker in order to reduce the likelihood of “marooning” BLEs. We always choose to pack the tied BLE that is the farthest from the timing path sources (i.e. is the closest to the timing sinks).¹ In Figure 3.7, for example, this second tie-breaker causes T-VPack to always choose BLE H as the seed of the first cluster, so the superior clustering solution of Figure 3.7(b) is achieved.

The criticality of a BLE is its BaseBLECrit slightly adjusted by these two tie-breakers:

$$\text{Criticality}(B) = \text{BaseBLECrit}(B) + \epsilon \cdot \text{TotalPathsAffected}(B) + \epsilon^2 \cdot D_{\text{source}}(B) \quad (3.8)$$

where ϵ is a small number (e.g. 0.01) to ensure that the second two terms function only as tie-breakers, and $D_{\text{source}}(B)$ is a BLE’s distance, or level, from the timing path sources.

We have found that incorporating the two tie breakers described above into the criticality portion of the BLE attraction function leads to a 3% increase in the average post-place-and-route circuit speed vs. not including the tie breakers.

Computational Complexity vs. Frequency of Timing Analysis

The computational complexity of T-VPack depends on how often timing analysis is performed on the circuit as BLEs are packed into logic clusters. If timing analysis is performed after each BLE is packed, one has the most up-to-date view of the critical-

1. Note that choosing to always pack the tied BLE closest to the timing path sources would work just as well. The key is simply to ensure that one consistently chooses BLEs from one end of a chain of tied BLEs, rather than from the middle or from a mix of the two ends.

ity of each BLE, but the algorithm is $O(n^2)$, where n is the number of BLEs in the circuit. In this case T-VPack requires about fifteen minutes on a 300 MHz UltraSparc workstation to pack the largest (8383 BLEs) MCNC benchmark circuit [136].

The CPU time can be reduced if timing analysis is performed less often. If timing analysis is performed only after every P BLEs are packed, the algorithm complexity is $O(n^2/P)$. If one performs timing analysis only once before any BLEs have been packed, the algorithm has the same complexity as VPack: $O(k_{\max} \cdot K \cdot n)$ where $k_{\max}$ is the maximum number of terminals of any net and K is the number of inputs to each BLE. In this case, T-VPack requires only a few seconds to pack an 8383 BLE circuit. Interestingly, performing only a single timing analysis and never updating connection criticalities as BLEs are packed does not noticeably degrade the result quality of T-VPack. Apparently the inaccuracy in the post-place-and-route critical path predicted by the timing analyzer in T-VPack (which assumes the same delay for each inter-cluster connection) is large enough that there is no benefit to updating connection criticalities as more information about the circuit packing becomes available. Performing no timing analysis at all (and simply assuming every connection in the circuit is equally critical) does reduce the effectiveness of T-VPack, and reduces circuit speed by 5% to 10%. Accordingly, throughout this book we present the T-VPack results obtained when timing analysis is performed only once, before any BLEs have been packed.

3.1.4 Result Quality of T-VPack vs. VPack

Table 3.1 summarizes the performance of the basic VPack algorithm and the enhanced, T-VPack, algorithm on the 10 largest MCNC benchmark circuits. The logic cluster targeted in this experiment contains 8 BLEs. The inputs required for 98% logic utilization columns in Table 3.1 list the minimum number of input pins (I) a logic cluster must have in order for the packing algorithm to fill clusters to 98% capacity, on average. Notice that T-VPack requires slightly (3%, on average) fewer cluster inputs to achieve 98% utilization than the basic VPack algorithm. The criticality term in the T-VPack attraction function, (3.4), makes T-VPack favour clustering a BLE with its fan-in or fan-out vs. clustering it with BLEs with which it shares inputs. For large clusters (5 or more BLEs) this preference to cluster a BLE with its fan-in and fan-out causes T-VPack to achieve slightly better logic utilization than VPack. Other researchers have also found that grouping circuit blocks with their fan-in or fan-out tends to be an effective clustering technique [52].

The remaining columns in Table 3.1 compare the post-place-and-route performance of circuits packed with the two different algorithms. To generate the results listed in these columns, the number of inputs per cluster was set to 18, which allows good logic utilization for all 10 circuits. For details of the FPGA routing architecture used, see Section 6.2.5.

TABLE 3.1 Comparison of VPack and T-VPack result quality.

Circuit	Cluster Inputs (<i>I</i>) Required for 98 % Logic Utilization		Minimum Channel Width for Successful Routing ($W_{\min}$)		Post-Place-and- Route Critical Path (ns) $W = \infty^a$		Post-Place-and- Route Critical Path (ns) $W = W_{\min} + 20\%$	
	VPack	T- VPack	VPack	T- VPack	VPack	T- VPack	VPack	T- VPack
apex2	19	19	58	55	38.6	32.7	37.9	34.5
clma	17	17	75	64	83.0	64.7	84.3	77.1
elliptic	16	17	57	49	50.1	40.1	60.5	49.1
ex1010	20	20	61	58	45.7	42.8	53.0	56.3
frisc	16	16	57	58	75.1	56.4	80.2	65.3
pdc	20	18	82	76	56.7	53.3	57.9	81.8
s298	18	15	48	28	49.9	49.7	58.0	63.3
s38417	14	14	47	42	51.2	42.3	59.7	45.0
s38584.1	13	12	43	44	40.3	29.3	40.2	30.3
spla	19	18	76	59	49.4	41.0	48.3	47.0
Arith. Av.	17.2	16.6	60.4	53.3	54.0	45.2	58.0	55.0
Geom. Av.	17.0	16.4	59.1	51.6	52.5	44.1	56.3	52.5

a. Note that the congestion-oblivious (infinite routing) delay is *not* a lower bound on the achievable delay, so occasionally the delay with limited routing is lower than the infinite-routing delay. In fact there is currently no known algorithm to route a net with guaranteed minimum Elmore delay, short of exhaustively searching all possibilities [129].

Two columns in Table 3.1 list the minimum number of tracks per channel ($W_{\min}$) required to successfully route the packed circuit produced by each algorithm. Somewhat surprisingly, using T-VPack instead of VPack results in circuits that require 12% fewer tracks for successful routing. To understand the reason for this result, one must compare the structure of the packed circuits produced by T-VPack and by VPack. The criticality term in the T-VPack attraction function, (3.4), makes T-VPack prefer to cluster a BLE with BLEs in its fan-in or fan-out, rather than with other BLEs that share inputs with it. As a result, T-VPack produces circuit packings in which many low-fanout nets have been completely absorbed into logic clusters. Overall, the output of T-VPack has fewer nets to route between clusters than the output of VPack, but the average fanout of each inter-cluster net is higher with T-VPack than with VPack.

The net result is that the output of T-VPack is somewhat easier to route than the output of VPack.¹

The post-place-and-route critical path columns in Table 3.1 compare the relative speeds of circuits implemented with the two different packing algorithms. One set of post-place-and-route speed results assumes that the circuits have essentially an infinite amount of general purpose routing available. In this case, the router is able to focus entirely on speed optimization, rather than congestion avoidance, so we obtain a good estimate of the speed difference between the two packings when the circuits are mapped into routing-rich FPGAs. The $W = W_{\min} + 20\%$ speed results, on the other hand, show the speed difference between the two packings when an FPGA has a more limited amount of interconnect — only 20% more than the minimum required by each circuit packing. As one would expect, T-VPack increases circuit speed vs. VPack, by 19% for the unlimited interconnect case, and 7% for the limited interconnect case, on average. Note also that in the limited interconnect case, the T-VPack circuits are faster than those of VPack despite the fact that the router is being given significantly (on average 12%) fewer tracks to route them than it is for the VPack-generated packings.

Overall it is clear that T-VPack outperforms the basic VPack algorithm in terms of both circuit speed and routing area required.

3.2 Placement: VPR

*Check xilinx papers
for early FPGA
placements.*

The Versatile Place and Route (VPR) CAD tool performs placement and either global routing or combined global-detailed routing for FPGAs [17, 134]. The name Versatile Place and Route reflects the primary goal in this tool's design: be able to target a wide variety of FPGA architectures. In the following sections we describe the capabilities of and algorithms used in the VPR placement tool. In Chapter 4, the routing portion of VPR is described.

1. This result shows the importance of using a full CAD flow, including placement and routing, to evaluate many FPGA issues. It would have been difficult or impossible to guess that the output of T-VPack would be easier to route than the output of VPack without actually placing and routing the outputs of both packing algorithms. In fact, since the circuit packings produced by T-VPack have more point-to-point connections to route between clusters (despite having fewer nets) one would likely guess that VPack's circuits would be easier to route.

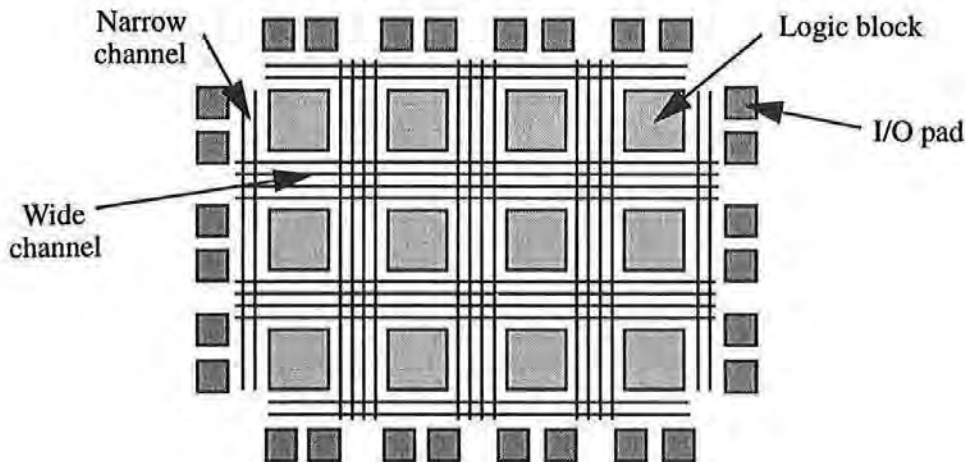


FIGURE 3.8 FPGA model assumed by VPR placer.

3.2.1 Overview of the VPR Placement Tool

In VPR, an FPGA is modelled as a set of legal slots, or discrete locations, at which logic blocks or I/O pads can be placed. In keeping with the versatility design goal, an FPGA architecture description file specifies:

- The number of logic block input and output pins,
- The number of I/O pads that fit into one row or one column of the FPGA, and
- The relative widths of the various routing channels across the FPGA.

The dimensions of the logic block array must also be set; if these are not specified on the command line, the smallest logic block array that will fit the circuit is used.

In Figure 3.8, for example, two I/O pads fit into each row or column of the FPGA, the logic block array is 4 x 3 logic blocks, and the channels between the logic block array and the I/O pads are only half as wide as those within the logic block array. Notice that perimeter I/O is assumed (i.e. wire-bonded pads, rather than flip-chip), so I/O blocks can only be placed on the edges of the FPGA.

Users can have VPR determine good locations for the I/O pads during placement, or they can lock the I/O pads in a configuration specified by an input file, or have VPR generate a random locked I/O configuration. This last capability allows the tool to measure the effect of random pin assignment on different architectures — an important issue in FPGAs [135]. Locked I/Os occur frequently in real FPGA designs, since the circuit board on which an FPGA is mounted is often fabricated before the FPGA circuit design is completed. In such cases, the placement tool must keep the circuit I/Os in the locations required by the circuit board design.

VPR uses the simulated annealing algorithm, which was described in detail in Section 2.2.2. Since the basic application of simulated annealing to placement is well known, the following sections focus on the aspects of our implementation that are improvements to the prior art.

3.2.2 New Adaptive Annealing Schedule

As described in Section 2.2.2, a good annealing schedule is essential to obtain high-quality solutions in a reasonable computation time with simulated annealing. Recall that an annealing schedule specifies the number of moves to attempt per temperature, how temperature varies throughout the anneal, and when the anneal should terminate. Our placement tool targets many different FPGA architectures, can use several different cost functions, and is used with a wide variety of circuits that span a large range of sizes. Consequently, we need an annealing schedule that automatically adapts to the current placement problem; fixed annealing schedules will not work well. We have developed a new schedule that incorporates some of the best features from the Huang et al [82], Lam and Delosme [83] and Swartz and Sechen [84] schedules, while using a new temperature update scheme and exit criterion.

Three parts of this annealing schedule are taken from prior work. First, we compute the initial temperature in the same way as Huang et al [82]. Let N_{blocks} be the total number of logic blocks plus the number of I/O pads in a circuit. We first create a random placement of the circuit. Next, we perform N_{blocks} moves (pairwise swaps) of logic blocks or I/O pads, and compute the standard deviation of the cost of these N_{blocks} different configurations. The initial temperature is set to 20 times this standard deviation, ensuring that virtually every move is accepted at the start of the anneal.

The second feature of our annealing schedule taken from prior work is the number of new placement configurations evaluated at each temperature. As in [84], we set the number of moves per temperature to

$$\text{MovesPerTemperature} = \text{InnerNum} \cdot (N_{\text{blocks}})^{4/3} \quad (3.9)$$

where the default value of InnerNum is 10. This default number can be overridden on the command line, however, to allow different CPU time / placement quality trade-offs. Reducing the number of moves per temperature by a factor of 10, for example, speeds up the placer by a factor of 10 and reduces the final placement quality by less than 10%.

It was shown in [83, 84] that it is desirable to keep the fraction of moves accepted, α , near 0.44 for as long as possible. We accomplish this in the same way as [83], by using the value of α to control a range limiter — only interchanges of blocks that are less than or equal to R_{limit} units apart in the x and y directions are attempted. A small value of R_{limit} increases α by ensuring that only blocks which are close together are considered for swapping. These “local swaps” tend to result in relatively small changes in the placement cost, increasing their probability of acceptance. Initially, R_{limit} is set to the span of the entire chip. Whenever the temperature is reduced, the value of R_{limit} is updated according to the value of α measured at the old temperature:

$$R_{\text{limit}}^{\text{new}} = R_{\text{limit}}^{\text{old}} \cdot (1 - 0.44 + \alpha) \quad (3.10)$$

and then clamped to the range $1 \leq R_{\text{limit}} \leq \text{maximum FPGA dimension}$. This results in R_{limit} being the size of the entire chip for the first part of the anneal, shrinking gradually during the middle stages of the anneal, and finally being 1 logic block at low temperatures.

The key difference between our new annealing schedule and previous schedules lies in our method of updating the temperature as the anneal progresses. When the temperature is so high that almost any move is accepted, we are essentially moving randomly from one placement to another and little improvement in cost is obtained. Conversely, if very few moves are being accepted (because the temperature is very low and the current placement is of fairly high quality), there is also little improvement in cost. With this motivation in mind, we created a temperature update scheme that increases the amount of time spent at the most productive temperatures — those where a significant fraction of, but not all, moves are being accepted. We use the fraction of moves being accepted, α , to directly control how quickly the temperature drops. A new temperature is computed as $T_{\text{new}} = \gamma \cdot T_{\text{old}}$, where the value of γ depends on the fraction of attempted moves that were accepted (α) at T_{old} , as shown in Table 3.2. The exact values of α and γ listed in Table 3.2 were found via experimentation. While the values in Table 3.2 led to the best performance, the performance of the annealer is not extremely sensitive to the exact value of γ as a function of α . So long as the function $\gamma(\alpha)$ has the right form — γ is near 1 for α around 0.44, and γ is significantly smaller for α near 1 or 0 — the annealer performs reasonably well.

We terminate the anneal when:

$$T < \epsilon \cdot \frac{\text{Cost}}{N_{\text{nets}}} , \quad (3.11)$$

TABLE 3.2 Temperature update schedule.

α	γ
$\alpha > 0.96$	0.5
$0.8 < \alpha \leq 0.96$	0.9
$0.15 < \alpha \leq 0.8$	0.95
$\alpha \leq 0.15$	0.8

where N_{nets} is the number of nets in the circuit, and we use an ϵ value of 0.005. The movement of a logic block will always affect at least one net. When the temperature is less than a small fraction of the average cost of a net, it is unlikely that any move that results in a cost increase will be accepted, so we terminate the anneal. Again, the performance of the annealer is not terribly sensitive to the ϵ factor in (3.11). Any value between 0.05 and 0.005 is reasonable, with smaller values giving slightly higher quality placements at the cost of slightly increased CPU time.

The annealing schedule described above has produced excellent results with a wide variety of cost functions, FPGA architectures, circuits, and moves per temperature (i.e. desired quality) values. For comparison purposes, we also implemented an annealing schedule based on that of Huang et al [82]¹. This annealing schedule was not sufficiently robust for our purposes, particularly with large circuits. For some circuits its temperature update scheme became too conservative, and the temperature decreased extremely slowly. For some other circuits its exit criterion did not function correctly, and a large amount of CPU time was wasted at very low temperatures with no significant quality improvement. Table 3.3 compares the two annealing schedules on a set of eight MCNC benchmark circuits [136]. On the six smaller benchmarks the two schedules were fairly comparable — our schedule was 1.35x faster, but resulted in placements with 0.9% higher cost. The Huang schedule broke down on the two larger circuits, however; it took 3.8x more CPU time on alu4, and did not complete after 40x as much CPU as our schedule required on bigkey. We want to be able to control the quality-time trade-off of our tools, so an annealing schedule that is unpredictable, i.e. spends huge amounts of time on some circuits, is not suitable for our purposes.

1. This annealing schedule used the initial temperature, exit criterion and temperature update scheme of [82]. It used the number of moves per temperature value from [84] (equation (3.10)), however, as this provides a parameter for controlling the CPU time / quality trade-off.

TABLE 3.3 Comparison of new annealing schedule to schedule of Huang et al.

Circuit	N_{blocks}	Schedule of Huang et al vs. new schedule	
		CPU ratio	Final cost difference
s1423	338	1.03x	+0.6%
e64	466	1.16x	-0.8%
table3	560	1.30x	-1.6%
sbc	561	1.24x	-0.4%
x3	648	1.40x	-2.4%
C5315	1200	1.99x	-0.7%
bigkey	1914	terminated after 40x	-5.3%
alu4	2319	3.82x	-1.5%
7 Circuit Average	870.3	1.71x	-1.0%
8 Circuit Average	1000.8	6.5x	-1.5%

We have not implemented the schedules of Lam and Delosme [83] or Swartz and Sechen [84], so we cannot compare our performance to theirs. Note however, that our schedule is much simpler than that of Lam. It is also completely adaptive (all parameters adapt to the problem at hand), while the Swartz schedule uses a hard-coded range-limiter variation and number of temperatures to visit.

3.2.3 New Cost Function: Linear Congestion

One of the FPGA architectural issues we investigate is global routing architecture. Recall that many global routing architectures have wider channels in some regions of the FPGA than in others; see Figure 1.1 for an example. To fully optimize for such architectures, those portions of a circuit that require more routing should be placed in regions of the FPGA that have wider routing channels. The key to obtaining such global-routing-architecture-aware placements is ensuring that the cost function used properly models the relative difficulty of routing connections in areas with different channel widths. Accordingly, we developed what we call a *linear congestion* cost function. Of all the alternatives we have explored, this cost function provides the best results in a reasonable computation time. Its functional form is

$$Cost_{linear\ congestion} = \sum_{i=1}^{N_{nets}} q(i) \left[\frac{bb_x(i)}{C_{av,x}(i)^\beta} + \frac{bb_y(i)}{C_{av,y}(i)^\beta} \right], \quad (3.12)$$

where the summation is over the N_{nets} in the circuit. For each net, i , $bb_x(i)$ and $bb_y(i)$ denote the horizontal and vertical spans of its bounding box, respectively. The $q(i)$ factor compensates for the fact that the bounding box wire length model underestimates the wiring necessary to connect nets with more than three terminals [89]. Its value depends on the number of terminals of net i . We obtained the appropriate values of $q(i)$ from [89]; $q(i)$ is 1 for nets with 3 or fewer terminals, and slowly increases to 2.79 for nets with 50 terminals. $C_{av,x}(i)$ and $C_{av,y}(i)$ are the average channel capacities (in tracks) in the x and y directions, respectively, over the bounding box of net i .

This cost function penalizes placements which require more routing in areas of the FPGA that have narrower channels. The exponent, β , in the cost function allows the relative cost of using narrow and wide channels to be adjusted. When β is zero the linear congestion cost function reverts to the standard bounding box cost function. The larger the value of β , the more wiring in narrow channels is penalized relative to wiring in wider channels; we have experimentally found that setting β to one results in the highest quality placements.

C_{av} depends only on the channel capacities, which do not change during a placement, and on the maximum and minimum coordinates of a bounding box. We therefore precompute all possible $C_{av,x}$ and $C_{av,y}$ values and store them in a two-dimensional array indexed by the bounding box minimum and maximum coordinates. Consequently, recomputing this cost function is essentially as fast as recomputing the traditional bounding box cost function.

In an FPGA where all channels have the same capacity, C_{av} is also a constant and hence the linear congestion cost function reduces to a bounding box cost function. In FPGAs where some channels are wider than others, however, this cost function results in higher quality placements than a bounding box cost function. The exact amount of routability improvement depends on the precise global routing architecture used; as one would expect, those in which there is a large difference between the widths of channels in different regions show the largest improvement. For the architectures studied in Chapter 5, placements produced with the linear congestion cost function typically require 5 to 10% fewer tracks to route than placements produced with a bounding box cost function.

For comparison, we also implemented the cost function of [89], which we call a *non-linear congestion* cost function. We described this cost function in Section 2.2.2. Recall that it divides the FPGA into an $M \times M$ array of regions and attempts to model

the routing resource demand and supply in each of these regions, penalizing placements that appear to result in routing demand exceeding supply in some regions.

Table 3.4 shows how the linear and nonlinear congestion cost functions compare to a bounding box cost function on one FPGA architecture. In this architecture, the 50% of the FPGA channels nearest the center are twice as wide as the other channels. 36 MCNC [136] benchmarks were placed and globally routed in the smallest FPGA that could accommodate each. Table 3.4 shows how the number of tracks in the FPGA required to complete global routing and the placement CPU time is affected by the cost function used. Note that 10 of the 36 circuits were I/O bound in the FPGA assumed, so separate columns are provided for the average over these 10 circuits, the average over the 26 logic-bound circuits, and the overall average. The routability improvements with the more advanced cost functions are larger for the I/O bound circuits, since there are “spare” logic blocks in the FPGAs in these cases, giving the placement tool the option of spreading out the logic to improve routability.

TABLE 3.4 Cost function performance comparison on an example FPGA.

Cost function	Average Tracks Required for Global Routing			CPU Time
	All 36 circuits	26 logic-bound	10 I/O bound	
Bounding box	1x	1x	1x	1x
Linear congestion	-7%	-4%	-16%	1x
Nonlinear cong. (16 regions)	-8%	-5%	-16%	5x
Nonlinear cong. (256 regions)	-10%	-6%	-22%	80x

Notice that the non-linear congestion cost function, when computed on a 4 x 4 grid (16 regions), produced placements which required, on average, only 1% fewer tracks to route than those produced by the linear congestion cost function. However, keeping track of the routing resource demand in the various chip regions is computationally expensive, and placement with this cost function requires five times greater CPU time than the linear congestion function. Dividing the FPGA into smaller subregions to make localized congestion more visible improved the performance of this cost function slightly; a non-linear congestion cost function computed on a 16 x 16 grid (256 regions) performs 3% better than our linear congestion cost function on this architecture. However, it consumes 80 times the CPU time. We consider the routability improvements gained by the nonlinear congestion cost function too small to warrant the huge increase in CPU time, and so prefer the linear congestion cost function developed in this work.

3.2.4 Incremental Net Bounding Box Updates

Even with a good annealing schedule, millions of potential block swaps will be evaluated in a typical placement run. The most computationally expensive part of evaluating a swap is computing the change in cost, ΔC , the swap would produce; it is crucial that this computation be made as fast as possible.

Consider the computation of ΔC caused by the swap of two blocks. The only terms in the summation of (3.12) that change are those corresponding to the nets attached to the two swapped blocks. The bounding boxes of all the nets attached to either of these two blocks must be recomputed, and then (3.12) can be used to determine ΔC . The recomputation of the net bounding boxes is the key step here; unless care is taken, it can dramatically slow the placer.

The straightforward way to re-evaluate a net's bounding box is to examine the location of each of its terminals. Unfortunately, this is an $O(k)$ operation for a k -terminal net. Large circuits typically have many high-fanout nets, a few of which have hundreds of terminals. As well, since high-fanout nets have terminals on so many blocks, swapping any two blocks has a high probability of disturbing some high-fanout nets.

We have developed an alternative to this brute-force computation, which we call *incremental bounding box evaluation*. For each net, we store the coordinate of each of the four sides of its bounding box ($x_{\min}$, $x_{\max}$, $y_{\min}$, $y_{\max}$), and the number of net terminals that lie on each of these sides ($N_{x_{\min}}$, $N_{x_{\max}}$, $N_{y_{\min}}$, $N_{y_{\max}}$). Figure 3.9 shows an example of this data storage.

Now say that some terminal of this net is moved via a swap from $(x_{\text{old}}, y_{\text{old}})$ to $(x_{\text{new}}, y_{\text{new}})$. Since we have stored the extra information shown in Figure 3.9, we can usually determine the new net bounding box by looking only at the terminal which moved, rather than all k terminals. Figure 3.10 lists the pseudo-code used to update the $x_{\min}$ and $N_{x_{\min}}$ values for a net i ; the code for the other four sides is similar.

Notice that there is only one case for which the net bounding box must be recomputed by the brute-force procedure: when the terminal moved is the only net terminal on a side of the bounding box, and it is moved inward, toward the bounding box center. In this case the recomputation is $O(k)$, while in all other cases it is $O(1)$. The probability of an arbitrary net terminal being on some side of the bounding box, and being the only terminal on that side of the bounding box is $\propto 1/k$, however. Hence the average net bounding box recomputation is $O\left(1 + \frac{1}{k} \cdot k\right) = O(1)$.

We have experimentally determined that our incremental bounding box update method is faster than the brute-force method for all nets with more than 4 terminals.

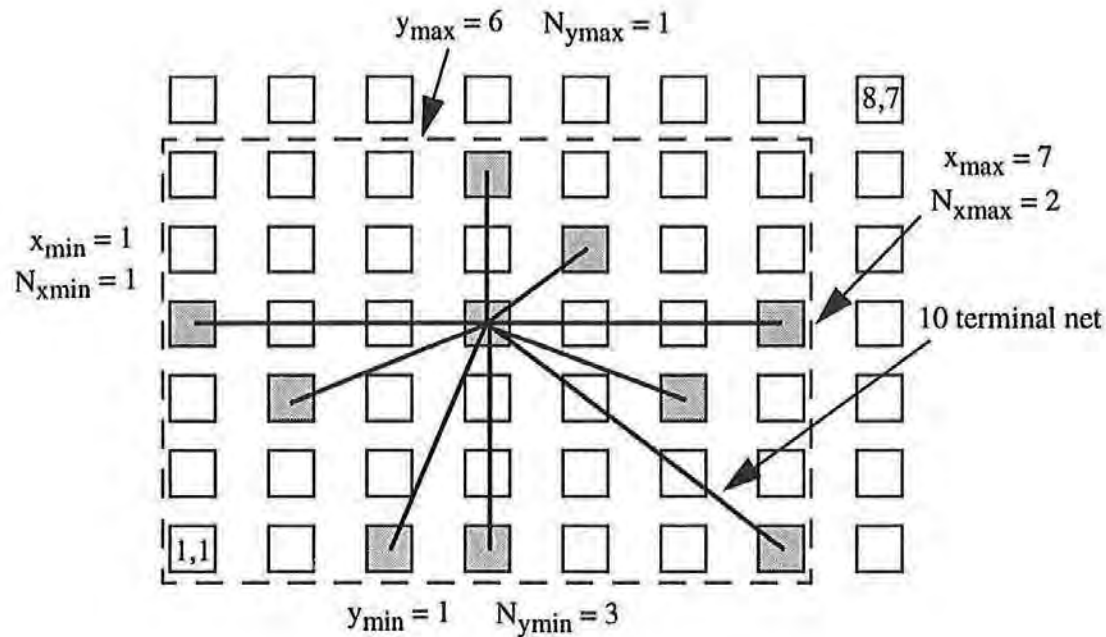


FIGURE 3.9 Data stored to enable incremental bounding box updates.

```

if ( $x_{new} \neq x_{old}$ ) { /* Terminal has moved horizontally */
    if ( $x_{new} < x_{min}(i)$ ) { /* Terminal moved left past old  $x_{min}$  edge */
         $x_{min}(i) = x_{new}$ ;
         $N_{xmin}(i) = 1$ ;
    }

    else if ( $x_{new} == x_{min}(i)$ ) { /* Terminal moved left to lie on the old  $x_{min}$  edge */
         $N_{xmin}++$ ;
    }

    else if ( $x_{old} == x_{min}(i)$ ) { /* Terminal was on  $x_{min}$  edge; moved right */
        if ( $N_{xmin} > 1$ ) { /* Still terminals on  $x_{min}$  edge? */
             $N_{xmin}--$ ;
        }
        else {
            BruteForceBoundingBoxRecompute ( $i$ );
        }
    }
}

```

FIGURE 3.10 Pseudo-code to update the bounding box of net i incrementally.

Using this more sophisticated technique for nets with more than 4 terminals yields, on average, a more than five times speedup in the placer. Table 3.5 compares the CPU time on a 300 MHz UltraSparc needed to place the ten largest MCNC benchmark circuits with and without incremental bounding box recalculation. In this experiment each logic block is a BLE (4-LUT / FF pair). The speedup due to incremental bounding box updates ranges from 2.52 times to 9.41 times, with an average speedup of 5.39 times. The variation in speedup is due to the different fanout distributions of these circuits — circuits with a higher average fanout benefit more. Since placement is so time-consuming, and there is considerable need for fast CAD tools as FPGA sizes increase [127], this speedup is very important. Another useful feature of the incremental bounding box code is that it makes the CPU time required to place a circuit highly predictable from the circuit size; this allows a CAD tool to give a user an accurate estimate of the time required to place a circuit.

TABLE 3.5 Placement CPU time vs. bounding box recalculation method.

Circuit	# Logic Blocks	CPU (s) without incremental bounding box	CPU (s) with incremental bounding box	Speedup
apex2	1878	116	46	2.52x
s298	1931	386	41	9.41x
frisc	3556	599	127	4.72x
elliptic	3604	864	125	6.91x
spla	3690	434	125	3.47x
pdc	4575	664	172	3.86x
ex1010	4598	706	183	3.86x
s38417	6406	1405	315	4.46x
s38584.1	6447	2815	340	8.28x
clma	8383	3207	499	6.43x
Arithmetic Average:	4507	1120	197	5.39x
Geometric Average:	4071	755	151	4.99x

The InnerNum value for these results was set to 1; to obtain the highest quality (5 - 10% better) results, InnerNum is set to ten, and therefore ten times more CPU is required than Table 3.5 lists. Hence one can see why speeding up placement is important!

3.3 Summary

In this chapter we have described our logic block packing tools, VPack and T-VPack, and the placement portion of the VPR placement and routing tool. VPack and T-VPack are the first publicly-described logic block packing tools targeting cluster-based logic blocks. VPR incorporates three enhancements over prior simulated-annealing placement algorithms: a new annealing schedule, a routability-enhancing linear congestion cost function, and an incremental net bounding box update method that reduces placement CPU time by a factor of over 5. The next chapter continues the description of the CAD tools developed for this book by describing the VPR router. At the end of the next chapter, the performance of the VPR placement and routing tool will be compared to that of prior CAD tools, allowing us to evaluate the result quality of the placement tool described in this chapter.

Routing Tools and Routing Architecture Generation

In this chapter we describe how the routing portion of VPR works. We begin by describing the spectrum of FPGA architectures that the router has targeted, and the understandable architecture parameters used to describe an FPGA to VPR. We then explain how a routing architecture is represented internally, and how the succinct description provided by a user is *automatically* turned into this highly detailed architecture representation. Next, we describe the two routers built into VPR; one is purely routability-driven, while the other is both timing- and routability-driven. The timing-driven router requires a fast and accurate net delay extractor and a path-based timing analyzer, both of which are also discussed. Finally, we compare the performance of VPR to that of several other published CAD tools, and show that it outperforms all the tools to which we have been able to compare.

4.1 Position within the CAD flow

Figure 4.1 shows where the VPR router fits into the CAD flow. Its input is a netlist of logic blocks and an architecture file which describes the target FPGA. A placement is either read in or the VPR placer is used to place the circuit, and then either the routability-driven or the timing-driven router built into VPR is invoked. Each of these routers can perform either a combined global-detailed routing, or global routing only. Once routing is complete, various statistics, such as routed wirelength, routing resource utilization and the delay of the critical path are output.

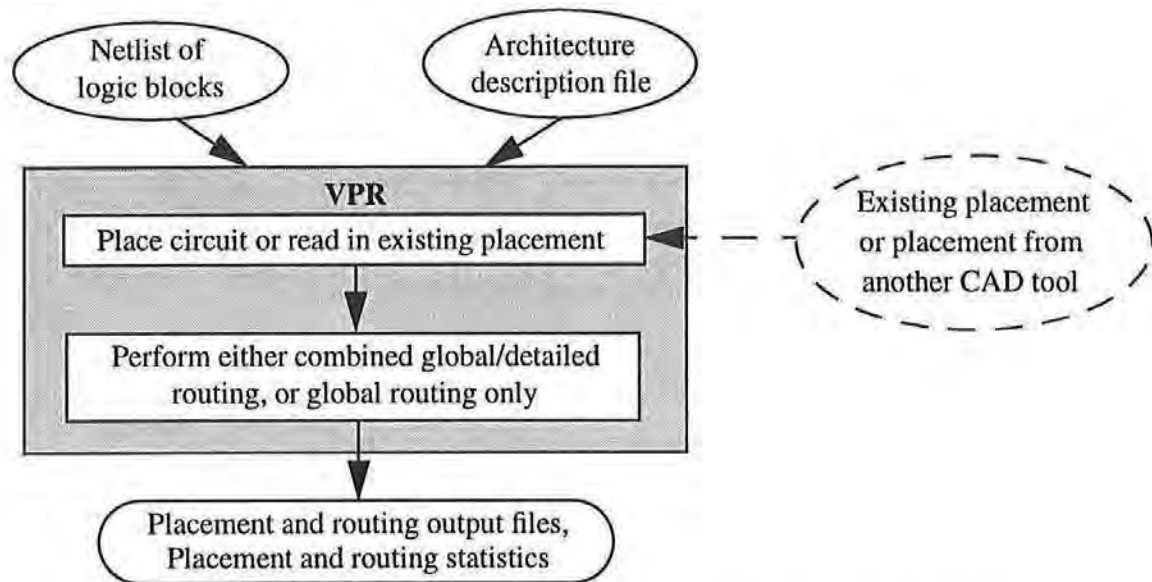


FIGURE 4.1 VPR router operation and position within the CAD flow.

4.2 Architecture Parameterization and Generation

Since our primary goal in this research is to investigate many different FPGA architectures, we would like our tools to:

1. Be as architecture-independent as possible, and
2. Be *easy* to use with different architectures — that is, we would like to be able to describe different architectures to VPR quickly.

To achieve the first goal, we have made the router, graphics routines, delay extractor, and various other routines all operate on a directed graph that describes the FPGA. This *routing-resource graph* representation is very general, and can describe a wide variety of FPGA architectures. Unfortunately, describing a new architecture by creating a routing-resource graph by hand is not feasible — the routing-resource graph to describe a typical FPGA containing 8000 4-LUTs is almost 30 MB in size. One possibility is to design a basic tile (a single logic block and its associated routing) manually, and create a program to automatically replicate and stitch together this tile into a routing-resource graph describing the entire FPGA. Even creating a basic tile manually is too time-consuming for our purposes, however. A typical tile contains several hundred programmable switches and wires, so it can take hours or days to describe even one tile. Furthermore, such a hand-crafted tile is designed for one value of routing channel width, W . In many of our experiments we wish to vary W in order to see

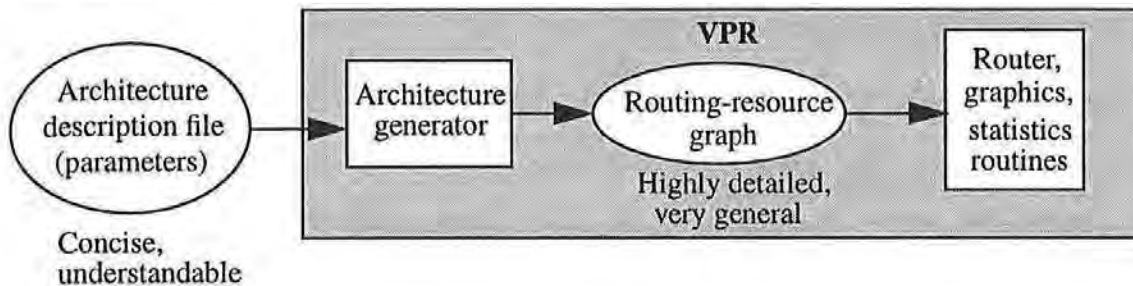


FIGURE 4.2 A concise architecture description is converted to a detailed graph description.

how routable a given FPGA architecture is, or how well our CAD tools optimize for routability. With a tile-based approach, we would have to hand-craft one tile for each different value of W , for each architecture. Since we wish to investigate hundreds of different FPGA architectures, and tens of W values for each of these architectures, we would have to create thousands or tens of thousands of these basic tiles.

In order to satisfy the ease of use goal, therefore, we have designed VPR so that it takes a concise, human-comprehensible architecture definition file, and uses an internal graph generator to create the highly detailed routing-resource graph representation which the router and other CAD routines use. Figure 4.2 illustrates the basic idea. If one wants to look at a completely new class of FPGAs that does not fit into the current format of the architecture description file, then, only the routing graph generator has to be modified; the router, graphics, timing analyzer, and statistics routines will all function correctly.

4.2.1 Architecture Parameterization

We want our architecture description file to be easy to create, so we tried to parameterize architectures in ways that are intuitive to FPGA researchers. By parameterizing our architectures we also make it easier to describe our results to other researchers, and to understand why one architecture is better than another. (Simply showing that one 30 MB routing-resource graph was superior to most others would not allow us to describe our results to others very easily!)

To perform routing we need more information about the architecture than is required for placement only. The architecture description file specifies:

- The number of logic block input and output pins,
- The side(s) of the logic block from which each input and output is accessible,

- The logical equivalence between the various input and output pins (e.g. all LUT inputs are functionally equivalent),
- The number of I/O pads that fit into one row or column of the FPGA,
- The relative widths of the horizontal and vertical channels, and
- The relative widths (number of tracks) of the channels in different regions of the FPGA.

If only global routing is desired, these are all the parameters needed. If combined global-detailed routing is desired, however, the architecture file must also specify:

- The switch block topology used to connect the routing tracks (i.e. which tracks connect to which at a switch block),
- The number of tracks to which each logic block input pin connects, $F_{c,input}$,
- The F_c value for logic block outputs, $F_{c,output}$,
- The F_c value for I/O pads, $F_{c,pad}$, and
- One or more *wire segment types*. For each segment type, one specifies:
 - The fraction of tracks in a channel that are of this segment type,
 - The segment length (number of logic blocks spanned by a wire segment),
 - The type of switch (pass-transistor or tri-state buffer, drive strength, etc.) used to connect a wire segment of this type to other routing segments,
 - The switch-block internal population of this segment type (discussed below), and
 - The connection-block internal population of this segment type (discussed below).

Note that the segmentation distribution (the fraction of routing tracks of each length), is specified as part of the wire type definitions.

We define internal population (described in Section 2.1.3) in a more general way than prior researchers. Recall that internal population describes whether or not logic blocks and routing wires can connect to the interior of a wire segment, or if connections to a wire can be made only at its ends. In [21], a wire segment is either completely internally populated or completely depopulated. We allow *partial depopulation* of the interior of a wire segment. For example, a length five segment spans five logic blocks. If we specify a connection-block population of 100%, this wire segment can connect to all five logic blocks it passes, so it is fully internally populated. If the connection-block population is 40%, it can only connect to the two logic blocks at its ends, so it is internally depopulated. If we specify a connection-block population of 60%, however, the wire can connect to the two logic blocks at its

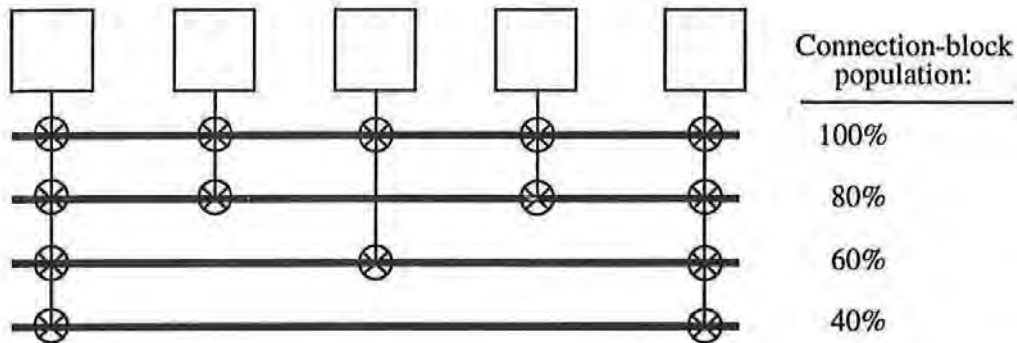


FIGURE 4.3 Possible connection-block population values for length 5 wire segments.

ends and one logic block in its interior, so it is partially internally depopulated. Figure 4.3 illustrates the four possible values of connection-block population for a length five wire. Switch-block population is specified in a similar, percentage, form.

Notice that we specify the distribution of wire types as fractions of the channel width, W , rather than as an absolute number of tracks of each type. For example, one might say there are 20% length = 2 wires and 80% length = 5 wires. This allows a user to attempt routings with different W values, to determine the routability of an architecture, without changing the architecture file. Similarly, the various F_c values can be specified either as absolute numbers (e.g. 5 tracks), or as a fraction of the tracks in a channel (e.g. $0.2 \cdot W$).

The number of tracks per channel, W , and the size of the logic block array size can be specified on the command line. If one or more of these parameters is not specified, VPR will determine the minimum value(s) needed to fit the circuit in the specified FPGA architecture.

Finally, to allow extraction of the delay of routed nets and path-based timing-analysis, one must specify various timing parameters in the architecture description file. These include:

- The input and output capacitance, equivalent resistance, and intrinsic delay of each type of switch used in the routing; as many switch types as desired can be defined.
- The capacitance and resistance of each type of wire segment,
- The delays of all the combinational and sequential elements within each logic block, and
- The delays of the I/O pads.

4.2.2 The Routing-Resource Graph

While the architecture parameters listed above are easy for FPGA architects to understand and specify, they are not appropriate for use as an internal architecture representation for a router. Internally, VPR uses a routing-resource graph [85] to describe the FPGA; this is more general than any parameterization, since it can specify arbitrary connectivity. It also makes it much faster to determine connectivity information, such as the wires to which a given wire segment can connect, since this information is explicitly contained in the graph.

Each wire and each logic block pin becomes a node in this routing-resource graph and each switch becomes a directed edge (for unidirectional switches, such as buffers) or a pair of directed edges (for bidirectional switches, such as pass transistors) between the two appropriate nodes. Figure 4.4 shows the routing-resource graph corresponding to a portion of an FPGA whose logic block contains a single 2-input, 1-output LUT.

Often FPGAs have logically equivalent pins; for example, all the input pins to a LUT are logically equivalent. This means that a router can complete a given connection using any one of the input pins of a LUT; changing the values stored in the LUT can compensate for any re-ordering of which connection connects to which input pin performed by the router. We model this logical equivalence in the routing-resource graph by adding *source* nodes at which all nets begin, and *sink* nodes at which all net terminals end. There is one source node for each set of logically-equivalent output pins, and there is an edge from the source to each of these output pins. Similarly, there is one sink node for each set of logically-equivalent input pins, and an edge from each of these input pins to the sink node.

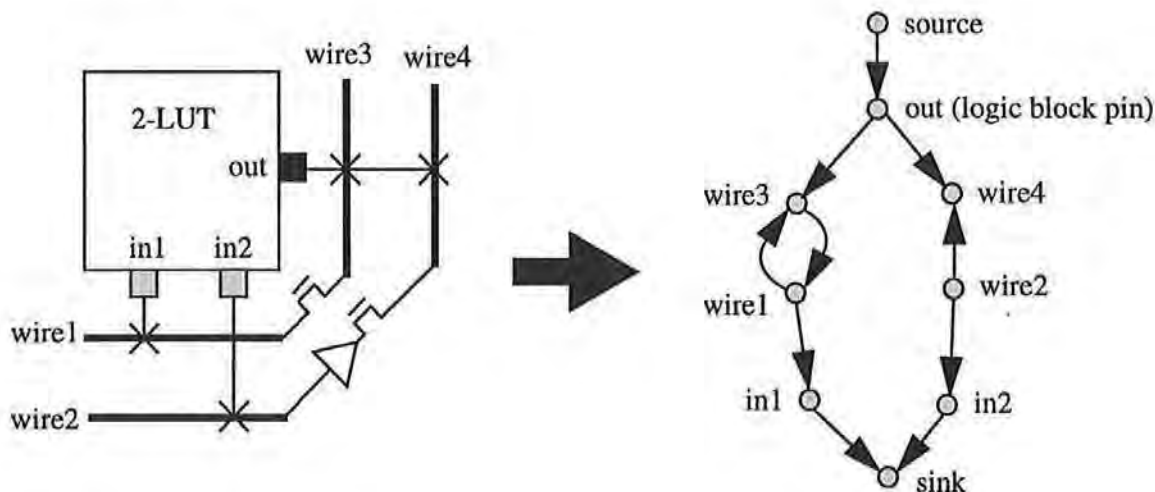


FIGURE 4.4 Modelling FPGA routing architecture as a directed graph.

To reduce the number of nodes in the routing-resource graph, and hence save memory, we assign a capacity to each node. A node's capacity is the maximum number of different nets which can use this node in a legal routing. Wire segments and logic block pins have capacity one, since only one net may use each. Sinks and sources can have larger capacities. For example, in a 4-input LUT, there is one group of four logically-equivalent inputs, so we have one sink of capacity four. If we could not assign a capacity of four to the sink, we would be forced to create four logically-equivalent sinks and connect them to the four input pins via a complete bipartite graph ($K_{4,4}$), wasting considerable memory.

By constructing an appropriate routing-resource graph we can perform global routing, instead of combined global-detailed routing, with no changes to our router code. Figure 4.5 shows a routing-resource graph for global routing. Again, each logic block pin becomes a node, and source and sink nodes are added for each logically-equiva-

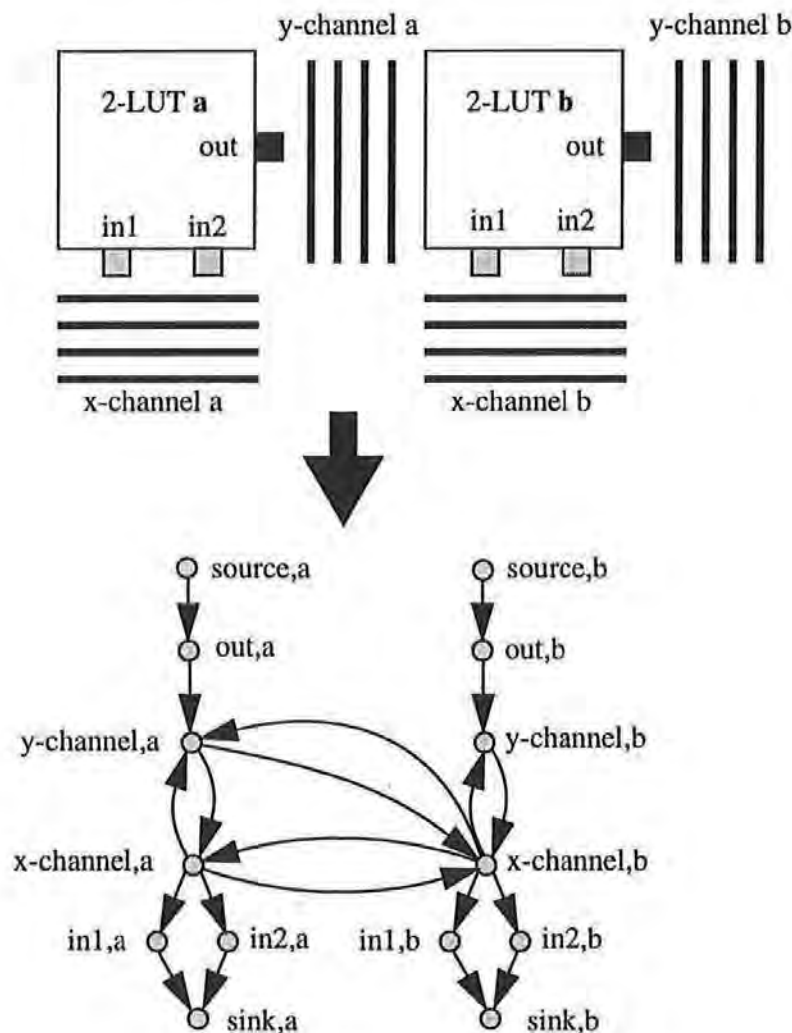


FIGURE 4.5 Modelling the FPGA global routing problem via a directed graph.

lent group of pins. Now, however, instead of each wire becoming a node, each channel segment (the length of channel that spans one logic block) becomes a routing-resource node. The capacity of each of these nodes is equal to the number of wires, or tracks, in the corresponding channel segment; in Figure 4.5, for example, the capacity of each channel segment node is 4. Edges are added to connect logic block input and output pins to the adjacent channel segments, and to connect adjacent channel segments together.

To perform timing-driven routing, timing analysis, and to graphically display the architecture (see Appendix A), we need more information than just the raw connectivity embodied in the nodes and edges of the routing-resource graph. Accordingly, we annotate each node in the graph with its type (wire, input pin, etc.), location in the FPGA array, capacitance and metal resistance. Each edge in the graph is marked with the index of its “switch type,” allowing retrieval of information about the switch intrinsic delay, equivalent resistance, input and output capacitance and whether the switch is a pass transistor or tri-state buffer.

4.2.3 Automatic Architecture Generation from Parameters

As the previous two sections have described, there are compelling reasons to allow designers to specify architectures in an understandable, parameterized format, and for the routing tools to work with a more detailed, graph-based, description. As Figure 4.2 shows, then, VPR must be capable of automatically generating a routing-resource graph from a set of specified architecture parameters. Generating an architecture automatically is a difficult problem for two reasons:

1. We want to create a *good* architecture with the specified parameters. That is, the unspecified properties of the architecture should be set to “reasonable” values.
2. Simultaneously satisfying all the parameters defining the architecture is difficult. In some cases, the specified parameters conflict and overspecify the FPGA, making it impossible to simultaneously satisfy all the specified constraints.

Consider the first problem. If we require a user to specify every conceivable parameter, and every interaction between these parameters, describing an architecture will be very time-consuming. Instead, we want to allow users to specify the important parameters, and have the VPR tool automatically adjust other parameters of the architecture so that a good FPGA results. For example, we require that a user specify the number of tracks to which input and output pins can connect, $F_{c,input}$ and $F_{c,output}$, rather than requiring a user to specify the complete connection block switch pattern. This certainly simplifies the task of describing an FPGA, but it means that VPR must generate a good connection block switch pattern automatically.

Let us consider this connection block problem in more detail. We decided that the switch pattern chosen should:

- Ensure that each of the W tracks in a channel can be connected to roughly the same number of input pins, and roughly the same number of output pins,
- Ensure that each pin can connect to a mix of different wire types (e.g. different length wires),
- Ensure that pins that appear on multiple sides of the logic block connect to different tracks on each side, to allow more routing options,
- Ensure that logically-equivalent pins connect to different tracks, again to allow more routing options, and
- Ensure that pathological switch topologies in which it is impossible to route from certain output pins to certain input pins (see Figure 4.6) do not occur.

Clearly this is a complex problem. In essence, the proper connection block pattern is a function of $F_{c,input}$, $F_{c,output}$, W , the segmentation distribution, the logical equivalence between pins, and the side(s) of a logic block from which each pin is accessible. The last condition is also a function of the switch block topology. VPR attempts to build a connection block that satisfies the five criteria above, but it will not necessarily perfectly satisfy them all for all architectures.

The second difficulty in generating an architecture automatically is simultaneously meeting all the user-defined specifications. We will illustrate this difficulty with an example that shows it often takes considerable thought to simultaneously satisfy the specifications. Consider an architecture in which:

- Each channel is three tracks wide.
- Each wire is of length 3.

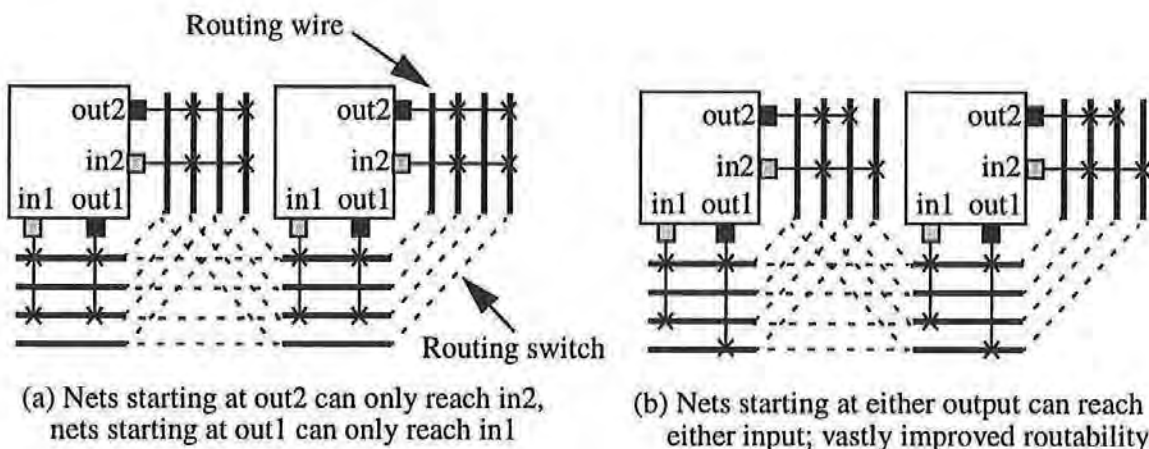


FIGURE 4.6 Example connection block patterns: (a) pathologically bad; (b) good.

- Each wire has an internal switch block population of 50%. That is, routing switches can connect only to the ends of a wire segment (2 of the 4 possible switch block locations).
- The switch block topology is *disjoint* [101]. In this switch block, wires in track 1 always connect only to other wires in track 1, and so on. This is the switch block topology used in the original Xilinx 4000 FPGAs [137].

Figure 4.7 shows the disjoint switch block topology, and a channel containing 3 wires of length 3. Notice that the “start points” of the wire segments are staggered [37]. This enhances routability, since each logic block in the FPGA can then reach a logic block two units away in either direction using only one wire segment. It also arises naturally in a tile-based layout, so staggering the start points of the segments in this way makes it easier to lay out the FPGA. A tile-based FPGA layout is one in which only a single logic block and its associated routing (one vertical channel segment and one horizontal channel segment) have to be laid out — the entire FPGA is created by replication of this basic tile.

The most straightforward way to create an FPGA with this architecture is to create one horizontal channel and one vertical channel, and replicate them across the array. Switches are then inserted between horizontal and vertical wire segments which the switch block and internal population parameters indicate should be connected. Figure 4.8 shows the results of such a technique, where only a few of the routing switches have been shown for clarity. Notice that this FPGA *does not* meet the specifications. By inserting routing switches at the ends of the horizontal segments, we are allowing connections into the middle of vertical segments. However, our specifications said that segments should have routing switches only at their ends. If we do not insert switches at the ends of the horizontal segments, however, we cannot connect to the ends of the horizontal segments, so the specifications are again violated. We call this problem a conflict between the *horizontal constraints* and the *vertical constraints*.

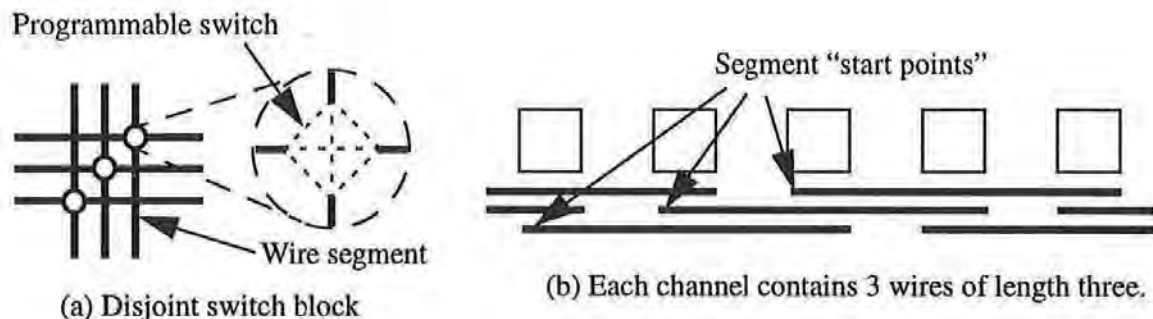


FIGURE 4.7 Architecture specification: (a) disjoint switch block; (b) segmentation distribution.

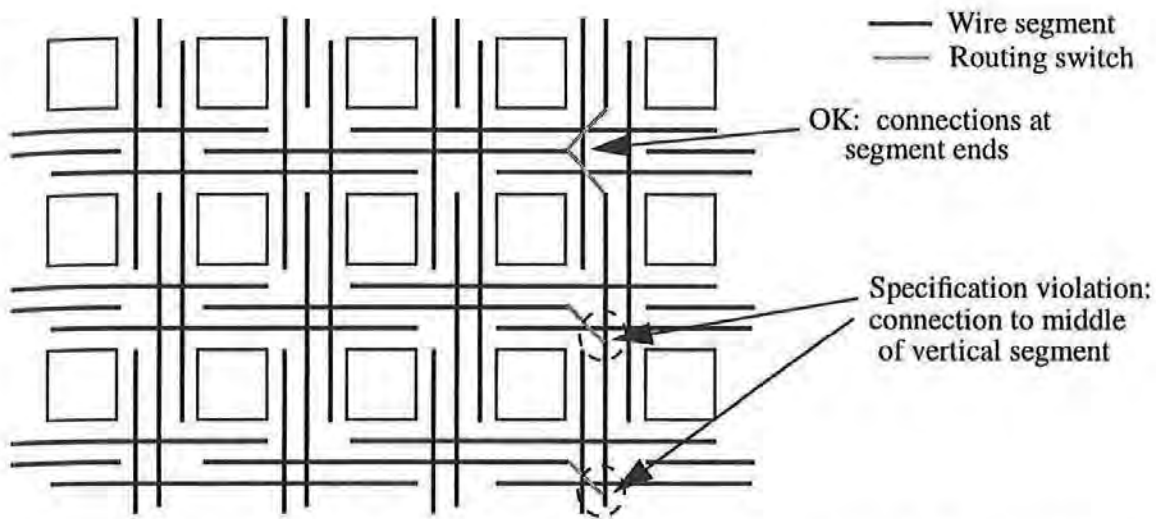


FIGURE 4.8 Replicating one channel causes the horizontal and vertical constraints to conflict.

The solution to this problem is shown in Figure 4.9. Instead of simply replicating a single channel, the “start points” of the segments in each channel have to be adjusted. As Figure 4.9 shows, this allows the horizontal and vertical constraints to be simultaneously satisfied. The specification for the FPGA has been completely realized — every segment connects to others only at its ends, and the switch block topology is disjoint. Figure 4.10 shows how one can implement this architecture using a single

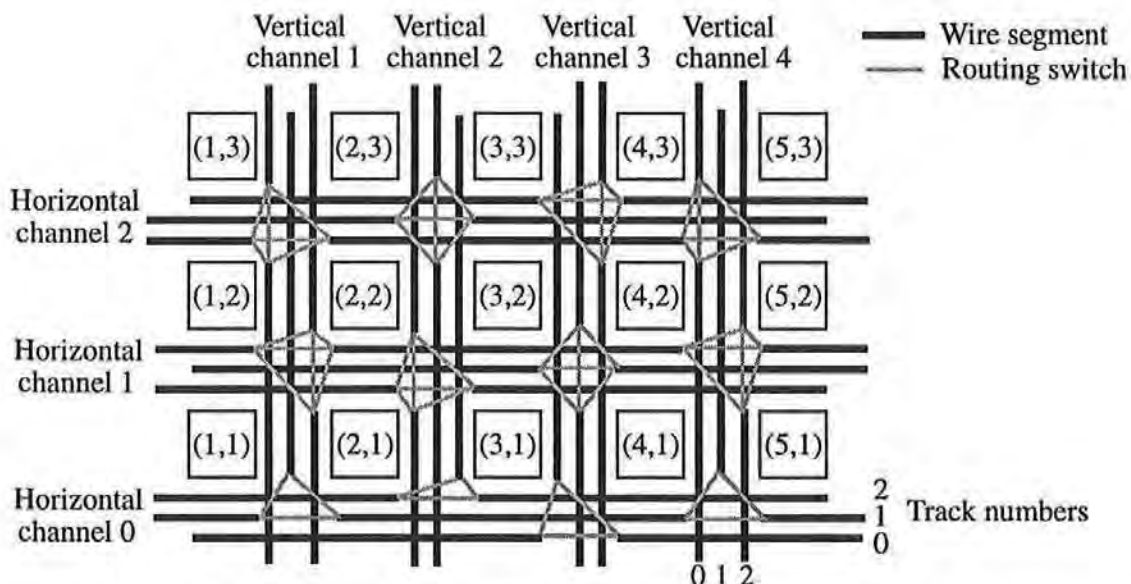


FIGURE 4.9 Adjusting the segment start points allows both the horizontal and vertical constraints to be satisfied. The FPGA coordinate system is also shown.

layout tile. This is an additional bonus of this “segment start point adjustment” technique — we not only meet our specifications fully, but create an easily laid-out FPGA.

In order to describe the adjustment of the segment start points more clearly, let us define an FPGA coordinate system. Let the logic block in the lower left corner of the logic block array have coordinates (1,1). The logic block to its right has coordinates (2,1), and the logic block above it has coordinates (1,2), as Figure 4.9 shows. A horizontal channel has the same y-coordinate as the logic block below it, and a vertical channel has the same x-coordinate as the logic block to its left. We also number the tracks within each channel from 0 to 2, with track 0 being the bottommost track in a horizontal channel, or the leftmost track in a vertical channel.

The proper adjustment shifts the start point of each segment back by 1 logic block, relative to its start point in channel j , when constructing channel $j+1$. For example, in Figure 4.9, the left ends of the wire segments in track 0, horizontal channel 0 line up with the logic blocks that satisfy

$$(i + 2) \text{ modulo } 3 = 0, \quad (4.1)$$

where i is the horizontal (x) coordinate of a logic block. In channel 1, track 0, however, the left ends of the wire segments line up with logic blocks that satisfy:

$$(i + 3) \text{ modulo } 3 = 0 \quad (4.2)$$

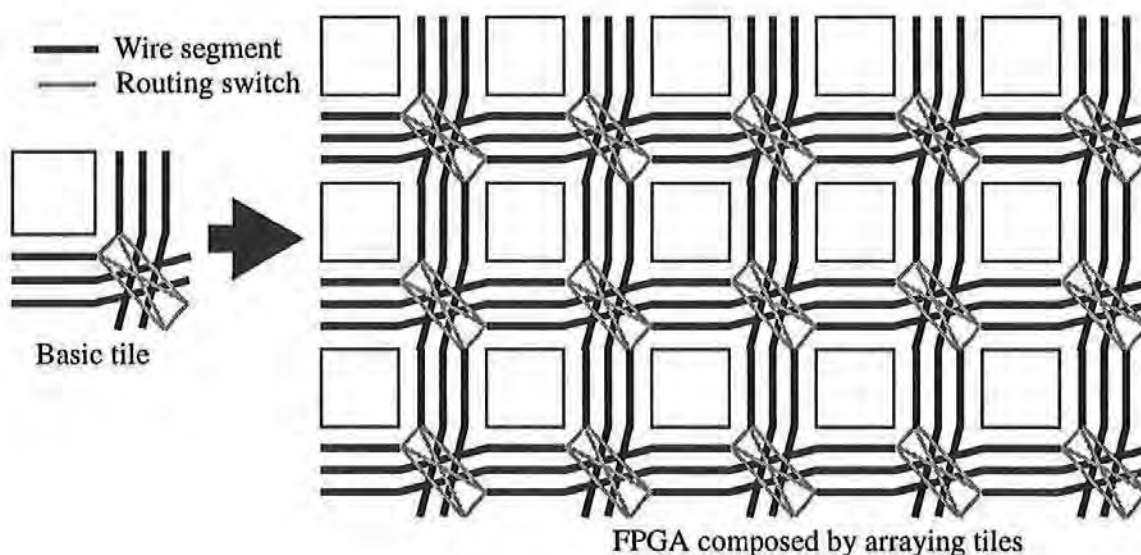


FIGURE 4.10 Tiled layout to implement FPGA of Figure 4.9.

A similar shifting back of start points must be performed in the vertical channels — the start point of segments in channel $i+1$ is moved back one logic block relative to their start point in channel i .

The shifting of segment start points above allows the horizontal and vertical constraints on an FPGA to be met if either of the following two conditions is met:

- The disjoint switch block topology is used. The segmentation distribution and segment internal populations can have any values. Or,
- All segments are fully switch-block populated. The segmentation distribution and switch block topology can have any values.

If either of these conditions is satisfied, the shifting of segment start points also makes a tile-based layout possible if one additional constraint is satisfied: the number of tracks of length L is divisible by L , for all segment lengths L .

We have not yet found a method to simultaneously satisfy the horizontal and vertical constraints when a switch block topology other than disjoint is used with internally-depopulated segments. It is an open question as to whether there is any method of satisfying both sets of constraints in this most general case. In cases where we cannot make the horizontal and vertical constraints agree, there are locations in the FPGA where a vertical wire wishes to connect to a horizontal wire, but the horizontal wire does not want a switch there, or vice versa. We resolve this conflict by inserting the switch, preferring to err on the side of too many switches in the routing, rather than too few.

The architecture generator currently contained in VPR is limited to generating architectures for detailed routing only when all the channels in the FPGA have the same width and segmentation distribution (VPR can generate routing graphs for global routing in which different channels have different capacities). Even with this restriction, we can explore a very large FPGA design space, and the architecture generation problem presents interesting challenges. A project is underway at the University of Toronto to extend the VPR architecture generator so that a user can specify different channel width and segmentation distributions for the horizontal and vertical directions [138]. This will allow architectural experimentation with Altera-like FPGAs.

We believe that the automatic generation of FPGA architectures to match a set of specifications is a fertile research area. Future research can investigate ways to ensure that the architectures created match all the specifications, and choose unspecified parameters intelligently enough that the best routability always results. Such research can explore not only better ways of generating architectures from the parameters we have listed in this section, but also whether a different architectural parameterization can be found such that all the specifications can always be satisfied.

4.3 Routability-Driven Router

Recall that VPR incorporates two different routers: one that is purely routability-driven, and one that is both routability and timing-driven. Both these routers can perform either combined global-detailed routing or global routing alone simply by changing the routing-resource graph passed to them. In this section we describe a purely routability-driven router, while the next section describes the timing-driven router. Once a routing is complete, VPR's graphics can be used to examine it — see Appendix A for sample pictures.

4.3.1 Cost Functions and Routing Schedules

Our routability-driven router is based on the Pathfinder negotiated congestion algorithm [85]; this purely routability-driven variant of the Pathfinder algorithm was described in Section 2.2.3. In the discussion below we will focus on new enhancements in our router and on important portions of the algorithm implementation that were not described in [85].

We define the cost of a node somewhat differently than [85] (see Equation (2.5)); the cost of using routing resource n when it is reached by connecting it to routing resource m is:

$$Cost(n) = b(n) \cdot h(n) \cdot p(n) + BendCost(n, m), \quad (4.3)$$

where the $b(n)$, $h(n)$ and $p(n)$ are the base cost, historical congestion, and present congestion terms defined in Section 2.2.3. The $BendCost(n, m)$ term is an enhancement we have made to improve the results of global routing. It penalizes bends when global routing is being performed, since global routes with many bends make it difficult or impossible for a subsequent detailed routing phase to utilize long wire segments [36, 31]. Hence reducing the number of bends in a global routing tends to lead to detailed routes that are both faster and require fewer tracks. If global routing only is being performed, $BendCost(n, m)$ is 1 if making the connection from node m to node n implies a bend — i.e. node m is a horizontal channel segment and node n is a vertical channel segment or vice versa. Including this $BendCost$ term in the total cost of using a node produces routes with very few unnecessary bends and does not significantly increase the global routing track count. If combined global-detailed routing is being performed there is no need to penalize bends, so $BendCost(n, m)$ is always zero in this case.

Notice that the functional form of (4.3) is different than that of (2.5); we multiply $b(n)$ and $h(n)$ together rather than adding them. When adding terms in cost functions, it is

important to ensure they are properly normalized to roughly the same range of magnitude so that both terms have an effect. We avoid having to normalize $h(n)$ to $b(n)$ by converting the addition to a multiplication.

In [85], the base cost of a node, $b(n)$ was set to its intrinsic delay. We have found that this is not the best choice; on average, about 10% fewer tracks per channel are required when the base costs of Table 4.1 are used instead. Note that the performance of the router is not extremely sensitive to the exact base costs chosen; the congestion avoidance terms in (4.3) ensure that the primary goal of the router is congestion avoidance, regardless of the $b(n)$ values.

TABLE 4.1 Base costs of different types of routing resource.

Routing Resource, n	Base Cost, $b(n)$
Wire segment	1
Logic block output pin	1
Logic block input pin	0.95
Source	1
Sink	0

Four of the five $b(n)$ values in Table 4.1 have virtually the same value — this encourages the router to use as few of these resources as possible to route each connection. The $b(n)$ value for an logic block input pin and for a sink are set to less than 1 to save CPU time. Since the maze expansion used to route a connection terminates when it reaches a sink corresponding to one of the net terminals, some CPU savings can be obtained by costing resources so that sinks tend to be reached earlier in the maze expansion. In other words, we would like to cost logic block input pins and sinks so that the maze expansion checks if the logic block next to a wire segment contains one of the net sinks for which we are searching before it expands more wire segments. To achieve this behaviour, a logic block input pin has a base cost of slightly less than 1, and a sink has a base cost of zero. Since congestion can not occur at sinks, using a base cost of zero for them will not cause route failures. Using a lower cost for logic block inputs and sinks in this way speeds the routability-driven router up by 1.5 to 2 times, depending on the FPGA architecture and circuit being routed.

We experimented with several different possibilities for the base costs of wire segments of different lengths: intrinsic delay, 1, length, the square root of length, and length+1. For both the routability-driven router described in this section, and the timing-driven router of the next section, setting $b(n)$ to 1 regardless of a wire segment's length yields the best results. Using the intrinsic delay of each routing resource as its

$b(n)$ value increased the number of tracks required per channel by 10% on average. The other choices of $b(n)$ led to performance between these extremes. Setting the cost of a wire segment to 1 regardless of its length also led to the highest speed circuits for the routability-driven router, since it encourages connections to use the smallest number of routing resources possible.

Ebeling et al [85] did not describe the exact functional form of the $h(n)$ and $p(n)$ congestion avoidance terms in (4.3), so we list the equations we use below. The present congestion penalty is updated whenever any net is ripped-up and re-routed according to

$$p(n) = 1 + \max(0, [\text{occupancy}(n) + 1 - \text{capacity}(n)] \cdot p_{\text{fac}}), \quad (4.4)$$

where $\text{occupancy}(n)$ is the number of nets currently using routing resource n , and $\text{capacity}(n)$ is the maximum number of nets that can legally use node n . The historical congestion penalty is updated only after an entire routing iteration. Its value during routing iteration i is:

$$h(n)^i = \begin{cases} 1, & i = 1 \\ h(n)^{i-1} + \max(0, [\text{occupancy}(n) - \text{capacity}(n)] \cdot h_{\text{fac}}), & i > 1 \end{cases} \quad (4.5)$$

The values of h_{fac} and p_{fac} in each routing iteration define what we call the *routing schedule* [2]. We have found that it is sufficient to keep h_{fac} constant for all routing iterations; the fact that $h(n)$ is incremented after every iteration in which node n is overused provides sufficient increase in the historical congestion penalty. Any value of h_{fac} between 0.2 and 1 works equally well. To achieve the absolute highest quality results, p_{fac} should initially be small, allowing congestion with little penalty, and gradually increase from iteration to iteration. For these highest quality results, p_{fac} should be 0.5 or less in the first iteration, and 1.5 to 2 times its previous value in each subsequent iteration. When the cost of congestion is increased this gradually, however, it takes several routing iterations (typically 5 - 10) to route even fairly “easy” routing problems, where there is more routing available than is needed by the circuit. For such “easy” problems, the router can be sped up by a factor of two to three times by making congestion very expensive immediately — in this case, we make p_{fac} 10000 in the first routing iteration, ensuring the router will always avoid congestion if it can, even in the first routing iteration. The amount of quality sacrificed by making congestion expensive immediately is quite small — typically the minimum achievable track count increases by only 2% to 4%. Throughout this book we will use the slowly increasing congestion cost schedule to achieve the absolute highest quality, however.

If a circuit has not routed in 30 routing iterations we normally classify it as unroutable. Allowing the router to try 45 routing iterations before giving up reduces the minimum track count by only 1% to 4%, on average.

4.3.2 Speed Enhancements

In addition to tuning the routing schedule and optimizing the routing-resource base costs, we made two other speed enhancements to the Pathfinder negotiated-congestion algorithm. First, we do not allow net routings to go more than three channels outside of their net bounding box. Since our routability-driven router uses a breadth-first maze router to make connections, this enhancement significantly reduces CPU time. Restricting each route to remain within 3 channels of its bounding box had no noticeable effect on the quality of the routing.

The second speed enhancement aids the routing of high-fanout nets, and results in an order-of-magnitude router speedup on large circuits. Recall that to route a k -terminal net, the maze router contained in VPR is invoked $k-1$ times. In the first invocation, the maze routing wavefront expands out from the net source until it reaches any one of the $k-1$ net sinks. The path from source to sink is now the first part of this net's routing. In a traditional maze router, and in the Pathfinder algorithm, the maze expansion (the *PriorityQueue* in Figure 2.10) is emptied, and a new wavefront expansion is started from the entire net routing found thus far. After $k-1$ invocations of the maze router all k terminals of the net will be connected.

This approach requires considerable CPU time for high-fanout nets. High-fanout nets usually span most or all of the FPGA. Therefore, in the latter invocations of the maze router the partial routing used as the net source will be very large, and it will take a long time to expand the maze router wavefront out to the next sink. We have developed a more efficient method. When a net sink is reached, we add all the routing resource segments required to connect the sink and the current partial routing to the wavefront (i.e. the *PriorityQueue*) with a cost of 0. We do not empty the current maze routing wavefront, but instead continue expanding normally. Since the new path added to the partial routing has a cost of zero, the maze router will expand around it first. Since this new path is typically fairly small, it will take relatively little time to add this new wavefront, and the next sink will be reached much more quickly than if the entire wavefront expansion had been restarted. Figure 4.11 illustrates the difference graphically. The shortest path computed to most of the nodes in the wavefront is still valid after a net sink is reached; we are taking advantage of this by restarting only a local portion of the wavefront instead of restarting the entire wavefront.

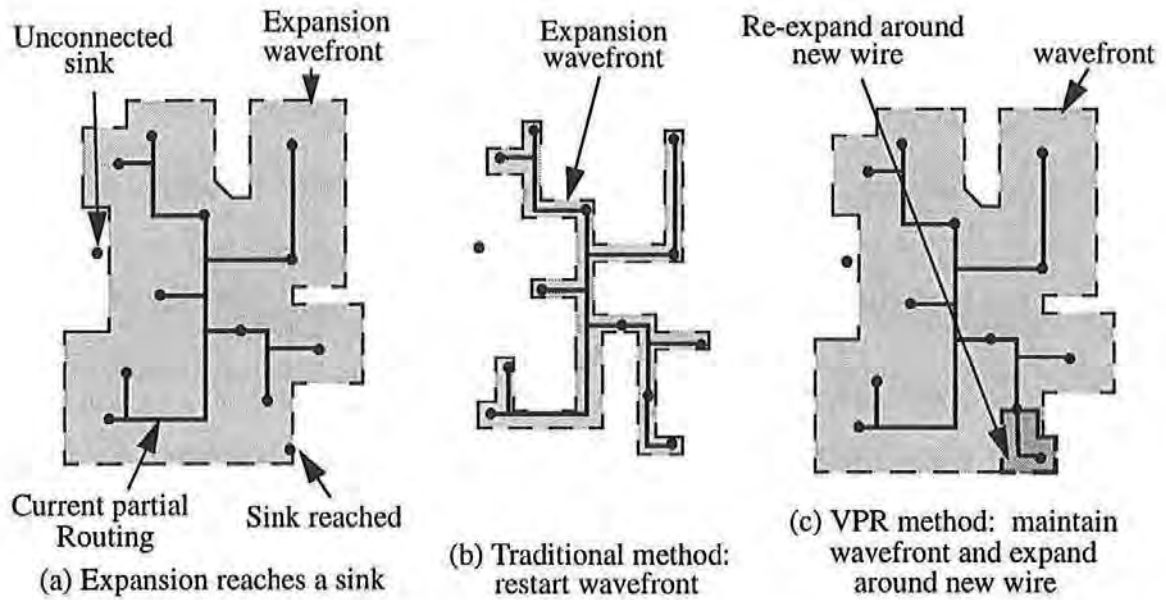


FIGURE 4.11 Difference between (b) complete and (c) local wavefront restart.

4.4 Timing-Driven Router

The timing-driven router developed for this work uses many of the ideas and algorithms of the Pathfinder negotiated congestion-delay router [85], which is described in Section 2.2.3. Our router contains two major improvements relative to the Pathfinder router, however: we optimize delay under the Elmore, rather than linear, delay model, and we dynamically vary the base cost of routing resources.

4.4.1 Superiority of Elmore Delay to the Linear Delay Model

Recall (as discussed in Section 2.2.3) that current academic FPGA combined global-detailed routers are either purely routability-driven or optimize only the linear delay model, in which every routing resource has a constant delay. The linear delay model is highly inaccurate for any FPGA which contains pass transistors in its routing, as the delay of a pass transistor strongly depends on the topology of the net routing into which it is inserted. For example, consider the two routing topologies in Figure 4.12, and the equivalent RC-circuit for each, as given in Section 2.2.4. Both the linear delay and Elmore delay models accurately predict the delay of a routing formed by a series of M wire segments connected by buffers:

$$T_{d, buffered} = M \cdot [T_{buf, intrinsic} + R_{buf} C_{total}] = M \cdot 250 \text{ (ps)} \quad (4.6)$$

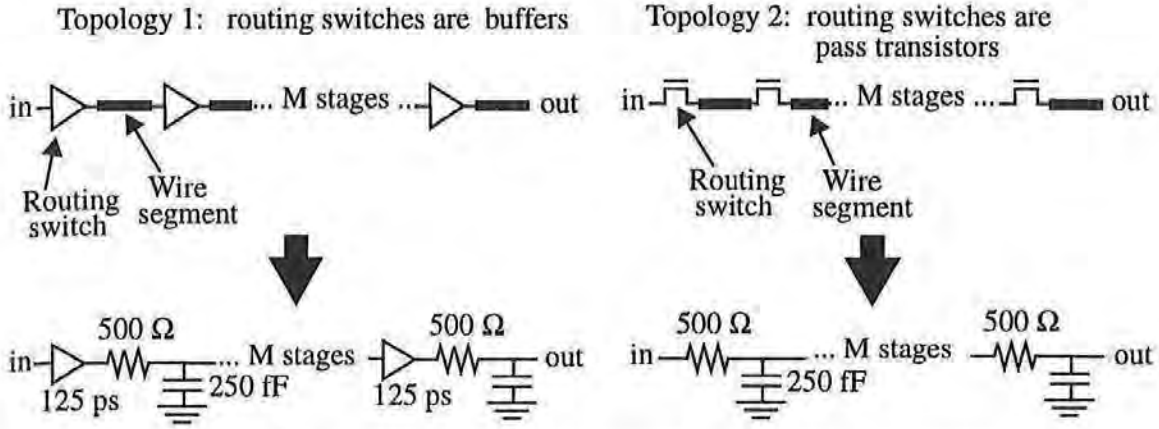


FIGURE 4.12 Two routing topologies and their equivalent circuits.

where the values of R_{buf} , $T_{\text{buf, intrinsic}}$, and C_{total} are taken from Figure 4.12. Note that C_{total} includes both the metal capacitance of a routing wire and the parasitic capacitance of the routing transistors attached to it.

The linear and Elmore delay models predict vastly different delays for a chain of pass transistors, however:

$$T_{d, \text{pass, linear}} = M \cdot R_{\text{pass}} \cdot C_{\text{total}} = M \cdot 125 \text{ (ps)} \quad (4.7)$$

$$T_{d, \text{pass, Elmore}} = \sum_{i=1}^M i \cdot R_{\text{pass}} C_{\text{total}} = M \frac{(M+1)}{2} R_{\text{pass}} C_{\text{total}} = M \frac{(M+1)}{2} \cdot 125 \text{ (ps)} \quad (4.8)$$

Notice that the Elmore delay accurately models the fact that the delay of a chain of pass transistors grows quadratically with the length of the chain, while the linear delay model does not. Setting equations (4.8) and (4.6) to be equal, we find that the chain of buffers is faster than the chain of pass transistors (for the R and C values we have assumed) whenever the number of routing switches in series, M , is more than 3. The linear delay model predicts that a chain of pass transistors is *always* faster than a chain of buffers, regardless of the length of the chain, M . It is obvious then, that using the linear delay model in an FPGA router will cause the router to make very poor decisions about when to use buffered routing switches and when to use pass transistor routing switches.

The second problem with the linear delay model is that it does not properly model the effects of fanout on net routings using pass transistors. Consider the two net topologies shown in Figure 4.13. Under the linear delay model, both topologies have the same delay to sink 1:

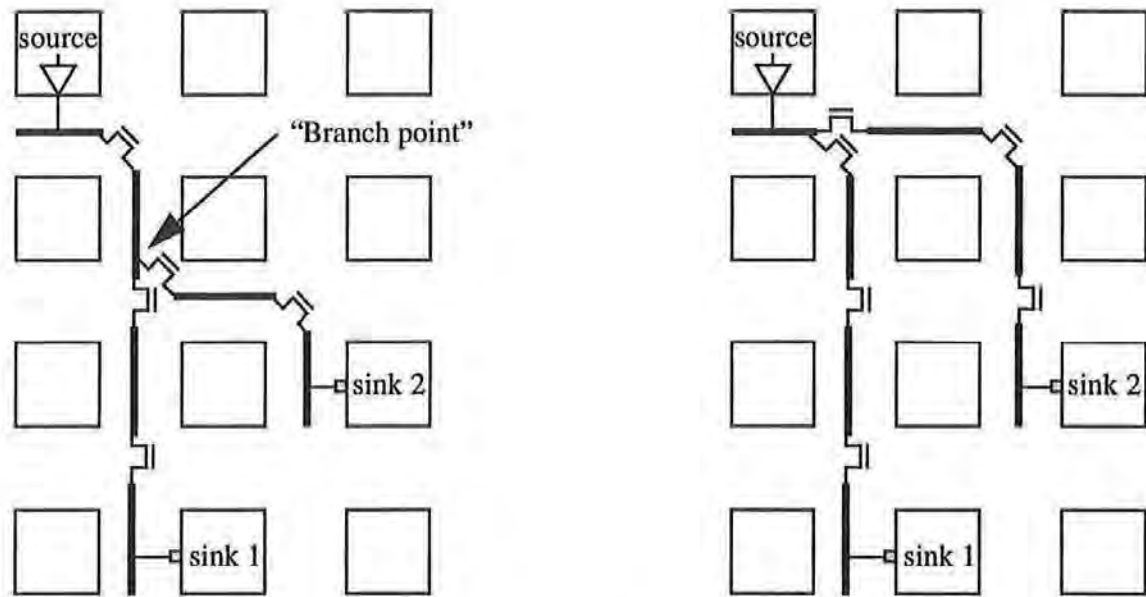


FIGURE 4.13 Two net topologies routed using pass transistor routing switches.

$$T_{linear} = T_{obuf} + R_{obuf}C_{total} + 3R_{pass}C_{total} \quad (4.9)$$

where we assume each wire has a total capacitance (metal plus parasitic switch capacitances) of C_{total} ; T_{obuf} and R_{obuf} denote the intrinsic delay and equivalent resistance of the logic block output pin buffer; and R_{pass} is the equivalent resistance of each pass transistor. Since both topologies have the same delay under the linear delay model, a router using this delay model would always pick the left-hand topology as superior, since it uses fewer routing resources.

Under the Elmore delay, however, the two topologies have different delays:

$$T_{Elmore, left} = T_{obuf} + 6R_{obuf}C_{total} + 8R_{pass}C_{total} \quad (4.10)$$

$$T_{Elmore, right} = T_{obuf} + 7R_{obuf}C_{total} + 6R_{pass}C_{total}. \quad (4.11)$$

If the resistance of the logic block output buffer is less than twice the resistance of a routing pass transistor (the typical case), the topology on the right has lower delay to sink 1. If sink 1 is time-critical therefore, the topology on the right is superior even though it uses more wirelength.

A related problem with the linear delay model is that it does not model the beneficial effect of isolating large capacitive loads with buffers. For example, if a buffered rout-

ing switch was used instead of a pass transistor at the “branch point” in the left routing topology of Figure 4.13, the delay to sink 1 drops because the capacitance in the branch of the routing leading to sink 2 is now isolated. This capacitance therefore does not load the output pin buffer or the pass transistor leading back to the net source. The Elmore delay correctly models this effect, but the linear delay model is completely oblivious to it.

It is clear then, that if an FPGA contains pass transistors, the linear delay model does not correctly rank routing alternatives. As mentioned in Section 2.2.4, however, the Elmore delay is much more accurate, and it has high “fidelity” [129, 130], meaning that it has a high probability of correctly ranking the routing alternatives.

4.4.2 Directly Optimizing the Elmore Delay

For all the reasons listed above, then, we would like our router to directly optimize the Elmore delay. Figure 4.14 shows the pseudo-code for this router. Like the Pathfinder algorithm described in Section 2.2.3, we perform routing iterations until there are no congested resources, where a routing iteration consists of ripping-up and re-routing every net in the FPGA. The manner in which we initially assume each connection is critical and route it for minimum delay, and gradually increase the cost of congestion, is also the same as in Pathfinder. The use of the Elmore delay, rather than the linear delay, leads to a significantly different algorithm to route each net, however. We focus on these new aspects of the router below.

Consider the routing of a single net, i , as described in the pseudo-code of Figure 4.14. Refer to Section 2.2.5 for a review of the nomenclature relevant to timing analysis and connection slacks. Initially the routing tree for net i , $RT(i)$, is just the routing-resource node corresponding to the net source. A wave expansion algorithm is invoked $k-1$ times to connect $RT(i)$ to each of the net’s $k-1$ sinks, with the most critical sink being connected first, and the least-critical sink being connected last. Consider the connection to sink j of net i . The cost to include a node, n , in a net’s routing is:

$$Cost(n) = Crit(i, j) \cdot delay_{Elmore}(n, topology) + [1 - Crit(i, j)] \cdot b(n) \cdot h(n) \cdot p(n) \quad (4.12)$$

We define the criticality of a connection, $crit(i, j)$ as:

$$Crit(i, j) = \max\left(\left[MaxCrit - \frac{slack(i, j)}{D_{max}}\right]^{\eta}, 0\right) \quad (4.13)$$

Let: **RT(i)** be the set of nodes, n , in the current routing of net(i); **RT(i)** also stores the $T_{\text{Elmore}}(n)$ and $R_{\text{upstream}}(n)$ for each of these nodes.

PriorityQueue be a set of $\{\text{TotalCost}(n), \text{PathCost}(n), R_{\text{upstream}}(n)\}$ tuples for each node, n , in the current wave expansion, sorted on **TotalCost**.

StoredPathCost and **StoredTotalCost** be arrays with one entry per routing resource node, with all entries initialized to a huge number.

```

Crit(i,j) = MaxCrit for all nets i and sinks j; /* First iteration routes all for high speed */
while (overused resources exist) { /* Illegal routing? */
  for (each net, i) {
    DynamicallyUpdateBaseCosts(i); /* Set base costs based on properties of net i */
    rip-up routing tree RT(i) and update affected p(n) values;
    RT(i) = NetSource(i);

    for (each sink, j, of net(i) in decreasing crit(i,j) order) {
      PriorityQueue = RT(i) at  $\{\text{TotalCost}(n) = \text{PathCost}(n) + \alpha \cdot \text{ExpectedCost}(n),$ 
         $\text{PathCost}(n) = \text{crit}(i,j) \cdot \text{delay}_{\text{Elmore}}(n), R_{\text{upstream}}(n)\}$  for each
        node  $n$  in RT(i);
      while (sink(i,j) not found) { /* Wave expansion */
        Remove lowest TotalCost node,  $m$ , from PriorityQueue;
        if ( $\text{TotalCost}(m) < \text{StoredTotalCost}(m) \ \&\& \ \text{PathCost}(m) <$ 
           $\text{StoredPathCost}(m)$ ) { /* Best path found to this node? */
           $\text{StoredPathCost}(m) = \text{PathCost}(m);$ 
           $\text{StoredTotalCost}(m) = \text{TotalCost}(m);$ 
          for (all fanout nodes  $n$  of node  $m$ ) { /* Expand node  $n$  */
            Compute  $R_{\text{upstream}}(m)$  from  $R_{\text{upstream}}(n)$  via (4.14)
            Add  $n$  to PriorityQueue at  $\{\text{TotalCost}(m) = \text{PathCost}(m) +$ 
               $\alpha \cdot \text{ExpectedCost}(m), \text{PathCost}(m) = \text{Cost}(m) + \text{PathCost}(n),$ 
               $R_{\text{upstream}}(m)\};$ 
          }
        }
      } /* End wave expansion */
      for (all nodes,  $n$ , in path from RT(i) to sink(i,j)) { /* Backtrace */
        Update  $p(n)$ ;
        Add  $n$  to RT(i) and store  $R_{\text{upstream}}(n)$ ;
      }
      Update Elmore delay of RT(i);
      For (all nodes  $n$  expanded during wave expansion) {
         $\text{StoredPathCost}(n) = \text{HugeNumber};$ 
         $\text{StoredTotalCost}(n) = \text{HugeNumber};$ 
      }
    }
  }
  Update  $h(n)$  for all  $n$ ;
  Perform timing analysis and update Crit(i,j) for all nets  $i$  and sinks  $j$ ;
} /* End of one routing iteration */

```

FIGURE 4.14 Pseudo-code of the timing-driven routing algorithm.

where $D_{\max}$ is the delay of the circuit critical path, and $\text{slack}(i,j)$ is the slack of the connection between the source and sink j of net i . η and MaxCrit are parameters that control how a connection's slack impacts the congestion-delay trade-off in cost function (4.12).

Note that the criticality equation used by Pathfinder is a special case of (4.13) with η and MaxCrit both equal to 1. When η is larger, nets with positive slack pay less attention to delay and more to congestion. MaxCrit^η is the maximum criticality any connection can have. Setting MaxCrit to 1 allows connections with 0 slack (i.e. connections on the critical path) to completely ignore congestion, while a MaxCrit of 0 forces all connections to ignore delay and makes the router completely routability-driven. We have experimentally determined that setting η to 1 is the best value, but setting MaxCrit to 0.99 results in better routability than setting it to 1, with no impact on circuit delay. Setting MaxCrit to 1 is dangerous, since nets on the critical path then pay no attention to congestion. Consequently, if two nets on the critical path are sharing the same routing-resource node the congestion may never be resolved if MaxCrit is 1.

Notice that the $\text{delay}_{\text{Elmore}}$ factor in (4.12) is a function of both the node, n , being expanded and the topology used to reach it. In order to calculate the Elmore delay increase resulting from adding a node to a partial routing, we need to know the *upstream resistance*, R_{upstream} , through which this node's capacitance must be discharged. The upstream resistance of a node, n , that was reached by connecting it to node m via a routing switch, *switch*, is

$$R_{\text{upstream}}(n) = \begin{cases} R(\text{switch}) + R_{\text{metal}}(n), & \text{if switch is a tri-state buffer} \\ R_{\text{upstream}}(m) + R(\text{switch}) + R_{\text{metal}}(n), & \text{if a pass transistor} \end{cases} \quad (4.14)$$

Notice that the upstream resistance accumulates through a chain of pass transistor switches, but not through a chain of buffered routing switches. $R_{\text{metal}}(n)$ is the metal resistance of this routing-resource node, and is 0 for all nodes that are not wiring segments. The Elmore delay of node n in this topology is then:

$$T_{\text{Elmore}}(n, \text{topology}) = T_{d, \text{intrinsic}}(\text{switch}) + \left[R_{\text{upstream}}(n) - \frac{R_{\text{metal}}(n)}{2} \right] C_{\text{total}}(n) \quad (4.15)$$

where $C_{\text{total}}(n)$ gives the total capacitance (metal plus parasitic switch capacitance) at node n . We subtract one-half of $R_{\text{metal}}(n)$ from $R_{\text{upstream}}(n)$ in (4.15) to account for the fact that the metal resistance is distributed. Note that in order to apply (4.14) and (4.15) we need to know the R_{upstream} value for the node via which node n was

reached; this implies we must store the R_{upstream} value associated with every node currently in the PriorityQueue.

We define $\text{PathCost}(n)$ to be the total cost of the path from the current partial routing tree to node n :

$$\text{PathCost}(n) = \sum_{l \in \text{path from RT}(i) \text{ to } n} \text{Cost}(l) \quad (4.16)$$

Using the PathCost of each node as the value on which we sort in the PriorityQueue results in a breadth-first search through the routing graph to route each connection. Note that the $\text{Crit}(i,j)$ values change for each sink, j , and also that the Elmore delay values of the partial routing tree $\text{RT}(i)$ must be updated after each sink is reached. Consequently, after a sink is reached the PathCost values stored in the PriorityQueue become invalid for every node. Hence, we cannot use our optimized breadth-first search from Section 4.3.2; the PriorityQueue must be emptied after each sink is reached, and the entire search begun from scratch. Since this would result in a very slow algorithm, and since Swartz et al have recently shown that a directed-search routing algorithm is faster than even our optimized breadth-first search [127], we employ a directed-search technique [105] instead.

To perform a directed-search, we must be able to estimate the total remaining cost, $\text{ExpectedCost}(n)$, from the current node to the target sink node. For source nodes and logic block output pin nodes we set $\text{ExpectedCost}(n)$ to zero. Since there is only one source node and at most a few output pin nodes explored during the maze expansion, this has little effect on CPU time. As well, it ensures that the router considers the option of connecting a logic block output pin to multiple tracks to optimize the routing of high-fanout nets.

To compute the ExpectedCost to route from a wire segment, n , to the target sink, j , we assume that:

- The connection to sink j will be completed using wires of the same type (length, connecting switch type, etc.) as node n , and
- There is no congestion along the shortest path to the target. That is, $p(m)$ and $h(m)$ are both 1 for all the nodes, m , that will be used to route the remainder of the connection.

We know the coordinates of both sink j and the current node, n , within the FPGA, so we can compute how many wires of the same length as node n will be needed to reach sink j . With this information and knowing the type of switch used to connect one wire of this type to another, we can compute the expected Elmore delay to the target. This is all the information we need to use (4.12) to compute the ExpectedCost to the target.

We then perform a directed search by using the $TotalCost$ of a node n as the sort value in the $PriorityQueue$:

$$TotalCost(n) = PathCost(n) + \alpha \cdot ExpectedCost(n, j). \quad (4.17)$$

The α parameter in (4.17) controls the aggressiveness of the directed search. An α of 0 leads to a breadth-first search, while an α of 1 leads to an A* search if $ExpectedCost(n, j)$ is a lower bound. We have found that an α value of 1.2 leads to the best CPU time without compromising quality; any value of α between 1 and 1.4 performs fairly well, however.

By choosing nodes, n , that minimize the $TotalCost(n)$ given by (4.17), we do *more* than just change the graph search method from breadth-first to directed; we also allow the router to perform more intelligent *buffer insertion*. To decide whether or not using a buffered switch is a net benefit at some point in the wave expansion, we need to know not only the discharge resistance back toward the net source, $R_{upstream}$, but also the total expected downstream capacitance needed to go from this point to the target sink. Equation (4.17) makes an approximation of this downstream capacitance visible through the $ExpectedCost$ from node n to the target sink, and therefore leads to more intelligent decisions as to what type of switch to use. Figure 4.15 illustrates a decision regarding whether to use a buffered switch or a pass transistor at some point in a wave expansion. If we choose the next node to expand in order to minimize the $PathCost$, given by (4.16), back to the net source, we create the net routing shown in Figure 4.15(b). If we instead choose the next node to expand in order to minimize the $TotalCost$, (4.17), to the target sink, we create the net routing shown in Figure 4.15(c). The routing tree of Figure 4.15(c) has lower delay to the sink and is therefore the correct choice.

Note that in most FPGA architectures it is not possible to separate the issue of buffer insertion from that of routing tree topology construction. Usually, once a routing topology is chosen for a net (to minimize wirelength, avoid congestion, use wires of the appropriate length, etc.) there is little or no choice remaining about which switches to use to connect the nodes in this routing tree. Consequently, we believe algorithms which simultaneously balance congestion avoidance, topology and wire length selection and buffering issues in one step are crucial in FPGA routing, and this is the approach we have taken in our timing-driven router.

$ExpectedCost(n)$ is not a lower bound on the cost needed to reach a sink from node n in some FPGA architectures, and this introduces one further requirement into the maze expansion. A node, n , can be included along the same path twice and result in a lower $TotalCost$ than would result if it were included in the routing path once. Figure 4.16 shows an example of this phenomenon. To avoid the creation of electri-

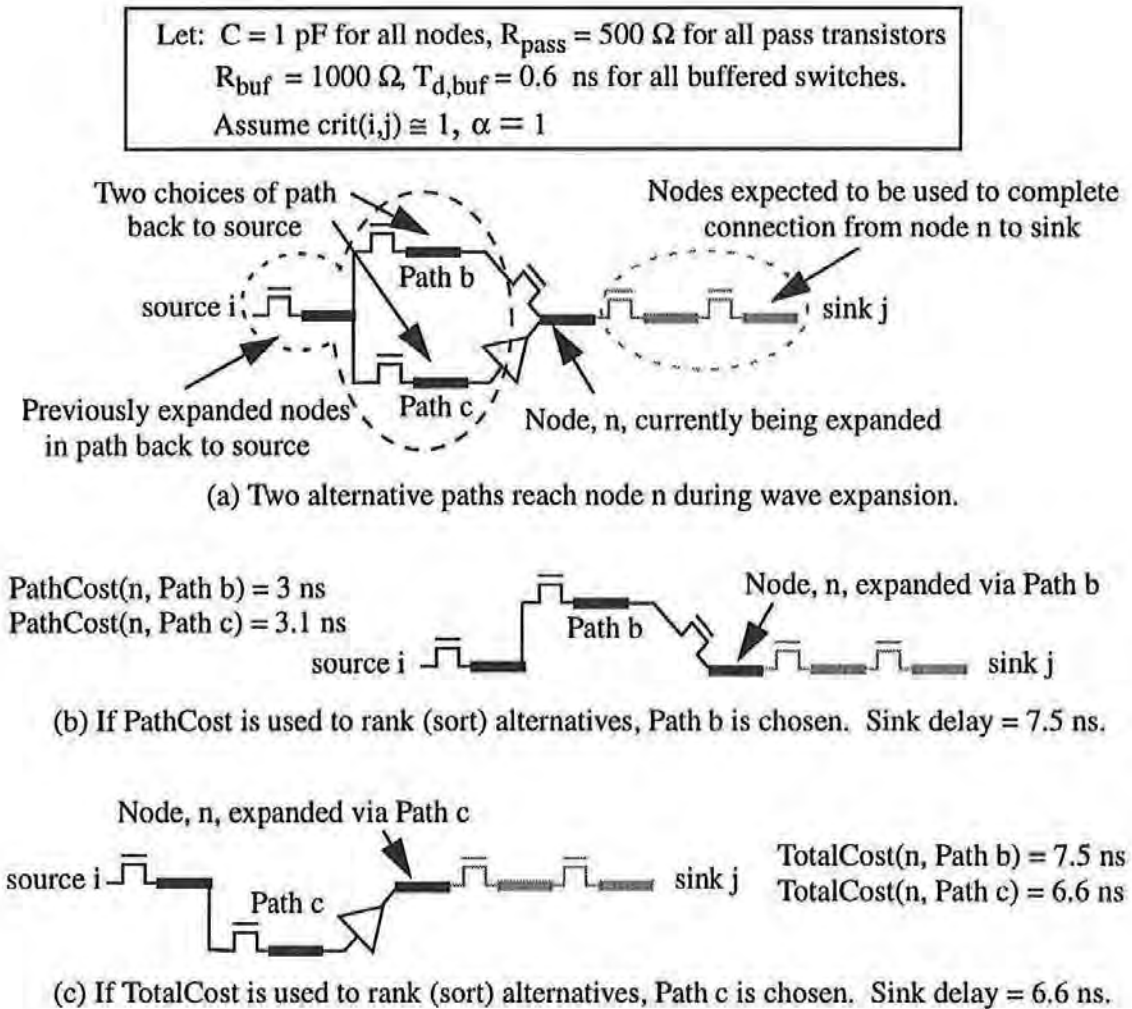


FIGURE 4.15 Wave expansion showing why ranking alternatives by TotalCost is necessary.

cally-illegal loops in a net's routing, we record both the TotalCost and PathCost at a node when it is expanded (these values are called $\text{StoredTotalCost}(n)$ and $\text{StoredPathCost}(n)$ in Figure 4.14), and we only allow a node to be re-expanded (re-inserted in the PriorityQueue) only if both the new PathCost and the new TotalCost are lower than the old values. In graph theoretic terms, the TotalCost defined by (4.17) is not a monotone path function, and maze routing algorithms are designed to operate only on monotone path functions [105]. Consequently, we need to augment the basic maze router algorithm to check for loops when using (4.17) as the cost of a path ending at a node, n .

Notice that in order to do a wave expansion that optimizes the Elmore delay, we must store three values for every entry, n , in the PriorityQueue: $\{\text{TotalCost}(n), \text{PathCost}(n), R_{\text{upstream}}(n)\}$. The queue is sorted based on the value of TotalCost(n), and

Let $R = 1000 \, \Omega$ for all switches, $C = 1 \, \text{pF}$ for all nodes,
and $T_{d,\text{buf}} = 0.3 \, \text{ns}$ for all buffered switches. Assume $\text{crit}(i,j) \equiv 1$, $\alpha = 1$

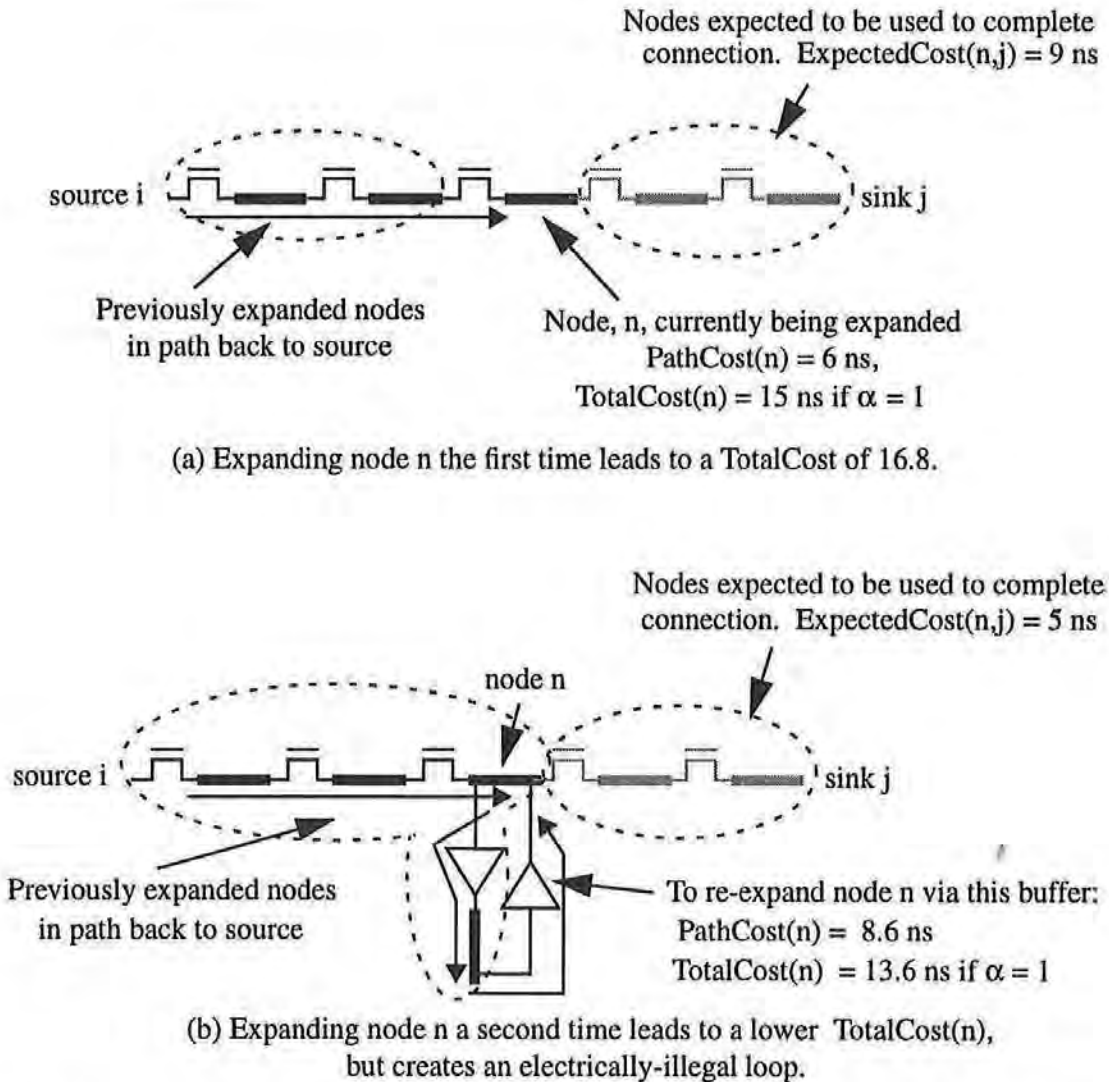


FIGURE 4.16 Adding a loop to a potential route can reduce the TotalCost of a node.

PathCost(n) and $R_{\text{upstream}}(n)$ are used only to expand a node n when it is removed from the PriorityQueue.

Once the target net sink, j , is reached, the PriorityQueue is emptied and a backtrace is performed to add the new connection to the routing tree, $RT(i)$. This new connection can add extra load capacitance to the rest of the routing tree, and hence affect the Elmore delay of other parts of the tree. We compute the load capacitance not isolated by buffers introduced by the new connection, and propagate it up the routing tree,

RT(i), until we encounter a buffer isolating us from the rest of the tree. The Elmore delay of the entire subtree rooted at this isolating buffer is then updated.

If there is another sink, j' , to connect, every node r in RT(i) is then placed back on the PriorityQueue with a PathCost of

$$PathCost(r) = crit(i, j') \cdot delay_{Elmore}(r, topology), \quad (4.18)$$

a TotalCost(r) given by (4.17), and the $R_{upstream}(r)$ value stored for each node in the routing tree. The directed search procedure is then started again to connect this routing tree to sink j' .

4.4.3 Net Routing Algorithm Complexity

The net routing algorithm above has the same complexity to route a k -terminal net as a conventional maze router, since we can evaluate both the cost of a node and the ExpectedCost from a node to the target sink in constant time. Routing a k -terminal net is an $O(k^2 \log k)$ operation if the FPGA is uncongested and the directed search is able to proceed straight to each target sink without backtracking. If the FPGA is highly congested, the directed search may have to search the entire routing graph before finding each sink in the worst case. In this case, the complexity of both the net routing algorithm above and a conventional maze router is $O(kR \log R)$, where R is the number of nodes in the routing-resource graph. This worst case complexity almost never occurs, however, even for "high-stress" routing in which there is considerable congestion. The typical complexity of the router is $O(k^2 \log k)$.

4.4.4 Dynamic Base Costs

One problem with signal routing algorithms based on any form of maze router is that they route a k -terminal net one connection, or sink, at a time. The best choice of wires to use in connecting to a sink depends on the other sinks to be connected, but a maze router does not see such overall issues. In [86], Nag and Rutenbar changed the cost of long wire segments as a function of the span of the net being routed, in order to bias the router to use the proper type of routing resources for each net. In a similar manner, we dynamically change the base cost of wire segments that connect via pass transistors, as a function of the net fanout.

Wire segments connected by pass transistors are not as efficient at routing high-fanout nets as wire segments connected by buffered routing switches. A routing tree connected entirely with pass transistors has no buffers isolating the capacitance of portions of the tree from other parts, so the switches tend to become heavily loaded,

resulting in a large delay, as the tree becomes large. By updating the Elmore delay of the routing tree, $RT(i)$, after each sink is reached, we make the router aware of some of these issues. In subsequent connections it will notice that portions of the routing tree are becoming slow, and will tend to avoid starting time-critical connections from these portions of the routing tree. Consequently, either the net's routing will become more "star-like," with many connections sharing relatively little wiring, or many of the subsequent connections will be made using wires that can be connected with buffers. Routing a high-fanout net as a star wastes considerable wiring, however. Similarly, routing the first several sinks of a net on wires that connect with pass transistors before beginning to use more wires connected with buffers also wastes wiring. A second problem is that if the router does not use a star-like topology because many of the net sinks are not delay-critical, it may slow down timing-critical connections made earlier for this net by adding capacitive loading to the routing tree in undesirable places. It would be better if the routing of a high-fanout net preferred to use buffered switches wherever possible, in order to maximize the possibility of re-using some of this wiring in later connections, and to minimize the possibility of later connections affecting the timing of the early connections.

The default base costs used in our timing-driven router are the same as those given in Table 4.1, except they are all normalized to the average delay of a routing resource so that both terms in (4.12) have roughly the same magnitude. We ran experiments to evaluate the effect of different base costs of wire segments as follows: we set the base cost of each wire to be proportional to its intrinsic delay, its length, the square root of its length, its length+1, and 1. Just as for the routability-driven router, we found that a using the same base cost for all wire segments (in this case, the average delay of a routing resource), regardless of length, was best.

Before we begin the routing of a net, however, the `DynamicallyUpdateBaseCosts` routine of Figure 4.14 is called to alter these default base costs as a function of the net fanout. The base costs of all wire segments that use pass transistors to connect to other wires of the same type are increased as a function of the net fanout:

$$b_{\text{pass transistor wires}}(n) = \text{AverageRoutingResourceDelay} \cdot \sqrt{k-1} \quad (4.19)$$

where k is the number of terminals on the net. Wire segments that connect to other wires via buffers have their base cost left as the average delay of a routing resource. By increasing the base cost of certain nodes this way, we ensure that nets with high fanout prefer to use buffered routing resources, allowing the creation of buffered "spines" early in the net construction. Note that connections that are very delay critical will still use pass transistor based routing if it is the fastest, since such connections pay little attention to base cost. Similarly, if the buffered routing resources become congested, the congestion avoidance terms in (4.12) will eventually force the use of

unbuffered routing resources even for high-fanout nets. We tried using $(k-1)$ instead of the square root of $k-1$ in Equation (4.19), but this made pass transistor based routing too expensive, and led to slightly inferior results.

The routing-resource graph is very large, so scanning through the entire graph changing $b(n)$ for each node n before routing each net would be prohibitively slow. Instead, at each node, n , we store a pointer to a base cost to be used for a routing resource of n 's type. Consequently, to change the base costs dynamically the number of values we must update is equal to the number of different routing-resource types in the FPGA. Since FPGAs tend to contain only a few different lengths of wire, and they are interconnected using only a few different types of switches, updating the base costs this way does not consume a significant amount of CPU time.

Table 4.2 shows the performance of our timing-driven router with and without this dynamic costing enhancement for 15 large MCNC circuits. The FPGA architecture in this experiment contains five different wire types, in equal proportions: lengths 1, 2, 4, 8 and long lines. The length 1 and 2 lines use pass transistor switches to connect to each other, while the other wires use buffered routing switches. Each logic block is a logic cluster containing 4 BLEs and with 10 inputs. Finally, each circuit is routed in an FPGA with 20% more tracks per channel than the absolute minimum required for successful routing. On 9 of the 15 circuits, dynamic costing improved the circuit speed; on 3 circuits dynamic costing slowed the circuits slightly, and on the remaining 3 there was no change. On average, the circuits sped up by 4%. This speedup is more significant than one might first assume. Without dynamic costing, the circuit delay is 5% higher than the minimum the router can achieve while ignoring congestion and routing every connection for minimum delay. With dynamic costing, the circuit delay is only 1% higher than this congestion-oblivious delay. Note also that the CPU time to route the circuits is 33% higher without dynamic costing than with it. By minimizing the construction of "stars" for high-fanout nets, dynamic costing results in shorter wave expansions and saves CPU time.

4.4.5 Routing Schedule

We found that the best routing schedule for the timing-driven router is very similar to the best schedule for the routability-driven router, which was described in Section 4.3.1. We set h_{fac} to 1 for all routing iterations, p_{fac} to 0.5 for the first two routing iterations, and to twice its previous value for all subsequent iterations. Making the cost of congestion very high even in the first few iterations sacrifices considerable result quality, often increasing the routed critical path delay by 33%. As well, making congestion expensive immediately does not consistently save CPU time for the timing-driven router. While making congestion very expensive early reduces the number of router iterations, it tends to make the directed search detour around congestion

TABLE 4.2 Effectiveness of dynamic base costs for routing resources.

Circuit	# 4-LUTs	With Dynamic Costing		Without Dynamic Costing		Delay (ns) Ignoring Congestion ^a
		Delay (ns)	CPU (s)	Delay (ns)	CPU (s)	
alu4	1522	35.2	23	35.7	30	34.1
apex2	1878	40.7	38	42.3	38	40.7
apex4	1262	35.0	22	34.6	28	35.0
bigkey	1707	22.2	19	22.2	26	22.2
des	1591	35.6	25	35.6	37	34.3
diffeq	1497	39.9	14	40.5	20	39.8
dsip	1370	23.0	18	23.8	33	22.7
elliptic	3604	65.8	56	66.9	81	65.3
ex5p	1064	35.1	19	36.2	27	35.1
frisc	3556	76.2	74	74.8	100	73.9
misex3	1397	31.2	22	31.2	32	31.5
s298	1931	59.7	26	62.0	37	59.3
seq	1750	36.9	31	35.5	43	36.0
spla	3690	53.7	138	72.0	171	54.9
tseng	1047	38.8	8	39.7	10	38.2
Arith. Av:	1924	41.9	36	43.5	48	41.5
Geom. Av:	1764	39.6	27	40.7	37	39.2

a. Recall that there is no known polynomial-time algorithm to route a net with guaranteed minimum Elmore delay [129], so the congestion-oblivious routing delay is *not* a lower bound on the achievable delay.

more in the early iterations. The net effect is sometimes a reduction in CPU time, and sometimes an increase.

For most FPGA architectures 30 router iterations are sufficient to route even difficult problems. For FPGAs composed of very large logic blocks with many pins, however, we found that the timing-driven router sometimes needed 60 iterations to route difficult problems.

4.5 Delay Extraction and Timing Analysis

VPR contains a delay extractor that computes the Elmore delay of any routed net. To make delay extraction faster, after the routing-resource graph is built we traverse the graph once and lump all the parasitic switch capacitance, plus the metal capacitance, into a total capacitance value, C_{total} , at each node. Every node in the routing-resource graph can have a different C_{total} , and a different distributed metal resistance, R_{metal} . Similarly, every switch in the FPGA can have a different R_{switch} and intrinsic delay. We compute the Elmore delay to every routing-resource node in a net in linear time via two depth-first traversals of the tree describing the net's routing. We can also incrementally update the Elmore delay of a tree as new connections are added to the tree.

VPR also includes a full path-based timing analyzer (timing analysis was described in Section 2.2.5) that can determine both the critical path of a circuit and the slack of every connection to be routed. This timing analyzer will work with *any* logic block. The architecture description file lists the delay of every combinational and sequential element in a logic block, and the circuit netlist can specify a different interconnection pattern between logic block inputs, outputs and these internal combinational and sequential elements for each logic block in the circuit. After the netlist is read, a graph describing the circuit timing relationships is created. We perform a breadth-first traversal of this graph in order to levelize it. Graph nodes with no inputs, such as input pads, are inserted in the level 0 list, while nodes with fan-in only from level 0 nodes go into the level 1 list and so on. Levelizing the timing graph in this way speeds the subsequent breadth-first traversals needed to determine the net slacks after each iteration of the router. After routing, the net delays computed by the delay extractor are back-annotated into the timing graph, and two breadth-first traversals of the graph are used to compute the critical path and the slack of each connection. This timing analyzer can also be operated as a stand-alone tool by invoking VPR with the appropriate command line options [134].

Since optimizing the timing of a routing usually requires multiple iterations of routing, extracting the net delays and performing timing analysis, we took considerable care to ensure delay extraction and timing analysis were both fast. Table 4.3 summarizes the time required to perform various operations on the largest MCNC benchmark, clma, which contains 8381 4-input LUTs and 33 registers. Finally, as Appendix A shows, VPR's built-in graphics can display the nets on the critical path to make the speed bottlenecks in an FPGA apparent.

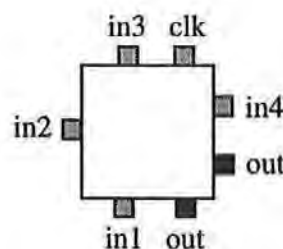
TABLE 4.3 CPU times to perform operations on circuit clma (8381 4-input LUTs).

Operation	CPU (s), 300 MHz UltraSparc
Extract net delays (30 455 two-point connections)	0.7 s
Build and levelize timing graph (78 000 nodes)	0.25 s
Back-annotate net delays and perform timing analysis	0.08 s

4.6 Router and Placement Algorithm Validation

4.6.1 Routability-Driven Router and Placement Algorithm

In this section we evaluate the quality of our routability-driven routing algorithm and our placement algorithm by comparing their results to those of other published CAD tools on a set of standard benchmark circuits [17]. The various FPGA parameters used in this section were always chosen to allow a direct comparison with previously published results. All the results in this section were obtained with a logic block consisting of a single BLE; that is, a 4-input LUT plus a flip flop, as shown in Figure 3.1. The clock net was not routed in sequential circuits, as it is usually routed via a dedicated routing network in commercial FPGAs. Each LUT input appears on one side of the logic block, while the logic block output is accessible from both the bottom and right sides, as shown in Figure 4.17. Each logic block input or output can connect to any track in the adjacent channel(s) (i.e. $F_c = W$). Each wire segment can connect to three other wiring segments at channel intersections (i.e. $F_s = 3$) and the switch block topology used is the disjoint topology shown in Figure 4.7(a). There are no standard benchmark results available for routing FPGAs that contain long wire segments, so all routing wires are of length 1 in the target architecture.

**FIGURE 4.17** Logic block pin locations.

Experimental Results with Input-Pin Doglegs

Most previous FPGA routing results have assumed that “input-pin doglegs” [101] are possible. One way to build the connection block between an input pin and the tracks to which it connects uses F_c independent pass transistors controlled by F_c SRAM bits. In this case it is possible to turn on two of these switches in order to electrically connect two tracks via the input pin, as shown in Figure 4.18(a). This type of connection is called an input-pin dogleg. Commercial FPGAs, however, use a multiplexer to implement the connection block from an input pin to a channel, as shown in Figure 4.18(b). Hence only one track may be connected to an input pin. Using a multiplexer rather than independent pass transistors saves considerable area in the FPGA layout. As well, normally there is a buffer between a track and the connection block multiplexers to which it connects in order to improve speed; this buffer also means that input-pin doglegs can not be used. Therefore, while we allow input-pin doglegs in this section in order to make a fair comparison with past results, it would be best if in the future FPGA routers were tested without input-pin doglegs.

Table 4.4 lists the minimum number of tracks per channel required for a successful routing by various CAD tools on a set of 9 MCNC benchmark circuits. All the results in Table 4.4 are obtained by routing a placement produced by Altor [67], a min-cut based placement tool. Three of the columns consist of two-step (global then detailed) routing, while the other routers perform combined global and detailed routing. VPR’s routability-driven router requires 10% fewer tracks than the second best router, and the third best router consists of VPR’s global route phase plus SEGA [100] for detailed routing.

Table 4.5 lists the number of tracks required to implement these benchmarks by tools which change both the placement and the routing of the circuits. The size column lists the number of logic blocks in each circuit. VPR uses 13% fewer tracks when it performs combined global and detailed routing than it does when SEGA is used to

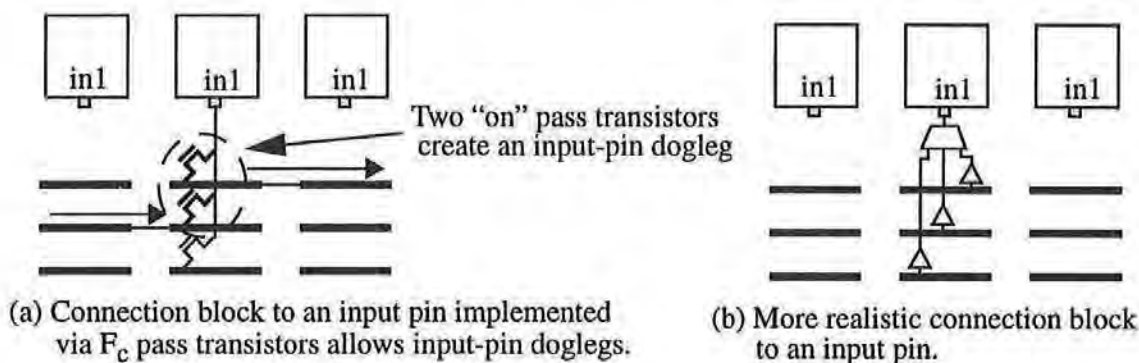


FIGURE 4.18 Two methods of building an input pin connection block.

TABLE 4.4 Tracks required to route placements generated by Altor (input-pin doglegs allowed).

Global Router	LocusRoute [97]		GBP [91]	OGC [92]	IKMB [93]	VPR	TRACER [94]	VPR [17]
Detail Router	CGE [99]	SEGA [100]				SEGA [101]		
9symml	9	9	9	9	8	7	6	6
alu2	12	10	11	9	9	8	9	8
alu4	15	13	14	12	11	10	11	9
apex7	13	13	11	10	10	10	8	8
example2	18	17	13	12	11	10	10	9
k2	19	16	17	16	15	14	14	12
term1	10	9	10	9	8	8	7	7
too_large	13	11	12	11	10	10	9	8
vda	14	14	13	11	12	12	11	10
Total	123	112	110	99	94	89	85	77

perform detailed routing on a VPR-generated global route. The FPR tool [88] first performs placement and global routing simultaneously with a partitioning-based algorithm, and then performs a separate detailed routing step. It requires 87% more total tracks than VPR, so it appears that performing global and detailed routing in one step is more important than performing placement and global routing together. Finally, allowing VPR to place the circuits instead of forcing it to use the Altor placements reduces the number of tracks VPR requires to route them by 40%, indicating that VPR's simulated annealing based placer is considerably better than the Altor min-cut placer.

Experimental Results Without Input Pin Doglegs

Table 4.6 compares the performance of VPR with that of the SPLACE/SROUTE [33] tool set, which does not allow input pin doglegs. SROUTE is an iterated maze router — when some nets fail to route, it rips up every net in the circuit and tries routing them again, with nets that would not route in the last iteration routed first in this new iteration. When both tools are allowed only to route an Altor-generated placement VPR requires 13% fewer tracks than SROUTE. Since VPR's routability-driven router

TABLE 4.5 Tracks required to place and route circuits (input-pin doglegs allowed).

Placement	Number of Logic Blocks in Circuit	FPR [88]	VPR	VPR
Global Routing				
Detailed Routing			SEGA	
9symm1	70	9	6	5
alu2	143	10	7	6
alu4	242	13	8	7
apex7	77	9	5	4
example2	120	13	5	5
k2	358	17	10	9
term1	54	8	5	5
too_large	148	11	7	6
vda	208	13	9	8
Total	—	103	62	55

is also based on a maze router, this difference is due to the more sophisticated congestion avoidance scheme used in VPR. When the tools are allowed to both place and route the circuits, VPR requires 29% fewer tracks than the SPLACE/SROUTE combination. Both VPR and SPLACE are based on simulated annealing. We believe the difference in result quality is due mainly to the higher quality of VPR's annealing schedule.

The benchmarks used in the previous three tables range in size from 54 to 358 logic blocks, and accordingly are too small to be very representative of today's FPGAs. Table 4.7 presents results for the 20 largest MCNC benchmark circuits, which range in size from 1047 to 8383 logic blocks. We use FlowMap [46] to technology map each circuit to 4-LUTs and flip flops, and VPack to combine flip flops and LUTs into BLEs. The number of I/O pads that fit per row or column is set to 2, in line with current commercial FPGAs. Each circuit is placed and routed in the smallest square FPGA which can contain it. Input-pin doglegs are not allowed. Note that three of the benchmarks, bigkey, des, and dsip, are pad-limited in the FPGA architecture assumed.

Table 4.7 compares the number of tracks required to place and completely route circuits with VPR with the number required to place and globally route the circuits with

TABLE 4.6 Tracks required to place and route circuits (no input-pin doglegs allowed).

Placement	Altor		SPLACE [33]	VPR
Global + Detailed Route	SROUTE [33]	VPR	SROUTE	
9symm1	7	6	7	5
alu2	9	8	8	6
alu4	12	10	9	7
apex7	9	9	6	4
example2	11	10	7	5
k2	15	14	11	9
term1	8	7	5	4
too_large	11	9	8	7
vda	12	10	10	8
Total	94	83	71	55

VPR and then perform detailed routing with SEGA. Using SEGA to perform detailed routing on a global route generated by VPR increases the total number of tracks required to route the circuits by 70% vs. having VPR perform the routing completely. This large track count increase is likely due to the reduced optimization possibilities when global and detailed routing are performed separately, particularly when input-pin doglegs are not allowed.

To encourage other FPGA researchers to publish routing results using these larger benchmarks, we have issued an “FPGA challenge” to the research community. We will pay researchers \$1 (Canadian!) for each track by which they reduce the total number of tracks required from that of the previously best results. The technology-mapped netlists, the placements generated by VPR and the currently best routing track total are available at <http://www.eecg.toronto.edu/~jayan/software/software.html>. In the year since this challenge was issued (in a presentation at the FPL ‘97 workshop in London, UK), no CAD tool has beaten our track count totals.

4.6.2 Timing-Driven Router

Unfortunately, there are no standard benchmark results available to compare timing-driven FPGA routers on the basis of timing. Lee and Wu [94] provided critical path

TABLE 4.7 Channel widths required to place and route the 20 largest MCNC benchmark circuits.

Circuit	# LBs	SEGA	VPR	Circuit	# LBs	SEGA	VPR	Circuit	# LBs	SEGA	VPR
alu4	1522	16	10	dsip	1370	9	7	s298	1931	18	7
apex2	1878	20	11	elliptic	3604	16	10	s38417	6406	10	8
apex4	1262	19	12	ex1010	4598	22	10	s38584.1	6447	12	9
bigkey	1707	9	7	ex5p	1064	16	13	seq	1750	18	11
clma	8383	25	12	frisc	3556	18	11	spla	3690	26	13
des	1591	11	7	misex3	1397	17	10	tseng	1047	9	6
diffeq	1497	10	7	pdc	4575	33	16	Total	—	334	197

delay values for several MCNC benchmark circuits routed with Tracer and with SEGA, but they did not provide information on the logic block delays, switch resistances etc. used to compute these delays. As well, they used the Penfield-Rubinstein rather than the Elmore delay, so even if we determined these values we could not directly compare our timing to theirs. Finally, the FPGA architecture they targeted included only unit length wires connected by pass transistors, so it is not a good model of current commercial FPGAs.

We instead validate our timing-driven router by comparing the minimum track count and the critical path delay it can achieve to that of our routability-driven router. The routing architecture we use for this comparison is very similar to that of the recent Xilinx 4000X series FPGAs [139, 140].¹ This routing architecture contains 25% length 1 wires, 12.5% length 2 wires, 37.5% length 4 wires, and 25% “one-quarter longs”, whose length is one-fourth of the chip. The length 1 and 2 wires connect via pass transistors, while the longer wires connect via tri-state buffers. As well, pass transistor switches also allow the length 4 wires to connect to the length 1 and 2 wires, and the one-quarter longs to connect to length 1 wires. The logic block used in this experiment is a logic cluster of four BLEs, with 10 inputs per logic block — this is more typical of the complexity of modern FPGA logic blocks than a single BLE logic block. We extracted the timing parameters for this FPGA architecture from SPICE simulations of a 0.35 μm TSMC process (see Section 6.2.3).

1. Many thanks to Jordan Swartz, who enhanced the VPR routing graph generator to create this Xilinx 4000X-like routing architecture [140].

Table 4.8 summarizes the results on the 20 largest MCNC circuits. The $W_{\min}$ columns give the minimum number of tracks per channel required by each router to successfully route each circuit. We define a low-stress routing [127] to be a routing in which there are 20% more tracks per channel than the minimum number of tracks per channel, $W_{\min}$, required by the timing-driven router. We present the critical path delay achieved by each router under low-stress routing conditions because this is the typical routing case in FPGAs — usually there are some spare tracks. The infinite-routing column gives the delay achievable by the timing-driven router when it ignores congestion completely — that is, when it assumes it has an infinite supply of routing. Finally, the CPU time column indicates the time required to complete each low-stress routing on a 300 MHz UltraSparc workstation.

In Section 4.6.1 we showed that other CAD tools to which we can compare required from 10% to 87% more tracks to route circuits than VPR's routability-driven router. Table 4.8 shows that our timing-driven router requires only 6% more tracks than our routability-driven router, on average, so it is clear that VPR's timing-driven router performs very well in terms of circuit routability.

The circuit timing columns show that a timing-driven router is crucial to good circuit speed. The circuits produced by the timing-driven router are from *1.5 to 5 times* faster than those produced by the routability-driven router, and the average circuit speedup due to timing-driven routing is over *2.6 times*. With only 20% extra tracks, the timing-driven router comes within 12% of the delay value it can achieve when ignoring congestion — that is, the delay value it could achieve with infinite tracks. Although not shown in Table 4.8, the average delay achieved by the timing-driven router for high-stress routing, where $W = W_{\min}$, is only 31% higher than the delay value it can achieve with infinite tracks. This graceful degradation in circuit delay as the routing problem becomes increasingly difficult is an attractive property.

Finally, notice that the CPU time required to complete a low-stress routing is from 3.3 to 16.3 times less (ten times less on average) for the timing-driven router than for the routability-driven router. This shows the superiority of the directed-search algorithm employed in the timing-driven router over even the optimized breadth-first search used in the routability-driven router. A breadth-first search slows down considerably on FPGAs like this which have large logic blocks, and hence large W values. The CPU time of a depth-first search, on the other hand, is largely insensitive to W . The timing-driven router is very fast; it required only 3.2 minutes to route *clma*, which contains over 8300 4-LUTs.

TABLE 4.8 Comparison of timing-driven router to routability-driven router.

Circuit	Routability-Driven Router			Timing-Driven Router			
	$W_{\min}$	Low-Stress Delay (ns)	Low-Stress CPU (s)	$W_{\min}$	Low-Stress Delay (ns)	Low-Stress CPU (s)	∞ -Routing Delay (ns)
alu4	32	91.6	186	35	41.5	56	35.7
apex2	35	91.6	257	38	43.6	40	42.2
apex4	32	80.1	163	35	40.7	28	34.9
bigkey	24	88.9	123	25	23.8	26	23.4
clma	43	233.8	2801	47	103.3	192	97.6
des	23	114.9	247	25	37.8	25	36.6
diffeq	24	68.7	67	25	44.5	11	38.7
dsip	22	174.9	115	22	35.4	26	25.0
elliptic	36	158.1	583	39	75.5	62	63.2
ex1010	35	173.6	785	37	62.6	93	54.6
ex5p	33	79.4	115	35	36.6	21	35.3
frisc	34	245.8	575	35	77.9	82	73.5
misex3	31	78.1	137	33	35.9	40	32.8
pdc	51	192.9	2624	55	75.4	161	71.2
s298	29	182.1	205	31	75.2	37	58.9
s38417	30	189.6	929	31	57.6	93	50.5
s38584.1	26	190.5	624	27	41.1	62	35.1
seq	34	81.7	250	36	41.0	41	36.7
spla	46	208.0	1400	49	63.8	100	57.2
tseng	22	71.5	34	23	40.1	8	38.2
Arith. Av.	32.1	139.8	611	34.2	52.7	60	47.1
Geom. Av.	31.2	127.3	312	33.1	49.3	45	44.0

4.7 Summary

In this chapter we described the routing portion of the Versatile Place and Route CAD tool. To achieve VPR's goal of flexibility, the entire routing portion of the tool works with a routing-resource graph representation of an FPGA; by changing the routing-resource graph, radically different FPGAs can be investigated. VPR includes an FPGA architecture generator that can generate a routing-resource graph from a succinct architectural description, making it easy to describe and investigate many architectures quickly. This automatic generation of good FPGA architectures from a list of parameter specifications is a new and interesting problem in FPGA CAD.

This chapter described the algorithms used in both the routability-driven and the timing-driven routers included in VPR. The VPR timing-driven router is the first FPGA router that directly optimizes the Elmore delay, and the first published router that can properly optimize for FPGAs that contain both pass transistors and tri-state buffers in their routing. VPR also includes high-quality net delay extraction and path-based timing analysis modules.

The result quality of the VPR routers and of the VPR placement algorithm is very high. The routability-driven router outperforms all routers to which we can compare in terms of routing circuits with the minimum amount of FPGA routing. As well, the combination of VPR's placement and routability-driven routing algorithms outperforms all combinations of place and route tools to which we can compare. The timing-driven router also produces excellent results. It requires only slightly (6%) more tracks per channel than the routability-driven router, and produces circuits that are 2.6 times faster, on average.

In the next three chapters, we use the VPR, VPack and T-VPack CAD tools described in this chapter and in Chapter 3 to investigate three different issues in FPGA architecture. The next chapter describes our investigation into the first of these three issues: FPGA global routing architecture.

Global Routing Architecture

In this chapter we investigate which *global routing architectures* lead to the best FPGA area-efficiency [2, 3]. We use the term global routing architecture to refer to the distribution of routing tracks across an FPGA; that is, the relative number of tracks contained in each channel of the FPGA. In the next section we describe some of the different types of global routing architectures, and explain why this is an important problem in FPGA design. Section 5.2 describes the experimental flow we use to evaluate different global routing architectures — this flow is based on the CAD tools described in Chapters 3 and 4. In Section 5.3 we investigate *directionally-biased* global routing architectures, in which the channels in the vertical direction have a different width than those in the horizontal direction. Section 5.4 examines *non-uniform* global routing architectures, which have wider channels in some regions of the FPGA than in others.

5.1 Motivation

Recall that in FPGAs all routing resources are prefabricated. This means that the number of routing tracks in each channel is set by the FPGA manufacturer; this is quite different from the situation with MPGAs and standard cells, where channels can be made narrower or wider according to the needs of a circuit. The goal of an FPGA architect is to distribute these routing resources in a manner that permits their efficient utilization by the largest class of circuits. If there are too few tracks in some area of the chip then many circuits will be unroutable, while if there are too many tracks they

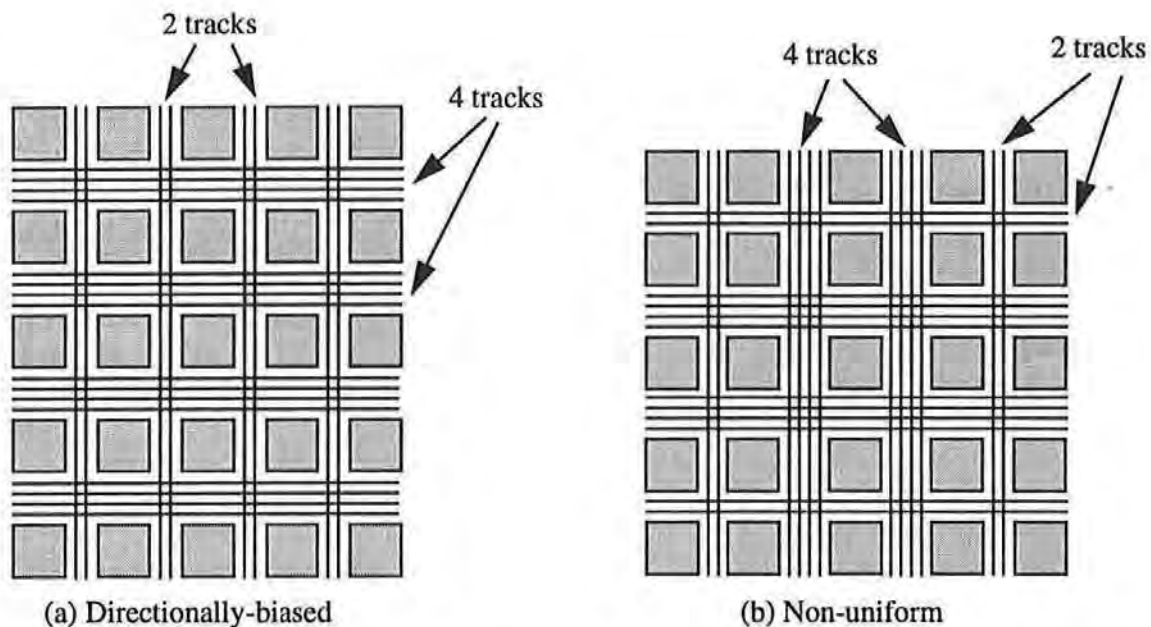


FIGURE 5.1 Global routing architectures: (a) directionally-biased; (b) non-uniform.

will often be wasted. The relatively low density of FPGAs makes an area-efficient global routing architecture essential.

This work addresses two fundamental questions concerning FPGA global routing architectures. First, should the number of tracks in the horizontal channels be different from the number in the vertical channels? Figure 5.1(a) depicts an FPGA with such a *directional bias*. In essence, we are investigating if there is an intrinsic property of circuits that makes a directional bias more area-efficient. If so, what amount of bias is best? Commercial FPGAs with both unbiased routing [4, 5] and biased routing [6, 7] exist, so this question has clear commercial relevance.

Second, should all routing channels in the same direction in an FPGA contain the same number of tracks or is a *non-uniform* routing architecture, in which some channels are wider than others, preferable? An example FPGA with a non-uniform routing architecture is shown in Figure 5.1(b).¹ Intuitively, such an architecture may facilitate routing in congested regions. However, if the wider channels cannot be used efficiently they will waste area. Many in the FPGA community believe that most routing congestion occurs near the center of an FPGA, and therefore expect that wider channels in this region would be beneficial. The Lucent Technologies ORCA 2C

1. Note that any given channel will always have the same number of tracks along its entire length. We did not consider varying the channel capacity along its length as this makes it very difficult, and likely impractical, to lay out the FPGA.

FPGAs employ a non-uniform routing architecture of this type, in which the center channel is wider than the others [141].

On the other hand, board-level constraints often force designers to fix the position of an FPGA's I/Os, and some believe that this increases congestion near the chip edges, and therefore the channel between the pads and the logic block array should be widened. The Xilinx 4000 and 5000 series FPGAs have a wide channel between the pads and logic, at least partially to improve routability when the I/O locations are fixed [11, 135]. In Section 5.4, we determine the best distribution of tracks across an FPGA both when the I/O assignment to pads is unconstrained and when it is fixed in a poor configuration.

5.2 Experimental Methodology

We evaluate FPGA architectures experimentally, using a CAD flow similar to that used in commercial FPGAs to automatically implement circuits. We compare the area-efficiency of different global routing architectures by technology-mapping, placing and globally routing 26 of the larger MCNC benchmark circuits [136] into each architecture. In this section we describe the CAD flow, the area-efficiency metric used to compare architectures, and several important architectural details.

5.2.1 CAD Flow

Figure 5.2 summarizes the CAD flow. First, SIS [142] is used to perform technology-independent logic optimization on a circuit. Next this circuit is technology-mapped by FlowMap [46] into four-input look-up tables (4-LUTs) and registers; the Flowpack post-processing algorithm [46] is then run to further optimize the mapping and reduce the number of LUTs required. The logic block used in these experiments is a single BLE (a 4-LUT plus a register); the structure of a BLE is shown in Figure 3.1(a). VPack packs 4-LUTs and registers together into these logic blocks.

The netlist of logic blocks and a description of the FPGA global routing architecture are then read into our placement and routing tool, VPR. VPR first places the circuit, and then repeatedly globally routes (or attempts to route) the circuit with different numbers of tracks in each channel, or *channel capacities*. The router used in this phase is the routability-driven router described in Section 4.3. VPR performs a binary search on the channel capacities, increasing them after a failed routing and reducing them after a successful one, until it finds the minimum number of tracks required for the circuit to globally route successfully on a given global routing architecture. Note

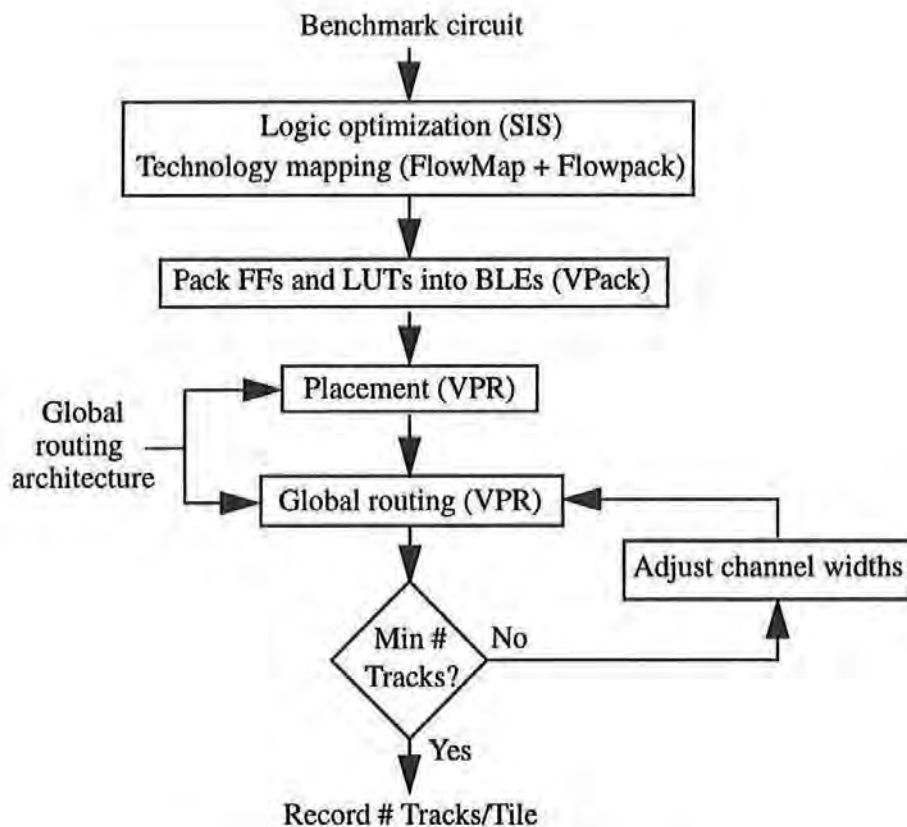


FIGURE 5.2 Architecture evaluation flow.

that while the absolute number of tracks per channel is adjusted upwards or downwards after each attempted routing, the *relative* numbers of tracks in the various channels across the FPGA are always kept at the values specified by the FPGA architecture. For example, VPR's first attempt at routing a circuit in an architecture with a two-to-one directional bias might assume horizontal channel capacities of twelve tracks and vertical channel capacities of six tracks. If this routing was successful, VPR would next attempt to route the circuit in an FPGA with horizontal channel capacities of six tracks and vertical channel capacities of three tracks, and so on until the minimum number of tracks required for routing is determined.

To compare architectures fairly, it is important that the CAD tools properly optimize for each architecture tested. Both the placement and routing phases of VPR take advantage of the biased and non-uniform nature of the different global routing architectures. Section 3.2.3 described the linear congestion cost function used by VPR during placement — this cost function leads to placements in which more congested parts of the design are placed in regions or directions with more routing. As described in Section 4.3.1, in the VPR routability-driven router the cost of a routing resource is a function of the congestion, or amount by which demand exceeds capac-

ity, at that resource. Consequently, our router will automatically act to relieve congestion in narrow channels by re-routing signals through wider channels whenever necessary.

The benchmark circuits used in this study consist of 14 combinational and 12 sequential MCNC benchmark circuits, which vary in size from 222 to 1878 BLEs.

5.2.2 Area-Efficiency Metric

Our goal is to measure the area-efficiency of different global routing architectures without reference to the detailed routing architecture (e.g. segmentation distribution and switch block topology). At this level, it is the amount of “global wiring” that changes as we vary the architecture. A simple track count will not accurately represent the wiring area of rectangular FPGAs, as the tracks in one direction are longer than those in the other. Accordingly, we define a *track segment* to be a prefabricated wire that spans one logic block; a channel of width W tracks that spans L logic blocks contains $W \cdot L$ track segments. The total number of track segments an FPGA must contain to globally route a circuit is a representative metric of the “global wiring” area. In order to average the results from circuits of differing sizes we use the average number of track segments per tile (i.e. per logic block) as our area measure. For example, in a square $M \times M$ uniform FPGA with W tracks in each channel, the total number of track segments is $2W \cdot M^2$, and the number of tracks per tile is $2W$. Note that the routing area is given by the total number of track segments in the entire FPGA, and not the number of track segments which are actually used by a circuit.

5.2.3 Significant FPGA Architectural Details

Several architectural parameters other than the global routing architecture must be specified in order to define an FPGA. We set these parameters to be as close to those of commercial FPGAs as possible.

First, the size of the logic block array used for a given circuit is set to the smallest value with the desired aspect ratio (number of columns / number of rows) with sufficient logic blocks to accommodate the circuit (e.g. 11×11 logic blocks). This situation, in which there is minimal “spare room” in the FPGA, presents the greatest challenge to routing completion and is normally the case manufacturers wish to optimize.

In this study the number of I/O pads that can fit into the height or width of a logic block is set to two. This number is commensurate with the relative sizes of I/O pads

and 4-LUTs in current FPGAs [4, 5, 7] and ensures that none of the 26 benchmarks is pad-limited.

Finally, we do not route the clock net (all the MCNC benchmarks use only a single clock) in sequential circuits, since this net is normally distributed through a special clocking network in commercial FPGAs.

5.3 Experimental Results: Directionally-Biased Routing

The experimental framework and tools described above were employed to answer the questions posed in Section 5.1: first, is there an area-efficiency advantage to a directionally-biased architecture? Recall that in a directionally-biased FPGA the numbers of routing tracks per channel in the horizontal and vertical directions are different. In essence, we are investigating if there is an exploitable directional bias in the basic nature of circuits. We characterize directionally-biased FPGAs by the ratio of the width of a horizontal channel to the width of a vertical channel, denoted as R_h . For example, Figure 5.1(a) depicts an FPGA with a 2:1 directional bias, i.e. $R_h = 2$.

We need to define an additional architectural feature which markedly affects our results: the positioning of the pins on the logic block. The two main cases of interest are illustrated in Figure 5.3. In Figure 5.3(a), the logic block input and output pins are distributed evenly around the entire perimeter of each logic block. We call this the *full-perimeter* pin positioning, and it is similar to the pin positioning used in Xilinx and Lucent Technologies FPGAs [4, 5]. Figure 5.3(b) illustrates the *top/bottom* pin positioning, which restricts the logic block input pin locations to lie only on the top and bottom of the logic block; it is similar to the pin positioning used in Actel FPGAs [6]. In all the results we show in this and subsequent chapters, each logic block pin appears on (i.e. is accessible from) only one side of a logic block. We have found that for the F_c values found in today's commercial FPGAs this leads to the most area-efficient FPGAs [143].

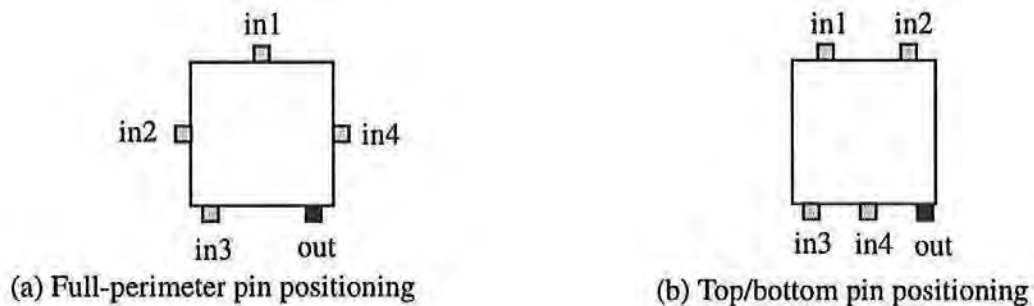


FIGURE 5.3 Logic block pin positions: (a) full-perimeter and (b) top/bottom.

We have also found that the ratio of the number of columns to the number of rows in an FPGA, which we call the logic block array aspect ratio, significantly affects area efficiency. Since most FPGAs have the same number of rows and columns, we first present the results for square (aspect ratio 1) FPGAs, before discussing the more general case of rectangular FPGAs in Section 5.3.2.

5.3.1 Results for Square Logic Block Arrays

Twenty-six large MCNC benchmarks were passed through the experimental flow of Figure 5.2 for values of R_h ranging from 1 to 4. As discussed in Section 5.2, the result for each circuit is the number of track segments per tile needed to successfully global route the circuit in an FPGA¹ with the specified value of R_h . Figure 5.4 is a plot of area-efficiency versus the degree of routing direction bias, R_h , for both types of pin positioning. The vertical axis is the average number of tracks per tile required to successfully route the 26 benchmarks. Table 5.1 summarizes how well each of the seven largest circuits maps into FPGAs with differing amounts of directional bias for both pin positionings; the results for individual circuits closely parallel the overall average shown in Figure 5.4.

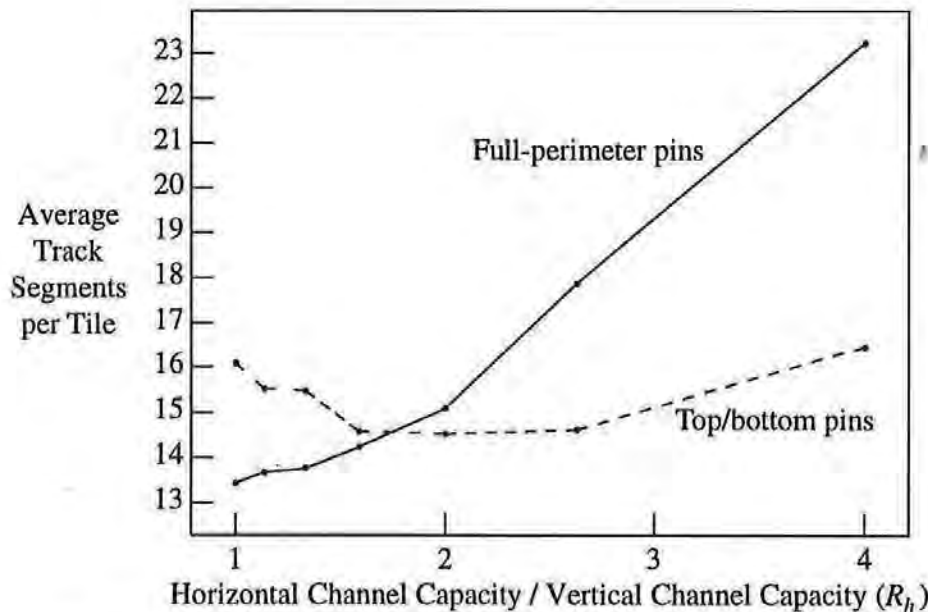


FIGURE 5.4 Area-efficiency vs. directional bias for square FPGAs.

1. Track segments are counted whether or not they are actually used, so this is a true representation of the area that must be devoted to routing in the layout.

TABLE 5.1 Routing requirements of the largest benchmarks vs. directional bias.

Circuit	Number of Logic Blocks	Tracks per Tile Required for Global Routing							
		Full-Perimeter Pins Case				Top/Bottom Pins Case			
		$R_h = 1$	$R_h = 1.33$	$R_h = 2$	$R_h = 4$	$R_h = 1$	$R_h = 1.33$	$R_h = 2$	$R_h = 4$
alu4	1522	16	16	17	28	16	18	17	18
apex2	1878	18	18	20	28	20	19	18	20
apex4	1262	18	18	20	28	22	21	20	23
diffeq	1497	12	11	12	19	14	14	12	13
ex5p	1064	20	21	23	35	22	21	20	28
misex3	1397	16	16	20	31	18	16	17	20
seq	1750	16	18	20	33	20	18	18	23

Figure 5.4 shows that for the full-perimeter logic block pin positioning the best architecture has no directional bias. However, when the pins are restricted to the top and bottom of the logic block, the most efficient architecture has horizontal channels which are roughly twice as thick as the vertical channels. Another important conclusion is that the best full-perimeter architecture is about 8% more area-efficient than the best top/bottom pin architecture.

The full-perimeter architecture is more area-efficient because there is a greater chance that the block input pins are closer to their desired connections when they are in this configuration than when they are in the top/bottom configuration. For example, consider the two routings of a multi-terminal net shown in Figure 5.5. The top/bottom

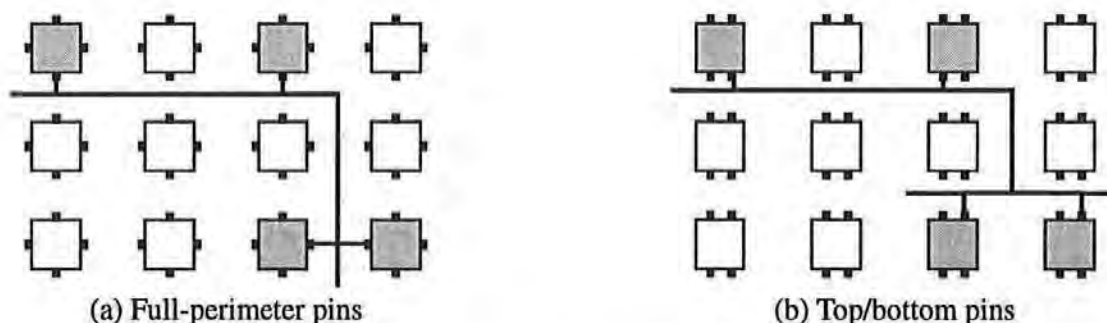


FIGURE 5.5 Example multi-terminal net routing with (a) full-perimeter and (b) top/bottom pins.

pin configuration needs six track segments to route this net, while the full-perimeter configuration requires only five. By making use of the functional equivalence of LUT input pins during routing, the router can often connect to a logic block pin adjoining a track segment it needs to use for other connections, essentially making the connection to this logic block for free. Since the top/bottom pin configuration has input pins bordering on only the horizontal channels, such “free” connections into logic blocks are less frequent, reducing area-efficiency.

The full-perimeter pins configuration achieves the highest area-efficiency when there is no directional bias to the routing because this makes the difficulty of routing to each of a logic block’s nearest neighbors roughly equal. Consequently, the placement software can use all the nearby logic block locations equally to cluster the fanout of a net around its driver. Essentially, this allows one to cluster tightly coupled portions of logic in the smallest possible area. The top/bottom pins configuration, on the other hand, prefers a 2:1 directional bias because every connection to a logic block pin must come from a horizontal channel. This extra pressure on the horizontal routing resources is significant, since the average distance routed between pins is only about 3 track segments.

5.3.2 Results for Rectangular Logic Block Arrays

We call any logic block array in which the number of rows does not equal the number of columns *rectangular*. The basic reason to use a rectangular logic block array is to change the aspect ratio of the resulting FPGA die. For example, if the layout of the basic FPGA tile is not square, an FPGA manufacturer may want to create a square die by using more columns than rows, or vice versa. On the other hand, FPGA manufacturers may want to increase the I/O-to-logic ratio by creating a rectangular FPGA die, as this increases the die perimeter and hence the number of pads. In this case, if the basic tile layout is square an FPGA manufacturer may use a rectangular logic block array to create a rectangular FPGA. We refer to the ratio of the number of columns in an FPGA to the number of rows as its logic block array aspect ratio. Figure 5.6 depicts an FPGA with a logic block array aspect ratio of two.

When the logic block array is rectangular, the channels in one direction are longer and have more blocks connected to them than the orthogonal channel, so the best amount of directional bias may change. Figure 5.7 is a plot of the required tracks per tile (averaged over the 26 circuits) versus R_h for various logic block array aspect ratios with the full-perimeter logic block pin positioning. There are two features of interest in Figure 5.7. First, notice that the minimum of the aspect ratio = 1 curve is the lowest of the three, indicating that a square logic block array is the most area-efficient. Secondly, the value of R_h at which the minimum area occurs increases as the aspect ratio increases. As the aspect ratio increases, the horizontal channels become longer

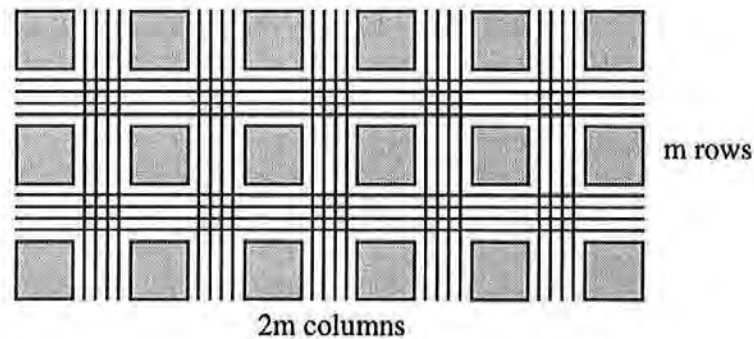


FIGURE 5.6 An FPGA with a logic block array aspect ratio of 2.

than the vertical channels and this results in greater demand for horizontal track segments. The best value of R_h increases from 1 for a square logic block array to 1.33 and 1.59 for aspect ratios of 2 and 3, respectively.

The solid curve in Figure 5.8 shows how area-efficiency varies with logic block array aspect ratio when we set R_h to the best value for each aspect ratio. The dotted curve in Figure 5.8 keeps R_h fixed at 1, which is the best value for a square logic block array. The routing resource requirements increase moderately with aspect ratio; an FPGA with a logic block array aspect ratio of 3 requires 18% more tracks per tile than an FPGA with a square logic block array when R_h is 1. When the most appropriate value of R_h is used for each aspect ratio, however, an FPGA with an aspect ratio of 3 requires only 4% more track segments than an FPGA with a square logic block array.

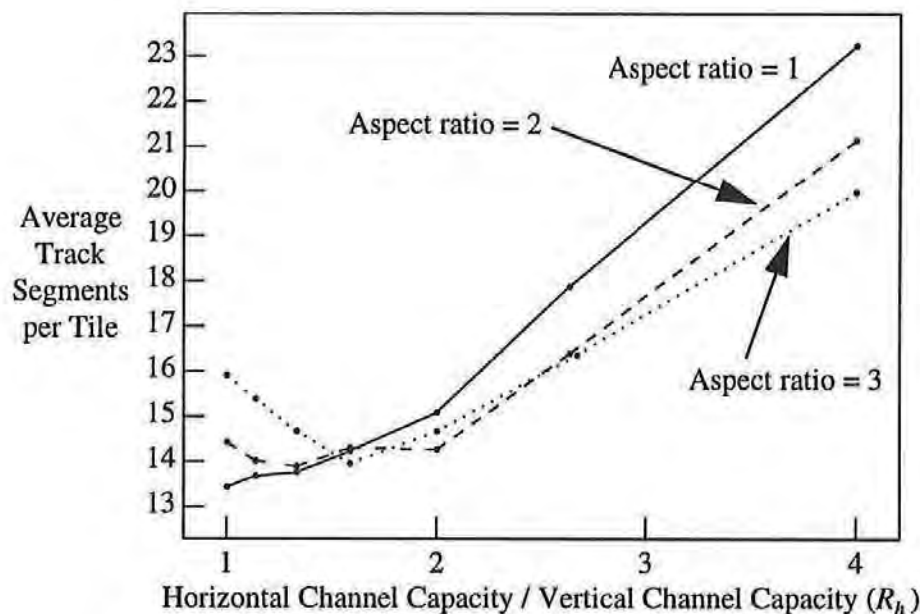


FIGURE 5.7 Area-efficiency of rectangular FPGAs with full-perimeter pins.

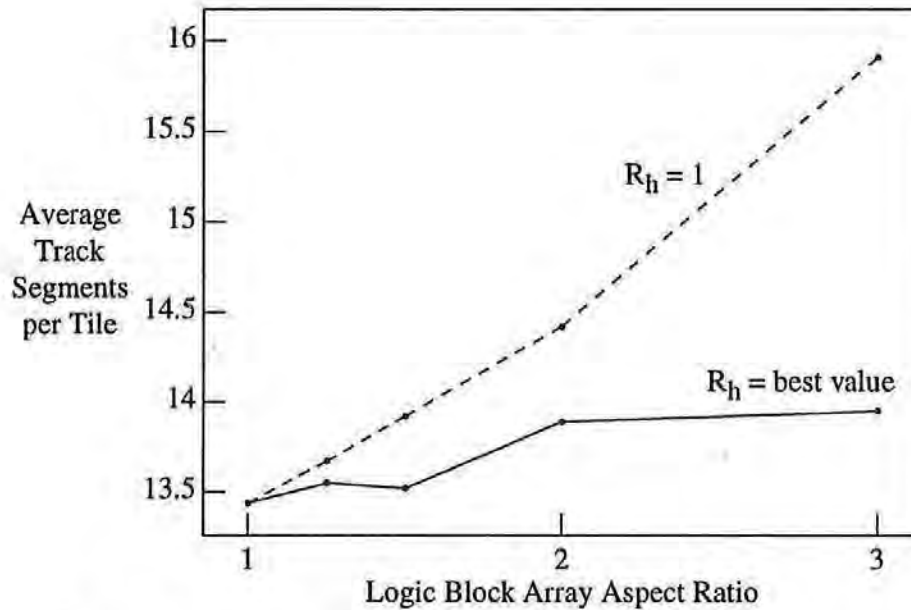


FIGURE 5.8 Area-efficiency vs. array aspect ratio for FPGAs with full-perimeter pins.

Thus we conclude that, as long as the horizontal and vertical channel widths are appropriately balanced, the logic block array aspect ratio can be increased with little impact on the core area.

The variation of routing area with logic block array aspect ratio is similar for FPGAs that use the top/bottom logic block pin positioning. In this case an FPGA with a logic block array aspect ratio of 3 requires only 5% more tracks per tile than an FPGA with a square logic block array. For FPGAs of this type, however, the increase in the best value of R_h as aspect ratio increases is less dramatic. The best square-array FPGA with top/bottom pins has horizontal channels which are twice as wide as vertical channels; the thicker horizontal channels are better able to cope with the increased pressure for horizontal tracks as aspect ratio increases.

5.4 Experimental Results: Non-Uniform Routing

The second key issue we explore concerns the area-efficiency obtained when the channels in different regions of an FPGA have different capacities. We only investigate FPGAs which use the full-perimeter pin positioning, as Section 5.3 showed that this pin positioning is best.

We define a non-uniform routing architecture to be one in which the number of tracks per channel changes from channel to channel across an FPGA. For example, Figure 5.1(b) illustrates a non-uniform FPGA in which the channels near the chip center are wider than those near the periphery. If congested regions of a circuit can be localized and placed in the portions of the FPGA with the widest channels, a non-uniform FPGA could have better area efficiency than a uniform FPGA. We will investigate three types of non-uniform FPGAs in which we vary the center/edge channel capacity ratio, the capacity of only the center channel, and the I/O channel capacity, respectively.

5.4.1 Center/Edge Capacity Ratio

Many in the FPGA community believe that most congestion occurs in the center of FPGAs, and hence expect having wider channels near the FPGA center and narrower channels near the edges to improve area-efficiency. To keep the layout problem tractable, we restrict ourselves to FPGAs which use channels of only two different widths. We can describe global routing architectures of this form with two parameters. Let R_w be the ratio of the widths of the channels near the center of the FPGA to the widths of the channels near the FPGA edges, i.e. $W_{\text{center}} / W_{\text{edge}}$. Let R_c be the ratio of the number of channels with width W_{center} to the total number of channels. For example, the FPGA of Figure 5.1(b) has $R_w = 2$ and $R_c = 0.5$.

Using the flow of Section 5.2, we again implemented 26 benchmark circuits in different architectures to determine their area-efficiency. We examined FPGAs with R_w

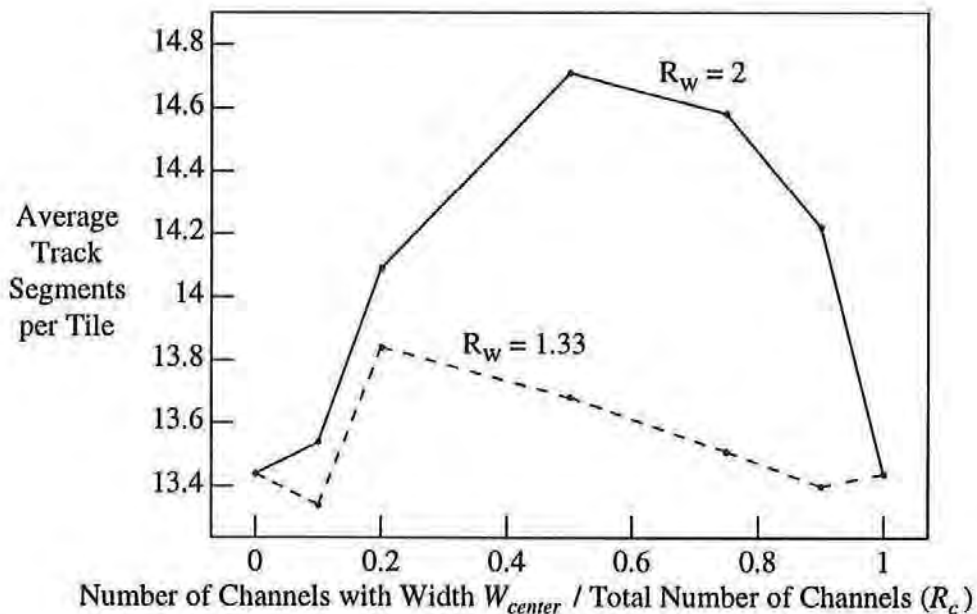


FIGURE 5.9 Area-efficiency vs. non-uniform routing architecture.

equal to 0.75, 1.18, 1.33, and 2, and with R_c values varying from 0 to 1. The relative density of FPGAs with $R_w = 1.33$ and $R_w = 2$ is summarized in Figure 5.9, which plots the average number of tracks per tile required by the 26 benchmarks in each architecture. Note that the points at which R_c equals 0 or 1 correspond to a *uniform* FPGA, in which all channels have the same capacity. The area required by each of the seven largest benchmarks in several representative architectures is listed in Table 5.2; again the results for individual circuits generally match the overall average, although there is some circuit-dependent behaviour. Note that the number of tracks per tile required for routing individual circuits in a non-uniform architecture is generally not an integer. Since some portions of the FPGA have wider channels than others, the number of tracks per tile is an average over the various tiles such an FPGA must contain.

TABLE 5.2 Track segments / tile required for global routing by the largest benchmark circuits vs. non-uniform architecture.

Circuit	Uniform ($R_w = 1$)	$R_w = 1.33$			$R_w = 2$		
		$R_c = 0.2$	$R_c = 0.5$	$R_c = 0.75$	$R_c = 0.2$	$R_c = 0.5$	$R_c = 0.75$
alu4	16	14.7	15.9	16.8	16.0	16.6	15.6
apex2	18	16.8	17.9	18.8	19.2	23.5	18.9
apex4	18	17.1	17.8	18.7	18.6	22.4	20.1
diffeq	12	12.8	13.8	11.4	14.4	13.6	15.6
ex5p	20	21.1	20.8	20.2	20.8	19.6	20.5
misex3	16	16.7	17.9	14.8	16.5	16.9	16.9
seq	16	16.7	18.0	16.9	18.6	19.9	17.2

The results generally show that the less uniform the channel widths, the worse the FPGA area-efficiency. The worst area-efficiency with $R_w = 2$ occurs when R_c is 0.5, meaning that half the FPGA channels are twice as wide as the other half. In fact, only two non-uniform FPGAs show even marginal area-efficiency improvements over the uniform case and both these FPGAs are very close to a uniform architecture. In one, the 10% of channels nearest the center are 33% wider than the other channels, while in the other the 90% of channels closest to the center are 33% wider than the channels nearest the edges. The reduction in tracks per tile over a uniform FPGA is less than 1% for both of these FPGAs; this marginal improvement is certainly not enough to justify the extra layout effort required in the physical design of such an FPGA.

These results are significant because many FPGA architects believe that there should be considerable benefit to these kinds of non-uniform architectures. The fundamental reason they do not show any benefit is that there is not much more congestion in the center of an FPGA than there is near its edges. In order to determine the “natural” routing demand distribution of circuits, we placed and routed the 26 benchmark circuits with all congestion avoidance features disabled, so that placement minimized wirelength and the router connected each net by the shortest path. Figure 5.10 plots the maximum and average number of tracks required by the horizontal channels as a function of the channel position within the FPGA, averaged over the 26 benchmark circuits. Demand for routing tracks is relatively constant over the middle 90% of the FPGA, and there is only a moderate decrease as one gets very close to the chip edges. Figure 5.11 plots the average and maximum tracks required by each horizontal channel versus the channel position for three representative benchmark circuits. There is some high-frequency variation from channel to channel, since the router is, in this case, not making any effort to route nets around congestion. Nevertheless, it is clear that these circuits closely mirror the behavior of the overall averages of Figure 5.10.

An additional reason for the poor area-efficiency of FPGAs with extra routing near their center is that typical circuits contain numerous local congestion “hotspots” (small regions where all the channels are full) and some of these hotspots occur quite close to the FPGA edge. Consequently, in order for an FPGA with thicker channels near its center to use fewer routing resources, the placement software must move all of these hotspots into the FPGA center. As discussed in Section 3.2.3, we spent con-

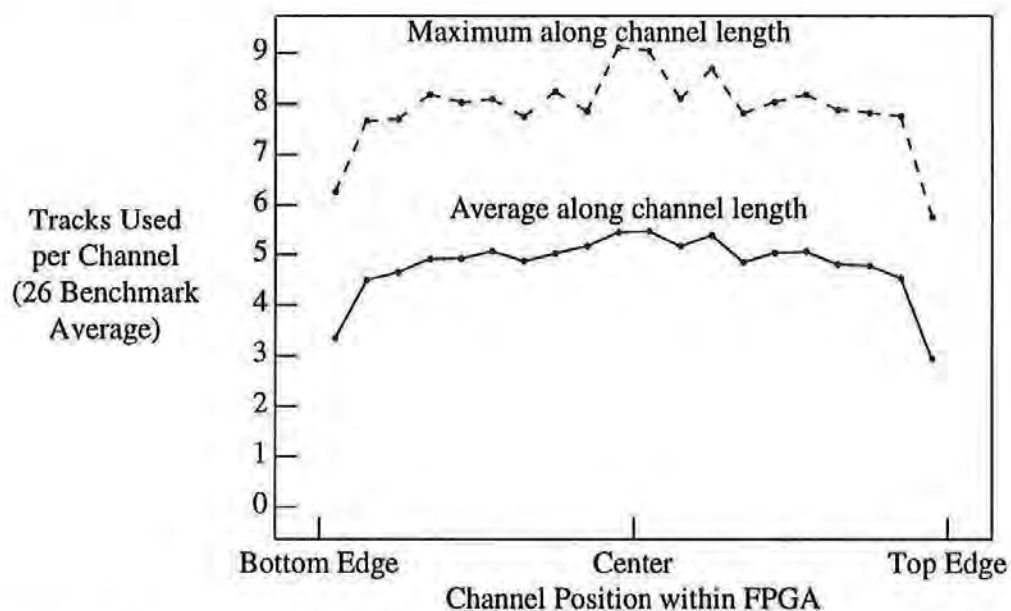


FIGURE 5.10 Average over benchmarks of track demand vs. position for horizontal channels.

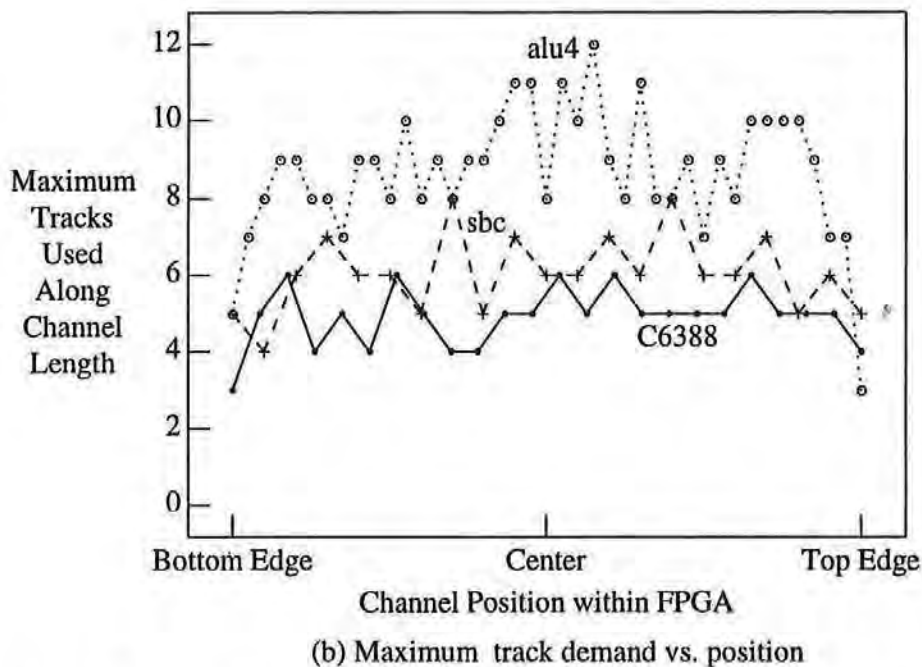
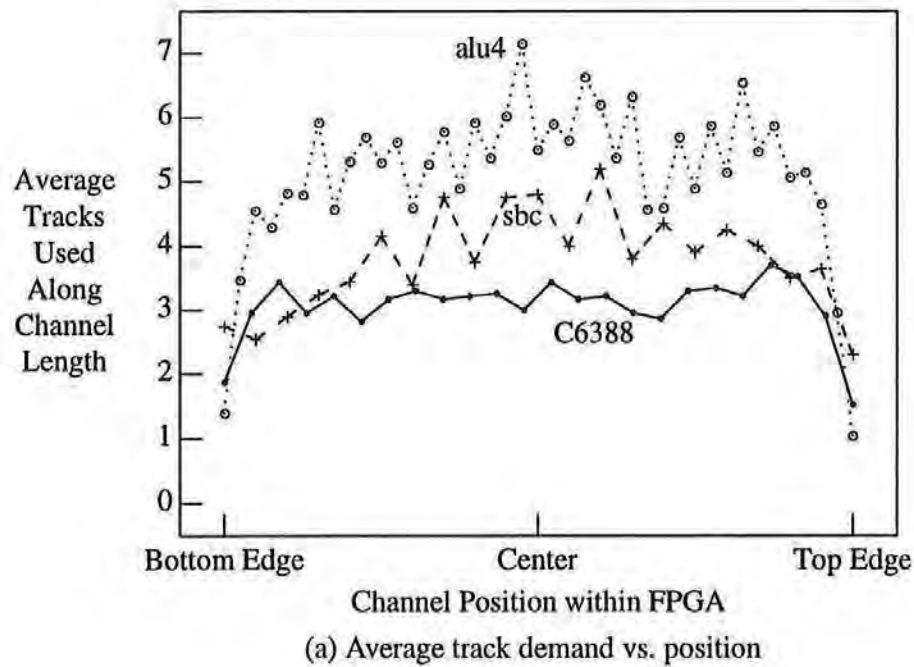


FIGURE 5.11 (a) Average and (b) maximum track demand vs. channel position for three circuits.

siderable time investigating placement cost functions that modelled congestion well. Of the cost functions we examined, the non-linear congestion cost function (see Section 2.2.2) performs the most detailed congestion modelling, at the cost of a very high CPU time requirement. When compared to the linear congestion cost function that

we used to generate the results of this chapter, the non-linear congestion cost function improved the performance of a uniform FPGA slightly more than it improved the performance of the non-uniform FPGAs tested. We believe it is therefore more effective for CAD tools to attempt to spread out congestion as much as possible, rather than to try to localize it to a designated portion of a chip.

Engineers with MPGA or standard cell design experience may find it surprising that having wider channels near the center of an FPGA doesn't improve area-efficiency. Since standard cell and sea-of-gates Mask-Programmed Gate Array (MPGA) designs often have some channels wider than others, one might expect that FPGA-based designs would also benefit from a mix of wide and narrow channels. There is a key difference between FPGAs and standard cells or MPGAs, however — in FPGAs the width of the various channels is determined by the FPGA architecture, not the circuit being implemented. If one builds an FPGA with some channels wider than others, *all* circuits are forced to route most of their connections through the predefined wide channels, whether it is efficient for a particular circuit to do so or not. In a standard cell or sea-of-gates MPGA design, on the other hand, a channel is made extra wide or extra narrow only if the circuit being implemented naturally demands it.

5.4.2 Single Center Channel

In an effort to improve routability, Lucent Technologies, has introduced an FPGA in which the center channel in each direction is extra-wide [141]. We define R_m to be the ratio of the width of these center channels to the width of the other channels. Figure 5.12 depicts an FPGA with this extra routing in its center.

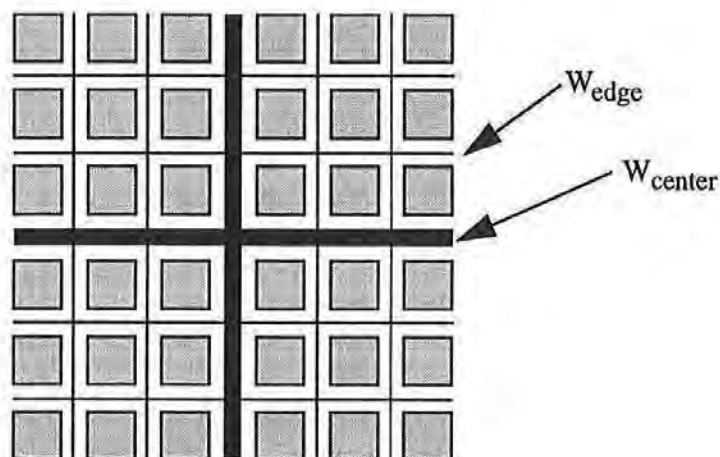


FIGURE 5.12 An FPGA with an extra-wide center channel.

The solid and dashed lines in Figure 5.13 show how area-efficiency varies with R_m for this type of FPGA when a linear congestion cost function and when a bounding box cost function are used during placement, respectively. The data show that the most area-efficient FPGA is one without an extra wide channels in the middle — i.e. $R_m = 1$. There is a sharp dip in the number of tracks/tile required at $R_m = 2$ when the linear congestion cost function is used, indicating an FPGA with area-efficiency almost as good as one with $R_m = 1$. This dip occurs at the first point at which the linear congestion cost function considers the cost of routing through narrow channels to connect two adjacent blocks to have the same cost as connecting two blocks separated by one intervening block through the extra wide channel. Consequently, the placer is able to make better use of the extra-wide channel at this point. The bounding box cost function leads to worse area-efficiency than the linear congestion cost function and the dip in required area near $R_m = 2$ does not occur. As with the non-uniform FPGAs of Figure 5.9 then, the best results are obtained by spreading extra routing resources over the entire FPGA rather than by adding them to only one region.

Aside from trying to create a more area-efficient FPGA, there is an alternate reason for making the center channel of an FPGA wider. FPGA manufacturers tend to create several FPGAs with the same basic architecture but with different numbers of logic blocks in order to appeal to customers with different capacity needs. As the number of logic blocks in a circuit increases, the demand for routing also increases [144], so at some point the channels should be widened. However, widening the channels for FPGAs with more logic blocks requires redoing the layout of the basic tile, which involves considerable time and expense. Since the center of an FPGA typically con-

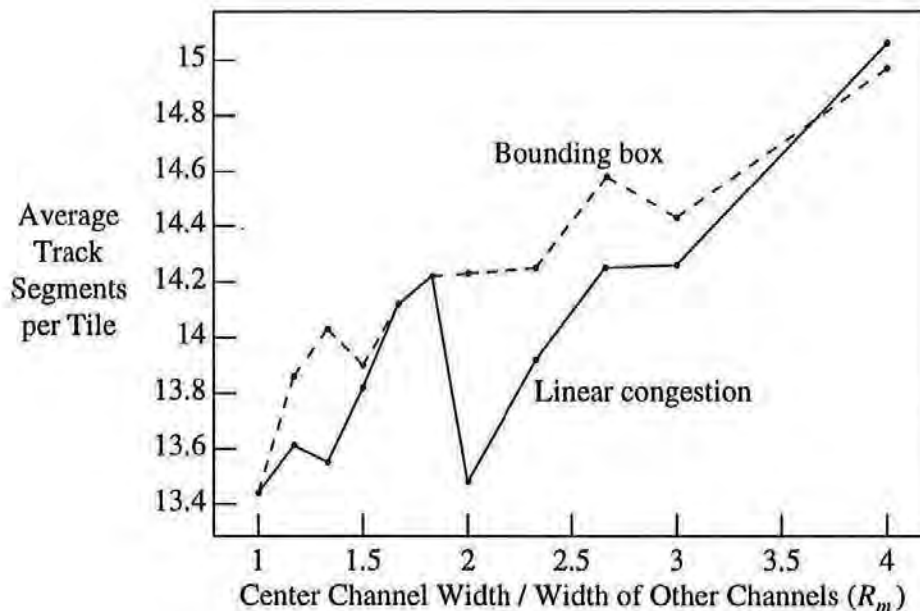


FIGURE 5.13 Effectiveness of an extra-wide center channel.

tains the programming logic, this area already requires its own custom layout, so extra routing tracks can be added to it with relatively little effort.

We found, however, that in most cases adding tracks to the center channel did not result in any significant reduction in routing pressure on the other channels. The only exceptions were FPGAs with R_m near two. When circuits were mapped into such FPGAs by a placer using the linear congestion cost function the routing pressure on the other channels was reduced. It is relatively difficult to improve the routability of an FPGA simply by adding tracks to the center channel because the placement tool must now move all portions of the circuit that require more routing than the “normal” channels can provide close to the center channel so that they can make use of the extra routing available there.

5.4.3 I/O Channel

Xilinx has added extra routing resources to the “I/O-channel” that runs between the I/O pads and the logic blocks, at least in part to ensure that fixed or “locked” I/O pad placement does not impact routability and speed [11]. We define R_{IO} to be the ratio of the width of this outermost channel to the width of the other channels. For example, Figure 5.14 depicts an FPGA with R_{IO} equal to 0.5. In the experiments of this section, all the channels running between logic blocks have the same width, W_{Logic} , so R_{IO} completely describes the global routing architecture.

Figure 5.15 is a plot of the average track segments per tile required for the 26 benchmark circuits versus R_{IO} . The solid line in Figure 5.15 shows the trend when the I/O locations are chosen by the placement tool, while the dashed line is found when the

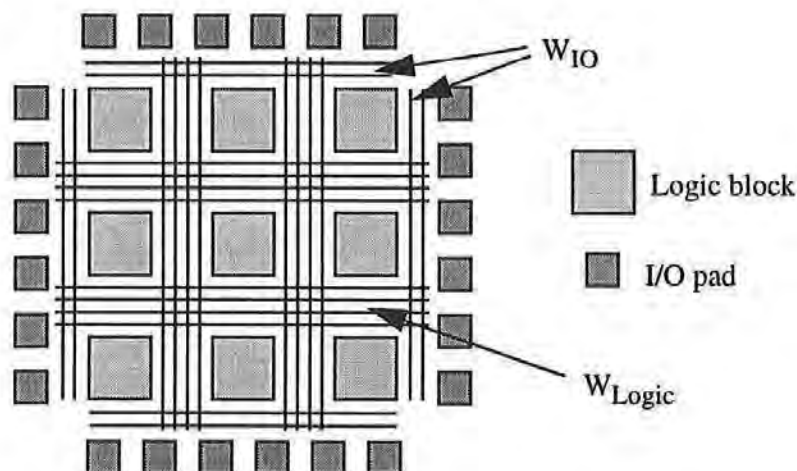


FIGURE 5.14 An FPGA with $R_{IO} = 0.5$.

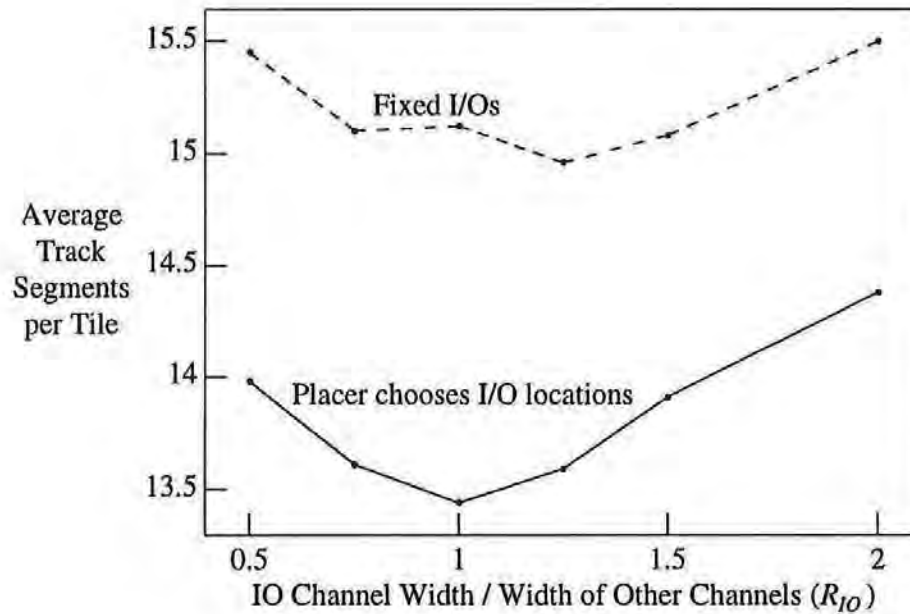


FIGURE 5.15 Routability vs. I/O channel width.

I/O pads are “fixed” in a random location, to model the effect of poor (from the FPGA’s point of view) pin constraints.

There are several features of interest in Figure 5.15. First notice that fixing the I/O locations increases the number of routing tracks required by 12% on average. Architects must take this into account when designing FPGAs. Secondly, the curve where the I/O locations are chosen by the placement tool has its minimum value when $R_{IO} = 1$, again showing that it is best to spread routing resources evenly across the chip. Fixing the I/O pins shifts the minimum in the tracks per tile curve slightly so that it now occurs when $R_{IO} = 1.25$. While fixing the I/O pins leads to a significant increase in the number of routing tracks required, this increase is, for the most part, spread over the FPGA and not confined to the channels connecting to the I/O pads. Consequently, one should not make very wide channels adjoining the pads in order to improve routability with pin constraints, although a small increase in the I/O channel capacity is a net benefit.

In interpreting the best values of R_{IO} we found, one should remember that in the FPGAs we consider the number of pins connecting to an I/O channel is roughly the same as the number of pins connecting to a logic channel. In architectures with different ratios of I/O pad to 4-LUT area, the best R_{IO} values will change as the ratio of the number of pins bordering a logic channel to the number bordering an I/O channel changes.

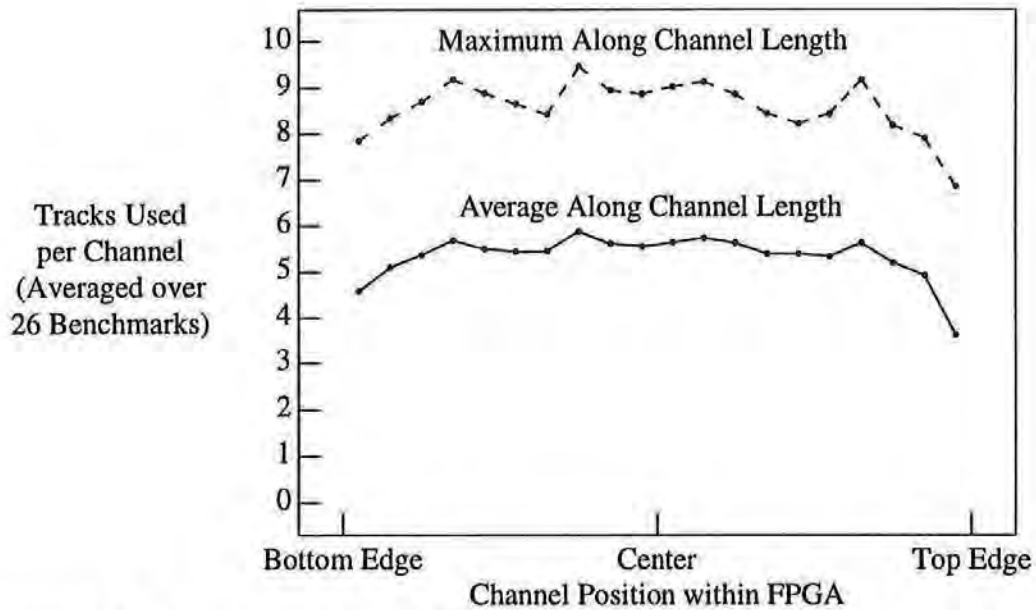
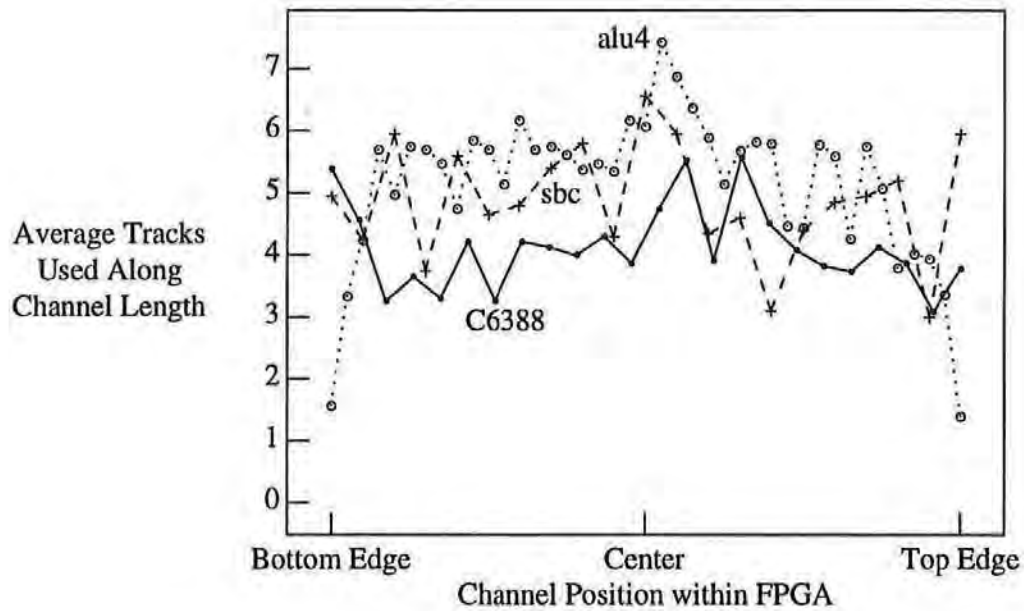
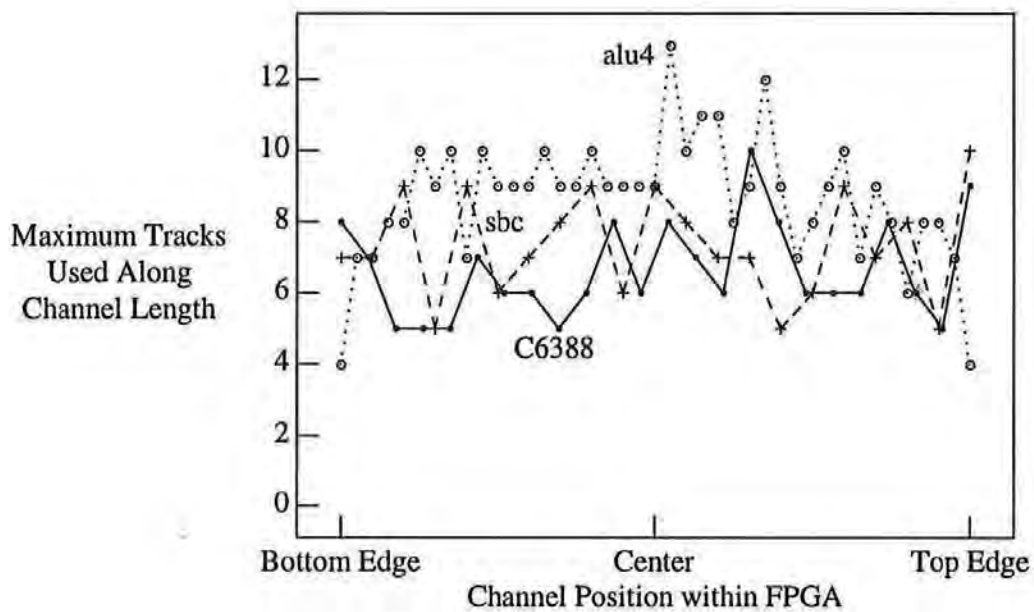


FIGURE 5.16 Average over benchmarks of horizontal track demand when I/Os are fixed in a random configuration.

In order to determine how the “natural” demand for tracks is altered when the I/O locations of a circuit are fixed in a poor configuration, we repeated the congestion-oblivious placement and routing experiments described in Section 5.4.1 with the I/O locations fixed in a random configuration. Figure 5.16 plots the maximum and average number of tracks required by the horizontal channels as a function of the channel position within the FPGA, averaged over the 26 benchmark circuits. Comparing with the corresponding curve obtained with movable I/Os (Figure 5.10), one sees that the curves have shifted up by approximately half a track, and that the drop off in track demand near the chip edges is less pronounced. Figure 5.17 shows how the “natural” track demand of three typical circuits vary with channel position. By comparing with Figure 5.11, one sees that the curve for *alu4* has changed little, while the *sbc* and *C6388* curves have each shifted up by approximately a track and show significantly more demand for routing tracks near the chip edges than they did when the I/Os were movable. This is due to the different I/O to logic ratios of these three circuits. *Alu4* has very few I/Os; it uses only 7% of the I/O pads available in the FPGA to which it is mapped. *C6388* and *sbc*, on the other hand, have considerably more I/O, and use 35% and 61% of the I/O locations available to them, respectively. As one would expect, then, fixing I/O locations has little effect on circuits with few I/Os. On the other hand, circuits with larger I/O requirements show an increase in routing track demand across the entire FPGA, with the greatest increase near the chip edges.



(a) Average track demand vs. position



(b) Maximum track demand vs. position

FIGURE 5.17 (a) Average and (b) maximum track demand vs. channel position for three circuits when I/Os are fixed in a random configuration.

5.5 Summary

The most interesting (and unexpected) result in this chapter is that the most area-efficient global routing structure is one with completely uniform channel capacities across the entire chip and in both horizontal and vertical directions. The basic reason is that most circuits “naturally” tend to have routing demands which are evenly spread across an FPGA. The only (slight) exception we found to this “uniform is better” rule occurred when the I/O locations of circuits were fixed by board-level constraints. In this case making the I/O channel 25% wider than the other channels was a net benefit.

Of almost equal note, the area-efficiency is decreased only slightly by some non-uniform or directionally-biased architectures, provided the pin placement on the logic blocks is well-matched to the channel capacity distribution. Hence if such architectures are desirable for other reasons the impact on core area doesn’t preclude their use.

More specifically, of the FPGA architectures studied, a full-perimeter pin position FPGA with no directional routing bias and uniform channel widths is most area-efficient. Employing a logic block with the top/bottom pin position requires approximately 8% more routing resources than full-perimeter FPGAs, and the most area-efficient top/bottom FPGA has twice as many horizontal routing tracks as vertical ones. We also found that one can construct FPGAs with rectangular logic block arrays which are only slightly less dense than square arrays provided one adjusts the degree of directional bias in the routing resources to best match the array aspect ratio.

Our experimental results in this chapter were gathered using VPR’s linear congestion cost function during placement because we felt the non-linear congestion cost function was too slow to be commercially viable. However, it is interesting to note that while the non-linear congestion cost function slightly improved the routability of circuits for all FPGA architectures, it improved routability the most for uniform routing architectures. Apparently it is easier for advanced CAD tools to spread out congested regions than it is to localize them to designated portions of a chip that have extra routing resources. Consequently, we expect that future advances in CAD tools will tend to slightly increase the advantages of uniform routing architectures over their non-uniform counterparts.