



The Design Warrior's Guide to FPGAs

Devices, Tools and Flows



Mentor
Graphics
XILINX



CD-ROM included
Containing a
portable edition
version of the
text plus valuable
additional tools

Clive "Max" Maxfield



The Design Warrior's Guide to FPGAs

Devices, Tools and Flows

This book belongs to Lee Bernstein
(rachel.girts@haynesboone.com)

Copyright Elsevier 2023

The Design Warrior's Guide to FPGAs

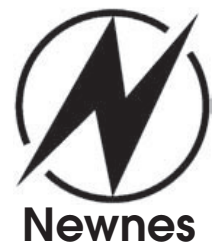
Devices, Tools and Flows

Clive “Max” Maxfield



AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Newnes is an imprint of Elsevier



Newnes is an imprint of Elsevier

200 Wheeler Road, Burlington, MA 01803, USA

Linacre House, Jordan Hill, Oxford OX2 8DP, UK

Copyright © 2004, Mentor Graphics Corporation and Xilinx, Inc.
All rights reserved.

Illustrations by Clive “Max” Maxfield

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.

Permissions may be sought directly from Elsevier’s Science & Technology Rights Department in Oxford, UK: phone: (+44) 1865 843830, fax: (+44) 1865 853333, e-mail: permissions@elsevier.com.uk. You may also complete your request on-line via the Elsevier homepage (<http://elsevier.com>), by selecting “Customer Support” and then “Obtaining Permissions.”



Recognizing the importance of preserving what has been written, Elsevier prints its books on acid-free paper whenever possible.

Library of Congress Cataloging-in-Publication Data

Application submitted

ISBN: 0-7506-7604-3

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library.

For information on all Newnes publications
visit our Web site at www.newnespress.com

04 05 06 07 08 09 10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

*To my wife Gina—the yummy-scrummy caramel, chocolate fudge,
and rainbow-colored sprinkles on the ice cream sundae of my life*

*Also, to my stepson Joseph and my grandchildren Willow, Gaige, Keegan, and Karma,
all of whom will be tickled pink to see their names in a real book!*

For your delectation and delight, the CD accompanying this book contains a fully-searchable copy of *The Design Warrior's Guide to FPGAs* in Adobe® Acrobat® (PDF) format. You can copy this PDF to your computer so as to be able to access *The Design Warrior's Guide to FPGAs* as required (this is particularly useful if you travel a lot and use a notebook computer).

The CD also contains a set of Microsoft® PowerPoint® files—one for each chapter and appendix—containing copies of the illustrations that are festooned throughout the book. This will be of particular interest for educators at colleges and universities when it comes to giving lectures or creating handouts based on *The Design Warrior's Guide to FPGAs*.

Last but not least, the CD contains a smorgasbord of datasheets, technical articles, and useful web links provided by Mentor and Xilinx.

Contents

Preface ix

Acknowledgments xi

Chapter 1

Introduction 1

What are FPGAs? 1
Why are FPGAs of interest? 1
What can FPGAs be used for? . . . 4
What's in this book? 6
What's not in this book? 7
Who's this book for? 8

Chapter 2

Fundamental Concepts 9

The key thing about FPGAs. 9
A simple programmable function . 9
Fusible link technologies 10
Antifuse technologies 12
Mask-programmed devices 14
PROMs 15
EPROM-based technologies 17
EEPROM-based technologies . . . 19
FLASH-based technologies 20
SRAM-based technologies 21
Summary 22

Chapter 3

The Origin of FPGAs. 25

Related technologies 25
Transistors 26
Integrated circuits 27

SRAMs, DRAMs, and
microprocessors. 28
SPLDs and CPLDs 28
ASICs (gate arrays, etc.) 42
FPGAs 49

Chapter 4

Alternative FPGA Architectures 57

A word of warning 57
A little background information . 57
Antifuse versus SRAM
versus 59
Fine-, medium-, and coarse-grained
architectures 66
MUX- versus LUT-based
logic blocks 68
CLBs versus LABs versus slices. . 73
Fast carry chains 77
Embedded RAMs. 78
Embedded multipliers, adders,
MACs, etc. 79
Embedded processor cores
(hard and soft) 80
Clock trees and clock managers . 84
General-purpose I/O 89
Gigabit transceivers 92
Hard IP, soft IP, and firm IP . . . 93
System gates versus real gates . . 95
FPGA years. 98

Chapter 5**Programming (Configuring)****an FPGA 99**

Weasel words 99

Configuration files, etc. 99

Configuration cells 100

Antifuse-based FPGAs 101

SRAM-based FPGAs 102

Using the configuration port . . 105

Using the JTAG port 111

Using an embedded processor. . 113

Chapter 6**Who Are All the Players?. . . 115**

Introduction. 115

FPGA and FPAA vendors . . . 115

FPNA vendors 116

Full-line EDA vendors 116

FPGA-specialist and independent EDA
vendors 117FPGA design consultants with special
tools 118Open-source, free, and low-cost design
tools 118**Chapter 7****FPGA Versus ASIC****Design Styles. 121**

Introduction. 121

Coding styles 122

Pipelining and levels of logic . . 122

Asynchronous design practices . 126

Clock considerations 127

Register and latch considerations 129

Resource sharing (time-division multi-
plexing) 130

State machine encoding 131

Test methodologies 131

Chapter 8**Schematic-Based Design Flows 133**

In the days of yore. 133

The early days of EDA 134

A simple (early) schematic-driven
ASIC flow. 141A simple (early) schematic-driven
FPGA flow 143Flat versus hierarchical schematics
. 148Schematic-driven FPGA
design flows today. 151**Chapter 9****HDL-Based Design Flows . . . 153**Schematic-based flows
grind to a halt 153

The advent of HDL-based flows 153

Graphical design entry lives on. 161

A positive plethora of HDLs . . 163

Points to ponder. 172

Chapter 10**Silicon Virtual Prototyping****for FPGAs 179**

Just what is an SVP? 179

ASIC-based SVP approaches . . 180

FPGA-based SVPs 187

Chapter 11**C/C++ etc.-Based Design Flows****. 193**Problems with traditional
HDL-based flows 193C versus C++ and concurrent
versus sequential 196

SystemC-based flows 198

Augmented C/C++-based flows 205

Pure C/C++-based flows 209

Different levels of synthesis
abstraction 213

Mixed-language design and verification environments . . .	214
Chapter 12 DSP-Based Design Flows . . .	217
Introducing DSP	217
Alternative DSP implementations	218
FPGA-centric design flows for DSPs.	225
Mixed DSP and VHDL/ Verilog etc. environments . . .	236
Chapter 13 Embedded Processor-Based Design Flows	239
Introduction.	239
Hard versus soft cores	241
Partitioning a design into its hardware and software components 245	
Hardware versus software views of the world	247
Using an FPGA as its own development environment . . .	249
Improving visibility in the design	250
A few coverification alternatives	251
A rather cunning design environment	257
Chapter 14 Modular and Incremental Design	259
Handling things as one big chunk	259
Partitioning things into smaller chunks	261
There's always another way . . .	264
Chapter 15 High-Speed Design and Other PCB Considerations	267
Before we start.	267
We were all so much younger then	267
The times they are a-changing .	269
Other things to think about . . .	272
Chapter 16 Observing Internal Nodes in an FPGA	277
Lack of visibility.	277
Multiplexing as a solution . . .	278
Special debugging circuitry . . .	280
Virtual logic analyzers.	280
VirtualWires	282
Chapter 17 Intellectual Property	287
Sources of IP	287
Handcrafted IP	287
IP core generators	290
Miscellaneous stuff	291
Chapter 18 Migrating ASIC Designs to FPGAs and Vice Versa	293
Alternative design scenarios . . .	293
Chapter 19 Simulation, Synthesis, Verification, etc. Design Tools.	299
Introduction.	299
Simulation (cycle-based, event-driven, etc.)	299
Synthesis (logic/HDL versus physically aware)	314
Timing analysis (static versus dynamic).	319
Verification in general	322
Formal verification	326

Miscellaneous 338

Chapter 20

Choosing the Right Device . . 343

So many choices 343

If only there were a tool. 343

Technology 345

Basic resources and packaging . 346

General-purpose I/O interfaces . 347

Embedded multipliers,
RAMs, etc. 348

Embedded processor cores. . . . 348

Gigabit I/O capabilities 349

IP availability 349

Speed grades. 350

On a happier note. 351

Chapter 21

Gigabit Transceivers 353

Introduction. 353

Differential pairs 354

Multiple standards 357

8-bit/10-bit encoding, etc. . . . 358

Delving into the transceiver
blocks 361

Ganging multiple transceiver
blocks together 362

Configurable stuff 364

Clock recovery, jitter, and
eye diagrams. 367

Chapter 22

Reconfigurable Computing. . 373

Dynamically reconfigurable logic 373

Dynamically reconfigurable
interconnect 373

Reconfigurable computing . . . 374

Chapter 23

Field-Programmable Node Arrays 381

Introduction. 381

Algorithmic evaluation 383

picoChip's picoArray technology 384

QuickSilver's ACM technology 388

It's silicon, Jim, but not
as we know it! 395

Chapter 24

Independent Design Tools . . 397

Introduction. 397

ParaCore Architect 397

The Confluence system
design language 401

Do you have a tool? 406

Chapter 25

Creating an Open-Source-Based Design Flow 407

How to start an FPGA design
shop for next to nothing . . . 407

The development platform: Linux
. 407

The verification environment . 411

Formal verification 413

Access to common IP components
. 416

Synthesis and implementation
tools 417

FPGA development boards . . . 418

Miscellaneous stuff 418

Chapter 26

Future FPGA Developments . 419

Be afraid, be very afraid 419

Next-generation architectures
and technologies 420

Don't forget the design tools . . 426

Expect the unexpected 427

Appendix A:

Signal Integrity 101 429

Before we start. 429

Capacitive and inductive coupling (crosstalk)	430	More taps than you know what to do with	468
Chip-level effects	431	Seeding an LFSR	470
Board-level effects.	438	FIFO applications	472
Appendix B:		Modifying LFSRs to sequence 2 ⁿ values.	474
Deep-Submicron Delay Effects 101		Accessing the previous value . .	475
.	443	Encryption and decryption applications	476
Introduction.	443	Cyclic redundancy check applications	477
The evolution of delay specifications	443	Data compression applications .	479
A potpourri of definitions. . . .	445	Built-in self-test applications . .	480
Alternative interconnect models	449	Pseudorandom-number-generation applications	482
DSM delay effects.	452	Last but not least	482
Summary	464	Glossary.	485
Appendix C:		About the Author	525
Linear Feedback Shift		Index	527
Registers 101	465		
The Ouroboras	465		
Many-to-one implementations .	465		

Alternative FPGA Architectures

A word of warning

In this chapter we introduce a plethora of architectural features. Certain options—such as using antifuse versus SRAM configuration cells—are mutually exclusive. Some FPGA vendors specialize in one or the other; others may offer multiple device families based on these different technologies. (Unless otherwise noted, the majority of these discussions assume SRAM-based devices.)

In the case of embedded blocks such as multipliers, adders, memory, and microprocessor cores, different vendors offer alternative “flavors” of these blocks with different “recipes” of ingredients. (Much like different brands of chocolate chip cookies featuring larger or smaller chocolate chips, for example, some FPGA families will have bigger/better/badder embedded RAM blocks, while others might feature more multipliers, or support more I/O standards, or ...)

The problem is that the features supported by each vendor and each family change on an almost daily basis. This means that once you’ve decided what features you need, you then need to do a little research to see which vendor’s offerings currently come closest to satisfying your requirements.

A little background information

Before hurling ourselves into the body of this chapter, we need to define a couple of concepts to ensure that we’re all marching to the same drumbeat. For example, you’re going to see the term *fabric* used throughout this book. In the context of

The word “fabric” comes from the Middle English *fabryke*, meaning “something constructed.”

The “ μ ” symbol stands for “micro” from the Greek *micros*, meaning “small” (hence the use of “ μ P” as an abbreviation for micro-processor.)

In the metric system, “ μ ” stands for “one millionth part of,” so 1 μ m represents “one millionth of a meter.”

DSM is pronounced by spelling it out as “D-S-M.”

UDSM is pronounced by spelling it out as “U-D-S-M.”

a silicon chip, this refers to the underlying structure of the device (sort of like the phrase “the fabric of civilized society”).

When you first hear someone using “fabric” in this way, it might sound a little snooty or pretentious (in fact, some engineers regard it as nothing more than yet another marketing term promoted by ASIC and FPGA vendors to make their devices sound more sophisticated than they really are). Truth to tell, however, once you get used to it, this is really quite a useful word.

When we talk about the *geometry* of an IC, we are referring to the size of the individual structures constructed on the chip—such as the portion of a *field-effect transistor (FET)* known as its *channel*. These structures are incredibly small. In the early to mid-1980s, devices were based on 3 μ m geometries, which means that their smallest structures were 3 millionths of a meter in size. (In conversation, we would say, “This IC is based on a three-micron technology.”)

Each new geometry is referred to as a *technology node*. By the 1990s, devices based on 1 μ m geometries were starting to appear, and feature sizes continued to plummet throughout the course of the decade. As we moved into the twenty-first century, high-performance ICs had geometries as small as 0.18 μ m. By 2002, this had shrunk to 0.13 μ m, and by 2003, devices at 0.09 μ m were starting to appear.

Any geometry smaller than around 0.5 μ m is referred to as *deep submicron (DSM)*. At some point that is not well defined (or that has multiple definitions depending on whom one is talking to), we move into the *ultradeep submicron (UDSM)* realm.

Things started to become a little awkward once geometries dropped below 1 μ m, not the least because it’s a pain to keep having to say things like “zero point one three microns.” For this reason, when conversing it’s becoming common to talk in terms of *nano*, where one nano (short for nanometer) equates to a thousandth of a micron—that is, one thousandth of a millionth of a meter. Thus, instead of mumbling, “point zero nine microns” (0.09 μ m), one can simply proclaim, “ninety

nano” (90 nano) and have done with it. Of course, these both mean exactly the same thing, but if you feel moved to regale your friends on these topics, it’s best to use the vernacular of the day and present yourself as hip and trendy rather than as an old fuddy-duddy from the last millennium.

Antifuse versus SRAM versus ...

SRAM-based devices

The majority of FPGAs are based on the use of SRAM configuration cells, which means that they can be configured over and over again. The main advantages of this technique are that new design ideas can be quickly implemented and tested, while evolving standards and protocols can be accommodated relatively easily. Furthermore, when the system is first powered up, the FPGA can initially be programmed to perform one function such as a self-test or board/system test, and it can then be reprogrammed to perform its main task.

Another big advantage of the SRAM-based approach is that these devices are at the forefront of technology. FPGA vendors can leverage the fact that many other companies specializing in memory devices expend tremendous resources on *research and development (R&D)* in this area. Furthermore, the SRAM cells are created using exactly the same CMOS technologies as the rest of the device, so no special processing steps are required in order to create these components.

In the past, memory devices were often used to qualify the manufacturing processes associated with a new technology node. More recently, the mixture of size, complexity, and regularity associated with the latest FPGA generations has resulted in these devices being used for this task. One advantage of using FPGAs over memory devices to qualify the manufacturing process is that, if there’s a defect, the structure of FPGAs is such that it’s easier to identify and locate the problem (that is, figure out what and where it is). For example, when IBM and UMC were rolling out their 0.09 μm (90 nano) processes,

R&D is pronounced by spelling it out as “R-and-D.”

IP is pronounced by spelling it out as “I-P.”

FPGAs from Xilinx were the first devices to race out of the starting gate.

Unfortunately, there’s no such thing as a free lunch. One downside of SRAM-based devices is that they have to be reconfigured every time the system is powered up. This either requires the use of a special external memory device (which has an associated cost and consumes real estate on the board) or of an on-board microprocessor (or some variation of these techniques—see also chapter 5).

Security issues and solutions with SRAM-based devices

Another consideration with regard to SRAM-based devices is that it can be difficult to protect your intellectual property, or IP, in the form of your design. This is because the configuration file used to program the device is stored in some form of external memory.

Currently, there are no commercially available tools that will read the contents of a configuration file and generate a corresponding schematic or netlist representation. Having said this, understanding and extracting the logic from the configuration file, while not a trivial task, would not be beyond the bounds of possibility given the combination of clever folks and computing horsepower available today.

Let’s not forget that there are reverse-engineering companies all over the world specializing in the recovery of “design IP.” And there are also a number of countries whose governments turn a blind eye to IP theft so long as the money keeps rolling in (you know who you are). So if a design is a high-profit item, you can bet that there are folks out there who are ready and eager to replicate it while you’re not looking.

In reality, the real issue here is not related to someone stealing your IP by reverse-engineering the contents of the configuration file, but rather their ability to clone your design, irrespective of whether they understand how it performs its magic. Using readily available technology, it is relatively easy

for someone to take a circuit board, put it on a “bed of nails” tester, and quickly extract a complete netlist for the board. This netlist can subsequently be used to reproduce the board. Now the only task remaining for the nefarious scoundrels is to copy your FPGA configuration file from its boot PROM (or EPROM, E²PROM, or whatever), and they have a duplicate of the entire design.

On the bright side, some of today’s SRAM-based FPGAs support the concept of *bitstream encryption*. In this case, the final configuration data is encrypted before being stored in the external memory device. The encryption key itself is loaded into a special SRAM-based register in the FPGA via its JTAG port (see also Chapter 5). In conjunction with some associated logic, this key allows the incoming encrypted configuration bitstream to be decrypted as it’s being loaded into the device.

The command/process of loading an encrypted bitstream automatically disables the FPGA’s read-back capability. This means that you will typically use unencrypted configuration data during development (where you need to use read-back) and then start to use encrypted data when you move into production. (You can load an unencrypted bitstream at any time, so you can easily load a test configuration and then reload the encrypted version.)

The main downside to this scheme is that you require a battery backup on the circuit board to maintain the contents of the encryption key register in the FPGA when power is removed from the system. This battery will have a lifetime of years or decades because it need only maintain a single register in the device, but it does add to the size, weight, complexity, and cost of the board.

Antifuse-based devices

Unlike SRAM-based devices, which are programmed while resident in the system, antifuse-based devices are programmed off-line using a special device programmer.

The proponents of antifuse-based FPGAs are proud to point to an assortment of (not-insignificant) advantages. First

JTAG is pronounced “J-TAG”; that is, by spelling out the ‘J’ followed by “tag” to rhyme with “bag.”

Radiation can come in the form of gamma rays (very high-energy photons), beta particles (high-energy electrons), and alpha particles.

It should be noted that rad-hard devices are not limited to antifuse technologies. Other components, such as those based on SRAM architectures, are available with special rad-hard packaging and triple redundancy design.

of all, these devices are nonvolatile (their configuration data remains when the system is powered down), which means that they are immediately available as soon as power is applied to the system. Following from their nonvolatility, these devices don't require an external memory chip to store their configuration data, which saves the cost of an additional component and also saves real estate on the board.

One noteworthy advantage of antifuse-based FPGAs is the fact that their interconnect structure is naturally "rad hard," which means they are relatively immune to the effects of radiation. This is of particular interest in the case of military and aerospace applications because the state of a configuration cell in an SRAM-based component can be "flipped" if that cell is hit by radiation (of which there is a lot in space). By comparison, once an antifuse has been programmed, it cannot be altered in this way. Having said this, it should also be noted that any flip-flops in these devices remain sensitive to radiation, so chips intended for radiation-intensive environments must have their flip-flops protected by *triple redundancy design*. This refers to having three copies of each register and taking a majority vote (ideally all three registers will contain identical values, but if one has been "flipped" such that two registers say 0 and the third says 1, then the 0s have it, or vice versa if two registers say 1 and the third says 0).

But perhaps the most significant advantage of antifuse-based FPGAs is that their configuration data is buried deep inside them. By default, it is possible for the device programmer to read this data out because this is actually how the programmer works. As each antifuse is being processed, the device programmer keeps on testing it to determine when that element has been fully programmed; then it moves onto the next antifuse. Furthermore, the device programmer can be used to automatically verify that the configuration was performed successfully (this is well worth doing when you're talking about devices containing 50 million plus programmable elements). In order to do this, the device programmer

requires the ability to read the actual states of the antifuses and compare them to the required states defined in the configuration file.

Once the device has been programmed, however, it is possible to set (grow) a special security antifuse that subsequently prevents any programming data (in the form of the presence or absence of antifuses) from being read out of the device. Even if the device is decapped (its top is removed), programmed and unprogrammed antifuses appear to be identical, and the fact that all of the antifuses are buried in the internal metallization layers makes it almost impossible to reverse-engineer the design.

Vendors of antifuse-based FPGAs may also tout a couple of other advantages relating to power consumption and speed, but if you aren't careful this can be a case of the quickness of the hand deceiving the eye. For example, they might tease you with the fact that an antifuse-based device consumes only 20 percent (approximately) of the standby power of an equivalent SRAM-based component, that their operational power consumption is also significantly lower, and that their interconnect-related delays are smaller. Also, they might casually mention that an antifuse is much smaller and thus occupies much less real estate on the chip than an equivalent SRAM cell (although they may neglect to mention that antifuse devices also require extra programming circuitry, including a large, hairy programming transistor for each antifuse). They will follow this by noting that when you have a device containing tens of millions of configuration elements, using antifuses means that the rest of the logic can be much closer together. This serves to reduce the interconnect delays, thereby making these devices faster than their SRAM cousins.

And both of the above points would be true ... if one were comparing two devices implemented at the same technology node. But therein lies the rub, because antifuse technology requires the use of around three additional process steps after the main manufacturing process has been qualified. For this (and related) reasons, antifuse devices are always at least

It's worth noting that when the MRAM technologies introduced in Chapter 2 come to fruition, these may well change the FPGA landscape.

This is because MRAM fuses would be much smaller than SRAM cells (thereby increasing component density and reducing track delays), and they would also consume much less power.

Furthermore, MRAM-based devices could be preprogrammed like antifuse-based devices (great for security) and reprogrammed like SRAM-based components (good for prototyping).

1821: England.
Michael Faraday invents
the first electric motor.

one—and usually several—generations (technology nodes) behind SRAM-based components, which effectively wipes out any speed or power consumption advantages that might otherwise be of interest.

Of course, the main disadvantage associated with antifuse-based devices is that they are OTP, so once you've programmed one of these little scallywags, its function is set in stone. This makes these components a poor choice for use in a development or prototyping environment.

EPROM-based devices

This section is short and sweet because no one currently makes—or has plans to make—EPROM-based FPGAs.

E²PROM/FLASH-based devices

E²PROM- or FLASH-based FPGAs are similar to their SRAM counterparts in that their configuration cells are connected together in a long shift-register-style chain. These devices can be configured off-line using a device programmer. Alternatively, some versions are in-system programmable, or ISP, but their programming time is about three times that of an SRAM-based component.

Once programmed, the data they contain is nonvolatile, so these devices would be “instant on” when power is first applied to the system. With regard to protection, some of these devices use the concept of a multibit key, which can range from around 50 bits to several hundred bits in size. Once you've programmed the device, you can load your user-defined key (bit-pattern) to secure its configuration data. After the key has been loaded, the only way to read data out of the device, or to write new data into it, is to load a copy of your key via the JTAG port (this port is discussed later in this chapter and also in chapter 5). The fact that the JTAG port in today's devices runs at around 20 MHz means that it would take billions of years to crack the key by exhaustively trying every possible value.

Two-transistor E²PROM and FLASH cells are approximately 2.5 times the size of their one-transistor EPROM cousins, but they are still way smaller than their SRAM counterparts. This means that the rest of the logic can be much closer together, thereby reducing interconnect delays.

On the downside, these devices require around five additional process steps on top of standard CMOS technology, which results in their lagging behind SRAM-based devices by one or more generations (technology nodes). Last but not least, these devices tend to have relatively high static power consumption due to their containing vast numbers of internal pull-up resistors.

Hybrid FLASH-SRAM devices

Last but not least, there's always someone who wants to add yet one more ingredient into the cooking pot. In the case of FPGAs, some vendors offer esoteric combinations of programming technologies. For example, consider a device where each configuration element is formed from the combination of a FLASH (or E²PROM) cell and an associated SRAM cell.

In this case, the FLASH elements can be preprogrammed. Then, when the system is powered up, the contents of the FLASH cells are copied in a massively parallel fashion into their corresponding SRAM cells. This technique gives you the nonvolatility associated with antifuse devices, which means the device is immediately available when power is first applied to the system. But unlike an antifuse-based component, you can subsequently use the SRAM cells to reconfigure the device while it remains resident in the system. Alternatively, you can reconfigure the device using its FLASH cells either while it remains in the system or off-line by means of a device programmer.

Summary

Table 4.1 briefly summarizes the key points associated with the various programming technologies described above:

1821: England.
Michael Faraday plots
the magnetic field
around a conductor.

Feature	SRAM	Antifuse	E2PROM / FLASH
Technology node	State-of-the-art	One or more generations behind	One or more generations behind
Reprogrammable	Yes (in system)	No	Yes (in-system or offline)
Reprogramming speed (inc. erasing)	Fast	---	3x slower than SRAM
Volatile (must be programmed on power-up)	Yes	No	No (but can be if required)
Requires external configuration file	Yes	No	No
Good for prototyping	Yes (very good)	No	Yes (reasonable)
Instant-on	No	Yes	Yes
IP Security	Acceptable (especially when using bitstream encryption)	Very Good	Very Good
Size of configuration cell	Large (six transistors)	Very small	Medium-small (two transistors)
Power consumption	Medium	Low	Medium
Rad Hard	No	Yes	Not really

Table 4-1. Summary of programming technologies

Fine-, medium-, and coarse-grained architectures

It is common to categorize FPGA offerings as being either *fine grained* or *coarse grained*. In order to understand what this means, we first need to remind ourselves that the main feature that distinguishes FPGAs from other devices is that their underlying fabric predominantly consists of large numbers of relatively simple programmable logic block “islands” embedded in a “sea” of programmable interconnect. (Figure 4-1).

In the case of a fine-grained architecture, each logic block can be used to implement only a very simple function. For example, it might be possible to configure the block to act as any 3-input function, such as a primitive logic gate (AND, OR, NAND, etc.) or a storage element (D-type flip-flop, D-type latch, etc.).

In reality, the vast majority of the configuration cells in an FPGA are associated with its interconnect (as opposed to its configurable logic blocks). For this reason, engineers joke that FPGA vendors actually sell only the interconnect, and they throw in the rest of the logic for free!

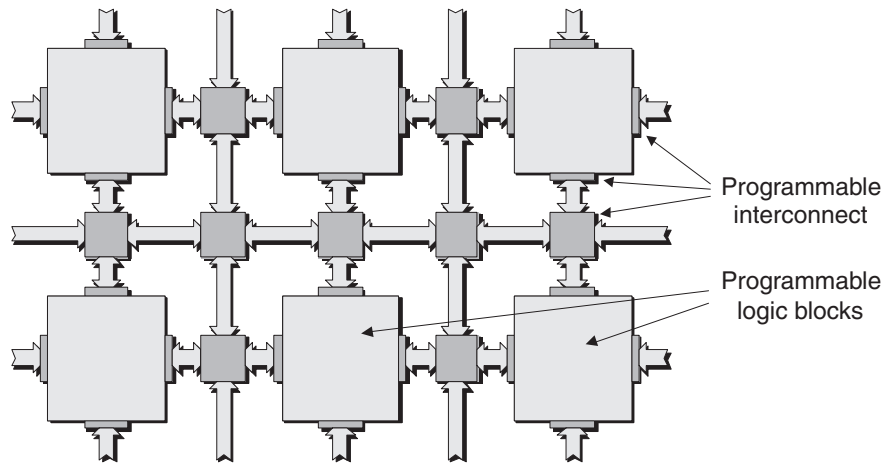


Figure 4-1. Underlying FPGA fabric.

In addition to implementing glue logic and irregular structures like state machines, fine-grained architectures are said to be particularly efficient when executing *systolic algorithms* (functions that benefit from massively parallel implementations). These architectures are also said to offer some advantages with regard to traditional logic synthesis technology, which is geared toward fine-grained ASIC architectures.

The mid-1990s saw a lot of interest in fine-grained FPGA architectures, but over time the vast majority faded away into the sunset, leaving only their coarse-grained cousins. In the case of a coarse-grained architecture, each logic block contains a relatively large amount of logic compared to their fine-grained counterparts. For example, a logic block might contain four 4-input LUTs, four multiplexers, four D-type flip-flops, and some fast carry logic (see the following topics in this chapter for more details).

An important consideration with regard to architectural granularity is that fine-grained implementations require a relatively large number of connections into and out of each block compared to the amount of functionality that can be supported by those blocks. As the granularity of the blocks increases to medium-grained and higher, the amount of connections into the blocks decreases compared to the amount of functionality

1821: England.
Sir Charles Wheatstone
reproduces sound.

they can support. This is important because the programmable interblock interconnect accounts for the vast majority of the delays associated with signals as they propagate through an FPGA.

One slight fly in the soup is that a number of companies have recently started developing really coarse-grained device architectures comprising arrays of nodes, where each node is a highly complex processing element ranging from an algorithmic function such as a *fast Fourier transform (FFT)* all the way up to a complete general-purpose microprocessor core (see also Chapters 6 and 23). Although these devices aren't classed as FPGAs, they do serve to muddy the waters. For this reason, LUT-based FPGA architectures are now often classed as medium-grained, thereby leaving the coarse-grained appellation free to be applied to these new node-based devices.

MUX- versus LUT-based logic blocks

There are two fundamental incarnations of the programmable logic blocks used to form the medium-grained architectures referenced in the previous section: MUX (multiplexer) based and LUT (lookup table) based.

MUX is pronounced to rhyme with "flux."

LUT is pronounced to rhyme with "nut."

MUX-based

As an example of a MUX-based approach, consider one way in which the 3-input function $y = (a \& b) \mid c$ could be implemented using a block containing only multiplexers (Figure 4-2).

The device can be programmed such that each input to the block is presented with a logic 0, a logic 1, or the true or inverse version of a signal (a , b , or c in this case) coming from another block or from a primary input to the device. This allows each block to be configured in myriad ways to implement a plethora of possible functions. (The x shown on the input to the central multiplexer in figure 4-2 indicates that we don't care whether this input is connected to a 0 or a 1.)

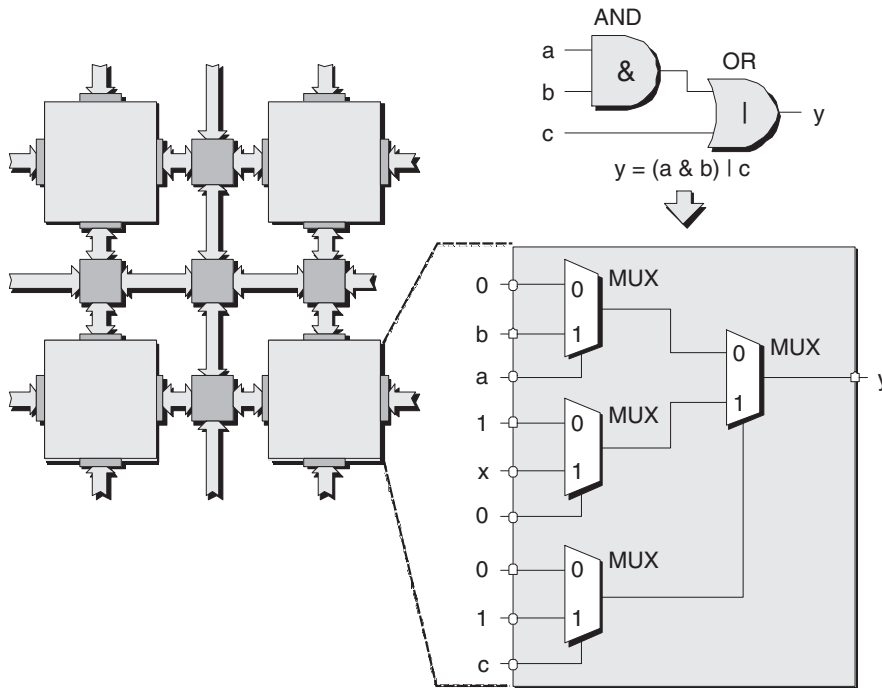


Figure 4-2. MUX-based logic block.

LUT-based

The underlying concept behind a LUT is relatively simple. A group of input signals is used as an index (pointer) to a lookup table. The contents of this table are arranged such that the cell pointed to by each input combination contains the desired value. For example, let's assume that we wish to implement the function:

$$y = (a \& b) \mid c$$

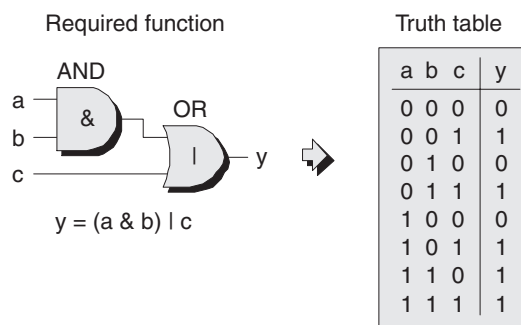


Figure 4-3. Required function and associated truth table.

If you take a group of logic gates several layers deep, then a LUT approach can be very efficient in terms of resource utilization and input-to-output delays. (In this context, “deep” refers to the number of logic gates between the inputs and the outputs. Thus, the function illustrated in figure 4-3 would be said to be two layers deep.)

However, one downside to a LUT-based architecture is that if you only want to implement a small function—such as a 2-input AND gate—somewhere in your design, you’ll end up using an entire LUT to do so. In addition to being wasteful in terms of resources, the resulting delays are high for such a simple function.

By comparison, in the case of mux-based architectures containing a mixture of muxes and logic gates, it's often possible to gain access to intermediate values from the signals linking the logic gates and the muxes. In this case, each logic block can be broken down into smaller fragments, each of which can be used to implement a simple function. Thus, these architectures may offer advantages in terms of performance and silicon utilization for designs containing large numbers of independent simple logic functions.

This can be achieved by loading a 3-input LUT with the appropriate values. For the purposes of the following examples, we shall assume that the LUT is formed from SRAM cells (but it could be formed using antifuses, E²PROM, or FLASH cells, as discussed earlier in this chapter). A commonly used technique is to use the inputs to select the desired SRAM cell using a cascade of transmission gates as shown in Figure 4-4. (Note that the SRAM cells will also be connected together in a chain for configuration purposes—that is, to load them with the required values—but these connections have been omitted from this illustration to keep things simple.)

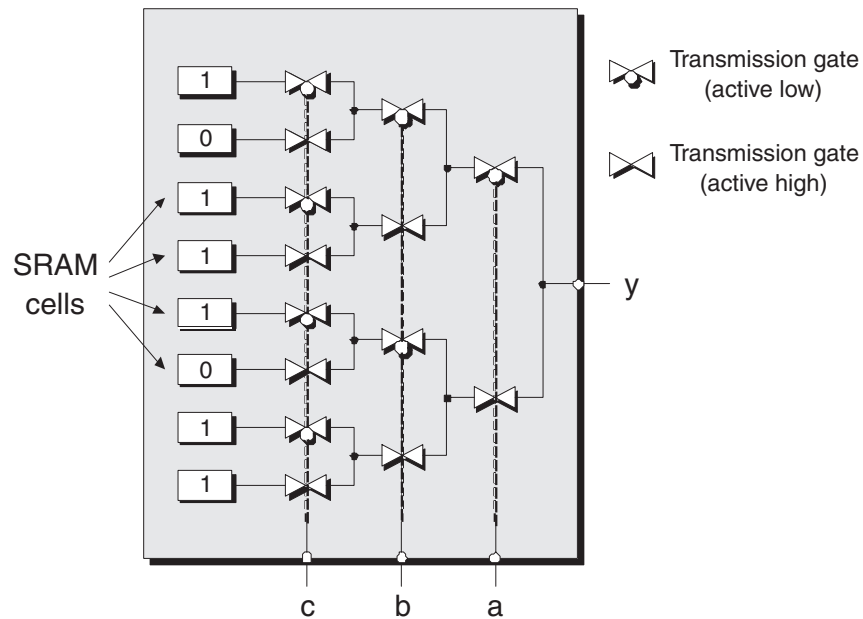


Figure 4-4. A transmission gate-based LUT (programming chain omitted for purposes of clarity).

If a transmission gate is enabled (active), it passes the signal seen on its input through to its output. If the gate is disabled, its output is electrically disconnected from the wire it is driving.

The transmission gate symbols shown with a small circle (called a “bobbie” or a “bubble”) indicate that these gates will be activated by a logic 0 on their control input. By compari-

son, symbols without bobbles indicate that these gates will be activated by a logic 1. Based on this understanding, it's easy to see how different input combinations can be used to select the contents of the various SRAM cells.

MUX-based versus LUT-based?

Once upon a time—when engineers handcrafted their circuits prior to the advent of today's sophisticated CAD tools—some folks say that it was possible to achieve the best results using MUX-based architectures. (Sad to relate, they usually don't explain exactly how these results were better, so this is largely left to our imaginations.) It is also said that MUX-based architectures have an advantage when it comes to implementing control logic along the lines of “if this input is *true* and this input is *false*, then make that output *true* ...”¹ However, some of these architectures don't provide high-speed carry logic chains, in which case their LUT-based counterparts are left as the leaders in anything to do with arithmetic processing.

Throughout much of the 1990s, FPGAs were widely used in the telecommunications and networking markets. Both of these application areas involve pushing lots of data around, in which case LUT-based architectures hold the high ground. Furthermore, as designs (and device capacities) grew larger and synthesis technology increased in sophistication, handcrafting circuits largely became a thing of the past. The end result is that the majority of today's FPGA architectures are LUT-based, as discussed below.

3-, 4-, 5-, or 6-input LUTs?

The great thing about an n -input LUT is that it can implement any possible n -input combinational (or combinatorial)

¹ Some MUX-based architectures—such as those fielded by QuickLogic (www.quicklogic.com)—feature logic blocks containing multiple layers of MUXes preceded by primitive logic gates like ANDs. This provides them with a large fan-in capability, which gives them an advantage for address decoding and state machine decoding applications.

As was noted in Chapter 3, some folks prefer to say “combinational logic,” while others favor “combinatorial logic.”

1822: England.
Charles Babbage starts
to build a mechanic
calculating machine
called the Difference
Engine.

logic function. Adding more inputs allows you to represent more complex functions, but every time you add an input, you double the number of SRAM cells.

The first FPGAs were based on 3-input LUTs. FPGA vendors and university students subsequently researched the relative merits of 3-, 4-, 5-, and even 6-input LUTs into the ground (whatever you do, don't get trapped in conversation with a bunch of FPGA architects at a party). The current consensus is that 4-input LUTs offer the optimal balance of pros and cons.

In the past, some devices were created using a mixture of different LUT sizes, such as 3-input and 4-input LUTs, because this offered the promise of optimal device utilization. However, one of the main tools in the design engineer's treasure chest is logic synthesis, and uniformity and regularity are what a synthesis tool likes best. Thus, all of the really successful architectures are currently based only on the use of 4-input LUTs. (This is not to say that mixed-size LUT architectures won't reemerge in the future as design software continues to increase in sophistication.)

LUT versus distributed RAM versus SR

The fact that the core of a LUT in an SRAM-based device comprises a number of SRAM cells offers a number of interesting possibilities. In addition to its primary role as a lookup table, some vendors allow the cells forming the LUT to be used as a small block of RAM (the 16 cells forming a 4-input LUT, for example, could be cast in the role of a 16×1 RAM). This is referred to as *distributed* RAM because (a) the LUTs are strewn (distributed) across the surface of the chip, and (b) this differentiates it from the larger chunks of *block* RAM (introduced later in this chapter).

Yet another possibility devolves from the fact that all of the FPGA's configuration cells—including those forming the LUT—are effectively strung together in a long chain (Figure 4-5).

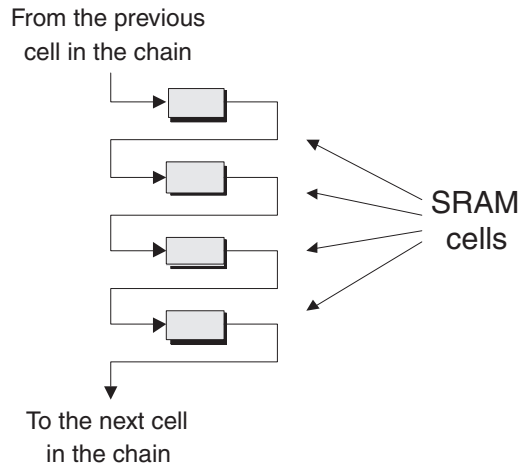


Figure 4-5. Configuration cells linked in a chain.

This aspect of the architecture is discussed in more detail in chapter 5. The point here is that, once the device has been programmed, some vendors allow the SRAM cells forming a LUT to be treated independently of the main body of the chain and to be used in the form of a shift register. Thus, each LUT may be considered to be multifaceted (figure 4-6).

CLBs versus LABs versus slices

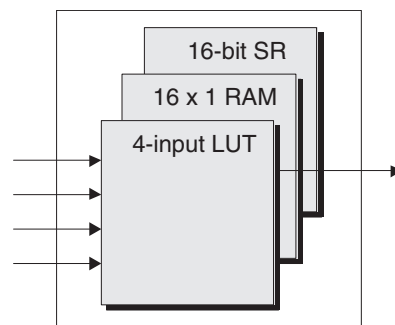


Figure 4-6. A multifaceted LUT.

“Man can not live by LUTs alone,” as the Bard would surely say if he were to be reincarnated accidentally as an FPGA designer. For this reason, in addition to one or more LUTs, a programmable logic block will contain other elements, such as multiplexers and registers. But before we delve

1822: France. Andre Ampere discovers that two wires carrying electric currents attract each other.

1827: England.
Sir Charles Wheatstone
constructs a
microphone.

into this topic, we first need to wrap our brains around some terminology.

A Xilinx logic cell

One niggle when it comes to FPGAs is that each vendor has its own names for things. But we have to start somewhere, so let's kick off by saying that the core building block in a modern FPGA from Xilinx is called a *logic cell* (LC). Among other things, an LC comprises a 4-input LUT (which can also act as a 16×1 RAM or a 16-bit shift register), a multiplexer, and a register (Figure 4-7).

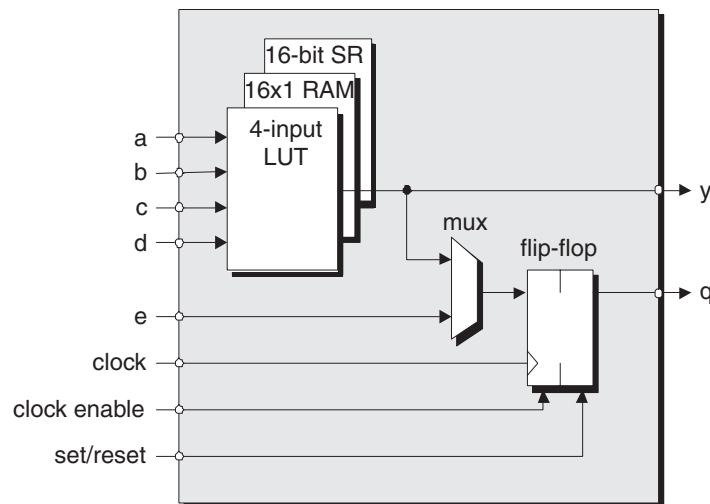


Figure 4-7. A simplified view of a Xilinx LC.

It must be noted that the illustration presented in Figure 4-7 is a gross simplification, but it serves our purposes here. The register can be configured to act as a flip-flop, as shown in Figure 4-7 or as a latch. The polarity of the *clock* (rising-edge triggered or falling-edge triggered) can be configured, as can the polarity of the *clock enable* and *set/reset* signals (active-high or active-low).

In addition to the LUT, MUX, and register, the LC also contains a smattering of other elements, including some spe-

cial fast carry logic for use in arithmetic operations (this is discussed in more detail a little later).

An Altera logic element

Just for reference, the equivalent core building block in an FPGA from Altera is called a *logic element (LE)*. There are a number of differences between a Xilinx LC and an Altera LE, but the overall concepts are very similar.

Slicing and dicing

The next step up the hierarchy is what Xilinx calls a *slice* (Altera and the other vendors doubtless have their own equivalent names). Why “slice”? Well, they had to call it something, and—whichever way you look at it—the term *slice* is “something.” At the time of this writing, a slice contains two logic cells (Figure 4-8).

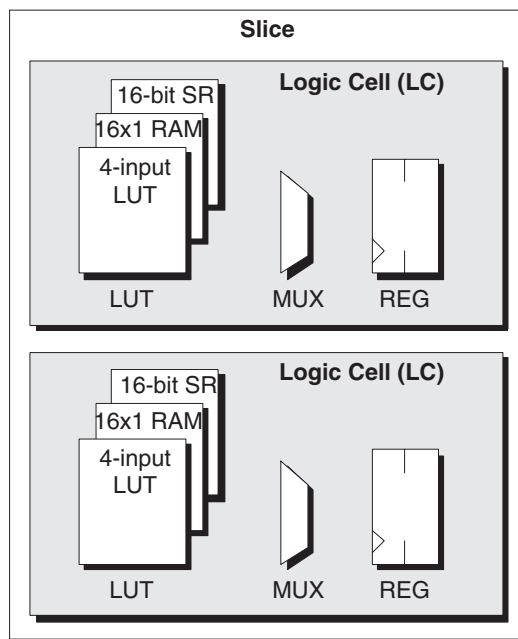


Figure 4-8. A slice containing two logic cells.

The reason for the “at the time of this writing” qualifier is that these definitions can—and do—change with the seasons.

1827: Germany. Georg Ohm investigates electrical resistance and defines Ohm’s Law.

The definition of what forms a CLB varies from year to year. In the early days, a CLB consisted of two 3-input LUTs and one register. Later versions sported two 4-input LUTs and two registers.

Now, each CLB can contain two or four slices, where each slice contains two 4-input LUTs and two registers. And as for the morrow ... well, it would take a braver man than I even to dream of speculating.

The internal wires have been omitted from this illustration to keep things simple; it should be noted, however, that although each logic cell's LUT, MUX, and register have their own data inputs and outputs, the slice has one set of *clock*, *clock enable*, and *set/reset* signals common to both logic cells.

CLBs and LABs

And moving one more level up the hierarchy, we come to what Xilinx calls a *configurable logic block (CLB)* and what Altera refers to as a *logic array block (LAB)*. (Other FPGA vendors doubtless have their own equivalent names for each of these entities, but these are of interest only if you are actually working with their devices.)

Using CLBs as an example, some Xilinx FPGAs have two slices in each CLB, while others have four. At the time of this writing, a CLB equates to a single logic block in our original visualization of “islands” of programmable logic in a “sea” of programmable interconnect (Figure 4-9).

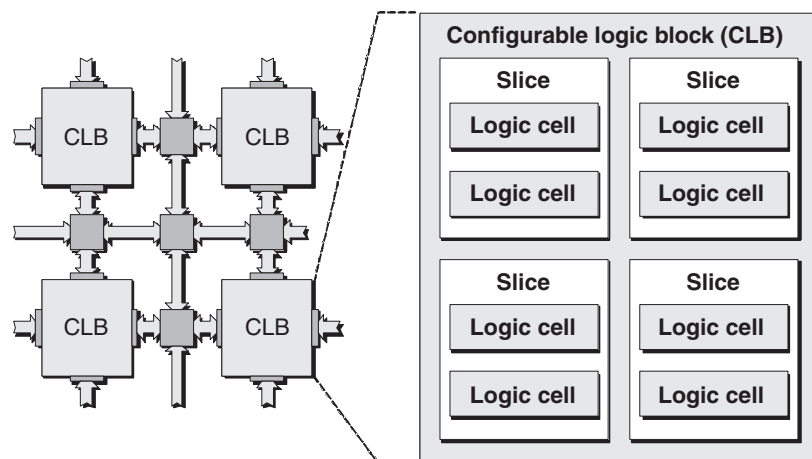


Figure 4-9. A CLB containing four slices (the number of slices depends on the FPGA family).

The point where a set of data or control signals enters or exits a logic function is commonly referred to as a “port.”

”In the case of a single-port RAM, data is written in and read out of the function using a common data bus.

There is also some fast programmable interconnect within the CLB. This interconnect (not shown in Figure 4-9 for reasons of clarity) is used to connect neighboring slices.

The reason for having this type of logic-block hierarchy—LC→ Slice (with two LCs)→ CLB (with four slices)—is that it is complemented by an equivalent hierarchy in the interconnect. Thus, there is fast interconnect between the LCs in a slice, then slightly slower interconnect between slices in a CLB, followed by the interconnect between CLBs. The idea is to achieve the optimum trade-off between making it easy to connect things together without incurring excessive interconnect-related delays.

Distributed RAMs and shift registers

We previously noted that each 4-bit LUT can be used as a 16×1 RAM. And things just keep on getting better and better because, assuming the four-slices-per-CLB configuration illustrated in figure 4-9, all of the LUTs within a CLB can be configured together to implement the following:

- Single-port 16×8 bit RAM
- Single-port 32×4 bit RAM
- Single-port 64×2 bit RAM
- Single-port 128×1 bit RAM
- Dual-port 16×4 bit RAM
- Dual-port 32×2 bit RAM
- Dual-port 64×1 bit RAM

Alternatively, each 4-bit LUT can be used as a 16-bit shift register. In this case, there are special dedicated connections between the logic cells within a slice and between the slices themselves that allow the last bit of one shift register to be connected to the first bit of another without using the ordinary LUT output (which can be used to view the contents of a selected bit within that 16-bit register). This allows the LUTs within a single CLB to be configured together to implement a shift register containing up to 128 bits as required.

Fast carry chains

A key feature of modern FPGAs is that they include the special logic and interconnect required to implement fast carry

In the case of a dual-port RAM, data is written into the function using one data bus (port) and read out using a second data bus (port). In fact, the read and write operations each have an associated address bus (used to point to a word of interest inside the RAM). This means that the read and write operations can be performed simultaneously.

chains. In the context of the CLBs introduced in the previous section, each LC contains special carry logic. This is complemented by dedicated interconnect between the two LCs in each slice, between the slices in each CLB, and between the CLBs themselves.

This special carry logic and dedicated routing boosts the performance of logical functions such as counters and arithmetic functions such as adders. The availability of these fast carry chains—in conjunction with features like the shift register incarnations of LUTs (discussed above) and embedded multipliers and the like (introduced below)—provided the wherewithal for FPGAs to be used for applications like DSP.

DSP is pronounced by spelling it out as “D-S-P.”

Embedded RAMs

A lot of applications require the use of memory, so FPGAs now include relatively large chunks of embedded RAM called *e-RAM* or *block RAM*. Depending on the architecture of the component, these blocks might be positioned around the periphery of the device, scattered across the face of the chip in relative isolation, or organized in columns, as shown in Figure 4-10.

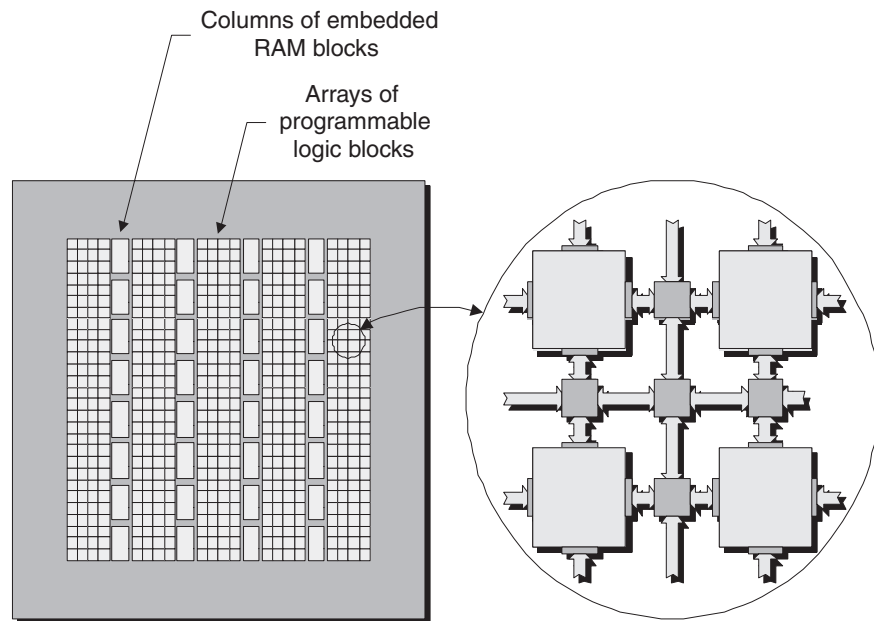


Figure 4-10. Bird's-eye view of chip with columns of embedded RAM blocks.

Depending on the device, such a RAM might be able to hold anywhere from a few thousand to tens of thousands of bits. Furthermore, a device might contain anywhere from tens to hundreds of these RAM blocks, thereby providing a total storage capacity of a few hundred thousand bits all the way up to several million bits.

Each block of RAM can be used independently, or multiple blocks can be combined together to implement larger blocks. These blocks can be used for a variety of purposes, such as implementing standard single- or dual-port RAMs, *first-in first-out (FIFO)* functions, state machines, and so forth.

Embedded multipliers, adders, MACs, etc.

Some functions, like multipliers, are inherently slow if they are implemented by connecting a large number of programmable logic blocks together. Since these functions are required by a lot of applications, many FPGAs incorporate special hard-wired multiplier blocks. These are typically located in close proximity to the embedded RAM blocks introduced in the previous point because these functions are often used in conjunction with each other (Figure 4-11).

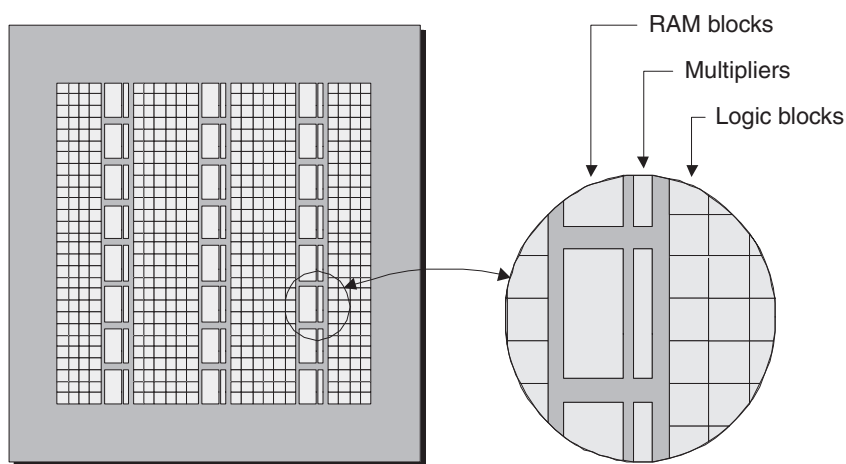


Figure 4-11. Bird's-eye view of chip with columns of embedded multipliers and RAM blocks.

Similarly, some FPGAs offer dedicated adder blocks. One operation that is very common in DSP-type applications is

FIFO is pronounced “fi” to rhyme with “hi,” followed by “fo” to rhyme with “no” (like the “Hi-Ho” song in “Snow White and the Seven Dwarfs”).

1829: England.
Sir Charles Wheatstone
invents the concertina.

called a multiply-and-accumulate (MAC) (Figure 4-12). As its name would suggest, this function multiplies two numbers together and adds the result to a running total stored in an accumulator.

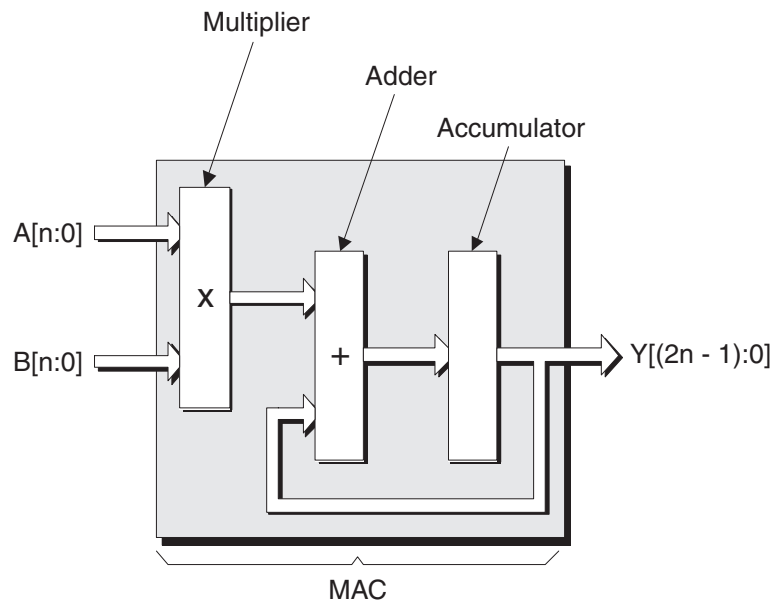


Figure 4-12. The functions forming a MAC.

If the FPGA you are working with supplies only embedded multipliers, you will have to implement this function by combining the multiplier with an adder formed from a number of programmable logic blocks, while the result is stored in some associated flip-flops, in a block RAM, or in a number of distributed RAMs. Life becomes a little easier if the FPGA also provides embedded adders, and some FPGAs provide entire MACs as embedded functions.

Embedded processor cores (hard and soft)

Almost any portion of an electronic design can be realized in hardware (using logic gates and registers, etc.) or software (as instructions to be executed on a microprocessor). One of the main partitioning criteria is how fast you wish the various functions to perform their tasks:

- *Picosecond and nanosecond logic*: This has to run insanely fast, which mandates that it be implemented in hardware (in the FPGA fabric).
- *Microsecond logic*: This is reasonably fast and can be implemented either in hardware or software (this type of logic is where you spend the bulk of your time deciding which way to go).
- *Millisecond logic*: This is the logic used to implement interfaces such as reading switch positions and flashing *light-emitting diodes (LEDs)*. It's a pain slowing the hardware down to implement this sort of function (using huge counters to generate delays, for example). Thus, it's often better to implement these tasks as microprocessor code (because processors give you lousy speed—compared to dedicated hardware—but fantastic complexity).

1831: England.
Michael Faraday
creates the first
electric dynamo.

The fact is that the majority of designs make use of microprocessors in one form or another. Until recently, these appeared as discrete devices on the circuit board. Of late, high-end FPGAs have become available that contain one or more embedded microprocessors, which are typically referred to as *microprocessor cores*. In this case, it often makes sense to move all of the tasks that used to be performed by the external microprocessor into the internal core. This provides a number of advantages, not the least being that it saves the cost of having two devices; it eliminates large numbers of tracks, pads, and pins on the circuit board; and it makes the board smaller and lighter.

Hard microprocessor cores

A hard microprocessor core is implemented as a dedicated, predefined block. There are two main approaches for integrating such a core into the FPGA. The first is to locate it in a strip (actually called “The Stripe”) to the side of the main FPGA fabric (Figure 4-13).

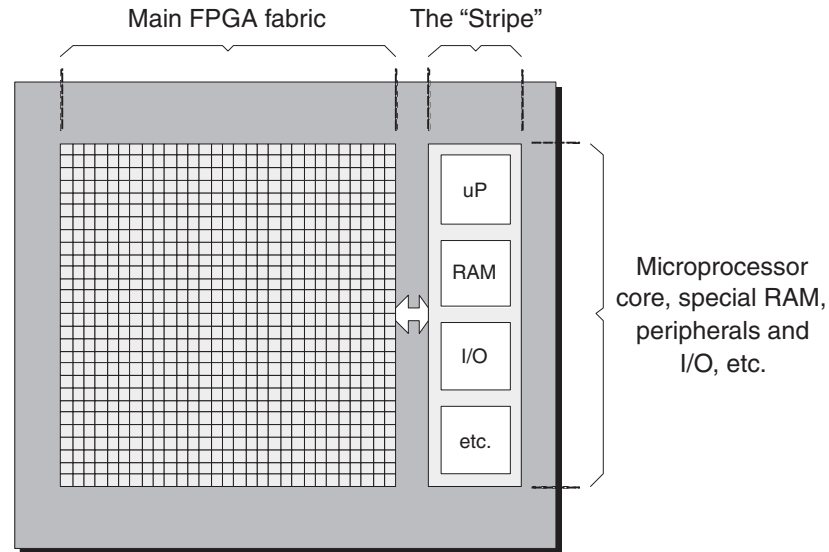


Figure 4-13. Birds-eye view of chip with embedded core outside of the main fabric.

MCM is pronounced by spelling it out as “M-C-M.”

In this scenario, all of the components are typically formed on the same silicon chip, although they could also be formed on two chips and packaged as a *multichip module* (MCM). The main FPGA fabric would also include the embedded RAM blocks, multipliers, and the like introduced earlier, but these have been omitted from this illustration to keep things simple.

One advantage of this implementation is that the main FPGA fabric is identical for devices with and without the embedded microprocessor core, which can help make things easier for the design tools used by the engineers. The other advantage is that the FPGA vendor can bundle a whole load of additional functions in the strip to complement the microprocessor core, such as memory, special peripherals, and so forth.

An alternative is to embed one or more microprocessor cores directly into the main FPGA fabric. One, two, and even four core implementations are currently available as I pen these words (Figure 4-14).

Once again, the main FPGA fabric would also include the embedded RAM blocks, multipliers, and the like introduced

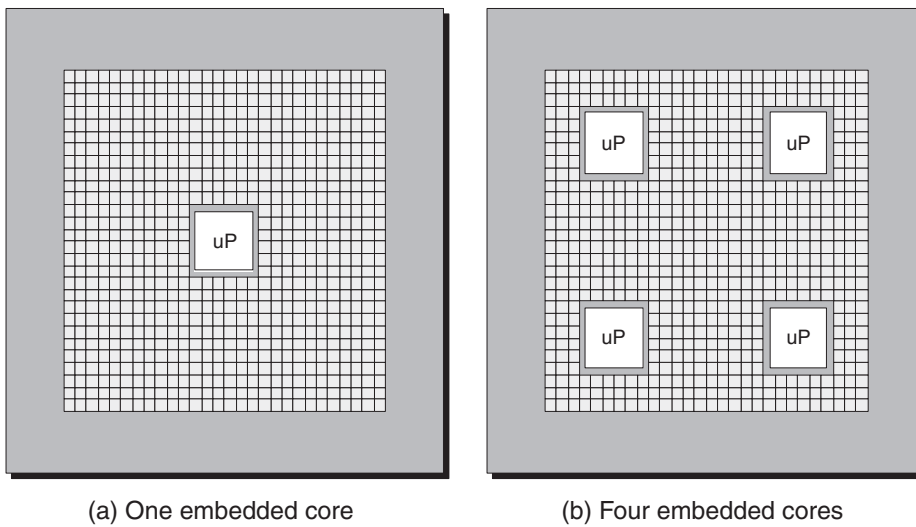


Figure 4-14. Bird's-eye view of chips with embedded cores inside the main fabric.

earlier, but these have been omitted from this illustration to keep things simple.

In this case, the design tools have to be able to take account of the presence of these blocks in the fabric; any memory used by the core is formed from embedded RAM blocks, and any peripheral functions are formed from groups of general-purpose programmable logic blocks. Proponents of this scheme will argue that there are inherent speed advantages to be gained from having the microprocessor core in intimate proximity to the main FPGA fabric.

Soft microprocessor cores

As opposed to embedding a microprocessor physically into the fabric of the chip, it is possible to configure a group of programmable logic blocks to act as a microprocessor. These are typically called *soft cores*, but they may be more precisely categorized as either “soft” or “firm” depending on the way in which the microprocessor’s functionality is mapped onto the logic blocks (see also the discussions associated with the hard IP, soft IP, and firm IP topics later in this chapter).

1831: England. Michael Faraday creates the first electrical transformer.

1831: England.
Michael Faraday
discovers magnetic
lines of force.

Soft cores are simpler (more primitive) and slower than their hard-core counterparts.² However, they have the advantage that you only need to implement a core if you need it and also that you can instantiate as many cores as you require until you run out of resources in the form of programmable logic blocks.

Clock trees and clock managers

All of the synchronous elements inside an FPGA—for example, the registers configured to act as flip-flops inside the programmable logic blocks—need to be driven by a clock signal. Such a clock signal typically originates in the outside world, comes into the FPGA via a special clock input pin, and is then routed through the device and connected to the appropriate registers.

Clock trees

Consider a simplified representation that omits the programmable logic blocks and shows only the clock tree and the registers to which it is connected (Figure 4-15).

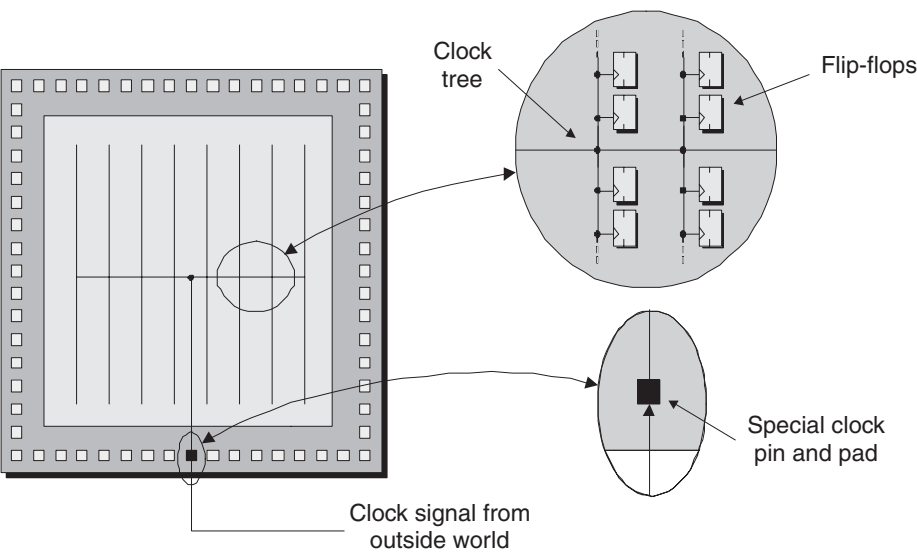


Figure 4-15. A simple clock tree.

² A soft core typically runs at 30 to 50 percent of the speed of a hard core.

This is called a “clock tree” because the main clock signal branches again and again (the flip-flops can be considered to be the “leaves” on the end of the branches). This structure is used to ensure that all of the flip-flops see their versions of the clock signal as close together as possible. If the clock were distributed as a single long track driving all of the flip-flops one after another, then the flip-flop closest to the clock pin would see the clock signal much sooner than the one at the end of the chain. This is referred to as *skew*, and it can cause all sorts of problems (even when using a clock tree, there will be a certain amount of skew between the registers on a branch and also between branches).

The clock tree is implemented using special tracks and is separate from the general-purpose programmable interconnect. The scenario shown above is actually very simplistic. In reality, multiple clock pins are available (unused clock pins can be employed as general-purpose I/O pins), and there are multiple *clock domains* (clock trees) inside the device.

Clock managers

Instead of configuring a clock pin to connect directly into an internal clock tree, that pin can be used to drive a special hard-wired function (block) called a *clock manager* that generates a number of *daughter clocks* (Figure 4-16).

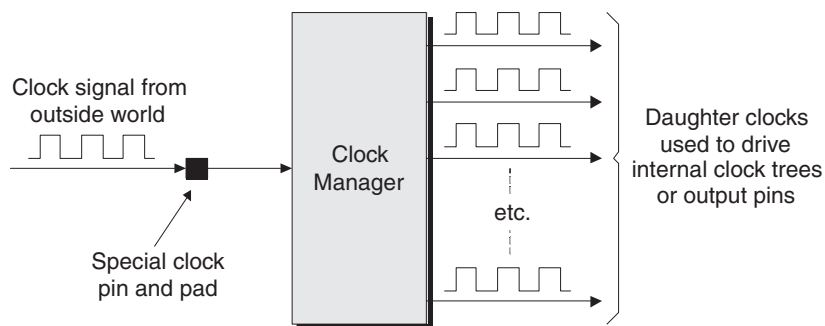


Figure 4-16. A clock manager generates daughter clocks.

These daughter clocks may be used to drive internal clock trees or external output pins that can be used to provide clock-

A clock manager as described here is referred to as a *digital clock manager (DCM)* in the Xilinx world. DCM is pronounced by spelling it out as “D-C-M.”

The term hertz was taken from the name of Heinrich Rudolf Hertz, a professor of physics at Karlsruhe Polytechnic in Germany, who first transmitted and received radio waves in a laboratory environment in 1888.

One hertz (Hz) equates to “one cycle per second,” so MHz stands for megahertz or “million Hertz.”

ing services to other devices on the host circuit board. Each family of FPGAs has its own type of clock manager (there may be multiple clock manager blocks in a device), where different clock managers may support only a subset of the following features:

Jitter removal: For the purposes of a simple example, assume that the clock signal has a frequency of 1 MHz (in reality, of course, this could be much, much higher). In an ideal environment each clock edge from the outside world would arrive exactly one millionth of a second after its predecessor. In the real world, however, clock edges may arrive a little early or a little late.

As one way to visualize this effect—known as *jitter*—imagine if we were to superimpose multiple edges on top of each other; the result would be a “fuzzy” clock (Figure 4-17).

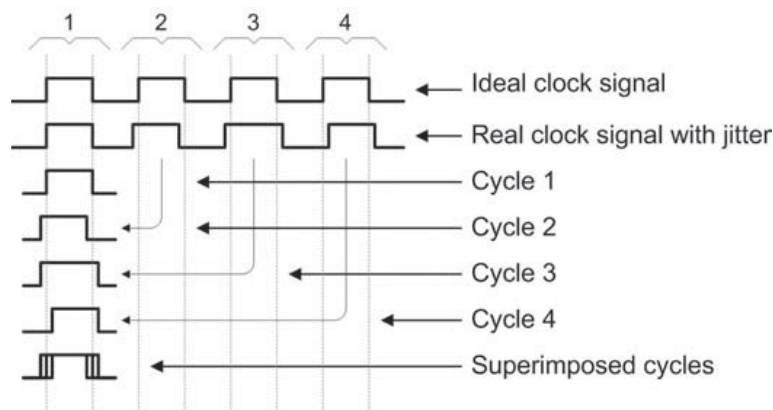


Figure 4-17. Jitter results in a fuzzy clock.

The FPGA’s clock manager can be used to detect and correct for this jitter and to provide “clean” daughter clock signals for use inside the device (Figure 4-18).

Frequency synthesis: It may be that the frequency of the clock signal being presented to the FPGA from the outside world is not exactly what the design engineers wish for. In this case, the clock manager can be used to generate daughter clocks with frequencies that are derived by multiplying or dividing the original signal.

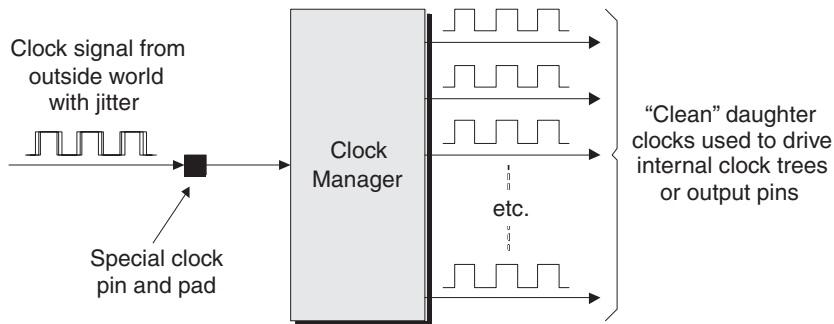


Figure 4-18. The clock manager can remove jitter.

As a really simple example, consider three daughter clock signals: the first with a frequency equal to that of the original clock, the second multiplied to be twice that of the original clock, and the third divided to be half that of the original clock (Figure 4-19).

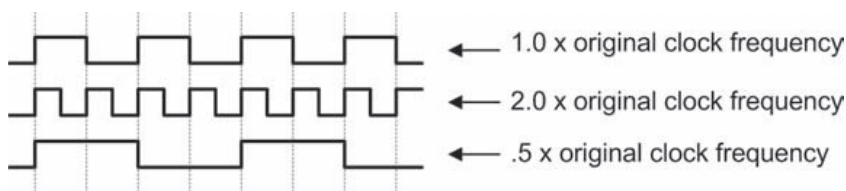


Figure 4-19. Using the clock manager to perform frequency synthesis.

Once again, Figure 4-19 reflects very simple examples. In the real world, one can synthesize all sorts of internal clocks, such as an output that is four-fifths the frequency of the original clock.

Phase shifting: Certain designs require the use of clocks that are phase shifted (delayed) with respect to each other. Some clock managers allow you to select from fixed phase shifts of common values such as 120° and 240° (for a three-phase clocking scheme) or 90° , 180° , and 270° (if a four-phase clocking scheme is required). Others allow you to configure the exact amount of phase shift you require for each daughter clock.

For example, let's assume that we are deriving four internal clocks from a master clock, where the first is in phase with the

1831: England. Michael Faraday discovers the principal of electro-magnetic induction.

original clock, the second is phase shifted by 90° , the third by 180° , and so forth (Figure 4-20).

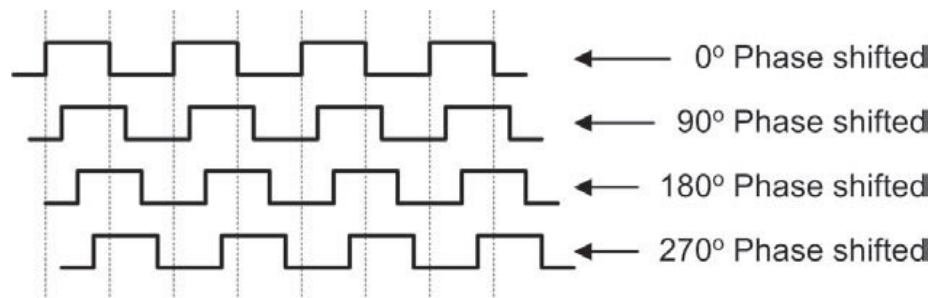


Figure 4-20. Using the clock manager to phase-shift the daughter clocks.

Auto-skew correction: For the sake of simplicity, let's assume that we're talking about a daughter clock that has been configured to have the same frequency and phase as the main clock signal coming into the FPGA. By default, however, the clock manager will add some element of delay to the signal as it performs its machinations. Also, more significant delays will be added by the driving gates and interconnect employed in the clock's distribution. The result is that—if nothing is done to correct it—the daughter clock will lag behind the input clock by some amount. Once again, the difference between the two signals is known as *skew*.

Depending on how the main clock and the daughter clock are used in the FPGA (and on the rest of the circuit board), this can cause a variety of problems. Thus, the clock manager may allow a special input to feed the daughter clock. In this case, the clock manager will compare the two signals and specifically add additional delay to the daughter clock sufficient to realign it with the main clock (Figure 4-21).

To be a tad more specific, only the *prime* (zero phase-shifted) daughter clock will be treated in this way, and all of the other daughter clocks will be phase aligned to this prime daughter clock.

Some FPGA clock managers are based on *phase-locked loops* (PLLs), while others are based on *digital delay-locked loops*

PLL is pronounced by spelling it out as "P-L-L."

"DLL is pronounced by spelling it out as "D-L-L."

At this time, I do not know why digital delay-locked loop is not abbreviated to "DDLL."

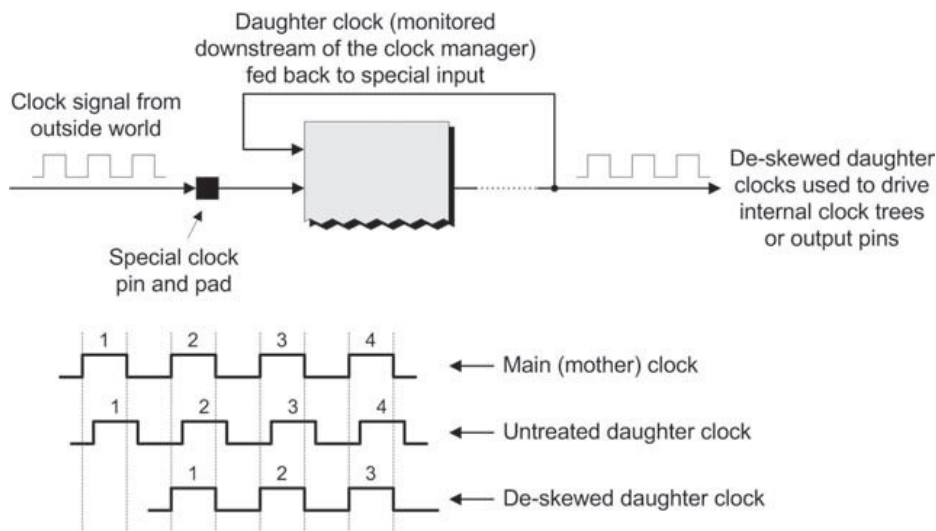


Figure 4-21. Deskewing with reference to the mother clock.

(DLLs). PLLs have been used since the 1940s in analog implementations, but recent emphasis on digital methods has made it desirable to match signal phases digitally. PLLs can be implemented using either analog or digital techniques, while DLLs are by definition digital in nature. The proponents of DLLs say that they offer advantages in terms of precision, stability, power management, noise insensitivity, and jitter performance.

General-purpose I/O

Today's FPGA packages can have 1,000 or more pins, which are arranged as an array across the base of the package. Similarly, when it comes to the silicon chip inside the package, flip-chip packaging strategies allow the power, ground, clock, and I/O pins to be presented across the surface of the chip. Purely for the purposes of these discussions (and illustrations), however, it makes things simpler if we assume that all of the connections to the chip are presented in a ring around the circumference of the device, as indeed they were for many years.

1831: England. Michael Faraday discovers that a moving magnet induces an electric current.

Configurable I/O standards

Let's consider for a moment an electronic product from the perspective of the architects and engineers designing the circuit board. Depending on what they are trying to do, the devices they are using, the environment the board will operate in, and so on, these guys and gals will select a particular standard to be used to transfer data signals. (In this context, "standard" refers to electrical aspects of the signals, such as their logic 0 and logic 1 voltage levels.)

The problem is that there is a wide variety of such standards, and it would be painful to have to create special FPGAs to accommodate each variation. For this reason, an FPGA's general-purpose I/O can be configured to accept and generate signals conforming to whichever standard is required. These general-purpose I/O signals will be split into a number of banks—we'll assume eight such banks numbered from 0 to 7 (Figure 4-22).

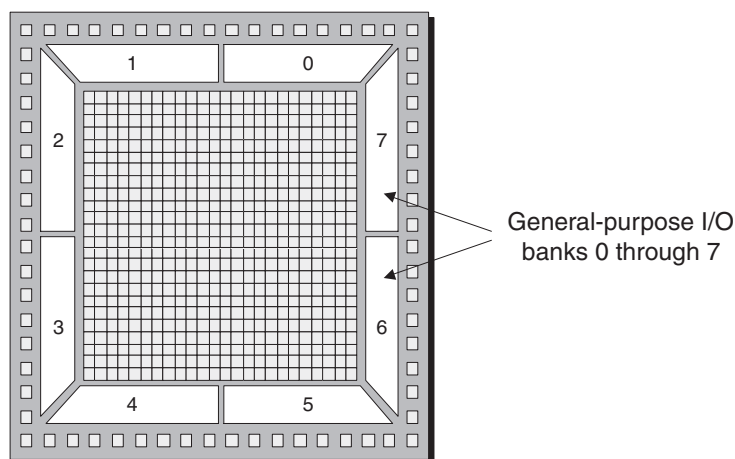


Figure 4-22. Bird's-eye view of chip showing general-purpose I/O banks.

The interesting point is that each bank can be configured individually to support a particular I/O standard. Thus, in addition to allowing the FPGA to work with devices using multiple I/O standards, this allows the FPGA to actually be used to interface between different I/O standards (and also to

translate between different protocols that may be based on particular electrical standards).

Configurable I/O impedances

The signals used to connect devices on today's circuit board often have fast *edge rates* (this refers to the time it takes the signal to switch between one logic value and another). In order to prevent signals reflecting back (bouncing around), it is necessary to apply appropriate terminating resistors to the FPGA's input or output pins.

In the past, these resistors were applied as discrete components that were attached to the circuit board outside the FPGA. However, this technique became increasingly problematic as the number of pins started to increase and their *pitch* (the distance between them) shrank. For this reason, today's FPGAs allow the use of internal terminating resistors whose values can be configured by the user to accommodate different circuit board environments and I/O standards.

Core versus I/O supply voltages

In the days of yore—circa 1965 to 1995—the majority of digital ICs used a ground voltage of 0V and a supply voltage of +5V. Furthermore, their I/O signals also switched between 0V (logic 0) and +5V (logic 1), which made life really simple.

Over time, the geometries of the structures on silicon chips became smaller because smaller transistors have lower costs, higher speed, and lower power consumption. However, these processes demanded lower supply voltages, which have continued to fall over the years (Table 4.2).

The point is that this supply (which is actually provided using large numbers of power and ground pins) is used to power the FPGA's internal logic. For this reason, this is known as the *core voltage*. However, different I/O standards may use signals with voltage levels significantly different from the core voltage, so each bank of general-purpose I/Os can have its own additional supply pins.

1832: England.
Charles Babbage
conceives the first
mechanical computer,
the Analytical Engine.

1832: England
Joseph Henry
discovers self-induction
or inductance.

Year	Supply (Core Voltage (V))	Technology Node (nm)
1998	3.3	350
1999	2.5	250
2000	1.8	180
2001	1.5	150
2003	1.2	130

Table 4.2. Supply voltages versus technology nodes.

It’s interesting to note that—from the 350 nm node onwards—the core voltage has scaled fairly linearly with the process technology. However, there are physical reasons not to go much below 1V (these reasons are based on technology aspects such as transistor input switching thresholds and voltage drops), so this “voltage staircase” might start to tail off in the not-so-distant future.

Gigabit transceivers

The traditional way to move large amounts of data between devices is to use a *bus*, a collection of signals that carry similar data and perform a common function (Figure 4-23).

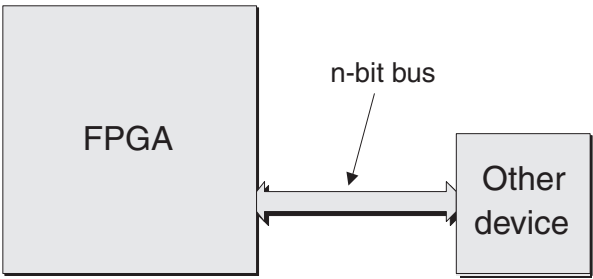


Figure 4-23: Using a bus to communicate between devices.

Early microprocessor-based systems circa 1975 used 8-bit buses to pass data around. As the need to push more data around and to move it faster grew, buses grew to 16 bits in width, then 32 bits, then 64 bits, and so forth. The problem is that this requires a lot of pins on the device and a lot of tracks

connecting the devices together. Routing these tracks so that they all have the same length and impedance becomes increasingly painful as boards grow in complexity. Furthermore, it becomes increasingly difficult to manage signal integrity issues (such as susceptibility to noise) when you are dealing with large numbers of bus-based tracks.

For this reason, today's high-end FPGAs include special hard-wired gigabit transceiver blocks. These blocks use one pair of differential signals (which means a pair of signals that always carry opposite logical values) to *transmit* (TX) data and another pair to *receive* (RX) data (Figure 4-24).

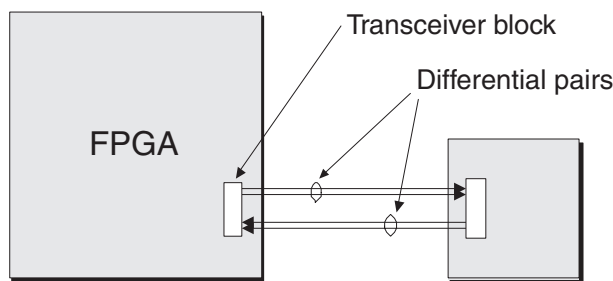


Figure 4-24: Using high-speed transceivers to communicate between devices.

These transceivers operate at incredibly high speeds, allowing them to transmit and receive billions of bits of data per second. Furthermore, each block actually supports a number (say four) of such transceivers, and an FPGA may contain a number of these transceiver blocks.

Hard IP, soft IP, and firm IP

Each FPGA vendor offers its own selection of *hard*, *firm*, and *soft IP*. Hard IP comes in the form of preimplemented blocks such as microprocessor cores, gigabit interfaces, multipliers, adders, MAC functions, and the like. These blocks are designed to be as efficient as possible in terms of power consumption, silicon real estate, and performance. Each FPGA family will feature different combinations of such blocks together with various quantities of programmable logic blocks.

1833: England.
Michael Faraday defines the laws of electrolysis.

IP is pronounced by spelling it out as "I-P."

HDL is pronounced by spelling it out as "H-D-L."

VHDL is pronounced by spelling it out as "V-H-D-L."

RTL is pronounced by spelling it out as "R-T-L."

At the other end of the spectrum, soft IP refers to a source-level library of high-level functions that can be included to the users' designs. These functions are typically represented using a hardware description language, or HDL, such as Verilog or VHDL at the *register transfer level (RTL)* of abstraction. Any soft IP functions the design engineers decide to use are incorporated into the main body of the design—which is also specified in RTL—and subsequently synthesized down into a group of programmable logic blocks (possibly combined with some hard IP blocks like multipliers, etc.).

Holding somewhat of a middle ground is firm IP, which also comes in the form of a library of high-level functions. Unlike their soft IP equivalents, however, these functions have already been optimally mapped, placed, and routed into a group of programmable logic blocks (possibly combined with some hard IP blocks like multipliers, etc.). One or more copies of each predefined firm IP block can be instantiated (called up) into the design as required.

The problem is that it can be hard to draw the line between those functions that are best implemented as hard IP and those that should be implemented as soft or firm IP (using a number of general-purpose programmable logic blocks). In the case of functions like the multipliers, adders, and MACs discussed earlier in this chapter, these are generally useful for a wide range of applications. On the other hand, some FPGAs contain dedicated blocks to handle specific interface protocols like the PCI standard. It can, of course, make your life a lot easier if this happens to be the interface with which you wish to connect your device to the rest of the board. On the other hand, if you decide you need to use some other interface, a dedicated PCI block will serve only to waste space, block traffic, and burn power in your chip.

Generally speaking, once FPGA vendors add a function like this into their device, they've essentially placed the component into a niche. Sometimes you have to do this to achieve the desired performance, but this is a classic problem

PCI is pronounced by spelling it out as "P-C-I."

because the next generation of the device is often fast enough to perform this function in its main (programmable) fabric.

System gates versus real gates

One common metric used to measure the size of a device in the ASIC world is that of *equivalent gates*. The idea is that different vendors provide different functions in their cell libraries, where each implementation of each function requires a different number of transistors. This makes it difficult to compare the relative capacity and complexity of two devices.

The answer is to assign each function an equivalent gate value along the lines of “Function A equates to five equivalent gates; function B equates to three equivalent gates ...” The next step is to count all of the instances of each function, convert them into their equivalent gate values, sum all of these values together, and proudly proclaim, “My ASIC contains 10 million equivalent gates, which makes it *much* bigger than your ASIC!”

Unfortunately, nothing is simple because the definition of what actually constitutes an equivalent gate can vary depending on whom one is talking to. One common convention is for a 2-input NAND function to represent one equivalent gate. Alternatively, some vendors define an equivalent gate as equaling an arbitrary number of transistors. And a more esoteric convention defines an ECL equivalent gate as being “one-eleventh the minimum logic required to implement a single-bit full adder” (who on earth came up with this one?). As usual, the best policy here is to make sure that everyone is talking about the same thing before releasing your grip on your hard-earned money.

And so we come to FPGAs. One of the problems FPGA vendors run into occurs when they are trying to establish a basis for comparison between their devices and ASICs. For example, if someone has an existing ASIC design that contains 500,000 equivalent gates and he wishes to migrate this design into an FPGA implementation, how can he tell if his design will fit into a particular FPGA. The fact that each

1837: America.
Samual Finley Breese
Morse exhibits an
electric telegraph.

1837: England
 Sir Charles Wheatstone
 and Sir William
 Fothergill Cooke patent
 the five-needle electric
 telegraph.

4-input LUT can be used to represent anywhere between one and more than twenty 2-input primitive logic gates makes such a comparison rather tricky.

In order to address this issue, FPGA vendors started talking about *system gates* in the early 1990s. Some folks say that this was a noble attempt to use terminology that ASIC designers could relate to, while others say that it was purely a marketing ploy that didn't do anyone any favors.

Sad to relate, there appears to be no clear definition as to exactly what a system gate is. The situation was difficult enough when FPGAs essentially contained only generic programmable logic in the form of LUTs and registers. Even then, it was hard to state whether or not a particular ASIC design containing x equivalent gates could fit into an FPGA containing y system gates. This is because some ASIC designs may be predominantly combinatorial, while others may make excessively heavy use of registers. Both cases may result in a suboptimal mapping onto the FPGA.

The problem became worse when FPGAs started containing embedded blocks of RAM, because some functions can be implemented much more efficiently in RAM than in general-purpose logic. And the fact that LUTs can act as distributed RAM only serves to muddy the waters; for example, one vendor's system gate count values now include the qualifier, "Assumes 20 percent to 30 percent of LUTs are used as RAM." And, of course, the problems are exacerbated when we come to consider FPGAs containing embedded processor cores and similar functions, to the extent that some vendors now say, "System gate values are not meaningful for these devices."

Is there a rule of thumb that allows you to convert system gates to equivalent gates and vice versa? Sure, there are lots of them! Some folks say that if you are feeling optimistic, then you should divide the system gate value by three (in which case three million FPGA system gates would equate to one million ASIC equivalent gates, for example). Or if you're feeling a tad more on the pessimistic side, you could divide the

system gates by five (in which case three million system gates would equate to 600,000 equivalent gates).

However, other folks would say that the above is only true if you assume that the system gates value encompasses all of the functions that you can implement using both the general-purpose programmable logic and the block RAMs. These folks would go on to say that if you remove the block RAMs from the equation, then you should divide the system gates value by ten (in which case, three million system gates would equate to only 300,000 equivalent gates), but in this case you still have the block RAMs to play with ... arrggghhhh!

Ultimately, this topic spirals down into such a quagmire that even the FPGA vendors are trying desperately not to talk about system gates any more. When FPGAs were new on the scene, people were comfortable with the thought of equivalent gates and not so at ease considering designs in terms of LUTs, slices, and the like; however, the vast number of FPGA designs that have been undertaken over the years means that engineers are now much happier thinking in FPGA terms. For this reason, speaking as someone living in the trenches, I would prefer to see FPGAs specified and compared using only simple counts of:

- Number of logic cells or logic elements or whatever (which equates to the number of 4-input LUTs and associated flip-flops/latches)
- Number (and size) of embedded RAM blocks
- Number (and size) of embedded multipliers
- Number (and size) of embedded adders
- Number (and size) of embedded MACs
- etc.

Why is this so hard? And it would be really useful to take a diverse suite of real-world ASIC design examples, giving their equivalent gate values, along with details as to their flops/latches, primitive gates, and other more complex functions, then to relate each of these examples to the number of

1842: England
Joseph Henry
discovers that an
electrical spark
between two
conductors is able to
induce magnetism in
needles—this effect is
detected at a distance
of 30 meters.

1842: Scotland. Alexander Bail demonstrates first electromechanical means to capture, transmit, and reproduce an image.

LUTs and flip-flops/latches required in equivalent FPGA implementations, along with the amount of embedded RAM and the number of other embedded functions.

Even this would be less than ideal, of course, because one tends to design things differently for FPGA and ASIC targets, but it would be a start.

FPGA years

We've all heard it said that each year for a dog is equivalent to seven human years, the idea being that a 10-year-old pooch would be 70 years old in human terms. Thinking like this doesn't actually do anyone much good. On the other hand, it does provide a useful frame of reference so that when your hound can no longer keep up with you on a long walk, you can say, "Well, it's only to be expected because the poor old fellow is almost 100 years old" (or whatever).

Similarly, in the case of FPGAs, it may help to think that one of their years equates to approximately 15 human years. Thus, if you're working with an FPGA that was only introduced to the market within the last year, you should view it as a teenager. On the one hand, if you have high hopes for the future, he or she may end up with a Nobel Peace Prize or as the President of the United States. On the other hand, the object of your affections will typically have a few quirks that you have to get used to and learn to work around.

By the time an FPGA has been on the market for two years (equating to 30 years in human terms), you can start to think of it as reasonably mature and a good all-rounder at the peak of its abilities. After three years (45 years old), an FPGA is becoming somewhat staid and middle-aged, and by four years (60 years old), you should treat it with respect and make sure that you don't try to work it like a carthorse!

Programming (Configuring) an FPGA

Weasel words

Before plunging headfirst into this topic, it's probably appropriate to preface our discussions with a few "weasel words" (always remember the saying, "Eagles may soar, but weasels rarely get sucked into jet engines at 10,000 feet!").

The point is that each FPGA vendor has its own unique terminology and its own techniques and protocols for doing things. To make life even more exciting, the detailed mechanisms for programming FPGAs can vary on a family-by-family basis. For these reasons, the following discussions are intended to provide only a generic introduction to this subject.

Configuration files, etc.

Section 2 of this book describes a variety of tools and flows that may be used to capture and implement FPGA designs. The end result of all of these techniques is a *configuration file* (sometimes called a *bit file*), which contains the information that will be uploaded into the FPGA in order to program it to perform a specific function.

In the case of SRAM-based FPGAs, the configuration file contains a mixture of *configuration data* (bits that are used to define the state of programmable logic elements directly) and *configuration commands* (instructions that tell the device what to do with the configuration data). When the configuration file is in the process of being loaded into the device, the information being transferred is referred to as the *configuration bitstream*.

1843: England.
Augusta Ada Lovelace
publishes her notes
explaining the concept
of a computer.

E²-based and FLASH-based devices are programmed in a similar manner to their SRAM-based cousins. By comparison, in the case of antifuse-based FPGAs, the configuration file predominantly contains only a representation of the configuration data that will be used to grow the antifuses.

Configuration cells

The underlying concept associated with programming an FPGA is relatively simple (i.e., load the configuration file into the device). It can, however, be a little tricky to wrap one's brain around all of the different facets associated with this process, so we'll start with the basics and work our way up. Initially, let's assume we have a rudimentary device consisting only of an array of very simple programmable logic blocks surrounded by programmable interconnect (Figure 5-1).

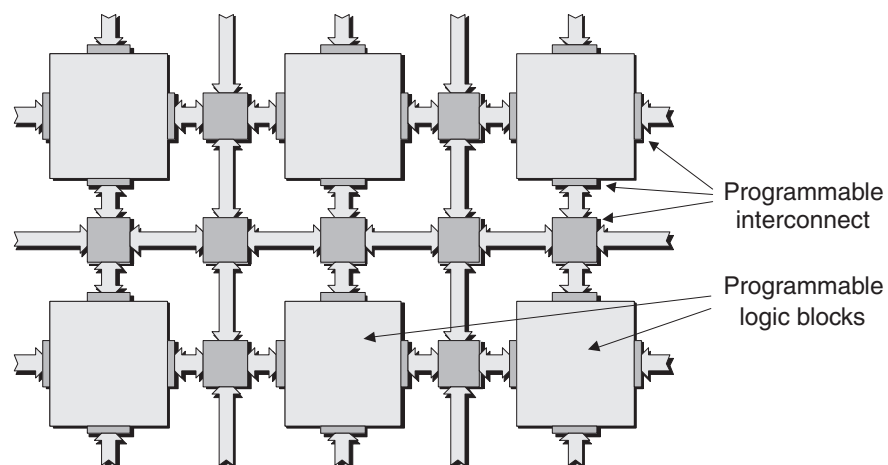


Figure 5-1. Top-down view of simple FPGA architecture.

Any facets of the device that may be programmed are done so by means of special configuration cells. The majority of FPGAs are based on the use of SRAM cells, but some employ FLASH (or E²) cells, while others use antifuses.

Irrespective of the underlying technology, the device's interconnect has a large number of associated cells that can be used to configure it so as to connect the device's primary inputs and outputs to the programmable logic blocks and

these logic blocks to each other. (In the case of the device's primary I/Os, which are not shown in Figure 5-1, each has a number of associated cells that can be used to configure them to accommodate specific I/O interface standards and so forth.)

For the purpose of this portion of our discussions, we shall assume that each programmable logic block comprises only a 4-input LUT, a multiplexer, and a register (Figure 5-2). The multiplexer requires an associated configuration cell to specify which input is to be selected. The register requires associated cells to specify whether it is to act as an edge-triggered flip-flop (as shown in Figure 5-2) or a level-sensitive latch, whether it is to be triggered by a positive- or negative-going clock edge (in the case of the flip-flop option) or an active-low or active-high enable (if the register is instructed to act as a latch), and whether it is to be initialized with a logic 0 or a logic 1. Meanwhile, the 4-input LUT is itself based on 16 configuration cells.

LUT is pronounced to rhyme with “nut.”

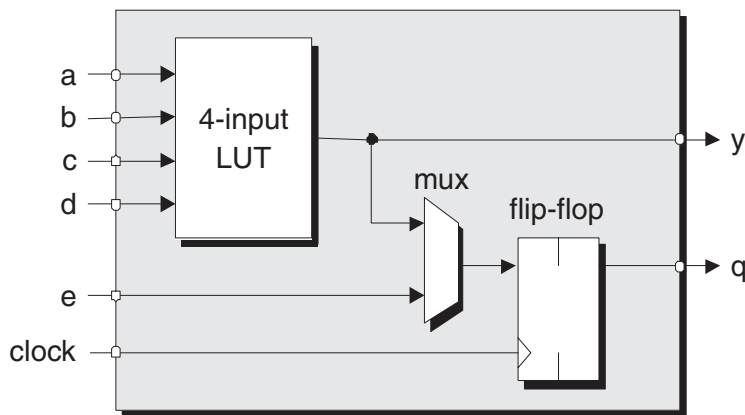


Figure 5-2. A very simple programmable logic block.

Antifuse-based FPGAs

In the case of antifuse-based FPGAs, the antifuse cells can be visualized as scattered across the face of the device at strategic locations. The device is placed in a special device programmer, the configuration (bit) file is uploaded into the device programmer from the host computer, and the device programmer uses this file to guide it in applying pulses of rela-

FLASH (and E²)–based devices are typically programmed in a similar manner to their SRAM cousins.

Unlike SRAM-based FPGAs, FLASH-based devices are nonvolatile. They retain their configuration when power is removed from the system, and they don't need to be reprogrammed when power is reapplied to the system (although they can be if required).

Also, FLASH-based devices can be programmed in-system (on the circuit board) or outside the system by means of a device programmer.

tively high voltage and current to selected pins to grow each antifuse in turn.

A very simplified way of thinking about this is that each antifuse has a “virtual” *x-y* location on the surface of the chip, where these *x-y* values are specified as integers. Based on this scenario, we can visualize using one group of I/O pins to represent the *x* value associated with a particular antifuse and another group of pins to represent the *y* value. (Things are more complicated in the real world, but this is a nice way to think about things that doesn't tax our brains too much.)

Once all of the fuses have been grown, the FPGA is removed from the device programmer and attached to a circuit board. Antifuse-based devices are, of course, *one-time programmable (OTP)* because once you've started the programming process, you're committed and it's too late to change your mind.

SRAM-based FPGAs

For the remainder of this chapter we shall consider only SRAM-based FPGAs. Remember that these devices are volatile, which means that they have to be programmed in-system (on the circuit board), and they always need to be reprogrammed when power is first applied to the system.

From the outside world, we can visualize all of the SRAM configuration cells as comprising a single (long) shift register. Consider a simple bird's-eye view of the surface of the chip showing only the I/O pins/pads and the SRAM configuration cells (Figure 5-3).

As a starting point, we shall assume that the beginning and end of this register chain are directly accessible from the outside world. However, it's important to note that this is only the case when using the *configuration port* programming mechanism in conjunction with the *serial load with FPGA as master* or *serial load with FPGA as slave* programming modes, as discussed below.

Also note that the *configuration data out* pin/signal shown in Figure 5-3 is only used if multiple FPGAs are to be config-

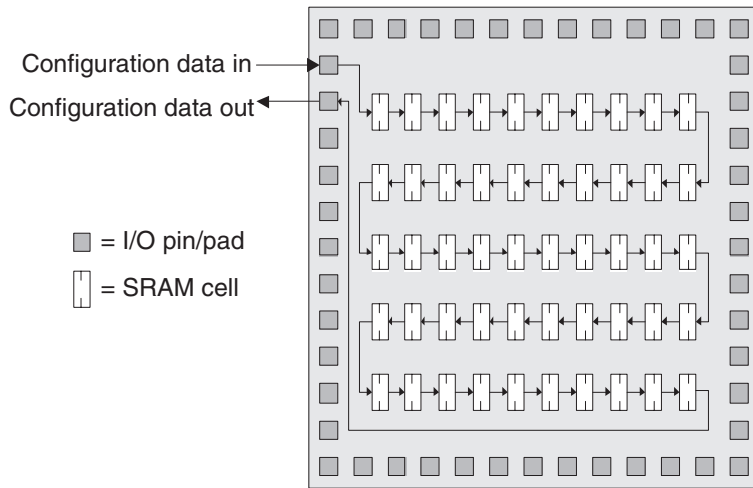


Figure 5-3. Visualizing the SRAM cells as a long shift register.

ured by cascading (daisy-chaining) them together or if it is required to be able to read the configuration data back out of the device for any reason.

The quickness of the hand deceives the eye

It isn't really necessary to know this bit, so if you're in a hurry, you can bounce over into the next section, but I found this interesting and thought you might find it to be so also. As figure 5-3 shows, the easiest way to visualize the internal organization of the SRAM programming cells is as a long shift register. If this were really the case, then each cell would be implemented as a flip-flop, and all of the flop-flops in the chain would be driven by a common clock.

The problem is that an FPGA can contain a humongous number of configuration cells. By 2003, for example, a reasonably high-end device could easily contain 25 million such cells! The core of a flip-flop requires eight transistors, while the core of a latch requires only four transistors. For this reason, the configuration cells in an SRAM-based FPGA are formed from latches. (In our example device with 25 million configuration cells, this results in a saving of 100 million transistors, which is nothing to sneeze at.)

Programming an FPGA can take a significant amount of time. Consider a reasonably high-end device containing 25 million SRAM-based configuration cells. Programming such a device using a serial mode and a 25 MHz clock would take one second. This isn't too bad when you are first powering up a system, but it means that you really don't want to keep on reconfiguring the FPGA when the system is in operation.

1843: England.
Sir Charles Wheatstone
and Sir William
Fothergill Cooke patent
the 2-needle electrical
telegraph.

The problem is that you can't create a shift register out of latches (well, actually you can, as is discussed a little later in this chapter, but not one that's 25 million cells long). The way the FPGA vendors get around this is to have a group of flip-flops—say 1,024—sharing a common clock and configured as a classic shift register. This group is called a *frame*.

The 25 million configuration cells/latches in our example device are also divided up into frames, each being the same length as the flip-flop frame. From the viewpoint of the outside world, you simply clock the 25 million bits of configuration data into the device. Inside the device, however, as soon as the first 1,024 bits have been serially loaded into the flop-flop frame, special-purpose internal circuitry automatically parallel copies/loads this data into the first latch frame. When the next 1,024 bits have been loaded into the flip-flop frame, they are automatically parallel copied/loaded into the second latch frame, and so on for the rest of the device. (The process is reversed when data is read out of the device.)

Programming embedded (block) RAMs, distributed RAMs, etc.

In the case of FPGAs containing large blocks of embedded (block) RAM, the cores of these blocks are implemented out of SRAM latches, and each of these latches is a configuration cell that forms a part of our “imaginary” register chain (as discussed in the previous section).

One interesting point is that each 4-input LUT (see Figure 5-2) can be configured to act as a LUT, as a small (16×1) chunk of distributed RAM, or as a 16-bit shift register. All of these manifestations employ the same group of 16 SRAM latches, where each of these latches is a configuration cell that forms a part of our imaginary register chain.

“But what about the 16-bit shift register incarnation,” you cry. “Doesn't this need to be implemented using real flip-flops?” Well, that's a good question—I'm glad you asked. In fact, a trick circuit is employed using the concept of a capaci-

tive latch that prevents classic race conditions (this is pretty much the same way designers built flip-flops out of discrete transistors, resistors, and capacitors in the early 1960s).

Multiple programming chains

Figure 5-3 shows the configuration cells presented as a single programming chain. As there can be tens of millions of configuration cells, this chain can be very long indeed. Some FPGAs are architected so that the configuration port actually drives a number of smaller chains. This allows individual portions of the device to be configured and facilitates a variety of concepts such as modular and incremental design (these concepts are discussed in greater detail in Section 2).

Quickly reinitializing the device

As was previously noted, the register in the programmable logic block has an associated configuration cell that specifies whether it is to be initialized with a logic 0 or a logic 1. Each FPGA family typically provides some mechanism such as an *initialization pin* that, when placed in its active state, causes all of these registers to be returned to their initialization values (this mechanism does not reinitialize any embedded [block] or distributed RAMs).

Using the configuration port

The early FPGAs made use of something called the *configuration port*. Even today, when more sophisticated techniques are available (like the JTAG interface discussed later in this chapter), the configuration port is still widely used because it's relatively simple and is well understood by stalwarts in the FPGA fraternity.

We start with a small group of dedicated *configuration mode* pins that are used to inform the device which configuration mode is going to be used. In the early days, only two pins were employed to provide four modes, as shown in Table 5-1.

Note that the names of the modes shown in this table—and also the relationship between the codes on the

1844: America. Morse Telegraph connects Washington and Baltimore.

1845: England. Michael Faraday discovers the rotation of polarised light by magnetism.

Mode Pins	Mode
0 0	Serial load with FPGA as master
0 1	Serial load with FPGA as slave
1 0	Parallel load with FPGA as master
1 1	Parallel load with FPGA as slave

Table 5-1. The four original configuration modes

mode pins and the modes themselves—are intended for use only as an example. The actual codes and mode names vary from vendor to vendor.

The mode pins are typically hardwired to the desired logic 0 and logic 1 values at the circuit board level. (These pins can be driven from some other logic that allows the programming mode to be modified, but this is rarely done in practice.)

In addition to the hard-wired mode pins, an additional pin is used to instruct the FPGA to actually commence the configuration, while yet another pin is used by the device to report back when it's finished (there are also ways to determine if an error occurred during the process). This means that in addition to configuring the FPGA when the system is first powered up, the device may also be reinitialized using the original configuration data, if such an occurrence is deemed necessary.

The configuration port also makes use of additional pins to control the loading of the data and to input the data itself. The number of these pins depends on the configuration mode selected, as discussed below. The important point here is that once the configuration has been completed, most of these pins can subsequently be used as general-purpose I/O pins (we will return to this point a little later).

Serial load with FPGA as master

This is perhaps the simplest programming mode. In the early days, it involved the use of an external PROM. This was subsequently superceded by an EPROM, then an E²PROM,

and now—most commonly—a FLASH-based device. This special-purpose memory component has a single data output pin that is connected to a *configuration data in* pin on the FPGA (Figure 5-4).

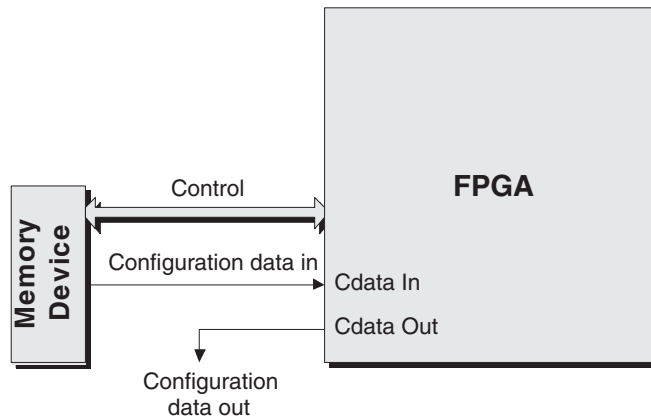


Figure 5-4. Serial load with FPGA as master.

The FPGA also uses several bits to control the external memory device, such as a reset signal to inform it when the FPGA is ready to start reading data and a clock signal to clock the data out.

The idea with this mode is that the FPGA doesn't need to supply the external memory device with a series of addresses. Instead, it simply pulses the reset signal to indicate that it wishes to start reading data from the beginning, and then it sends a series of clock pulses to clock the configuration data out of the memory device.

The *configuration data out* signal coming from the FPGA need only be connected if it is required to read the configuration data from the device for any reason. One such scenario occurs when there are multiple FPGAs on the circuit board. In this case, each could have its own dedicated external memory device and be configured in isolation, as shown in Figure 5-4. Alternatively, the FPGAs could be cascaded (daisy-chained) together and share a single external memory (Figure 5-5).

1845: England. First use of the electronic telegraph to help apprehend a criminal.

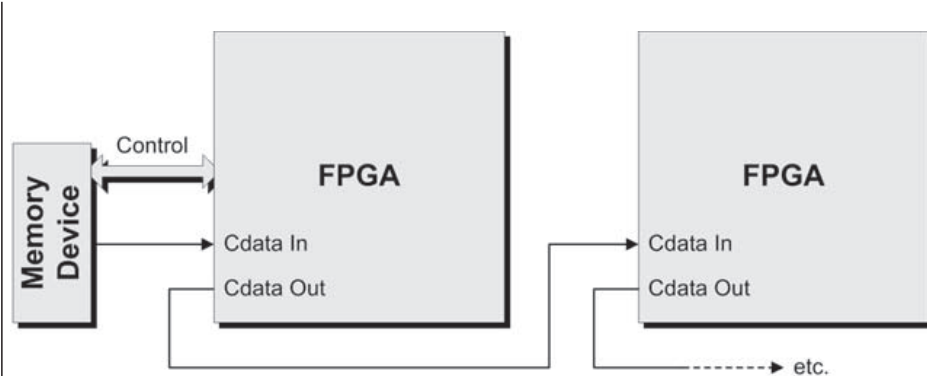


Figure 5-5. Daisy-chaining FPGAs.

Groups of four bits are also common and are given the special name of *nybble* (sometimes *nibble*). The idea is that “two nybbles make a byte,” which is a (little) joke. This goes to show that engineers do have a sense of humor; it’s just not very sophisticated.

When electronics and computing first started, defining things was something of a free-for-all. The end result was that different companies had their own definitions for things like bytes, and it was common to see 5-, 6-, 7-, 8-, and even 9-bit bytes. It was quite some time before the consensus settled on 8-bit bytes, at which time everyone was happy (apart from those who weren’t, but they don’t count).

In this scenario, the first FPGA in the chain (the one connected directly to the external memory) would be configured to use the *serial master* mode, while the others would be *serial slaves*, as discussed later in this chapter.

Parallel load with FPGA as master

In many respects, this is very similar to the previous mode, except that the data is read in 8-bit chunks from a memory device with eight output pins. Groups of eight bits are very common and are referred to as *bytes*. In addition to providing control signals, the original FPGAs also supplied the external memory device with an address that was used to point to whichever byte of configuration data was to be loaded next (Figure 5-6).

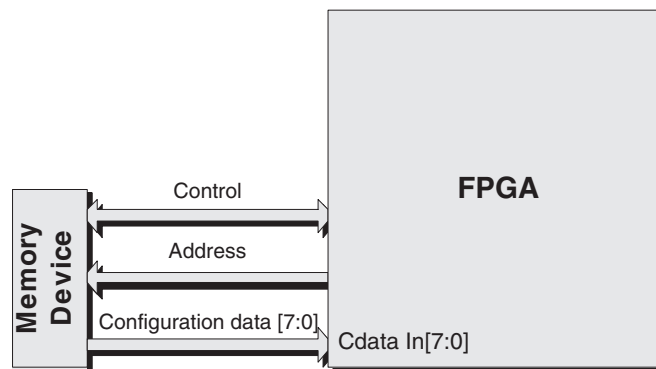


Figure 5-6. Parallel load with FPGA as master (original technique).

The way this worked was that the FPGA had an internal counter that was used to generate the address for the external memory. (The original FPGAs had 24-bit counters, which allowed them to address 16 million bytes of data.) At the beginning of the configuration sequence, this counter would be initialized with zero. After the byte of data pointed to by the counter had been read, the counter would be incremented to point to the next byte of data. This process would continue until all of the configuration data had been loaded.

It's easy to assume that this parallel-loading technique offers speed advantages, but it didn't for quite some time. This is because—in early devices—as soon as a byte of data had been read into the device, it was clocked into the internal configuration shift register in a serial manner. Happily, this situation has been rectified in more modern FPGA families. On the other hand, although the eight pins can be used as general-purpose I/O pins once the configuration data has been loaded, in reality this is less than ideal. This is because these pins still have the tracks connecting them to the external memory device, which can cause a variety of signal integrity problems.

The real reason why this technique was so popular in the days of yore is that the special-purpose memory devices used in the *serial load with FPGA as master* mode were quite expensive. By comparison, this parallel technique allowed design engineers to use off-the-shelf memory devices, which were much cheaper.

Having said this, special-purpose memory devices created for use with FPGAs are now relatively inexpensive (and being FLASH-based, they are also reusable). Thus, modern FPGAs now use a new variation on this parallel-loading technique. In this case, the external memory is a special-purpose device that doesn't require an external address, which means that the FPGAs no longer requires an internal counter for this purpose (Figure 5-7).

As for the serial mode discussed earlier, the FPGA simply pulses the external memory device's reset signal to indicate

1845: England/France.
First telegraph cable is
laid across the English
Channel.

1846: Germany.
Gustav Kirchhoff
defines Kirchhoff's laws
of electrical networks.

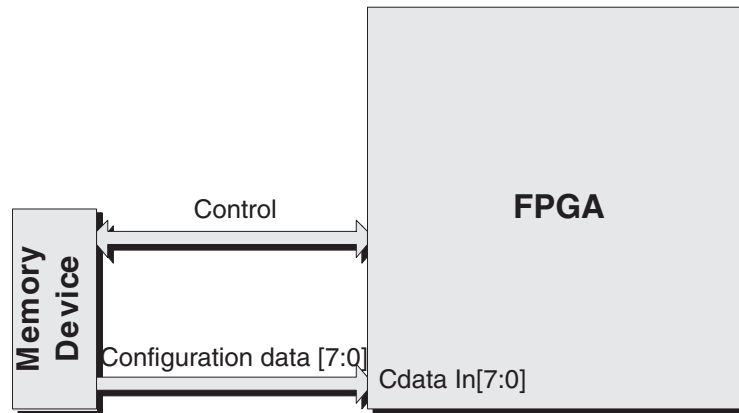


Figure 5-7. Parallel load with FPGA as the master (modern technique).

that it wishes to start reading data from the beginning, and then it sends a series of clock pulses to clock the configuration data out of the memory device.

Parallel load with FPGA as slave

The modes discussed above, in which the FPGA is the master, are attractive because of their inherent simplicity and also because they only require the FPGA itself, along with a single external memory device.

However, a large number of circuit boards also include a microprocessor, which is typically already used to perform a wide variety of housekeeping tasks. In this case, the design engineers might decide to use the microprocessor to load the FPGA (Figure 5-8).

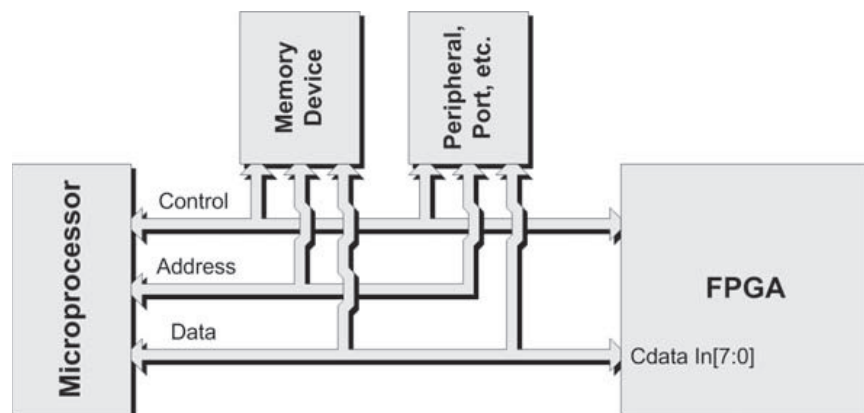


Figure 5-8. Parallel load with FPGA as slave.

The idea here is that the microprocessor is in control. The microprocessor informs the FPGA when it wishes to commence the configuration process. It then reads a byte of data from the appropriate memory device (or peripheral, or whatever), writes that data into the FPGA, reads the next byte of data from the memory device, writes that byte into the FPGA, and so on until the configuration is complete.

This scenario conveys a number of advantages, not the least being that the microprocessor might be used to query the environment in which its surrounding system resides and to then select the configuration data to be loaded into the FPGA accordingly.

Serial load with FPGA as slave

This mode is almost identical to its parallel counterpart, except that only a single bit is used to load data into the FPGA (the microprocessor still reads data out of the memory device one byte at a time, but it then converts this data into a series of bits to be written to the FPGA).

The main advantage of this approach is that it uses fewer I/O pins on the FPGA. This means that—following the configuration process—only a single I/O pin has the additional track required to connect it to the microprocessor's data bus.

Using the JTAG port

Like many other modern devices, today's FPGAs are equipped with a JTAG port. Standing for the *Joint Test Action Group* and officially known to engineers by its IEEE 1149.1 specification designator, JTAG was originally designed to implement the *boundary scan* technique for testing circuit boards and ICs.

A detailed description of JTAG and boundary scan is beyond the scope of this book. For our purposes here, it is sufficient to understand that the FPGA has a number of pins that are used as a JTAG port. One of these pins is used to input JTAG data, and another is used to output that data. Each of the FPGAs remaining I/O pins has an associated JTAG regis-

JTAG is pronounced by spelling out the "J," followed by "tag" to rhyme with "bag."

1847: England.
George Boole publishes
his first ideas on
symbolic logic.

ter (a flip-flop), where these registers are daisy-chained together (Figure 5-9).

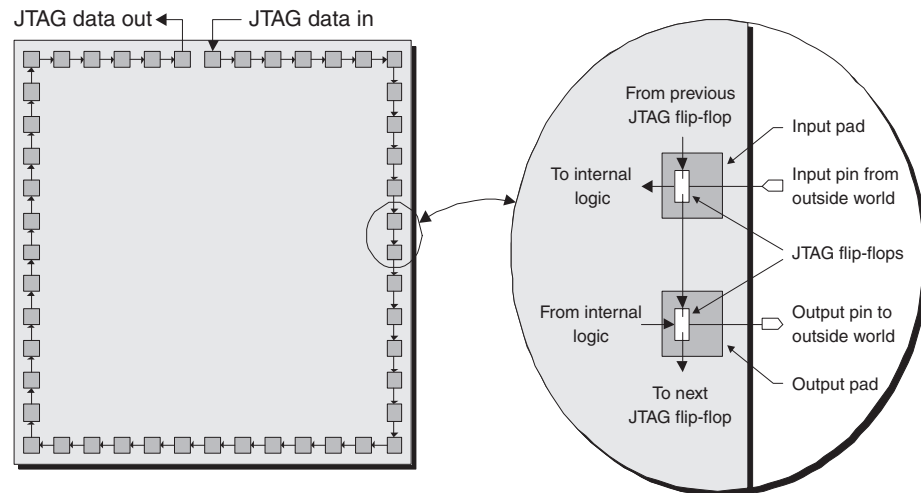


Figure 5-9. JTAG boundary scan registers.

The idea behind boundary scan is that, by means of the JTAG port, it's possible to serially clock data into the JTAG registers associated with the input pins, let the device (the FPGA in this case) operate on that data, store the results from this processing in the JTAG registers associated with the output pins, and, ultimately, to serially clock this result data back out of the JTAG port.

However, JTAG devices also contain a variety of additional JTAG-related control logic, and, with regard to FPGAs, JTAG can be used for much more than boundary scans. For example, it's possible to issue special commands that are loaded into a special JTAG command register (not shown in Figure 5-9) by means of the JTAG port's data-in pin. One such command instructs the FPGA to connect its internal SRAM configuration shift register to the JTAG scan chain. In this case, the JTAG port can be used to program the FPGA. Thus, today's FPGAs now support five different programming modes and, therefore, require the use of three mode pins, as shown in Table 5-2 (additional modes may be added in the future).

Mode Pins	Mode
0 0 0	Serial load with FPGA as master
0 0 1	Serial load with FPGA as slave
0 1 0	Parallel load with FPGA as master
0 1 1	Parallel load with FPGA as slave
1 x x	Use only the JTAG port

Table 5-2. Today's five configuration modes

Note that the JTAG port is always available, so the device can initially be configured via the traditional configuration port using one of the standard configuration modes, and it can subsequently be reconfigured using the JTAG port as required. Alternately, the device can be configured using only the JTAG port.

Using an embedded processor

But wait, there's more! In chapter 4, we discussed the fact that some FPGAs sport embedded processor cores, and each of these cores will have its own dedicated JTAG boundary scan chain. Consider an FPGA containing just one embedded processor (Figure 5-10).

The FPGA itself would only have a single external JTAG port. If required, a JTAG command can be loaded via this port that instructs the device to link the processor's local JTAG chain into the device's main JTAG chain. (Depending on the vendor, the two chains could be linked by default, in which case a complementary command could be used to disengage the internal chain.)

The idea here is that the JTAG port can be used to initialize the internal microprocessor core (and associated peripherals) to the extent that the main body of the configuration process can then be handed over to the core. In

1850: England.
Francis Galton invents
the Teletype printer.

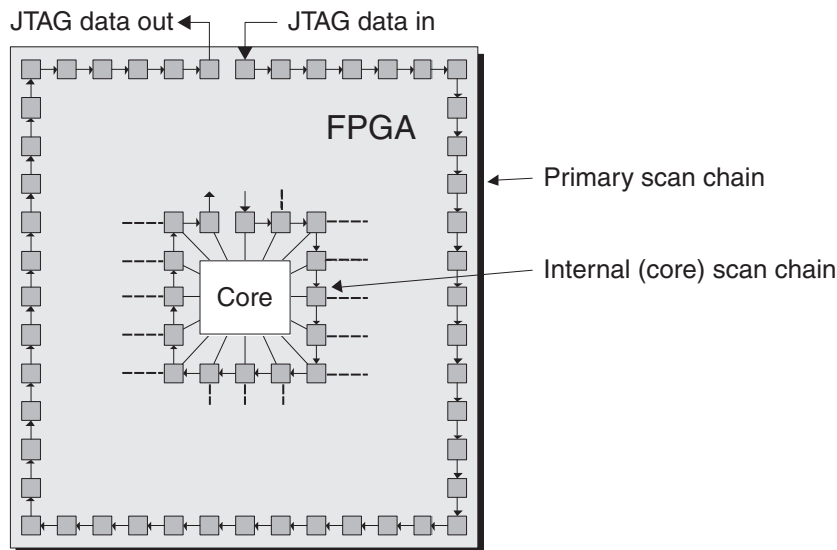


Figure 5-10. Embedded processor boundary scan chain.

some cases, the core might be used to query the environment in which the FPGA resides and to then select the configuration data to be loaded into the FPGA accordingly.